ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Pentek 6516-008* at our website: *Click HERE*

OPERATING MANUAL

MODEL 6516

16-Channel Digital Receiver with
TMS320C40-Compatible Comm Port I/O
for VMEbus Systems

Pentek, Inc.
One Park Way
Upper Saddle River, NJ 07458
(201) 818-5900
http://www.pentek.com/

Copyright © 1998 - 1999

Manual Part No: 800.65160 Rev.: B - July 28, 1999

Pentek Model 6516 Operating Manual - Revision History

Date	Rev	Applicable Serial #'s	Comments
10/19/98	Pre.	9841000 - 9930000	Initial product release.
4/20/99	A	9841000 - 9930000	Add information on Options 008, 009 & 018. Some spec changes in Sec. 1.6 - most notably Comm out rate decreased to 7 MB/sec. Add Figs. showing mezz. boards & front panels for these options. Refer to the Reset signal to the Harris chips as Sync, rathet than Reset. Sec. 2.4.1.1 & Fig. 2-8 - Comm Port cable is Model 2104, NOT 2014. Sec. 2.4.1.2 & Fig. 2-9: Recommended input cable is 3M 3756/80 (stranded wire, NOT 3749/80 (solid wire). Sec. 3.4 - mention 4290 as a VME master using 32-bit addressing. Sec. 3.4.1.1 - mention Opt. 018. Sec. 3.4.4 - State that Page Register was included to allow for future expansion. Expand Table 3-8 to include VME addr-esses of all Shadow Regs. Add new table (3-9) w/ 'C40 addr-esses of shadows. Secs. 3.5.2 & 3.5.4, Tables 3-10 & 3-12: Add info on PIO interrupt mask & status bits for Opt. 008. Table 3-10 & Sec. 3.5.2.3 - in Opt. 008, D14 & D10 Reserved, D12 for FPDP2 & D8 for FPDP1. Table 3-12 & Secs. 3.5.4.2 & 3.5.4.5 - in Opt. 008, D27, D25, D14 & D10 Reserved, D26 & D12 for FPDP2 & D24 & D8 for FPDP1 Sec. 3.5.7 - Numerous changes to text & tables. Sec. 3.5.8 & all its subsections - Add C & D bits. Table 3-19, Sec. 3.5.8.1 & Fig. 3-1: Fixed functions of D28 (Ch. D) & D20 (Ch. C). Combine the two sections about the two Receiver control registers into a single section (now 3.5.9). Add new sec. (3.5.11) dealing with Opt. 008's PIO register (upper word of Mux_Mode). Sec. 3.5.13 - Add info for Channel C & D Threshold Detectors. Sec. 3.5.14 - Give data formats for Opt. 018. Sec. 4.3.1 - Delete inapplicable paragraph about MIX bus access, replace with a paragraph stating the VME can write the Global Flash directly, and mentions the example listings in Appendix A.
5/3/99	A.1	9841000 - 9930000	More operating detail added for Option 009 - Sec. 3.5.11, Tables 3-24 & 3-25, Fig. 3-3.
6/8/99	A.2	9841000 - 9930000	Add cautionary note about FPDP cable construction to Sec . 2.4.3.2.
7/28/99	B	9930000 - Forward	Design changes to the Input Control Register for Options 008 & 018: Table 3-19 - D0 & D16 are now unused; D28 & D20 functions changed; Sec. 3.5.8.1 - changed description of Crossover bits & added new table (3-20) describing opeartion as a function of Option; Deleted Sec. 3.5.8.8.

Appendix B, Harris HSP50016 Data Sheet is republished with permission of Harris Corporation, 1025 W. NASA Boulevard, Melbourne, FL 32919. Copyright © 1996-98, Harris Corporation.

WARRANTY

Pentek warrants that all products manufactured by Pentek conform to published Pentek specifications and are free from defects in materials and workmanship for a period of one year from the date of delivery when used under normal operating conditions and within the service conditions for which they were furnished.

The obligation of Pentek arising from a warranty claim shall be limited to repairing or at its option, replacing without charge, any product which in Pentek's sole opinion proves to be defective within the scope of the warranty.

Pentek must be notified in writing of the defect or nonconformity within the warranty period and the affected product returned to Pentek within thirty days after discovery of such defect or nonconformity.

Buyer shall prepay shipping charges, taxes, duties and insurance for products returned to Pentek for warranty service. Pentek shall pay for the return of products to buyer except for products returned from another country.

Pentek shall have no responsibility for any defect or damage caused by improper installation, unauthorized modification, misuse, neglect, inadequate maintenance, accident or for any product which has been repaired or altered by anyone other than Pentek or its authorized representatives.

The warranty described above is buyer's sole and exclusive remedy and no other warranty, whether written or oral, is expressed or implied. Pentek specifically disclaims fitness for a particular purpose. Under no circumstances shall Pentek be liable for any direct, indirect, special, incidental or consequential damages, expenses, losses or delays (including loss of profits) based on contract, tort, or any other legal theory.

Copyright © 1998-9, Pentek, Inc. All Rights Reserved. Contents of this publication may not be reproduced in any form without written permission.
Printed in the United States of America.

Table of Contents

Page

Chapter 1: Overview

1.1 General Description 13
1.2 Operating Principle 13
1.3 Applications 13
1.4 Flexible Input Connections 13
1.5 Block Diagram 13
Figure 1-1: Model 6516 Block Diagram 14
1.6 Specifications 15

Chapter 2: Installation and Connections

2.1 Inspection 17
2.2 VMEbus A16 Base Address Switches 17
Figure 2-1: Rotary Switches 17
Figure 2-2: Model 6516 - Main Board (004.65160), Component Side 18
Figure 2-3: Model 6516 - Standard Mezzanine Board (004.65161), Component Side 19
Figure 2-4: FPDP Mezzanine Board (004.65162), Component Side 20
Figure 2-5: Option 009 Mezzanine Board (004.65163), Component Side 21
2.3 Jumper Settings 22
2.4 Model 6516 Front Panels 22
2.4.1 Standard Model 6516 Front Panel 22
Figure 2-6: Standard Model 6516 Front Panel 22
2.4.1.1 Comm Port Connectors 23
Figure 2-7: Model 6516 - Comm Port Pinouts 23
Figure 2-8: Comm Port Cable - Model 2104 23
2.4.1.2 Digitized Signal Input Connectors 24
Figure 2-9: Standard Model 6516 - Front Panel Digital Signal Input Connectors - Pin Assignments 25
Figure 2-10: Data Input Circuit 26
2.4.1.3 LED Indicators 26
2.4.2 Model 6516 Option 009 Front Panel 27
Figure 2-11: Model 6516 Option 009 Front Panel 27
2.4.2.1 Comm Port Connectors 27
2.4.2.2 Digitized Signal Input Connectors 27
2.4.2.3 Sync Connector 28
Table 2-1: Model 6516, Option 009 - Sync Connectror Pinout 28
2.4.2.4 LED Indicators 29

Table of Contents

Page

Chapter 2: Installation and Connections (continued)

2.4 Model 6516 Front Panels (continued)
2.4.3 The FPDP (Option 008) Front Panel 29
Figure 2-12: Model 6516 Option 008 Front Panel 29
2.4.3.1 Comm Port Connectors 29
2.4.3.2 Digitized Signal Input Connectors 29
Figure 2-13: Model 6516 Option 008 - Input Circuits 30
Figure 2-14: Model 6516 Option 008 - FPDP Input Connectors - Pin Assignments 31
2.4.3.3 Sync Connector 32
2.4.3.4 LED Indicators 33
2.4.4 Model 6516 Option 018 Front Panel 34
Figure 2-15: Model 6516 Option 018 Front Panel 34
2.4.4.1 Comm Port Connectors 34
2.4.4.2 Digitized Signal Input Connectors 35
2.4.4.3 Sync Connector 35
2.4.4.4 LED Indicators 35
2.5 Inserting and Removing the Model 6516 in the VMEbus Card Cage 36
2.5.1 Inserting the Unit 36
2.5.2 Removing the Unit 36

Chapter 3: Memory Maps and Register Descriptions

3.1 Programming Overview 37
3.2 VMEbus Address Modifier Codes 37
Table 3-1: VMEbus Address Modifiers 37
3.3 VMEbus Access 38
3.4 VMEbus A16 Slave Memory Map 38
Table 3-2: VMEbus A16 Slave Memory Map 38
3.4.1 VMEbus Control Register 39
Table 3-3: VMEbus Control Register 39
3.4.1.1 'C40 Boot Source Select 39
3.4.1.2 Test Bus Controller Enable 40
3.4.1.3 'C40 Reset 40
3.4.1.4 Board Reset 40
3.4.1.5 VMEbus IRQ Level Select 41
3.4.1.6 VMEbus IACK Vector Address 41

Table of Contents

Page

***Chapter 3: Memory Maps and Register Descriptions** (continued)*

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.2 VMEbus Address Register 41

Table 3-4: VMEbus Address Register 41

3.4.2.1 A32_Base Address 42

3.4.2.2 A24 Enable 42

3.4.2.3 A24_Base Address 42

3.4.2.4 Word Swap 42

3.4.3 VMEbus Interrupt to 'C40 43

3.4.4 Global Bus Address Page Register 43

Table 3-5: Global Bus Address Page Register 43

3.4.5 VMEbus Status Register 44

Table 3-6: VMEbus Status Register 44

3.4.5.1 Pending 'C40 Interrupt 44

3.4.5.2 TBC_Ready 44

3.4.6 Test Bus Controller Registers 44

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map 45

Table 3-7: Dual-Mapped Resources -'C40 Global and VMEbus A32/A24 Slave Address Spaces 45

3.5.1 Global SRAM 46

Table 3-8: VMEbus Addresses of HSP50016 Control Register Shadows in Global SRAM .46

Table 3-9: TMS320C40 Addresses of HSP50016 Control Register Shadows in Global SRAM .47

3.5.2 'C40 Interrupt Mask Register #1 47

Table 3-10: 'C40 Interrupt Mask Register #1 48

3.5.2.1 Threshold Detector Interrupts 48

3.5.2.2 PIO Interrupts (Option 008 Only) 49

3.5.2.3 Clock Not Ready Flag Interrupts 49

3.5.2.4 FIFO Full Flag Interrupts 49

3.5.3 'C40 Interrupt Mask Register #2 50

Table 3-11: 'C40 Interrupt Mask Register #2 50

3.5.3.1 FIFO Half Full Flag Interrupts 50

3.5.3.2 FIFO Not Empty Flag Interrupt 51

3.5.4 Receiver Status Register #1 51

Table 3-12: Receiver Status Register #1 52

3.5.4.1 'C40 Interrupt to VMEbus Status Bit 51

3.5.4.2 Clock LED Status Bits 52

3.5.4.3 Detector Flag Status Bits 53

3.5.4.4 PIO Interrupt Status Bit 53

3.5.4.5 Clock Not Ready Flag Status Bits 53

3.5.4.6 FIFO Full Flag Status Bits 54

Table of Contents

Page

Chapter 3: Memory Maps and Register Descriptions *(continued)*

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.5 Receiver Status Register #2 54
Table 3-13: Receiver Status Register #2 54
3.5.5.1 FIFO Half-Full Flag Status Bits 54
3.5.5.2 FIFO Empty Flag Status Bits 55
3.5.6 FIFO Reset Register 55
Table 3-14: FIFO Reset Register 55
3.5.6.1 FIFO Reset Bits 56
3.5.6.2 Comm Port Initialization Bit 56
3.5.7 Output Comm Port Control Register 57
Table 3-15 Output Comm Port Control Register 57
3.5.7.1 Pack Mode Bits 58
Table 3-16: Enabled Receiver Groups for Packed Mode 58
3.5.7.2 Data Ordering Bits 58
Table 3-17: Comm Port Data Packing and Ordering Formats (Tagging Disabled): Example - FIFO_Mode Dual 60
Table 3-18: Comm Port Data Packing and Ordering Formats (Tagging Disabled): Example - FIFO_Mode Single 61
3.5.7.3 FIFO Select Bits 62
3.5.7.4 Comm Port Enable Bits 62
3.5.8 Input Control Register 62
Table 3-19: Input Control Register 63
3.5.8.1 Input Crossover Bits 64
Table 3-20: Input Crossover Bits 64
3.5.8.2 User (OVLD) LED Bits 65
3.5.8.3 FIFO Mode Bits 65
Table 3-21: FIFO Mode Bits 65
3.5.8.4 Sync Output Enable Bits 66
3.5.8.5 Sync Input Source Bits 66
Figure 3-1: Synchronization Diagram 67
3.5.8.6 Software Sync Bits 68
3.5.8.7 FPDP Clock Select (Option 008 Only) 68
3.5.9 Receiver Control Registers 68
Table 3-22: Receiver Control Register #1 69
Table 3-23: Receiver Control Register #2 70
Figure 3-2: HSP50016 - Control Interface Timing Relationships 70
3.5.9.1 Receiver Setup Data Bits 70
3.5.9.2 Setup Data Clock Bits 71
3.5.9.3 Setup Data Strobe Bits 71
3.5.9.4 Receiver Chip Select Bits 71

Table of Contents

Page

***Chapter 3: Memory Maps and Register Descriptions** (continued)*

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.10 Multiplex and Mode Register ... 72

Table 3-24: Multiplex and Mode Register ... 72

3.5.10.1 Input Signal Source Selection Bits ... 72

3.5.10.2 Receiver Mode Select Bits ... 73

3.5.11 Option 009 Multiplex and Mode Register ... 73

Table 3-25: Model 6516 Option 009 - Multiplex and Mode Register ... 73

Figure 3-3: Model 6516 Option 009 - Signal Routing ... 74

Table 3-26: Model 6516, Option 009 - Input Data & Reset (Sync) Routing ... 75

3.5.12 PIO Register (Option 008 Only) ... 75

Table 3-27: PIO Register (Option 008 Only) ... 75

3.5.12.1 PIO Status Bits ... 76

3.5.12.2 PIO Catch Bits ... 77

3.5.12.3 PIO Hold Bits ... 77

3.5.12.4 PIO Clear Bits ... 77

3.5.13 Tag Enable Register ... 78

Table 3-28: Tag Enable Register ... 78

Table 3-29: Comm Port Data Packing and Ordering Formats (Tagging Enabled) (Examples - for FIFOs 1 & 2) ... 79

3.5.14 Threshold Detection Registers ... 80

Table 3-30: Threshold Detection Registers ... 80

3.5.14.1 Threshold Detector Enable Bit ... 80

3.5.14.2 Threshold Level Field ... 81

Table 3-31: Threshold Level Bits ... 81

3.5.14.3 Crossing Count Field - Bits D2 through D0 ... 81

Table 3-32: Threshold Crossing Count Value To Set Status Flag ... 81

3.5.15 Receiver Output Data FIFOs ... 82

Table 3-33: Direct FIFO Readout Data Formats - FIFO #1 Examples ... 83

Table 3-34: Model 6516 Option 018 - 32-bit Direct FIFO Readout Formats - FIFO #1 Examples ... 84

3.5.16 TMS320C40 IACK Vector Address ... 85

3.5.17 TMS320C40 Interrupt to VMEbus ... 85

3.5.18 Global Flash EEPROM ... 85

3.6 TMS320C40 Local Bus Resources ... 86

Table 3-35: TMS320C40 Local Bus Memory Map ... 86

3.6.1 Boot EEPROM ... 86

3.6.2 Local SRAM ... 86

Table of Contents

Page

Chapter 4: 65xx Series Firmware Command Set

4.1 Progamming Model 6516 Digital Receiver ... 87
4.2 Disabling the Model 6516 Firmware ... 88
4.2.1 Temporary Solution ... 88
4.2.2 Permanent Solution ... 88
4.3 Flash EEPROM Operations ... 89
4.3.1 Converting COFF Files into Intel HEX Format ... 89
Figure 4-1: Linker Command File for COFF-HEX File Conversion ... 90
4.3.2 Flash EEPROM Sectors ... 91
4.4 Firmware Command Format and Structure ... 91
Figure 4-2: 65XX Firmware Command Word Defined in 65xxcmd.h ... 91
Table 4-1: Flash EEPROM Sector Start Addresses ... 91
4.4.1 Cmd Field ... 92
4.4.2 Handle Field ... 92
4.4.3 Wait Flag ... 92
4.4.4 Data Fmt ... 92
4.4.4.1 P65XX_DEFAULT_TYPE ... 92
4.4.4.2 P65XX_IEEE_FLOAT_TYPE ... 92
4.4.4.3 P65XX_INTEGER_TYPE ... 92
4.4.5 Input Channel / Receiver Mask ... 93
4.4.6 Optional Parameters ... 93
4.5 65XX Firmware Commands ... 93
4.5.1 P65XX_OPEN_CONNECTION ... 94
4.5.2 P65XX_CLOSE_CONNECTION ... 94
4.5.3 P65XX_CLOSE _ALL_CONNECTIONS ... 95
4.5.4 P65XX_RESET_REGS ... 95
4.5.5 P65XX_INIT_RCVRS ... 96
4.5.6 P65XX_INIT_INPUT_CHANNELS ... 97
4.5.7 P65XX_SET_LO ... 98
4.5.8 P65XX_SET_DECIMATION ... 99
4.5.9 P65XX_SET_INP_CLK_RATE ... 100
4.5.10 P65XX_FLIP_SPECTRUM ... 101
4.5.11 P65XX_OFFSET_SPECTRUM ... 102
4.5.12 P65XX_SET_DATA_MODE ... 103
4.5.13 P65XX_SET_DATA_SRC ... 104
4.5.14 P65XX_SET_OUTPUT_SRC ... 105
4.5.15 P65XX_SET_PACKING_MODE ... 106
4.5.16 P65XX_SET_DATA_ORDER ... 107
4.5.17 P65XX_SELECT_FIFO ... 108
4.5.18 P65XX_TAG_ENABLE ... 109
4.5.19 P65XX_SET_THRESHOLD_DETECTION ... 110

Table of Contents

Page

Chapter 4: 65xx Series Firmware Command Set (continued)

4.5 65XX Firmware Commands (continued)
- 4.5.20 P65XX_RELEASE_FIFO 111
- 4.5.21 P65XX_RESET_FIFO 112
- 4.5.22 P65XX_FLUSH_FIFO 113
- 4.5.23 P65XX_GET_RCVR_STATUS 114
- 4.5.24 P65XX_GET_RCVR_DATA 115
- **Table 4-2: P65XX_GET_RCVR_DATA Comm Port Data Format 116**
- 4.5.25 P65XX_STOP_RCVR DATA 116
- 4.5.26 P65XX_DISABLE_FIRMWARE 117
- 4.5.27 P65XX_MODIFY_INTR_MASK 117
- 4.5.28 P65XX_LOOK 120
- 4.5.29 P65XX_SET 120
- 4.5.30 P65XX_GET_REVISION 121

4.6 VMEbus Specific Commands 122
- **Figure 4-3: Model 6516 - Single 'C40 Command Process Cycle 123**
- **Table 4-3: Commands & Addresses for the VMEbus 124**
- **Table 4-4: Sending Commands to the VMEbus 124**
- 4.6.1 P65XX_COMMAND_COMPLETE 125
- 4.6.2 P65XX_COMMAND_ERROR 125
- 4.6.3 P65XX_PARAM_ERROR 125
- 4.6.4 P65XX_OPEN_ERROR 125

4.7 Model 6516 Specific Firmware Commands 126
- 4.7.1 P6516_SET_RCVR_WORD 126
- 4.7.2 P6516_SET_INPUT_RESET_MODE 127
- 4.7.3 P6516_SET_FIFO_MODE 128
- 4.7.4 P6516_SET_INPUT_CROSSOVER 129

Appendix A: Programming Examples

A.1 Model 6516 Programming Examples A-1
- A.1.1 Programming the Model 6516 from a Model 4285 ..... A-1
- A.1.2 Programming the Model 6516 from a SBC Running VxWorks A-1
- A.1.3 Loading Flash EEPROM from Local 'C40 ..... A-2
- A.1.4 Loading Flash EEPROM from VME SBC Running VxWorks A-2

Table of Contents

Page

Appendix A: Programming Examples (continued)

Example Program ..... code page #
6516.h ..... code page 1
65xxcmd.h ..... code page 6
vme.h ..... code page 13
4285.h ..... code page 14
init6516.c ..... code page 16
main.c ..... code page 42
cmd85.c ..... code page 49
vme85.c ..... code page 52
cmdvxw.h ..... code page 57
main65XX.c ..... code page 58
cmdvxw.c ..... code page 66
am29f040.h ..... code page 72
am29f040.c ..... code page 73
loadhx65.c ..... code page 76
ldflsh65.c ..... code page 78

Appendix B: Harris HSP50016 Digital Down Converter

B.1 Introduction B-1
Digital Down Converter 3-198
Features 3-198
Applications 3-198
Ordering Information 3-198
Pinouts 3-199
48 Pin CPGA - Bottom view 3-199
48 Pin CPGA - Top view 3-199
44 lead PLCC - Top view 3-199
Pin Description 3-200
Figure 1 - Functional Block Diagram 3-201
Figure 2 - Phase Generator Block Diagram 3-202
Functional Description 3-202
Local Oscillator 3-203
Figure 3 - Phase Word Parameters for CW Case 3-203
Figure 4A - Phase Word During Up CHIRP 3-204
Figure 4B - Up CHIRP 3-204
Figure 5A - Phase Word During Down CHIRP 3-205
Figure 5B - Down CHIRP 3-205
Figure 6 - Up/Down CHIRP 3-205

Table of Contents

Page

Appendix B: Harris HSP50016 Digital Down Converter (continued)

Functional Description (continued)
- Mixer 3-206
- High Decimation Filter 3-206
  - **Figure 7A - Frequency Response of High Decimation Filter - from DC to First Null 3-206**
  - **Figure 7B - DDC HC Frequency Response 3-206**
- Scaling Multiplers 3-207
- FIR Filter 3-207
  - **Figure 8A - DDC, HDF, FIR, & Composite Filter Response 3-207**
  - **Figure 8B - DDC Filter Responses 3-207**
  - **Figure 9 - FIR Compensation for HDF Roll Off 3-208**
  - **Table 1 - FIR Output Rate and Decimation 3-208**
- Output Formatter 3-208
  - **Table 2 - Floating Point Format 3-209**
  - **Figure 10 - Data Output Modes 3-209**
  - **Figure 11 - Timing for CLK, IQCLK, IQSTB, I and Q 3-210**
  - **Figure 12 - IQSTB Timing 3-210**
  - **Figure 13 - Requested Data Output Timing 3-210**
  - **Figure 14 - Control Word Timing Diagram 3-210**
- Example Clock Calculations 3-211
- Control Word Input 3-211
  - **Table 3 - Control Word 0 ..... 3-212**
  - **Table 4 - Phase Generator/Test Enable/Output Register 3-212**
  - **Table 5 - Phase Generator Register 3-213**
  - **Table 6 - Phase Generator/Output Time Slot Register 3-213**
  - **Table 7 - Phase Generation/HDF OUTPUT Register 3-214**
  - **Table 8 - HDF/Output Register 3-214**
  - **Table 9 - Input and Output Format Register 3-216**
  - **Table 10 - Phase Offset Register 3-217**

Summary 3-218

Applications 3-218
- Down Conversion 3-218
  - **Figure 15 - Down Conversion 3-218**
  - **Figure 16 - Circuit for Single Channel Operation 3-219**
  - **Table 11 - Sample Format for Control Word 1 ..... 3-219**
  - **Table 12 - Sample Format for Control Word 2 ..... 3-219**
  - **Table 13 - Sample Format for Control Word 3 ..... 3-219**
  - **Table 14 - Sample Format for Control Word 4 ..... 3-219**
  - **Table 15 - Sample Format for Control Word 5 ..... 3-219**
  - **Table 16 - Sample Format for Control Word 6 ..... 3-220**
  - **Table 17 - Summary of Control Words for the Example 3-220**

Table of Contents

Page

Appendix B: Harris HSP50016 Digital Down Converter (continued)

Applications (continued)
Quadrature to Real Conversion ... 3-220
Figure 17 - Quadrature to Real Conversion of an Output Signal ... 3-221
Up Conversion by *f*"/2 ... 3-221
Figure 18 - Up Conversion by *f*"/2 ... 3-221
Quadrature Spectral Reversal ... 3-222
Figure 19 - Up Conversion of Filter Output Signal ... 3-222
Real Spectral Reversal ... 3-222
Figure 20 - Quadrature to Real Conversion of an Output Signal ... 3-222
High Decimation Filter Only ... 3-222
Multichannel Operation ... 3-222
Figure 21 - Circuit for Multiple Channel Operation ... 3-223
Figure 22 - Timing for Multiple Channel Operation ... 3-223
Figure 23 - Circuit for Multiple Channel Operation ... 3-224
References ... 3-224
Absolute Maximum Ratings ... 3-225
Operating Conditions ... 3-225
Thermal Information ... 3-225
Die Characteristics ... 3-225
DC Electrical Specifications ... 3-225
AC Electrical Specifications ... 3-225
AC Test Load Circuit ... 3-226
Waveforms ... 3-227
Figure 24 - Timing Relative to CLK ... 3-227
Figure 25 - Timing Relative to CCLK ... 3-227
Figure 26 - Timing Relative to TCK ... 3-228
Figure 27 - Output Rise and Fall Times ... 3-228

Chapter 1: Overview

1.1 General Description

The Model 6516 is a general purpose multichannel digital receiver VME board which accepts digitized data at sampling rates up to 50 MHz. It may be configured with up to 16 channels of narrowband receivers to perform frequency down conversion, lowpass filtering, and decimation of the sampled output.

The receiver output signals are delivered through front panel 'C40 Comm Ports for implementing DSP functions on Pentek's 'C40 DSP processor boards. Another set of front panel 'C40 Comm Ports can be used for receiver tuning control from the DSP. Receiver tuning, filter, and FIFO control is available from the VMEbus. The FIFO output data may also be read over the VMEbus.

1.2 Operating Principle

The Model 6516 uses the highly integrated HSP50016, from Harris Semiconductor, which contains a tunable local oscillator, a mixer and a tunable low−pass output filter. The local oscillator frequency and the output filter cutoff frequency for each receiver are independently programmable over either the VMEbus or the Comm Port interface, providing extremely flexible and agile operation.

1.3 Applications

This device provides an effective way to create multichannel Digital Signal Processing and analysis systems for selected frequency bands. Since the selected bands are down−converted, they make it economically feasible to perform DSP functions, such as FFT spectrum analysis, on RF signals. Typical applications include frequency−division multiplexed communications systems, modem modulation schemes, and analysis of many forms of radar signals.

1.4 Flexible Input Connections

All input data is accepted on multi−pin front panel connectors with a single−ended TTL input stage. An input multiplexer allows the same digital data stream to feed all 16 receivers, in order to simultaneously extract up to sixteen different frequency bands. Operating modes and receiver frequencies are programmable from either the VMEbus or the Comm Port interface.

1.5 Block Diagram

Figure 1−1, on the next page, shows the block diagram of Pentek's Model 6516 16−Channel Digital Receiver VMEbus Board with 'C40 Comm Port I/O.

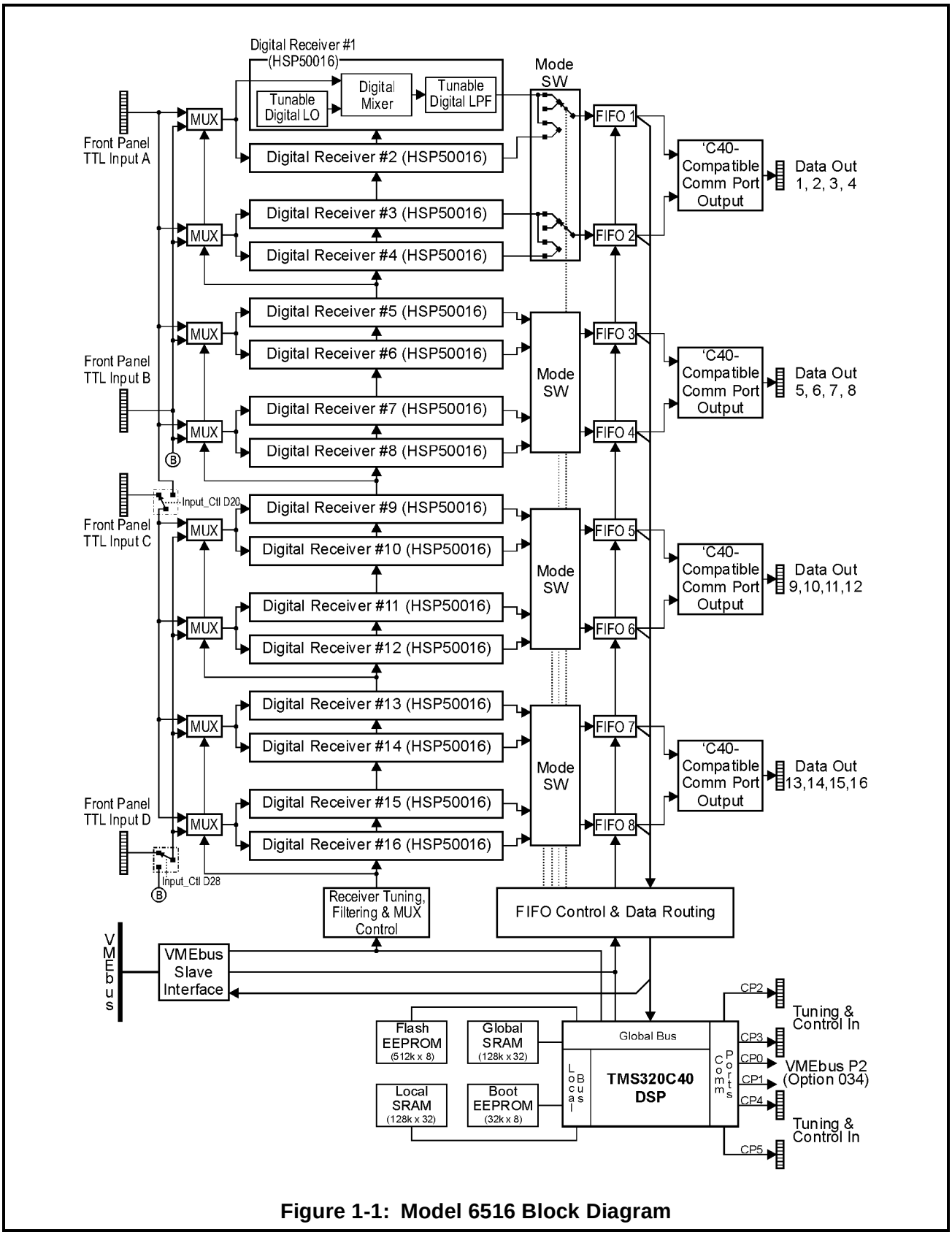


Figure 1-1: Model 6516 Block Diagram

1.6 Specifications

Receiver Chips: Harris HSP50016 – 16–bit input data.
See Appendix B for chip specifications

Standard: 16 (sixteen) – 16–bit output data

Option 018: 8 (eight) – 32–bit output data

Digital Inputs:

Standard: Single–ended TTL inputs compatible with the standard ver–sions of Models 6402, 6410, 6420 and 6441. Data rates up to 41 MHz can be provided by these units.

Option 008: FPDP Receiver Interface, compatible with Pentek Model 64xx Option 008 A/D converters, Model 4404 VXI Local Bus – FPDP adapter and Model 6226 FPDP I/O VIM Module.

Input Connectors:

Standard: Front panel 80–pin 3M connector (mates with 3M #82080–6006) for .025" pitch cable. Accepts up to 16 bits of data using 80–conductor cable & 80–pin connector.

Option 008: Front Panel 80–pin KEL connector (mates with KEL # 8825E–080–175) for for .025" pitch cable. Accepts up to 16 bits of data using 80–conductor cable & 80–pin connector.

Input Signals:

Data: 16 (sixteen) input lines/channel.

Clock: 1 (one) input line/channel. Clock rate up to 50 MHz max.

External Reset: 1 (one) input line/channel.

Data Setup Time: 2 nsec (min.) before rising edge of clock.

Data Hold Time: 8 nsec (min.) after rising edge of clock .

Input Multiplexers: Each receiver pair can independently select A/D Channel A or A/D Channel B. In addition, receivers 9 through 16 can select A/D Channel C or A/D Channel D.

Output Comm Ports: Four, one for each quad of receivers; byte serial with 4 bytes per 32–bit longword; data may be packed (two 16–bit data words per longword) or unpacked (one 16–bit data word per longword).

Buffers: Active.

Functions: Each port delivers data from two or four receivers.

Maximum Data Rate: 7 MBytes/sec.

FIFO Memory:

Standard: 4k x 16 FIFO for each receiver channel pair.

Option 021: 16k x 16 FIFO for each receiver channel pair.

1.6 Specifications (continued)

VMEbus Interface: Slave; A32/D32; I1−7; Powers up with no A24/A32 address mapping; Bus Grant daisy chain bussed through; Interrupt daisy chain supported.

- **A16 Memory Map:** A16 Base Address set with Rotary Switch; 256 byte address space max; A24/A32 base address configuration registers; Interrupt Vector Register; 'C40 Test Bus Controller for 'C40 J−TAG; Status/Control Registers ('C40 Reset, 'C40 Interrupt, IRQ level select, A24 enable, word swap, etc.)
- **A24/32 Memory:** Mapping disabled on power up, configured with A16 registers; 1 MB address space in A24; 4 MB address space in A32; All Data FIFO's (no data packing in VME memory − 16 bits only); Receiver, Multiplexer & Reset Control functions; Interrupt Mask Registers; Threshold Detectors; Clock Timeout; Threshold Select Registers − one of four thresholds per A/D Input; 'C40 Status/ Control Registers; 128k x 32 'C40 Shared Global SRAM; 256k x 8 Flash memory (A32 only).
- **VMEbus Data Rate:** 200 nsec DS to DTACK delay max (with no contention for global bus by 'C40); 280 nsec bus cycle time max (with 80 nsec master response overhead).

Processor: One Texas Instruments TMS320C40 − 50 MHz; On−board XDS−510 14−pin J−TAG emulator connector; Texas Instruments Test Bus Controller mapped into A16 VMEbus space; four active buffered Comm Ports to front panel; Comm Ports may optionally be made available over VMEbus P2 connector (Option 034).

- **'C40 Global Bus:** 128k x 32 Global SRAM, 1 wait state, Shared with VMEbus; 256k x 8 Flash User EEPROM; Receiver, Multiplexer & Reset Control registers; all data FIFOs.
- **'C40 Local Bus:** 128k x 32 Local SRAM − 0 wait state; 32k x 8 Boot PROM (factory code).

Power: 3.0 A @ + $5V_{DC}$ max.

Dimensions: 6U (160 mm x 233.35 mm), Single (0.8 in.) slot width.

Chapter 2: Installation and Connections

2.1 Inspection

After unpacking the unit, inspect it carefully for possible damage to connectors or components. If any damage is discovered, please contact Pentek immediately, at the phone number shown on this manual's title page. Please save the original shipping container and packing material, in case re−shipment is required.

The Model 6516 contains two printed circuit (PC) assemblies. The board with the VMEbus P1 and P2 connectors (Pentek part # 004.65160) contains the VME interface, the 'C40 DSP its associated memory resources, the Comm Ports and the FIFOs. The mezzanine board houses the input connectors and the Digital Receiver circuitry. Three mezzanine boards currently exist, the standard mezzanine (Pentek part # 004.65161), the FPDP mezzanine (004.65162 − used for Options 008 and 018), and the Option 009 Mezzanine (004.65163). Drawings of the component sides of all four of these PC assem−blies are shown in Figures 2−2 through 2−5, on the next four pages.

2.2 VMEbus A16 Base Address Switches

Three separate address spaces are defined for the VMEbus environment. These three regions are known as A16, A24 and A32. Each device in your VMEbus card cage must have a unique address in any address space in which it must be accessible. On the Model 6516 the A16_base address (i. e., the lowest address in the A16 region occupied by this device) is set by means of two rotary switches. These switches, SW1 and SW2, are located at the top rear of the VME board, with SW1 adjacent to the VME P1 connec−tor, and SW2 directly to the left of SW1. Figure 2−1, below, shows what these switches look like and Figure 2−2, on the next page, shows the location of this pair of switches.

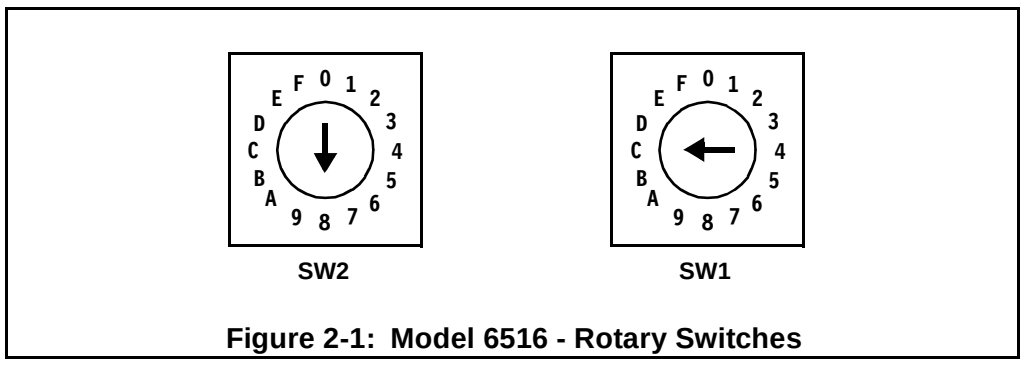


Figure 2-1: Model 6516 - Rotary Switches

The settings of these 16−position rotary switches each define four bits of the 6516's A16_base address. Each of the 16 positions on the switches is denoted with a hex digit (0 − F), whose four−bit binary value will be used as part of the device's base address. SW2 defines the four most significant bits (MSBs) of the address (A15 − A12) and SW1 defines the next four address bits (A11 − A8). Thus, with 16 settings available on each of two switches, one of 256 different base addresses may be selected for any 6516. Figure 2−1, above, shows the switches set for an A16_base address of **`0x8C00`**.

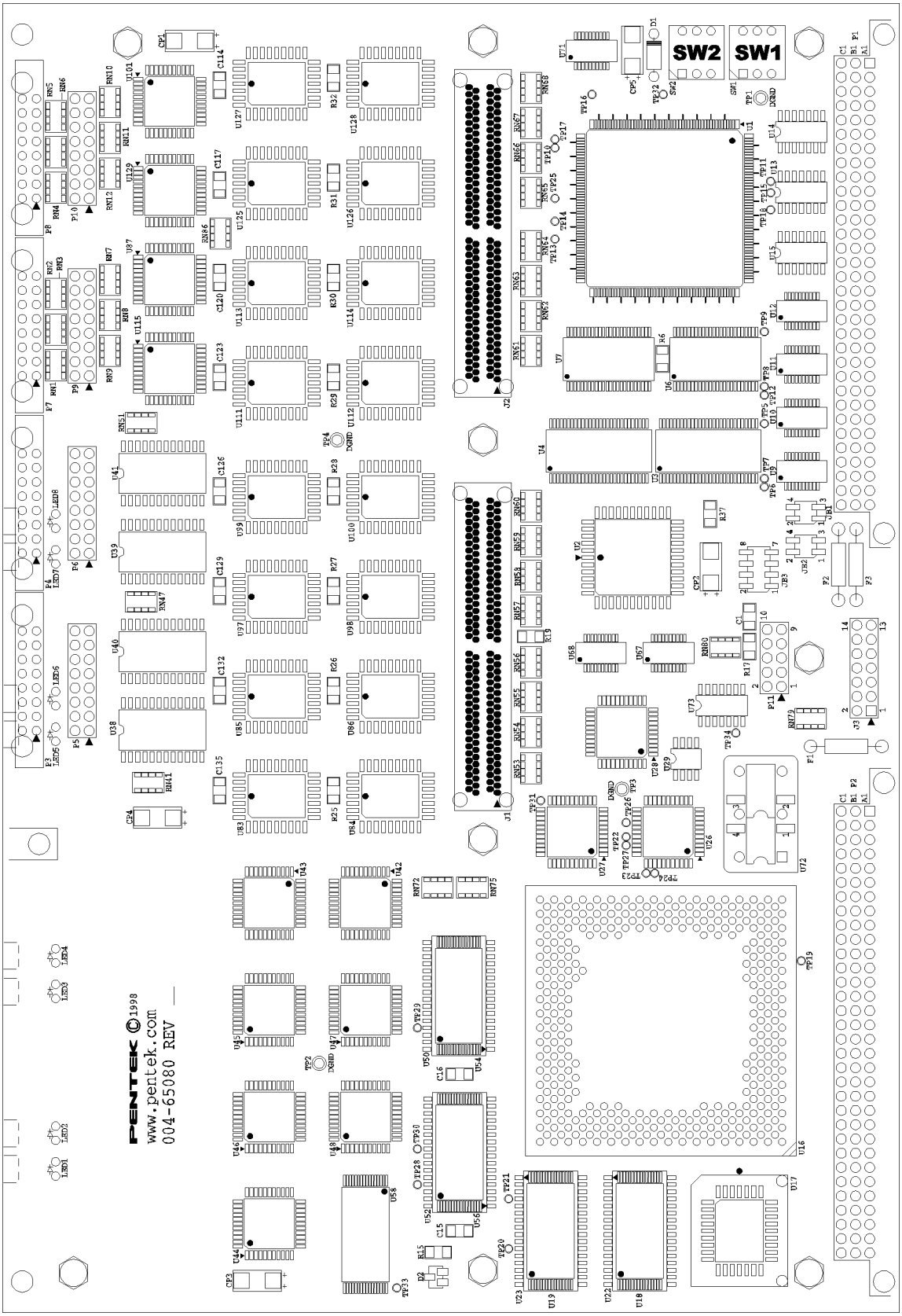


Figure 2-2: Model 6516 - Main Board (004.65160), Component Side

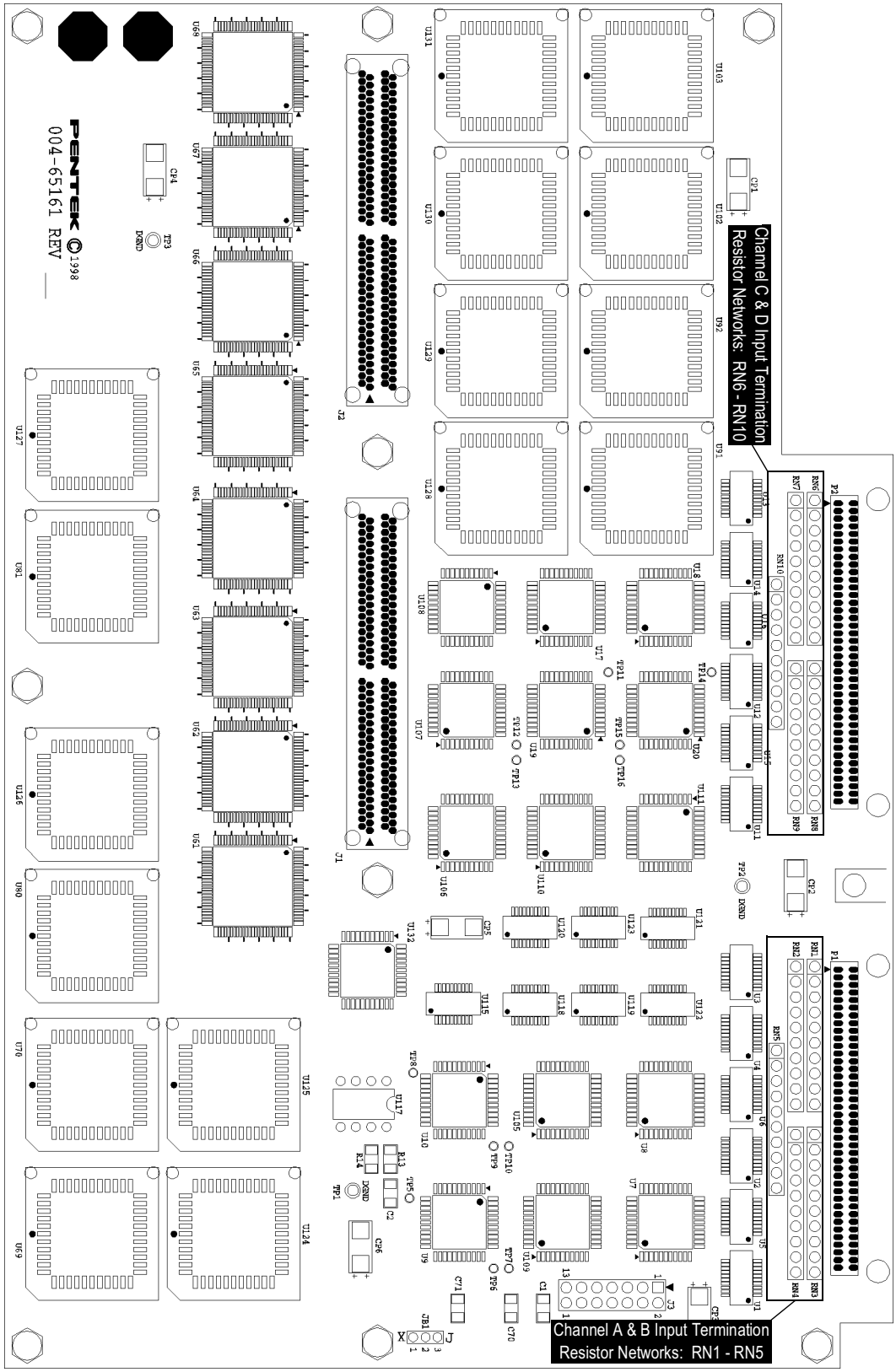


Figure 2-3: Model 6516 - Standard Mezzanine Board (004.65161), Component Side

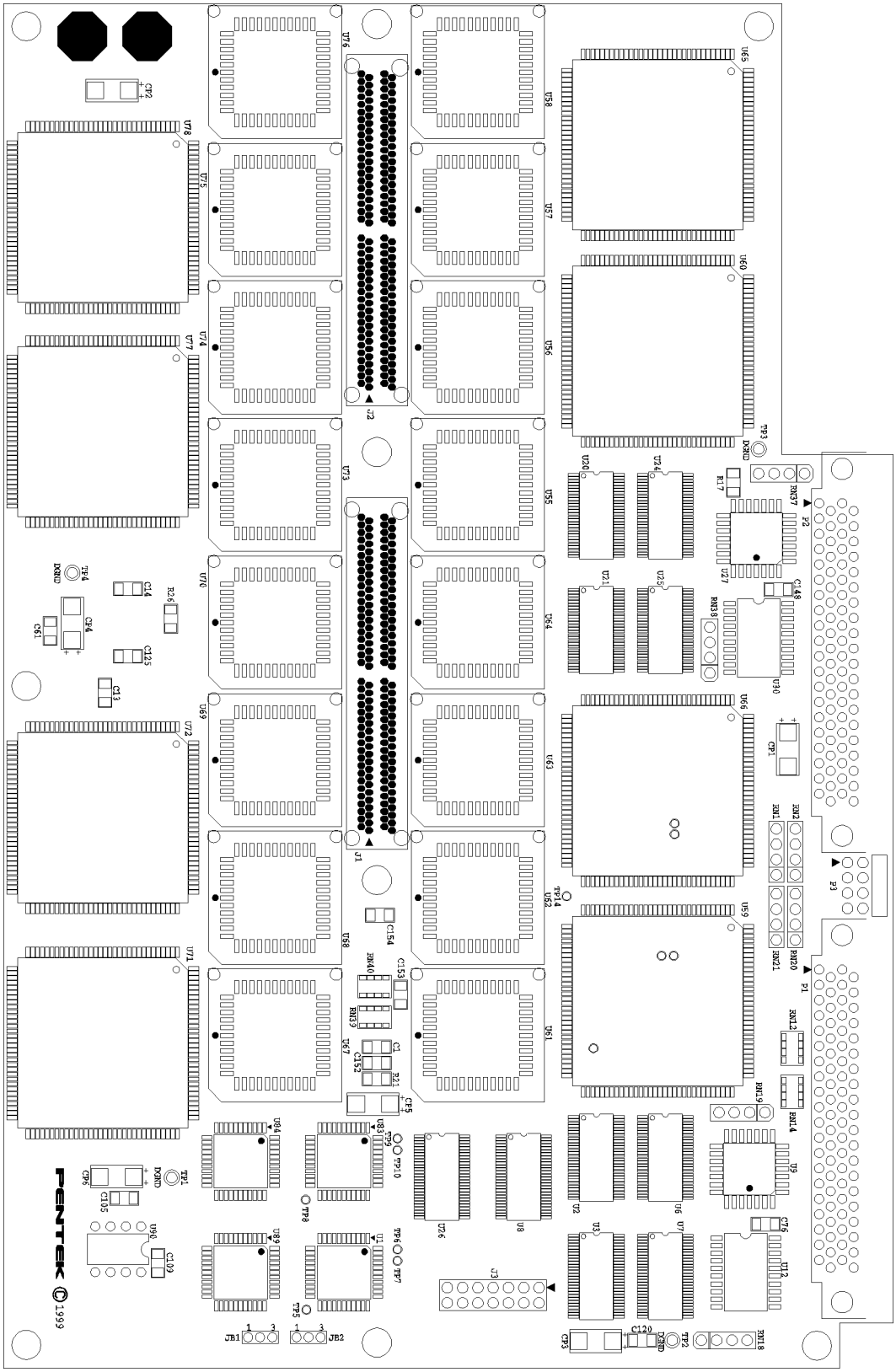


Figure 2-4: FPDP Mezzanine Board (004.65162), Component Side

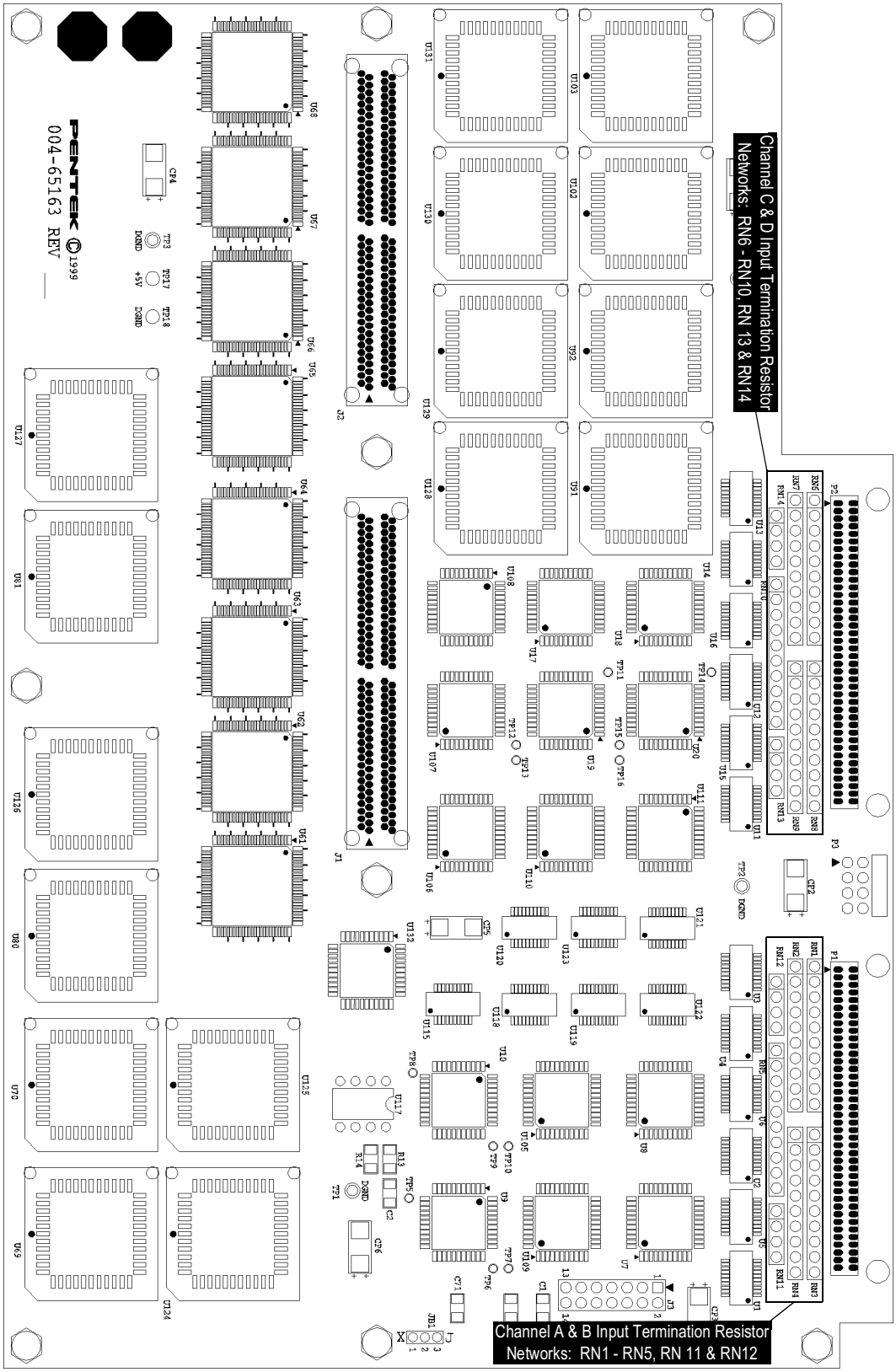


Figure 2-5: Option 009 Mezzanine Board (004.65163), Component Side

2.2 VMEbus A16 Base Address Switches (continued)

Throughout this manual, all VMEbus addresses will be given as offsets from the assigned base address. For example, the A24 and A32 base addresses for the 6516 are defined by writing to a register at address A16_base+0x04.

2.3 Jumper Settings

Although there are a few jumper blocks on the PC boards for the Model 6516, all of them are set at the factory during assembly, test and calibration.

None of the jumper settings on this device should be changed by the user!!

2.4 Model 6516 Front Panels

There are four different Front Panels for the Model 6516: the standard panel (Pentek part # 300.65162), the Option 009 panel (part # 300.65166), the FPDP panel (for Option 008 – part # 300.65164), and the Option 018 panel (part # 300.65168). The standard panel's features will be discussed first, and subsequent sections will describe the differences between the optional panels and the standard panel.

2.4.1 Standard Model 6516 Front Panel

The front panel of Pentek's Standard Model 6516 Octal Digital Receiver VMEbus Board contains a 80–pin flat cable connector, which can carry two 16–bit input data streams. The panel also has four Comm Port connectors for the receiver data outputs, two red Overload indicator LEDs, and two green Clock Ready indicator LEDs. The subsections below discuss the connectors on the front panel, the kinds of connectors can be used to mate with them, and the types of signals that should be applied to them or are available on them. The LEDs are also discussed as well as the conditions they are used to indicate. Figure 2–6, at the right, depicts the front panel of the Standard Model 6516.

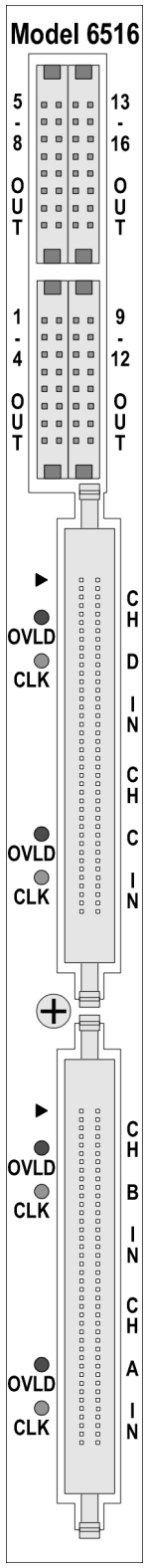


Figure 2-6: Standard Model 6516 Front Panel

2.4 Model 6516 Front Panels (continued)

2.4.1 Standard Model 6516 Front Panel (continued)

2.4.1.1 Comm Port Connectors

The four connectors at the top of the Model 6516's front panel are 16–pin, dual in–line ribbon cable connectors for the TMS320C40–compatible Comm Port outputs. These ports provide an alternative to the VMEbus for reading data from the Model 6516, in applications where a 'C40 processor will handle the data. **These ports are NOT driven by the on–board 'C40 processor, but are PAL–controlled, and can be connected to 'C40 input or output Comm Port. These ports have built–in token forcing, but cannot be configured as input ports on the Model 6516.**

Each connector carries data from two or four receivers. The connector on the top left carries data from receivers 5, 6, 7 and 8. The connector below it (left, above center) carries data from receivers 1, 2, 3, and 4. The connectors on the right side of the panel carry data from receivers 13, 14, 15, and 16 (top right), and receivers 9, 10, 11, and 12 (right, above center). For details about programming the 6516 to use these ports, please refer to Section 3.5.7 of this manual. Figure 2–7, below left, shows the Comm Port pin assignments. The connectors are keyed to prevent improper insertion.

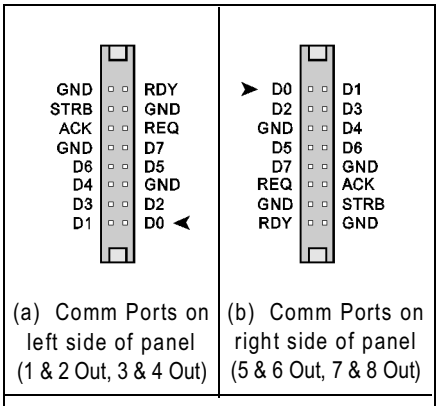


Figure 2-7: Model 6516 - Comm Port Pinouts

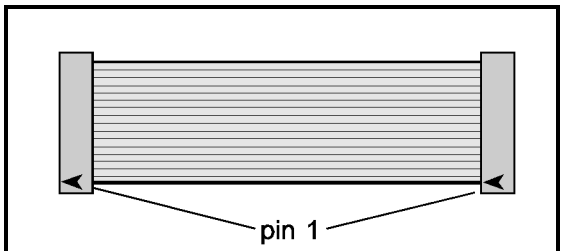


Figure 2-8: Comm Port Cable - Model 2104

Comm Port cables can be easily assembled by using a 3M hand press with locator plate #3443–110 and standard 16–conductor, 0.050 pitch flat ribbon cable such as 3M #3365/16. Every Model 6516 is shipped with four mating connectors (3M #50116–B000) and the optional strain reliefs (3M #3448–50116). The use of these standard components allows the user to easily configure the Comm Port cables to minimize excess cable length for any given interconnect pattern.

2.4 Model 6516 Front Panels (continued)

2.4.1 Standard Model 6516 Front Panel (continued)

2.4.1.1 Comm Port Connectors (continued)

As an aid to consistent assembly techniques, make sure that pin 1 of each socket at the end of the cable is connected to pin 1 of the ribbon cable. Pin 1 of the cable is identified by the colored stripe. A typical Comm Port cable is shown in Figure 2–8, on the lower right of the previous page. Comm Port cables are available preas–sembled from Pentek as Model 2104 (to mate with the connectors on the Models 4284, 4270, 4269, 4257 and 4254) and Model 2015 (to mate with the connectors on the Model 4285). Cable lengths of up to 18–in. can be accommodated.

Each of the twelve signal lines on each Comm Port is back–termi–nated with a series 24Ω resistor to properly match the characteristic impedance of the flat ribbon wiring used in the Comm Port cables.

2.4.1.2 Digitized Signal Input Connectors

The standard Model 6516 accepts single–ended TTL level signals, and is compatible with the standard versions of Pentek A/D Converter Models 6402, 6410, 6420 and 6441. The input circuitry is contained on a daughter card inside the module, called the Mezzanine board.

The input connector is an 80–pin flat ribbon cable header manufac–tured by 3M. The part number for the header is 3M #81080–620203.

The flat cable socket connector is 3M #82080–6006, and the strain relief is 3M #3858–080. Pentek's Part number for the connector, with the strain relief, is 353.08005. Two of each of these items are supplied with the Standard Model 6516. The recommended .025" pitch flat ribbon cable is 3M #3756/80.

The 80–pin connector is composed of two identical 40–pin groups, one for each input channel. The 80–conductor ribbon cable can be split in half, forming two identical 40–pin cables. The pin assign–ment for the connector is shown on the next page in Figure 2–9.

For each of the input channels, the input signal consists of 16 sin–gle–ended data lines (D0 – D15), a data strobe (CK), and a Sync (or reset) control (Sync). Note that each input line has a ground line opposite it in the charts in Figure 2–9. This creates the optimal signal line sequence in the flat ribbon cable, very desirable in con–trolling crosstalk and impedance. It is recommended that the driving end of the cable have all of the GND lines connected to signal ground to help maintain signal integrity.

Input Connector Pin Assignments for Lower (& Upper) Connector			
Pin #	**Signal**	**Pin #**	**Signal**
1	Ch B (D) - D1 in	**2**	GND
3	Ch B (D) - D2 in	**4**	GND
5	Ch B (D) - D3 in	**6**	GND
7	Ch B (D) - D4 in	**8**	GND
9	Ch B (D) - D5 in	**10**	GND
11	Ch B (D) - D6 in	**12**	GND
13	Ch B (D) - D7 in	**14**	GND
15	Ch B (D) - D8 in	**16**	GND
17	Ch B (D) - D9 in	**18**	GND
19	Ch B (D) - D10 in	**20**	GND
21	Ch B (D) - D11 in	**22**	GND
23	Ch B (D) - D12 in	**24**	GND
25	Ch B (D) - D13 in	**26**	GND
27	Ch B (D) - D14 in	**28**	GND
29	Ch B (D) - D15 in	**30**	GND
31	Ch B (D) - Reset	**32**	GND
33	Reserved	**34**	GND
35	Ch B (D) - CK	**36**	GND
37	Ch B (D) - D0 in	**38**	GND
39	GND	**40**	GND
41	Ch A (C) - D4 in	**42**	GND
43	Ch A (C) - D5 in	**44**	GND
45	Ch A (C) - D6 in	**46**	GND
47	Ch A (C) - D7 in	**48**	GND
49	Ch A (C) - D8 in	**50**	GND
51	Ch A (C) - D9 in	**52**	GND
53	Ch A (C) - D10 in	**54**	GND
55	Ch A (C) - D11 in	**56**	GND
57	Ch A (C) - D12 in	**58**	GND
59	Ch A (C) - D13 in	**60**	GND
61	Ch A (C) - D14 in	**62**	GND
63	Ch A (C) - D15 in	**64**	GND
65	Ch A (C) - Reset	**66**	GND
67	Reserved	**68**	GND
69	Ch A (C) - CK	**70**	GND
71	Ch A (C) - D0 in	**72**	GND
73	GND	**74**	GND
75	Ch A (C) - D1 in	**76**	GND
77	Ch A (C) - D2 in	**78**	GND
79	Ch A (C) - D3 in	**80**	GND

Note: In each input channel, Bit D0 is the LSB.

The mating cable socket connector is 3M #82080-6006,
and the strain relief is 3M #3858-080.
Two each of these items is supplied with the Model 6516.
The recommended .025" pitch flat ribbon cable is
3M # 3756/080.

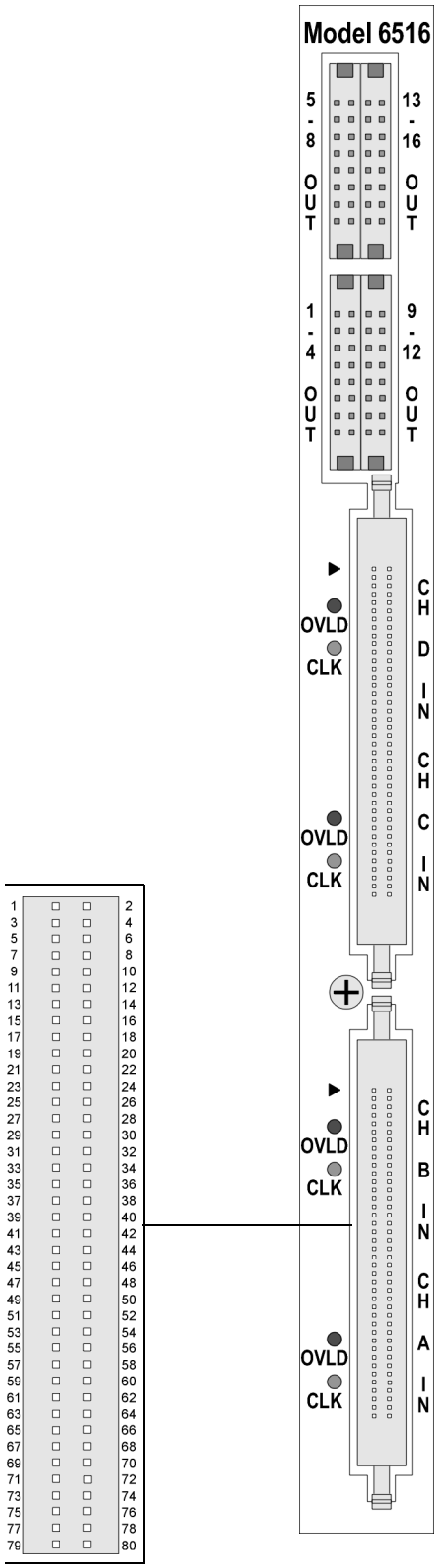


Figure 2-9: Standard Model 6516 - Front Panel Digital Signal Input Connectors - Pin Assignments

2.4 Model 6516 Front Panels (continued)

2.4.1 Standard Model 6516 Front Panel (continued)

2.4.1.2 Digitized Signal Input Connectors (continued)

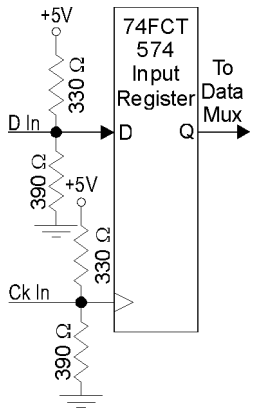


Figure 2-10: Model 6516 - Data Input Circuit

All input lines are normally terminated with 330Ω pullup resistors to +5V and 390Ω pulldown resistors to GND. If necessary, different termination networks may be specified when the unit is ordered. Additionally, the resistor networks used at these locations (RN1 – RN10, see Figure 2–3 on page 19) are socketed, to allow them to be changed easily in the field. The proper termination will depend upon the type of line driver used in your application, which in turn depends on the length of your cables, the clock frequency and other subtleties. The input circuitry for the data lines and the clock is shown schematically in Figure 2–10, at the left. The data register is a type 74FCT574, an edge–triggered D–type flip–flop. FCT is a high–performance, TTL compatible CMOS logic family manufactured by Integrated Device Technology (IDT), among others. The other input lines (CK and Sync) are received by a type 74FCT241, an octal buffer in the same FCT family. These lines are terminated by the same type of termination resistor arrangement as the data lines.

2.4.1.3 LED Indicators

Two LED indicators are provided for each input channel. These LEDs are discussed below.

- **Clock (CLK) LED (Green) –** The green LED for a given input channel will be lit when a clock signal for that channel is detected. If this LED is not lit, then no clock signal has been detected and no data from this stream can be processed.

- **Overload (OVLD) LED (Red) –** The red LED for a given input channel was initially intended as an indicator for the threshold detection function (hence the name – this, in fact, is what the firmware uses these LEDs for), and may still be used as such if desired. The states of these LEDs are controlled by two user–addressable bits in the Input Control Registers. See Section 3.5.8.2 for further details.

2.4 Model 6516 Front Panels (continued)

2.4.2 Model 6516 Option 009 Front Panel

The front panel for Pentek's Model 6516 Option 009 is essentially identical to the one used on the standard model, except for the addition of an 8–pin connector for sync signals, located between the two input data connectors. The items on this panel will be discussed in the subsections below.

2.4.2.1 Comm Port Connectors

The Comm Port connectors on the 6516 Option 009 are identical to those on the standard model. For complete information on these connectors, please refer to Section 2.4.1.1, starting on page 23 of this manual.

2.4.2.2 Digitized Signal Input Connectors

The two input data connectors on the Model 6516 Option 009 are very nearly identical to the ones used on the standard model. The only difference is that, on both connectors, the pins that carry the Sync (a.k.a. Reset) signals for the standard unit are unconnected here. In other words, pins 31 and 65 on both connectors are unused.

Aside from the above exception, all the information that appears in Section 2.4.1.2, beginning on page 24 of this manual, and in Figures 2–9 and 2–10, is also applicable to the Option 009 front panel.

CAUTION!!

When inserting and removing the Input Data connectors, be careful that the locking tabs do not bend any pins on the SYNC connector, located between them.

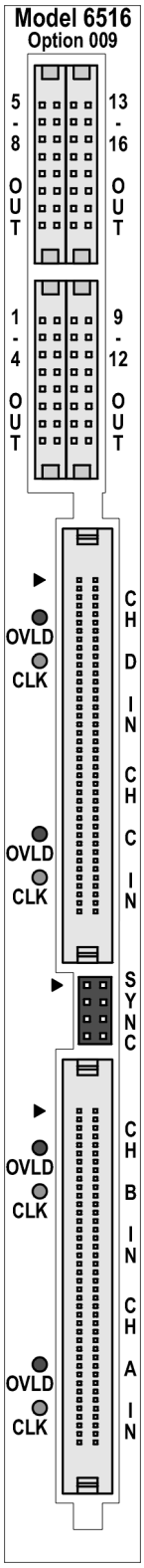


Figure 2-11: Model 6516 Option 009 Front Panel

2.4 Model 6516 Front Panels (continued)

2.4.2 Model 6516 Option 009 Front Panel (continued)

2.4.2.3 Sync Connector

The Sync connector on the Model 6516 Option 009, located between the two input data connectors, is an 8–pin, right angle, dual inline header, with the pins on a 0.100 x 0.100 grid. This connector is Samtec part # BST–104–08–L–D–230RA (Pentek part # 354.00807).

Care must be taken is selecting a mating connector for this interface, because of the limited clearance between the ends of this connector and the adjacent data input connectors (0.063" (1.6 mm)). For applications where the cable carrying the sync signals will only be connected from one source board to one target board, two styles of mating connectors may be used – either discrete 2–pin sockets or an 8–pin block. The two–pin connector housing may be Berg part number 65039–035 (Pentek #353.00201) or an equivalent, and the 8–pin block is Berg # 65043–033 (Pentek #353.00805) or equivalent. These connectors both use discrete .025" square socket pins, Berg #48254–000 (Pentek #354.00104). Berg makes two hand tools used in assembling these connectors, part #'s HT–208 (the pin crimper for 22 – 26 AWG wire) and HT–80PV (a pin removal tool).

If the sync cable must be daisy–chained among several boards, we would recommend obtaining a custom IDC cable strip from Samtec, Inc. Samtec's part number for these items begin with IDSD–04–D, and continue with optional characters specifying end assembly, connector plating, assembled length, number of connectors, distance between connectors, etc. Complete information about these pre–assembled cables can be found on pages 158 – 159 of Samtec's July 1998 Full Line Catalog (F–198–1).

Table 2-1: Model 6516, Option 009 - Sync Connectror Pinout

Pin #	Signal	Pin #	Signal
1	Sync 1	2	Signal
2	Sync 2	4	
3	Sync 3	6	Ground
4	Sync 4	8	

The pinout of the Sync Connector is given in Table 2–1, at the left. The Sync signals may be either inputs or outputs, depending upon programming (see Sections 3.5.8.4 and 3.5.8.5 for programming details). Separate termination resistor networks used for each Sync pin, and the networks are socketed, so that they can be removed if you will use a given pin as an output, or changed if needed for your application. The installed network is a 330 Ω pullup resistor to +5 V and a 390 Ω pulldown resistor to GND, as shown in Figure 2–10, on page 26 of this manual. When used as inputs, each sync signal drives 2 74FCT–type loads. When used as outputs, the sync pins are driven by the output of a 74FCT241.

2.4 Model 6516 Front Panels (continued)

2.4.2 Model 6516 Option 009 Front Panel (continued)

2.4.2.4 LED Indicators

The LED Indicators on the front panel of the Model 6516 Option 009 are identical to those on the standard model. All information given in Section 2.4.1.3, on page 26 of this manual, is also applicable to the indicators on this panel.

2.4.3 The FPDP (Option 008) Front Panel

The front panel for the Model 6516 Option 008, with FPDP input connectors and circuitry, is shown in Figure 2–12, at the right. This device also has a separate sync connector located between the FPDP input con–nectors. The differences between the standard Model 6516 and the Model 6516 Option 008 are all in the input stage, where the connectors and input circuits are dif–ferent. All features of this panel will be discussed in the subsections beginning below.

2.4.3.1 Comm Port Connectors

The Comm Port connectors on the 6516 Option 008 are identical to those on the standard model. For complete informa–tion on these connectors, please refer to Section 2.4.1.1, beginning on page 23 of this manual.

2.4.3.2 Digitized Signal Input Connectors

The input connectors used on the Model 6516 Option 008 are of the type specified by the FPDP specification, i. e., KEL part #8831E–080–170L (Pentek part # 351.08050). This connector connects 80 conductors in two rows of pins to the mating connector; however, the board pinout pattern is in four rows.

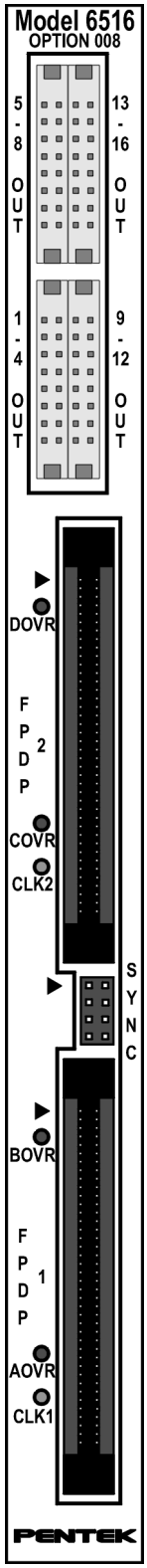
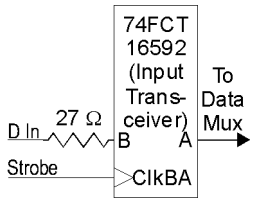


Figure 2-12: Model 6516 Option 008 Front Panel

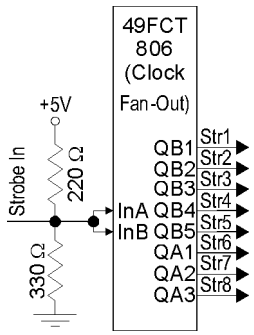
2.4 Model 6516 Front Panels (continued)

2.4.3 The FPDP (Option 008) Front Panel (continued)

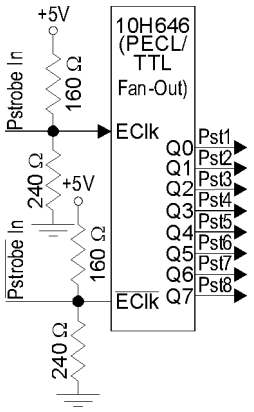
2.4.3.2 Digitized Signal Input Connectors (continued)



(a) Data Inputs



(b) Strobe Input



(c) PStrobe Inputs

Figure 2-13: Model 6516 Option 008 - Input Circuits

The mating connector takes a single ribbon cable of 80 conductors on 25 mil pitch and includes a locking arrangement to prevent unintentional disconnection. The mating connector's part number, with the strain relief, is KEL #8825E–080–175 (Pentek part # 353.08050). Two of these connectors are included with each Model 6516 Option 008. The recommended .025" pitch flat ribbon cable is 3M #3756/80.

The pinout of the input connectors conforms to the FPDP **inverted** connector pin assignment, with pin 1 at the upper left, as shown in Figure 2–14, on the next page.

NOTE: **Exercise caution in constructing your FPDP cable.** Remember that for an inverted FPDP connector (the type used on this unit), the connector is crimped on to the cable such that **Pin 1 of the connector is connected to pin 80 of the cable**. If a non–inverted connector is connected to the Model 6516 Option 008's data input connector, signal lines will be grounded and fuses are likely to blow in the driver, the receiver or both units.

The two 80–pin connectors each contain a 32–bit FPDP data bus, which is divided into two 16–bit data busses on the Model 6516 Option 008, one data bus for each input channel. The connector nearest the bottom of the panel carries data for Input Channel A (on pins 1 – 24) and Input Channel B (on pins 26 – 48). The other FPDP connector (the one nearest the center of the panel) carries data for Input Channel C (on pins 1 – 24) and Input Channel D (on pins 26 – 48). The remaining pins of the connectors each carry a group of timing & handshaking signals shared by the two input channels.

The 32 data input signals on each connector are series–terminated by 27 Ω resistors, and are then received by a pair of 74FCT16592 registered transceivers. FCT is a high–performance, TTL compatible CMOS logic family manufactured by Integrated Device Technology (IDT), among others. The input circuit is shown schematically in Figure 2–13(a), at the top left. One of these devices is provided for each input channel, and another one is provided for the DVALID, SYNC, PIO1 and PIO2 signals, which are also series–terminated with 27 Ω resistors. The SUSPEND, DIR and NRDY signals are unused on the Model 6516 Option 008, and are simply pulled up to +5 V through 1 kΩ resistors. All of the resistors and termination networks referred to in this paragraph are surface–mount parts, and are therefore not socketed.

FPDP Inverted Input Connector Pin Assignments for Lower (& Upper) Connector			
Pin #	**Signal**	**Pin #**	**Signal**
1	GND	2	D00: Ch A (C) - D0 in
3	D01: Ch A (C) - D1 in	4	GND
5	D02: Ch A (C) - D2 in	6	D03: Ch A (C) - D3 in
7	GND	8	D04: Ch A (C) - D4 in
9	D05: Ch A (C) - D5 in	10	GND
11	D06: Ch A (C) - D6 in	12	D07: Ch A (C) - D7 in
13	GND	14	D08: Ch A (C) - D8 in
15	D09: Ch A (C) - D9 in	16	GND
17	D10: Ch A (C) - D10 in	18	D11: Ch A (C) - D11 in
19	GND	20	D12: Ch A (C) - D12 in
21	D13: Ch A (C) - D13 in	22	GND
23	D14: Ch A (C) - D14 in	24	D15: Ch A (C) - D15 in
25	GND	26	D16: Ch B (D) - D0 in
27	D17: Ch B (D) - D1 in	28	GND
29	D18: Ch B (D) - D2 in	30	D19: Ch B (D) - D3 in
31	GND	32	D20: Ch B (D) - D4 in
33	D21: Ch B (D) - D5 in	34	GND
35	D22: Ch B (D) - D6 in	36	D23: Ch B (D) - D7 in
37	GND	38	D24: Ch B (D) - D8 in
39	D25: Ch B (D) - D9 in	40	GND
41	D26: Ch B (D) - D10 in	42	D27: Ch B (D) - D11 in
43	GND	44	D28: Ch B (D) - D12 in
45	D29: Ch B (D) - D13 in	46	GND
47	D30: Ch B (D) - D14 in	48	D31: Ch B (D) - D15 in
49	GND	50	$\overline{\text{DVALID}}$
51	GND	52	$\overline{\text{SYNC}}$
53	GND	54	$\overline{\text{PSTROBE}}$
55	GND	56	PSTROBE
57	GND	58	Reserved 1
59	GND	60	Reserved 2
61	GND	62	PIO1
63	GND	64	PIO2
65	GND	66	GND
67	GND	68	$\overline{\text{SUSPEND}}$
69	GND	70	Reserved 3
71	GND	72	$\overline{\text{DIR}}$
73	GND	74	$\overline{\text{NRDY}}$
75	GND	76	GND
77	GND	78	GND
79	STROB	80	GND

Note: In each input channel, Bit D0 is the LSB.

The mating cable socket connector, with strain releif, is KEL # 8825E-080-175.
Two of these items are supplied with the Model 6516 Option 008.
The recommended .025" pitch flat ribbon cable is 3M # 3756/080.

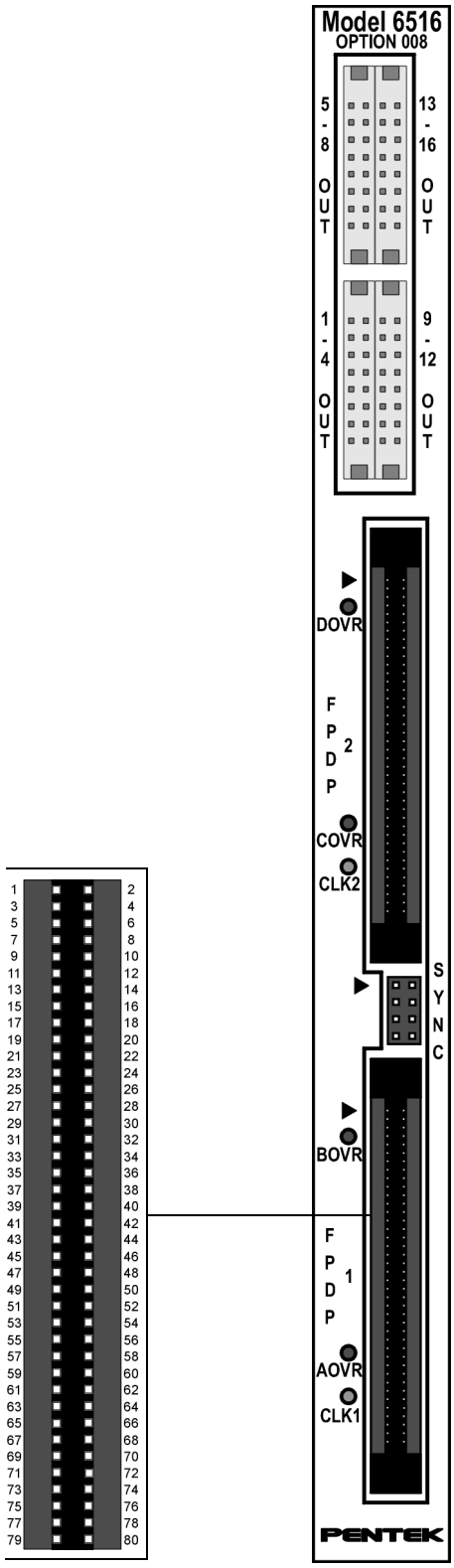


Figure 2-14: Model 6516 Option 008 - FPDP Input Connectors - Pin Assignments

2.4 Model 6516 Front Panels (continued)

2.4.3 The FPDP (Option 008) Front Panel (continued)

2.4.3.2 Digitized Signal Input Connectors (continued)

The clock signal to the input transceivers is selectable (see Section 3.5.8.7 for programming information), and may come from either the TTL STROB signal, or the differential PECL PSTROBE, $\overline{\text{PSTROBE}}$ signal pair. PECL is a logic family using standard ECL parts with a single, +5 V supply. The PECL logic high level is ≈ +4.2 V, and the logic low level is ≈ +3.4 V

The STROB input from each FPDP connector is pulled up to +5 V through 220 Ω, and pulled down to ground with 330 Ω. The signal is then delivered to a 49FCT806A clock buffer for fan−out. Figure 2−13(b), at the left center of page 30, shows this input circuit. The termination network on the PSTROBE, $\overline{\text{PSTROBE}}$ signal pair is a 160 Ω pullup to +5 V, and a 240 Ω pulldown to ground. The signal pair is differentially received by a 10H646, which translates the signal to single−ended TTL levels and provides fan−out for the clock signal. This input curcuit is shown in Figure 2−13(c), bottom left on page 30. The terminations used on the strobe signals are those recommended by the FPDP specification, and the parts used are not socketed.

CAUTION!!

When inserting and removing the FPDP connectors, be careful not to bend any pins on the SYNC connector, located between them.

2.4.3.3 Sync Connector

This connector, and the signals it is intended to carry, are identical to the connector and signals described for the Model 6516 Option 009's Sync connector, described in Section 2.4.2.3, beginning on page 28 of this manual. All the information given there pertains to this connector as well.

2.4 Model 6516 Front Panels (continued)

2.4.3 The FPDP (Option 008) Front Panel (continued)

2.4.3.4 LED Indicators

Three LED indicators are provided on the front panel of the Model 6516 Option 008 for each FPDP connector, one clock LED and two Overload LEDs. These will be discussed below.

- **Clock (CLK(N)) LED (Green) –** The green LED located next to, and near the bottom of, each FPDP connector will be lit when a clock signal is detected for that input pair. If this LED is not lit, then no clock signal has been detected and no data from this stream can be processed. The clock signal, in this case, refers to either the TTL STROB signal or the differential PECL PSTROBE, $\overline{\text{PSTROBE}}$ signal pair, whichever is selected as the input transceiver clock signal in the Input Control Register (see Section 3.5.8.7 for details). The **CLK1** LED, located nearest the bottom of the panel, indicates the presence (when lit) or absence (when not lit) of a clock signal for the FPDP1 connector, i. e., for Input Channels A and B. The **CLK2** LED, located above and to the left of the SYNC connector, indicates the presence (when lit) or absence (when not lit) of a clock signal for the FPDP2 connector, i. e., for Input Channels C and D.

- **Overload ((X)OVR) LED (Red) –** The two red LEDs adjacent to each FPDP connector were initially intended as indicators for the threshold detection function (hence the name – this, in fact, is what the firmware uses these LEDs for), and may still be used as such if desired. The states of these LEDs are controlled by user–addressable bits in the Input Control Registers (see Section 3.5.8.2 for further details). As used by the firmware, these four red LEDs indicate a threshold crossing on Channels D (the red LED marked **DOVR**), C (marked **COVR**), B (**BOVR)** and A (**AOVR**).

2.4 Model 6516 Front Panels (continued)

2.4.4 Model 6516 Option 018 Front Panel

Option 018 for the Model 6516 is available ONLY in combination with Option 008, and makes the following changes to Option 008: (1) The output resolution is 32 bits, rather than 16 bits; (2) The four Comm Ports at the top of the panel are real 'C40 Comm Ports (configured as inputs by default), rather than hardware–driven, output–only ports. The features of this panel are discussed in the subsections below.

2.4.4.1 Comm Port Connectors

The four connectors at the top of the front panel of the Model 6516 Option 018 are connected, through buffers, to four of the Comm Ports on the on–board 'C40 processor. They carry signals that eventually find their way to Comm Ports 2, 3, 4 and 5 of the 'C40. As all experienced 'C40 users know, the 'C40's default power up configuration has Comm Ports 0, 1 and 2 configured as outputs, and Comm Ports 3, 4 and 5 configured as inputs. Circuitry is provided to force the token on Comm Port 2, making it an input Port.

The replacement of the hardware–controlled output ports with true 'C40 Comm ports implies that receiver output data is not available via the Comm ports in this option. That is , in fact, the case. Receiver output data is only available over the VMEbus on the Model 6516 Option 018.

The pinouts of these connectors is shown in Figure 2–7, on page 23. Information on mating connectors, cabling, tooling, etc. for these connectors can be found beginning below the aforementioned figure, in Section 2.4.1.1 of this manual.

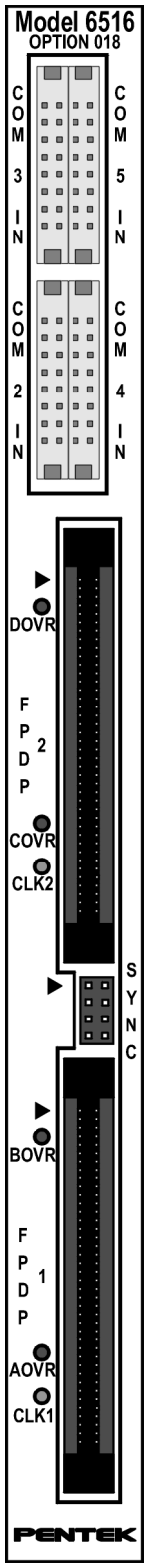


Figure 2-15: Model 6516 Option 018 Front Panel

2.4 Model 6516 Front Panels (continued)

2.4.4 Model 6516 Option 018 Front Panel (continued)

2.4.4.2 Digitized Signal Input Connectors

The two FPDP connectors on the front panel of the Model 6516 Option 018 are in every way identical to those on the Model 6516 Option 008. For details, please refer to Section 2.4.3.2, beginning on page 29 of this manual.

2.4.4.3 Sync Connector

This connector, and the signals it is intended to carry, are identical to the connector and signals described for the Model 6516 Option 009's Sync connector, described in Section 2.4.2.3, beginning on page 28 of this manual. All the information given there pertains to this connector as well.

2.4.4.4 LED Indicators

The LED indicators on the front panel of the Model 6516 Option 018 are identical in quantity, color and function to those on the Model 6516 Option 008. Please refer to Section 2.4.3.4, on page 33 of this manual, for complete details.

2.5 Inserting and Removing the Model 6516 in the VMEbus Card Cage

CAUTION!!

Never INSERT or REMOVE the Model 6516 while card cage power is turned on!

2.5.1 Inserting the Unit

Align the Model 6516 card edge with the cage card guides and push evenly on both ejector handles until the front panel seats against the upper and lower rails. Secure the unit in the cage by tightening the upper and lower retaining screws in the front panel.

2.5.2 Removing the Unit

Loosen the upper and lower retaining screws; they are captive screws and will remain part of the panel assembly. Gently push both ejector handles apart (away from the center of the panel) until the unit unseats itself from the cage connectors. Then pull outward to remove the unit.

Chapter 3: Memory Maps and Register Descriptions

3.1 Programming Overview

The Model 6516 is controlled by commands sent over the VMEbus to memory−mapped control registers. Each of these registers will be described in detail in this chapter. The 6516 acts as a VMEbus slave device.

Six control registers (five associated with VMEbus operations and the other with the Test Bus Controller) are mapped into the Model 6516's VMEbus A16 memory space. Access to this address space is determined by the A16 Base Address Switches, which are described in Section 2.2. All control register addresses are given as offsets from the base address defined by the settings of these switches. The form used is A16_base+0xYZ, where YZ is the hexadecimal offset from the Base Address. A VMEbus memory map to all the Model 6516's registers is included in this chapter.

The data FIFO memories and the control registers associated with Digital Receiver operations are mapped into A24 and/or A32 VMEbus memory space(s). These resources can also be accessed by the 'C40 processor, on its global bus. This chapter also includes complete memory maps for the 'C40 (both the local and global busses) and for VMEbus A24/A32 space.

3.2 VMEbus Address Modifier Codes

There are three addressing modes defined for VMEbus transfers: 32−bit, 24−bit and 16−bit modes. These modes are provided to support a variety of bus devices ranging from simple I/O functions with only a few memory locations to large RAM arrays with hundreds of Megabytes of storage. The addressing mode used for a given transaction is determined by a special 6−bit code that is placed on six dedicated VMEbus lines by the current bus master. This code is known as the Address Modifier, or AM, code.

The VMEbus specification defines the usage of certain AM codes as shown in Table 3−1, below. The Model 6516 responds only to those codes shown in the table.

Table 3-1: VMEbus Address Modifiers

AM Code	Access	AM Code	Access
0x09	A32 - Extended Nonprivileged Data Access	0x0B	A32 - Extended Nonprivileged Block Transfer
0x0D	A32 - Extended Supervisory Data Access	0x0F	A32 - Extended Supervisory Block Transfer
0x29	A16 - Short Non-Privileged Access	0x2D	A16 - Short Supervisory Access
0x39	A24 - Standard Nonprivileged Data Access	0x3B	A24 - Standard Nonprivileged Block Transfer
0x3D	A24 - Standard Supervisory Data Access	0x3F	A24 - Standard Supervisory Block Transfer
NOTE: The model 6516 **ONLY** responds to the address modifier codes shown in this table.			

3.3 VMEbus Access

The Model 6516 presents two distinct memory maps to VMEbus masters, one in A16 space and another in A32 and/or A24 space. When power is initially applied to this device, only A16 access is enabled. The 6516 occupies 256 Bytes of VME A16 memory. A32 access is enabled by the first write to the VMEbus Address Register. Once enabled, A32 access cannot be disabled without cycling power or resetting the device. Block access is permitted in A32 space only (A24 block accesses (AM codes 0x3F and 0x3B) will be DTACKed by the 6516, but no block transfers can be conducted with those modifiers). The 6516 occupies 4 MBytes of A32 VMEbus memory. A bit is provided in the VME Address register to enable A24 access. When A24 Access is enabled, the 6516 occupies 1 MByte of A24 memory AND 4 MBytes of A32 memory. The resources available in A16 space are accessible exclusively to the VMEbus. Those in the A24 and A32 memory maps can also be accessed by the on−board 'C40 processor.

The sections that follow present the memory maps for all the resources available on the Model 6516. A16−accessible resources are discussed first. Each section provides a memory map table giving the address space in question, and then describes the registers and other available resources.

3.4 VMEbus A16 Slave Memory Map

The memory address map for the six resources available in VMEbus A16 space on the Model 6516 is given in Table 3−2, below. Note that most of these resources are accessible for both read and write operations. The exceptions are the VME−to−'C40 interrupt address, which is accessible only for write operations, and the VMEbus Status Register, which is accessible only for read operations. To access these registers, the bus master that conducts the read or write cycle must select A16 address space by using AM code 2D or 29 (see Table 3−1, previous page). All resources on the Model 6516 are mapped on 4−byte boundaries (i. e., the least significant hex address digit is either 0, 4, 8 or C), to simplify master access to the board by DSP−based devices that use 32−bit addressing, such as Pentek's Models 4283, 4284, 4285 and 4290.

Table 3-2: Model 6516 - VMEbus A16 Slave Memory Map

Address	Access	Resource	Symbolic Name
A16_base+0x00	R/W	VMEbus Control Register	Ctrl_Reg
A16_base+0x04	R/W	VMEbus Address Register	Addr_Reg
A16_base+0x08	W. O.	VMEbus Interrupt to 'C40	C40_Int
A16_base+0x0C	R/W	Global Bus Address Page Register	Page _Reg
A16_base+0x10	R. O.	VMEbus Status Register	VME_Stat
A16_base+0x80 - A16_base+0xFF	R/W	Test Bus Controller Registers	TBC

3.4 VMEbus A16 Slave Memory Map (continued)

All VMEbus A16 space addresses are expressed in hexadecimal notation, as offsets from the A16_base address. This address is determined by the settings of the two 16–position rotary switches, SW1 and SW2 (see Section 2.2). The A16 memory space occupied by this device is a 256 Byte block starting at the A16_base address. All accesses within this region cause the 6516 to handshake with the VMEbus, **and no other VME–bus device may be mapped within this same 256–byte region of A16 address space**. If more than one 6516 will be used in the same card cage, the A16 Base Address switches must be set to give each board a unique address.

3.4.1 VMEbus Control Register – R/W @ A16_base+0x00

The VMEbus Control Register contains an 8–bit IACK Vector field and a 3–bit IRQ Level field. It also has individual bits that can reset the entire board or just the 'C40 processor, a bit that determines the source of the code that will be used to boot the 'C40 and a bit to enable VME access to the Test Bus Controller. Table 3–3, below, shows the bit arrangement within the Model 6516's VMEbus Control Register. The subsections that follow the table give detailed descriptions of these bits and bit fields.

Table 3-3: Model 6516 - VMEbus Control Register - R/W @ A16_base+0x00

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Name	Not Used	C40_Boot _Source	TBC_ Enable	C40_ Reset	Board _Reset	IRQ_Level _Select_2	IRQ_Level _Select_1	IRQ_Level _Select_0
Bit Function	Write as 0, Mask on Read	1 = Comm Port 0 = EEPROM	1 = Enabled 0 = Disabled	1 = Reset 0 = Run	1 = Reset 0 = Run	1,1,1=IRQ7 1,0,0=IRQ4	1,1,0=IRQ6 0,1,1=IRQ3 0,0,1=IRQ1	1.0.1=IRQ5 0,1,0=IRQ2 0,0,0=None
Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Field	VMEbus IACK Vector Address							
Destination	VME_D07	VME_D06	VME_D05	VME_D04	VME_D03	VME_D02	VME_D01	VME_D00

3.4.1.1 'C40 Boot Source Select – D14

When this bit is cleared to the logic '0' state, the 'C40 processor on the Model 6516 will boot from the code contained in the unit's EEPROM, at 'C40 address 0x0030 0000. This is the default case. Setting this bit to the logic '1' state, and toggling the 'C40 Reset bit (i. e., writing a '1', then a '0' to D12 in this register, see Section 3.4.1.3) causes the 'C40 to look to its Comm Ports for boot code.

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.1 VMEbus Control Register (continued)

3.4.1.1 'C40 Boot Source Select (continued)

Remember that on the Standard Model 6516, the Front Panel Comm Ports are PAL based, output−only devices, and are unconnected to the 'C40. However, this feature is very useful in units equipped with Option 018, where the PAL−controlled output ports are replaced with 'C40 input ports, and Option 034, in which the 'C40's Comm Ports are wired out to the VME−bus P2 connector. When instructed to boot from the Comm Ports, the 'C40 polls the ports and attempts to boot from the first port at which data is found.

3.4.1.2 Test Bus Controller Enable – D13

The 74ACT8990 is used as a JTAG−compliant Test Bus Controller on the Model 6516. To enable access to the 'C40 JTAG Interface from the VMEbus, set this bit to the logic '1' state. When this bit is cleared to the logic '0' state (the default state after power−up or reset), JTAG access to the 'C40 is disabled. For more information about the 74ACT8990, please refer to Texas Instruments' **Advanced Logic and Bus Interface Logic Data Book**.

3.4.1.3 'C40 Reset – D12

The Model 6516's 'C40 Processor will be held in its Reset state whenever this bit is set to the logic '1' state. To release the 'C40 and allow it to operate normally, this bit must be cleared to the logic '0' state. The 'C40 Reset function should be implemented as a toggle, i. e., this bit should be set to logic '1', then cleared back to '0'.

Note that this bit affects the processor only, not the Digital Receiver chips or the data FIFOs.

3.4.1.4 Board Reset – D11

All circuits and devices on the Model 6516 (including the 'C40 processor, the Digital Receiver Chips, and the data FIFOs) are held in the Reset state while this bit is in the logic '1' state. For normal operation of the digital receiver board, this bit must be cleared to the logic '0' state. The software reset function should be implemented as a toggle, i. e., this bit should be set to logic '1', then cleared back to '0'.

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.1 VMEbus Control Register (continued)

3.4.1.5 VMEbus IRQ Level Select – D10 > D8

This bit field should be written with the binary code for the desired VMEbus Interrupt Requester level for the Model 6516 (see Table 3–3, on page 39). Since IRQ level 0 is undefined, clearing all three of these bits to the logic '0' state disables the 6516's Interrupter interface.

3.4.1.6 VMEbus IACK Vector Address – D7 > D0

This 8–bit field is used to store the IACK Vector Address, which will be placed on the VME Data bus when the Model 6516 issues an interrupt to VME. Table 3–3, on page 39, shows the correspon–dence between the bits in this register and the VME Data Bus lines that are their destinations.

3.4.2 VMEbus Address Register – R/W @ A16_base+0x04

This 16–bit register contains a 10–bit field used for the VME A32 base address, a 4–bit field used for the VME A24 base address, a bit to enable A24 space VME slave access, and a Word Swap bit. A32 VME slave access is enabled by the first write operation to this register. Table 3–4, below, shows the arrangement of bits in the 6516's VMEbus Address Register. The subsec–tions that follow the table will provide more details about the register's bit fields and bits.

Table 3-4: Model 6516 - VMEbus Address Register - R/W @ A16_base+0x04

Bit #	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6
Bit Field	A32 Base Address									
Match	VME_A31	VME_A30	VME_A29	VME_A28	VME_A27	VME_A26	VME_A25	VME_A24	VME_A23	VME_A22

Bit #	D5	D4	D3	D2	D1	D0
Bit Name	A24_Enable	A24 Base Address				Word_Swap
Bit Function	1 = Enable 0 = Disable	VME_A23	VME_A22	VME_A21	VME_A20	1 = Swap 0 = No Swap

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.2 VMEbus Address Register (continued)

3.4.2.1 A32_Base Address – D15 > D6

When power is initially applied to the Model 6516 (or after a reset), the only enabled VMEbus interface is the A16 space slave. A32 slave access is enabled by the first write to this register, at the address written to the register's ten most significant bits. The bits in D15 – D6 in this register will be compared to the VMEbus A31 – A22 address lines, respectively, to determine if the 6516 is being addressed during a VME transaction. Once this register has been written, the only way to disable A32 slave access to the 6516 is to reset it (see Section 3.4.1.4), or to cycle power.

3.4.2.2 A24 Enable – D5

When this bit is cleared to the logic '0' state (the power–up default condition), A24 slave access to the 6516 is disabled. To enable A24 access, set this bit to the logic '1' state and write the desired base address to bits D4 – D1 in this register (see Section 3.4.2.3, below). Note that, because the first write to this register enables A32 slave access, it is not possible to enable A24 access to the 6516 without also enabling A32 access.

3.4.2.3 A24_Base Address – D4 > D1

When A24 slave access to the 6516 is enabled (see Section 3.4.2.2, above) the 6516 will respond to VME data cycles in A24 space when VME address bits A23 – A20 match the contents of bits D4 – D1 in this register, respectively.

3.4.2.4 Word Swap – D0

Word Swap Mode is used to allow Intel type processors to access the 6516's memory, by setting D0 in this register to a ‘1’. Leave this bit in the ‘0’ state if you are working with a Motorola host.

This is a very common function in VME cards. Remember that Intel and Motorola processors address words differently; with an Intel host, the upper 16 bits of the 32 bit longword are stored at locations with the VME address line A1 = '1'. A Motorola host stores those same bits at locations with A1 = '0'.

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.3 VMEbus Interrupt to 'C40 – W. O. @ A16_base+0x08

Any VMEbus write operation to this address causes an interrupt to be asserted on the 6516's 'C40 processor. This interrupt will drive the IIOF1 input of the 'C40. Since this device is not a VMEbus interrupt handler, the assertion of this interrupt will not result in a VMEbus IACK cycle. The data written during this operation is neither used nor stored and is therefore trivial.

3.4.4 Global Bus Address Page Register – R/W @ A 16_base+0x0C

This register allows the VMEbus to access resources on the 'C40's Global bus, when Global bus address bits above A19 are required (this corresponds to VME address bit A21; remember that the A32_base address uses VME address bits A31 – A22). The bits in this register are mapped to 'C40 Global address bits A27 – A20, which correspond to VME address bits A29 – A22, as shown in Table 3–5, below.

Table 3-5: Model 6516 - Global Bus Address Page Register - R/W @ A16_base+0x0C

Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Field	Global Bus Address Page							
Global Bits	Gbus_A27	Gbus_A26	Gbus_A25	Gbus_A24	Gbus_A23	Gbus_A22	Gbus_A21	Gbus_A20
VME Bits	VME_A29	VME_A28	VME_A27	VME_A26	VME_A25	VME_A24	VME_A23	VME_A22

Note that in the Model 6516's present configuration, there are no resources on the 'C40's Global Bus that require the bits this register provides for VMEbus access (i. e., no resources are mapped at 'C40 addresses above 0x800F FFFF, or VMEbus address above VME_base+0x003F FFFF). This register was included to allow for possible future expansion of the memory map.

3.4 VMEbus A16 Slave Memory Map (continued)

3.4.5 VMEbus Status Register – R. O. @ A16_base+0x10

Only two of the 16 bits in this register are used. One indicates whether an interrupt to the 'C40 processor is pending, and the other gives the status of the Test Bus Controller. Table 3–6, below, shows the register's bit arrangement. The subsections beginning on the next page describe the status bits.

Table 3-6: Model 6516 - VMEbus Status Register - R. O. @ A16_base+0x10

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Name	Pending_C40 _Interrupt	Not Used	TBC_ Ready	Not Used				
Bit Function	1 = True 0 = False	Mask when Reading	1 = Not Ready 0 = Ready	Mask when Reading.				
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Name	Not used							
Bit Function	Mask when Reading.							

3.4.5.1 Pending 'C40 Interrupt – D15

This bit will read back on the logic '1' state when the VMEbus has issued an interrupt to the 'C40 Processor that has not yet been acknowledged. Write operations to this bit have no effect. VME interrupts are issued by writing to the A16 VME address A16_base+0x08. The 'C40 acknowledges this interrupt by writing to address 0x8004 0000 in its Global memory map.

3.4.5.2 TBC_Ready – D13

If this bit reads back in the logic '0' state, the JTAG Interface to the Model 6516's 'C40 processor is ready for use. When the JTAG interface is busy, this bit will read back in the logic '1' state. For further information on the use of this interface, please refer to the 74ACT8990 Data Sheet in Texas Instruments' **Advanced Logic and Bus Interface Logic Data Book**.

3.4.6 Test Bus Controller Registers – R/W from A16_base+0x80 to A16_base+0xFF

A JTAG–compliant interface to the Model 6516's on–board TMS320C40 pro–cessor is provided by the Texas Instruments 74ACT8990. The register set of this device can be accessed from the VMEbus in the address range beginning at A16_base+0x80 and ending at A16_base+0xFF. For further information on these registers and their functions, please refer to the 74ACT8990 Data Sheet in Texas Instruments' **Advanced Logic and Bus Interface Logic Data Book**.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map

Table 3–7, below, shows the memory map for the resources shared by the VMEbus and the 'C40 processor on the 6516. The term VME_base is used for the A32 or A24 base address, depending upon the address modifier in use. The table gives both VME and 'C40 addresses for each resource. The subsections that begin on the next page describe these resources in detail.

Table 3-7: Model 6516 - Dual-Mapped Resources -'C40 Global and VMEbus A32/A24 Slave Address Spaces

Symbolic Name	Resource	Size	Access	VME Address	'C40 Address
SRAM	Global SRAM‡	128k x 32	R / W	VME_base+0x0000 0000 - VME_base+0x0007 FFFF	0x8000 0000 - 0x8001FFFF
C40_Int_Mask_1	'C40 Interrupt Mask Register #1	1 x 32	R / W	VME_base+0x0008 0000	0x8002 0000
C40_Int_Mask_2	'C40 Interrupt Mask Register #2	1 x 16	R / W	VME_base+0x0008 8000	0x8002 2000
Rec_Stat_Reg_1	Receiver Status Register #1	1 x 32	R. O.	VME_base+0x0009 0000	0x8002 4000
Rec_Stat_Reg_2	Receiver Status Register #2	1 x 16	R. O.	VME_base+0x0009 8000	0x8002 6000
F_Rst_Reg	FIFO Reset Register	1 x 8	R / W	VME_base+0x000A 0000	0x8002 8000
Out_Comm_Ctl	Output Comm Port Control Register	1 x 16	R / W	VME_base+0x000A 8000	0x8002 A000
Thr_Det_C	Chan. C Threshold Detection Reg.	1 x 16	R / W	VME_base+0x000B 8040	0x8002 E010
Thr_Det_D	Chan. D Threshold Detection Reg.	1 x 16	R / W	VME_base+0x000B 80C0	0x8002 E030
Rcvr_Ctl_2	Receiver Control Register #2	1 x 32	D32 - R/W*	VME_base+0x000B 8140	0x8002 E050
Input_Ctl	Input Control Register	1 x 32	D32 - R/W*	VME_base+0x000B 8240	0x8002 E090
Rcvr_Ctl_1	Receiver Control Register #1	1 x 32	D32 - R/W*	VME_base+0x000B 8280	0x8002 E0A0
Mux_Mode(_PIO)**	Multiplex and Mode (and PIO**) Register	1 x 16 **1 x 32	R / W	VME_base+0x000B 82C0	0x8002 E0B0
Tag_En	Tag Enable Register	1 x 8	R / W	VME_base+0x000B 8300	0x8002 E0C0
Thr_Det_A	Chan. A Threshold Detection Reg.	1 x 16	R / W	VME_base+0x000B 8340	0x8002 E0D0
Thr_Det_B	Chan. B Threshold Detection Reg.	1 x 16	R / W	VME_base+0x000B 8380	0x8002 E0E0
FIFO_1	FIFO - Receivers 1 & 2	8k x 16	R. O.	VME_base+0x000C 0000 - VME_base+0x000C 7FFF	0x8003 0000 0x8003 1FFF
FIFO_2	FIFO - Receivers 3 & 4	8k x 16	R. O.	VME_base+0x000C 8000 - VME_base+0x000C FFFF	0x8003 2000 - 0x8003 3FFF
FIFO_3	FIFO - Receivers 5 & 6	8k x 16	R. O.	VME_base+0x000D 0000 - VME_base+0x000D 7FFF	0x8003 4000 - 0x8003 5FFF
FIFO_4	FIFO - Receivers 7 & 8	8k x 16	R. O.	VME_base+0x000D 8000 - VME_base+0x000D FFFF	0x8003 6000 - 0x8003 7FFF
FIFO_5	FIFO - Receivers 9 & 10	8k x 16	R. O.	VME_base+0x000E 0000 - VME_base+0x000E 7FFF	0x8003 8000 - 0x8003 9FFF
FIFO_6	FIFO - Receivers 11 & 12	8k x 16	R. O.	VME_base+0x000E 8000 - VME_base+0x000E FFFF	0x8003 A000 - 0x8003 BFFF
FIFO_7	FIFO - Receivers 13 & 14	8k x 16	R. O.	VME_base+0x000F 0000 - VME_base+0x000F 7FFF	0x8003 C000 - 0x8003 DFFF
FIFO_8	FIFO - Receivers 15 & 6	8k x 16	R. O.	VME_base+0x000F 8000 - VME_base+0x000F FFFF	0x8003 E000 - 0x8003 FFFF
C40_IACK	'C40 Interrupt Acknowledge Address	1 x 16	W. O.†	VME_base+0x0010 0000	0x8004 0000
VME_Int	'C40 Interrupt to VMEbus	1 x 16	W. O.†	VME_base+0x0014 0000	0x8005 0000
Flash	Global Flash EEPROM	512k x 8	R / W†	VME_base+0x0020 0000 - VME_base+0x003F FFFF	0x8008 0000 - 0x800F FFFF

* - The Input Control Register and the Receiver Control Register can only be properly accessed in D32 mode. Dual-cycle D16 write operations to these registers will result in the upper word not bring stored.

** - Mux_Mode register is 32 bits wide in Model 6516 Option 008. The upper 16 bits are used for features involving the FPDP PIO signals.

† - C40_IACK, VME_Int and Flash are accessible to the 'C40 and in A32 VME space only (i. e., these resources cannot be accessed in VME A24 space.)

‡ - The firmware uses the last portion of SRAM to store a copy of the HSP50016 Control Words. See Section 3.5.1 for additional informaiton.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.1 Global SRAM – R/W @ SRAM

A 512 kByte bank of shared SRAM is provided for data to be shared by VMEbus masters and the 6516's 'C40 processor. This bank occupies VME memory space from the VME_base address to VME_base+0x0007 FFFF, and 'C40 memory space from address 0x8000 0000 to address 0x8001 FFFF. Remember that each VME longword address is aligned on 'C40 byte addresses. The Firmware uses the 511 highest addresses in this SRAM bank to store shadow copies of the 40–bit Write–Only Control Registers of the HSP50016 digital receivers. Each 40–bit Control Word occupies two longwords (i. e., eight bytes) in this region. Thus, the eight control words associated with each receiver occupy sixteen longwords (64 bytes), resulting in a VMEbus address range of 0x40 for each receiver's shadow registers. Table 3–8, below, gives the VMEbus addresses used for each receiver's shadow registers. Table 3–9, at the top of the next page, gives the 'C40 addresses in SRAM for the shadow registers. Note that the two addresses given in these two tables for the shadow of a given control word for a given receiver actually point to the same physical memory location (for example, the address VME_base+0x0007 F800 (the shadow of Receiver1, control word 0 low) is the same physical SRAM location as address 0x8001 FE00 in the 'C40's memory map).

Table 3-8: Model 6516 - VMEbus Addresses of HSP50016 Control Register Shadows in Global SRAM (VME_base+0x0007 . . .)

Control Word	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 5	Rec. 6	Rec. 7	Rec. 8	Rec. 9	Rec. 10	Rec. 11	Rec. 12	Rec. 13	Rec. 14	Rec. 15	Rec. 16
0 Low	F800	F840	F880	F8C0	F900	F940	F980	F9C0	FA00	FA40	FA80	FAC0	FB00	FB40	FB80	FBC0
0 High	F804	F844	F884	F8C4	F904	F944	F984	F9C4	FA04	FA44	FA84	FAC4	FB04	FB44	FB84	FBC4
1 Low	F808	F848	F888	F8C8	F908	F948	F988	F9C8	FA08	FA48	FA88	FAC8	FB08	FB48	FB88	FBC8
1 High	F80C	F84C	F88C	F8CC	F90C	F94C	F98C	F9CC	FA0C	FA4C	FA8C	FACC	FB0C	FB4C	FB8C	FBCC
2 Low	F810	F850	F890	F8D0	F910	F950	F990	F9D0	FA10	FA50	FA90	FAD0	FB10	FB50	FB90	FBD0
2 High	F814	F854	F894	F8D4	F914	F954	F994	F9D4	FA14	FA54	FA94	FAD4	FB14	FB54	FB94	FBD4
3 Low	F818	F858	F898	F8D8	F918	F958	F998	F9D8	FA18	FA58	FA98	FAD8	FB18	FB58	FB98	FBD8
3 High	F81C	F85C	F89C	F8DC	F91C	F95C	F99C	F9DC	FA1C	FA5C	FA9C	FADC	FB1C	FB5C	FB9C	FBDC
4 Low	F820	F860	F8A0	F8E0	F920	F960	F9A0	F9E0	FA20	FA60	FAA0	FAE0	FB20	FB60	FBA0	FBE0
4 High	F824	F864	F8A4	F8E4	F924	F964	F9A4	F9E4	FA24	FA64	FAA4	FAE4	FB24	FB64	FBA4	FBE4
5 Low	F828	F868	F8A8	F8E8	F928	F968	F9A8	F9E8	FA28	FA68	FAA8	FAE8	FB28	FB68	FBA8	FBE8
5 High	F82C	F86C	F8AC	F8EC	F92C	F96C	F9AC	F9EC	FA2C	FA6C	FAAC	FAEC	FB2C	FB6C	FBAC	FBEC
6 Low	F830	F870	F8B0	F8F0	F930	F970	F9B0	F9F0	FA30	FA70	FAB0	FAF0	FB30	FB70	FBB0	FBF0
6 High	F834	F874	F8B4	F8F4	F934	F974	F9B4	F9F4	FA34	FA74	FAB4	FAF4	FB34	FB74	FBB4	FBF4
7 Low	F838	F878	F8B8	F8F8	F938	F978	F9B8	F9F9	FA38	FA78	FAB8	FAFA	FB38	FB78	FBB8	FBFB
7 High	F83C	F87C	F8BC	F8FC	F93C	F97C	F9BC	F9FC	FA3C	FA7C	FABC	FAFC	FB3C	FB7C	FBBC	FBFC

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.1 Global SRAM (continued)

Table 3-9: Model 6516 - TMS320C40 Addresses of HSP50016 Control Register Shadows in Global SRAM (0x8001 . . .)

Control Word	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 5	Rec. 6	Rec. 7	Rec. 8	Rec. 9	Rec. 10	Rec. 11	Rec. 12	Rec. 13	Rec. 14	Rec. 15	Rec. 16
0 Low	FE00	FE10	FE20	FE30	FE40	FE50	FE60	FE70	FE80	FE90	FEA0	FEB0	FEC0	FED0	FEE0	FEF0
0 High	FE01	FE11	FE21	FE31	FE41	FE51	FE61	FE71	FE81	FE91	FEA1	FEB1	FEC1	FED1	FEE1	FEF1
1 Low	FE02	FE12	FE22	FE32	FE42	FE52	FE62	FE72	FE82	FE92	FEA2	FEB2	FEC2	FED2	FEE2	FEF2
1 High	FE03	FE13	FE23	FE33	FE43	FE53	FE63	FE73	FE83	FE93	FEA3	FEB3	FEC3	FED3	FEE3	FEF3
2 Low	FE04	FE14	FE24	FE34	FE44	FE54	FE64	FE74	FE84	FE94	FEA4	FEB4	FEC4	FED4	FEE4	FEF4
2 High	FE05	FE15	FE25	FE35	FE45	FE55	FE65	FE75	FE85	FE95	FEA5	FEB5	FEC5	FED5	FEE5	FEF5
3 Low	FE06	FE16	FE26	FE36	FE46	FE56	FE66	FE76	FE86	FE96	FEA6	FEB6	FEC6	FED6	FEE6	FEF6
3 High	FE07	FE17	FE27	FE37	FE47	FE57	FE67	FE77	FE87	FE97	FEA7	FEB7	FEC7	FED7	FEE7	FEF7
4 Low	FE08	FE18	FE28	FE38	FE48	FE58	FE68	FE78	FE88	FE98	FEA8	FEB8	FEC8	FED8	FEE8	FEF8
4 High	FE09	FE19	FE29	FE39	FE49	FE59	FE69	FE79	FE89	FE99	FEA9	FEB9	FEC9	FED9	FEE9	FEF9
5 Low	FE0A	FE1A	FE2A	FE3A	FE4A	FE5A	FE6A	FE7A	FE8A	FE9A	FEAA	FEBA	FECA	FEDA	FEEA	FEFA
5 High	FE0B	FE1B	FE2B	FE3B	FE4B	FE5B	FE6B	FE7B	FE8B	FE9B	FEAB	FEBB	FECB	FEDB	FEEB	FEFB
6 Low	FE0C	FE1C	FE2C	FE3C	FE4C	FE5C	FE6C	FE7C	FE8C	FE9C	FEAC	FEBC	FECC	FEDC	FEEC	FEFC
6 High	FE0D	FE1D	FE2D	FE3D	FE4D	FE5D	FE6D	FE7D	FE8D	FE9D	FEAD	FEBD	FECD	FEDD	FEED	FEFD
7 Low	FE0E	FE1E	FE2E	FE3E	FE4E	FE5E	FE6E	FE7E	FE8E	FE9E	FEAE	FEBE	FECE	FEDE	FEEE	FEFE
7 High	FE0F	FE1F	FE2F	FE3F	FE4F	FE5F	FE6F	FE7F	FE8F	FE9F	FEAF	FEBF	FECF	FEDF	FEEF	FEFF

3.5.2 'C40 Interrupt Mask Register #1 – R/W @ C40_Int_Mask_1

This 32–bit register can be accessed by the 'C40 processor and by the VMEbus. The upper 8 bits are unused, and should be written with zeros and masked when read. Four bits (one associated with each input channel) in the next lower byte field are used enable or disable (i. e., mask) generation of a 'C40 interrupt when the Threshold Detector flags go true. In units equipped with Option 008, the other four bits in this field are used for 'C40 interrupts based on the states of the FPDP PIO signals. The next lower byte field also contains only four active bits (one for each channel), used for 'C40 interrupts based on the state of the Clock Not Ready flags. The lowest byte field contains one bit associ–ated with each output FIFO, and is used for FIFO Full interrupts to the 'C40 processor. All of the interrupts under this register's control will be asserted on the IIOF0 input of the 'C40 processor. Table 3–10, at the top of the next page, shows the bit arrangement within 'C40 Interrupt Mask Register #1. The subsections beginningbelow the table give detailed descriptions of the register's bit fields.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.2 'C40 Interrupt Mask Register #1 (continued)

Table 3-10: Model 6516 - 'C40 Interrupt Mask Register #1 - R/W @ C40_Int_Mask_1								
Bit #	**D31**	**D30**	**D29**	**D28**	**D27**	**D26**	**D25**	**D24**
Bit Field	Not Used (Write as 0, mask when reading)							
Bit #	**D23***	**D22**	**D21***	**D20**	**D19***	**D18**	**D17***	**D16**
Bit Field	Threshold Detector (& PIO*) Interrupts							
I/P Channel	FPDP2-PIO2*	Th. Chan. D	FPDP1-PIO2*	Th. Chan. C	FPDP2-PIO1*	Th. Chan. B	FPDP1-PIO1*	Th. Chan. A
Bit Function	1 = Interrupt Enabled 0 = Interrupt Masked							
Bit #	**D15**	**D14†**	**D13**	**D12**	**D11**	**D10†**	**D9**	**D8**
Bit Field	Clock Not Ready Flag Interrupts							
I/P Channel	Reserved	Channel D	Reserved	Channel C	Reserved	Channel B	Reserved	Channel A
Bit Function	Write as 0, Mask on reads	1 = Enabled 0 = Masked	Write as 0, Mask on reads	1 = Enabled 0 = Masked	Write as 0, Mask on reads	1 = Enabled 0 = Masked	Write as 0, Mask on reads	1 = Enabled 0 = Masked
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Field	FIFO Full Flag Interrupts							
Receiver #	Receiver 8	Receiver 7	Receiver 6	Receiver 5	Receiver 4	Receiver 3	Receiver 2	Receiver 1
Bit Function	1 = Interrupt Enabled 0 = Interrupt Masked							
* - in Option 008 only - In other units these bits are Reserved & Unused - Write with 0's, mask when reading. † - In Option 008, these bits are Reserved and unused. Also, the Channel A clock bit in this group is for FPDP1, and the Channel C clock bit is for FPDP2.								

3.5.2.1 Threshold Detector Interrupts – D22, D20, D18 & D16

Setting one of these four bits to the logic '1' state will cause a 'C40 interrupt to be generated when the Threshold Detector Flag becomes true for the input channel associated with the given bit. Threshold interrupts will be asserted on the 'C40's IIOF0 input. These interrupts are disabled for those channels whose associated bits in this field are cleared to the logic '0' state.

The threshold conditions for a given input channel are determined by that channel's Threshold Detection Register. See Section 3.5.14 for information about these registers.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.2 'C40 Interrupt Mask Register #1 (continued)

3.5.2.2 PIO Interrupts – D23, D21, D19 & D17 – Option 008 ONLY

The Model 6516 Option 008 can be programmed to generate a 'C40 interrupt when the conditions in the PIO portion of the Mux_Mode_PIO register are met (see Section 3.5.12 for details). These interrupts are enabled when the corresponding bits in this register are set to the logic '1' state, and are masked when the corresponding bits are cleared to the logic '0' state.

NOTE: In units not equipped with Option 008, these four bits are reserved. In such units, these bits should **ALWAYS** be written with logic '0's and masked when read.

3.5.2.3 Clock Not Ready Flag Interrupts – D14, D12, D10 & D8

Setting one of these four bits to the logic '1' state will cause a 'C40 interrupt to be generated when the clock signal is not detected on the input channel associated with a given bit. Clock interrupts will be asserted on the 'C40's IIOF0 input. These interrupts are disabled for those channels whose associated bits in this field are cleared to the logic '0' state.

In the Model 6516 Option 008, Bits D14 and D10 are unused, D12 is the FPDP2 Clock Not Ready Flag, and D8 is the FPDP1 Clock Not Ready Flag.

NOTE: The other four bits in this 8–bit field (D15, D13, D11, & D9) are reserved. They should **ALWAYS** be written with logic '0's and masked when read.

3.5.2.4 FIFO Full Flag Interrupts – D7 > D0

Setting any of these eight bits to the logic '1' state will cause a 'C40 interrupt to be generated when the data FIFO for the associated receiver becomes full. FIFO Full interrupts will be asserted on the 'C40's IIOF0 input. These interrupts are disabled for receivers whose associated bits in this field are cleared to the logic '0' state. The FIFO Full interrupt is often used to indicate that data has been lost in a real–time system.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.3 'C40 Interrupt Mask Register #2 – R/W @ 'C40_Int_Mask_2

This 16–bit register, which can be accessed by the 'C40 processor and by the VMEbus, contains two 8–bit fields. Each field contains one bit associated with each digital receiver. One field will enable or disable (i. e., mask) generation of 'C40 interrupts when the FIFO Half–Full flags go true. The other is used for FIFO Not Empty interrupts to the 'C40. Table 3–11, below, shows the bit arrangement within Interrupt Mask Register #2. The subsections beginning below the table give detailed descriptions of the register's bit fields.

Table 3-11: Model 6516 - 'C40 Interrupt Mask Register #2 - R/W @ C40_Int_Mask_2

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Field	FIFO Half-Full Flag Interrupts							
FIFO #	FIFO 8	FIFO 7	FIFO 6	FIFO 5	FIFO 4	FIFO 3	FIFO 2	FIFO 1
Bit Function	1 = Interrupt Enabled 0 = Interrupt Masked							
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Field	FIFO Not Empty Flag Interrupts							
FIFO #	FIFO 8	FIFO 7	FIFO 6	FIFO 5	FIFO 4	FIFO 3	FIFO 2	FIFO 1
Bit Function	1 = Interrupt Enabled 0 = Interrupt Masked							

3.5.3.1 FIFO Half Full Flag Interrupts – D15 > D8

Setting any of these eight bits to the logic '1' state will cause a 'C40 interrupt to be generated when the data FIFO for the associated receiver has reached the half–full point. FIFO Half–Full interrupts will be asserted on the 'C40's IIOF3 input. These interrupts are disabled for receivers whose associated bits in this field are cleared to the logic '0' state.

The FIFO Half–full condition is often used in real–time systems to trigger the FIFO readout routine. This prevents the data loss that can occur if one waits until the FIFO is full, and reduces the one–sample–at–a–time processing overhead associated with the use of the FIFO Not Empty condition.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.3 'C40 Interrupt Mask Register #2 (continued)

3.5.3.2 FIFO Not Empty Flag Interrupts – D7 > D0

Setting any of these eight bits to the logic '1' state will cause a 'C40 interrupt to be generated when the data FIFO for the associated receivers contains at least one sample. FIFO Empty interrupts will be asserted on the 'C40's IIOF2 input. These interrupts are disabled for receivers whose associated bits in this field are cleared to the logic '0' state. The FIFO Not Empty condition is usually used in systems where the data rate is low and the processor is relatively unburdened.

3.5.4 Receiver Status Register #1 – R. O. @ Rec_Stat_Reg_1

This 32–bit, read–only register, which can be accessed by the 'C40 processor and by the VMEbus, contains a bit associated with VMEbus interrupt status and a 2–bit field associated with the Front Panel Clock LEDs in its most significant byte field. The next two lower byte fields each use two bits to show the status of the Threshold Detect and Clock Not ready flags for each input channel. The least significant byte field uses all eight bits to give the Full Flag status for each receiver's FIFO. This register is generally read to determine the source of an interrupt to the 'C40, but the condition flags contained within it are valid even for masked interrupt conditions. Write operations to this register will have no effect. Table 3–12, at the top of the next page, shows the bit arrangement within Receiver Status Register #1. The subsections beginning below give detailed descriptions of the register's bit fields.

3.5.4.1 'C40 Interrupt to VMEbus Status Bit – D31

When this bit reads back on the logic '1' state, it indicates that a VMEbus interrupt issued by the 6516's 'C40 processor has not yet been acknowledged. It is set to the logic '1' (true) state when the 'C40 writes to address 0x8005 0000 in its Global Bus memory map (the VME_Int address, see Table 3–7 and Section 3.5.17). The bit is cleared to the logic '0' state during the VMEbus IACK cycle. A '0' in this bit means that there is no longer a pending VMEbus interrupt from the 'C40.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.4 Receiver Status Register #1 (continued)

Table 3-12: Model 6516 - Receiver Status Register #1 - R. O. @ Rec_Stat_Reg_1

Bit #	D31	D30	D29	D28	D27†	D26	D25†	D24
Bit Name	C40_to_VME _Int_Stat	Reserved			Input_D_Clk _LED_Status	Input_C_Clk _LED_Status	Input_B_Clk _LED_Status	Input_A_Clk _LED_Status
Bit Function	1=Int. Pending 0 = No Int.	Indeterminate			1 = LED Off 0 = LED On			
Bit #	D23*	D22	D21*	D20	D19*	D18	D17*	D16
Bit Field	Threshold Detector (& PIO*) Status							
I/P Channel	FPDP2-PIO2*	Th. Chan. D	FPDP1-PIO2*	Th. Chan. C	FPDP2-PIO1*	Th. Chan. B	FPDP1-PIO1*	Th. Chan. A
Bit Function	1 = False 0 = True							
Bit #	D15	D14†	D13	D12	D11	D10†	D9	D8
Bit Field	Clock Not Ready Flag Status							
I/P Channel	Reserved	Channel D	Reserved	Channel C	Reserved	Channel B	Reserved	Channel A
Bit Function	Indeterminate	1 = False 0 = True	Indeterminate	1 = False 0 = True	Indeterminate	1 = False 0 = True	Indeterminate	1 = False 0 = True
Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Field	FIFO Full Flag Status							
FIFO #	FIFO 8	FIFO 7	FIFO 6	FIFO 5	FIFO 4	FIFO 3	FIFO 2	FIFO 1
Bit Function	1 = FIFO Full Flag False 0 = FIFO Full Flag True							

* - In Option 008 only - In other units these bits are Reserved and their readout states are indeterminate.
† - In Option 008, these bits are Reserved and their readout states are indeterminate.
Also, the Channel A clock bits in these groups are for FPDP1, and the Channel C clock bits are for FPDP2.

3.5.4.2 Clock LED Status Bits – D27 > D24

This four–bit field contains one bit associated with each input channel. D27 is associated with Input Channel D, D26 with Chan–nel C, D25 with Channel B and D24 is associated with Channel A. The bit associated with a given channel will read back in the logic '1' state if the Clock LED for that input channel is OFF, i. e., if a clock signal has **not been detected** on that channel. These bits are not latched, and always indicate the current state of the associated LEDs. Under normal operating conditions, one would expect to read a '0' back in any bit associated with a used input channel, and a '1' in a bit associated with an unused input channel.

In the Model 6516 Option 008, D27 and D25 are Reserved, and their readout states are indeterminate. D26 is for the FPDP2 Clock LED, and D24 is for the FPDP1 Clock LED.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.4 Receiver Status Register #1 (continued)

3.5.4.3 Detector Flag Status Bits – D22, D20, D18 and D16

If any of these four bits reads back in the logic '0' state, the data on the associated input channel has exceeded the programmed threshold value at least the programmed number of times. A '1' in either of these bits indicates that the threshold detection conditions have not been met in associated channel. These bits are not latched, and always indicate the current state of the associated channel's threshold detector.

The threshold data value and count for a given input channel are determined by that channel's Threshold Detection Register. See Section 3.5.14 for information about these registers.

3.5.4.4 PIO Interrupt Status Bits – (D23, D21, D19 and D17) – Option 008 ONLY

If any of these four bits read back in the logic '0' state, then the interrupt conditions programmed in the PIO portion of the Mux_Mode_PIO register (see Section 3.5.12 for details) for the associated signal have been met. When one of these bits reads back in the logic '1' state, the interrupt conditions for the associated PIO signal have not been met.

NOTE: In units not equipped with Option 008, the data returned in these four bits is both indeterminate and meaningless.

3.5.4.5 Clock Not Ready Flag Status Bits – D14, D12, D10 & D8

If any of these four bits reads back in the logic '0' state, the input channel that drives the receiver associated with that bit has lost its clock signal. Under normal operating conditions, the bits in this field associated with those receivers driven by an active channel should read back in the logic '1' state. Those bits associated with receivers that have selected an unused channel as their input source will read back in the logic '0' state. These bits are not latched, and always indicate the current state of the associated receiver's clock signal.

In the Model 6516 Option 008, D14 and D10 are Reserved, and their readout states are indeterminate. D12 is the FPDP2 Clock Not Ready Status Bit, and D8 is for the FPDP1 clock.

NOTE: The data returned in the other four bits in this 8–bit field (D15, D13, D11 & D9) is both indeterminate and meaningless.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.4 Receiver Status Register #1 (continued)

3.5.4.6 FIFO Full Flag Status Bits – D7 > D0

If any of these 8 bits reads back in the logic '0' state, the data FIFO for the associated receiver has become full, indicating that data may have been lost. A logic '1' in any of these bits indicates that the associated receiver's FIFO is not yet full. These bits are not latched, and always indicate the current state of the associated receiver's FIFO.

3.5.5 Receiver Status Register #2 – R. O. @ Rec_Stat_Reg_2

This 16–bit, read–only register, which can be accessed by the 'C40 processor and by the VMEbus, contains two 8–bit fields. Each field contains a bit asso–ciated with a pair of digital receivers. One field gives the current status of the FIFO Full Flags, and the other gives the FIFO Empty Status. This register is generally read to determine the source of an interrupt to the 'C40, but the condition flags contained within it are valid even for masked interrupt condi–tions. Write operations to this register will have no effect. Table 3–13, below, shows the bit arrangement within Status Register #2. The subsections begin–ning below the table give detailed descriptions of the register's bit fields.

Table 3-13: Model 6516 - Receiver Status Register #2 - R. O. @ Rec_Stat_Reg_2

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Field	FIFO Half-Full Flag Status							
FIFO #	FIFO 8	FIFO 7	FIFO 6	FIFO 5	FIFO 4	FIFO 3	FIFO 2	FIFO 1
Bit Function	1 = FIFO Half-Full Flag False 0 = FIFO Half-Full Flag True							
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Field	FIFO Empty Flag Status							
FIFO #	FIFO 8	FIFO 7	FIFO 6	FIFO 5	FIFO 4	FIFO 3	FIFO 2	FIFO 1
Bit Function	1 = FIFO Empty Flag False 0 = FIFO Empty Flag True							

3.5.5.1 FIFO Half–Full Flag Status Bits – D15 > D8

If any of these 8 bits reads back in the logic '0' state, the data FIFO for the associated receiver pair has become half–full. A logic '1' in any of these bits indicates that the associated receiver's FIFO is not yet half–full. These bits are not latched, and always indicate the current state of the associated receiver's FIFO. If polling is being used as an alternative to interrupts to determine when to read data from the 6516's FIFO's in a real–time system, these are the flags that we suggest you poll.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.5 Receiver Status Register #2 (continued)

3.5.5.2 FIFO Empty Flag Status Bits – D7 > D0

If any of these 8 bits reads back in the logic '0' state, the data FIFO for the associated receiver pair has become empty. A logic '1' in any of these bits indicates that the associated receiver's FIFO contains data. These bits are not latched, and always indicate the current state of the associated receiver's FIFO.

3.5.6 FIFO Reset Register – R/W @ F_Rst_Reg

This 16–bit register, which can be accessed by the 'C40 processor and by the VMEbus, contains one bit to reset the data FIFO associated with each of the 6516's digital receivers. A ninth bit is used to initialize the 'C40 Comm Ports after the board is reset. Table 3–14, below, shows which of these bits is associated with which receiver's FIFO.

Table 3-14: Model 6516 - FIFO Reset Register - R/W @ F_Rst_Reg

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Name	Not Used							Comm_Port_Init
Bit Function	Not Applicable							1 = Run 0 = Initialize
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Name	FIFO_8_Reset	FIFO_7_Reset	FIFO_6_Reset	FIFO_5_Reset	FIFO_4_Reset	FIFO_3_Reset	FIFO_2_Reset	FIFO_1_Reset
Bit Function	1 = Release & Run 0 = Reset FIFO							

NOTE: When reading the contents of the FIFOs via the Comm Port interface, the ports are read in FIFO pairs. The Odd FIFO reset bit also resets the Comm Port Control circuit. Because of this, if you want to use the Comm Port interface to read all the data from one FIFO in the pair before reading the other FIFO's data, the lower–numbered FIFO in the pair **MUST** be read first.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.6 FIFO Reset Register (continued)

3.5.6.1 FIFO Reset Bits – D7 > D0

FIFOs on the Model 6516 are held in the reset state for as long as the associated Reset bit remains in the logic '0' state. The reset function should therefore be written as a toggle (i. e., clear the bit of interest to the logic '0' state, then set it back to logic '1'). When a FIFO is reset, any data it may contain is erased, the Full and Half–full flags become false, and the Empty flag goes true. Note that these bits also reset the circuitry that controls FIFO write opera–tions, which guarantees that the correct I/Q ordering within the FIFO will be maintained.

3.5.6.2 Comm Port Initialization Bit – D8

This bit is used to reset the 'C40 Comm Ports, which are brought, through buffers, to the Model 6516's Front Panel. The Comm Ports are held in the reset state for as long as this bit remains in the logic '0' state. When this bit is set to the logic '1' state, the ports are released and Comm Ports 2, 3, 4 and 5 are configured as inputs. The function that implements this reset should be written as a toggle (i. e., this bit should be cleared to the logic '0' state, then set to the logic '1' state). The firmware for this device toggles this bit as a part of the reset and boot routines. However, this initialization can only be performed once for each 'C40 reset.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register – R/W @ Out_Comm_Ctl

Each of the four Output Comm Port connectors on the 6516's Front Panel provides the data from one or two pairs of digital receivers. For example, depending on the associated FIFO Mode bits in the input control register, you can enable digital receivers 1 & 3, or digital receivers 1 & 2 and 3 & 4. This 16–bit register, which can be accessed by the 'C40 processor and by the VMEbus, contains four bit fields, each of which is associated with one, or two, receiver pair's Output Comm Port. One bit in each field enables the data packing feature (two data samples contained in each 32–bit word). Another selects the order in which the samples are packed. A third selects which FIFO data is carried by the port (if packed mode is not enabled or if Time ordering is selected), and the last bit enables the pair's Comm Port. Table 3–15, below, shows the bit arrangement within the Output Comm Port Control Register. The subsections beginning at the top of the next page give detailed descriptions of the register's bit fields.

NOTE: The settings of this register affect the presentation of data at the Model 6516's Front Panel Output Comm Ports **ONLY**. Except for the Port Enable bits (which disable direct FIFO readout), they do not affect data read from the FIFOs, by the on–board 'C40 or over the VMEbus, which can **only** be read in Non–Packed Mode. Accordingly, since the VMEbus is the only output path for data on the Model 6516 Option 018, the only bits of concern in this register on devices so equipped are the Port_Enable bits (D12, D8, D4 and D0) which must always be set to the logic '1' state.

Table 3-15: Model 6516 - Output Comm Port Control Register - R/W @ Out_Comm_Ctl

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Comm Port	Receivers 13 & 15 (Single Mode), or 13, 14, 15 & 16 (Dual Mode) - FIFOs 7 & 8				Receivers 9 & 11 (Single Mode), or 9, 10, 11, & 12 (Dual Mode) - FIFOs 5 & 6			
Bit Name	Pack_ Mode	Data_ Order*	FIFO _Select	Port_ Enable	Pack_ Mode	Data_ Order*	FIFO_ Select	Port_ Enable
Bit Function	1 = Packed 0=Non-Packed	1 = I/Q (Time) 0 = Receiver	1 = Odd FIFO 0 = Even FIFO	1 = Disabled 0 = Enabled	1 = Packed 0=Non-Packed	1 = I/Q (Time) 0 = Receiver	1 = Odd FIFO 0 = Even FIFO	1 = Disabled 0 = Enabled
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Port	Receivers 5 & 7 (Single Mode), or 5, 6, 7, & 8 (Dual Mode) - FIFOs 3 & 4				Receivers 1 & 3 (Single Mode), or 1, 2, 3 & 4 (Dual Mode) - FIFOs 1 & 2			
Bit Name	Pack_ Mode	Data_ Order*	FIFO _Select	Port_ Enable	Pack_ Mode	Data_ Order*	FIFO _Select	Port_ Enable
Bit Function	1 = Packed 0=Non-Packed	1 = I/Q (Time) 0 = Receiver	1 = Odd FIFO 0 = Even FIFO	1 = Disabled 0 = Enabled	1 = Packed 0=Non-Packed	1 = I/Q (Time) 0 = Receiver	1 = Odd FIFO 0 = Even FIFO	1 = Disabled 0 = Enabled

NOTE: The Single/Dual Mode select bits (FIFO_MODE) are in the Input Control Register. Please refer to Table 3-19 and Section 3.5.8.3 for programming details.

* - When a Data_Order bit is set to the logic '1' state, I/Q ordering is selected if the associated receivers are programmed for C Complex operation, and Time ordering is selected if they are programmed for Real-only operation.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register (continued)

3.5.7.1 Pack Mode Bits – D15, D11, D7 & D3

In Packed Mode, each 32–bit word delivered by a Comm Port contains two 16–bit samples. In Non–Packed Mode, only one 16–bit sample is contained in each 32–bit Comm Port word, justified in the upper 16 bits. Packed mode is enabled for a given Comm Port group by setting the Pack_Mode bit to the logic '1' state. Clearing the Pack_Mode bit to the logic '0' state will cause Non–Packed samples to be delivered. When data is Non–Packed, the lower 16 bits of the Comm Port longword will be indeterminate. Table 3–17, on the next page, shows examples of Packed and Non–Packed data formats. **In order to read data from all receivers associated with a given Comm Port, Packed Mode MUST be enabled, AND the FIFO_Mode bit (**in the Input Control Register, see Section 3.5.8.3**) for that port must be set.** Table 3–16, below shows the two possible receiver combinations for the Comm Ports.

Table 3-16: Model 6516 - Enabled Receiver Groups for Packed Mode

Comm Port	FIFO Mode Single	FIFO Mode Dual
1 - 4	1, 3	1, 2, 3, 4
5 - 8	5, 7	5, 6, 7, 8
9 - 12	9, 11	9, 10, 11, 12
13 - 16	13, 15	13, 14, 15, 16

NOTE: The Single/Dual Mode select bits (FIFO_MODE) are in the Input Control Register. Please refer to Table 3-19 and Section 3.5.8.3 for programming details.

NOTE: For Option 018, these four bits are unused.

3.5.7.2 Data Ordering Bits – D14, D10, D6 & D2

When Packed Mode is enabled, the data in a given longword may be ordered by Receiver, by I/Q status (for complex data) or by time (for real–only data). When the packed data is ordered by Receiver, for example, a 16–bit sample from receiver 3 or receivers 3 & 4 occupy bits D31 – D16 in the longword, and a 16–bit sample from receiver 1 or receivers 1 & 2 occupy bits D15 – D0. Packed data will be ordered by Receiver over a given Comm Port when the Data_Order bit associated with that port is cleared to the logic '0' state.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register (continued)

3.5.7.2 Data Ordering Bits (continued)

When the Data_Order bit associated with a given port is set to the logic '1' state, and that port's selected receivers are programmed for complex operation (by clearing the associated Receiver's Mode bit in the Multiplex and Mode Register to the logic '0' state (see Section 3.5.10), and appropriately programming the HSP50016 chips (see Appendix B) I/Q ordering is selected for that port. When I/Q ordering is enabled, only the selected FIFO sends data to the Comm Port (see FIFO Select Bits, Section 3.5.7.3). When data is ordered by I/Q status, the 16–bit In–Phase (I) sample is located in bits D31 – D16 and the 16–bit Quadrature (Q) sample is contained in bits D15 – D0 of the Comm Port's longword.

If a given receiver is programmed for the real–only operating mode (again, see Section 3.5.10 and Appendix B), and the Data_Order bit associated with that receiver's port is set to the logic '1' state, Time ordering is selected for that port. Time ordering is used in combination with Packed mode (see Section 3.5.7.1, on page 58), to allow each longword delivered by the Comm Port to carry two real samples from the same FIFO (see FIFO Select Bits, Section 3.5.7.3), or sample #1 from a receiver pair, depending on the FIFO Mode bit from the Input Control Register. In Packed, Time Ordered, Real mode, the earlier sample will be placed in the lower 16 bits of the Comm Port longword (D15 – D0), and the later sample will be in the upper word (D16 – D31) when only one receiver of the pair is enabled.

Table 3–18, on the page after next, shows examples of Receiver Ordered data in Packed and Non–Packed modes, and of I/Q and Time ordered data in Packed mode.

NOTES: (1) **The only format in which data from both receivers 1 & 3 (in single FIFO mode) or receivers 1, 2, 3, and 4 (in Dual FIFO mode) in a given group can be delivered over the Comm Port interface is the Packed, Receiver Ordered format.**

(2) When I/Q or Time Ordering is enabled, Packed Mode MUST also be enabled.

(3) For Option 018, these four bits are unused.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register (continued)

3.5.7.2 Data Ordering Bits (continued)

Table 3-17: Model 6516 - Comm Port Data Packing and Ordering Formats (Tagging Disabled): Example - FIFO_Mode Bit for FIFOs 1 & 2 Set for Dual (1)

Packed, Receiver Ordered Mode, Complex Data

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	#3 Receiver, Q Sample 1	(LSB)	(MSB)	#1 Receiver, Q Sample 1	(LSB)
2	(MSB)	#3 Receiver, I Sample 1	(LSB)	(MSB)	#1 Receiver, I Sample 1	(LSB)
3	(MSB)	#4 Receiver, Q Sample 1	(LSB)	(MSB)	#2 Receiver, Q Sample 1	(LSB)
4	(MSB)	#4 Receiver, I Sample 1	(LSB)	(MSB)	#2 Receiver, I Sample 1	(LSB)
5	(MSB)	#3 Receiver, Q Sample 2	(LSB)	(MSB)	#1 Receiver, Q Sample 2	(LSB)
6	(MSB)	#3 Receiver, I Sample 2	(LSB)	(MSB)	#1 Receiver, I Sample 2	(LSB)
7	(MSB)	#4 Receiver, Q Sample 2	(LSB)	(MSB)	#2 Receiver, Q Sample 2	(LSB)
8	(MSB)	#4 Receiver, I Sample 2	(LSB)	(MSB)	#2 Receiver, I Sample 2	(LSB)

Packed, Receiver Ordered Mode, Real Data Only

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	#3 Receiver, Sample 1	(LSB)	(MSB)	#1 Receiver, Sample 1	(LSB)
2	(MSB)	#4 Receiver, Sample 1	(LSB)	(MSB)	#2 Receiver, Sample 1	(LSB)
3	(MSB)	#3 Receiver, Sample 2	(LSB)	(MSB)	#1 Receiver, Sample 2	(LSB)
4	(MSB)	#4 Receiver, Sample 2	(LSB)	(MSB)	#2 Receiver, Sample 2	(LSB)

Non-Packed, Receiver Ordered Mode, Complex Data

FIFO Select = Odd *(Even)*	Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
	1	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(LSB)		I n d e t e r m i n a t e	
	2	(MSB)	#1 *(#3)* Receiver, I Sample 1	(LSB)		I n d e t e r m i n a t e	
	3	(MSB)	#2 *(#4)* Receiver, Q Sample 1	(LSB)		I n d e t e r m i n a t e	
	4	(MSB)	#2 *(#4)* Receiver, I Sample 1	(LSB)		I n d e t e r m i n a t e	
	5	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(LSB)		I n d e t e r m i n a t e	
	6	(MSB)	#1 *(#3)* Receiver, I Sample 2	(LSB)		I n d e t e r m i n a t e	
	7	(MSB)	#2 *(#4)* Receiver, Q Sample 2	(LSB)		I n d e t e r m i n a t e	
	8	(MSB)	#2 *(#4)* Receiver, I Sample 2	(LSB)		I n d e t e r m i n a t e	

Non-Packed, Receiver Ordered Mode, Real Data Only

FIFO Select = Odd *(Even)*	Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
	1	(MSB)	#1 *(#3)* Receiver, Sample 1	(LSB)		I n d e t e r m i n a t e	
	2	(MSB)	#2 *(#4)* Receiver, Sample 1	(LSB)		I n d e t e r m i n a t e	
	3	(MSB)	#1 *(#3)* Receiver, Sample 2	(LSB)		I n d e t e r m i n a t e	
	4	(MSB)	#2 *(#4)* Receiver, Sample 2	(LSB)		I n d e t e r m i n a t e	

Packed, I/Q Ordered Mode, Complex Data

FIFO Select = Odd *(Even)*	Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
	1	(MSB)	#1 *(#3)* Receiver, I Sample 1	(LSB)	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(LSB)
	2	(MSB)	#2 *(#4)* Receiver, I Sample 1	(LSB)	(MSB)	#2 *(#4)* Receiver, Q Sample 1	(LSB)
	3	(MSB)	#1 *(#3)* Receiver, I Sample 2	(LSB)	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(LSB)
	4	(MSB)	#2 *(#4)* Receiver, I Sample 2	(LSB)	(MSB)	#2 *(#4)* Receiver, Q Sample 2	(LSB)

Packed, Time Ordered Mode, Real Data Only

FIFO Select = Odd *(Even)*	Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
	1	(MSB)	#2 *(#4)* Receiver, Sample 1	(LSB)	(MSB)	#1 *(#3)* Receiver, Sample 1	(LSB)
	2	(MSB)	#2 *(#4)* Receiver, Sample 2	(LSB)	(MSB)	#1 *(#3)* Receiver, Sample 2	(LSB)
	3	(MSB)	#2 *(#4)* Receiver, Sample 3	(LSB)	(MSB)	#1 *(#3)* Receiver, Sample 3	(LSB)
	4	(MSB)	#2 *(#4)* Receiver, Sample 4	(LSB)	(MSB)	#1 *(#3)* Receiver, Sample 4	(LSB)

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register (continued)

3.5.7.2 Data Ordering Bits (continued)

Table 3-18: Model 6516 - Comm Port Data Packing and Ordering Formats (Tagging Disabled): Example - FIFO_Mode Bit for FIFOs 1 & 2 Set for Single (0)

Packed, Receiver Ordered Mode, Complex Data

Sample #	D31	D16	D15	D0
1	(MSB) #3 Receiver, Q Sample 1 (LSB)		(MSB) #1 Receiver, Q Sample 1 (LSB)	
2	(MSB) #3 Receiver, I Sample 1 (LSB)		(MSB) #1 Receiver, I Sample 1 (LSB)	
3	(MSB) #3 Receiver, Q Sample 2 (LSB)		(MSB) #1 Receiver, Q Sample 2 (LSB)	
4	(MSB) #3 Receiver, I Sample 2 (LSB)		(MSB) #1 Receiver, I Sample 2 (LSB)	

Packed, Receiver Ordered Mode, Real Data Only

Sample #	D31	D16	D15	D0
1	(MSB) #3 Receiver, Sample 1 (LSB)		(MSB) #1 Receiver, Sample 1 (LSB)	
2	(MSB) #3 Receiver, Sample 2 (LSB)		(MSB) #1 Receiver, Sample 2 (LSB)	
3	(MSB) #3 Receiver, Sample 3 (LSB)		(MSB) #1 Receiver, Sample 3 (LSB)	
4	(MSB) #3 Receiver, Sample 4 (LSB)		(MSB) #1 Receiver, Sample 4 (LSB)	

Non-Packed, Receiver Ordered Mode, Complex Data

FIFO Select = Odd *(Even)*	Sample #	D31	D16	D15	D0
	1	(MSB) #1 *(#3)* Receiver, Q Sample 1 (LSB)		Indeterminate	
	2	(MSB) #1 *(#3)* Receiver, I Sample 1 (LSB)		Indeterminate	
	3	(MSB) #1 *(#3)* Receiver, Q Sample 2 (LSB)		Indeterminate	
	4	(MSB) #1 *(#3)* Receiver, I Sample 2 (LSB)		Indeterminate	

Non-Packed, Receiver Ordered Mode, Real Data Only

FIFO Select = Odd *(Even)*	Sample #	D31	D16	D15	D0
	1	(MSB) #1 *(#3)* Receiver, Sample 1 (LSB)		Indeterminate	
	2	(MSB) #1 *(#3)* Receiver, Sample 2 (LSB)		Indeterminate	
	3	(MSB) #1 *(#3)* Receiver, Sample 3 (LSB)		Indeterminate	
	4	(MSB) #1 *(#3)* Receiver, Sample 4 (LSB)		Indeterminate	

Packed, I/Q Ordered Mode, Complex Data

FIFO Select = Odd *(Even)*	Sample #	D31	D16	D15	D0
	1	(MSB) #1 *(#3)* Receiver, I Sample 1 (LSB)		(MSB) #1 *(#3)* Receiver, Q Sample 1 (LSB)	
	2	(MSB) #1 *(#3)* Receiver, I Sample 2 (LSB)		(MSB) #1 *(#3)* Receiver, Q Sample 2 (LSB)	
	3	(MSB) #1 *(#3)* Receiver, I Sample 3 (LSB)		(MSB) #1 *(#3)* Receiver, Q Sample 3 (LSB)	
	4	(MSB) #1 *(#3)* Receiver, I Sample 4 (LSB)		(MSB) #1 *(#3)* Receiver, Q Sample 4 (LSB)	

Packed, Time Ordered Mode, Real Data Only

FIFO Select = Odd *(Even)*	Sample #	D31	D16	D15	D0
	1	(MSB) #1 *(#3)* Receiver, Sample 2 (LSB)		(MSB) #1 *(#3)* Receiver, Sample 1 (LSB)	
	2	(MSB) #1 *(#3)* Receiver, Sample 4 (LSB)		(MSB) #1 *(#3)* Receiver, Sample 3 (LSB)	
	3	(MSB) #1 *(#3)* Receiver, Sample 6 (LSB)		(MSB) #1 *(#3)* Receiver, Sample 5 (LSB)	
	4	(MSB) #1 *(#3)* Receiver, Sample 8 (LSB)		(MSB) #1 *(#3)* Receiver, Sample 7 (LSB)	

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.7 Output Comm Port Control Register (continued)

3.5.7.3 FIFO Select Bits – D13, D9, D5 & D1

When the other bits in these bit fields are configured so the associated Comm Port delivers data from only one FIFO in the associated FIFO pair (i. e., if the data is either to be Non–Packed or Packed and not Receiver ordered), these bits select which FIFO outputs data. When one of these bits is set to the logic '1' state, the odd (lower) numbered FIFO's data is delivered to the Comm Port interface. When one of these bits is cleared to the logic '0' state, the even (higher) numbered FIFO is the FIFO whose data is output.

NOTE: For Option 018, these four bits are unused.

3.5.7.4 Comm Port Enable Bits – D12, D8, D4 & D0

These bits are used to enable or disable their associated Comm Port interfaces. When one of these bits is held in the logic '1' state, the Port is disabled, and the FIFO may be read by the 'C40 or over the VMEbus. To enable the Comm Port interface for a given receiver pair, clear the associated bit to the logic '0' state.

NOTES: (1) When Comm Port readout of a given receiver's FIFO is enabled, direct readout of that FIFO (by either the 'C40 or VME) is disabled. To enable direct FIFO readout, the Comm Port must be disabled.

(2) For Option 018, these four bits must always be set to the logic '1' state, to enable readout over the VMEbus.

3.5.8 Input Control Register – R/W @ Input_Ctl

This 32–bit register, which can be accessed by the 'C40 processor and by the VMEbus. Three bits associated with Sync (or Reset) functions are provided for each input channel (12 total); to select the source of the Sync signal, to act as the Sync signal, and allow the Sync line on the input connector to function as an output. Four bits are used to enable to the FIFOs for the even numbered receivers. For each input channel, bits are provided to override the driver for the **OVLD** LED on the Front Panel. Two bits control the input crossover function, and two more bits are provided that select the logic family for the FPDP Strobe signal for Option 008 units. Table 3–19, at the top of the next page, shows the bit arrangement within the Input Control Register. The subsections beginning below the table give detailed descriptions of each of the register's bits.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

Table 3-19: Model 6516 - Input Control Register - R/W @ Input_Ctl

Bit #	D31	D30	D29	D28†	D27	D26	D25*	D24†
Bit Name	User_OVLD_LED_D	FIFO 7 & 8 Mode	Reserved	Input_Crossover_1	Input Channel D: Sync_Out_Enable	Input Channel D: Sync_In_Source	Input Channel D: Software_Sync	FPDP2_Clk_Select
Bit Function	1 = ON 0 = OFF	1 = Dual 0 = Single	Not Applicable	See Table 3-20	1 = Enable 0 = Disabled	1 = Software 0 = Front Panel	1 = Run 0 = Reset	1 = PECL 0 = TTL
Bit #	**D23**	**D22**	**D21**	**D20†**	**D19**	**D18**	**D17***	**D16**
Bit Name	User_OVLD_LED_C	FIFO 5 & 6 Mode	Reserved	Input_Crossover_0	Input Channel C: Sync_Out_Enable	Input Channel C: Sync_In_Source	Input Channel C: Software_Sync	Reserved
Bit Function	1 = ON 0 = OFF	1 = Dual 0 = Single	Not Applicable	See Table 3-20	1 = Enabled 0 = Disabled	1 = Software 0 = Front Panel	1 = Run 0 = Reset	Not Applicable
Bit #	**D15**	**D14**	**D13**	**D12**	**D11**	**D10**	**D9***	**D8†**
Bit Name	User_OVLD_LED_B	FIFO 3 & 4 Mode	Reserved		Input Channel B: Sync_Out_Enable	Input Channel B: Sync_In_Source	Input Channel B: Software_Sync	FPDP1_Clk_Select
Bit Function	1 = ON 0 = OFF	1 = Dual 0 = Single	Not Applicable		1 = Enabled 0 = Disabled	1 = Software 0 = Front Panel	1 = Run 0 = Reset	1 = PECL 0 = TTL
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1***	**D0**
Bit Name	User_OVLD_LED_A	FIFO 1 & 2 Mode	Reserved		Input Channel A: Sync_Out_Enable	Input Channel A: Sync_In_Source	Input Channel A: Software_Sync	Reserved
Bit Function	1 = ON 0 = OFF	1 = Dual 0 = Single	Not Applicable		1 = Enabled 0 = Disabled	1 = Software 0 = Front Panel	1 = Run 0 = Reset	Not Applicable

NOTE: This register can only be properly accessed in D32 mode.
Dual-cycle D16 write operations to this register result in the upper word not being stored.

* - The D25, D17, D9 and D1 bits are only active if the Sync_In_Source bit for the associated channel selects the Software path (i. e., if D26, D18, D10 and/or D2 are set to the logic '1' state)

† - In units equipped with Option 008 and/or Option 018 **ONLY**!
In all other units, these bits are Reserved & Unused.

NOTE: This register can only be properly accessed in D32 mode. Dual–cycle D16 write operations to this register will result in the upper word not bring stored.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

3.5.8.1 Input Crossover Bits – D28 & D20

In units NOT equipped with Option 008 and/or Option 018, bit D28 controls the crossover function for Channel D, and D20 for Channel C. This function allows the associated receivers to select between a different pair of input channels. These bits must be set to the logic '1' state in order to direct the A and/or B inputs to Receivers 9 – 16. When these bits are cleared to the logic '0' state, Receivers 9 – 16 can select either Channel C or D as their input sources.

In units equipped with Options 008 and/or 018, bits D20 and D28 are used in combination to control the crossover function for both FPDP1 and FPDP2. When both bits are cleared to the logic '0' state, the FPDP1 input (Channels A & B) is routed to Receivers 1 – 8, and the FPDP2 input (Channels C & D) input is routed to Receivers 9 – 16. If D28 is cleared to the logic '0' state and D20 is set to the logic '1' state, then all 16 receivers can select between the two input channels carried by FPDP1 (Channels A & B). If D28 is set to the logic '1' state and D20 is cleared to the logic '0' state, then all 16 receivers can select between the two input channels carried by FPDP2 (Channels C & D). Setting both of these bits simultaneously is an illegal state, and will cause no receivers to be driven.

Selection of one input channel or the other (in the pair selected by these bits) for a given pair of receivers is made by bits D15 – D8 in the Multiplex and Mode Register (see Section 3.5.10.1).

Table 3–20, below, summarizes the settings of the Input Crossover bits in the 6516's Input Control Register.

Table 3-20: Model 6516 - Input Crossover Bits

Bit Settings		Selectable Input Channels for Receivers	
D28	**D20**	**Std. or Opt. 009**	**Opt. 008 or Opt. 018**
0	**0**	A or B -> 1 thru 8 C or D -> 9 thru 16	A or B -> 1 thru 8 C or D -> 9 thru 16
0	**1**	A or B -> 1 thru 16 D -> 9 thru 16	A or B -> 1 thru 16
1	**0**	A or B -> 1 thru 16 C -> 9 thru 16	C or D -> 1 thru 16
1	**1**	A or B -> 1 thru 16	RESERVED - DO NOT USE!

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

3.5.8.2 User (OVLD) LED Bits – D31, D23, D15 & D7

These bits control the state of the Front Panel **OVLD** (red) LED for each input channel. Setting one of these bits to the logic '1' state turns the **OVLD** LED for the associated channel ON. Clearing the bit to the logic '0' state turns the associated LED OFF. D31 controls Channel D's LED, D23 control's Channel C's, D 15 is used for Channel B and D7 for Channel A. These LEDs were initially intended as indicators for the threshold detector function (which explains the front panel designation), but are now completely under software control. The firmware, however, does use these LEDs as threshold detect indicators.

3.5.8.3 FIFO Mode Bits – D30, D22, D14, and D6

The FIFO Mode Bits determine whether the associated FIFO receives data from one receiver or two. Single mode enables only the odd numbered receivers, and Dual mode enables both even and odd numbered receiverss. You cannot enable only the even receiverss. Each Comm Port outputs data from two FIFOs, so in Single FIFO mode, a Comm Port will carry data from two reveiv–ers, and in Dual FIFO mode, data from four receivers will be avail–able on a Port. , Table 3–21, below, shows which receivers are enabled in each mode.

Table 3-21: Model 6516 - FIFO Mode Bits

Control Bit	D6		D14		D22		D30		
FIFO Mode	FIFO 1	FIFO 2	FIFO 3	FIFO 4	FIFO 5	FIFO 6	FIFO 7	FIFO 8	
Single Mode (0)	1	3	5	7	9	11	13	15	Receivers Enabled
Dual Mode (1)	1 & 2	3 & 4	5 & 6	7 & 8	9 & 10	11 & 12	13 & 14	15 & 16	

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

3.5.8.4 Sync Output Enable Bits – D27, D19, D11 & D3

When one of these bits is set to the logic '1' state, the Sync line on the corresponding channel's input connector functions as an output. Clear this bit to the logic '0' state in the associated channel to disable this function. These bits, in tandem with the Sync Input Source Bits (see Section 3.5.8.5, below) allow multiple channels to be synchronized simultaneously in a Master/Slave scenario. D27 makes this selection for the Channel D connector, D19 for Channel C, D11 for Channel B and D3 for Channel A.

When other digital receiver boards are used as slaves, you should program the "master" channel's source bit for internal (software) sync, and enable the sync output for that channel. The "slave" channels are programmed for external (hardware) sync and for the Sync line as an input. The Sync lines for all channels used in this manner must be externally connected. When the "master" channel's Software Sync Bit (see Section 3.5.8.6) is toggled, all the connected "slave" channels are synchronized to the "master". When using a single receiver board, the sync lines do not have to be externally connected to synchronize the channels. Figure 3–1, on the next page, shows how the signals are synchronized in the Model 6516. Note that, for the sake of simplicity, the additional input crossover features available only in Option 008 have been excluded from this figure.

3.5.8.5 Sync Input Source Bits – D26, D18, D10, and D2

These bits determine whether the Sync signal for the associated channel comes from an external, hardware source (via the Front Panel Input Connector) or are generated internally, in software. To select the Software (internal) path for the Sync signal in a given channel, set the associated bit to the logic '1' state. When the Software path is selected, the Software Sync bit (see Section 3.5.8.6, below) for a given channel is toggled to sync all receivers using this channel as an input. Clear the associated bit to the logic '0' state to select the Front Panel (external hardware) sync path for a given channel's receivers. Bit D26 selects the Sync Source for receivers driven by Channel D, D18 makes the selection for Channel C's receivers, D10 for Channel B and D2 for Channel A.

In Master/Slave synchronization situations, this bit should be set to the logic '1' state in the master channel (Sync Source = Internal), and cleared to the logic '0' state in the slave channels (Sync Source = External).

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

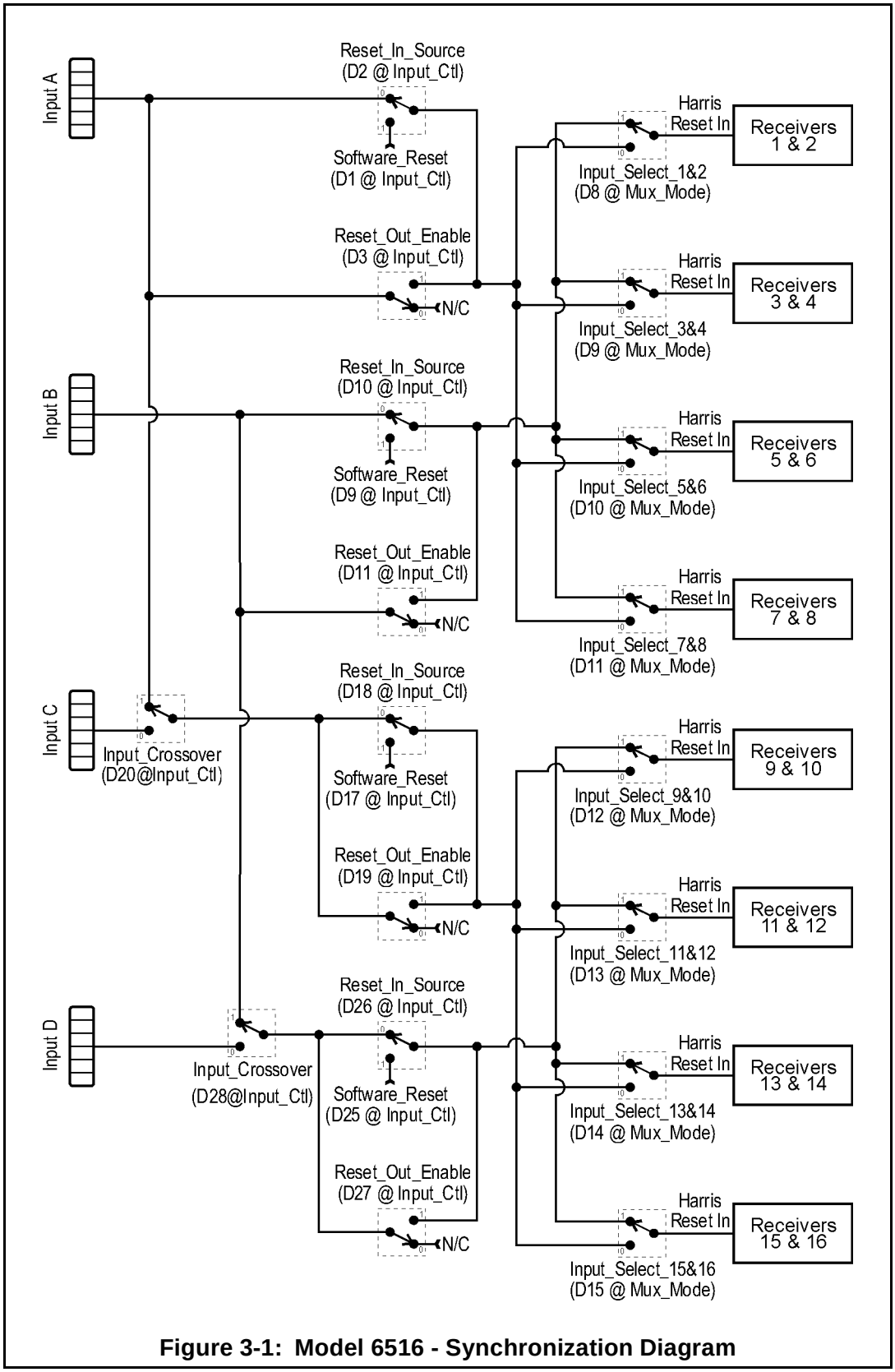


Figure 3-1: Model 6516 - Synchronization Diagram

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.8 Input Control Register (continued)

3.5.8.6 Software Sync Bits – D25, D17, D9, and D1

These bits are active only if the Sync Input Source bit for the associated channel (D26, D18, D10 and/or D2, see Section 3.5.8.5, above), is set to the logic '1' state to select the software sync path. When that condition is true, and one of these bits is cleared to the logic '0' state, all receivers driven by the associated input channel are held in reset. To allow the receivers driven by a given input channel to perform their conversions when the software sync path is selected, set the Sync bit associated with that channel to the logic '1' state. D25 is the soft sync bit for receivers driven by Channel D, D17 provides the soft sync for Channel C's receivers, D19 for Channel B and D1 for Channel A.

If the Sync Input Source bit for a given channel selects the hardware sync path (i. e., is cleared to the logic '0' state), the state of the Software Sync bit for that channel is trivial.

3.5.8.7 FPDP Clock Select – D24 (FPDP2) & D8 (FPDP1) – Option 008 Only

In units equipped with Option 008, bits D24 and D8 in the Input Control Register are used to select which of the two available Strobe signals will become the Clock for the digital receivers. D24 selects the clock for the channels driven by FPDP2, and D8 makes the selection for the channels driven by FPDP2. When one of these bits is set to the logic '1' state, the differential PECL PSTROBE, $\overline{\text{PSTROBE}}$ signal pair is selected. Clearing one of these bits to the logic '0' state selects the TTL–level STROB signal as the clock for the associated channels. In non–Option 008 units, these bits are Reserved.

3.5.9 Receiver Control Registers – R/W @ Rcvr_Ctl_1 & Rcvr_Ctl_2

The HSP50016 digital receiver chips on the Model 6516 are controlled by writing data to the eight 40–bit control registers contained within each chip. These registers are written by serially shifting the data into the part, via a four–pin control interface. Appendix B of this manual is the Harris HSP50016 Manual. The Control Word Input Section provides complete details about programming the Harris chips' internal control registers.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.9 Receiver Control Registers (continued)

Two 32–bit registers, which can both be accessed by the 'C40 processor and by the VMEbus, each provide eight 4–bit fields. Each field drives the control interface for one receiver chip. Receiver Control Register #1 controls the interface for the odd–n umbered chips, and receiver Control register #2 is for the even numbered chips. Table 3–22, below, tells which field is associated with which receiver, and which bit drives which input in the control interface for Control Register #1. Table 3–23, at the top of the next page, gives these associations for the even numbered receivers in Control Register #2. Figure 3–2, also on the next page, shows the timing relationships at the control interface pins. The subsections that begin below the figure provide brief descriptions of the bits.

NOTE: These registers can only be properly accessed in D32 mode. Dual–cycle D16 write operations to these registers will result in the upper word not bring stored.

Table 3-22: Model 6516 - Receiver Control Register #1 - R/W @ Rcvr_Ctl_1

Receiver	Receiver 15				Receiver 13			
Bit #	D31	D30	D29	D28	D27	D26	D25	D24
Receiver	Receiver 11				Receiver 9			
Bit #	D23	D22	D21	D20	D19	D18	D17	D16
Receiver	Receiver 7				Receiver 5			
Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Receiver	Receiver 3				Receiver 1			
Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Name	Setup _Data	Data_ Clock	Data_ Strobe	Chip_ Select	Setup _Data	Data_ Clock	Data_ Strobe	Chip_ Select
Bit Function	Latched on Neg Clk edge	Negative edge active	Pulse High to enable	1 = Not Selected 0 = Selected	Latched on Neg Clk edge	Negative edge active	Pulse High to enable	1 = Not Selected 0 = Selected

NOTE: This register can only be properly accessed in D32 mode.
Dual-cycle D16 write operations to this register result in the upper word not being stored.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.9 Receiver Control Registers (continued)

Table 3-23: Model 6516 - Receiver Control Register #2 - R/W @ Rcvr_Ctl_2

Receiver	Receiver 16				Receiver 14			
Bit #	D31	D30	D29	D28	D27	D26	D25	D24
Receiver	Receiver 12				Receiver 10			
Bit #	D23	D22	D21	D20	D19	D18	D17	D16
Receiver	Receiver 8				Receiver 6			
Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Receiver	Receiver 4				Receiver 2			
Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Name	Setup _Data	Data_ Clock	Data_ Strobe	Chip_ Select	Setup _Data	Data_ Clock	Data_ Strobe	Chip_ Select
Bit Function	Latched on Neg Clk edge	Negative edge active	Pulse High to enable	1 = Not Selected 0 = Selected	Latched on Neg Clk edge	Negative edge active	Pulse High to enable	1 = Not Selected 0 = Selected

NOTE: This register can only be properly accessed in D32 mode.
Dual-cycle D16 write operations to this register result in the upper word not being stored.

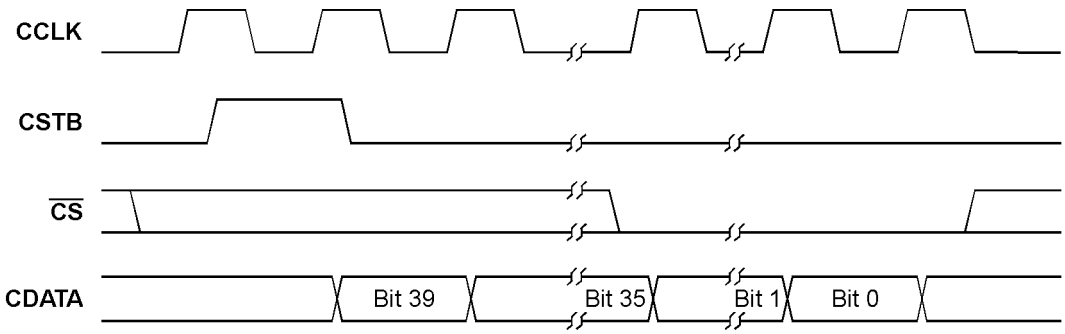


Figure 3-2: HSP50016 - Control Interface Timing Relationships

3.5.9.1 Receiver Setup Data Bits – D31 (R15), D27 (R13), D23 (R11), D19 (R9), D15 (R7), D11 (R5), D7 (R3) & D3 (R1) @ Rcvr_Ctl_1, and D31 (R16), D27 (R14), D23 (R12), D19 (R10), D15 (R8), D11 (R6), D7 (R4), & D3 (R2) @ Rcvr_Ctl_2

This bit drives the associated Harris chip's **CDATA** input pin. Data is shifted into the Harris Control Registers, MSB (i. e., bit 39) first, on the high–to–low transition of the clock signal (see Section 3.5.9.2, at the top of the next page).

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.9 Receiver Control Registers (continued)

3.5.9.2 Setup Data Clock Bits – D30 (R15), D26 (R13), D22 (R11), D18 (R9), D14 (R7), D10 (R5), D6 (R3) & D2 (R1) @ Rcvr_Ctl_1, and D30 (R16), D26 (R14), D22 (R12), D18 (R10), D14 (R8), D10 (R6), D6 (R4), & D2 (R2) @ Rcvr_Ctl_2

This bit drives the associated Harris chip's **CCLK** input pin. Data at the **CDATA** input pin (see Section 3.5.9.1, above) is shifted into the Harris Control Registers on the high–to–low transition of this pin.

3.5.9.3 Setup Data Strobe Bits – D29 (R15), D25 (R13), D21 (R11), D17 (R9), D13 (R7), D9 (R5), D5 (R3) & D1 (R1) @ Rcvr_Ctl_1, and D29 (R16), D25 (R14), D21 (R12), D17 (R10), D13 (R8), D9 (R6), D5 (R4), & D1 (R2) @ Rcvr_Ctl_2

This bit drives the associated Harris chip's **CSTB** input pin. The Strobe Bit must be pulsed high before the first setup data bit (bit 39, the MSB) is written into any of the Harris Control Registers. The Strobe bit for a given receiver must be set to the logic '1' state after the clock bit for that receiver has been set. The strobe must remain high until after the clock bit has been cleared to the logic '0' state. These bits must be returned to the logic '0' state before any setup data can be written, and must remain in the '0' state for nor–mal receiver operation.

3.5.9.4 Receiver Chip Select Bits – D28 (R15), D24 (R13), D20 (R11), D16 (R9), D12 (R7), D8 (R5), D4 (R3) & D0 (R1) @ Rcvr_Ctl_1, and D28 (R16), D24 (R14), D20 (R12), D16 (R10), D12 (R8), D8 (R6), D4 (R4), & D0 (R2) @ Rcvr_Ctl_2

This bit drives the associated Harris chip's **CS** input pin. The Chip Select Bit should be taken to the logic '0' state before the data strobe is pulsed high, and should remain a '0' until the last setup data bit (bit 0, the LSB) has been shifted in. Return this bit to the logic '1' state for normal receiver operation.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.10 Multiplex and Mode Register – R/W @ Mux_Mode

This 16–bit register, which can be accessed by the 'C40 processor and by the VMEbus, contains two 8–bit fields. Each field contains one bit associated with each receiver. One field selects the input channel for the associated receivers. The other field is used to configure the output FIFOs of the associated receiv–ers for Real–Only or Complex Mode operations. Table 3–24, below, shows the bit arrangement within the Multiplex and Mode Register. The subsections that begin below the table describe how these bits are used.

NOTE: In units equipped with Option 008, there are 32 bits in this register. The high order word is used to control PIO functions. These are documented in Section 3.5.12, which begins on the next page.

Table 3-24: Model 6516 - Multiplex and Mode Register - R/W @ Mux_Mode

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Name	Receiver_ 15 & 16_ Input	Receiver_ 13 & 14_ Input	Receiver_ 11 & 12_ Input	Receiver_ 9 & 10_ Input	Receiver_ 7 & 8_ Input	Receiver_ 5 & 6_ Input	Receiver_ 3 & 4_ Input	Receiver_ 1 & 2_ Input
Bit Function	1 = Channel D* 0 = Channel C*				1 = Channel B 0 = Channel A			
Bit #	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Bit Name	Receiver_ 15 & 16_ Mode	Receiver_ 13 & 14_ Mode	Receiver_ 11 & 12_ Mode	Receiver_ 9 & 10_ _Mode	Receiver_ 7 & 8_ _Mode	Receiver_ 5 & 6_ _Mode	Receiver_ 3 &4_ _Mode	Receiver_ 1 & 2_ _Mode
Bit Function	1 = Real Only 0 = Complex							

NOTE: In units equipped with Option 008, this is a 32-bit register.
The upper 16 bits control PIO functions, and are documented in **Section 3.5.12**.
* - When bits D28 and/or D20 of the Input_Ctl Register (**Table 3-19**) are set to '1', D15-D12 correspond to Channel B and/or A.

3.5.10.1 Input Signal Source Selection Bits – D15 through D8

Any of the eight receiver pairs on the Model 6516 can select either the Channel A or Channel B input connector at the source of their input data. Channel A is selected for a given receiver pair when the associated bit is cleared to the logic '0' state, and Channel B is selected when the bit is set to the logic '1' state. Note that, in order for receivers 9 – 16 to receive any data, the Reserved bits (D28 and D20) in the Input Control Register must be set to the logic '1' state (see Section 3.5.8.1).

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.10 Multiplex and Mode Register (continued)

3.5.10.2 Receiver Mode Select Bits – D7 through D0

The programming of these bits depends upon how the internal Control Registers within the Harris receivers are programmed. If a receiver is programmed to operate in Real–Only Mode, the corre–sponding bit in this field must be set to the logic '1' state, to allow the receiver's FIFO to accept data only from the 'I' output. If the receiver is operating in Complex Mode, The corresponding bit must be cleared to the logic '0' state, to allow the FIFO to take data from both the 'I' and 'Q' outputs of the receiver.

3.5.11 Option 009 Multiplex and Mode Register

To implement the sync signal routing changes for the Model 6516, Option 009, significant changes were made in the functions of bits D15 – D8 in the Multi–plex and Mode register. In the original implementation (see Section 3.5.10, beginning on the previous page) each of these bits was used to select the input source for a pair of adjacent channels (D8 selected the input for R1 & R2, D9 for R3 & R4, and so on). For Option 009, each bit selects the input data source for one pair of receivers (R(n) and R(n+8)), and the sync source for another (adjacent) pair, as shown in Table 3–25, below.

Table 3-25: Model 6516, Option 009 - Multiplex and Mode Register - R/W @ Mux_Mode

Bit #	D15	D14	D13	D12	D11	D10	D9	D8
Bit Name	R8/R16_Data R15/R16_Sync	R7/R15_Data R11/R12_Sync	R4/R12_Data R13/R14_Sync	R3/R11_Data R9/R10_Sync	R6/R14_Data R7/R8_Sync	R5/R13_Data R3/R4_Sync	R2/R10_Data R5/R6_Sync	R1/R9_Data R1/R2_Sync
Bit Function	1 = Data D/Sync 4 * 0 = Data C/Sync 3 *				1 = Data B/Sync 2 0 = Data A/Sync 1			
Example	1	0	1	0	1	0	1	0

*** Note:** When Bits D28 and/or D20 of the Input_Ctl register (Table 3-18) are set to '1', D15-D12 select Data B and/or A, Sync 1 and/or 2.

Figure 3–3, on the next page, shows the sync and data routing for the Model 6516 Option 009. Receivers R1 through R8 can select between sync signals S1 and S2. Receivers R9 through R16 can select between sync signals S3 and S4, or, by means of the crossover bits (D20 and D28) in the Input Control Register (see Table 3–19 and/or Section 3.5.8.1 in the Model 6516 Operating Manual), can also use the sync signals S1 and/or S2.

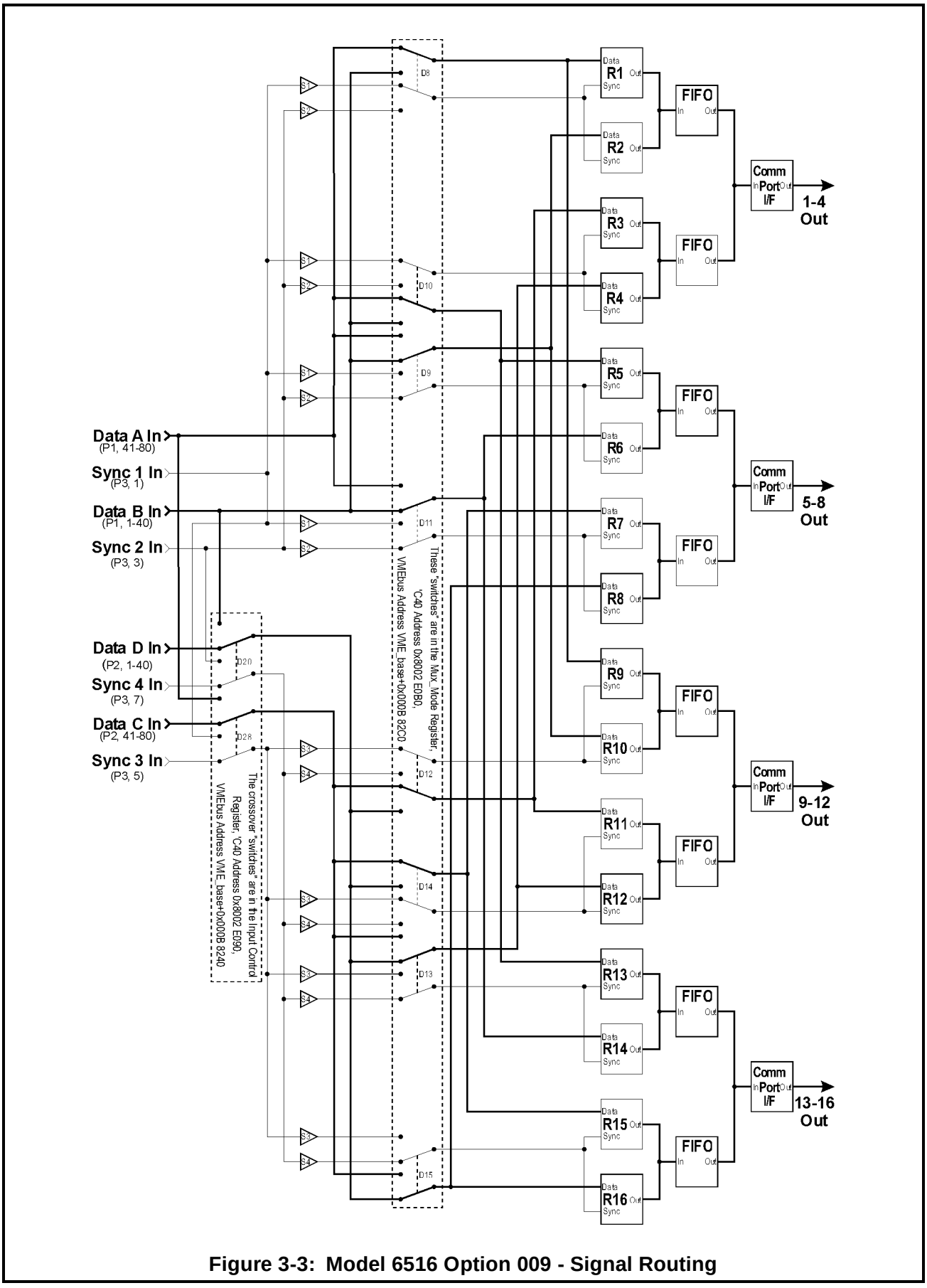


Figure 3-3: Model 6516 Option 009 - Signal Routing

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.11 Option 009 Multiplex and Mode Register (continued)

The input configuration that will likely prove the most useful that can be obtained with this option is achieved when the bit field referred to above contains the hex data 0xAA, as shown in the row labeled "Example" in Table 3–25, on the page before the previous. In this configuration, all 16 possible combinations of input data and sync are available. In Figure 3–3, on the previous page, the "switches" are set for this configuration, and Table 3–26, below, tells which receiver processes which input data stream, and which sync signal used to tune each receiver.

Table 3-26: Model 6516, Option 009 - Input Data & Reset (Sync) Routing

D15 - D8 in Mux_Mode Register = 0xAA			Reset (Sync) Source ◄ Common Data ►			
Data Source	Common Sync ▲▼	**Channel A**	Receiver 1 (S1)	Receiver 5 (S2)	Receiver 9 (S3)	Receiver 13 (S4)
		Channel B	Receiver 2 (S1)	Receiver 6 (S2)	Receiver 10 (S3)	Receiver 14 (S4)
		Channel C	Receiver 3 (S1)	Receiver 7 (S2)	Receiver 11 (S3)	Receiver 15 (S4)
		Channel D	Receiver 4 (S1)	Receiver 8 (S2)	Receiver 12 (S3)	Receiver 16 (S4)

3.5.12 PIO Register – Option 008 ONLY – R/W @ Mux_Mode_PIO

In units equipped with Option 008, the Multiplex and Mode Register, described in Section 3.5.10, beginning on page 72, has 16 more bits than on standard units. The additional word (D31 – D16) is used to determine what the Model 6516 does in response to signals received on the FPDP PIO pins. One byte in this word is assigned to the pair of PIO signals on each FPDP connector. D31 – D24 are associated with the PIO signals on FPDP2, and D23 – D16 work with the PIO signals on FPDP1. Four bits are associated with each PIO signal line – a Status bit, a "catch" bit, a hold bit and a clear bit. Table 3–27, below, shows the bit arrangement in the PIO portion of the 6516 Option 008's Multiplex, Mode & PIO register, and the subsections beginning at the top of the next page describe how these bits are used.

Table 3-27: Model 6516 - PIO Register - R/W @ Mux_Mode_PIO - Option 008 ONLY

Bit #	**D31***	**D30***	**D29**	**D28**	**D27**	**D26**	**D25**	**D24**
Bit Name	FPDP2-PIO2 _Status	FPDP2-PIO1 _Status	FPDP2-PIO2 _Catch	FPDP2-PIO1 _Catch	FPDP2-PIO2 _Hold	FPDP2-PIO1 _Hold	FPDP2-PIO2 _Clear	FPDP2-PIO1 _Clear
Bit #	**D23***	**D22***	**D21**	**D20**	**D19**	**D18**	**D17**	**D16**
Bit Name	FPDP1-PIO2 _Status	FPDP1-PIO1 _Status	FPDP1-PIO2 _Catch	FPDP1-PIO1 _Catch	FPDP1-PIO2 _Hold	FPDP1-PIO1 _Hold	FPDP1-PIO2 _Clear	FPDP1-PIO1 _Clear
Bit Function	Hold - "caught" PIO state Pass - PIO Value latched by strobe		1 = Catch 0's 0 = Catch 1's		1 = Hold "caught" bit 0 = Pass-through		1 = Enable Status Bit 0 = Clear Status Bit	
* - These bits are Read-Only.								

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.12 PIO Register – Option 008 ONLY (continued)

NOTE: The Model 6516 Option 018 defines the PIO signal lines as inputs, and it is strongly recommended that these signals meet the same timing requirements with respect to clock edges (STROB or PSTROBE, $\overline{\text{PSTROBE}}$) as the DVALID signal.

3.5.12.1 PIO Status Bits – D31 (FPDP2, PIO2), D30 (FPDP2, PIO1), D23 (FPDP1, PIO2) & D22 (FPDP1, PIO1) – Read Only

These four read–only bits are essentially the targets for the operations of all the other bits in this register. The state of the Status bit for a given PIO signal is dependent upon the state of the PIO signal itself, and the states of the other four bits associated with that signal.

If the PIO Clear bit (see Section 3.5.12.4) associated with a given PIO signal is held in the logic '0' state, then the Status bit associated with that signal will always read back as a logic '0'. The PIO Clear bit must be set to the logic '1' state to enable the associated PIO status bit.

If a given PIO Hold bit (see Section 3.5.12.3) is cleared to the logic '0' state, then the associated PIO Status bit directly follows the state of its PIO signal line. The state of the PIO signal is latched into the status register by the strobe signal selected as the FPDP clock in the Input Control Register (see Section 3.5.8.7). If a PIO Hold bit is set to the logic '1' state, then that PIO Status bit will monitor its PIO input and will hold the first occurance of the logic state selected by the associated PIO Catch bit.

If a PIO Catch bit (see Section 3.5.12.2) is programmed for detecting 0's, the associated PIO Status bit is quiescently in the logic '0' state, and will transition to a logic '1' when its PIO input does. If a Catch bit is programmed to detect 0's, the associated Status bit is quiescently a logic '1', and will transition to logic '0' when its PIO input does.

The PIO status bits can be used to generate interrupts to the 'C40. When the PIO Status Bit detects the "catch" state at the input, the associated PIO Interrupt Status Bit in Receiver Status Register #1 (see Section 3.5.4.4 and Table 3–12) will be cleared (i. e., become true). If the associated Interrupt Mask bit is set to the logic '1' state (see Section 3.5.2.2 and Table 3–10), a 'C40 interrupt will be generated.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.12 PIO Register – Option 008 ONLY (continued)

3.5.12.2 PIO Catch Bits – D29 (FPDP2, PIO2), D28 (FPDP2, PIO1), D21 (FPDP1, PIO2) & D20 (FPDP1, PIO1)

The PIO Catch bits determine which state of the PIO input signal can be used to generate a 'C40 inerrupt. When the Catch bit is set to the logic '1' state, interrupts can be generated when the PIO input goes low. When the Catch bit is cleared to the logic '0' state, interrupts can be generated when the PIO input goes high. The quiescent states of the PIO Status bits (see Section 3.5.12.1) are adjusted according to the programming of the associated Catch bits (normally 0 if catching 1's, normally 1 if catching 0's).

3.5.12.3 PIO Hold Bits – D27 (FPDP2, PIO2), D26 (FPDP2, PIO1), D19 (FPDP1, PIO2) & D18 (FPDP1, PIO1)

When a PIO Hold bit is cleared to the logic '0' state, the associated PIO Status bit follows the state PIO input signal, latched by the clock signal selected in the Input Control Register (see Section 3.5.8.7). When a hold bit is set to the logic '1' state, the associated Status bit will, upon the first transition of its PIO input to the state selected by the associated Catch bit (see Section 3.5.12.2, above), assume and remain in that logic state until cleared by the associated PIO Clear bit (see Section 3.5.12.4, below).

3.5.12.4 PIO Clear Bits – D25 (FPDP2, PIO2), D24 (FPDP2, PIO1), D17 (FPDP1, PIO2) & D16 (FPDP1, PIO1)

When a PIO Clear bit is cleared to the logic '0" state, the associated PIO status bit is held in its quiescent state, as determined by the associated PIO Catch bit (see Section 3.5.12.2). To release a PIO Status bit, and allow the operations described in the sections above to take place, the associated PIO Clear bit must be set to the logic '1' state.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.13 Tag Enable Register – R/W @ Tag_En

The feature controlled by this 8–bit register allows each output sample read from the Model 6516's FIFOs to be individually identified as an In–phase (I) or a Quadrature (Q) sample. The register can be accessed by the VMEbus and by the 'C40 processor. When one of these bits is set to the logic '1' state, a '0' is substituted for the LSB of each 16–bit 'Q' sample, and a '1' is substituted for the LSB of each 16–bit 'I' sample. This presents a small trade–off in accuracy, as the LSB of each 16–bit sample will be lost to the Tag bit. To disable the Tagging function and allow all 16 bits from all receivers to be read out, leave these bits cleared to their default logic '0' states. Table 3–28, below, shows which bit enables Tag mode for which receiver. Table 3–29, at the top of the next page, shows examples of Tagged data in the relevant formats.

Table 3-28: Model 6516 - Tag Enable Register - R/W @ Tag_En

Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Name	Receivers_ 15 & 16 _Tag_Enable	Receivers_ 13 & 14 _Tag_Enable	Receivers_ 11 & 12 _Tag_Enable	Receivers_ 9 & 10 _Tag_Enable	Receivers_ 7 & 8 _Tag_Enable	Receivers_ 5 & 6 _Tag_Enable	Receivers_ 3 & 4 _Tag_Enable	Receivers_ 1 & 2 _Tag_Enable
Bit Function	1 = Tag on LSB 0 = Data on LSB							

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.13 Tag Enable Register (continued)

Table 3-29: Model 6516 - Comm Port Data Packing and Ordering Formats (Tagging Enabled) (Examples - for FIFOs 1 & 2)

Packed, Receiver Ordered Mode, Complex Data - Dual FIFO Mode

Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D1	D0
1	(MSB)	#3 Receiver, Q Sample 1	(2SB)	0	(MSB)	#1 Receiver, Q Sample 1	(2SB)	0
2	(MSB)	#3 Receiver, I Sample 1	(2SB)	1	(MSB)	#1 Receiver, I Sample 1	(2SB)	1
3	(MSB)	#4 Receiver, Q Sample 1	(2SB)	0	(MSB)	#2 Receiver, Q Sample 1	(2SB)	0
4	(MSB)	#4 Receiver, I Sample 1	(2SB)	1	(MSB)	#2 Receiver, I Sample 1	(2SB)	1
5	(MSB)	#3 Receiver, Q Sample 2	(2SB)	0	(MSB)	#1 Receiver, Q Sample 2	(2SB)	0
6	(MSB)	#3 Receiver, I Sample 2	(2SB)	1	(MSB)	#1 Receiver, I Sample 2	(2SB)	1
7	(MSB)	#4 Receiver, Q Sample 2	(2SB)	0	(MSB)	#2 Receiver, Q Sample 2	(2SB)	0
8	(MSB)	#4 Receiver, I Sample 2	(2SB)	1	(MSB)	#2 Receiver, I Sample 2	(2SB)	1

Packed, Receiver Ordered Mode, Complex Data - Single FIFO Mode

Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D1	D0
1	(MSB)	#3 Receiver, Q Sample 1	(2SB)	0	(MSB)	#1 Receiver, Q Sample 1	(2SB)	0
2	(MSB)	#3 Receiver, I Sample 1	(2SB)	1	(MSB)	#1 Receiver, I Sample 1	(2SB)	1
3	(MSB)	#3 Receiver, Q Sample 2	(2SB)	0	(MSB)	#1 Receiver, Q Sample 2	(2SB)	0
4	(MSB)	#3 Receiver, I Sample 2	(2SB)	1	(MSB)	#1 Receiver, I Sample 2	(2SB)	1

Non-Packed, Receiver Ordered Mode, Complex Data - Dual FIFO Mode

FIFO Select = Odd *(Even)* FIFO	Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D0
	1	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(2SB)	0	I n d e t e r m i n a t e		
	2	(MSB)	#1 *(#3)* Receiver, I Sample 1	(2SB)	1	I n d e t e r m i n a t e		
	3	(MSB)	#2 *(#4)* Receiver, Q Sample 1	(2SB)	0	I n d e t e r m i n a t e		
	4	(MSB)	#2 *(#4)* Receiver, I Sample 1	(2SB)	1	I n d e t e r m i n a t e		
	5	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(2SB)	0	I n d e t e r m i n a t e		
	6	(MSB)	#1 *(#3)* Receiver, I Sample 2	(2SB)	1	I n d e t e r m i n a t e		
	7	(MSB)	#2 *(#4)* Receiver, Q Sample 2	(2SB)	0	I n d e t e r m i n a t e		
	8	(MSB)	#2 *(#4)* Receiver, I Sample 2	(2SB)	1	I n d e t e r m i n a t e		

Non-Packed, Receiver Ordered Mode, Complex Data - Single FIFO Mode

FIFO Select = Odd *(Even)* FIFO	Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D0
	1	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(2SB)	0	I n d e t e r m i n a t e		
	2	(MSB)	#1 *(#3)* Receiver, I Sample 1	(2SB)	1	I n d e t e r m i n a t e		
	3	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(2SB)	0	I n d e t e r m i n a t e		
	4	(MSB)	#1 *(#3)* Receiver, I Sample 2	(2SB)	1	I n d e t e r m i n a t e		

Packed, I/Q Ordered Mode, Complex Data - Dual FIFO Mode

FIFO Select = Odd *(Even)* FIFO	Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D1	D0
	1	(MSB)	#1 *(#3)* Receiver, I Sample 1	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(2SB)	0
	2	(MSB)	#2 *(#4)* Receiver, I Sample 1	(2SB)	1	(MSB)	#2 *(#4)* Receiver, Q Sample 1	(2SB)	0
	3	(MSB)	#1 *(#3)* Receiver, I Sample 2	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(2SB)	0
	4	(MSB)	#2 *(#4)* Receiver, I Sample 2	(2SB)	1	(MSB)	#2 *(#4)* Receiver, Q Sample 2	(2SB)	0

Packed, I/Q Ordered Mode, Complex Data - Single FIFO Mode

FIFO Select = Odd *(Even)* FIFO	Sample #	D31	- - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - -	D1	D0
	1	(MSB)	#1 *(#3)* Receiver, I Sample 1	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 1	(2SB)	0
	2	(MSB)	#1 *(#3)* Receiver, I Sample 2	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 2	(2SB)	0
	3	(MSB)	#1 *(#3)* Receiver, I Sample 3	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 3	(2SB)	0
	4	(MSB)	#1 *(#3)* Receiver, I Sample 4	(2SB)	1	(MSB)	#1 *(#3)* Receiver, Q Sample 4	(2SB)	0

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.14 Threshold Detection Registers – R/W @ Thr_Det_A, Thr_Det_B, Thr_Det_C & Thr_Det_C

These registers determine the conditions that will set the Threshold Detector flags in Status Register #1 (D22, D20, D18 & D16 @ Stat_Reg_1, see Section 3.5.4.3). Those flags, in turn, can be enabled to produce Threshold Detector Interrupts by setting the appropriate bits in Interrupt Mask Register #1 to the logic '1' state (D22, D20, D18 & D16 @ Int_Mask_1, see Section 3.5.2.1). The register at symbolic address Thr_Det_A is used for the receivers driven by Input Channel A, the one at Thr_Det_B is used for those receivers driven by Channel B, Thr_Det_C is for Channel C's receivers, and Thr_Det D is for Channel D's. Both of these registers are accessible to both the 'C40 processor and VMEbus masters. The upper eight bits of the Multiplex and Mode Register (see Section 3.5.10.1) determines which receivers are driven by which input channel. The contents of the Threshold Detection Registers are summarized in Table 3–30, below. The subsections beginning below the table describe the functions of the registers' bits.

Table 3-30: Model 6516 - Threshold Detection Registers - R/W @ Thr_Det_A, Thr_Det_B, Thr_Det_C & Thr_Det_D

Bit #	D7	D6	D5	D4	D3	D2	D1	D0
Bit Name	Detector _Enable	Threshold _Level_1	Threshold _Level_0	Not Used		Crossing_ Count_2	Crossing_ Count_1	Crossing_ Count_0
Bit Function	1 = Enable 0 = Disable*	See Table 3-31, page 81		Not Applicable		See Table 3-32, page 81		
* - Negative edge resets counter								

3.5.14.1 Threshold Detector Enable Bit – D7

When this bit is cleared to the logic '0' state, the threshold detection function is disabled for receivers driven by the associated input channel. The low–going transition of this bit (from logic'1' to logic '0') resets the Threshold Crossing Counter (see Section 3.5.14.3) to the zero count. To enable the threshold detection function and release the Threshold Crossing Counter, set this bit to the logic '1' state. The counter reset function should be implemented as a toggle (i. e., clear this bit, then set it).

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.14 Threshold Detection Registers (continued)

3.5.14.2 Threshold Level Field – Bits D6 & D5

Bits D6 & D5 of the Threshold Detection Registers are used to select the data input level that will be recognized as the threshold, according to Table 3–31, below. The Threshold Crossing Counter is incremented **for each data sample** on the channel of interest that is either greater than or equal to the positive threshold data value, or less than or equal to the negative threshold data value.

Table 3-31: Model 6516 - Threshold Level Bits

D6 (Lev_1)	D5 (Lev_0)	Threshold Setting	Positive Data Value (Hex)	Negative Data Value (Hex)
0	0	$\pm \frac{1}{2}$ of Full Scale	≥ 0x0400	≤ 0x0BFF
0	1	$\pm \frac{5}{8}$ of Full Scale	≥ 0x0500	≤ 0x0AFF
1	0	$\pm \frac{3}{4}$ of Full Scale	≥ 0x0600	≤ 0x09FF
1	1	$\pm \frac{7}{8}$ of Full Scale	≥ 0x0700	≤ 0x08FF

3.5.14.3 Crossing Count Field – Bits D2 through D0

Bits D2 – D0 of the Threshold Detection Registers are used to select the value in the Threshold Crossing Counter that will set the Threshold Detection flags in Status Register #1 to the logic '1' state, for those receivers driven by the input channel of interest. These status flags can generate 'C40 interrupts if enabled in Interrupt Mask Register #1 (see Section 3.5.4.3 and Section 3.5.2.1). Increasing binary values in this 3–bit field result in increasing odd power–of 2 counter values setting the flags, according to Table 3–32, below.

Table 3-32: Model 6516 - Threshold Crossing Count Value To Set Status Flag

D2	D1	D0	Count
0	0	0	2 (2^1)
0	0	1	8 (2^3)
0	1	0	32 (2^5)
0	1	1	128 (2^7)
1	0	0	512 (2^9)
1	0	1	2048 (2^{11})
1	1	0	8192 (2^{13})
1	1	1	32768 (2^{15})

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.15 Receiver Output Data FIFOs – R. O. @ FIFO_1 through FIFO_8

Output data from each digital receiver on the Model 6516 is stored in a FIFO. Data can be read from these eight FIFOs over the VMEbus, and by the on–board 'C40, as well as over the Comm Port interface by off–card 'C40–type devices. The contents of the Output Comm Port Control Register do not affect the format of data read directly from the FIFOs (except that direct readout is disabled if the Comm Ports are enabled). Data packing is unavailable in this mode (i. e., 32–bit reads from the FIFOs will only yield a single, 16–bit data sample). The Tag function, however is available from the FIFOs. An attempt to read an empty FIFO will cause wait states to be asserted on the device mas–tering the read cycle.

Note that FIFO readout may be accomplished in either of two ways. All reads from a given FIFO may be performed at a single address, with each read producing the next available FIFO sample. Alternatively, block transfers can be used, by incrementing addresses within the given FIFO's address range. The address range for each FIFO is 8 kWords (16 kBytes). Again, each read operation will produce the next available sample from the FIFO. Please refer to Table 3–7, on page 45, for the address range allotted to each receiver pair's FIFO.

NOTE: Because the FIFO address range is 8 kWords, units equipped with Option 021 (16 kSample FIFOs) each FIFO will require 2 block transfer operations of 8 kWords each to empty if full.

Option 018 for the Model 6516 provides eight receiver channels, with 32–bit output data. Beacuse data packing within the FIFO is not available, two FIFO reads are required to obtain each 32–bit sample. In this case, the first, and all subsequent odd FIFO reads deliver the low–order words (with the tag bit, if tagging is enabled), and the second, and all subsequent even FIFO reads deliver the high order words (without a tag bit).

Data read from the FIFOs is always ordered by receiver, never packed, and may be tagged or untagged. For a complete description of data ordering modes, see Section 3.5.7.2 and Tables 3–17 & 3–18. For a complete desc–ription of data tagging, see Section 3.5.13 and Table 3–29. The formats available when the Model 6516's FIFOs are read directly are shown in Table 3–33, on the next page. Table 3–34, on the page after next, gives the data formats for 32–bit output data from the FIFOs on the Model 6516 Option 018. Note that because the Comm Ports on the front panel of Option 018 are 'C40 input ports, direct FIFO readout is the only method of obtaining output data from these devices. Also, because Option 018 pro–vides only 8 receiver channels (the odd–numbered channels only), there is no Dual–channel FIFO mode.

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.15 Receiver Output Data FIFOs (continued)

Table 3-33: Model 6516 - Direct FIFO Readout Data Formats - FIFO #1 Examples

Untagged, Real Data Only - Dual Mode

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - I Sample 1	(LSB)		Indeterminate	
2	(MSB)	Receiver #2 - I Sample 1	(LSB)		Indeterminate	
3	(MSB)	Receiver #1 - I Sample 2	(LSB)		Indeterminate	
4	(MSB)	Receiver #2 - I Sample 2	(LSB)		Indeterminate	

Untagged, Real Data Only - Single Mode

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - I Sample 1	(LSB)		Indeterminate	
2	(MSB)	Receiver #1 - I Sample 2	(LSB)		Indeterminate	
3	(MSB)	Receiver #1 - I Sample 3	(LSB)		Indeterminate	
4	(MSB)	Receiver #1 - I Sample 4	(LSB)		Indeterminate	

Untagged, Complex Data - Dual Mode

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - Q Sample 1	(LSB)		Indeterminate	
2	(MSB)	Receiver #1 - I Sample 1	(LSB)		Indeterminate	
3	(MSB)	Receiver #2 - Q Sample 1	(LSB)		Indeterminate	
4	(MSB)	Receiver #2 - I Sample 1	(LSB)		Indeterminate	
5	(MSB)	Receiver #1 - Q Sample 2	(LSB)		Indeterminate	
6	(MSB)	Receiver #1 - I Sample 2	(LSB)		Indeterminate	
7	(MSB)	Receiver #2 - Q Sample 2	(LSB)		Indeterminate	
8	(MSB)	Receiver #2 - I Sample 2	(LSB)		Indeterminate	

Untagged, Complex Data - Single Mode

Sample #	D31	- - - - - - - - - - - - - - - - - -	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - Q Sample 1	(LSB)		Indeterminate	
2	(MSB)	Receiver #1 - I Sample 1	(LSB)		Indeterminate	
3	(MSB)	Receiver #1 - Q Sample 2	(LSB)		Indeterminate	
4	(MSB)	Receiver #1 - I Sample 2	(LSB)		Indeterminate	

Tagged, Complex Data - Dual Mode

Sample #	D31	- - - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - Q Sample 1	(2SB)	0		Indeterminate	
2	(MSB)	Receiver #1 - I Sample 1	(2SB)	1		Indeterminate	
3	(MSB)	Receiver #2 - Q Sample 1	(2SB)	0		Indeterminate	
4	(MSB)	Receiver #2 - I Sample 1	(2SB)	1		Indeterminate	
5	(MSB)	Receiver #1 - Q Sample 2	(2SB)	0		Indeterminate	
6	(MSB)	Receiver #1 - I Sample 2	(2SB)	1		Indeterminate	
7	(MSB)	Receiver #2 - Q Sample 2	(2SB)	0		Indeterminate	
8	(MSB)	Receiver #2 - I Sample 2	(2SB)	1		Indeterminate	

Tagged, Complex Data - Single Mode

Sample #	D31	- - - - - - - - - - - - - - - - -	D17	D16	D15	- - - - - - - - - - - - - - - - - -	D0
1	(MSB)	Receiver #1 - Q Sample 1	(2SB)	0		Indeterminate	
2	(MSB)	Receiver #1 - I Sample 1	(2SB)	1		Indeterminate	
3	(MSB)	Receiver #1 - Q Sample 2	(2SB)	0		Indeterminate	
4	(MSB)	Receiver #1 - I Sample 2	(2SB)	1		Indeterminate	

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.15 Receiver Output Data FIFOs (continued)

Table 3-34: Model 6516 Option 018 - 32-bit Direct FIFO Readout Formats - FIFO #1 Examples

Untagged, Real Data Only						
Sample #	**D31**	**- - - - - - - - - - - - - - - - - - -**	**D16**	**D15**	**- - - - - - - - - - - - - - - - - - -**	**D0**
1	(B15)	Receiver #1 - I Sample 1 Lo	(LSB)		I n d e t e r m i n a t e	
2	(MSB)	Receiver #1 - I Sample 1 Hi	(B16)		I n d e t e r m i n a t e	
3	(B15)	Receiver #1 - I Sample 2 Lo	(LSB)		I n d e t e r m i n a t e	
4	(MSB)	Receiver #1 - I Sample 2 Hi	(B16)		I n d e t e r m i n a t e	
Untagged, Complex Data						
Sample #	**D31**	**- - - - - - - - - - - - - - - - - - -**	**D16**	**D15**	**- - - - - - - - - - - - - - - - - - -**	**D0**
1	(B15)	Receiver #1 - Q Sample 1 Lo	(LSB)		I n d e t e r m i n a t e	
2	(MSB)	Receiver #1 - Q Sample 1 Hi	(B16)		I n d e t e r m i n a t e	
3	(B15)	Receiver #1 - I Sample 1 Lo	(LSB)		I n d e t e r m i n a t e	
4	(MSB)	Receiver #1 - I Sample 1 Hi	(B16)		I n d e t e r m i n a t e	
5	(B15)	Receiver #1 - Q Sample 2 Lo	(LSB)		I n d e t e r m i n a t e	
6	(MSB)	Receiver #1 - Q Sample 2 Hi	(B16)		I n d e t e r m i n a t e	
7	(B15)	Receiver #1 - I Sample 2 Lo	(LSB)		I n d e t e r m i n a t e	
8	(MSB)	Receiver #1 - I Sample 2 Hi	(B16)		I n d e t e r m i n a t e	
Tagged, Complex Data						
Sample #	**D31**	**- - - - - - - - - - - - - - - - - -**	**D17** **D16**	**D15**	**- - - - - - - - - - - - - - - - - - -**	**D0**
1	(B15)	Receiver #1 - Q Sample 1 Lo	(B01) 0		I n d e t e r m i n a t e	
2	(MSB)	Receiver #1 - Q Sample 1 Hi	(B16)		I n d e t e r m i n a t e	
3	(B15)	Receiver #1 - I Sample 1 Lo	(B01) 1		I n d e t e r m i n a t e	
4	(MSB)	Receiver #1 - I Sample 1 Hi	(B16)		I n d e t e r m i n a t e	
5	(B15)	Receiver #1 - Q Sample 2 Lo	(B01) 0		I n d e t e r m i n a t e	
6	(MSB)	Receiver #1 - Q Sample 2 Hi	(B16)		I n d e t e r m i n a t e	
7	(B15)	Receiver #1 - I Sample 2 Lo	(B01) 1		I n d e t e r m i n a t e	
8	(MSB)	Receiver #1 - I Sample 2 Hi	(B16)		I n d e t e r m i n a t e	

3.5 'C40 Global and VMEbus A32/A24 Slave Memory Map (continued)

3.5.16 TMS320C40 IACK Vector Address – W. O. @ 'C40 Address 0x8004 0000

The VMEbus can issue an interrupt to the Model 6516's 'C40 Processor by writing to address A16_base+0x08. Writing this address will assert an interrupt at the 'C40's IIOF1 input, and set the Pending_C40_Interrupt Bit (D15 at address A16_base+0x10) to the logic '1' state.

The 'C40 should acknowledge the IIOF1 interrupt by writing arbitrary data (the data is unused) to Global bus address 0x8004 0000. This will clear the Pending_C40_Interrupt bit in the VMEbus Status Register (A16_base+0x10) to the logic '0' state. Note that an A32 VME master can also access this location, at address A32_base+0x0010 0000, but the usefulness of such an access is questionable, to say the least. No A24 access to this resource is provided.

3.5.17 TMS320C40 Interrupt to VMEbus – W. O. @ 'C40 Address 0x8005 0000

Upon receiving one of the FIFO–related, Overload or Clock Timeout interrupts covered in Section 3.5.2 and Section 3.5.3, the 'C40 processor on the 6516 may respond by passing the interrupt to the VMEbus. The 'C40 processor interrupts VME by writing arbitrary data (the data is unused) to Global bus address 0x8005 0000. The interrupt is asserted on the IRQ Level selected by the IRQ_Level_Select field in the VMEbus Control Register (bits D10 – D8 at the A16_base address – see Section 3.4.1.5 and Table 3–3). When this interrupt is generated, the MSB (D31, the VME Interrupt Status Bit) of Status Register #1 (see Section 3.5.4.1) is set to the logic '1' (true) state. The VMEbus IACK cycle sets the status bit back to the logic '0' (false) state. The 6516's IACK Vector Address is stored in the 8 LSBs of the VMEbus Control Register (see Table 3–3 and Section 3.4.1.6).

Note that an A32 VME master can also access this location, at address A32_base+0x0014 0000, but the usefulness of such an access is highly questionable. No A24 access to this resource is provided.

3.5.18 Global Flash EEPROM – R/W @ Flash

A 512 kByte bank of Flash EEPROM is available to both the 6516's 'C40 Processor and A32 VMEbus masters (no A24 access to this resource is provided). This memory region is arranged as byte data located on longword addresses. In other words, one byte of data may be read from or written to the low byte (D0 – D7) of each longword address (i. e., every fourth byte address). When shipped, each Model 6516 contains firmware in Sectors 1 & 2. Customized application code may be stored in this non–volatile memory bank, to facilitate the creation of standalone signal processing systems. However, if you want to write your own code to perform the functions provided by the firmware, you must erase the Flash EEPROM prior to writing any custom code to the EEPROM. The 'C' program **`erase65.c`**, located in Appendix A, can be used to erase the EEPROM.

3.6 TMS320C40 Local Bus Resources

The Local Bus interface of the Model 6516's TMS320C40 processor contains two banks of memory, a 32 kByte (8 k–Longword) EEPROM and a 512 kByte (128 k–Longword) SRAM. Table 3–35, below, shows the 'C40's Local Bus memory map. The subsections that begin at the top of the next page describe the resources.

Table 3-35: Model 6516 - TMS320C40 Local Bus Memory Map

Address Range	Size	Access	Resource
0x0030 0000 - 0x0030 7FFF	8k x 32	R. O.	Boot EEPROM
0x4000 0000 - 0x4001 FFFF	128k x 32	R / W	Local SRAM

3.6.1 **Boot EEPROM** – R. O. from 0x0030 0000 to 0x0030 7FFF

This 8 k–Longword (32 kByte) EEPROM stores the factory supplied boot code for the 'C40 processor on the Model 6516. The 'C40 boots from this EEPROM by default. If 6516 is reset after the D14 bit in the VMEbus Control Register (at the A16_base address) has been set to the logic '1' state, the 'C40 will boot from the Comm Port interface instead of this EEPROM.

3.6.2 **Local SRAM** – R/W from 0x4000 0000 to 0x4001 FFFF

This 128k–longword (512 kByte) bank of SRAM can be accessed only by the 'C40 processor on the 6516.

Chapter 4: 65xx Series Firmware Command Set

4.1 Progamming Model 6516 Digital Receiver

The Model 6516 Digital Receiver can be programmed using a command set provided by the firmware, directly from the VMEbus, or by using a combination of these two methods. Pentek recommends using firmware commands to modify the settings of the Harris HSP50016 because these registers are written by serially shifting the control reg–ister settings into them. Pentek also recommends using 'C' library routines to control the reset and release of the FIFOs, and to perform synchronization of the DDRs. The examples which best illustrate how to do this are **`main65xx.c`** and **`main.c`** in the Appendix A of this manual.

Below is a sequence of Firmware commands used to provide the initial settings of the board:

1) Send the **`P65XX_CLOSE_ALL_CONNECTIONS`** command. This command restores the connection table on the 6516.

2) Send the **`P65XX_RESET_REGS`** command. This command resets the registers on the 6516 to a default state.

3) Send the **`P65XX_OPEN_CONNECTION`** command for each Receiver channel. This is necessary because the firmware keeps track of each receiver's setup. A handle is returned to identify this connection. If the setups are identical on all the receivers with the exception of the Local Oscillator Frequency (LO), then it is only necessary to open one connection. In this case, the returned handle can be used to setup all receiver channels.

4) Send the **`P65XX_INIT_INPUT_CHANNELS`** command to setup the input channels. The **`P6516_INPUT_PARAMS`** data structure is sent with this command. These parameters specify the input crossover, threshold detection, and input reset modes.

5) Send the **`P65XX_INIT_RCVRS`** command to each receiver using the handle returned from the **`P65XX_OPEN_CONNECTION`** command. The **`P6516_RCVR_PARAMS`** data structure is sent with this command. These parameters specify all the DDR setup parameters.

6) Send the **`P65XX_MODIFY_INTR_MASK`** specifying the interrupts to be enabled and the method of notification.

The only other command you are likely to need is **`P65XX_SET_LO`**. This command changes the tuning frequency on one or more DDRs.

4.1 Progamming Model 6516 Digital Receiver (continued)

All the other functions can be controlled using the 'C' functions provided in **`init6516.c`**. Some of the functions you will need are:

- **`P6516InitRegAddr`** – This function sets up a data structure containing the addresses of all the 6516's Registers as viewed from the VME Controller Board.
- **`P6516ResetFifo`** – This function resets one of more FIFOs.
- **`P6516ReleaseFifo`** – This function releases one or more FIFOs.
- **`P6516SetInputResetMode`** – This function specifies the Input Reset Source for one or more input channels. The sync master would specify a bit mask of (**`P6516_RESET_OUT_ENA | P6516_FP_RESET_IN_SRC`**). The sync slaves would specify a bit mask of (**`P6516_RESET_OUT_DIS | P6516_FP_RESET_IN_SRC`**). When the sync master wants to sync the other channels, the Software Reset bit must be toggled between **`P6516_SW_RESET_ON`** and **`P6516_SW_RESET_OFF`**.

4.2 Disabling the Model 6516 Firmware

The firmware on the Model 6509 can be disabled either temporarily or permanently. The temporary solution allows you to disable the firmware, but requires you to disable it each time the reciever is powered on.

4.2.1 Temporary Solution

Setup the Model 6516's A24/A32 base address, then write the value **`0xC33C 3CC3`** to the A24/A32 base address. After writing this value you must reset the Model 6516. This can be done by writing **`0x800`** to the VMEbus Control Register, at the A16_base address. This process must be repeated upon each power–up cycle.

4.2.2 Permanent Solution

The permanent solution for disabling the firmware involves erasing the FLASH memory that contains the firmware. The program **`erase65.c`** is included in Appendix A, which can be used to permanently disable the Model 6516 firmware.

4.3 Flash EEPROM Operations

The Flash EEPROM supports non−volatile storage of programs for boot loading. Other data, such as short tables, may also be stored here. This resource is mapped into the 'C40's Local Bus memory map, beginning at address 0x8008 0000 and VMEbase + 0x0020 0000 (in A32 space only).

Any data to be stored in this memory region must be in Intel HEX format. A program for converting COFF program files into this format, **`hex30`**, is provided by Texas Instruments as a part of their Optimizing 'C' Compiler package.

In most cases, the goal is to have the processor boot load the code in the Flash EEPROM immediately upon power−up or warm reset. The procedures involved in accomplishing that goal involve all of the operations you might want to perform using the Flash memory. Therefore, the subsections below describe the procedures for loading bootable HEX code into the Flash EEPROM, and boot loading into that code.

4.3.1 Converting COFF Files into Intel HEX Format

Program files to be written into the 6516's Flash EEPROM must be in Intel HEX format. COFF (Common Object File Format) files, of the type normally written into the SRAM regions of the Model 65xx, can be converted into Intel HEX format by creating an appropriately−named linker command file for the program, and passing the linker command file's name to the **`hex30`** program, supplied by Texas Instruments. The linker command file name should have the extension **`.cmd`**. An example of such a linker command file, **`boot65xx.cmd`**, is presented in Figure 4−1, on page 90, and is also included in Appendix A of this manual. The command line syntax to create the Intel HEX file **`firm65XX.hex`** from the COFF file **`firm65XX.xXX`**, using the linker command file shown in Figure 4−1 would be:

```
hex30 boot65XX.cmd<enter>
```

4.3 Flash EEPROM Operations (continued)

4.3.1 Converting COFF Files into Intel HEX Format (continued)

```
/*********************************************************************
*
*   File : boot65XX.cmd
*
*   Description: This file is an input to hex30.  It is used in the
*                conversion form COFF File Format to a Bootable Image
*                file format
*
*********************************************************************/
firm65XX.xXX                                /* Input file         */
-o firm65XX.hex                             /* HEX output file    */
-i                                          /* intel format       */
-memwidth 8                                 /* 8-bit memory       */
-boot                                       /* Convert all .sect  */
-bootorg     0h                             /* Boot source addr   */
-cg   3DEF4000h                             /* Global mem conf.   */
-cl   3DEF4230h                             /* Local mem conf.    */
-e    40000c00h                             /* PC after load addr */
-iack 40000000h                             /* IACK dummy read    */
-ivtp 002FFC00h                             /* IV pointer         */
-tvtp 002FFC00h                             /* TV pointer         */
```

Figure 4-1: Model 6516 - Linker Command File for COFF-HEX File Conversion

To modify the linker command file shown above to work with COFF files of your own, substitute the name of your COFF file on the **`/*Input file*/`** line, and the name you want the new HEX file to be given after the **`-o`** on the **`/*HEX output file*/`** line. Examine the map file created when your COFF file was compiled to find the entry point, and substitute that address after the **`-e`** on the **`/*PC after load addr*/`** line.

The procedure above created a HEX file, residing on your workstation, that is suitable for writing into the 6516's Global Flash EEPROM. This file can be downloaded directly to the EEPROM, as this resource is accessible to the VMEbus (in the memory region from VME_base+0x0020 0000 – VME_base+0x003F FFFF). Appendix A of this manual provides listings of two programs to write an Intel HEX file into the Global Flash EEPROM, loadhx65.c and ldflsh65.c.

DO NOT ATTEMPT TO OVERWRITE THE BOOT FLASH EEPROM ON THE MODEL 6516!

4.3 Flash EEPROM Operations (continued)

4.3.2 Flash EEPROM Sectors

The Flash EEPROM used on the Model 65xx is partitioned into eight 64 kByte sectors. If a given Flash EEPROM sector contains a valid program, then the data 0xA55A 5AA5 must appear in that sector's starting (i. e., first or lowest) address. The second location in the EEPROM contains the boot address. Table 4–1, below, lists the starting addresses of each Flash EEPROM sector.

Table 4-1: Model 6516 - Flash EEPROM Sector Start Addresses

Sector	'C40 Start Address	VME Start Address
0	0x8008 0000	0x0020 0000
1	0x8009 0000	0x0040 0000
2	0x800A 0000	0x0080 0000
3	0x800B 0000	0x00C0 0000
4	0x800C 0000	0x0120 0000
5	0x800D 0000	0x0140 0000
6	0x800E 0000	0x0180 0000
7	0x800F 0000	0x01C0 0000

4.4 Firmware Command Format and Structure

Each firmware command contains a command word, a mask specifying input channel or receiver, and an optional set of parameters. Macros are provided for building the command word. Example macros are provided for each firmware command definition starting in Section 4.5. The command word, P65XX_CMD_WORD, is defined in 65xxcmd.h and contains cmd, handle, wait_flag, data_fmt, and spare fields. Figure 4–2, below, shows the 40–bit P65XX_CMD_WORD definition as it appears in 65xxcmd.h

P65XX_CMD_WORD Bits				
D37...D29	D28...D21	D20	D19...D12	D11...D0
spare	data_fmt	wait_flag	handle	cmd
Input Channel or Receiver Mask (optional)				
Parameter 1 (optional)				
⋮				

Figure 4-2: 65XX Firmware Command Word Defined in 65xxcmd.h

4.4 Firmware Command Format and Structure (continued)

4.4.1 Cmd Field

The CMD field contains the actual executable command. All commands are listed in Section 4.5, starting on page 93.

4.4.2 Handle Field

The handle field specifies the connection affected by the command. Each receiver will usually have a separate connection handle.

4.4.3 Wait Flag

The wait flag field specifies that the processor sending the command is waiting for a reply. A reply consists of a command completion status and any return data. **When sending commands over the Comm Ports with the wait flag set to 0, return status and data are NOT returned to the caller.** When the wait flag is set using the VME Interface, a VME Interrupt is generated indicating the command is complete.

4.4.4 Data Fmt

The data_fmt field indicates the data format of all floating point parameter values associated with the command. The two parameters affected by this field are the LO (Local Oscillator Frequency) and the Input Clock Rate. All other parameters ignore this field. There are three possible values for this field; P65XX_DEFAULT_TYPE, P65XX_IEEE_FLOAT_TYPE, and P65XX_INTEGER_TYPE.

4.4.4.1 P65XX_DEFAULT_TYPE

The 'C40's Two's Complement Floating Point Format is the default data type for all floating point parameters.

4.4.4.2 P65XX_IEEE_FLOAT_TYPE

When specified, the 'C40 converts a 32 bit IEEE Float to a 'C40 Float. It is necessary to use this parameter when using IEEE Std. 754 Single Precision floating format since it is not compatible with the 'C40 floating point format.

4.4.4.3 P65XX_INTEGER_TYPE

When floating point data types aren't available, a 32–bit integer value can be specified by using this parameter.

4.4 Firmware Command Format and Structure (continued)

4.4.5 Input Channel / Receiver Mask

The Input Channel or Receiver Mask specifies which of the input channels or receivers are affected by the associated command.

The following are constants defined in 65xxcmd.h:

P65XX_INPUT_A	P65XX_INPUT_B
P65XX_INPUT_C	P65XX_INPUT_D
P65XX_RECEIVER_1	P65XX_RECEIVER_2
P65XX_RECEIVER_3	P65XX_RECEIVER_4
P65XX_RECEIVER_5	P65XX_RECEIVER_6
P65XX_RECEIVER_7	P65XX_RECEIVER_8
P65XX_RECEIVER_9	P65XX_RECEIVER_10
P65XX_RECEIVER_11	P65XX_RECEIVER_12
P65XX_RECEIVER_13	P65XX_RECEIVER_14
P65XX_RECEIVER_15	P65XX_RECEIVER_16

4.4.6 Optional Parameters

Some commands also contain a list of additional parameters which may need to be specified when using a specific command. Optional parameters are listed in the command definition.

4.5 65XX Firmware Commands

This section contains all 65XX Firmware commands. Commands specific to the Pentek Model 6516 are listed in Section 4.7. Each description of the Firmware command set includes an example code fragment showing how the command is used. However, the commands do not include the complete code needed to make the call work. The setup and initialization of arrays and pointers, for example, are not included in every example macro. The code fragment shown below shows everything that needs to be included each time a command is used.

```
unsigned long handle;              /* Not included in future examples */
unsigned long status;              /* Not included in future examples */
unsigned long numRData;            /* Not included in future examples */
unsigned long returnData[2];       /* Not included in future examples */
unsigned long cmdParams[2];        /* Not included in future examples */

#define CMD_COMM_PORT 2           /* Local Comm Port for sending commands */
```

4.5 65XX Firmware Commands (continued)

4.5.1 P65XX_OPEN_CONNECTION

Command:	**P65XX_OPEN_CONNECTION**
Input Parameters:	None
Return Data:	Handle - Identifier for this command.
Description:	Opens a connection to one or more of the receivers. A connection usually needs to be established for each receiver.

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,

P65XX_BUILD_CMD_WAIT(P65XX_OPEN_CONNECTION,0),
                  0, (void *)NULL, 0, &numRdata, returnData);
                  handle=*returnData;
```

Notes: **A connection is opened and a handle is returned.**

4.5.2 P65XX_CLOSE_CONNECTION

Command:	**P65XX_CLOSE_CONNECTION**
Input Parameters:	Handle Identifies the connection to close.
Return Data:	None
Description:	Closes an existing connection.

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,

P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_CONNECTION,
                  handle), 0, (void *)NULL, 0, NULL,NULL);
```

Notes: **The open connection corresponding to this connection is closed.**

4.5 65XX Firmware Commands (continued)

4.5.3 P65XX_CLOSE _ALL_CONNECTIONS

Command: **P65XX_CLOSE_ALL_CONNECTIONS**
Input Parameters: None
Return Data: None
Description: Closes all connections
Example:

```
status = P65XXSendCommandWait(CMD_COMM_PORT,

P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_ALL_CONNECTIONS,0),
            0, (void *)NULL, 0, NULL, NULL);
```

Notes: **All connections are closed.**

4.5.4 P65XX_RESET_REGS

Command: **P65XX_RESET_REGS**
Input Parameters: None
Return Data: None
Description: Sets all control registers to a default state.
Example:

```
status = P65XXSendCommandWait(CMD_COMM_PORT,

P65XX_BUILD_CMD_WAIT(P65XX_RESET_REGS,0),0,
                  (void *)NULL, 0, NULL, NULL);
```

Notes: **All control registers are reset to their default state.**

4.5 65XX Firmware Commands (continued)

4.5.5 P65XX_INIT_RCVRS

Command:	**P65XX_INIT_RCVRS**	
Input Parameters:	Handle	Identifies the open connection.
	Data Fmt	Specifies the data format for all floating point parameters.
	Receiver mask	Identifies the receivers to be initialized.
	Setup Parameter Structure	The P65XX_RCVR_PARAMS data structure contains the setup parameters. Each 65XX Model has a unique data structure specified in 65XX.h. XX is the specific model receiver, such as 6516.h
Return Data:	None	
Description:	Specifies all setup parameters for one or more receivers.	

Example:

```
P6516_RCVR_PARAMS p65xxRcvrParams;
          status = P65XXSendCommandWait(CMD_COMM_PORT,
          P65XX_BUILD_CMD_WAIT(P65XX_INIT_RCVRS,handle),
          SIZEOF(P65XX_RCVR_PARAMS), (void *)&p65XXRcvrParams,
          P65XX_RECEIVER_1, NULL, NULL);
```

Notes: **Receiver 1 is initalized with the settings specified in P65XX_RCVR_PARAMS structure.**

4.5 65XX Firmware Commands (continued)

4.5.6 P65XX_INIT_INPUT_CHANNELS

Command:	**P65XX_INIT_INPUT_CHANNELS**	
Input Parameters:	Input Mask	Identifies the input channels to be initialized.
	Setup Parameter Structure	The 65XX_INPUT_PARAMS data structure contains the setup parameters. Each 65XX Model has a unique data structure specified in 65XX.h. XX is the specific model receiver, such as 6516.h
Return Data:	None	
Description:	Specifies all setup parameters for the Input Channels.	

Example:

```
P6516_INPUT_PARAMS p65xxInputParms;
status = P65XXSendCommandWait(CMD_COMM_PORT,

P65XX_BUILD_CMD_WAIT(P65XX_INIT_INPUT_CHANNELS,
              0), sizeof(P65XX_INPUT_PARAMS),
              (void *)&p65xxInputParams, P65XX_INPUT_B, NULL,
              NULL);
```

Notes: **Input channel B is initalized with the settings specified in `p65XXinputParams` structure.**

4.5 65XX Firmware Commands (continued)

4.5.7 P65XX_SET_LO

Command:	**P65XX_SET_LO**	
Input Parameters:	Handle	Identifies the open connection.
	Data fmt	Specifies the data format.
	Receiver Mask	Identifies the receivers to be modified.
	LO	Local Oscillator frequency specified in floating point.
Return Data:	None	
Description:	Sets the local oscillator frequency for one or more receivers. The LO is dependent on the Input Clock Rate going into the receiver.	

Example:

```
cmdParams[0] = 24.0e6;
status = P65XXSendCommandWait (CMD_COMM_PORT,
           P65XX_BUILD_CMD_WAIT(P65XX_SET_LO, handle), 1,
           (void *)cmdParams, P65XX_RECEIVER_2, NULL, NULL);
```

Notes:

The local oscillator frequency for receiver 2 was set to 24 MHz. By performing an OR operation on the receiver masks, you can set up multiple receivers in a single command. For example;

```
... (void *) cmdParams,
    P65XX_RECEIVER_2 | P65XX_RECEIVER_4, ...
```

4.5 65XX Firmware Commands (continued)

4.5.8 P65XX_SET_DECIMATION

Command:	**P65XX_SET_DECIMATION**		
Input Parameters:	Handle	Identifies the open connection.	
	Receiver Mask	Identifies the receivers to be modified.	
	Decimation Rate	Decimation Rate is dependent on Data Mode. The 6516, 6514, 6508, 6509, and 6510 have the following range of values:	
		Real Mode -	Min: 32
			Max: 32768
		Complex Mode -	Min: 64
			Max: 65536
Return Data:	None		
Description:	Sets the decimation rate frequency for one or more receivers.		

Example:

```
cmdParams[0] = 2400;
status = P65XXSendCommandWait (CMD_COMM_PORT,
           P65XX_BUILD_CMD_WAIT(P65XX_SET_DECIMATION,
           handle), 1, (void *)cmdParams, P65XX_RECEIVER_2,
           NULL, NULL);
```

Notes: **The decimation rate of receiver 2 was set to 2400.**
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.9 P65XX_SET_INP_CLK_RATE

Command:	**P65XX_SET_INP_CLK_RATE**	
Input Parameters:	Handle	Identifies the open connection.
	Data fmt	Specifies the data format.
	Receiver Mask	Identifies the receivers to be modified.
	Input Clock Rate	Input Clock Rate specified in floating point.
Return Data:	None	
Description:	Specifies the input clock rate into the digital receiver for one or more receivers. The LO is adjusted in the digital receiver to reflect the changes in the input clock rate.	

Example:

```
cmdParams[0] = 50.0e6;
status = P65XXSendCommandWait (CMD_COMM_PORT,
           P65XX_BUILD_CMD_WAIT(P65XX_SET_INP_CLK_RATE,
           handle), 1, (void *)cmdParams, P65XX_RECEIVER_6,
           NULL, NULL);
```

Notes: **An input clock rate of 50 MHz for receiver 6 was set.**
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.10 P65XX_FLIP_SPECTRUM

Command:	**P65XX_FLIP_SPECTRUM**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Flip Spectrum	Flip Output Spectrum: P65XX_FLIP_OFF P65XX_FLIP_ON
Return Data:	None	
Description:	Inverts the output spectrum for one or more receivers. For addtional information, refer to the Harris Chip appendix, where this is refered to as Spectral Reverse (Control Register 4, Bit D0).	

Example:

```
cmdParams[0] = P65XX_FLIP_ON;
status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_FLIP_SPECTRUM, handle),
             1, (void *)cmdParams, P65XX_RECEIVER_6, NULL,
             NULL);
```

Notes:	**Output spectrum for receiver 6 is inverted.** **Supports multiple receivers using a single command.**

4.5 65XX Firmware Commands (continued)

4.5.11 P65XX_OFFSET_SPECTRUM

Command:	**P65XX_OFFSET_SPECTRUM**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Offset Spectrum	Offset Output Spectrum: P65XX_OFFSET_OFF P65XX_OFFSET_ON
Return Data:	None	
Description:	Offsets the output spectrum for one or more receivers. For addtional information, refer to the Harris Chip appendix, where this is refered to as Up Convert (Control Register 4, Bit D32). Up Convert is only applicable when using Complex Data.	

Example:

```
cmdParams[0] = P65XX_OFFSET_OFF;
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_OFFSET_SPECTRUM,
              handle), 1,(void *)cmdParams, P65XX_RECEIVER_8,
              NULL, NULL);
```

Notes:

Offset Spectrum for receiver 8 is it toggled off.
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.12 P65XX_SET_DATA_MODE

Command:	**P65XX_SET_DATA_MODE**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Data Mode	Output Data Mode: P65XX_COMPLEX P65XX_REAL
Return Data:	None	
Description:	Specifies the output data format for one or more receivers.	

Example:

```
cmdParams[0] = P65XX_COMPLEX;
status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_SET_DATA_MODE, handle),
             1, (void *)cmdParams, P65XX_RECEIVER_8, NULL, NULL);
```

Notes:

Receiver 8 is set to use complex data mode.
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.13 P65XX_SET_DATA_SRC

Command:	**P65XX_SET_DATA_SRC**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Data Source	Input Data Source: P65XX_INPUT_A P65XX_INPUT_B P65XX_INPUT_C P65XX_INPUT_D
Return Data:	None	
Description:	Specifies the input data source for one or more receivers.	

Example:

```
cmdParams[0] = P65XX_INPUT_B;
status = P65XXSendCommandWait (CMD_COMM_PORT,
            P65XX_BUILD_CMD_WAIT(P65XX_SET_DATA_SRC, handle), 1,
            (void *)cmdParams, P65XX_RECEIVER_5, NULL, NULL);
```

Notes:
Receiver 5 is setup to use input source B.
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.14 P65XX_SET_OUTPUT_SRC

Command:	**P65XX_SET_OUTPUT_SRC**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Output Source	Output Source: P65XX_FIFO P65XX_COMM_PORT
Return Data:	None	
Description:	Specifies the destination of the data for one or more receivers.	

Example:

```
cmdParams[0] = P65XX_COMM_PORT;
status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_SET_OUTPUT_SRC,
             handle), 1, (void *)cmdParams, P65XX_RECEIVER_1 |
             P65XX_RECEIVER_2 | P65XX_RECEIVER_3 |
             P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **Receivers 1 through 4 send all output to the front panel comm ports. Supports multiple receivers using a single command.**

4.5 65XX Firmware Commands (continued)

4.5.15 P65XX_SET_PACKING_MODE

Command:	**P65XX_SET_PACKING_MODE**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Packing Mode	Packing Mode: P65XX_PACKED P65XX_UNPACKED
Return Data:	None	
Description:	Specifies the output format for data being transfered over Comm Ports for one or more receivers. Data read directly from FIFO is always unpacked.	

Example:

```
cmdParams[0] = P65XX_UNPACKED;
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_SET_PACKING_MODE,
              handle), 1, (void *)cmdParams, P65XX_RECEIVER_1,
              NULL, NULL);
```

Notes: **Receiver 1 sends output to the front panel comm ports in unpacked format. Supports multiple receivers using a single command.**

4.5 65XX Firmware Commands (continued)

4.5.16 P65XX_SET_DATA_ORDER

Command:	**P65XX_SET_DATA_ORDER**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Data Order	Real Mode: P65XX_RECEIVER_ORDER P65XX_TIME_ORDER Complex Mode: P65XX_RECEIVER_ORDER P65XX_IQ_ORDER
Return Data:	None	
Description:	Specifies the data order for data being transfered over Comm Ports for one or more receivers.	

Example:

```
cmdParams[0] = P65XX_IQ_ORDER;

status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_SET_DATA_ORDER, handle),
             1, (void *)cmdParams, P65XX_RECEIVER_7, NULL, NULL);
```

Notes: **Receiver 7 will receive complex mode data in IQ order.**
Supports multiple receivers using a single command.

4.5 65XX Firmware Commands (continued)

4.5.17 P65XX_SELECT_FIFO

Command:	**P65XX_SELECT_FIFO**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Receiver	Selected FIFO: P65XX_EVEN_FIFO for receivers 3 & 4, 7 & 8, 11 & 12, 15 & 16 P65XX_ODD_FIFO for receivers 1 & 2, 5 & 6, 9 & 10, 13 & 14
Return Data:	None	
Description:	Specifies the FIFOs to be sent over the Comm Port when data is Packed IQ Data, Packed Time Ordered Data, or Unpacked Receiver Ordered data.	

Example:

```
cmdParams[0] = P65XX_EVEN_FIFO;
status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_SELECT_FIFO,
             handle), 1, (void *)cmdParams, P65XX_RECEIVER_3 |
             P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **Receiver s 3&4 data is output to comm port 1-4 out..**

4.5 65XX Firmware Commands (continued)

4.5.18 P65XX_TAG_ENABLE

Command:	**P65XX_TAG_ENABLE**	
Input Parameters:	Handle	Identifies the open connection.
	Receiver Mask	Identifies the receivers to be modified.
	Tag Enable	Tag Enable: P65XX_TAG_OFF P65XX_TAG_ON
Return Data:	None	
Description:	Specifies wheter the LSB of the data is a data bit or an indicator of I/Q data for one or more receivers.	

Example:

```
cmdParams[0] = P65XX_TAG_ON;
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_TAG_ENABLE, handle),
              1, (void *)cmdParams, P65XX_RECEIVER_3, |
              P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **Turns on the I/Q TAG bit for receivers 3&4.**

4.5 65XX Firmware Commands (continued)

4.5.19 P65XX_SET_THRESHOLD_DETECTION

Command:	**P65XX_SET_THRESHOLD_DETECTION**	
Input Parameters:	Input Mask	Identifies the input channels affected.
	Threshold Level/ Count	The Threshold Level and Count are specified in a single 32-bit word. Threshold Levels: P65XX_THRSHLD_DETECT_OFF P65XX_ONE_HALF_FSCALE P65XX_FIVE_EIGHTS_FSCALE P65XX_THREE_FOURTHS_FSCALE P65XX_SEVEN_EIGHTS_FSCALE Threshold Counts: P65XX_CROSS_CNT_2 P65XX_CROSS_CNT_8 P65XX_CROSS_CNT_32 P65XX_CROSS_CNT_128 P65XX_CROSS_CNT_512 P65XX_CROSS_CNT_2028 P65XX_CROSS_CNT_8192 P65XX_CROSS_CNT_32768
Return Data:	None	
Description:	Specifies the data input level recognized as the threshold and the number of times the threshold is crossed before the threshold detection flag is set for one or more of the input channels.	

Example:

```
cmdparam[0] = P65XX_THREE_FOURTHS_FSCALE |
P65XX_CROSS_CNT_512;
STATUS=P65XXSEND_COMMAND_WAIT(CMD_COMM_PORT,
                P65XX_BUILD_CMD_WAIT(
                P65XX_SET_THRESHOLD_DETECTION, 0), 1,
                (void *)cmdParams, P65XX_INPUT_B, NULL, NULL);
```

Notes: **The threshold detection flag will be set when channel B's input level crosses the ¾ FSCALE Threshold 512 times.**

4.5 65XX Firmware Commands (continued)

4.5.20 P65XX_RELEASE_FIFO

Command:	**P65XX_RELEASE_FIFO**	
Input Parameters:	Handle	Identifies the open connection
	Receiver Mask	Identifies the receivers to be released
Return Data:	None	
Description:	Specifies to release the FIFO for one or more receivers. FIFO release is dependent on the method used to read the data. When reading from the FIFO, the FIFOs are released independently. When reading data through the front panel Comm Port outputs, the FIFOs are released in pairs. **NOTE**: This command only places the FIFO into the released state, and does not reset the FIFO. To reset FIFO see the information in P65XX_ RESET_FIFO, or P65XX_FLUSH_FIFO.	

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
             P65XX_BUILD_CMD_WAIT(P65XX_RELEASE_FIFO, handle),
             0, (void *)NULL, P65XX_RECEIVER_1 |
             P65XX_RECEIVER_2 | P65XX_RECEIVER_3 |
             P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **FIFOs are released for receivers 1 through 4.**

4.5 65XX Firmware Commands (continued)

4.5.21 P65XX_RESET_FIFO

Command:	**P65XX_RESET_FIFO**	
Input Parameters:	Handle	Identifies the open connection
	Receiver Mask	Identifies the receivers to be reset
Return Data:	None	
Description:	Specifies to reset the FIFO for one or more receivers. FIFO reset is dependent on the method used to read the data. When reading from the FIFO, the FIFOs are reset independently. When reading data through the front panel Comm Port outputs, the FIFOs are reset in pairs. **NOTE**: This command only places the FIFO into the reset state, and does not release the FIFO. To release FIFO see the information in P65XX_ RELEASE_FIFO, or P65XX_FLUSH_FIFO.	

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
                P65XX_BUILD_CMD_WAIT(P65XX_RESET_FIFO, handle),
                0, (void *)NULL, P65XX_RECEIVER_1 |
                P65XX_RECEIVER_3, NULL, NULL);
```

Notes: FIFO is reset on receivers 1 and 3.

4.5 65XX Firmware Commands (continued)

4.5.22 P65XX_FLUSH_FIFO

Command:	**P65XX_FLUSH_FIFO**	
Input Parameters:	Handle	Identifies the open connection
	Receiver Mask	Identifies the receivers to be released
Return Data:	None	
Description:	Specifies to reset and release the FIFO for one or more receivers. FIFO flushing is dependent on the method used to read the data. When reading from the FIFO, the FIFOs are flushed independently. When reading data through the front panel Comm Port outputs, the FIFOs are flushed in pairs. **NOTE**: Combines RELEASE_FIFO and RESET_FIFO into a single command.	

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_FLUSH_FIFO, handle),
              0, (void *)NULL, P65XX_RECEIVER_5, NULL, NULL);
```

Notes: FIFO is reset and released on receiver 5.

4.5 65XX Firmware Commands (continued)

4.5.23 P65XX_GET_RCVR_STATUS

Command:	**P65XX_GET_RCVR_STATUS**
Input Parameters:	None
Return Data:	Receiver Status Register #1 Receiver Status Register #2
Description:	Returns the current value of Receiver Status Register 1 and Receiver Status Register 2.

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_GET_RCVR_STATUS, 0),
              0, (void *)NULL, NULL, &numRData, returnData);
```

Notes:	**Status Register 1 status is contained in returnData [0]. Status register 2 status is contained in returnData [1].**

4.5 65XX Firmware Commands (continued)

4.5.24 P65XX_GET_RCVR_DATA

Command:	**P65XX_GET_RCVR_DATA**	
Input Parameters:	None	
Input Parameters:	Receiver Mask	Identifies the receivers to be read.
	Communications Port	Specifies the 'C40 Comm Port indicating the destination of the data. Values 2 through 5 are valid.
	Data Type/Latency	Specifies the data type and if it is buffered or output immediately. The Type and the Latency are specified in a single 32-bit word. Data Types: P65XX_REAL_DATA *Output Real Data* P65XX_COMPLEX_DATA *Output Complex Data* Latency Types: P65XX_BUFFERED_DATA *Data is output to the Comm Port after the FIFOis half full.* P65XX_NON_BUFFERED_DATA *Data is output to the Comm Port as soon as it is available in the FIFO.*
Return Data:	None	
Description:	Multiplexes data from specified receivers and outputs the data to a single 'C40 Comm Port. **NOTE**: The format of the data sent through the Comm Ports is specified in Table 4-2 on the next page.	

Example:

```
cmdParams [0]= 5  /* 6516 Boards 13-16 IN Comm Port */
cmdParams [1]= P65XX_COMPLEX_DATA | P65XX_NONBUFFERED_DATA;
status=P65XX_SEND_COMMAND_WAIT(CMD_COMM_PORT,
                  P65XX_BUILD_CMD_WAIT(P65XX_GET_RCVR_DATA, 0),
                  2, (void *)cmdParams, P65XX_RECEIVER_1 |
                  P65XX_RECEIVER_2 |P65XX_RECEIVER_3 |
                  P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **Complex data from receivers 1, 2, 3, and 4 are output to Comm Port 5 (labled 13-16 in) as soon as it's available.**

4.5 65XX Firmware Commands (continued)

4.5.24 P65XX_GET_RCVR_DATA (continued)

Table 4-2: P65XX_GET_RCVR_DATA Comm Port Data Format

Data Mode	Receiver #	D31...D0		Notes
Real	1	I		In Real Mode, data is right justified and sign extended.
	2	I		
	...N	I		
Data Mode	**Receiver #**	**D31...D16**	**D15...D0**	**Notes**
Complex	1	I	Q	In Complex Mode, I is in upper word, Q is in lower word.
	2	I	Q	
	...N	I	Q	

4.5.25 P65XX_STOP_RCVR DATA

Command: **P65XX_STOP_RCVR_DATA**

Input Parameters: none

Return Data: none

Description: Aborts the data stream specified by the P65XX_GET_RCVR_DATA command.

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
                P65XX_BUILD_CMD_WAIT(P65XX_STOP_RCVR_DATA, 0),
                o, (void *)NULL, NULL, NULL, NULL);
```

Notes:

4.5 65XX Firmware Commands (continued)

4.5.26 P65XX_DISABLE_FIRMWARE

Command:	**P65XX_DISABLE_FIRMWARE**
Input Parameters:	none
Return Data:	none
Description:	Erases the boot sector in flash memory. When the board is reset, the firmware is no longer loaded.

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_DISABLE_FIRMWARE, 0),
              0, (void *)NULL, NULL, NULL, NULL);
```

Notes:

4.5.27 P65XX_MODIFY_INTR_MASK

Command:	**P65XX_MODIFY_INTR_MASK**	
Input Parameters:	Interrupt Command	Specifies whether the interrupt should be enabled/disabled, the method of notification, and if the interrupt should be level or edge triggered. Combine this command with following constant to achieve the desired operation: P65XX_DISABLE_INTERRUPT *Disables the interrupts specified in the second parameter.* P65XX_ENABLE_INTERRUPT *Enables the interrupts specified in the second parameter.* P65XX_GEN_VME_INTR *Generates a VME Bus Interrupt when interrupt condition occurs.* P65XX_EDGE_TRIG_INTR *Interrupts are edge triggered.* P65XX_LEVEL_TRIG_INTR *Interrupts are level triggered.* P65XX_MASK_ON_OCCURRENCE *Mask Interrupts after it occurs*

4.5 65XX Firmware Commands (continued)

4.5.27 P65XX_MODIFY_INTR_MASK (continued)

Command: **P65XX_MODIFY_INTR_MASK *(continued)***

Input Parameters: ***(continued)***	Interrupt Conditions	Defines the interrupts which are enabled or disabled. P65XX_FIFO_NOT_EMPTY_1 P65XX_FIFO_NOT_EMPTY_2 P65XX_FIFO_NOT_EMPTY_3 P65XX_FIFO_NOT_EMPTY_4 P65XX_FIFO_NOT_EMPTY_5 P65XX_FIFO_NOT_EMPTY_6 P65XX_FIFO_NOT_EMPTY_7 P65XX_FIFO_NOT_EMPTY_8 P65XX_FIFO_HALF_FULL_1 P65XX_FIFO_HALF_FULL_2 P65XX_FIFO_HALF_FULL_3 P65XX_FIFO_HALF_FULL_4 P65XX_FIFO_HALF_FULL_5 P65XX_FIFO_HALF_FULL_6 P65XX_FIFO_HALF_FULL_7 P65XX_FIFO_HALF_FULL_8 P65XX_FIFO_FULL_1 P65XX_FIFO_FULL_2 P65XX_FIFO_FULL_3 P65XX_FIFO_FULL_4 P65XX_FIFO_FULL_5 P65XX_FIFO_FULL_6 P65XX_FIFO_FULL_7 P65XX_FIFO_FULL_8 P65XX_CLOCK_NOT_READY_A P65XX_CLOCK_NOT_READY_B P65XX_CLOCK_NOT_READY_C P65XX_CLOCK_NOT_READY_D P65XX_THRSHLD_DETECT_A P65XX_THRSHLD_DETECT_B P65XX_THRSHLD_DETECT_C P65XX_THRSHLD_DETECT_D

4.5 65XX Firmware Commands (continued)

4.5.27 P65XX_MODIFY_INTR_MASK (continued)

Command: **P65XX_MODIFY_INTR_MASK *(continued)***

Description

Enables/Disables the interrupt masks for detecting FIFO Not Empty, FIFO Half Full, FIFO Full, Threshold Detection, and Clock Not Ready. This command also specifies the method of notification. One method of notification is to have the firmware generate a VME Bus Interrupt when a condition occurs. The other method is to have the firmware write the status register associated with the interrupting condition to a specified Comm Port when a condition occurs.

IMPORTANT: Level Triggered Interrupts which are not masked on occurrence can put the 'C40 in an infinite interrupt loop. For this reason, it is recommended to mask all Level Triggered Interrupts on occurrence. When in an infinite interrupt loop, the 'C40 may not respond to Comm Port Commands.

Example:

```
cmdParams [0] = P65XX_GEN_VME_INTR;
cmdParams [1] = P65XX_CLOCK_NOT_READY;
status = P65XXSendCommandWait (CMD_COMM_PORT,
              P65XX_BUILD_CMD_WAIT(P65XX_MODIFY_INTR_MASK, 0),
              2, (void *)cmdParams, NULL, NULL, NULL);
```

Notes: **Generates a VMEBus interrupt when clock A is not in a ready state.**

4.5 65XX Firmware Commands (continued)

4.5.28 P65XX_LOOK

Command: P65XX_LOOK

Input Parameters: 65XX Address

Return Data: Contents of 65XX Address.

Description: Reads a location of the 65XX.

Example:

```
cmdParams[0] = 0x80000000;
status = P65XXSendCommandWait (CMD_COMM_PORT,
               P65XX_BUILD_CMD_WAIT(P65XX_LOOK, 0), 1,
               (void *)cmdParams, NULL, &numRData, returnData);
```

Notes: returnData contains the contents of the 0x80000000 address.

4.5.29 P65XX_SET

Command: P65XX_SET

Input Parameters: 65XX Address

Data Value

Return Data: none

Description: Sets a location on the 65XX.

Example:

```
cmdParams [0] = 0x40005000;
cmdParams [1] = 0x12345678;
status = P65XXSendCommandWait (CMD_COMM_PORT,
               P65XX_BUILD_CMD_WAIT(P65XX_SET, 0), 2,
               (void *)cmdParams, NULL, &numRData, returnData);
```

Notes: Location 0x40005000 is set to 0x12345678.

4.5 65XX Firmware Commands (continued)

4.5.30 P65XX_GET_REVISION

Command:	P65XX_GET_REVISION	
Input Parameters:	none	
Return Data:	Revision	This value is the 65XX model in Bit D16 - D31 and the firmware revision in bits D0 - D15. For Example, 6516000a = Model 6516 Firmware Revision A.
Description:	Gets the current version of the firmware. The firmware revision can also be viewed at the first location in Global SRAM or at the VME Base Address in A24/A32 Space.	

Example:

```
status = P65XXSendCommandWait (CMD_COMM_PORT,
                P65XX_BUILD_CMD_WAIT(P65XX_GET_REVISION, 0),
                0, (void *)NULL, NULL, &numRData, returnData);
```

Notes: *returnData contains the current revision of the receiver.

4.6 VMEbus Specific Commands

Another method of sending commands to the 65XX is viag the VMEbus Interface. The Global SRAM is used for passing parameters, commands, and status between the VMEbus and the 'C40 processor. The Global SRAM has a 4K x 32 region partitioned into 64 command table entries, used for staging a series of commands or parameters for a specified group of channels. Each command table entry's memory location is separated from the previous command table entry by 256 bytes.

The structure of a command table entry is relatively simple: the first word in the table is a command word containing the command and some control bits. The second word specifies the input or receiver channels affected by the command. Words 3 – 64 contain parameters associated with the command. The format of the command word is defined in Section 4.4.

The 'C40 processes commands in the following manner: the 'C40 waits for the VMEbus to generate a 'C40 interrupt. Upon receiving this interrupt, the 'C40 scans the first word of each command table entry to see if the table entry needs processing. If processing is required, it executes the command, then clears the command word to indicate the command is complete. The return status is stored in the next location following the command word. Any return data is stored in the locations after the return status. If the Wait Flag was specified in the Command Word, a VMEbus interrupt is generated. After the command table is completely scanned, the 'C40 goes idle, waiting for the next VME Interrupt. Figure 4–3, illustrates a single processing cycle. This cycle has to be performed a total of 64 times.

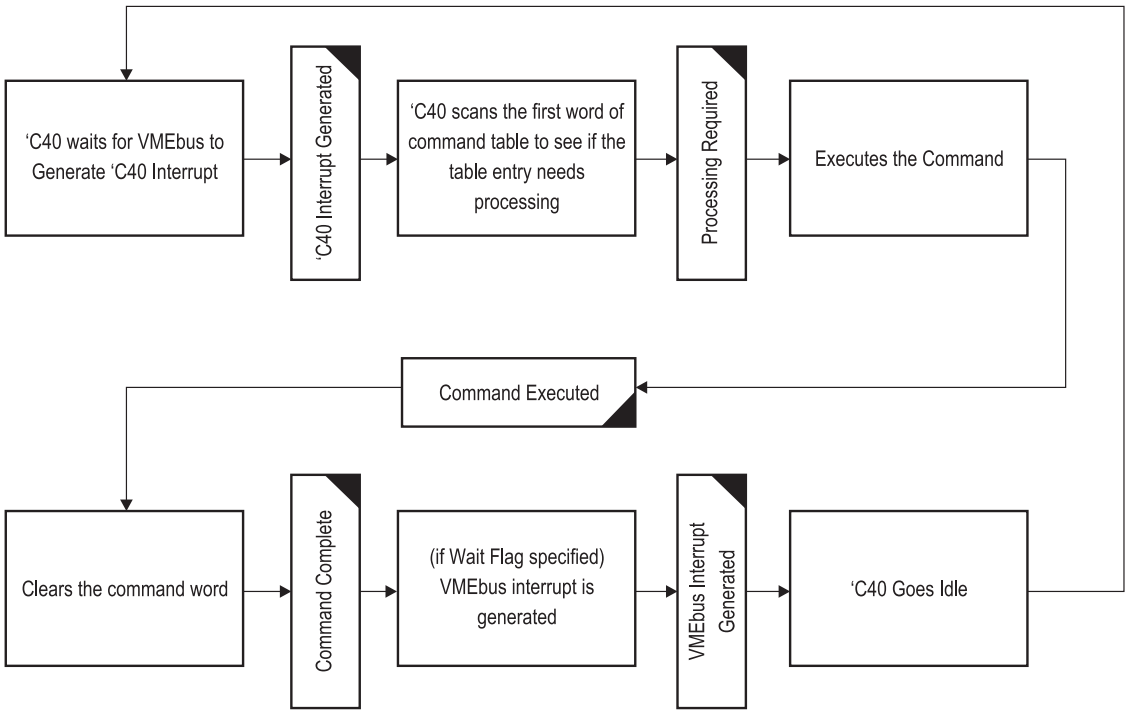


Figure 4-3: Model 6516 - Single 'C40 Command Process Cycle

4.6 VMEbus Specific Commands (continued)

Table 4-3: Commands & Addresses for the VMEbus	
Description	**Address**
Command Word	VME_base + 0xC000 + (0x100 * command table entry†)
Input Channel/Receiver Mask	VME_base + 0xC004 + (0x100 * command table entry†)
Parameter (n)	VME_base + 0xC004 + (0x4 * N) + (0x100 * command table entry†)
Examples:	Parameter 1 = VMEBase + 0xC008 Parameter 2 = VMEBase + 0xC00C
† - command table entry is equal to 0 through 63.	

After the command completes:

Table 4-4: Sending Commands to the VMEbus		
Description	**Address**	**Notes**
Command word	VME_base + 0xC000+ (0x100 * command table entry†) = 0	
Num Return Data/Status	VME_base + 0xC004 + (0x100 * command table entry†)	The Num Return Data/Status Word contains the return status of the command in bits D0 - D15 and the number of returned data words in bits D16 - D32.
Return Data (n)	VME_base + 0xC004 + (0x4 * N) + (0x100 * command table entry†)	
Examples:	Return Data 1 = VME_base + 0xC008 + (0x100 * command table entry†) Return Data 2 = VME_base + 0xC00C + (0x100 * command table entry†)	
† - command table entry is equal to 0 through 63.		

Return Status can be set to:
`P65XX_COMMAND_COMPLETE;`
`P65XX_COMMAND_ERROR;`
`P65XX_PARAM_ERROR;` or
`P65XX_OPEN_ERROR`.
These commands are described in the sub–sections that follow.

4.6 VMEBus Specific Commands (continued)

4.6.1 P65XX_COMMAND_COMPLETE

Command: **P65XX_COMMAND_COMPLETE**
Description: Command completed successfully

4.6.2 P65XX_COMMAND_ERROR

Command: **P65XX_COMMAND_ERROR**
Description: Command not recognized

4.6.3 P65XX_PARAM_ERROR

Command: **P65XX_PARAM_ERROR**
Description: Input parameters error

4.6.4 P65XX_OPEN_ERROR

Command: **P65XX_OPEN_ERROR**
Description: Error opening connection.

4.7 Model 6516 Specific Firmware Commands

4.7.1 P6516_SET_RCVR_WORD

Command:	**P6516_SET-RCVR_WORD**	
Input Parameters:	Receiver mask	Identifies the receivers to be modified
	Hi Control Word	Control Register Hi Word Bits D32 - D39
	Lo Control Word	Control Register Lo Word Bits D0 - D31
Return Data:	none	
Description:	Programs one of the HSP50016 Control Registers on one or more receivers.	

Example:

```
cmdParams [0] = 0x81;
cmdParams [1] = 0x00000000;

status = P65XXSendCommandWait (CMD_COMM_PORT,
         P65XX_BUILD_CMD_WAIT(P6516_SET_RCVR_WORD, 0), 2,
         (void *)cmdParams, P65XX_RECEIVER_5, NULL, NULL);
```

Notes:	**Set control register 4 on receiver 5 to generate complex data with up conversion.**

4.7 Model 6516 Specific Firmware Commands (continued)

4.7.2 P6516_SET_INPUT_RESET_MODE

Command:	**P6516_SET_INPUT_RESET_MODE**	
Input Parameters:	Input Channel Mask	Identifies the inputs to be modified. The setting of this word is identical to bits D1 - D3 in the Input Control Register. Combine with the following constants to achieve the desired operation:
	Input Reset Mode	P6516_SW_RESET_ON P6516_SW_RESET_OFF P6516_FP_RESET_IN_SRC P6516_SW_RESET_IN_SRC P6516_RESET_OUT_DIS P6516_RESET_OUT_EN
Return Data:	none	
Description:	Specifies the Input Reset Source for one or more input channels. This command is only applicable for the Model 6516.	

Example:

```
cmdParams [0] = P6516_SW_RESET_OFF | P6516_SW_RESET_IN_SRC |
                P6516_RESET_OUT_DIS;

status = P65XXSendCommandWait (CMD_COMM_PORT,
         P65XX_BUILD_CMD_WAIT(P6516_SET_INPUT_RESET_MODE,
         0), 1, (void *)cmdParams, P65XX_INPUT_A |
         P65XX_INPUT_B, NULL, NULL);
```

Notes: Input channels A & B Reset is de-asserted.

4.7 Model 6516 Specific Firmware Commands (continued)

4.7.3 P6516_SET_FIFO_MODE

Command:	**P6516_SET_FIFO_MODE**	
Input Parameters:	Receiver mask	Identifies the receivers to be modified.
	FIFO_Mode	FIFO Mode: P6516_SINGLE_RCVR P6516_DUAL_RCVR
Return Data:	none	
Description:	Selects the number of receivers stored in each FIFO.	

Example:

```
cmdParams [0] = P6516_DUAL_RCVR;

status = P65XXSendCommandWait (CMD_COMM_PORT,
         P65XX_BUILD_CMD_WAIT(P6516_SET_FIFO_MODE, handle),
         1, (void *)cmdParams, P65XX_RECEIVER_1 |
         P65XX_RECEIVER_2 | P65XX_RECEIVER_3 |
         P65XX_RECEIVER_4, NULL, NULL);
```

Notes: **Turns on dual receiver mode for FIFOs 1 &2.**

4.7 Model 6516 Specific Firmware Commands (continued)

4.7.4 P6516_SET_INPUT_CROSSOVER

Command:	**P6516_SET_INPUT_CROSSOVER**	
Input Parameters:	Input Crossover	Combine the following constants: P6516_CROSSOVER_A_TO_C_ON P6516_CROSSOVER_A_TO_C_OFF P6516_CROSSOVER_B_TO_D_ON P6516_CROSSOVER_B_TO_D_OFF
Return Data:	none	
Description:	Programs one of the HSP50016 Control Registers on one or more receivers.	

Example:

```
cmdParams [0] = P6516_CROSSOVER_A_TO_C_ON |
                P6516_CROSSOVER_B_TO_D_ON;

status = P65XXSendCommandWait (a16Base, vmebase,
         P65XX_BUILD_CMD_WAIT(P6516_SET_INPUT_CROSSOVER, 0),
         1, (void *)cmdParams, 0, NULL, NULL);
```

Notes: **All receivers whose selected input is channel C get channel A's input and all receivers whose selected input is channel D get channel B's input.**

This page is intentionally blank

Appendix A: Programming Examples

A.1 Model 6516 Programming Examples

This appendix includes programming examples for the Model 6516. These examples demonstrate two methods of programming the Model 6516. One method uses the Pentek Model 4285 TMS320C40 DSP board to control the receiver, while the other method programs the Model 6516 from a Single Board Computer (SBC) running VxWorks. Below are the files used in each method.

A.1.1 Programming the Model 6516 from a Model 4285

The example for programming the Model 6516 from the Model 4285 uses the following files in this appendix:

- 6516.h (code page 1)
- 65xxcmd.h (code page 6)
- vme.h (code page 13)
- 4285.h (code page 14)
- init6516.c (code page 16)
- main.c (code page 42)
- cmd85.c (code page 49)
- vme85.c (code page 52)

A.1.2 Programming the Model 6516 from a SBC Running VxWorks

The other example Pentek provides programs the Model 6516 from a Single Board Computer(SBC) running VxWorks. This example uses the following files in this appendix:

- 6516.h (code page 1)
- cmdvxw.h (code page 57)
- 65xxcmd.h (code page 6)
- main65XX.c (code page 58)
- init6516.c (code page 16)
- cmdvxw.c (code page 66)

A.1 Model 6516 Programming Examples (continued)

A.1.3 Loading Flash EEPROM from Local 'C40

To load the flash EEPROM from the local 'C40 processor, use the programming examples listed below.

- am29f040.c (code page 73)
- am29f040.h (code page 72)
- loadhx65.c (code page 76)

A.1.4 Loading Flash EEPROM from VME SBC Running VxWorks

To load the flash EEPROM from a Single Board Computer (SBC) running VxWorks, use the programming examples listed below.

- am29f040.c (code page 73)
- am29f040.h (code page 72)
- ldflsh65.c (code page 78)

```
/*****************************************************************************
*
*   File :  6516.h
*
*   Description:  This file contains the 6516 Memory Map, Register
*                 Definitions, and function prototypes.
*
*   $Revision: 2 $   $Date: 10/28/98 9:40a $
*
*   $History: 6516.h $
*
*
*****************************************************************************/

#ifndef P6516_H
#define P6516_H

/* Board Configuration Parameter Constants */
#define P6516_NUM_RCVRS            16
#define P6516_NUM_COMM_PORTS       4
#define P6516_NUM_INPUT_CHANNELS   4
#define P6516_FIFO_SIZE            4096

/* 6516 Local C40 Reg Base - This is the Local C40 Address for accessing
                             the 6516's SRAM and Registers
*/
#define P6516_C40_REG_BASE         0x80000000

/* 6516 Memory Map.  This addresses are byte offsets from the VMEBase.  To use
                     these offsets from the local C40, they must be divided
                     by 4.
*/

#define P6516_GLOBAL_SRAM          0x00000000
#define P6516_HSP_SHADOW_REG_BASE  0x0007f800
#define P6516_INTR_MASK_REG1       0x00080000
#define P6516_INTR_MASK_REG2       0x00088000
#define P6516_STATUS_REG1          0x00090000
#define P6516_STATUS_REG2          0x00098000
#define P6516_FIFO_RESET_REG       0x000A0000
#define P6516_COMM_PORT_CNTL_REG   0x000A8000
#define P6516_THRSHLD_DETECT_C     0x000B8040
#define P6516_THRSHLD_DETECT_D     0x000B80C0
#define P6516_RECV_CNTL_REG2       0x000B8140
#define P6516_INPUT_CNTL_REG       0x000B8240
#define P6516_RECV_CNTL_REG        0x000B8280
#define P6516_MODE_MUX_REG         0x000B82C0
#define P6516_TAG_ENABLE_REG       0x000B8300
#define P6516_THRSHLD_DETECT_A     0x000B8340
#define P6516_THRSHLD_DETECT_B     0x000B8380
#define P6516_RCVR_1_FIFO          0x000C0000
#define P6516_RCVR_2_FIFO          0x000C8000
#define P6516_RCVR_3_FIFO          0x000D0000
#define P6516_RCVR_4_FIFO          0x000D8000
#define P6516_RCVR_5_FIFO          0x000E0000
#define P6516_RCVR_6_FIFO          0x000E8000
#define P6516_RCVR_7_FIFO          0x000F0000
#define P6516_RCVR_8_FIFO          0x000F8000
#define P6516_INTR_ACKN_REG        0x00100000
#define P6516_VME_INTR_REG         0x00140000
#define P6516_GLOBAL_FLASH         0x00200000

/* Constants specific to each 6516 Registers are listed below */
```

```
/* Interrupt Mask Register 2 */

#define P6516_IM2_FIFO_NOT_EMPTY_1   0x00000001
#define P6516_IM2_FIFO_NOT_EMPTY_2   0x00000002
#define P6516_IM2_FIFO_NOT_EMPTY_3   0x00000004
#define P6516_IM2_FIFO_NOT_EMPTY_4   0x00000008
#define P6516_IM2_FIFO_NOT_EMPTY_5   0x00000010
#define P6516_IM2_FIFO_NOT_EMPTY_6   0x00000020
#define P6516_IM2_FIFO_NOT_EMPTY_7   0x00000040
#define P6516_IM2_FIFO_NOT_EMPTY_8   0x00000080
#define P6516_IM2_FIFO_HALF_FULL_1   0x00000100
#define P6516_IM2_FIFO_HALF_FULL_2   0x00000200
#define P6516_IM2_FIFO_HALF_FULL_3   0x00000400
#define P6516_IM2_FIFO_HALF_FULL_4   0x00000800
#define P6516_IM2_FIFO_HALF_FULL_5   0x00001000
#define P6516_IM2_FIFO_HALF_FULL_6   0x00002000
#define P6516_IM2_FIFO_HALF_FULL_7   0x00004000
#define P6516_IM2_FIFO_HALF_FULL_8   0x00008000


/* Interrupt Mask Register 1 */

#define P6516_IM1_FIFO_FULL_1        0x00000001
#define P6516_IM1_FIFO_FULL_2        0x00000002
#define P6516_IM1_FIFO_FULL_3        0x00000004
#define P6516_IM1_FIFO_FULL_4        0x00000008
#define P6516_IM1_FIFO_FULL_5        0x00000010
#define P6516_IM1_FIFO_FULL_6        0x00000020
#define P6516_IM1_FIFO_FULL_7        0x00000040
#define P6516_IM1_FIFO_FULL_8        0x00000080
#define P6516_IM1_CLOCK_NOT_READY_A  0x00000100
#define P6516_IM1_CLOCK_NOT_READY_B  0x00000400
#define P6516_IM1_CLOCK_NOT_READY_C  0x00001000
#define P6516_IM1_CLOCK_NOT_READY_D  0x00004000
#define P6516_IM1_THRSHLD_DETECT_A   0x00010000
#define P6516_IM1_THRSHLD_DETECT_B   0x00040000
#define P6516_IM1_THRSHLD_DETECT_C   0x00100000
#define P6516_IM1_THRSHLD_DETECT_D   0x00400000

/* 6516's HSP50016 Shadow Register Index's  - The Control Registers on the
   HSP50016 aren't readable from the Local C40 or VME.  The 6516 Firmware
   maintains a Shadow Register Copy of these registers to aid in debugging
   of the firmware and customer's applications.

   Each Control Word contains 40 bits of data.  The HI Word contains bits
   D32 - D39.  The LO Word contains bits D0 - D31.

*/
#define P6516_HSP_RCVR_1_SHADOW_REG    0x0
#define P6516_HSP_RCVR_2_SHADOW_REG    0x40
#define P6516_HSP_RCVR_3_SHADOW_REG    0x80
#define P6516_HSP_RCVR_4_SHADOW_REG    0xC0
#define P6516_HSP_RCVR_5_SHADOW_REG    0x100
#define P6516_HSP_RCVR_6_SHADOW_REG    0x140
#define P6516_HSP_RCVR_7_SHADOW_REG    0x180
#define P6516_HSP_RCVR_8_SHADOW_REG    0x1c0
#define P6516_HSP_RCVR_9_SHADOW_REG    0x200
#define P6516_HSP_RCVR_10_SHADOW_REG   0x240
#define P6516_HSP_RCVR_11_SHADOW_REG   0x280
#define P6516_HSP_RCVR_12_SHADOW_REG   0x2c0
#define P6516_HSP_RCVR_13_SHADOW_REG   0x300
#define P6516_HSP_RCVR_14_SHADOW_REG   0x340
#define P6516_HSP_RCVR_15_SHADOW_REG   0x380
#define P6516_HSP_RCVR_16_SHADOW_REG   0x3c0
```

```
/* Offsets to each of the Control Registers */
#define P6516_HSP_CONTROL_WORD_0_HI    0
#define P6516_HSP_CONTROL_WORD_0_LO    1
#define P6516_HSP_CONTROL_WORD_1_HI    2
#define P6516_HSP_CONTROL_WORD_1_LO    3
#define P6516_HSP_CONTROL_WORD_2_HI    4
#define P6516_HSP_CONTROL_WORD_2_LO    5
#define P6516_HSP_CONTROL_WORD_3_HI    6
#define P6516_HSP_CONTROL_WORD_3_LO    7
#define P6516_HSP_CONTROL_WORD_4_HI    8
#define P6516_HSP_CONTROL_WORD_4_LO    9
#define P6516_HSP_CONTROL_WORD_5_HI    10
#define P6516_HSP_CONTROL_WORD_5_LO    11
#define P6516_HSP_CONTROL_WORD_6_HI    12
#define P6516_HSP_CONTROL_WORD_6_LO    13
#define P6516_HSP_CONTROL_WORD_7_HI    14
#define P6516_HSP_CONTROL_WORD_7_LO    15

/* P6516_RCVR_PARAMS - This parameter structure contains all the
                       settings specific to setting up a receiver.
*/
typedef struct P6516_RCVR_PARAMS
        {
            unsigned long decimation_rate;
            unsigned long data_mode;
            unsigned long data_source;
            unsigned long packing_mode;
            unsigned long data_order;
            unsigned long data_transfer;
            unsigned long fifo_select;
            unsigned long tag_enable;
            unsigned long flip_spectrum;
            unsigned long offset_spectrum;
            unsigned long wait_ram_full;
            unsigned long fifo_mode;
            float         lo;
            float         input_clock_rate;
            unsigned long spare[5];
        } P6516_RCVR_PARAMS;

/* P6516_INPUT_PARAMS - This structure defines parameters specific to
                        Input Channels.
*/
typedef struct P6516_INPUT_PARAMS
        {
            unsigned long threshold_level;
            unsigned long threshold_cross_cnt;
            unsigned long input_crossover;
            unsigned long input_reset_mode;
        } P6516_INPUT_PARAMS;

/* P6516_REG_ADDR - This structure contains the 6516's register addresses.
*/
typedef struct P6516_REG_ADDR
        {
            volatile unsigned long *Int_Mask_1;
            volatile unsigned long *Int_Mask_2;
            volatile unsigned long *Stat_Reg_1;
            volatile unsigned long *Stat_Reg_2;
            volatile unsigned long *Fifo_Reset;
            volatile unsigned long *Out_Comm_Ctl;
            volatile unsigned long *Rcvr_Ctl;
            volatile unsigned long *Rcvr_Ctl_2;
```

```
            volatile unsigned long *Input_Ctl;
            volatile unsigned long *Mode_Mux;
            volatile unsigned long *Tag_En;
            volatile unsigned long *Thrshld_Detect_A;
            volatile unsigned long *Thrshld_Detect_B;
            volatile unsigned long *Thrshld_Detect_C;
            volatile unsigned long *Thrshld_Detect_D;
            volatile unsigned long *Fifo;
            volatile unsigned long *Intr_Ackn_Addr;
            volatile unsigned long *VME_Intr_Reg;
            volatile unsigned long *HSP_Shadow_Reg_Base;
        } P6516_REG_ADDR;

/* Constants used for updating the Harris Rcvr Chips with new setup
   parameters
*/
#define P65XX_NO_REG_UPDATE    0
#define P65XX_REG_UPDATE       1

/* 6516 Function Prototypes */
void P6516InitRegAddr(unsigned long regBaseAddr, P6516_REG_ADDR *p6516Reg);
void P6516InitRcvrRegs(P6516_RCVR_PARAMS *rcvr_params,
                       P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516InitInputRegs(P6516_INPUT_PARAMS *input_params,
                        P6516_REG_ADDR *p6516Reg, unsigned int input_mask);
void P6516ResetRegs(P6516_REG_ADDR *p6516Reg);
void P6516SetRcvrDefaults(P6516_RCVR_PARAMS *rcvr_params);
void P6516SetInputDefaults(P6516_INPUT_PARAMS *input_params);
void P6516SelectInputSource(P6516_RCVR_PARAMS *rcvr_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516SetInputCrossover(P6516_INPUT_PARAMS *input_params,
                            P6516_REG_ADDR *p6516Reg);
void P6516SelectDataMode(P6516_RCVR_PARAMS *rcvr_params,
                         P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516SelectDataFormat(P6516_RCVR_PARAMS *rcvr_params,
                           P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516SelectTagEnable(P6516_RCVR_PARAMS *rcvr_params,
                          P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516SetCommPortToken(P6516_REG_ADDR *p6516Reg);
void P6516SetDataTransfer(P6516_RCVR_PARAMS *rcvr_params,
                          P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask);
void P6516ResetFifo(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                    unsigned int rcvr_mask);
void P6516ReleaseFifo(P6516_RCVR_PARAMS *rcvr_params,
                      P6516_REG_ADDR *p6516Reg,
                      unsigned int rcvr_mask);
void P6516SetFifoMode(P6516_RCVR_PARAMS *rcvr_params,
                      P6516_REG_ADDR *p6516Reg,
                      unsigned int rcvr_mask);
void P6516SetIntrMask1(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask);
void P6516SetIntrMask2(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask);
void P6516ClearIntrMask1(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask);
void P6516ClearIntrMask2(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask);
void P6516SetLo(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                unsigned int rcvr_mask, unsigned long update_reg);
void P6516SetThresholdDetection(P6516_INPUT_PARAMS *input_params,
                                P6516_REG_ADDR *p6516Reg,
                                unsigned int input_mask);
void P6516SetThresholdLED(P6516_REG_ADDR *p6516Reg,
                          unsigned int input_mask,
                          unsigned long on_off_flag);
void P6516SetDecimationRate(P6516_RCVR_PARAMS *rcvr_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask,
                            unsigned long update_reg);
```

```
void P6516InitRcvr(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                   unsigned int rcvr_mask, unsigned long update_reg);
void P6516InitAllRcvr(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                      unsigned int rcvr_mask, unsigned long update_reg);
void P6516SetRcvrWord(P6516_REG_ADDR *p6516Reg, unsigned long HAHi,
                      unsigned long HALow, register unsigned int rcvr_mask);
void P6516ResetReceiver(P6516_REG_ADDR *p6516Reg, unsigned int input_mask);
void P6516ReleaseReceiver(P6516_REG_ADDR *p6516Reg, unsigned int input_mask);
void P6516SetInputResetMode(P6516_INPUT_PARAMS *input_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int input_mask);
unsigned int P6516DetermineHDFDataShift(unsigned int dec_factor);
double P6516DetermineScaleMult(unsigned int dec_factor);
int *P6516ReadFifoCmplx(P6516_RCVR_PARAMS *rcvr_params,
                        P6516_REG_ADDR *p6516Reg,
                        register volatile int *samples,
                        register int num_samples, unsigned int inputFifo);
int *P6516ReadCommPortCmplx(P6516_RCVR_PARAMS *rcvr_params,
                            register volatile int *samples,
                            register int num_samples,
                            unsigned int inputCommPort);
void P6516UnpackCmplxData(P6516_RCVR_PARAMS *rcvr_params,
                          register volatile int *inSamples,
                          register volatile int *outSamples1,
                          register volatile int *outSamples2,
                          register volatile int *outSamples3,
                          register volatile int *outSamples4,
                          register int num_samples);
int *P6516ReadFifoReal(P6516_RCVR_PARAMS *rcvr_params,
                       P6516_REG_ADDR *p6516Reg,
                       register volatile int *samples,
                       register int num_samples, unsigned int inputFifo);
int *P6516ReadCommPortReal(P6516_RCVR_PARAMS *rcvr_params,
                           register volatile int *samples,
                           register int num_samples,
                           unsigned int inputCommPort);
void P6516UnpackRealData(P6516_RCVR_PARAMS *rcvr_params,
                         register volatile int *inSamples,
                         register volatile int *outSamples1,
                         register volatile int *outSamples2,
                         register volatile int *outSamples3,
                         register volatile int *outSamples4,
                         register int num_samples);


#endif
```

```
/*****************************************************************************
*
*   File : 65xxcmd.h
*
*   Description:
*      This file defines constants used to communicate with the 65XX
*      using their Communication Ports or via the VMEBus Interface.
*
*****************************************************************************/

#ifndef P65XX_CMD_H
#define P65XX_CMD_H

/* Command Word Format */
typedef union P65XX_CMD_WORD
        {
            struct
            {
                unsigned int  cmd                :12;
                unsigned int  handle             :8;
                unsigned int  wait_flag          :1;
                unsigned int  data_fmt           :2;
                unsigned int  spare              :9;
            } fields;
            unsigned long cmd_word;
        } P65XX_CMD_WORD;

/* Command Word Macros */
#define P65XX_BUILD_CMD(cmd,handle) (cmd | (handle << 12) |\
                                     P65XX_NO_WAIT_COMPLETE | P65XX_DEFAULT_TYPE)
#define P65XX_BUILD_CMD_WAIT(cmd,handle) (cmd | (handle << 12) |\
                                          P65XX_WAIT_COMPLETE |\
                                          P65XX_DEFAULT_TYPE)
#define P65XX_BUILD_CMD_IEEE(cmd,handle,wait_flag) (cmd | (handle << 12) |\
                                                    wait_flag |\
                                                    P65XX_IEEE_FLOAT_TYPE)
#define P65XX_BUILD_CMD_INTEGER(cmd,handle,wait_flag) (cmd | (handle << 12) |\
                                                       wait_flag |\
                                                       P65XX_INTEGER_TYPE)
/* Command Word Masks */
#define P65XX_CMD_MASK                0x00000FFFL
#define P65XX_HANDLE_MASK             0x000FF000L
#define P65XX_WAIT_FLAG_MASK          0x00100000L
#define P65XX_DATA_FMT_MASK           0x00600000L

/* Wait Flag Constants */
#define P65XX_NO_WAIT_COMPLETE        0x00000000L
#define P65XX_WAIT_COMPLETE           0x00100000L

/* Data Format Constants */
#define P65XX_DEFAULT_TYPE            0x00000000L
#define P65XX_IEEE_FLOAT_TYPE         0x00200000L
#define P65XX_INTEGER_TYPE            0x00400000L

/* VME Command Table - In Dual Port SRAM, a command table has been setup
   for passing parameters, commands, and status between the VMEBus and
   the C40 Processor.  The Command Table Space is divided into 64 Command
   Table Entries.  Each Command Table Entry is 64 Words in Length.
*/

/* Offset from the start of Global SRAM where commands are written from VME.
   This is a C40 Offset.
   The actual VMEBus offset is P65XX_VME_COMMAND_ADDR * 4.
*/
```

```
#define P65XX_VME_COMMAND_ADDR         0x00003000L

/* Number of Command Table Entries */
#define P65XX_NUM_CMD_TBL_ENTRIES      64

/* Size of each Command Table Entry in 32 Bit Words.  From the VMEBus,     */
/* the length of each Command Table Entry is P65XX_CMD_TBL_ENTRY_SIZE * 4. */
#define P65XX_CMD_TBL_ENTRY_SIZE       64

/* Constant defining maximum number of Handles or Open Connections */
#define P65XX_NUM_DEV_HANDLES          16

/* Constant defining the maximum number of commands */
#define P65XX_NUM_COMMANDS             120

/* 65XX Commands  */
#define P65XX_OPEN_CONNECTION          1
#define P65XX_CLOSE_CONNECTION         2
#define P65XX_CLOSE_ALL_CONNECTIONS    3
#define P65XX_RESET_REGS               4
#define P65XX_INIT_RCVRS               5
#define P65XX_INIT_INPUT_CHANNELS      6
#define P65XX_SET_LO                   7
#define P65XX_SET_DECIMATION           8
#define P65XX_SET_INP_CLK_RATE         9
#define P65XX_FLIP_SPECTRUM            10
#define P65XX_OFFSET_SPECTRUM          11
#define P65XX_SET_DATA_MODE            12
#define P65XX_SET_DATA_SRC             13
#define P65XX_SET_OUTPUT_SRC           14
#define P65XX_SET_PACKING_MODE         15
#define P65XX_SET_DATA_ORDER           16
#define P65XX_SELECT_RECEIVER          17   /* Provided for backwards capability */
                                            /* P65XX_SELECT_FIFO is preferred    */
#define P65XX_TAG_ENABLE               18
#define P65XX_SET_THRESHOLD_DETECTION  19
#define P65XX_RESET_FIFO               20
#define P65XX_RELEASE_FIFO             21
#define P65XX_FLUSH_FIFO               22
#define P65XX_GET_RCVR_STATUS          23
#define P65XX_GET_RCVR_DATA            24
#define P65XX_STOP_RCVR_DATA           25
#define P65XX_MODIFY_INTR_MASK         26
#define P65XX_SELECT_FIFO              27

#define P65XX_DISABLE_FIRMWARE         41
#define P65XX_LOOK                     42
#define P65XX_SET                      43
#define P65XX_GET_REVISION             44

/* 6504 Specific Commands */
#define P6504_SET_INPUT_SYNC_MODE      51
#define P6504_SET_RCVR_SYNC_MODE       52

/* 6505 Specific Commands */

/* 6508 Specific Commands */
#define P6508_SET_INPUT_RESET_MODE     71
#define P6508_SET_RCVR_WORD            72

/* 6509 Specific Commands */
#define P6509_SET_INPUT_SYNC_MODE      81
#define P6509_SET_RCVR_SYNC_MODE       82
#define P6509_SET_FILTER_RSP           83
```

```
/* 6510 Specific Commands */
#define P6510_SET_INPUT_RESET_MODE     91
#define P6510_SET_RCVR_WORD            92
#define P6510_SET_INPUT_CROSSOVER      93

/* 6514 Specific Commands */
#define P6514_SET_INPUT_RESET_MODE     101
#define P6514_SET_RCVR_WORD            102
#define P6514_SET_FIFO_MODE            103

/* 6516 Specific Commands */
#define P6516_SET_INPUT_RESET_MODE     111
#define P6516_SET_RCVR_WORD            112
#define P6516_SET_FIFO_MODE            113
#define P6516_SET_INPUT_CROSSOVER      114

/* The second parameter of each command is the input channel/receiver mask.
   The constants used to define the receivers or input channels are defined
   as:
*/

/* Receiver Channels */
#define P65XX_RECEIVER_1        0x01
#define P65XX_RECEIVER_2        0x02
#define P65XX_RECEIVER_3        0x04
#define P65XX_RECEIVER_4        0x08
#define P65XX_RECEIVER_5        0x10
#define P65XX_RECEIVER_6        0x20
#define P65XX_RECEIVER_7        0x40
#define P65XX_RECEIVER_8        0x80
#define P65XX_RECEIVER_9        0x100
#define P65XX_RECEIVER_10       0x200
#define P65XX_RECEIVER_11       0x400
#define P65XX_RECEIVER_12       0x800
#define P65XX_RECEIVER_13       0x1000
#define P65XX_RECEIVER_14       0x2000
#define P65XX_RECEIVER_15       0x4000
#define P65XX_RECEIVER_16       0x8000

/* Input Channels  */

/* Data Source - This Selection determines which input connector   */
/* contains the clock and data.                                    */
#define P65XX_INPUT_A           0x01
#define P65XX_INPUT_B           0x02
#define P65XX_INPUT_C           0x04
#define P65XX_INPUT_D           0x08

/* Data Modes */
#define P65XX_COMPLEX           0
#define P65XX_REAL              1

/* Packing Modes */
#define P65XX_PACKED            0
#define P65XX_UNPACKED          1

/* Data Order */
#define P65XX_RECEIVER_ORDER    0
#define P65XX_IQ_ORDER          1          /* Complex Data Only */
#define P65XX_TIME_ORDER        1          /* Real Data Only    */

/* Channel Select - These constants are provided for backwards capatibility */
/*                  Fifo Select Constants have replaced this function      */
```

```
#define P65XX_EVEN_RECEIVER     0          /* Receiver 2, 4, 6, 8 */
#define P65XX_ODD_RECEIVER      1          /* Receiver 1, 3, 5, 7 */

/* Fifo Select */
#define P65XX_EVEN_FIFO         0          /* Fifo 2, 4, 6, 8 */
#define P65XX_ODD_FIFO          1          /* Fifo 1, 3, 5, 7 */

/* Flip Spectrum or Spectral Reversal */
#define P65XX_FLIP_OFF          0
#define P65XX_FLIP_ON           1

/* Offset Spectrum or Up Convert */
#define P65XX_OFFSET_OFF        0
#define P65XX_OFFSET_ON         1

/* Data Transfer Modes */
#define P65XX_FIFO              0
#define P65XX_COMM_PORT         1

/* Tag Enable */
#define P65XX_TAG_OFF           0
#define P65XX_TAG_ON            1

/* Wait on Ram Full - 6508/6510 Only */
#define P65XX_WAIT_RAM_FULL_OFF 0
#define P65XX_WAIT_RAM_FULL_ON  1

/* Threshold Level */
#define P65XX_THRSHLD_DETECT_OFF     0x00
#define P65XX_ONE_HALF_FSCALE        0x80
#define P65XX_FIVE_EIGHTS_FSCALE     0xA0
#define P65XX_THREE_FOURTHS_FSCALE   0xC0
#define P65XX_SEVEN_EIGHTS_FSCALE    0xE0

/* Threshold Crossing Count */
#define P65XX_CROSS_CNT_2            0
#define P65XX_CROSS_CNT_8            1
#define P65XX_CROSS_CNT_32           2
#define P65XX_CROSS_CNT_128          3
#define P65XX_CROSS_CNT_512          4
#define P65XX_CROSS_CNT_2048         5
#define P65XX_CROSS_CNT_8192         6
#define P65XX_CROSS_CNT_32768        7

/* Interrupt Command - Parameter 1 */

#define P65XX_DISABLE_INTERRUPT      0x0
#define P65XX_GEN_CP_2_OUTPUT        0x2
#define P65XX_GEN_CP_3_OUTPUT        0x3
#define P65XX_GEN_CP_4_OUTPUT        0x4
#define P65XX_GEN_CP_5_OUTPUT        0x5
#define P65XX_GEN_VME_INTR           0x10
#define P65XX_EDGE_TRIG_INTR         0x1000
#define P65XX_LEVEL_TRIG_INTR        0x0000
#define P65XX_ENABLE_INTERRUPT       0x4000
#define P65XX_MASK_ON_OCCURRENCE     0x8000

/* Interrupt Command Conditions - Parameter 2 */
#define P65XX_FIFO_NOT_EMPTY_1       0x1L
#define P65XX_FIFO_NOT_EMPTY_2       0x2L
#define P65XX_FIFO_NOT_EMPTY_3       0x4L
#define P65XX_FIFO_NOT_EMPTY_4       0x8L
#define P65XX_FIFO_NOT_EMPTY_5       0x10L
#define P65XX_FIFO_NOT_EMPTY_6       0x20L
```

```
#define P65XX_FIFO_NOT_EMPTY_7       0x40L
#define P65XX_FIFO_NOT_EMPTY_8       0x80L
#define P65XX_FIFO_HALF_FULL_1       0x100L
#define P65XX_FIFO_HALF_FULL_2       0x200L
#define P65XX_FIFO_HALF_FULL_3       0x400L
#define P65XX_FIFO_HALF_FULL_4       0x800L
#define P65XX_FIFO_HALF_FULL_5       0x1000L
#define P65XX_FIFO_HALF_FULL_6       0x2000L
#define P65XX_FIFO_HALF_FULL_7       0x4000L
#define P65XX_FIFO_HALF_FULL_8       0x8000L
#define P65XX_FIFO_FULL_1            0x10000L
#define P65XX_FIFO_FULL_2            0x20000L
#define P65XX_FIFO_FULL_3            0x40000L
#define P65XX_FIFO_FULL_4            0x80000L
#define P65XX_FIFO_FULL_5            0x100000L
#define P65XX_FIFO_FULL_6            0x200000L
#define P65XX_FIFO_FULL_7            0x400000L
#define P65XX_FIFO_FULL_8            0x800000L
#define P65XX_CLOCK_NOT_READY_A      0x1000000L
#define P65XX_CLOCK_NOT_READY_B      0x2000000L
#define P65XX_CLOCK_NOT_READY_C      0x4000000L
#define P65XX_CLOCK_NOT_READY_D      0x8000000L
#define P65XX_THRSHLD_DETECT_A       0x10000000L
#define P65XX_THRSHLD_DETECT_B       0x20000000L
#define P65XX_THRSHLD_DETECT_C       0x40000000L
#define P65XX_THRSHLD_DETECT_D       0x80000000L

/* Receiver Data Type and Latency */
#define P65XX_BUFFERED_DATA          0x1
#define P65XX_NON_BUFFERED_DATA      0x2
#define P65XX_COMPLEX_DATA           0x0
#define P65XX_REAL_DATA              0x100

/* 6508 Specific Constants */

/* Input Channel Reset Modes */
#define P6508_SW_RESET_ON            0x0
#define P6508_SW_RESET_OFF           0x2
#define P6508_FP_RESET_IN_SRC        0x0
#define P6508_SW_RESET_IN_SRC        0x4
#define P6508_RESET_OUT_DIS          0x0
#define P6508_RESET_OUT_ENA          0x8

/* 6509 Specific Constants */

/* Input Channel Sync Modes */
#define P6509_SW_AS_ON               0x0
#define P6509_SW_AS_OFF              0x1
#define P6509_SW_SS_ON               0x0
#define P6509_SW_SS_OFF              0x2
#define P6509_FP_SYNC_SRC            0x0
#define P6509_SW_SYNC_SRC            0x4
#define P6509_SYNC_OUT_DIS           0x0
#define P6509_SYNC_OUT_ENA           0x8

/* Bits of Input Control Register as in page 47 of the 6504 manual. */

/* Input Channel Sync Modes */
#define P6504_SW_AS_ON               0x0
#define P6504_SW_AS_OFF              0x1
#define P6504_SW_SS_ON               0x0
#define P6504_SW_SS_OFF              0x2
#define P6504_SW_GS_ON               0x0
#define P6504_SW_GS_OFF              0x4
```

```
#define P6504_FP_SYNC_SRC            0x0
#define P6504_SW_SYNC_SRC            0x8
#define P6504_SYNC_OUT_DIS           0x0
#define P6504_SYNC_OUT_ENA           0x10

/* Receiver Channel Sync Modes - The description for these bits are defined
                                 in the GC1011A Data Sheet
*/
#define P6509_GC_SS_OFF              0x01
#define P6509_GC_AS_ON               0x02
#define P6509_GC_AS_MUX              0x04
#define P6509_GC_LD_FREQ             0x08
#define P6509_GC_AS_FREQ             0x10
#define P6509_GC_SS_DIAG             0x20
#define P6509_GC_SYNC_TEST           0x40
#define P6509_GC_SS_MUX              0x80

/* Filter Selection */
#define P6509_80_PERCENT             0x00
#define P6509_90_PERCENT             0x01

/* Receiver 6504 Channel Sync Modes - The description for these bits
   is in the 6504 manual page B-10. */
#define P6504_GC_SS_OFF              0x01
#define P6504_GC_AS_ON               0x02
#define P6504_GC_AS_MUX              0x04
#define P6504_GC_LD_FREQ             0x08
#define P6504_GC_AS_FREQ             0x10
#define P6504_GC_SS_DIAG             0x20
#define P6504_GC_SYNC_TEST           0x40
#define P6504_GC_SS_MUX              0x80

/* 6510 Specific Constants */

/* Input Channel Reset Modes */
#define P6510_SW_RESET_ON            0x0
#define P6510_SW_RESET_OFF           0x2
#define P6510_FP_RESET_IN_SRC        0x0
#define P6510_SW_RESET_IN_SRC        0x4
#define P6510_RESET_OUT_DIS          0x0
#define P6510_RESET_OUT_ENA          0x8

/* Input Crossover */
#define P6510_CROSSOVER_A_TO_C_OFF   0x00000000L
#define P6510_CROSSOVER_A_TO_C_ON    0x00100000L
#define P6510_CROSSOVER_B_TO_D_OFF   0x00000000L
#define P6510_CROSSOVER_B_TO_D_ON    0x10000000L

/* 6514 Specific Constants */

/* Input Channel Reset Modes */
#define P6514_SW_RESET_ON            0x0
#define P6514_SW_RESET_OFF           0x2
#define P6514_FP_RESET_IN_SRC        0x0
#define P6514_SW_RESET_IN_SRC        0x4
#define P6514_RESET_OUT_DIS          0x0
#define P6514_RESET_OUT_ENA          0x8

/* Fifo Modes */
#define P6514_SINGLE_RCVR            0
#define P6514_DUAL_RCVR              1

/* 6516 Specific Constants */
```

```
/* Input Channel Reset Modes */
#define P6516_SW_RESET_ON            0x0
#define P6516_SW_RESET_OFF           0x2
#define P6516_FP_RESET_IN_SRC        0x0
#define P6516_SW_RESET_IN_SRC        0x4
#define P6516_RESET_OUT_DIS          0x0
#define P6516_RESET_OUT_ENA          0x8

/* Input Crossover */
#define P6516_CROSSOVER_A_TO_C_OFF   0x00000000L
#define P6516_CROSSOVER_A_TO_C_ON    0x00100000L
#define P6516_CROSSOVER_B_TO_D_OFF   0x00000000L
#define P6516_CROSSOVER_B_TO_D_ON    0x10000000L

/* Fifo Modes */
#define P6516_SINGLE_RCVR            0
#define P6516_DUAL_RCVR              1

/* 65XX Command Status */
#define P65XX_COMMAND_COMPLETE       0
#define P65XX_COMMAND_ERROR          1
#define P65XX_PARAM_ERROR            2
#define P65XX_OPEN_ERROR             3

/* 65XX Command Function Prototypes */
unsigned long P65XXSendCommand(unsigned int comm_port, unsigned int command,
                               unsigned int num_params, void *cmdParams,
                               unsigned int input_rcvr_mask);
unsigned long P65XXSendCommandWait(unsigned int comm_port, unsigned int command,
                                   unsigned int num_params, void *cmdParams,
                                   unsigned int input_rcvr_mask,
                                   unsigned long *numRData,
                                   unsigned long *returnData);
void P65XXReadCommPortData(unsigned int data_port,
                           register volatile int *insamples,
                           register unsigned long num_samples);
void P65XXUnpackCmplxData(register int *inSamples, register int *outSamples,
                          register unsigned long num_samples,
                          register unsigned int rcvr_index,
                          register unsigned int numRcvrs);
void P65XXUnpackRealData(register int *inSamples, register int *outSamples,
                         register unsigned long num_samples,
                         register unsigned int rcvr_index,
                         register unsigned int numRcvrs);


#endif
```

```
/*****************************************************************************
*
*   File : vme.h
*
*   Description: This include file contains constants specific to the
*                VMEBus.
*
*****************************************************************************/
#ifndef VME_H /* only look at this file once */
#define VME_H

/* VME Address Modifier Codes */
/*  A24 modes                 */
#define VME_AM_A24_SUP_BLT 0x3f   /* Supervisory block transfer */
#define VME_AM_A24_SUP_PRG 0x3e   /* Supervisory program        */
#define VME_AM_A24_SUP_DAT 0x3d   /* Supervisory data           */
#define VME_AM_A24_NPR_BLT 0x3b   /* User block transfer        */
#define VME_AM_A24_NPR_PRG 0x3a   /* User program               */
#define VME_AM_A24_NPR_DAT 0x39   /* User data                  */

/* A16 modes                 */
#define VME_AM_A16_SUP_ACC 0x2d   /* Supervisory access         */
#define VME_AM_A16_NPR_ACC 0x29   /* User access                */

/* A32 modes                 */
#define VME_AM_A32_SUP_BLT 0x0f   /* Supervisory block transfer   */
#define VME_AM_A32_SUP_PRG 0x0e   /* Supervisory program          */
#define VME_AM_A32_SUP_DAT 0x0d   /* Supervisory data             */
#define VME_AM_A32_NPR_BLT 0x0b   /* User block transfer          */
#define VME_AM_A32_NPR_PRG 0x0a   /* User program                 */
#define VME_AM_A32_NPR_DAT 0x09   /* User data                    */


/* Address Modifier Mask - These constants are used to differentiate
                           one address modifier from another
*/
#define VME_AM_MASK        0x30
#define VME_A16_AM         0x20
#define VME_A24_AM         0x30
#define VME_A32_AM         0x00

#endif /* __VME_DEFS_H_ */
```

```
/*****************************************************************************
*
*   File : 4285.h
*
*   Description: This include file contains contants, macros, and
*                memory map specific to the 4285 MIX Baseboard.
*
*****************************************************************************/
#ifndef P4285_H
#define P4285_H

#define P4285_LCL_CNTL_REG   ((volatile unsigned int *)0x380000)

/* Before the P4285_LCL_CNTL_REG is modified for a VME BUS Page Address,
   Address Modifier, or Function Code Change, a access needs to be made
   to Global Address Space
*/
#define P4285_DUMMY_GLOBAL_ACCESS ((volatile unsigned int *)0xc0000000L)


/* 4285 Processor ID's */
#define P4285_PROC_A        0x0
#define P4285_PROC_B        0x1
#define P4285_PROC_C        0x2
#define P4285_PROC_D        0x3
#define P4285_PROC_E        0x4
#define P4285_PROC_F        0x5
#define P4285_PROC_G        0x6
#define P4285_PROC_H        0x7


/* 4285 Vic Registers */
#define P4285_VIC_REG_BASE  ((volatile unsigned int *)0xA8000000L)

#define P4285_VIC_VIICR     P4285_VIC_REG_BASE
#define P4285_VIC_VICR1     (P4285_VIC_REG_BASE + 0x1)
#define P4285_VIC_VICR2     (P4285_VIC_REG_BASE + 0x2)
#define P4285_VIC_VICR3     (P4285_VIC_REG_BASE + 0x3)
#define P4285_VIC_VICR4     (P4285_VIC_REG_BASE + 0x4)
#define P4285_VIC_VICR5     (P4285_VIC_REG_BASE + 0x5)
#define P4285_VIC_VICR6     (P4285_VIC_REG_BASE + 0x6)
#define P4285_VIC_VICR7     (P4285_VIC_REG_BASE + 0x7)
#define P4285_VIC_DMASICR   (P4285_VIC_REG_BASE + 0x8)
#define P4285_VIC_LICR1     (P4285_VIC_REG_BASE + 0x9)
#define P4285_VIC_LICR2     (P4285_VIC_REG_BASE + 0xA)
#define P4285_VIC_LICR3     (P4285_VIC_REG_BASE + 0xB)
#define P4285_VIC_LICR4     (P4285_VIC_REG_BASE + 0xC)
#define P4285_VIC_LICR5     (P4285_VIC_REG_BASE + 0xD)
#define P4285_VIC_LICR6     (P4285_VIC_REG_BASE + 0xE)
#define P4285_VIC_LICR7     (P4285_VIC_REG_BASE + 0xF)
#define P4285_VIC_ICGSICR   (P4285_VIC_REG_BASE + 0x10)
#define P4285_VIC_ICMSICR   (P4285_VIC_REG_BASE + 0x11)
#define P4285_VIC_EGSICR    (P4285_VIC_REG_BASE + 0x12)
#define P4285_VIC_ICGSIVBR  (P4285_VIC_REG_BASE + 0x13)
#define P4285_VIC_ICMSIVBR  (P4285_VIC_REG_BASE + 0x14)
#define P4285_VIC_LIVBR     (P4285_VIC_REG_BASE + 0x15)
#define P4285_VIC_EGIVBR    (P4285_VIC_REG_BASE + 0x16)
#define P4285_VIC_ICSR      (P4285_VIC_REG_BASE + 0x17)
#define P4285_VIC_ICR0      (P4285_VIC_REG_BASE + 0x18)
#define P4285_VIC_ICR1      (P4285_VIC_REG_BASE + 0x19)
#define P4285_VIC_ICR2      (P4285_VIC_REG_BASE + 0x1A)
#define P4285_VIC_ICR3      (P4285_VIC_REG_BASE + 0x1B)
#define P4285_VIC_ICR4      (P4285_VIC_REG_BASE + 0x1C)
#define P4285_VIC_ICR5      (P4285_VIC_REG_BASE + 0x1D)
```

```
#define P4285_VIC_ICR6      (P4285_VIC_REG_BASE + 0x1E)
#define P4285_VIC_ICR7      (P4285_VIC_REG_BASE + 0x1F)
#define P4285_VIC_VIRSR     (P4285_VIC_REG_BASE + 0x20)
#define P4285_VIC_VIVBR1    (P4285_VIC_REG_BASE + 0x21)
#define P4285_VIC_VIVBR2    (P4285_VIC_REG_BASE + 0x22)
#define P4285_VIC_VIVBR3    (P4285_VIC_REG_BASE + 0x23)
#define P4285_VIC_VIVBR4    (P4285_VIC_REG_BASE + 0x24)
#define P4285_VIC_VIVBR5    (P4285_VIC_REG_BASE + 0x25)
#define P4285_VIC_VIVBR6    (P4285_VIC_REG_BASE + 0x26)
#define P4285_VIC_VIVBR7    (P4285_VIC_REG_BASE + 0x27)
#define P4285_VIC_TTR       (P4285_VIC_REG_BASE + 0x28)
#define P4285_VIC_LBTR      (P4285_VIC_REG_BASE + 0x29)
#define P4285_VIC_BTDR      (P4285_VIC_REG_BASE + 0x2A)
#define P4285_VIC_ICR       (P4285_VIC_REG_BASE + 0x2B)
#define P4285_VIC_ARCR      (P4285_VIC_REG_BASE + 0x2C)
#define P4285_VIC_AMSR      (P4285_VIC_REG_BASE + 0x2D)
#define P4285_VIC_BESR      (P4285_VIC_REG_BASE + 0x2E)
#define P4285_VIC_DMASR     (P4285_VIC_REG_BASE + 0x2F)
#define P4285_VIC_SS0CR0    (P4285_VIC_REG_BASE + 0x30)
#define P4285_VIC_SS0CR1    (P4285_VIC_REG_BASE + 0x31)
#define P4285_VIC_SS1CR0    (P4285_VIC_REG_BASE + 0x32)
#define P4285_VIC_SS1CR1    (P4285_VIC_REG_BASE + 0x33)
#define P4285_VIC_RCR       (P4285_VIC_REG_BASE + 0x34)
#define P4285_VIC_BTCR      (P4285_VIC_REG_BASE + 0x35)
#define P4285_VIC_BTLR1     (P4285_VIC_REG_BASE + 0x36)
#define P4285_VIC_BTLR0     (P4285_VIC_REG_BASE + 0x37)
#define P4285_VIC_SRR       (P4285_VIC_REG_BASE + 0x38)
#define P4285_VIC_BTLR2     (P4285_VIC_REG_BASE + 0x39)

#endif
```

```
/*****************************************************************************
*
*   File : init6516.c
*
*   Description: This file contains routines for setting up the 6516.
*
*
*   $History: init6516.c $
*
*
*****************************************************************************/

#include "math.h"
#include "6516.h"
#include "65xxcmd.h"

#ifdef FLOAT_64
#include "flotar.h"   /* Tartan's 64 Bit Floating Point Library */
#endif

/*****************************************************************************
 Function: P6516InitRegAddr

 Description: This function initializes the P65XX_REG_ADDR structure with
              the addresses to each of the 6516 registers
 Inputs:
          regBaseAddr - Local address window to the 6516's Global Sram
          p6516Reg - Pointer to the P6516516G_ADDR Table
 Outputs:
          Register Addresses stored in Address Table
 Returns:
 Notes:  Assumes a C30 or C40 is calling this routine

*****************************************************************************/
void P6516InitRegAddr(unsigned long regBaseAddr, P6516_REG_ADDR *p6516Reg)
{
    p6516Reg->Int_Mask_1 = (unsigned long *)regBaseAddr +
                                            (P6516_INTR_MASK_REG1 / 4);
    p6516Reg->Int_Mask_2 = (unsigned long *)regBaseAddr +
                                            (P6516_INTR_MASK_REG2 / 4);
    p6516Reg->Stat_Reg_1 = (unsigned long *)regBaseAddr +
                                            (P6516_STATUS_REG1 / 4);
    p6516Reg->Stat_Reg_2 = (unsigned long *)regBaseAddr +
                                            (P6516_STATUS_REG2 / 4);
    p6516Reg->Fifo_Reset = (unsigned long *)regBaseAddr +
                                            (P6516_FIFO_RESET_REG / 4);
    p6516Reg->Out_Comm_Ctl = (unsigned long *)regBaseAddr +
                                              (P6516_COMM_PORT_CNTL_REG / 4);
    p6516Reg->Input_Ctl = (unsigned long *)regBaseAddr +
                                           (P6516_INPUT_CNTL_REG / 4);
    p6516Reg->Rcvr_Ctl = (unsigned long *)regBaseAddr +
                                          (P6516_RECV_CNTL_REG / 4);
    p6516Reg->Rcvr_Ctl_2 = (unsigned long *)regBaseAddr +
                                            (P6516_RECV_CNTL_REG2 / 4);
    p6516Reg->Mode_Mux = (unsigned long *)regBaseAddr +
                                          (P6516_MODE_MUX_REG / 4);
    p6516Reg->Tag_En = (unsigned long *)regBaseAddr +
                                        (P6516_TAG_ENABLE_REG / 4);
    p6516Reg->Thrshld_Detect_A = (unsigned long *)regBaseAddr +
                                                  (P6516_THRSHLD_DETECT_A / 4);
    p6516Reg->Thrshld_Detect_B = (unsigned long *)regBaseAddr +
                                                  (P6516_THRSHLD_DETECT_B / 4);
    p6516Reg->Thrshld_Detect_C = (unsigned long *)regBaseAddr +
                                                  (P6516_THRSHLD_DETECT_C / 4);
```

```
    p6516Reg->Thrshld_Detect_D = (unsigned long *)regBaseAddr +
                                                (P6516_THRSHLD_DETECT_D / 4);
    p6516Reg->Fifo = (unsigned long *)regBaseAddr +
                                        (P6516_RCVR_1_FIFO / 4);
    p6516Reg->VME_Intr_Reg = (unsigned long *)regBaseAddr +
                                        (P6516_VME_INTR_REG / 4);
    p6516Reg->Intr_Ackn_Addr = (unsigned long *)regBaseAddr +
                                        (P6516_INTR_ACKN_REG / 4);
    p6516Reg->HSP_Shadow_Reg_Base = (unsigned long *)regBaseAddr +
                                            (P6516_HSP_SHADOW_REG_BASE / 4);
}
/*************************************************************************
 Function: P6516ResetRegs

 Description: This function initializes the 6516 registers to a known state
 Inputs:
         p6516Reg - Pointer to the 6516's address table
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516ResetRegs(P6516_REG_ADDR *p6516Reg)
{
    *p6516Reg->Int_Mask_1 = 0;
    *p6516Reg->Int_Mask_2 = 0;
    *p6516Reg->Fifo_Reset = 0x100;
    *p6516Reg->Out_Comm_Ctl = 0x1111;       /* Disable Comm Ports    */
    *p6516Reg->Input_Ctl = 0x06060606;      /* Enable Software Reset */
    *p6516Reg->Rcvr_Ctl = 0x11111111;
    *p6516Reg->Rcvr_Ctl_2 = 0x11111111;
    *p6516Reg->Tag_En = 0x0;
    *p6516Reg->Mode_Mux = 0;

    *p6516Reg->Thrshld_Detect_A = 0;
    *p6516Reg->Thrshld_Detect_B = 0;
    *p6516Reg->Thrshld_Detect_C = 0;
    *p6516Reg->Thrshld_Detect_D = 0;
}
/*************************************************************************
 Function: P6516InitRcvrRegs

 Description:  This function initializes the 6516 Receivers to the settings
               specified
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516InitRcvrRegs(P6516_RCVR_PARAMS *rcvr_params,
                       P6516_REG_ADDR *p6516Reg,
                       unsigned int rcvr_mask)
{
    unsigned long temp;

    /* Select Input Source */
    P6516SelectInputSource(rcvr_params, p6516Reg, rcvr_mask);

    /* Set Data Mode */
    P6516SelectDataMode(rcvr_params, p6516Reg, rcvr_mask);
```

```
    /* Set Fifo Mode */
    P6516SetFifoMode(rcvr_params, p6516Reg, rcvr_mask);

    /* Select Data Format */
    P6516SelectDataFormat(rcvr_params, p6516Reg, rcvr_mask);

    /* Select Tagging */
    P6516SelectTagEnable(rcvr_params, p6516Reg, rcvr_mask);

    /* Set Data Transfer Method */
    P6516SetDataTransfer(rcvr_params, p6516Reg, rcvr_mask);

    /* Perform a One time Initialization of the Harris Receivers */
    P6516InitAllRcvr(rcvr_params, p6516Reg, rcvr_mask, P65XX_NO_REG_UPDATE);

    /* Initialize Receiver Parameters */
    P6516InitRcvr(rcvr_params, p6516Reg, rcvr_mask, P65XX_REG_UPDATE);
}
/*************************************************************************
 Function: P6516InitInputRegs

 Description:  This function initializes the 6516 Input Channels to the
               settings specified
 Inputs:
        input_params - Pointer to the P65XX_INPUT_PARAMS structure containing
                       the input channel settings
        p6516Reg - Pointer to the 6516's address table
        input_mask - Mask specifying which input channels are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516InitInputRegs(P6516_INPUT_PARAMS *input_params,
                        P6516_REG_ADDR *p6516Reg,
                        unsigned int input_mask)
{
    /* Set Threshold Detection */
    P6516SetThresholdDetection(input_params, p6516Reg, input_mask);

    /* Set Input Crossover */
    P6516SetInputCrossover(input_params, p6516Reg);

    /* Set Input Reset Mode */
    P6516SetInputResetMode(input_params, p6516Reg, input_mask);
}

/*************************************************************************
 Function: P6516SetRcvrDefaults

 Description:  This function places default settings in the receiver parameter
               structure.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure being
                      initialized
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516SetRcvrDefaults(P6516_RCVR_PARAMS *rcvr_params)
{
    rcvr_params->decimation_rate = 64;
    rcvr_params->data_mode = P65XX_COMPLEX;
    rcvr_params->data_source = P65XX_INPUT_A;
    rcvr_params->packing_mode = P65XX_PACKED;
```

```
    rcvr_params->data_order = P65XX_IQ_ORDER;
    rcvr_params->data_transfer = P65XX_FIFO;
    rcvr_params->fifo_select = P65XX_ODD_FIFO;
    rcvr_params->tag_enable = P65XX_TAG_OFF;
    rcvr_params->flip_spectrum = P65XX_FLIP_OFF;
    rcvr_params->offset_spectrum = P65XX_OFFSET_ON;
    rcvr_params->wait_ram_full = P65XX_WAIT_RAM_FULL_ON;
    rcvr_params->fifo_mode = P6516_DUAL_RCVR;
    rcvr_params->lo =  3.0e6;
    rcvr_params->input_clock_rate = 40.0e6;
}

/*************************************************************************
 Function: P6516SetInputDefaults

 Description:  This function places default settings in the input channel
               parameter structure.
 Inputs:
        input_params - Pointer to the P65XX_INPUT_PARAMS structure being
                       initialized.
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516SetInputDefaults(P6516_INPUT_PARAMS *input_params)
{
    input_params->threshold_level = P65XX_THRSHLD_DETECT_OFF;
    input_params->threshold_cross_cnt = P65XX_CROSS_CNT_2;
    input_params->input_crossover = P6516_CROSSOVER_A_TO_C_OFF |
                                    P6516_CROSSOVER_B_TO_D_OFF;
    input_params->input_reset_mode = P6516_SW_RESET_IN_SRC |
                                     P6516_SW_RESET_OFF;
}

/*************************************************************************
 Function: P6516SelectInputSource

 Description:  This function selects the input source for the receiver.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receivers are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516SelectInputSource(P6516_RCVR_PARAMS *rcvr_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long mask = 0;
    unsigned long data_source = rcvr_params->data_source;

    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & (0x3 << (i * 2)))
            mask |= 0x100 << i;
    }

    if (data_source == P65XX_INPUT_A)
      *p6516Reg->Mode_Mux = *p6516Reg->Mode_Mux & ~mask;
    else if (data_source == P65XX_INPUT_B)
      *p6516Reg->Mode_Mux = *p6516Reg->Mode_Mux | mask;
```

```
    else if (data_source == P65XX_INPUT_C)
      *p6516Reg->Mode_Mux = *p6516Reg->Mode_Mux & ~mask;
    else /* data_source == P65XX_INPUT_D  */
      *p6516Reg->Mode_Mux = *p6516Reg->Mode_Mux | mask;
}
/*****************************************************************************
 Function: P6510SetInputCrossover

 Description:  This function sets the input crossover for input channels A
               to C and B to D.
 Inputs:
       input_params - Pointer to the P65XX_INPUT_PARAMS structure containing
                      the input crossover settings
       p6510Reg - Pointer to the 6510's address table
 Outputs:
 Returns:

 Notes:
*****************************************************************************/
void P6516SetInputCrossover(P6516_INPUT_PARAMS *input_params,
                            P6516_REG_ADDR *p6516Reg)
{
    unsigned long input_ctl = *p6516Reg->Input_Ctl & ~0x10100000L;

    *p6516Reg->Input_Ctl = input_ctl | input_params->input_crossover;
}

/*****************************************************************************
 Function: P6516SelectDataMode

 Description:  This function sets the data mode for a receiver as Real or
               Complex.
 Inputs:
       rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                     the settings
       p6516Reg - Pointer to the 6516's address table
       rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SelectDataMode(P6516_RCVR_PARAMS *rcvr_params,
                         P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long mask = 0;

    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & (0x3 << (i * 2)))
            mask |= 0x1 << i;
    }

    if (rcvr_params->data_mode == P65XX_REAL)
      *p6516Reg->Mode_Mux = (*p6516Reg->Mode_Mux & ~mask) | mask;
    else   /* data_mode == P65XX_COMPLEX */
      *p6516Reg->Mode_Mux = (*p6516Reg->Mode_Mux & ~mask);
}
/*****************************************************************************
 Function: P6516SelectTagEnable

 Description:  This function enables/disables tagging of the output data.
 Inputs:
       rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
```

```
                        the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SelectTagEnable(P6516_RCVR_PARAMS *rcvr_params,
                          P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long mask = 0;

    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & (0x3 << (i * 2)))
            mask |= 0x1 << i;
    }

    if (rcvr_params->tag_enable == P65XX_TAG_ON)
      *p6516Reg->Tag_En = (*p6516Reg->Tag_En & ~mask) | mask;
    else   /* tag_enable == P65XX_TAG_OFF */
      *p6516Reg->Tag_En = (*p6516Reg->Tag_En & ~mask);
}
/*****************************************************************************
 Function: P6516SelectDataFormat

 Description:  This function selects the format of the data output to the
               front panel communication ports.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SelectDataFormat(P6516_RCVR_PARAMS *rcvr_params,
                           P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask)
{
     unsigned long temp_data = 0;
     unsigned long data_word = 0;
     unsigned long out_mask = 0xffff;
     unsigned int  i;

     if (rcvr_params->packing_mode == P65XX_PACKED)
       temp_data = 0x8;

     if (rcvr_params->data_order == P65XX_IQ_ORDER)
       temp_data |= 0x4;

     if (rcvr_params->fifo_select == P65XX_ODD_FIFO)
       temp_data |= 0x2;

     for (i = 0; i < P6516_NUM_COMM_PORTS; ++i)
     {
        if (rcvr_mask & (0xf << (i * 4)))
          {
            data_word |= temp_data << (i * 4);
              out_mask = (out_mask & ~(0xf << (i * 4))) | (1 << (i * 4));
          }
     }
     *p6516Reg->Out_Comm_Ctl = (*p6516Reg->Out_Comm_Ctl & out_mask) |
```

```
                                data_word;
}

/*****************************************************************************
 Function: P6516SetCommPortToken

 Description:  This function initializes the input communication ports to the
               C40.  This function makes the 4 front panel communication ports
               input ports.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
 Outputs:
 Returns:
 Notes:  This function doesn't affect the four output ports.
         This function can be called after the processor has been
         reset one time ONLY.
*****************************************************************************/
void P6516SetCommPortToken(P6516_REG_ADDR *p6516Reg)
{
     volatile unsigned long reset_word;

     reset_word = *p6516Reg->Fifo_Reset;
     reset_word &= ~0x100;
     *p6516Reg->Fifo_Reset = reset_word;
     reset_word |= 0x100;
     *p6516Reg->Fifo_Reset = reset_word;
}

/*****************************************************************************
 Function: P6516SetDataTransfer

 Description:  This function sets the output source for reading the data.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetDataTransfer(P6516_RCVR_PARAMS *rcvr_params,
                          P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask)
{
     int           i;
     unsigned long data_word = 0;
     unsigned long temp_data = 0;
     unsigned long out_mask = 0xffff;

     if (rcvr_params->data_transfer == P65XX_FIFO)
       temp_data = 0x1;

     for (i = 0; i < P6516_NUM_COMM_PORTS; ++i)
     {
        if (rcvr_mask & (0xf << (i * 4)))
          {
            data_word |= temp_data << (i * 4);
              out_mask = (out_mask & ~(0xf << (i * 4))) | (0xE << (i * 4));
          }
     }
     *p6516Reg->Out_Comm_Ctl = (*p6516Reg->Out_Comm_Ctl & out_mask) |
                                data_word;
}
```

```
/*****************************************************************************
 Function: P6516ResetFifo

 Description:  This function puts the fifo in a reset state for the given
               receivers.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:  When reading data directly from the fifos, each fifo can be reset
         indiviually.  When reading from Comm Ports, Fifos must be reset in
         pairs.
*****************************************************************************/
void P6516ResetFifo(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                    unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long reset_word = *p6516Reg->Fifo_Reset;
    unsigned long mask = 0;

    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & (0x3 << (i * 2)))
            mask |= 0x1 << i;
    }

    /* In VME Mode, all fifos can be reset individually.  In Comm Port   */
    /* Mode, the fifos have to be reset in pairs(1&2, 3&4, 5&6, 7&8).    */
    if (rcvr_params->data_transfer == P65XX_FIFO)
    {
        *p6516Reg->Fifo_Reset = *p6516Reg->Fifo_Reset & ~mask;
    }
    else   /* P65XX_COMM_PORT */
    {
        for (i = 0; i < P6516_NUM_RCVRS / 4; ++i)
        {
          if (mask & (0x3 << (i * 2)))
          {
                reset_word = reset_word & ~(0x3 << (i * 2));
           }
        }
        *p6516Reg->Fifo_Reset = reset_word;
    }
}
/*****************************************************************************
 Function: P6516ReleaseFifo

 Description:  This function release the fifo(s) for the given receivers.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:  When reading data directly from the fifos, each fifo can be released
         indiviually.  When reading from Comm Ports, Fifos must be released in
         pairs.
*****************************************************************************/
void P6516ReleaseFifo(P6516_RCVR_PARAMS *rcvr_params,
                      P6516_REG_ADDR *p6516Reg,
```

```
                      unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long reset_word = *p6516Reg->Fifo_Reset;
    unsigned long mask = 0;

    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & (0x3 << (i * 2)))
            mask |= 0x1 << i;
    }

    /* In VME Mode, all fifos can be released individually. In Comm Port   */
    /* Mode, the fifos have to be reset in pairs(1&2, 3&4, 5&6, 7&8).      */

    if (rcvr_params->data_transfer == P65XX_FIFO)
    {
        *p6516Reg->Fifo_Reset = *p6516Reg->Fifo_Reset | mask;
    }
    else   /* P65XX_COMM_PORT */
    {
        for (i = 0; i < P6516_NUM_RCVRS / 4; i++)
        {
          if (mask & (0x3 << (i * 2)))
          {
                reset_word = reset_word | (0x3 << (i * 2));
          }
        }
        *p6516Reg->Fifo_Reset = reset_word;
    }
}
/*************************************************************************
 Function: P6516SetFifoMode

 Description:  This function sets the number of rcvr channels stored in a
               fifo.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516SetFifoMode(P6516_RCVR_PARAMS *rcvr_params,
                      P6516_REG_ADDR *p6516Reg,
                      unsigned int rcvr_mask)
{
    unsigned long i;
    unsigned long mask = 0;

    for (i = 0; i < P6516_NUM_RCVRS / 4; ++i)
    {
        if (rcvr_mask & (0xf << (i * 4)))
            mask |= 0x40 << (8 * i);
    }

    if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
      *p6516Reg->Input_Ctl = (*p6516Reg->Input_Ctl & ~mask) | mask;
    else   /* fifo_mode == P6516_SINGLE_RCVR */
      *p6516Reg->Input_Ctl = (*p6516Reg->Input_Ctl & ~mask);
}
/*************************************************************************
```

```
 Function: P6516SetThresholdDetection

 Description:  This function sets up threshold detection on a specified
               input channel.
 Inputs:
        input_params - Pointer to the P65XX_INPUT_PARAMS structure containing
                       the input channel settings
        p6516Reg - Pointer to the 6516's address table
        input_mask - Mask specifying which input channels are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetThresholdDetection(P6516_INPUT_PARAMS *input_params,
                                P6516_REG_ADDR *p6516Reg,
                                unsigned int input_mask)
{
    unsigned int i;
    volatile unsigned long *thrshld_detect[P6516_NUM_INPUT_CHANNELS];

    thrshld_detect[0] = p6516Reg->Thrshld_Detect_A;
    thrshld_detect[1] = p6516Reg->Thrshld_Detect_B;
    thrshld_detect[2] = p6516Reg->Thrshld_Detect_C;
    thrshld_detect[3] = p6516Reg->Thrshld_Detect_D;

    for (i = 0; i < P6516_NUM_INPUT_CHANNELS; i++)
    {
      if (input_mask & (1 << i))
            *thrshld_detect[i] = input_params->threshold_level |
                                 input_params->threshold_cross_cnt;
    }
    /* Turn off Threshold Detect LED, if turning off detection */
    if (input_params->threshold_level == P65XX_THRSHLD_DETECT_OFF)
        P6516SetThresholdLED(p6516Reg, input_mask, 0);
}
/*****************************************************************************
 Function: P6516SetThresholdLED

 Description:  This function turns ON/OFF the threshold LED on front panel
               for a given input channel.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        input_mask - input mask
        on_off_flag - Led Setting: ON = 1, OFF = 0;
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetThresholdLED(P6516_REG_ADDR *p6516Reg,
                          unsigned int input_mask,
                          unsigned long on_off_flag)
{
    unsigned long i;
    unsigned long ledFlag = 0;

    for (i = 0; i < P6516_NUM_INPUT_CHANNELS; ++i)
    {
        if (input_mask & (1 << i))
            ledFlag |= 0x80 << (8 * i);
    }
    if (on_off_flag)   /* LED On */
        *p6516Reg->Input_Ctl |= ledFlag;
    else  /* Led Off */
        *p6516Reg->Input_Ctl &= ~ledFlag;
```

```
}
/*****************************************************************************
 Function: P6516SetIntrMask1

 Description:  This function enables the fifo full, clock not ready and
               threshold detection interrupts.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        intr_mask - Mask containing interrupts to be enabled
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetIntrMask1(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask)
{
    unsigned long intr_cntl = (intr_mask & 0x01ff0000L) >> 16;

    if (intr_mask & P65XX_CLOCK_NOT_READY_B)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_B;
    if (intr_mask & P65XX_CLOCK_NOT_READY_C)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_C;
    if (intr_mask & P65XX_CLOCK_NOT_READY_D)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_D;

    if (intr_mask & P65XX_THRSHLD_DETECT_A)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_A;
    if (intr_mask & P65XX_THRSHLD_DETECT_B)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_B;
    if (intr_mask & P65XX_THRSHLD_DETECT_C)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_C;
    if (intr_mask & P65XX_THRSHLD_DETECT_D)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_D;

    *p6516Reg->Int_Mask_1 |= intr_cntl;
}
/*****************************************************************************
 Function: P6516ClearIntrMask1

 Description:  This function disables the fifo full, clock not ready and
               threshold detection interrupts.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        intr_mask - Mask containing interrupts to be disabled
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516ClearIntrMask1(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask)
{
    unsigned long intr_cntl = (intr_mask & 0x01ff0000) >> 16;

    if (intr_mask & P65XX_CLOCK_NOT_READY_B)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_B;
    if (intr_mask & P65XX_CLOCK_NOT_READY_C)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_C;
    if (intr_mask & P65XX_CLOCK_NOT_READY_D)
        intr_cntl |= P6516_IM1_CLOCK_NOT_READY_D;

    if (intr_mask & P65XX_THRSHLD_DETECT_A)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_A;
    if (intr_mask & P65XX_THRSHLD_DETECT_B)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_B;
    if (intr_mask & P65XX_THRSHLD_DETECT_C)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_C;
```

```
    if (intr_mask & P65XX_THRSHLD_DETECT_D)
        intr_cntl |= P6516_IM1_THRSHLD_DETECT_D;

    *p6516Reg->Int_Mask_1 &= ~intr_cntl;
}
/*************************************************************************
 Function: P6516SetIntrMask2

 Description:  This function enables the fifo half full and fifo not empty
               interrupts.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        intr_mask - Mask containing interrupts to be enabled
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516SetIntrMask2(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask)
{
    *p6516Reg->Int_Mask_2 |= intr_mask & 0xffffL;
}
/*************************************************************************
 Function: P6516ClearIntrMask2

 Description:  This function disables the fifo half full and fifo not empty
               interrupts.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        intr_mask - Mask containing interrupts to be disabled
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516ClearIntrMask2(P6516_REG_ADDR *p6516Reg, unsigned long intr_mask)
{
    *p6516Reg->Int_Mask_2 &= ~intr_mask & 0xffffL;
}
/*************************************************************************
 Function: P6516SetLO

 Description:  This function sets the Local Oscillator Frequency for the
               specified receiver
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the reciever's settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:  64 Bit floating point provides a more accurate LO Setting
*************************************************************************/
void P6516SetLo(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                unsigned int rcvr_mask, unsigned long update_reg)
{
    unsigned long HALo;
    unsigned long HAHi;
    unsigned int freq;

    /* Calculate the Minimum Phase Increment                          */
    /* Phase Incr = ((2**33) / Input Clock Rate) * Desired LO / 2     */
    /* Center Freq is scaled by 2 to prevent overflow on real-int     */
    /* conversion                                                     */
#ifndef FLOAT_64
    freq = (pow(2.0, 33) / rcvr_params->input_clock_rate) *
```

```
                (rcvr_params->lo / 2.0);
    freq *= 2;
#else
    float64  f64temp;
    float64  f64temp1 = {0x00000000, 8.589934592e+09};

    divf64f32(&f64temp, &f64temp1, rcvr_params->input_clock_rate);
    mpyf64f32(&f64temp1, &f64temp, rcvr_params->lo);
    freq = roundu32f64(&f64temp1);
#endif

    /* Control Register 1: Normal Mode (CW), */
    HALo = (freq << 4) | 0x1 | (rcvr_params->wait_ram_full << 3);
    HAHi = 0x20 | (update_reg << 4) | (freq >> 28);
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

}
/*****************************************************************************
 Function: P6516SetDecimationRate

 Description:  This function sets the decimation rate for the specified
               receiver.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the reciever's settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetDecimationRate(P6516_RCVR_PARAMS *rcvr_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int rcvr_mask,
                            unsigned long update_reg)
{
    register unsigned long HALo;
    register unsigned long HAHi;
    unsigned long dec_factor;
    unsigned int  scaling;
    double        scaleMult;

    /* Adjust Decimation Rate by 4D */
    if (rcvr_params->data_mode == P65XX_COMPLEX)
        dec_factor = rcvr_params->decimation_rate / 4;
    else
        dec_factor = rcvr_params->decimation_rate / 2;

    /* Control Register 4: adjust gain for HDF decimation , offset and flip */
    if (rcvr_params->data_mode == P65XX_COMPLEX)
    {
        HALo = P6516DetermineHDFDataShift(dec_factor) |
                                            rcvr_params->flip_spectrum;
        HAHi = 0x80 | rcvr_params->offset_spectrum;
    }
    else
    {
        HALo = 0x80000000 | P6516DetermineHDFDataShift(dec_factor) |
                rcvr_params->flip_spectrum;
        HAHi = 0x80;
    }
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

    /* Control Register 5: HDF decimation  */
    scaleMult = P6516DetermineScaleMult(dec_factor);
```

```
    scaling = scaleMult * pow(2.0, 20);
    scaling &= 0x001fffe0;
    HALo = scaling | (dec_factor - 1) << 21;
    HAHi = 0xa0 | (update_reg << 4) | ((dec_factor - 1) >> 11);
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

}
/*************************************************************************
 Function: P6516InitRcvr

 Description:  This function initializes the all the Harris Control registers
               with the current settings.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516InitRcvr(P6516_RCVR_PARAMS *rcvr_params, P6516_REG_ADDR *p6516Reg,
                   unsigned int rcvr_mask, unsigned long update_reg)
{
    /* Set Tuning Frequency - no update */
    P6516SetLo(rcvr_params, p6516Reg, rcvr_mask, 0);

    /* Set Decimation Rate */
    P6516SetDecimationRate(rcvr_params, p6516Reg, rcvr_mask, update_reg);
}

/*************************************************************************
 Function: P6516InitAllRcvr

 Description:  This function performs a one-time initialization of all the
               control registers whose settings don't change.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the settings
        p6516Reg - Pointer to the 6516's address table
        rcvr_mask - Mask specifying which receviers are affected
        update_reg - Flag specifying that whether to update the control
                     registers
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void P6516InitAllRcvr(P6516_RCVR_PARAMS *rcvr_params,
                      P6516_REG_ADDR *p6516Reg,
                      unsigned int rcvr_mask, unsigned long update_reg)
{
    unsigned long HALo;
    unsigned long HAHi;

    HALo = 0x0;             /* Control Register 2: max phase increment */
    HAHi =  0x40;
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

    HALo = 0x0;             /* Control Register 3: phase offset = 0 */
    HAHi =  0x60;
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

    HALo = 0x0D526002;      /* Control Register 6: hardware specific */
    HAHi =  0xc0;
```

```
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);

    /* Control Register 7: Wait on Ram Full */
    HALo = 0x00000002;
    HAHi =  0xe0 | (update_reg << 4);
    P6516SetRcvrWord(p6516Reg, HAHi, HALo, rcvr_mask);
}

/*************************************************************************
 Function: P6516SetRcvrWord

 Description:  This function sets a control register for one or more
               receivers.
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        HAHi - Bit D32 - D39 of the Control Register
        HALo - Bit D0 - D31 of the Control Register
        rcvr_mask - Mask specifying which receviers are affected
 Outputs:
 Returns:
 Notes:
*************************************************************************/
#define Data_Bit_Mask 0x8
#define Clock_Hi      0x4
#define Clock_Lo      0xB
#define STB_Hi        0x2
#define STB_Lo        0xD
#define CS_Hi         0x1
#define CS_Lo         0xE
#define CLOCK_DELAY   5

void P6516SetRcvrWord(P6516_REG_ADDR *p6516Reg, unsigned long HAHi,
                      unsigned long HALo, register unsigned int rcvr_mask)
{
    register volatile unsigned long *rcvr_ctl = p6516Reg->Rcvr_Ctl;
    unsigned long temp_data;
    unsigned long Shift_Mask;
    unsigned long data_word;
    unsigned long clkLo;
    unsigned long clkHi;
    unsigned long csHi;
    unsigned long csLo;
    unsigned long stbHi;
    unsigned long stbLo;
    int           Bit_Count;
    volatile int  delay;
    register int  i;
    register int  j;
    unsigned long rcvr_ctl_mask;
    volatile unsigned long *hsp_shadow_reg =
                               &p6516Reg->HSP_Shadow_Reg_Base[(HAHi >> 5)*2];
    unsigned long shadow_offset;

    for (j = 0; j < 2; ++j)
    {
        if (rcvr_mask & (0x5555 << j))
        {
            if (j == 0)
                rcvr_ctl = p6516Reg->Rcvr_Ctl;
            else
                rcvr_ctl = p6516Reg->Rcvr_Ctl_2;
            clkLo = 0;
            clkHi = 0;
            csHi = 0;
```

```
csLo = 0;
stbHi = 0;
stbLo = 0;
rcvr_ctl_mask = 0;
for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
{
    if (rcvr_mask & ((1 << j) << (i * 2)))
    {
        clkLo |= Clock_Lo << (4 * i);
        clkHi |= Clock_Hi << (4 * i);
        csHi |= CS_Hi << (4 * i);
        csLo |= CS_Lo << (4 * i);
        stbHi |= STB_Hi << (4 * i);
        stbLo |= STB_Lo << (4 * i);

        shadow_offset = 0x10 * (j + (2 * i));
        *(hsp_shadow_reg + shadow_offset) = HAHi;
        *(hsp_shadow_reg + shadow_offset + 1) = HALo;
    }
    else
        rcvr_ctl_mask |= 0xf << (4 * i);
}

/* Pulse Strobe for Start of new Command word    */
*rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | (clkHi | csHi | stbHi);
for (delay = 0; delay < CLOCK_DELAY; delay++);
*rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) |
            (clkLo & (csHi | stbHi));
for (delay = 0; delay < CLOCK_DELAY; delay++);
/* Strobe changes on Clock */
*rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) |
            ((clkHi | csHi) & stbLo);

for (Bit_Count = 0; Bit_Count <= 7; Bit_Count++)
{
    if (Bit_Count >= 4)
        Shift_Mask = (clkHi & csLo);  /* Strobe in Address */
    else
        Shift_Mask = (clkHi | csHi);  /* Strobe in Data    */

    temp_data = ((HAHi >> 4) & Data_Bit_Mask) & 0xf;

    data_word = 0;
    for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
    {
        if (rcvr_mask & ((1 << j) << (i * 2)))
            data_word |= temp_data << (4 * i);
    }
    /* Send data for Harris Chips     */
    *rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | data_word |
                Shift_Mask;

    for (delay = 0; delay < CLOCK_DELAY; delay++);

    /* Set Clock LO  */
    *rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | (data_word & clkLo);

    HAHi = (HAHi << 1);   /* Move the next bit into position  */
}

for (Bit_Count = 0; Bit_Count <= 31; Bit_Count++)
{
    Shift_Mask = (clkHi & csLo);
```

```
                temp_data = ((HALo >> 28) & Data_Bit_Mask) & 0xf;

                data_word = 0;
                for (i = 0; i < P6516_NUM_RCVRS / 2; ++i)
                {
                    if (rcvr_mask & ((1 << j) << (i * 2)))
                        data_word |= temp_data << (4 * i);
                }

                /* Send data for Harris Chips     */
                *rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | data_word |
                            Shift_Mask;

                for (delay = 0; delay < CLOCK_DELAY; delay++);

                /* Set Clock LO  */
                *rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | (data_word & clkLo);

                HALo = (HALo << 1);    /* Move the next bit into position  */
            }
            /* Set CS=1 till next Frame Sync */
            *rcvr_ctl = (*rcvr_ctl & rcvr_ctl_mask) | (data_word | csHi);
        }
    }
}
/*************************************************************************
 Function: P6516ResetReceiver

 Description:  This function puts all receiver chips driven by the specified
               input channels in a reset state
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        input_mask - Mask specifying which input channels are affected
 Outputs:
 Returns:
 Notes: The receiver will not output data which in a reset state
*************************************************************************/
void P6516ResetReceiver(P6516_REG_ADDR *p6516Reg, unsigned int input_mask)
{
    unsigned long i;
    unsigned long input_cntl;

    input_cntl = *p6516Reg->Input_Ctl;
    for (i = 0; i < P6516_NUM_INPUT_CHANNELS; ++i)
    {
        if (input_mask & (1 << i))
        {
            input_cntl &=  ~(0xf << (8 * i));
            input_cntl |= (0x4 << (8 * i));
        }
    }
    *p6516Reg->Input_Ctl = input_cntl;
}
/*************************************************************************
 Function: P6516ReleaseReceiver

 Description:  This function puts all receiver chips driven by the specified
               input channels in a run state
 Inputs:
        p6516Reg - Pointer to the 6516's address table
        input_mask - Mask specifying which input channels are affected
 Outputs:
 Returns:
 Notes: The receiver will output data while in a run state
```

```
*****************************************************************************/
void P6516ReleaseReceiver(P6516_REG_ADDR *p6516Reg, unsigned int input_mask)
{
    unsigned long i;
    unsigned long input_cntl;

    input_cntl = *p6516Reg->Input_Ctl;
    for (i = 0; i < P6516_NUM_INPUT_CHANNELS; ++i)
    {
        if (input_mask & (1 << i))
        {
            input_cntl &=  ~(0xf << (8 * i));
            input_cntl |= (0x6 << (8 * i));
        }
    }
    *p6516Reg->Input_Ctl = input_cntl;
}

/*****************************************************************************
 Function: P6516SetInputResetMode

 Description:  This function sets the input reset mode for each of the input
               channels.
 Inputs:
        input_params - Pointer to the P65XX_INPUT_PARAMS structure containing
                       the input channel settings
        p6516Reg - Pointer to the 6516's address table
        input_mask - Mask specifying which input channels are affected
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P6516SetInputResetMode(P6516_INPUT_PARAMS *input_params,
                            P6516_REG_ADDR *p6516Reg, unsigned int input_mask)
{
    unsigned long i;
    unsigned long input_cntl;

    input_cntl = *p6516Reg->Input_Ctl;
    for (i = 0; i < P6516_NUM_INPUT_CHANNELS; ++i)
    {
        if (input_mask & (1 << i))
        {
           input_cntl &=  ~(0xf000f << (8 * i));
           input_cntl |= (input_params->input_reset_mode << (8 * i));
           input_cntl |= (((input_params->input_reset_mode & 0x6) << 16) <<
                                                                  (8 * i));
       }
   }
   *p6516Reg->Input_Ctl = input_cntl;
}

/*****************************************************************************
 Function: P6516DetermineHDFDataShift

 Description:  This function determines the HDF Data Shift which is used
               by the harris receiver to compensate for the truncation due
               to the 5 stage decimation filter.
 Inputs:
        dec_factor - Decimation Factor
 Outputs:
        HDF Data Shift setting for control register 4
 Returns:
 Notes:
```

```
*****************************************************************************/
unsigned int P6516DetermineHDFDataShift(unsigned int dec_factor)
{
    unsigned int dataShift;
    double       power;
    double       temp;

    /* Determine the Log2(dec_factor - 1) */
    power = log10((double)dec_factor - 1.0) / log10(2.0);

    /* Data shift = 75 - Ceiling(5*Log2(R))   */
    /*    R = Decimation Rate                 */
    temp = 5.0 * power;
    dataShift = 75.0 - ceil(temp);
    if ((int)dataShift < 0)
      dataShift = 0;
    dataShift = dataShift << 1;

    return(dataShift);
}

/*****************************************************************************
 Function: P6516DetermineScaleMult

 Description:  This function determines the scaling multiplier which is used
               by the harris receiver to compensate for HDF Gain.
 Inputs:
        dec_factor - Decimation Factor
 Outputs:
        Scaling Multiplier Gain for Control Register 5
 Returns:
 Notes:
*****************************************************************************/
double P6516DetermineScaleMult(unsigned int dec_factor)
{
    double       power;
    int          power1;
    double       temp;
    double       fraction;
    double       ScaleMult;


    /* Determine the Log2(dec_factor) */
    power = log10((double)dec_factor - 1.0) / log10(2.0);

    /* Determine if dec_factor is a factor of 2 */
    fraction = frexp((double) dec_factor, &power1);
    fraction *= 2.0;

    /* If decimation is a power of 2, then the scale multiplier is 1 */
    /* otherwise calculate the Scale Multiplier                      */
    /*     Scale Multiplier = 2**(Ceiling(5Log2(R))) / R**5          */
    /*           where R = decimation factor                         */
    if (fraction != 1.0)
    {
      temp = 5.0 * power;
      temp = ceil(temp);
      ScaleMult = pow(2.0, temp) / pow((double) dec_factor - 1.0, 5.0);
    }
    else
      ScaleMult = 1.0;

    return(ScaleMult);
}
```

```
/*****************************************************************************
 Function: P6516ReadFifoCmplx

 Description:  This function reads complex data from the fifo.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
        p6516Reg - Pointer to the 6516's address table
        samples - Pointer to a data buffer to store data
        num_samples - Number of samples to be read
        inputFifo - Fifo to be read (1 - 8)
 Outputs:
       Data is stored to the data buffer in the following format:
             Real,Imag,Real,Imag......
 Returns:
        samples - Address of the next available location in data buffer
 Notes:
        Data is sign extended and right justified
*****************************************************************************/
int *P6516ReadFifoCmplx(P6516_RCVR_PARAMS *rcvr_params,
                        P6516_REG_ADDR *p6516Reg,
                        register volatile int *samples,
                        register int num_samples,
                        unsigned int inputFifo)
{
    int          num_samples_to_read;
    register volatile int *fifo = (int *)p6516Reg->Fifo +
                                  ((0x8000 * (inputFifo - 1)) / 4);
    register int temp;
    register int temp1;
    register int i;

    while (num_samples > 0)
    {
        /* wait FIFO half-full */
      while ((*p6516Reg->Stat_Reg_2 & (0x100 << (inputFifo-1))) != 0);

      /* Determine how many samples left to read */
      if (num_samples <= P6516_FIFO_SIZE/2)
            num_samples_to_read = num_samples;
      else
          num_samples_to_read = P6516_FIFO_SIZE/2;

      /* Read out data from the fifo */
      for (i = 0; i < num_samples_to_read; i++)
      {
          temp = *fifo;    /* store real and imaginary */
          temp1 = *fifo;
          *samples++ = temp1 >> 16;    /* Real(I) Value            */
          *samples++ = temp >> 16;     /* Imag(Q) Value            */
      }

      /* Decrement number of samples needed */
      num_samples -= num_samples_to_read;
    }
    return((int *)samples);
}
/*****************************************************************************
 Function: P6516ReadCommPortCmplx

 Description:  This function reads complex data from the Comm Port.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
```

```
        samples - Pointer to a data buffer to store data
        num_samples - Number of samples to be read
        inputCommPort - Comm Port to be read (0 - 5)
 Outputs:
       Data is stored to the data buffer in the following format:
             Unpacked Data -  Real,Imag,Real,Imag ....  Sign Extended
                                                        Right Justified
             Packed, IQ Ordered - Real/Imag, Real/Imag ....
             Packed, Receiver Ordered - Imag(Even)/Imag(Odd),
                                        Real(Even)/Real(Odd).
 Returns:
        samples - Address of the next available location in data buffer
 Notes:
*************************************************************************/
int *P6516ReadCommPortCmplx(P6516_RCVR_PARAMS *rcvr_params,
                            register volatile int *samples,
                            register int num_samples,
                            unsigned int inputCommPort)
{
    int          num_samples_to_read;
    int          temp;
    register int i;
    volatile unsigned int *comStatus = (unsigned int *) 0x100040 +
                                                       0x10 * inputCommPort;
    register volatile int *comInput = (int *)comStatus + 1;

    if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
        num_samples *= 2;

    /* Read out data from the fifo */
    for (i = 0; i < num_samples; i++)
    {
      while ((*comStatus & 0x1e00) == 0);
        if (rcvr_params->packing_mode == P65XX_UNPACKED)
        {
            temp = *comInput;
            *samples++ = *comInput >> 16;   /* Real(I) Value            */
            *samples++ = temp >> 16;        /* Imag(Q) Value            */
        }
        else if (rcvr_params->data_order == P65XX_IQ_ORDER)
        {
          *samples++ = *comInput;   /* Real(I)/Imag(Q) */
        }
        else if (rcvr_params->data_order == P65XX_RECEIVER_ORDER)
        {
          *samples++ = *comInput;   /* Imag(Q)/Imag(Q) */
          *samples++ = *comInput;   /* Real(I)/Real(I) */
        }
    }
    return((int *)samples);
}
/*************************************************************************
 Function: P6516UnpackCmplxData

 Description:  This function unpacks a complex data buffer read from a Comm
               Port based on the packing mode selected.  This routine will
               unpack up to four receivers.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
        inSamples - Pointer to a data buffer containing complex data
        outSamples1 - Pointer to the output data buffer containing the first
                      rcvr
        outSamples2 - Pointer to the output data buffer containing the 2nd
```

```
                    rcvr
        outSamples3 - Pointer to the output data buffer containing the 3rd
                    rcvr
        outSamples4 - Pointer to the output data buffer containing the 4th
                    rcvr
        num_samples - Number of samples to be unpacked per receiver
 Outputs:
        Data is stored to the data buffer in the following format:
             Real,Imag,Real,Imag......
 Returns:
 Notes:
        Data is sign extended and right justified
*****************************************************************************/
void P6516UnpackCmplxData(P6516_RCVR_PARAMS *rcvr_params,
                          register volatile int *inSamples,
                          register volatile int *outSamples1,
                          register volatile int *outSamples2,
                          register volatile int *outSamples3,
                          register volatile int *outSamples4,
                          register int num_samples)
{
    register int i;
    register int temp;
    register int temp1;

    /* Read out data from the fifo */
    for (i = 0; i < num_samples; i++)
    {
        if (rcvr_params->packing_mode == P65XX_UNPACKED)
        {
          *outSamples1++ = *inSamples++;
          *outSamples1++ = *inSamples++;
            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            {
            *outSamples2++ = *inSamples++;
            *outSamples2++ = *inSamples++;
            }
        }
        else if (rcvr_params->data_order == P65XX_IQ_ORDER)
        {
          temp = *inSamples++;
          *outSamples1++ = temp >> 16;           /* Real(I) Value   */
          *outSamples1++ = (temp << 16) >> 16;   /* Imag(Q) Value   */

            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            {
              temp = *inSamples++;
              *outSamples2++ = temp >> 16;           /* Real(I) Value   */
              *outSamples2++ = (temp << 16) >> 16;   /* Imag(Q) Value   */
            }
      }
        else if (rcvr_params->data_order == P65XX_RECEIVER_ORDER)
        {
            temp = *inSamples++;     /* Imag (Q) Value */
            temp1 = *inSamples++;    /* Real (I) Value */
          *outSamples1++ = (temp1 << 16) >> 16;  /* Real(I) Value   */
          *outSamples1++ = (temp << 16) >> 16;   /* Imag(Q) Value   */

            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            {
              *outSamples3++ = temp1 >> 16;  /* Real(I) Value   */
              *outSamples3++ = temp >> 16;   /* Imag(Q) Value   */

                temp = *inSamples++;     /* Imag (Q) Value */
```

```
                temp1 = *inSamples++;    /* Real (I) Value */
              *outSamples2++ = (temp1 << 16) >> 16;  /* Real(I) Value   */
              *outSamples2++ = (temp << 16) >> 16;   /* Imag(Q) Value   */

              *outSamples4++ = temp1 >> 16;  /* Real(I) Value   */
              *outSamples4++ = temp >> 16;   /* Imag(Q) Value   */
            }
            else
            {
              *outSamples2++ = temp1 >> 16;  /* Real(I) Value   */
              *outSamples2++ = temp >> 16;   /* Imag(Q) Value   */
            }
        }
    }
}
/*************************************************************************
 Function: P6516ReadFifoReal

 Description:  This function reads real data from the fifo.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
        p6516Reg - Pointer to the 6516's address table
        samples - Pointer to a data buffer to store data
        num_samples - Number of samples to be read
        inputFifo - Fifo to be read (1 - 8)
 Outputs:
       Data is stored to the data buffer
 Returns:
        samples - Address of the next available location in data buffer
 Notes:
        Data is sign extended and right justified
*************************************************************************/
int *P6516ReadFifoReal(P6516_RCVR_PARAMS *rcvr_params,
                       P6516_REG_ADDR *p6516Reg,
                       register volatile int *samples,
                       register int num_samples, unsigned int inputFifo)
{
    register volatile int *fifo = (int *)p6516Reg->Fifo +
                                  ((0x8000 * (inputFifo - 1)) / 4);
    int          num_samples_to_read;
    register int temp;
    register int i;

    while (num_samples > 0)
    {
        /* wait FIFO half-full */
      while ((*p6516Reg->Stat_Reg_2 & (0x100 << (inputFifo-1))) != 0);

      /* Determine how many samples left to read */
      if (num_samples <= P6516_FIFO_SIZE/2)
            num_samples_to_read = num_samples;
      else
          num_samples_to_read = P6516_FIFO_SIZE/2;

      /* Read out data from the fifo */
      for (i = 0; i < num_samples_to_read; i++)
      {
          temp = *fifo;               /* Store real data          */
          *samples++ = temp >> 16;    /* Real(I) Value            */
      }

      /* Decrement number of samples needed */
      num_samples -= num_samples_to_read;
```

```
    }
    return((int *)samples);
}
/*************************************************************************
 Function: P6516ReadCommPortReal

 Description:  This function reads real data from the Comm Port.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
        samples - Pointer to a data buffer to store data
        num_samples - Number of samples to be read
        inputCommPort - Comm Port to be read (0 - 5)
 Outputs:
       Data is stored to the data buffer in the following format:
             Unpacked Data -  Real,Real,Real,...        Sign Extended
                                                        Right Justified
             Packed, Time Ordered - Selected Rcvr Smpl1/Smpl2,Smpl1/Smpl2 ...
             Packed, Receiver Ordered - Real(Even)/Real(Odd),
                                        Real(Even),Real(Odd).
 Returns:
        samples - Address of the next available location in data buffer
 Notes:
*************************************************************************/
int *P6516ReadCommPortReal(P6516_RCVR_PARAMS *rcvr_params,
                           register volatile int *samples,
                           register int num_samples,
                           unsigned int inputCommPort)
{
    volatile unsigned int *comStatus = (unsigned int *) 0x100040 +
                                                      0x10 * inputCommPort;
    register volatile int *comInput = (int *)comStatus + 1;
    int          num_samples_to_read;
    register int i;

    if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
        num_samples *= 2;

    if (rcvr_params->data_order == P65XX_TIME_ORDER)
        num_samples /= 2;

    /* Read out data from the Comm Port */
    for (i = 0; i < num_samples; i++)
    {
        while ((*comStatus & 0x1e00) == 0);
        if (rcvr_params->packing_mode == P65XX_UNPACKED)
        {
            *samples++ = *comInput >> 16;    /* Real Value    */
        }
        else if (rcvr_params->data_order == P65XX_RECEIVER_ORDER)
        {
          *samples++ = *comInput;     /* Real Even/Real Odd */
        }
        else if (rcvr_params->data_order == P65XX_TIME_ORDER)
        {
          *samples++ = *comInput;     /* Selected Rcvr Smpl2/Smpl1 */
        }
    }

    return((int *)samples);
}
/*************************************************************************
 Function: P6516UnpackRealData
```

```
 Description:  This function unpacks a real data buffer read from a Comm Port
               based on the packing mode selected.
 Inputs:
        rcvr_params - Pointer to the P65XX_RCVR_PARAMS structure containing
                      the receiver settings
        inSamples - Pointer to a data buffer containing complex data
        outSamples1 - Pointer to the output data buffer for 1st receiver
        outSamples2 - Pointer to the output data buffer for 2nd receiver
        outSamples3 - Pointer to the output data buffer for 3rd receiver
        outSamples4 - Pointer to the output data buffer for 4th receiver
        num_samples - Number of samples to be unpack per receiver
 Outputs:
        Data is stored to the output data buffer in the following format:
             Real,Real,Real,...
 Returns:
 Notes:
        Data is sign extended and right justified
*****************************************************************************/
void P6516UnpackRealData(P6516_RCVR_PARAMS *rcvr_params,
                         register volatile int *inSamples,
                         register volatile int *outSamples1,
                         register volatile int *outSamples2,
                         register volatile int *outSamples3,
                         register volatile int *outSamples4,
                         register int num_samples)
{
    register int i;
    register int temp;
    register int temp1;

    if (rcvr_params->data_order == P65XX_TIME_ORDER)
        num_samples /= 2;

    /* Read out data from the fifo */
    for (i = 0; i < num_samples; i++)
    {
        if (rcvr_params->packing_mode == P65XX_UNPACKED)
        {
          *outSamples1++ = *inSamples++;
            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            *outSamples2++ = *inSamples++;
        }
        else if (rcvr_params->data_order == P65XX_TIME_ORDER)
        {
          temp = *inSamples++;
          *outSamples1++ = (temp << 16) >> 16;   /* Real Sample 1   */
          *outSamples1++ = temp >> 16;           /* Real Sample 2   */

            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            {
              temp = *inSamples++;
              *outSamples2++ = (temp << 16) >> 16;   /* Real Sample 1   */
              *outSamples2++ = temp >> 16;           /* Real Sample 2   */
            }
        }
        else if (rcvr_params->data_order == P65XX_RECEIVER_ORDER)
        {
            temp = *inSamples++;
          *outSamples1++ = (temp << 16) >> 16;  /* Real(I) Value   */
            if (rcvr_params->fifo_mode == P6516_DUAL_RCVR)
            {
              *outSamples3++ = temp >> 16;          /* Real(I) Value   */

                temp = *inSamples++;
```

```
                *outSamples2++ = (temp << 16) >> 16;  /* Real(I) Value   */
                *outSamples4++ = temp >> 16;          /* Real(I) Value   */
              }
              else
                *outSamples2++ = temp >> 16;          /* Real(I) Value   */
          }
      }
}
```

```
/*****************************************************************************
*
*   File : main.c
*
*   Description: This program demonstrates how to initialize the 6514
*                using it's VMEBus interface.  Data is received from the
*                65XX Front Panel Comm Ports.
*
*
*   $History: main.c $
*
*
*****************************************************************************/
#include <stdlib.h>
#include <snstdio.h>
#include "math.h"
#include "65xxcmd.h"
#include "4285.h"
#include "vme.h"

/* Define the Digital Receiver Board Model */
#define P6514     0
#define P6516     1

#if P6514
#include "6514.h"

#define P65XX_NUM_RCVRS        P6514_NUM_RCVRS
#define P65XX_NUM_INPUTS       P6514_NUM_INPUT_CHANNELS

typedef P6514_REG_ADDR         P65XX_REG_ADDR;
typedef P6514_RCVR_PARAMS      P65XX_RCVR_PARAMS;
typedef P6514_INPUT_PARAMS     P65XX_INPUT_PARAMS;
#define RCVR_PARAM_TBL_SIZE    sizeof(P6514_RCVR_PARAMS)
#define INPUT_PARAM_TBL_SIZE   sizeof(P6514_INPUT_PARAMS)

#define InitRegAddr            P6514InitRegAddr
#define ResetFifo              P6514ResetFifo
#define ReleaseFifo            P6514ReleaseFifo
#define SetInputResetMode      P6514SetInputResetMode
#define ReadCommPortCmplx      P6514ReadCommPortCmplx
#define ReadCommPortReal       P6514ReadCommPortReal
#define UnpackCmplxData        P6514UnpackCmplxData
#define UnpackRealData         P6514UnpackRealData

#endif

#if P6516
#include "6516.h"

#define P65XX_NUM_RCVRS        P6516_NUM_RCVRS
#define P65XX_NUM_INPUTS       P6516_NUM_INPUT_CHANNELS

typedef P6516_REG_ADDR         P65XX_REG_ADDR;
typedef P6516_RCVR_PARAMS      P65XX_RCVR_PARAMS;
typedef P6516_INPUT_PARAMS     P65XX_INPUT_PARAMS;
#define RCVR_PARAM_TBL_SIZE    sizeof(P6516_RCVR_PARAMS)
#define INPUT_PARAM_TBL_SIZE   sizeof(P6516_INPUT_PARAMS)

#define InitRegAddr            P6516InitRegAddr
#define ResetFifo              P6516ResetFifo
#define ReleaseFifo            P6516ReleaseFifo
#define SetInputResetMode      P6516SetInputResetMode
#define ReadCommPortCmplx      P6516ReadCommPortCmplx
```

```
#define ReadCommPortReal       P6516ReadCommPortReal
#define UnpackCmplxData        P6516UnpackCmplxData
#define UnpackRealData         P6516UnpackRealData

#endif

/* The following constants define some of the parameters to this program */

/* RCVR_MASK - Defines which receivers are initialized and supplying data.
               Only two receiver pairs can be specified(i.e. 1,2,3,&4,
               5,6,7,&8, 9,10,11,&12, or 13,14,15,&16) when the Packing Mode
               is Packed. When the Packing Mode is Unpacked, only one
               receiver pair can be specified(i.e. 1&2, 3&4, 5&6, 7&8,
               9&10, 11&12, 13&14, 15&16).
*/
#define RCVR_MASK           (P65XX_RECEIVER_5 | P65XX_RECEIVER_6 | \
                             P65XX_RECEIVER_7 | P65XX_RECEIVER_8)

/* DATA_SOURCE - Defines which input channel is suppliying data to the
                 receiver
*/
#define DATA_SOURCE         P65XX_INPUT_A

/* PACKING_MODE - Defines whether Packed or Unpacked Data is sent to the front
                  panel Output Comm Ports.
*/
#define PACKING_MODE        P65XX_PACKED

/* DATA_MODE - Defines whether the data supplied by the receiver is real or
               complex.
*/
#define DATA_MODE           P65XX_COMPLEX

/* DECIMATION_RATE - Defines the output data rate of the receiver.
                     Output Rate = (INPUT_CLOCK_RATE/ DECIMATION_RATE)
*/
#define DECIMATION_RATE     64

/* LO - Defines the tuning frequency of the receivers
*/
#define LO                  (float) 12.0e6

/* INPUT_CLOCK_RATE - Defines the input clock rate to the receivers
*/
#define INPUT_CLOCK_RATE    (float) 40.0e6

/* INPUT_CROSSOVER - Defines whether Input Channels A & B crossover to
                     Channels C & D.  This effects the 6516 only.
*/
#define INPUT_CROSSOVER     (P6516_CROSSOVER_A_TO_C_ON |\
                             P6516_CROSSOVER_B_TO_D_ON)

/* DATA_COMM_PORT - Defines the C40 Comm Port which will receive data from
                    the Digital Receiver Board.
*/
#define DATA_COMM_PORT      5

/* Number of Samples to acquire - In complex Mode, this program refers to */
/* one sample as one real plus one imaginary sample.                      */
#define NUM_CMPLX_SAMPLES        512
#define NUM_REAL_SAMPLES         1024

/* Base Address of the 65XX Board */
#define P65XX_A16_BASE_ADDR      0xFF00
```

```
#define P65XX_VME_BASE_ADDR      0x80000000
#define P65XX_VME_ACCESS_MODE    VME_AM_A32_NPR_DAT

/* Data Buffers for storing received data */
int     inSamples[NUM_REAL_SAMPLES * 4];
int     Samples[NUM_REAL_SAMPLES * 4];

/*****************************************************************************
 Function: set65XXRcvrParams

 Description: This routine initializes the Receiver Parameter Table for
              a specified receiver.
 Inputs:
        params - Pointer to a P65XX_RCVR_PARAMS table entry.
        rcvr - Receiver being setup(1 - 16)
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void set65XXRcvrParams(P65XX_RCVR_PARAMS *params, int rcvr)
{
    params->decimation_rate = DECIMATION_RATE;
    params->data_mode = DATA_MODE;
    params->data_source = DATA_SOURCE;
    params->lo = LO;
    params->input_clock_rate = INPUT_CLOCK_RATE;
    params->packing_mode = PACKING_MODE;
    params->data_transfer = P65XX_COMM_PORT;
    params->data_order = P65XX_RECEIVER_ORDER;
    params->fifo_select = (rcvr & 0x2) >> 1;
    params->tag_enable = P65XX_TAG_OFF;
    params->flip_spectrum = P65XX_FLIP_OFF;
    params->offset_spectrum = P65XX_OFFSET_ON;
#if P6514
    params->wait_ram_full = P65XX_WAIT_RAM_FULL_ON;
    params->fifo_mode = P6514_DUAL_RCVR;
#endif
#if P6516
    params->wait_ram_full = P65XX_WAIT_RAM_FULL_ON;
    params->fifo_mode = P6516_DUAL_RCVR;
#endif
}
/*****************************************************************************
 Function: set65XXInputParams

 Description: This routine initializes the Input Channel Parameter Table for
              a specified input channel.
 Inputs:
        params - Pointer to a P65XX_INPUT_PARAMS table entry.
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void set65XXInputParams(P65XX_INPUT_PARAMS *params)
{
    params->threshold_level = P65XX_THRSHLD_DETECT_OFF;
    params->threshold_cross_cnt = P65XX_CROSS_CNT_2;
#if P6514
    params->input_reset_mode = P6514_SW_RESET_IN_SRC | P6514_SW_RESET_OFF |
                               P6514_RESET_OUT_DIS;
#endif
#if P6516
    params->input_reset_mode = P6516_SW_RESET_IN_SRC | P6516_SW_RESET_OFF |
                               P6516_RESET_OUT_DIS;
```

```
    params->input_crossover = INPUT_CROSSOVER;
#endif
}

/*****************************************************************************
 Function: main

 Description: This program initializes the specified receivers and
              acquires data.

*****************************************************************************/
main()
{
    register int          i;
    register int          j;
    int                   temp;
    unsigned int          cmdParams[2];
    unsigned long         status;
    int                   *samples;
    P65XX_REG_ADDR        p65xxReg;
    P65XX_RCVR_PARAMS     p65xxRcvrParams[P65XX_NUM_RCVRS];
    P65XX_INPUT_PARAMS    p65xxInputParams;
    unsigned int          handle[P65XX_NUM_RCVRS] = {0, 0, 0, 0, 0, 0, 0, 0,
                                                     0, 0, 0, 0, 0, 0, 0, 0};
    unsigned long         returnData[2];
    unsigned long         numRData;
    volatile unsigned int *com;
    unsigned long         a_rcvr;
    unsigned long         numRcvrs = 0;
    unsigned long         a16Base;
    unsigned long         vmeBase;
    unsigned long         input_mask;

    /* Setup the 4285 for A16 Accesses */
    a16Base = P4285SetupVMETransfer(VME_AM_A16_NPR_ACC, P65XX_A16_BASE_ADDR,
                                    32);

    /* Set the 65XX's A32/A24 Base Address */
    P4285Set65XXBaseAddress(a16Base, P65XX_VME_BASE_ADDR,
                            P65XX_VME_ACCESS_MODE);

    /* Setup the 4285 for A32 Accesses */
    vmeBase = P4285SetupVMETransfer(VME_AM_A32_NPR_DAT, P65XX_VME_BASE_ADDR,
                                    32);

    /* Initialize Pointers to 65XX Registers */
    InitRegAddr(vmeBase, &p65xxReg);

    /* Initialize SNIO interface */
    if (snstdio_init(NULL) != OK)
        exit(1);
    atexit((void (*)())snstdio_shutdown);

    /* Close all prior open connections */
    if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, 0,
                        P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_ALL_CONNECTIONS, 0),
                        0, NULL, 0, &numRData, returnData)) !=
                                                    P65XX_COMMAND_COMPLETE)
    {
        printf("Error Closing All Connections - Status = %lx\n", status);
        exit(1);
    }

    /* Reset all 65XX Registers */
```

```
    if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, 63,
                            P65XX_BUILD_CMD_WAIT(P65XX_RESET_REGS, 0),
                            0, NULL, 0, &numRData, returnData)) !=
                                                P65XX_COMMAND_COMPLETE)
    {
        printf("Error Resetting Registers - Status = %lx\n", status);
        exit(1);
    }

    /* Initialize Input Channels */
#if P6514
    input_mask = P65XX_INPUT_A | P65XX_INPUT_B;
#endif
#if P6516
    input_mask = P65XX_INPUT_A | P65XX_INPUT_B |
                 P65XX_INPUT_C | P65XX_INPUT_D;
#endif
    /* Initialize Input Channel Parameter Data structure */
    set65XXInputParams(&p65xxInputParams);

    /* Initialize the Input Channel */
    if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, 11,
                          P65XX_BUILD_CMD_WAIT(P65XX_INIT_INPUT_CHANNELS, 0),
                          sizeof(P65XX_INPUT_PARAMS),
                          (void *)&p65xxInputParams, input_mask,
                          &numRData, returnData)) != P65XX_COMMAND_COMPLETE)
    {
        printf("Error Initializing Input Channels - Status = %lx\n", status);
        exit(1);
    }

    /* Open a connection and initialize each receiver  */
    for (i = 0; i < P65XX_NUM_RCVRS; ++i)
    {
        if (RCVR_MASK & (1 << i))
        {
            /* Initialize Parameter Data structure */
            set65XXRcvrParams(&p65xxRcvrParams[i], i + 1);

            /* Establish Connection for this receiver */
            if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, 55,
                              P65XX_BUILD_CMD_WAIT(P65XX_OPEN_CONNECTION, 0),
                              0, NULL, 1 << i, &numRData, returnData)) !=
                                                  P65XX_COMMAND_COMPLETE)
            {
                printf("Error Opening Connection Rcvr %d - Status = %lx\n",
                                 i + 1, status);
                exit(1);
            }

            /* Save the handle corresponding to this connection */
            handle[i] = *returnData;

            /* Initialize the Receiver */
            if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, 33,
                              P65XX_BUILD_CMD_WAIT(P65XX_INIT_RCVRS, handle[i]),
                              sizeof(P65XX_RCVR_PARAMS),
                              (void *)&p65xxRcvrParams[i], 1 << i,
                              &numRData, returnData)) != P65XX_COMMAND_COMPLETE)
            {
                printf("Error Initializing Rcvr %d - Status = %lx\n",
                                 i + 1, status);
                exit(1);
            }
```

```
            /* Save an index into one of the valid receivers  */
            /* data structures                                */
            a_rcvr = i;
        }
    }

    /* Put the Receiver in a RESET Mode.  This turns off the data stream */
    /* going into the fifo.  This only has to occur once after the       */
    /* receivers have been initialized.                                  */
#if P6514
    p65xxInputParams.input_reset_mode = P6514_SW_RESET_ON |
                                        P6514_SW_RESET_IN_SRC;
#endif
#if P6516
    p65xxInputParams.input_reset_mode = P6516_SW_RESET_ON |
                                        P6516_SW_RESET_IN_SRC;
#endif
    SetInputResetMode(&p65xxInputParams, &p65xxReg, input_mask);

    /* Put the Receiver in a Acquire Mode */
#if P6514
    p65xxInputParams.input_reset_mode = P6514_SW_RESET_OFF |
                                        P6514_SW_RESET_IN_SRC;
#endif
#if P6516
    p65xxInputParams.input_reset_mode = P6516_SW_RESET_OFF |
                                        P6516_SW_RESET_IN_SRC;
#endif
    SetInputResetMode(&p65xxInputParams, &p65xxReg, input_mask);

    /* Main Data Acquisition and Processing Loop */
    while (1)
    {
        /* Reset all the Receiver Fifos */
        ResetFifo(&p65xxRcvrParams[a_rcvr], &p65xxReg, RCVR_MASK);

        /* Clear out old data from the Data Comm Port */
        com = (volatile unsigned int *)(0x100040 + 0x10 * DATA_COMM_PORT);
        while ((*com & 0x1e00) != 0)
            temp = *(com + 1);

      /* Release the Fifos and begin acquisition */
        ReleaseFifo(&p65xxRcvrParams[a_rcvr], &p65xxReg, RCVR_MASK);

      /* Read the data from the 65XX Fifo or Comm Port */
        if (p65xxRcvrParams[a_rcvr].data_mode == P65XX_COMPLEX)
        {
            (void) ReadCommPortCmplx(&p65xxRcvrParams[a_rcvr], inSamples,
                                     NUM_CMPLX_SAMPLES, DATA_COMM_PORT);
            UnpackCmplxData(&p65xxRcvrParams[a_rcvr], inSamples,
                        &Samples[0], &Samples[NUM_REAL_SAMPLES],
                        &Samples[NUM_REAL_SAMPLES * 2],
                        &Samples[NUM_REAL_SAMPLES * 3], NUM_CMPLX_SAMPLES);

        }
        else
        {
            (void) ReadCommPortReal(&p65xxRcvrParams[a_rcvr], inSamples,
                                    NUM_REAL_SAMPLES, DATA_COMM_PORT);
            UnpackRealData(&p65xxRcvrParams[a_rcvr], inSamples,
                         &Samples[0], &Samples[NUM_REAL_SAMPLES],
                         &Samples[NUM_REAL_SAMPLES * 2],
                         &Samples[NUM_REAL_SAMPLES * 3], NUM_REAL_SAMPLES);
```

```
        }

        /* For Each Receiver, Process the data  */
        numRcvrs = 0;
        for (i = 0; i < P65XX_NUM_RCVRS; ++i)
        {
            if (RCVR_MASK & (1 << i))
            {
                /* Add processing for receiver channel here */
                ++numRcvrs;
            }
        }
    }
    /* Close the connection for each open connection */
    for (i = 0; i < P65XX_NUM_RCVRS; ++i)
    {
        if (RCVR_MASK & (1 << i))
        {
            /* Establish Connection for this receiver */
            if ((status = P4285Send65XXCommandWait(a16Base, vmeBase, i,
                      P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_CONNECTION, handle[i]),
                      0, NULL, 0, &numRData, returnData)) !=
                                                    P65XX_COMMAND_COMPLETE)
            {
                printf("Error Closing Connection Rcvr %d - Status = %lx\n",
                                i + 1, status);
                exit(1);
            }
        }
    }
}
```

```
/*****************************************************************************
*
*   File : cmd85.c
*
*   Description: This file contains routines for sending commands to the
*                65XX Receiver Boards VMEBus Interface from the 4285
*                Baseboard.
*
*****************************************************************************/
#include <stdlib.h>
#include "65xxcmd.h"
#include "4285.h"
#include "vme.h"

/*****************************************************************************
 Function: P4285Send65XXCommandWait

 Description: This routine sends a command to the 65XX board via the VME
              Bus Interface and waits for the command to complete.
 Inputs:
        a16Base - C40 Local Address for accessing 65XX board's A16
                  Base Address
        vmeBase - C40 Local Address for accessing 65XX board's VME
                  Base Address
        cmdTblIndex - Index into Command Table(0 - 63)
        command   - Command being sent
        num_params - The number of parameters which must follow command
        cmdParams - Pointer to the command parameters
        input_rcvr_mask - Input Channel or Receiver mask
        numRdata - Pointer to store the number of returned data
        returnData - Buffer pointer to store the return data
 Outputs:
        numRdata - Number of returnned data
        returnData - Returned Data
 Returns:
        Status - Status of the Command
 Notes: The wait_flag should not be set in the command word.  If the wait
        flag is set, then the 65XX board will generate a VMEBus Interrupt
        when command completes.  This function is not set to respond to the
        VMEBus Interrupt.
*****************************************************************************/
unsigned long P4285Send65XXCommandWait(unsigned long a16Base,
                                       unsigned long vmeBase,
                                       unsigned long cmdTblIndex,
                                       unsigned int command,
                                       unsigned int num_params,
                                       void *cmdParams,
                                       unsigned int input_rcvr_mask,
                                       unsigned long *numRData,
                                       unsigned long *returnData)
{
    unsigned int   i;
    unsigned int   status;
    register volatile unsigned long  *vme_command;

    /* Initialize pointer to Command location in 65XX DPR */
    vme_command = (unsigned long *)(vmeBase + P65XX_VME_COMMAND_ADDR +
                                    cmdTblIndex * P65XX_CMD_TBL_ENTRY_SIZE);

    /* Write Input Channel/Receiver Mask */
    *(vme_command + 1) = input_rcvr_mask;

    /* Write the command parameters */
    for (i = 0; i < num_params; ++i)
```

```
        *(vme_command + 2 + i) = *((unsigned long *)cmdParams + i);

    /* Write Command, Clear Wait Flag */
    *vme_command  = command & ~P65XX_WAIT_COMPLETE;

    /* Interrupt the C40 on the 65XX Board */
    *(unsigned long *)(a16Base + 2) = 0x1;

    /* Wait for Command to Complete */
    while (*vme_command != 0)
    {
        asm("  nop");
        asm("  nop");
        asm("  nop");
        asm("  nop");
    }

    /* Read Status and return data */
    status = *(vme_command + 1);
    if ((numRData != NULL) && (returnData != NULL))
    {
        *numRData = status >> 16;
        for (i = 0; i < *numRData; ++i)
            *returnData++ = *(vme_command + 2 + i);
    }

    return(status & 0xffff);
}
/*****************************************************************************
 Function: P4285Send65XXCommand

 Description: This routine generates a command to the 65XX Board via the
              VMEBus Interface.
 Inputs:
        a16Base - C40 Local Address for accessing 65XX board's A16
                  Base Address
        vmeBase - C40 Local Address for accessing 65XX board's VME
                  Base Address
        cmdTblIndex - Index into Command Table(0 - 63)
        command   - Command being sent
        num_params - The number of parameters which must follow command
        cmdParams - Pointer to the command parameters
        input_rcvr_mask - Input Channel or Receiver mask
 Outputs:
 Returns:
 Notes: Returns immediately to caller without waiting for command to complete
*****************************************************************************/
void P4285Send65XXCommand(unsigned long a16Base, unsigned long vmeBase,
                          unsigned long cmdTblIndex,
                          unsigned int command, unsigned int num_params,
                          void *cmdParams, unsigned int input_rcvr_mask)
{
    unsigned int   i;
    unsigned int   status;
    register volatile unsigned long  *vme_command;

    /* Initialize pointer to Command location in 65XX DPR */
    vme_command = (unsigned long *)(vmeBase + P65XX_VME_COMMAND_ADDR +
                                    cmdTblIndex * P65XX_CMD_TBL_ENTRY_SIZE);

    /* Write Input Channel/Receiver Mask */
    *(vme_command + 1) = input_rcvr_mask;

    /* Write the command parameters */
```

```
    for (i = 0; i < num_params; ++i)
        *(vme_command + 2 + i) = *((unsigned long *)cmdParams + i);

    /* Write Command */
    *vme_command  = command;

    /* Interrupt the C40 on the 65XX Board */
    *(unsigned long *)(a16Base + 2) = 0x1;

}

/***************************************************************************
 Function: P4285Set65XXBaseAddress

 Description:  This routine sets the 65XX Board's A32/A24 Base Address based
               on the specified access mode.
 Inputs:
        a16Base - C40 Local Address for accessing 65XX board's A16
                  Base Address
        vmeBase - C40 Local Address for accessing 65XX board's VME
                  Base Address
        accessMode - Address Modifier
 Outputs:
 Returns:
 Notes:
***************************************************************************/
void P4285Set65XXBaseAddress(unsigned long a16Base, unsigned long vmeBase,
                             unsigned int accessMode)
{
    unsigned int *a16 = (unsigned int *)a16Base;

    if ((accessMode & VME_AM_MASK) == VME_A24_AM)        /* If A24   */
        *(a16 + 1) = (((vmeBase >> 19) & 0x1e) | 0x20);
    else                                                 /* Else A32 */
        *(a16 + 1) = ((vmeBase >> 16) & 0xffc0);

}
/***************************************************************************
 Function: P4285Set65XXIRQLevelVector

 Description:  This routine sets the 65XX Board's Interrupt Level and Vector
 Inputs:
        a16Base - C40 Local Address for accessing 65XX board's A16 Base
                  Address
        irqLevel - VMEBus Interrupt Level (1 - 7)
        irqVector - VMEBus Interrupt Vector (0 - 255)
 Outputs:
 Returns:
 Notes:
***************************************************************************/
void P4285Set65XXIRQLevelVector(unsigned long a16Base, unsigned long irqLevel,
                                unsigned long irqVector)
{
    unsigned int *a16 = (unsigned int *)a16Base;

    *a16 = (*a16 & ~0x03ff) | (irqLevel << 8) | irqVector;
}
```

```
/*****************************************************************************
*
*   File : vme85.c
*
*   Description: This file contains routines for setting VME Master Access
*                and DMA capabilities on the 4285.
*
*****************************************************************************/
#include "4285.h"
#include "vme.h"

/*****************************************************************************
 Function: P4285SetupVMETransfer

 Description: This routine sets up the 4285 for mastering the VMEBus and
              returns the local C40 Address for accessing the specified
              VMEBus Address.
 Inputs:
        addressModifier - The address modifier defining the VMEBus access mode
        address         - The VMEBus Address being accessed
        dataWidth       - The access data width (16 or 32)
 Outputs:
 Returns:
        Returns the Local Address for accessing the requested VME Address
 Notes:
*****************************************************************************/
unsigned long P4285SetupVMETransfer(unsigned int addressModifier,
                                    unsigned long address,
                                    unsigned int dataWidth)
{
    unsigned long access_addr;
    unsigned int  addr_type = addressModifier & 0x6;
    unsigned int  addr_mod =  addressModifier & VME_AM_MASK;
    unsigned int  vme_page = 0;

    switch(addr_mod)
    {
        case VME_A32_AM:
            vme_page = address >> 26;
            if (dataWidth == 16)
                access_addr = 0xb1000000L;
            else
                access_addr = 0xb0000000L;
            break;
        case VME_A24_AM:
            if (dataWidth == 16)
                access_addr = 0xb3000000L;
            else
                access_addr = 0xb2000000L;
            break;
        case VME_A16_AM:
            if (dataWidth == 16)
                access_addr = 0xb5000000L;
            else
                access_addr = 0xb4000000L;
            break;
    }

    /* Make dummy read to global SRAM before modifying VME Page */
    (void) *P4285_DUMMY_GLOBAL_ACCESS;

    *P4285_LCL_CNTL_REG = (*P4285_LCL_CNTL_REG & 0x0fff) | (vme_page << 12) |
                          (addr_type << 9);
```

```
    access_addr |= ((address & 0xFFFFFFL) / 4);

    return(access_addr);
}

/*****************************************************************************
 Function: P4285SetupVICAccess

 Description: This routines sets up the 4285 for accessing the VIC.
 Inputs:
       addr_bits - This parameter specifies the settings of bits D8 and D9
                   in the Local Status and Control Register
 Outputs:
 Returns:
        Previous setting of bits D8 and D9
 Notes:
*****************************************************************************/
/* Setup Access to VIC Registers */
unsigned int P4285SetupVICAccess(unsigned int addr_bits)
{
    unsigned int old_access;

    /* Make dummy read to global SRAM before modifying Address Bits */
    (void) *P4285_DUMMY_GLOBAL_ACCESS;

    /* Save old settings */
    old_access = (*P4285_LCL_CNTL_REG >> 8) & 0x3;

    /* Set VIC access */
    *P4285_LCL_CNTL_REG = (*P4285_LCL_CNTL_REG & ~0x300) | (addr_bits << 8);

    return(old_access);
}


/*****************************************************************************
 Function: P4285DMATransferToVME

 Description:  This routine starts up a DMA transfer to VME and waits for
               it to complete.
 Inputs:
        source - The local address in Spooler or Global SRAM
        dest - The VME Address receiving the data
        size - The number of bytes to transfer
        wait_complete - Wait for completion flag. This routine will not
                        return to caller until DMA has completed when this
                        bit is set.
 Outputs:
        status - This return status contains the current settings in the
                 VIC's DMA Status and Bus Error Status Registers.
                 D0 - D7 contain the DMA Status Register contents
                 D8 - D15 contain the Bus Error Status register contents
 Returns:
 Notes:
*****************************************************************************/
unsigned int P4285DMATransferToVME(unsigned int *source, unsigned int *dest,
                                   unsigned int size,
                                   unsigned int wait_complete)
{
    unsigned int old_access;
    unsigned int status = 0;

    /* Setup Access to VIC Registers */
    old_access = P4285SetupVICAccess(0x3);
```

```
    /* Initialize DMA Registers */
    *P4285_VIC_DMASR = 0x0;
    *P4285_VIC_BESR = 0x0;
    *P4285_VIC_BTLR0 = size & 0xff;
    *P4285_VIC_BTLR1 = (size >> 8) & 0xff;
    *P4285_VIC_BTLR2 = (size >> 16) & 0xffff;

    *P4285_VIC_BTDR = 0x0d;
    *P4285_VIC_BTCR = 0x44;

    /* Restore VME Address Bits */
    (void) P4285SetupVICAccess(old_access);

    /* Start DMA Transfer */
    if (((unsigned int)source & 0xe0000000L) == 0xa0000000L)
        *dest = ((unsigned int)source & 0x3ffff) << 2;
    else  /* Global Memory */
        *dest = (((unsigned int)source & 0x3ffff) << 2) | 0x200000;

    if (wait_complete)
    {
        status = P4285WaitDMAComplete();
        P4285SetDMAOff();
    }
    return(status);
}
/*****************************************************************************
 Function: P4285DMATransferFromVME

 Description:  This routine starts up a DMA transfer from VME and waits for
               it to complete.
 Inputs:
        source - The VME Address containing the data
        dest - The Local C40 Address receiving the data
        size - The number of bytes to transfer
        wait_complete - Wait for completion flag. This routine will not
                        return to caller until DMA has completed when this
                        bit is set.
 Outputs:
        status - This return status contains the current settings in the
                 VIC's DMA Status and Bus Error Status Registers.
                 D0 - D7 contain the DMA Status Register contents
                 D8 - D15 contain the Bus Error Status register contents
 Returns:
 Notes:
*****************************************************************************/
unsigned int P4285DMATransferFromVME(unsigned int *source,
                                     unsigned int *dest,
                                     unsigned int size,
                                     unsigned int wait_complete)
{
    unsigned int old_access;
    unsigned int status = 0;

    /* Setup Access to VIC Registers */
    old_access = P4285SetupVICAccess(0x3);

    *P4285_VIC_DMASR = 0x0;
    *P4285_VIC_BESR = 0x0;
    *P4285_VIC_BTLR0 = size & 0xff;
    *P4285_VIC_BTLR1 = (size >> 8) & 0xff;
    *P4285_VIC_BTLR2 = (size >> 16) & 0xffff;
```

```
    *P4285_VIC_BTDR = 0x0d;
    *P4285_VIC_BTCR = 0x54;

    /* Restore VME Address Bits */
    (void) P4285SetupVICAccess(old_access);

    /* Start DMA Transfer */
    if (((unsigned int)dest & 0xe0000000L) == 0xa0000000L)
        *source = ((unsigned int)dest & 0x3ffff) << 2;
    else  /* Global Memory */
        *source = (((unsigned int)dest & 0x3ffff) << 2) | 0x200000;

    if (wait_complete)
    {
        status = P4285WaitDMAComplete();
        P4285SetDMAOff();
    }
    return(status);
}
/*****************************************************************************
 Function: P4285WaitDMAComplete

 Description:  This routine waits for a DMA transfer to complete.
 Inputs:
 Outputs:
        status - This return status contains the current settings in the
                 VIC's DMA Status and Bus Error Status Registers.
                 D0 - D7 contain the DMA Status Register contents
                 D8 - D15 contain the Bus Error Status register contents
 Returns:
 Notes:
*****************************************************************************/
unsigned int P4285WaitDMAComplete()
{
    unsigned int dmasr;
    unsigned int besr = 0;
    unsigned int status = 0;
    unsigned int old_access;

    /* Setup Access to VIC Registers */
    old_access = P4285SetupVICAccess(0x3);

    while (((dmasr = *P4285_VIC_DMASR) & 0x1) != 0)
    {
        besr = *P4285_VIC_BESR & 0x7f;
        if (besr || (dmasr & 0x1e))
            break;
    }

    /* Mask off DMA Complete Bit */
    status = (dmasr & 0x1e) | (besr << 8);

    /* Restore VME Address Bits */
    (void) P4285SetupVICAccess(old_access);

    return(status);

}

/*****************************************************************************
 Function: P4285SetDMAOff

 Description:  This routine turns off the DMA transfer.
 Inputs:
```

```
 Outputs:
 Returns:
 Notes: The DMA transfer must be turned off upon it's completion
*****************************************************************************/
void P4285SetDMAOff()
{
    unsigned int old_access;
    unsigned int temp;

    /* Setup Access to VIC Registers */
    old_access = P4285SetupVICAccess(0x3);

    /* Turn off DMA */
    *P4285_VIC_BTCR = 0x0;

    /* Restore VME Address Bits */
    (void) P4285SetupVICAccess(old_access);
}

/*****************************************************************************
 Function: P4285SetDMACompleteInterrupt

 Description: This routine sets up the VIC to generate a DMA Completion
              interrupt.
 Inputs:
        ipl - The ipl level in which the interrupt should be generated.
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void P4285SetDMACompleteInterrupt(unsigned int ipl)
{
    unsigned int old_access;

    /* Setup Access to VIC Registers */
    old_access = P4285SetupVICAccess(0x3);

    if (ipl > 0)
        *P4285_VIC_DMASICR = ipl;
    else
        *P4285_VIC_DMASICR = 0x80;

    /* Restore VME Address Bits */
    (void) P4285SetupVICAccess(old_access);

}
```

```
/*****************************************************************************
*
*   File :  cmdvxw.h
*
*   Description:  This file contains function prototypes for interfacing
*                 a Single Board Computer running vxWorks to a 65XX
*                 Digital Receiver Board.
*
*****************************************************************************/
#ifndef CMDVXW_H
#define CMDVXW_H

unsigned long VxWSend65XXCommandWait(unsigned long a16Base,
                                     unsigned long vmeBase,
                                     unsigned long cmdTblIndex,
                                     unsigned int command,
                                     unsigned int num_params,
                                     void *cmdParams,
                                     unsigned int input_rcvr_mask,
                                     unsigned long *numRData,
                                     unsigned long *returnData);
void VxWSend65XXCommand(unsigned long a16Base, unsigned long vmeBase,
                        unsigned long cmdTblIndex,
                        unsigned int command, unsigned int num_params,
                        void *cmdParams, unsigned int input_rcvr_mask);
void VxWSet65XXBaseAddress(unsigned long a16Base, unsigned long vmeBase,
                           unsigned int accessMode);
void VxWSet65XXIRQLevelVector(unsigned long a16Base, unsigned long irqLevel,
                              unsigned long irqVector);

#endif
```

```
/*****************************************************************************
*
*   File : main65XX.c
*
*   Description: This program demonstrates how to initialize and receive data
*                using the VMEBus Interface for the 6514 and 6516 Digital
*                Receiver Board.
*
*****************************************************************************/

#include "vxWorks.h"
#include "stdio.h"
#include "stdlib.h"
#include "sysLib.h"
#include "msgQLib.h"
#include "iv.h"
#include "vme.h"
#include "taskLib.h"
#include "usrLib.h"
#include "cmdvxw.h"
#include "65xxcmd.h"

/* Define the Digital Receiver Board Model */
#define P6514     0
#define P6516     1

#if P6514
#include "6514.h"

#define P65XX_NUM_FIFOS        8
#define P65XX_NUM_RCVRS        P6514_NUM_RCVRS
#define P65XX_NUM_INPUTS       P6514_NUM_INPUT_CHANNELS

typedef P6514_REG_ADDR         P65XX_REG_ADDR;
typedef P6514_RCVR_PARAMS      P65XX_RCVR_PARAMS;
typedef P6514_INPUT_PARAMS     P65XX_INPUT_PARAMS;
#define RCVR_PARAM_TBL_SIZE    sizeof(P6514_RCVR_PARAMS)
#define INPUT_PARAM_TBL_SIZE   sizeof(P6514_INPUT_PARAMS)

#define InitRegAddr            P6514InitRegAddr
#define ResetFifo              P6514ResetFifo
#define ReleaseFifo            P6514ReleaseFifo
#define SetInputResetMode      P6514SetInputResetMode

#endif
#if P6516
#include "6516.h"

#define P65XX_NUM_FIFOS        8
#define P65XX_NUM_RCVRS        P6516_NUM_RCVRS
#define P65XX_NUM_INPUTS       P6516_NUM_INPUT_CHANNELS

typedef P6516_REG_ADDR         P65XX_REG_ADDR;
typedef P6516_RCVR_PARAMS      P65XX_RCVR_PARAMS;
typedef P6516_INPUT_PARAMS     P65XX_INPUT_PARAMS;
#define RCVR_PARAM_TBL_SIZE    sizeof(P6516_RCVR_PARAMS)
#define INPUT_PARAM_TBL_SIZE   sizeof(P6516_INPUT_PARAMS)

#define InitRegAddr            P6516InitRegAddr
#define ResetFifo              P6516ResetFifo
#define ReleaseFifo            P6516ReleaseFifo
#define SetInputResetMode      P6516SetInputResetMode
#endif
```

```
/* The following constants define some of the parameters to this program */

/* RCVR_MASK - Defines which receivers are initialized and supplying data.
*/
#define RCVR_MASK (P65XX_RECEIVER_1 | P65XX_RECEIVER_2 |\
                   P65XX_RECEIVER_3 | P65XX_RECEIVER_4 |\
                   P65XX_RECEIVER_5 | P65XX_RECEIVER_6 |\
                   P65XX_RECEIVER_7 | P65XX_RECEIVER_8 |\
                   P65XX_RECEIVER_9 | P65XX_RECEIVER_10 |\
                   P65XX_RECEIVER_11 | P65XX_RECEIVER_12 |\
                   P65XX_RECEIVER_13 | P65XX_RECEIVER_14 |\
                   P65XX_RECEIVER_15 | P65XX_RECEIVER_16)

/* DATA_SOURCE - Defines which input channel is suppliying data to
                 the receiver
*/
#define DATA_SOURCE         P65XX_INPUT_A

/* DATA_MODE - Defines whether the data supplied by the receiver is real or
               complex.
*/
#define DATA_MODE           P65XX_COMPLEX

/* DECIMATION_RATE - Defines the output data rate of the receiver.
                     Output Rate = (INPUT_CLOCK_RATE/ DECIMATION_RATE)
*/
#define DECIMATION_RATE     64

/* LO - Defines the tuning frequency of the receivers
*/
#define LO                  12000000L

/* INPUT_CLOCK_RATE - Defines the input clock rate to the receivers
*/
#define INPUT_CLOCK_RATE    40000000L

/* Modes of Operation */
#define POLLING   0       /* Poll for Fifo Full Condition */
#define INTERRUPT 1       /* Wait for Fifo Full Interrupt */

/* A16 and A32 Base Addresses */
#define P65XX_A16_BASE_ADDR    0xff00
#define P65XX_VME_BASE_ADDR    0x40000000L
#define P65XX_VME_ACCESS_MODE  VME_AM_EXT_USR_DATA

/* Interrupt Parameters */
#define P65XX_INTR_LEVEL        4
#define P65XX_INTR_VECTOR       208

/* Number of Samples to acquire - In complex Mode, this program refers to */
/* one sample as one real plus one imaginary sample.                      */
#define NUM_CMPLX_SAMPLES        512
#define NUM_REAL_SAMPLES         1024

/* Data Buffers for storing received data */
int     inSamples[NUM_REAL_SAMPLES * 2];

volatile char    FifoFullIntr;
void fifoFullIntr();

/*************************************************************************
 Function: set65XXRcvrParams
```

```
 Description: This routine initializes the Receiver Parameter Table for
              a specified receiver.
 Inputs:
        params - Pointer to a P65XX_RCVR_PARAMS table entry.
        rcvr - Receiver being setup(1 - 8)
 Outputs:
 Returns:
 Notes:
         All Recievers stored in a Fifo must have the same decimation rate
*****************************************************************************/
void set65XXRcvrParams(P65XX_RCVR_PARAMS *params, int rcvr)
{
    params->decimation_rate = DECIMATION_RATE;
    params->data_mode = DATA_MODE;
    params->data_source = DATA_SOURCE;
    *((unsigned long *)&params->lo) = LO + 2000 * rcvr;
    *((unsigned long *)&params->input_clock_rate) = INPUT_CLOCK_RATE;
    params->data_transfer = P65XX_FIFO;

    /* Packing mode, data order, and fifo_select are only used when the */
    /* data_transfer mode is P65XX_COMM_PORT.                           */
    params->packing_mode = P65XX_UNPACKED;
    params->data_order = P65XX_RECEIVER_ORDER;
    /* For receivers 3,4,7,8,11,12,15,16, select Even Fifo */
    /* For receivers 1,2,5,6,9,10,13,14, select Odd Fifo   */
    if (rcvr & 2)
        params->fifo_select = P65XX_EVEN_FIFO;
    else
        params->fifo_select = P65XX_ODD_FIFO;

    params->tag_enable = P65XX_TAG_OFF;
    params->flip_spectrum = P65XX_FLIP_OFF;
    params->offset_spectrum = P65XX_OFFSET_ON;
    params->wait_ram_full = P65XX_WAIT_RAM_FULL_ON;
    params->fifo_mode = P6514_DUAL_RCVR;            /* Two Rcvrs per fifo */
}

/*****************************************************************************
 Function: set65XXInputParams

 Description: This routine initializes the Input Channel Parameter Table for
              a specified input channel.
 Inputs:
        params - Pointer to a P65XX_INPUT_PARAMS table entry.
        input - Input Channel being setup
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void set65XXInputParams(P65XX_INPUT_PARAMS *params, int input)
{
    params->threshold_level = P65XX_THRSHLD_DETECT_OFF;
    params->threshold_cross_cnt = P65XX_CROSS_CNT_2;
#if P6514
    params->input_reset_mode = P6514_SW_RESET_IN_SRC | P6514_SW_RESET_OFF |
                               P6514_RESET_OUT_DIS;
#endif
#if P6516
    params->input_reset_mode = P6516_SW_RESET_IN_SRC | P6516_SW_RESET_OFF |
                               P6516_RESET_OUT_DIS;
    params->input_crossover = P6516_CROSSOVER_A_TO_C_ON |
                              P6516_CROSSOVER_B_TO_D_ON;
#endif
}
```

```
/*****************************************************************************
 Function: main65XX

 Description: This program initializes the specified receivers and
              acquires data.

*****************************************************************************/
main65XX()
{
    register int           i;
    register int           j;
    unsigned long          cmdParams[2];
    unsigned long          status;
    int                    *samples;
    P65XX_REG_ADDR         p65xxReg;
    P65XX_RCVR_PARAMS      p65xxRcvrParams[P65XX_NUM_RCVRS];
    P65XX_INPUT_PARAMS     p65xxInputParams;
    unsigned int           handle[P65XX_NUM_RCVRS] = {0, 0, 0, 0, 0, 0, 0, 0,
                                                      0, 0, 0, 0, 0, 0, 0, 0};
    unsigned long          returnData[2];
    unsigned long          numRData;
    unsigned long          a_rcvr;
    unsigned long          rcvr;
    unsigned long          num_samples;
    char                   filename[20];
    FILE                   *outfile;
    volatile long          *fifo;
    unsigned long vmeBase;              /* VMEbus A24 & A32 memory pointer */
    unsigned long a16Base;              /* VMEbus A16 memory pointer       */

    /* Map Vme Address to Local Address */
    sysBusToLocalAdrs(P65XX_VME_ACCESS_MODE, (char *)P65XX_VME_BASE_ADDR,
                      (char **)&vmeBase);
    printf("VMEBase = %lx\n", vmeBase);

    /* Map A16 Address to Local Address */
    sysBusToLocalAdrs(VME_AM_SUP_SHORT_IO, (char *)P65XX_A16_BASE_ADDR,
                      (char **)&a16Base);
    printf("A16Base = %lx\n", a16Base);

    /* Set 65XX Board's Base Address */
    VxWSet65XXBaseAddress(a16Base, P65XX_VME_BASE_ADDR,
                          P65XX_VME_ACCESS_MODE);

    /* Initialize pointers to 65XX Registers */
    InitRegAddr(vmeBase, &p65xxReg);

#if INTERRUPT
    /* Set 65XX Board's IRQ Level and Vector */
    VxWSet65XXIRQLevelVector(a16Base, P65XX_INTR_LEVEL, P65XX_INTR_VECTOR);

    /* Set 65XX Board's IRQ Level and Vector */
    if (intConnect(INUM_TO_IVEC(P65XX_INTR_VECTOR), fifoFullIntr) != OK)
    {
         printf("STATUS: INTERRUPT ERROR cant connect int");
         exit(1);
    }

    /* Set Fifo Full Interrupt flag to false */
    FifoFullIntr = 0;

    /* Turn on the Interrupt */
    sysIntEnable(P65XX_INTR_LEVEL);
```

```
#endif

    /* Close all prior open connections */
    if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, 0,
                        P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_ALL_CONNECTIONS, 0),
                        0, NULL, 0, &numRData, returnData)) !=
                                                    P65XX_COMMAND_COMPLETE)
    {
        printf("Error Closing All Connections - Status = %lx\n", status);
        exit(1);
    }

    /* Reset all 65XX Registers */
    if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, 63,
                                P65XX_BUILD_CMD_WAIT(P65XX_RESET_REGS, 0),
                                0, NULL, 0, &numRData, returnData)) !=
                                            P65XX_COMMAND_COMPLETE)
    {
        printf("Error Resetting Registers - Status = %lx\n", status);
        exit(1);
    }

    /* Initialize Input Channels */
    for (i = 0; i < P65XX_NUM_INPUTS; ++i)
    {
         /* Initialize Input Channel Parameter Data structure */
         set65XXInputParams(&p65xxInputParams, i);

         /* Initialize the Input Channel */
         if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, 10 + i,
                           P65XX_BUILD_CMD_WAIT(P65XX_INIT_INPUT_CHANNELS, 0),
                           sizeof(P65XX_INPUT_PARAMS) / 4,
                           (void *)&p65xxInputParams, 1 << i,
                           &numRData, returnData)) != P65XX_COMMAND_COMPLETE)
         {
             printf("Error Initializing Input %c - Status = %lx\n", 'A' + i,
                    status);
             exit(1);
         }
    }
    /* Open a connection and initialize each receiver  */
    for (i = 0; i < P65XX_NUM_RCVRS; ++i)
    {
        if (RCVR_MASK & (1 << i))
        {
            /* Initialize Parameter Data structure */
            set65XXRcvrParams(&p65xxRcvrParams[i], i + 1);

            /* Establish Connection for this receiver */
            if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, i,
                              P65XX_BUILD_CMD_WAIT(P65XX_OPEN_CONNECTION, 0),
                              0, NULL, 1 << i, &numRData, returnData)) !=
                                                    P65XX_COMMAND_COMPLETE)
            {
                printf("Error Opening Connection Rcvr %d - Status = %lx\n",
                                 i + 1, status);
                exit(1);
            }

            /* Save the handle corresponding to this connection */
            handle[i] = *returnData;

            /* Initialize the Receiver */
            if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, i,
```

```
                                P65XX_BUILD_CMD_INTEGER(P65XX_INIT_RCVRS,\
                                            handle[i], P65XX_WAIT_COMPLETE),
                                sizeof(P65XX_RCVR_PARAMS) / 4,
                                (void *)&p65xxRcvrParams[i], 1 << i,
                                &numRData, returnData)) !=
                                            P65XX_COMMAND_COMPLETE)
            {
                printf("Error Initializing Rcvr %d - Status = %lx\n",
                                 i + 1, status);
                exit(1);
            }

            /* Save an index into one of the valid receivers */
            /* data structures                               */
            a_rcvr = i;
        }
    }

    /* Reset all the Receiver Fifos */
    ResetFifo(&p65xxRcvrParams[a_rcvr], &p65xxReg, RCVR_MASK);

#if INTERRUPT
    /* Enable Fifo Full Interrupt on one of the receivers */
    cmdParams[0] = P65XX_GEN_VME_INTR | P65XX_ENABLE_INTERRUPT |
                   P65XX_MASK_ON_OCCURRENCE | P65XX_EDGE_TRIG_INTR;
    cmdParams[1] = P65XX_FIFO_FULL_1;

    if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, 23,
                              P65XX_BUILD_CMD_WAIT(P65XX_MODIFY_INTR_MASK, 0),
                              2, cmdParams, 0, &numRData, returnData)) !=
                                                  P65XX_COMMAND_COMPLETE)
    {
        printf("Error Enabling Interrupt - Status = %lx\n", status);
        exit(1);
    }
#endif

    /* Put the Receiver in a RESET Mode.  This turns off the data stream */
    /* going into the fifo.  This only has to occur once after the       */
    /* receivers have been initialized.                                  */
#if P6514
    p65xxInputParams.input_reset_mode = P6514_SW_RESET_ON |
                                        P6514_SW_RESET_IN_SRC;
#endif
#if P6516
    p65xxInputParams.input_reset_mode = P6516_SW_RESET_ON |
                                        P6516_SW_RESET_IN_SRC;
#endif
    SetInputResetMode(&p65xxInputParams, &p65xxReg, DATA_SOURCE);

    /* Release all the Receiver Fifos */
    ReleaseFifo(&p65xxRcvrParams[a_rcvr], &p65xxReg, RCVR_MASK);

#if P6514
    p65xxInputParams.input_reset_mode = P6514_SW_RESET_OFF |
                                        P6514_SW_RESET_IN_SRC;
#endif
#if P6516
    p65xxInputParams.input_reset_mode = P6516_SW_RESET_OFF |
                                        P6516_SW_RESET_IN_SRC;
#endif
    SetInputResetMode(&p65xxInputParams, &p65xxReg, DATA_SOURCE);

    if (p65xxRcvrParams[a_rcvr].data_mode == P65XX_COMPLEX)
```

```
        num_samples = NUM_CMPLX_SAMPLES * 2;
    else
        num_samples = NUM_REAL_SAMPLES;

    for (i = 0; i < P65XX_NUM_FIFOS; ++i)
    {
        if (RCVR_MASK & (3 << (i * 2)))
        {
#if POLLING
            /* Wait for Fifo to Fill Up */
            while (*p65xxReg.Stat_Reg_1 & (0x1 << i));
#else
            /* Wait for Fifo to Fill Up */
            while (!FifoFullIntr);
#endif
            /* Store data for each receiver in the pair */
            for (rcvr = 2 * i; rcvr <= (2 * i) + 1; ++rcvr)
            {
                /* Open an ASCII file for storing data */
                sprintf(filename, "rcvr%d.dat", rcvr + 1);
                outfile = fopen(filename, "w+");
                if (outfile == NULL)
                {
                    printf("Error opening file %s\n", filename);
                    exit(1);
                }

                printf("Saving Rcvr %d Data to rcvr%d.dat...", rcvr + 1,
                                                        rcvr + 1);

                /* If rcvr is a Odd number(corresponds to rcvr 2, 4, 6, 8, */
                /* 10, 12, or 14), then it's data has been already read.   */
                if (rcvr & 0x1)
                {
                    if (p65xxRcvrParams[rcvr].data_mode == P65XX_COMPLEX)
                    {
                        for (j = 2; j < num_samples * 2; j+=4)
                        {
                            fprintf(outfile, "%d\n%d\n",
                                            inSamples[j + 1] >> 16,
                                            inSamples[j] >> 16);
                        }
                    }
                    else
                    {
                        for (j = 1; j < num_samples * 2; j+=2)
                        {
                            fprintf(outfile, "%d\n", inSamples[j] >> 16);
                        }
                    }
                }
                else  /* First channel being read from fifo */
                {
                    /* Write Fifo Data to File */
                    fifo = p65xxReg.Fifo + 0x2000 * i;
                    for (j = 0; j < num_samples * 2; ++j)
                        inSamples[j] = *fifo;
                    if (p65xxRcvrParams[rcvr].data_mode == P65XX_COMPLEX)
                    {
                        for (j = 0; j < num_samples * 2; j+=4)
                        {
                            fprintf(outfile, "%d\n%d\n",
                                            inSamples[j + 1] >> 16,
                                            inSamples[j] >> 16);
```

```
                    }
                }
                else   /* Real Data */
                {
                    for (j = 0; j < num_samples * 2; j+=2)
                    {
                        fprintf(outfile, "%d\n", inSamples[j] >> 16);
                    }
                }
            }
            /* Close File */
            fclose(outfile);
            printf("Done\n");
        }
    }
}

    /* Close All Open Connections */
    if ((status = VxWSend65XXCommandWait(a16Base, vmeBase, 0,
                        P65XX_BUILD_CMD_WAIT(P65XX_CLOSE_ALL_CONNECTIONS, 0),
                        0, NULL, 0, &numRData, returnData)) !=
                                                    P65XX_COMMAND_COMPLETE)
    {
        printf("Error Closing All Connections - Status = %lx\n", status);
        exit(1);
    }

#if INTERRUPT
    /* Turn off the Interrupt */
    sysIntDisable(P65XX_INTR_LEVEL);
#endif

    printf("Program Complete\n");
}

/*************************************************************************
 Function: fifoFullIntr

 Description: This routine servers as an interrupt handler for Fifo Full
              interrupts generated by the 65XX Board
 Inputs:
 Outputs:
 Returns:
 Notes:
*************************************************************************/
void fifoFullIntr()
{
    /* Set Fifo full Interrupt Flag */
    FifoFullIntr = 1;
}
```

```
/*************************************************************************
*
*   File :  cmdvxw.c
*
*   Description:  This file contains routines for interfacing a Single Board
*                 Computer running vxWorks to a 65XX Digital Receiver Board.
*
*************************************************************************/
#include "vxWorks.h"
#include "sysLib.h"
#include "msgQLib.h"
#include "iv.h"
#include "vme.h"
#include "taskLib.h"
#include "usrLib.h"
#include "65xxcmd.h"
#include "cmdvxw.h"

/*************************************************************************
 Function: VxWSendCommandWait

 Description: This routine generates a command to the 65XX Board via the
              VMEBus Interface and waits for a response.
 Inputs:
        a16Base - A16 Base Address of 65XX Board
        vmeBase - VME Base Address of 65XX Board
        cmdTblIndex - Index into 65XX VME Command Table(0 - 63)
        command - Command being sent
        num_params - The number of parameters which must follow command
        cmdParams - Pointer to the command parameters
        input_rcvr_mask - Input Channel or Receiver mask
        numRdata - Pointer to store the number of returned data
        returnData - Buffer pointer to store the return data
 Outputs:
        numRdata - Number of returnned data
        returnData - Returned Data
 Returns:
        Status - Status of the Command
 Notes:
*************************************************************************/
unsigned long VxWSend65XXCommandWait(unsigned long a16Base,
                                     unsigned long vmeBase,
                                     unsigned long cmdTblIndex,
                                     unsigned int command,
                                     unsigned int num_params,
                                     void *cmdParams,
                                     unsigned int input_rcvr_mask,
                                     unsigned long *numRData,
                                     unsigned long *returnData)
{
    unsigned long   i;
    unsigned int   status;
    unsigned long  a16RegBase;
    register volatile unsigned long *vme_command;

    /* Initialize pointer to Command location in 65XX DPR */
    vme_command = (unsigned long *)vmeBase + P65XX_VME_COMMAND_ADDR +
                                    cmdTblIndex * P65XX_CMD_TBL_ENTRY_SIZE;

    /* Write Input Channel/Receiver Mask */
    *(vme_command + 1) = input_rcvr_mask;

    /* Write the command parameters */
    for (i = 0; i < num_params; ++i)
```

```
        *(vme_command + 2 + i) = *((unsigned long *)cmdParams + i);

    /* Write Command */
    *vme_command  = command;

    /* Interrupt the C40 on the 6508 Board */
    *((unsigned long *)a16Base + 2) = 0x1;

    /* Wait for Command to Complete */
    while (*vme_command != 0)
    {
    }

    /* Read Status and return data */
    status = *(vme_command + 1);
    if ((numRData != NULL) && (returnData != NULL))
    {
        *numRData = status >> 16;
        for (i = 0; i < *numRData; ++i)
            *returnData++ = *(vme_command + 2 + i);
    }

    return(status & 0xffff);
}
/*****************************************************************************
 Function: VxWSend65XXCommand

 Description: This routine generates a command to the 65XX Board via the
              VMEBus Interface.
 Inputs:
        a16Base - A16 Base Address of 65XX Board
        vmeBase - VME Base Address of 65XX Board
        cmdTblIndex - Index into 65XX VME Command Table(0 - 63)
        command   - Command being sent
        num_params - The number of parameters which must follow command
        cmdParams - Pointer to the command parameters
        input_rcvr_mask - Input Channel or Receiver mask
 Outputs:
 Returns:
        Command Complete status
 Notes: Returns immediately to caller without waiting command to complete
*****************************************************************************/
void VxWSend65XXCommand(unsigned long a16Base, unsigned long vmeBase,
                        unsigned long cmdTblIndex,
                        unsigned int command, unsigned int num_params,
                        void *cmdParams, unsigned int input_rcvr_mask)
{
    unsigned int   i;
    unsigned int   status;
    register volatile unsigned long  *vme_command;

    /* Initialize pointer to Command location in 6508 DPR */
    vme_command = (unsigned long *)vmeBase + P65XX_VME_COMMAND_ADDR +
                                    cmdTblIndex * P65XX_CMD_TBL_ENTRY_SIZE;

    /* Write Input Channel/Receiver Mask */
    *(vme_command + 1) = input_rcvr_mask;

    /* Write the command parameters */
    for (i = 0; i < num_params; ++i)
        *(vme_command + 2 + i) = *((unsigned long *)cmdParams + i);

    /* Write Command */
    *vme_command  = command;
```

```
    /* Interrupt the C40 on the 65XX Board */
    *((unsigned long *)a16Base + 2) = 0x1;
}

/*****************************************************************************
 Function: VxWSet65XXBaseAddress

 Description: This routine sets the 65XX's A32/A24 Base Address.
 Inputs:
        a16Base - A16 Base Address of 65XX Board
        vmeBase - VME Base Address of 65XX Board
        accessMode - Address Modifier
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void VxWSet65XXBaseAddress(unsigned long a16Base, unsigned long vmeBase,
                           unsigned int accessMode)
{
    unsigned long *a16 = (unsigned long *)a16Base;

    if ((accessMode & 30) == 0x30)                      /* If A24   */
        *(a16 + 1) = ((((vmeBase >> 19) & 0x1e) | 0x20) << 16);
    else                                                 /* Else A32 */
        *(a16 + 1) = (((vmeBase >> 16) & 0xffc0) << 16);
}
/*****************************************************************************
 Function: VxWSet65XXIRQLevelVector

 Description: This routine sets the 65XX's Interrupt Level and Vector
 Inputs:
        a16Base - A16 Base Address of 65XX Board
        irqLevel  - Interrupt Level
        irqVector  - Interrupt Vector
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void VxWSet65XXIRQLevelVector(unsigned long a16Base, unsigned long irqLevel,
                              unsigned long irqVector)
{
    unsigned long *a16 = (unsigned long *)a16Base;

    *a16 = (*a16 & ~0x03ff0000) | (irqLevel << 24) | (irqVector << 16);
}
```

```
/*****************************************************************************
*
*   File : erase65.c
*
*   Description: This file contains a routine which commands the 65XX board
*                to erase the firmware.  Any C40 Processor with an external
*                communication port can disable the firmware.  This routine
*                can also command the 65XX board to disable the firmware using
*                the same command over the VMEBus.  This example uses the 4285
*                Baseboard, but can easily be modified for any platform.
*
*****************************************************************************/

#include <stdlib.h>
#include "65xxcmd.h"
#include "4285.h"
#include "vme.h"

/* Specify the interface which will be used to send the command */
#define USE_VME_INTERFACE        1
#define USE_COMM_PORT_INTERFACE  0

/* When using a C40 Communication Port to command the 65XX board to disable
   the firmware, specify the Local C40 Comm Port which is connected to one
   of the 65XX Boards Input Comm Ports.  This Comm Port must be a defaulted
   output Comm Port.
*/
#define CMD_COMM_PORT          2

/* When using the VMEBus Interface to command the 65XX board to disable
   the firmware, specify the 65XX's Boards Base Addresses.
*/
/* Base Address of the 65XX Board */
#define P65XX_A16_BASE_ADDR      0xFF00
#define P65XX_VME_BASE_ADDR      0x00800000
#define P65XX_VME_ACCESS_MODE    VME_AM_A24_NPR_DAT

main()
{
#if USE_COMM_PORT_INTERFACE
    volatile unsigned int *com;
    unsigned int          temp;

    /* Clear out Command Comm Port */
    com = (volatile unsigned int *)(0x100040 + 0x10 * CMD_COMM_PORT);
    while ((*com & 0x1e00) != 0)
        temp = *(com + 1);

    /* Send Disable Firmware Command */
    P65XXSendCommand(CMD_COMM_PORT,
                     P65XX_BUILD_CMD(P65XX_DISABLE_FIRMWARE, 0), 0, NULL, 0);
#endif
#if USE_VME_INTERFACE
    unsigned long a16Base;
    unsigned long vmeBase;

    /* Setup the 4285 for A16 accesses */
    a16Base = P4285SetupVMETransfer(VME_AM_A16_NPR_ACC, P65XX_A16_BASE_ADDR,
                                    32);

    /* Set the 65XX's A32/A24 Base Address */
    P4285Set65XXBaseAddress(a16Base, P65XX_VME_BASE_ADDR,
                            P65XX_VME_ACCESS_MODE);
```

```
    /* Setup the 4285 for A32 Accesses */
    vmeBase = P4285SetupVMETransfer(P65XX_VME_ACCESS_MODE,
                                    P65XX_VME_BASE_ADDR, 32);

    /* Send Disable Firmware Command */
    P4285Send65XXCommand(a16Base, vmeBase,
                         P65XX_BUILD_CMD(P65XX_DISABLE_FIRMWARE, 0), 0,
                         NULL, 0);
#endif
}
```

```
/*****************************************************************************
*
*   File : boot65XX.cmd
*
*   Description: This file is an input to hex30.  It is used in the
*                conversion form COFF File Format to a Bootable Image
*                file format
*
*****************************************************************************/

firm65XX.xXX            /* Input file         */
-o firm65XX.hex         /* HEX output file    */
-i                      /* intel format       */
-memwidth 8             /* 8-bit memory       */
-boot                   /* Convert all .sect  */
-bootorg     0h         /* Boot source addr   */
-cg   3DEF4000h         /* Global mem conf.   */
-cl   3DEF4230h         /* Local mem conf.    */
-e    40000c00h         /* PC after load addr */
-iack 40000000h         /* IACK dummy read    */
-ivtp 002FFC00h         /* IV pointer         */
-tvtp 002FFC00h         /* TV pointer         */
```

```
/*****************************************************************************
*
*   File : am29f040.h
*
*   Description: This include file contains function prototypes for
*                programming Flash EEPROM(AM29F0400).
*
*****************************************************************************/

#ifndef AM29F040_H
#define AM29F040_H

void AM29F040EraseSector(volatile unsigned long *flashBaseP,
                         unsigned long sector, unsigned long size);
void AM29F040WriteEnable(volatile unsigned long *flashBaseP);
int AM29F040FlashSet(volatile unsigned long *flashBaseP,
                     unsigned long offset, unsigned long data);
unsigned long AM29F040FlashGet(volatile unsigned long *flashBaseP,
                               unsigned long offset);
void AM29F040FlashDelay(unsigned long count);

#endif
```

```
/*****************************************************************************
*
*   File :  am29f040.c
*
*   Description: This source file contains routines for erasing, writing,
*                and reading Flash EEPROM (AM29F040).
*
*****************************************************************************/

#include "am29f040.h"

/*****************************************************************************
 Function: AM29F040EraseSector

 Description: This function erases one or more flash EEPROM sectors.
 Inputs:
         flashBase   -  Flash EEPROM Base Address
         sector      -  Sector being erased(0 - 7)
         numSectors  -  Number of Sectors to be erased
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void AM29F040EraseSector(volatile unsigned long *flashBase,
                         unsigned long sector,
                         unsigned long numSectors)
{
    unsigned long i;

    flashBase[0x5555] = 0xaa;
    flashBase[0x2aaa] = 0x55;
    flashBase[0x5555] = 0x80;
    flashBase[0x5555] = 0xaa;
    flashBase[0x2aaa] = 0x55;
    flashBase[sector] = 0x30;
    for (i = 1; i < numSectors; ++i)
        flashBase[sector + 0x10000 * i] = 0x30;

    /* Wait for sector erase to complete */
    for (i = 0; i < numSectors; ++i)
        while (!(flashBase[sector + 0x10000 * i]  & 0x80));
}

/*****************************************************************************
 Function: AM29F040WriteEnable

 Description: This function enable the flash EEPROM for writing.
 Inputs:
         flashBase   -  Flash EEPROM Base Address
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void AM29F040WriteEnable(volatile unsigned long *flashBase)
{
    flashBase[0x5555] = 0xaa;
    flashBase[0x2aaa] = 0x55;
    flashBase[0x5555] = 0xa0;
}

/*****************************************************************************
 Function: AM29F040FlashSet
```

```
 Description: This function writes a 32 bit word of data into the flash
              EEPROM.
 Inputs:
         flashBase   -  Flash EEPROM Base Address
         offset      -  Offset into flash
         data        -  32 Bit word to be written
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
int AM29F040FlashSet(volatile unsigned long *flashBase,
                     unsigned long offset,
                     unsigned long data)
{
    unsigned char rom,i;
    unsigned char byte;
    volatile unsigned long *addr;

    addr = flashBase + offset;

    for(i=0;i<4;i++)
    {
        byte = (unsigned char)((data >> (i*8)));
        AM29F040WriteEnable(flashBase);
        AM29F040FlashDelay(0x20);
        addr[i] = byte;
        AM29F040FlashDelay(0x20);
        rom = addr[i];
        while(((rom = addr[i]) & 0x80) != (byte & 0x80))
        {
            if(rom & 0x20)
            {
                AM29F040FlashDelay(0x10);
                if ((addr[i] & 0x80) != (byte & 0x80))
                    return(-1);
            }
        }
    }
    return(0);
}

/*****************************************************************************
 Function: AM29F040FlashGet

 Description: This function reads a 32 bit word of data from flash
              EEPROM.
 Inputs:
         flashBase   -  Flash EEPROM Base Address
         offset      -  Offset into flash
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
unsigned long AM29F040FlashGet(volatile unsigned long *flashBase,
                               unsigned long offset)
{
    unsigned char i;
    unsigned char byte;
    volatile unsigned long *addr;
    unsigned long data = 0;

    addr = flashBase + offset;

    for(i = 0;i < 4; i++)
```

```
    {
        byte = (*addr++ & 0xff);
        data |= byte << (8 * i);
    }
    return(data);
}

/*****************************************************************************
 Function: AM29F040FlashDelay

 Description: This function is a simple delay loop.
 Inputs:
         count   -  Number of counts to delay
 Outputs:
 Returns:
 Notes:
*****************************************************************************/
void AM29F040FlashDelay(unsigned long count)
{
    volatile unsigned long i;

    for (i = 0; i < count; ++i);
}
```

```
/*****************************************************************************
*
*   File : loadhx65.c
*
*   Description: This program uses SwiftNet's SNIO to load the Flash EEPROM
*                on the 65XX Boards
*
*****************************************************************************/
#include <stdlib.h>
#include <snstdio.h>
#include "am29f040.h"
main()
{
    char filename[128];
    char string[80];
    unsigned long *addrP;
    FILE *fileP;
    int done = 0;
    unsigned long i;
    unsigned long length;
    long word_count = 0;
    unsigned long data;
    unsigned long data1;
    unsigned char ch;
    unsigned long offset = 0;

    /* Initialize SNIO interface */
    if (snstdio_init(NULL) != OK)
        exit(1);
    atexit((void (*)())snstdio_shutdown);

    printf("65XX Flash Loader\n");

    printf("Specify filename(.hex): ");
    fflush(stdout);

    scanf("%s", filename);
    printf("\n");

    printf("Opening %s\n", filename);

    /* Open file */
    if ((fileP = fopen(filename, "r")) == NULL)
    {
         printf("Cannot open file: %s\n", filename);
         exit(-1);
    }

    /* Erase any previous programs */
    printf("Erasing Flash...");
    AM29F040EraseSector((unsigned long *) 0x80080000, 0L, 8L);
    printf("Done\n");

    /* Loading Flash Memory */

    /* Load Boot Flag indicating program present */
    if (AM29F040FlashSet((unsigned long *)0x80080000, 0L, 0xa55a5aa5L) != 0)
        printf("Error Setting Flash at %lx\n", 0x80080000);

    /* Specify starting address of bootcode */
    if (AM29F040FlashSet((unsigned long *)0x80080000, 4L, 0x80090000L) != 0)
        printf("Error Setting Flash at %lx\n", 0x80080004);

    /* Load program into flash */
```

```
    printf("\nLoading Flash...");
    while (done == 0)
    {
        /* See if this is a hex file */
        if ( feof( fileP ) ) {
            printf("\nEOF1 was detected !");
            exit(0);
        }
        ch = fgetc( fileP );
        if (ch != ':') {
            printf("\nNot a HEX file !");
            exit(0);
        }

        /* Get data length */

        string[0] = fgetc( fileP );
        string[1] = fgetc( fileP );
        string[2] = 0;
      sscanf(string,"%x",&length);
      length = length/4;
      if (length == 0) {
            done=1;
            break;
        }

        /* Skip over the rest of header */
        for (i=0; i<6; i++)
            ch = fgetc( fileP );

        /* Get data and store it into buffer */
        for (i=0; i<length; i++)
        {
            string[6] = fgetc( fileP );
            string[7] = fgetc( fileP );
            string[4] = fgetc( fileP );
            string[5] = fgetc( fileP );
            string[2] = fgetc( fileP );
            string[3] = fgetc( fileP );
            string[0] = fgetc( fileP );
            string[1] = fgetc( fileP );
            string[8] = 0;
            sscanf(string,"%lx",&data);
            if (AM29F040FlashSet((unsigned long *)0x80090000,
                                 offset + i * 4, data) != 0)
                printf("Error Setting Flash at %lx\n",
                                              0x80090000 + offset + i * 4);
            ++word_count;
        }

        offset = offset + length * 4;

        /* Skip over cksum and CR/LF */
        fgets(string,80,fileP);
    }
    printf("Done\n");
    printf("\nFlash Loading Complete\n");

    /* Close File */
    fclose(fileP);

    /* Exit Program */
    exit(0);
}
```

```
/*****************************************************************************
*
*   File : ldflsh65.c
*
*   Description: This program loads a program into flash EEPROM on the 65XX
*                Board.  The file being loaded must be in Intel HEX format.
*
*****************************************************************************/

#include "vxWorks.h"
#include "stdio.h"
#include "stdlib.h"
#include "sysLib.h"
#include "msgQLib.h"
#include "iv.h"
#include "vme.h"
#include "taskLib.h"
#include "usrLib.h"
#include "am29f040.h"
#include "cmdvxw.h"

/* Base Address of the 65XX Board */
#define P65XX_A16_BASE_ADDR      0xFF00
#define P65XX_VME_BASE_ADDR      0x40000000
#define P65XX_VME_ACCESS_MODE    VME_AM_EXT_USR_DATA  /* Flash EEPROM is */
                                                      /* only accessible */
                                                      /* from VME Space  */
ldflsh65()
{
    char filename[128];
    char string[80];
    unsigned long *addrP;
    FILE *fileP;
    int done = 0;
    unsigned long i;
    unsigned long length;
    long word_count = 0;
    unsigned long data;
    unsigned long data1;
    unsigned char ch;
    unsigned long offset = 0;
    unsigned long temp;
    unsigned long a16Base;
    unsigned long vmeBase;

    /* Map Vme Address to Local Address */
    sysBusToLocalAdrs(P65XX_VME_ACCESS_MODE, (char *)P65XX_VME_BASE_ADDR,
                      (char **)&vmeBase);
    printf("VMEBase = %lx\n", vmeBase);

    /* Map A16 Address to Local Address */
    sysBusToLocalAdrs(VME_AM_SUP_SHORT_IO, (char *)P65XX_A16_BASE_ADDR,
                      (char **)&a16Base);
    printf("A16Base = %lx\n", a16Base);

    /* Set 65XX Board's Base Address */
    VxWSet65XXBaseAddress(a16Base, P65XX_VME_BASE_ADDR,
                          P65XX_VME_ACCESS_MODE);

    printf("65XX Flash Loader\n");

    printf("Specify filename(.hex): ");
    fflush(stdout);
```

```
scanf("%s", filename);
printf("\n");

printf("Opening %s\n", filename);

/* Open file */
if ((fileP = fopen(filename, "r")) == NULL)
{
     printf("Cannot open file: %s\n", filename);
     exit(-1);
}

/* Erase any previous programs */

printf("Erasing Flash...");
AM29F040EraseSector((unsigned long *) (vmeBase + 0x200000L), 0L, 2L);
printf("Done\n");

/* Loading Flash Memory */

/* Load Boot Flag indicating program present */
if (AM29F040FlashSet((unsigned long *)(vmeBase + 0x200000), 0L,
                                                    0xa55a5aa5L) != 0)
    printf("Error Setting Flash at %lx\n", vmeBase + 0x200000);

/* Specify starting address of bootcode */
if (AM29F040FlashSet((unsigned long *)(vmeBase + 0x200000), 4L,
                                                    0x80090000L) != 0)
    printf("Error Setting Flash at %lx\n", vmeBase + 0x200004);

/* Load program into flash */
printf("\nLoading Flash...");
while (done == 0)
{
    /* See if this is a hex file */
    if ( feof( fileP ) ) {
        printf("\nEOF1 was detected !");
        exit(0);
    }
    ch = fgetc( fileP );
    if (ch != ':') {
        printf("\nNot a HEX file !");
        exit(0);
    }

    /* Get data length */

    string[0] = fgetc( fileP );
    string[1] = fgetc( fileP );
    string[2] = 0;
  sscanf(string,"%x",&length);
  length = length/4;
  if (length == 0) {
        done=1;
        break;
    }

    /* Skip over the rest of header */
    for (i=0; i<6; i++)
        ch = fgetc( fileP );

    /* Get data and store it into buffer */
    for (i=0; i<length; i++)
```

```
        {
            string[6] = fgetc( fileP );
            string[7] = fgetc( fileP );
            string[4] = fgetc( fileP );
            string[5] = fgetc( fileP );
            string[2] = fgetc( fileP );
            string[3] = fgetc( fileP );
            string[0] = fgetc( fileP );
            string[1] = fgetc( fileP );
            string[8] = 0;
            sscanf(string,"%lx",&data);
            if (AM29F040FlashSet((unsigned long *)(vmeBase + 0x240000),
                          offset + i * 4, data) != 0)
                printf("Error Setting Flash at %lx\n",
                                        (vmeBase + 0x240000) + offset + i * 4);
            ++word_count;
        }

        offset = offset + length * 4;

        /* Skip over cksum and CR/LF */
        fgets(string,80,fileP);
    }
    printf("Done\n");
    printf("\nFlash Loading Complete\n");

    /* Close File */
    fclose(fileP);

    /* Reset the 65XX board */
    VxWReset65XXBoard(a16Base);

    /* Allow Board to reset and load firmware */
    for (i = 0; i < 0x10000; ++i);

    /* Read the first location in Global SRAM */
    temp = *(unsigned long *)vmeBase;

    printf("%lx Firmware loaded - Revision %x\n", temp >> 16,
                                                  temp & 0xffff);
}
```

Appendix B: Harris HSP50016 Digital Down Converter

B.1 Introduction

Included in its entirety, beginning on the page after next, is the manual for the Harris HSP50016 digital Down Converter. Sixteen of these devices are used on the Model 6516.

This page is intentionally blank

Digital Down Converter

The Digital Down Converter (DDC) is a single chip synthesizer, quadrature mixer and lowpass filter. Its input data is a sampled data stream of up to 16 bits in width and up to a 75 MSPS data rate. The DDC performs down conversion, narrowband low pass filtering and decimation to produce a baseband signal.

The internal synthesizer can produce a variety of signal formats. They are: CW, frequency hopped, linear FM up chirp, and linear FM down chirp. The complex result of the modulation process is lowpass filtered and decimated with identical real filters in the in-phase (I) and quadrature (Q) processing chains.

Lowpass filtering is accomplished via a High Decimation Filter (HDF) followed by a fixed Finite Impulse Response (FIR) filter. The combined response of the two stage filter results in a -3dB to -102dB shape factor of better than 1.5. The stopband attenuation is greater than 106dB. The composite passband ripple is less than 0.04dB. The synthesizer and mixer can be bypassed so that the chip operates as a single narrow band low pass filter.

The chip receives forty bit serial commands as a control input. This interface is compatible with the serial I/O port available on most microprocessors.

The output data can be configured in fixed point or single precision floating point. The fixed point formats are 16, 24, 32, or 38-bit, two's complement, signed magnitude, or offset binary.

The circuit provides an IEEE 1149.1 Test Access Port.

Features

- 75 MSPS Input Data Rate
- 16-Bit Data Input; Offset Binary or 2's Complement Format
- Spurious Free Dynamic Range Through Modulator >102dB
- Frequency Selectivity: <0.006Hz
- Identical Lowpass Filters for I and Q
- Passband Ripple: <0.04dB
- Stopband Attenuation: >104dB
- Filter -3dB to -102dB Shape Factor: <1.5
- Decimation Factors from 32 to 131,072
- IEEE 1149.1 Test Access Port
- HSP50016-EV Evaluation Board Available

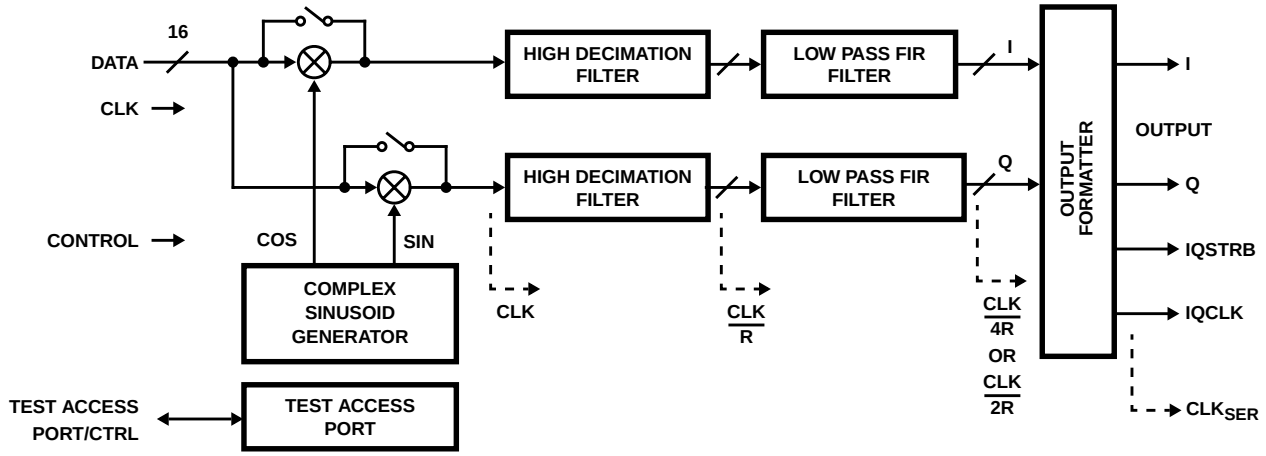
Applications

- Cellular Base Stations
- Smart Antennas
- Channelized Receivers
- Spectrum Analysis
- Related Products: HI5703, HI5746, HI5766 A/Ds

Ordering Information

PART NUMBER	TEMP. RANGE (^{o}C)	PACKAGE	PKG. NO.
HSP50016JC-52	0 to 70	44 Ld PLCC	N44.65
HSP50016JC-75	0 to 70	44 Ld PLCC	N44.65
HSP50016GC-52	0 to 70	48 Ld CPGA	G48.A

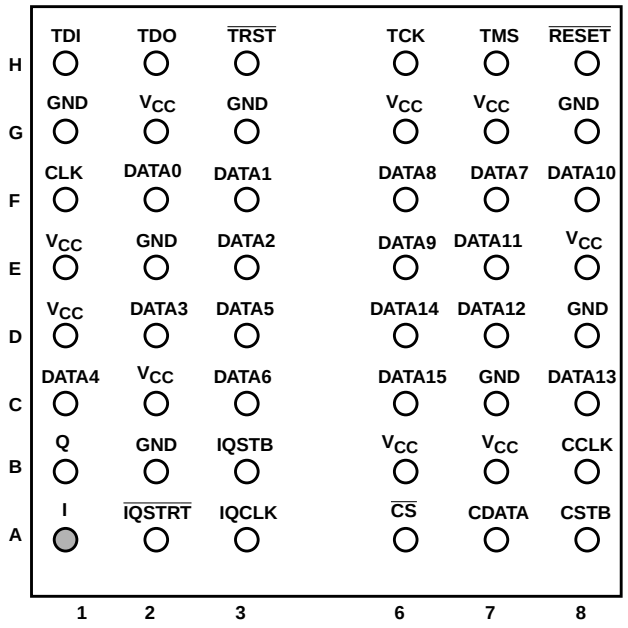
Block Diagram



CAUTION: These devices are sensitive to electrostatic discharge; follow proper IC Handling Procedures.
1-800-4-HARRIS or 407-727-9207 | Copyright © Harris Corporation 1999

Pinouts

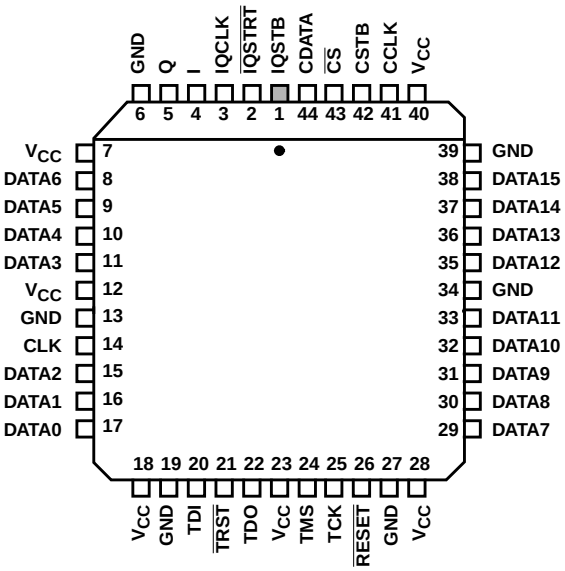
48 PIN CPGA
BOTTOM VIEW



48 PIN CPGA
TOP VIEW

	1	2	3		6	7	8
A	I	IQSTRT	IQCLK		CS	CDATA	CSTB
B	Q	GND	IQSTB		V_{CC}	V_{CC}	CCLK
C	DATA4	V_{CC}	DATA6		DATA15	GND	DATA13
D	V_{CC}	DATA3	DATA5		DATA14	DATA12	GND
E	V_{CC}	GND	DATA2		DATA9	DATA11	V_{CC}
F	CLK	DATA0	DATA1		DATA8	DATA7	DATA10
G	GND	V_{CC}	GND		V_{CC}	V_{CC}	GND
H	TDI	TDO	TRST		TCK	TMS	RESET

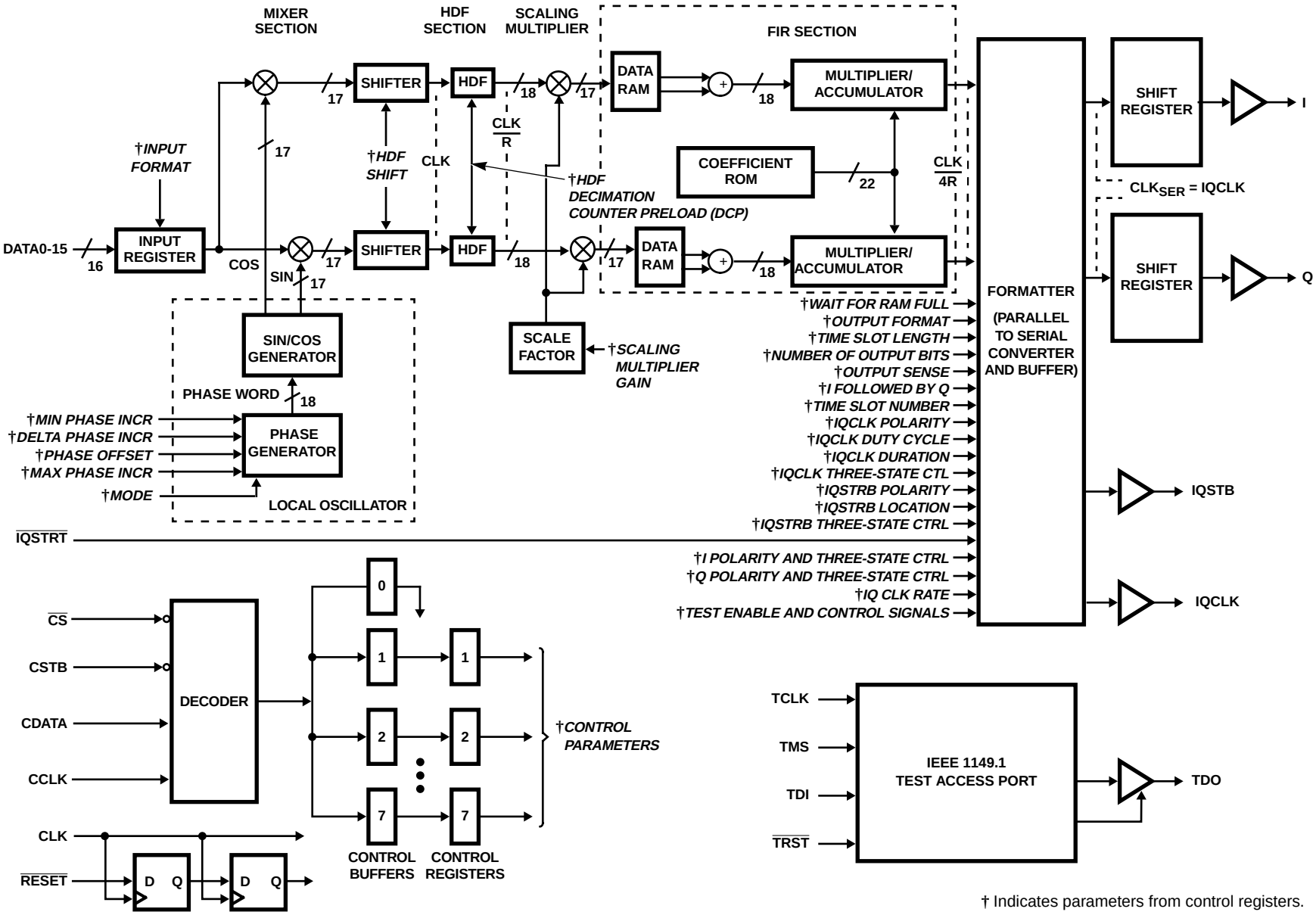
44 LEAD PLCC
TOP VIEW



Pin Description

NAME	TYPE	DESCRIPTION
V_{CC}	-	+5V Power.
GND	-	Ground.
DATA0-15	I	Input Data Bus. Selectable between two's complement and offset binary. DATA0 is the LSB.
CLK	I	Clock for input data bus. f_S is the frequency of CLK, which is also the input sample rate.
$\overline{\text{RESET}}$	I	$\overline{\text{RESET}}$ initializes the internal state of the DDC. During $\overline{\text{RESET}}$, all internal processing stops. $\overline{\text{RESET}}$ facilitates the synchronization of multiple chips for Auto Three-State operation. If the Force bits in Control Word 7 are inactive and the IEEE Test Access Port is in an Idle state, $\overline{\text{RESET}}$ causes the IQCLK, IQSTB, I and Q outputs to go to a high impedance state. All Control Registers are updated from their respective Control Buffer Registers on the third rising edge of CLK after the deassertion of $\overline{\text{RESET}}$. If $\overline{\text{RESET}}$ is deasserted t_{RS} nanoseconds prior to the rising edge of CLK, the internal reset will deassert synchronously. If t_{RS} is violated, then the circuit contains a synchronizer which will cause reset to be deasserted internally one or more clocks later. An initial reset is required to guarantee proper operation of the DDC. Active low.
I	O	The I output has three modes: I data; I data followed by Q data; real data.
Q	O	The Q output has two modes: Q data and the carry out of the Phase Adder.
IQCLK	O	IQ Clock: Bit or word clock for the I and Q outputs.
IQSTB	O	IQ Strobe: Beginning or end of word indicator for I and Q.
$\overline{\text{IQSTRT}}$	I	IQ Start: Initiates output data sequence. Active low.
CDATA	I	Control Data: Port for control data input.
CCLK	I	Control Data Clock: Control data input bit clock.
CSTB	I	Control Data Strobe: Beginning of word indicator for control data.
$\overline{\text{CS}}$	I	Chip Select: Enables control data loading of DDC. Active low.
TCK	I	Test Clock: Bit Clock for IEEE 1149.1 Data. This signal should be either tied low or pulled high when the TAP is not used.
TMS	I	Test Port Mode Select: This signal should be either left unconnected or pulled high when the TAP is not used.
TDI	I	Test Data Input for IEEE Test Port: This signal should be either left unconnected or pulled high when the TAP is not used.
TDO	O	Test Data Output for IEEE Test Port: This output will be in the high impedance state when the TAP is not used.
$\overline{\text{TRST}}$	I	Test Port Reset. Active Low. This signal should be tied low when the TAP is not used.

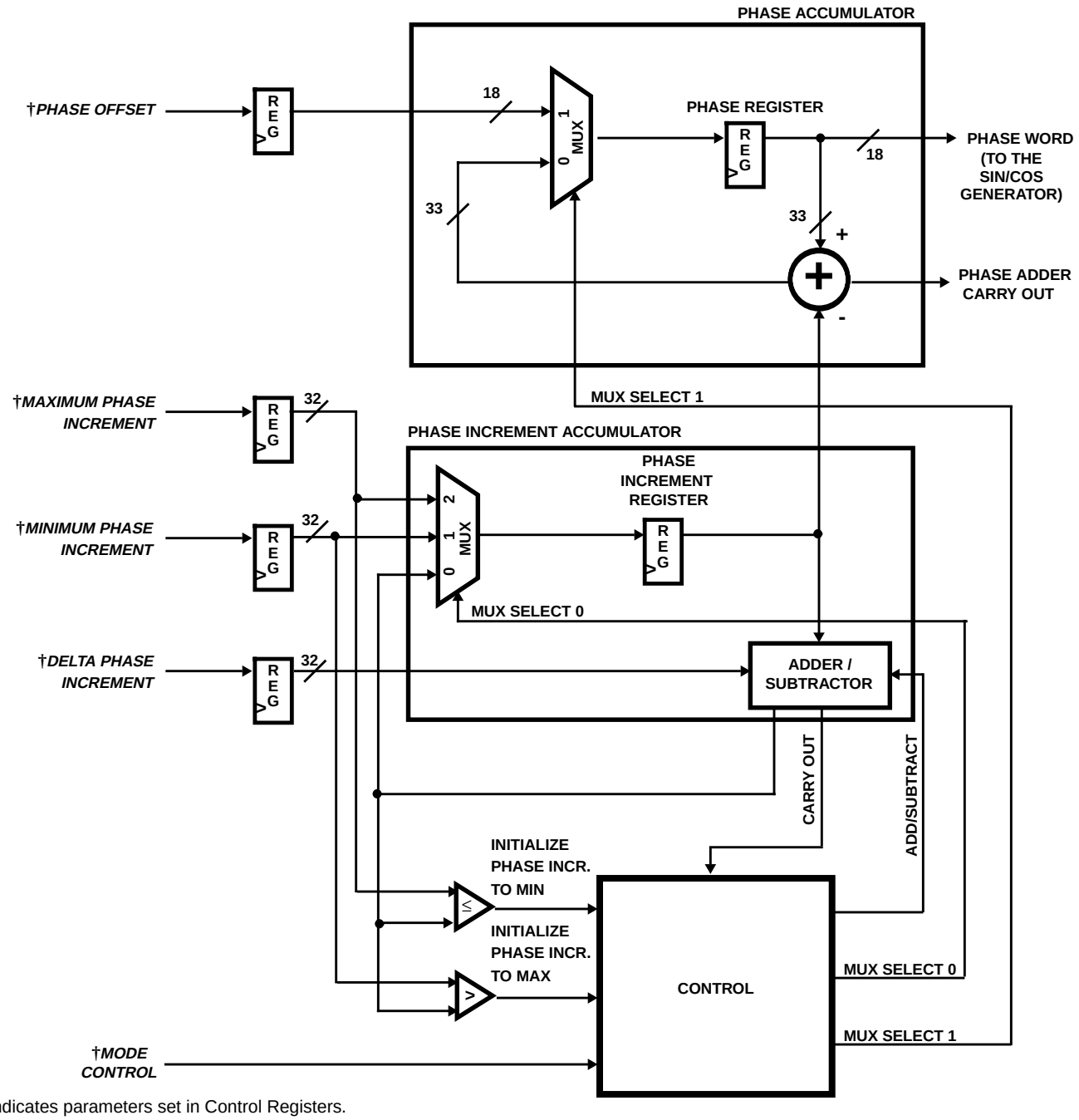
DDC Functional Block Diagram



† Indicates parameters from control registers.

FIGURE 1. FUNCTIONAL BLOCK DIAGRAM

Phase Generator Block Diagram



† Indicates parameters set in Control Registers.

FIGURE 2. PHASE GENERATOR BLOCK DIAGRAM

Functional Description

The primary function of the DDC is to extract a narrow frequency band of interest from a wideband input, convert that band to baseband and output it in either a quadrature or real form. This narrow band extraction is accomplished by down converting and centering the band of interest at DC. The conversion is done by multiplying the input data with a quadrature sinusoid. A quadrature lowpass filter is applied to the multiplier outputs. Identical real lowpass filters are provided in the in-phase (I) and quadrature phase (Q) processing branches. Each filtering chain consists of a cascaded HDF and FIR filter, which extracts the band of interest. During filtering, the signal is decimated by a rate which is proportional to the output bandwidth. The bandwidth of the resulting signal is the double sided passband width of the lowpass filters. An Output Formatter manipulates the filter output to provide the data in a variety of serial data formats.

Local Oscillator

Signal data clocked into the DATA0-15 input of the DDC is multiplied by a quadrature sinusoid in the Mixer Section (see Figure 1). The data input to the DDC is a 16-bit real data stream which is sampled on the rising edges of CLK. It can be in two's complement or offset binary format.

The input data is passed to a mixer, which is composed of two real multipliers. One of these multiplies the input data samples by the in-phase (cosine) component of the quadrature sinusoid, and the other multiplies the input data samples by the quadrature (sine) component. The in-phase and quadrature data paths are designated I and Q respectively. The sine and cosine are generated in the local oscillator as shown in Figure 1.

The local oscillator is programmed to produce a quadrature sinusoid with programmable frequency and phase. The frequency can be constant (Continuous Wave - CW), linearly increasing (up chirp), linearly decreasing (down chirp), or linear up/down chirp. The initial phase of the waveform is set by the phase offset.

The phase, frequency and chirp limits of the quadrature sinusoid are controlled by the Phase Generator (Figure 2). The output of the Phase Generator is an 18-bit phase word that represents the current phase angle of the complex sinusoid. The Phase Generator automatically increments the phase angle by a preprogrammed amount on every rising edge of CLK. Stepping the output phase from 0 through full scale (2^{18} - 1) steps the phase angle of the quadrature sinusoid from 0 to $(-2+2^{-17})\pi$ radians. ***NOTE: The phase is stepped in a clockwise (decreasing) direction to support down conversion.*** The frequency of the complex sinusoid is determined by the number of clocks needed for the phase to step though its full range of 2π radians. The required phase increment for a given local oscillator frequency is calculated by:

$$\text{Phase Increment} = \text{INT}\left[(f_C/f_S)\,2^{33}\right]\text{H}$$
$$f_C = (\text{Phase Incr})\,f_S\,2^{-33};\ 0 < f_C < f_S/2 \qquad \text{(EQ. 1)}$$

where:

f_C is the desired local oscillator frequency

f_S is the input sampling frequency

Phase Increment is the Control Word Value (in Hex)

There are five parameters which control the Phase Generator: Phase offset, minimum phase increment, maximum phase increment, delta phase increment and Mode Control. These values are programmed via Control Words 2, 3, and 4. Mode Control is used to select the function of the other parameters.

The phase offset is the initial setting of the phase word going to the SIN/COS Generator. Subsequent phases of the sinusoid are calculated relative to this offset. The minimum phase increment has two mode dependent functions: when the SIN/COS Generator is forming a CW waveform, the minimum phase increment is the phase step taken on every clock. When the SIN/COS Generator is producing a chirped sinusoid, the minimum phase increment is the smallest phase step taken. Maximum phase increment is only used during Chirped Modes; it is the largest allowable phase increment. During Chirp Modes, the delta phase increment is the difference between successive phase increments.

The four phase parameters are stored in their respective registers in the Phase Generator. The Phase Register stores the current phase angle. On the first clock following the deassertion of $\overline{\text{RESET}}$, the 18 MSBs of the Phase Register are loaded from the Phase Offset Register. On every rising edge of CLK thereafter, the output of the Phase Increment Register is subtracted from the 32 LSBs of the current phase. The 33-bit difference is stored back in the Phase Register on the next CLK. The 18 most significant bits of the Phase Register form the phase word, which is the input to the SIN/COS Generator.

Figure 3 gives a graphic representation of the phase parameters for the CW case. To understand their interrelationships, the phase should be visualized as the angle of a rotating vector. When the local oscillator in the DDC is programmed to generate a CW waveform, the multiplexers are configured so that the Minimum Phase Increment is stored in the Phase Increment Register; this value is subtracted from the output of the Phase Register on every CLK and the difference becomes the new Phase Register value. The Delta Phase Increment and Maximum Phase Increment are ignored when generating a CW.

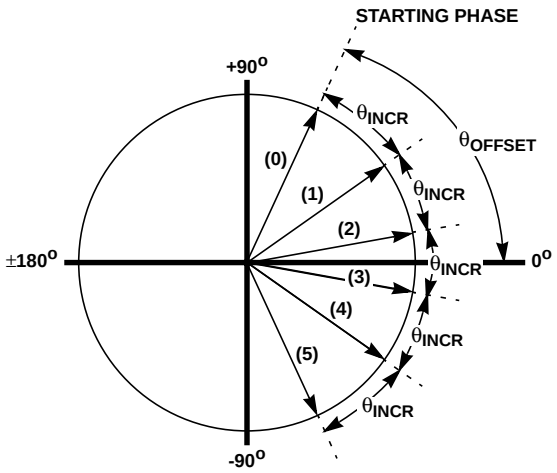


FIGURE 3. PHASE WORD PARAMETERS FOR CW CASE

In Up Chirp Mode the local oscillator generates a signal with a linearly increasing frequency (Figure 4A). The Phase Increment Register is initially loaded with the minimum Phase Increment value; on every clock, the contents of the Phase Increment Register is subtracted from the current output of the Phase Register. Simultaneously, the Delta

Phase Increment Register is added to the 24 LSBs of the output of the Phase Increment Register. On the next CLK, that sum is stored back in the Phase Increment Register, the new phase is stored in the Phase Register and the process is repeated. The phase increment is allowed to grow until the next phase increment would equal or exceed the maximum phase increment value. When this happens, the Phase Increment Register is reset to the minimum phase increment and the cycle starts over again.

NOTE: The phase increment is never equal to the maximum phase increment, since the Phase Increment Register is reloaded if the next phase increment value would be greater than the maximum phase increment.

From the time the Phase Generator starts at the minimum phase increment until it reaches the maximum phase increment, the phase word on clock n is given by:

$$\text{Phase Word} = \text{Phase Offset} - [\text{Minimum Phase Increment} + n\ (\text{Delta Phase Increment})] \quad \text{(EQ. 2)}$$

An example of the outputs of the Phase Increment Register, Phase Register, and the I output of the SIN/COS Generator are shown in Figure 4B.

In Down Chirp Mode the local oscillator generates a signal with a linearly decreasing frequency (Figure 5A). The maximum phase increment is loaded into the Phase Increment Register and the phase offset value goes into the Phase Register. The delta phase increment is subtracted from the 24 LSBs of the phase increment to form a new phase increment at each clock. The phase increment is allowed to diminish until it reaches the minimum phase increment value, then it is reset to the maximum phase increment value and the cycle is repeated. Note that the value of the phase increment can be equal to, but never less than the minimum phase increment, since the Phase Increment Register is reloaded if the next phase increment value would be less than the minimum phase increment. This feature protects the DDC from exceeding the Nyquist frequency. In this case, from the time the Phase Generator starts at the maximum phase increment until it reaches the minimum phase increment, the phase word on clock n is given by:

$$\text{Phase Word} = \text{Phase Offset} - [\text{Minimum Phase Increment} - n\ (\text{Delta Phase Increment})] \quad \text{(EQ. 3)}$$

See Figure 5B for a graphical representation of this process.

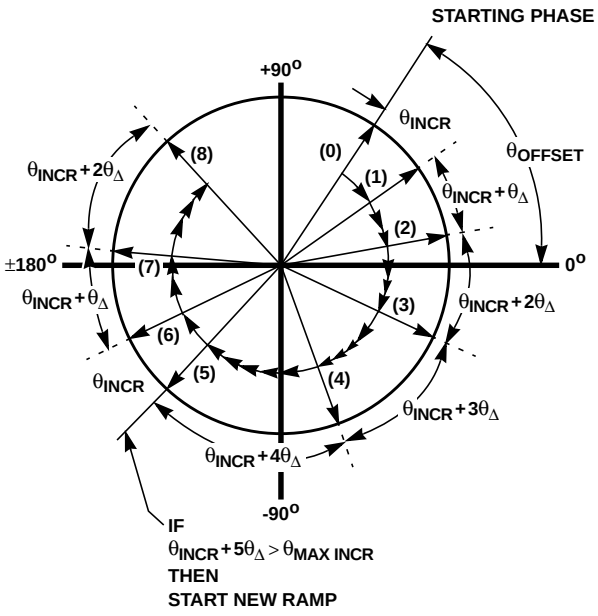


FIGURE 4A. PHASE WORD DURING UP CHIRP

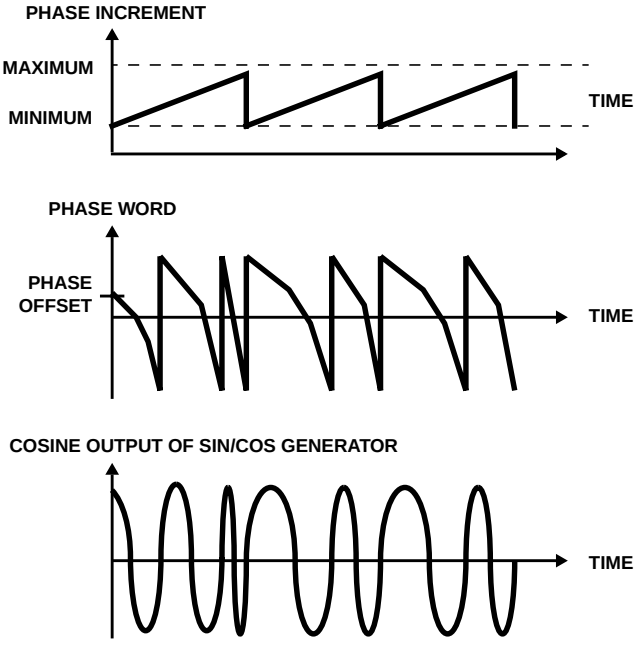


FIGURE 4B. UP CHIRP

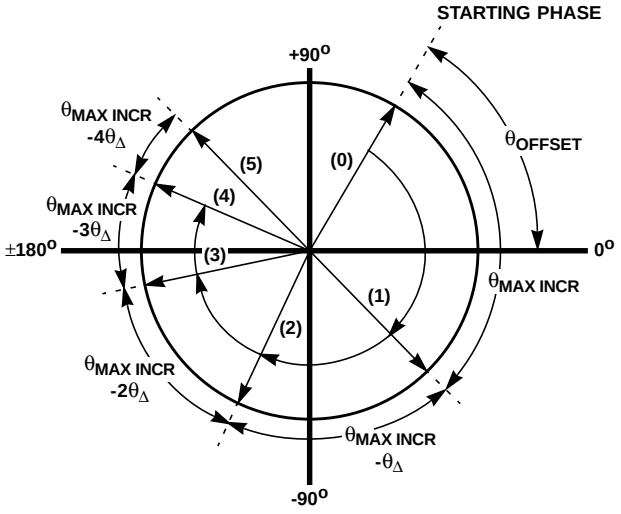


FIGURE 5A. PHASE WORD DURING DOWN CHIRP

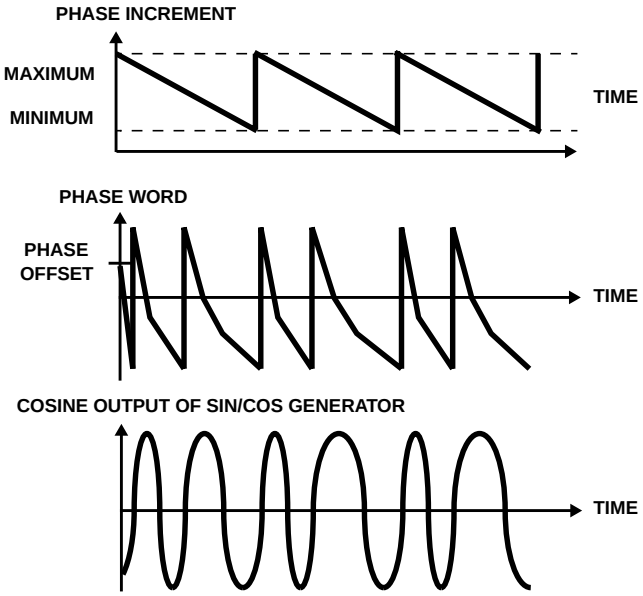


FIGURE 5B. DOWN CHIRP

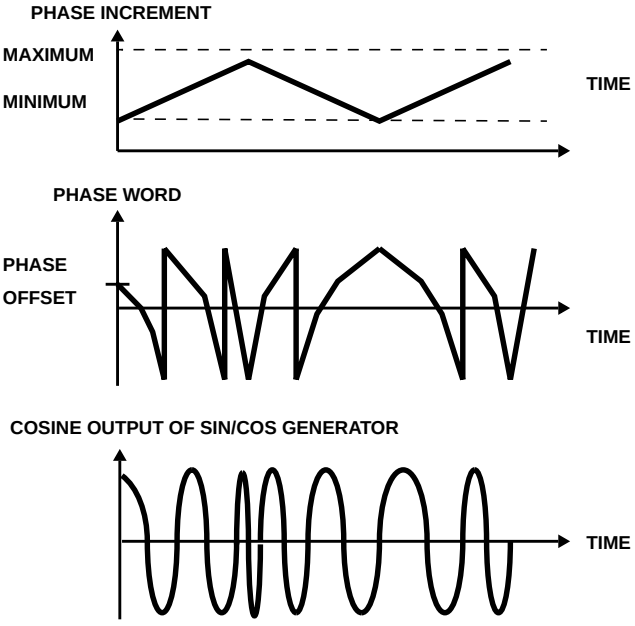


FIGURE 6. UP/DOWN CHIRP

In Up/down Chirp Mode, the phase accumulator is set to the phase offset value and the minimum phase increment is loaded into the Phase Increment Register. The delta phase increment is added to the 24 LSBs of the Phase Increment Register to form a new phase increment at each clock. The phase increment is allowed to grow until it nears the maximum phase increment value (as defined in the up chirp description). The delta phase increment value is then subtracted from the least significant bits of the Phase Increment Register to form a new phase increment at each clock. The phase increment is allowed to diminish until it reaches the minimum phase increment value (as defined in the down chirp description). The Phase Increment Register is then reloaded with the minimum phase increment, and the up/down cycle begins again. See Figure 6 for a graphical representation of this process.

The minimum and maximum phase increments have allowable values from 0 to 2^{32}-1. This corresponds to the phase increment:

$$0 < \text{Phase Increment} < \pi(1 - 2^{-32}) \text{ radians} \quad \text{(EQ. 4)}$$

The Delta Phase Increment parameter can take on values from 0 to 2^{24} - 1 which corresponds to the Delta Phase Increment:

$$0 < \text{Delta Phase Increment} < \pi(2^{-8} - 2^{-32}) \text{ radians} \quad \text{(EQ. 5)}$$

The output of the phase accumulator forms the input to the SIN/COS Generator which in turn produces a quadrature vector which rotates clockwise: the outputs are cos(ωn) and -sin(ωn). The outputs of the SIN/COS Generator are two's complement values which are scaled to prevent overflow in subsequent operations in the DDC under normal operation. The scale factor has a negligible effect on the end to end DDC gain.

The frequency resolution of the DDC = (frequency of CLK)/(Number of Phase Register bits). At the maximum clock rate, this results in a frequency selectivity of 75MHz/2^{33} = 0.009Hz. The 18-bit phase word yields a phase noise figure of greater than 102dB.

Mixer

The Mixer performs quadrature modulation by multiplying the output of the SIN/COS Generator by the input data. The outputs of the I and Q multipliers are symmetrically rounded to 17 bits to preserve the 102dB spurious free dynamic range (SFDR). The result of the quadrature modulation process is passed to the High Decimating Filter (HDF) Section.

High Decimation Filter

The High Decimation Filter (HDF) Section is comprised of two real HDF filters, one processing the I data branch and one processing the Q data branch. Each branch has the lowpass response shown in Figure 7. The normalized HDF frequency impulse response is given by the equation:

$$H(f) = \left[\frac{\text{Sin}(\pi F_S)}{\text{Sin}(\pi F_S/R)}\right]^5 \left[\frac{1}{R}\right]^5 \quad \text{(EQ. 6)}$$

where F_S is the input sampling rate; R is the decimation (rate change) factor.

Figure 7A shows this equation plotted from DC to the first null, while Figure 7B shows the equation plotted from DC response to f_S.

NOTE: The HDF is a true FIR filter; i.e., the phase is linear.

The data path through the HDF was designed to ensure a true 16-bit noise floor (approximately 98dB) at the output of the DDC. The structure of the HDF filter used in the DDC is a five stage decimation filter. The width of each successive stage decreases such that the LSBs are lost due to truncation [1]. As a result, the data must be processed in the MSBs of the filter so that the noise due to truncation is below the required noise floor. Thus, the input data of the HDF must be shifted so that its output data fills the HDF output word. The shift is a function of the desired HDF decimation rate R and the number of HDF filter stages (which is fixed at 5). The shift is performed by the Data Shifter, which positions the input data to the HDF for the maximum dynamic range while avoiding overflow errors. The shift factor is programmed into the Shift field of Control Word 4. The value in this field is calculated by the equation:

$$\text{Shift} = 75 - \text{Ceiling}(5 \log_2(R)) \quad \text{(EQ. 7)}$$

where R is the HDF decimation factor and Ceiling(X) denotes the ceiling function of X; i.e., the result is X if X is an integer, otherwise the result is the next higher integer.

During $\overline{\text{RESET}}$, the HDF is initialized and will not output any information until it is filled with new data.

NOTE: The output rate of the HDF is CLK divided by the HDF decimation factor (CLK/R). The HDF decimation counter preload (DCP) is programmed in Control Word 5, bits 21-35 and has the value: DCP = R -I, where R is the HDF decimation factor.

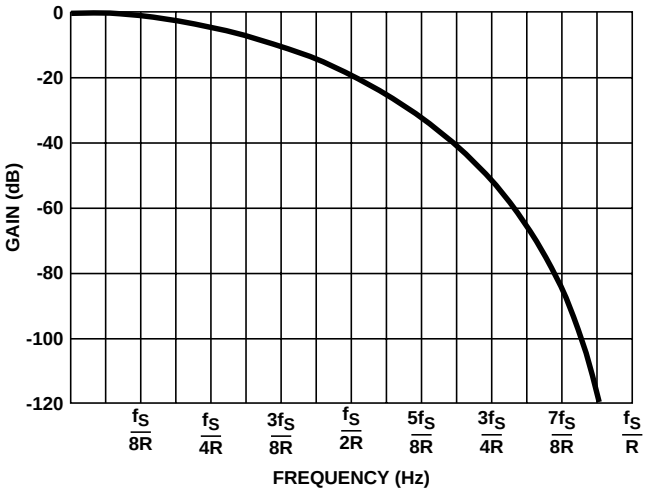


FIGURE 7A. FREQUENCY RESPONSE OF HIGH DECIMATION FILTER FROM DC TO FIRST NULL (FOR R = 16) Gain (dB) = 20log [H(f)]

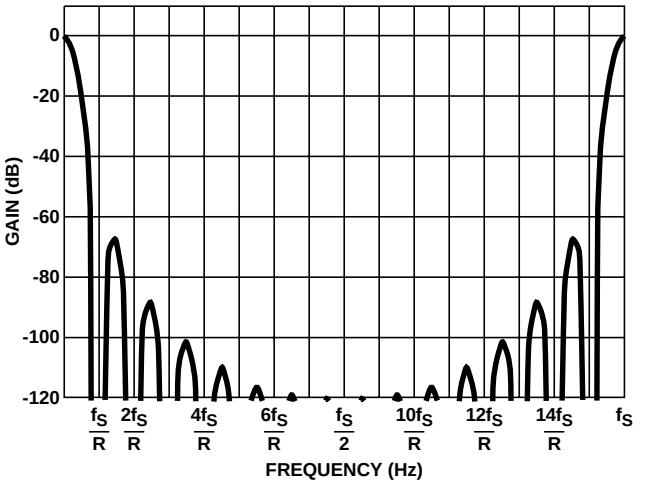


FIGURE 7B. DDC HC FREQUENCY RESPONSE (FOR R = 16) Gain (dB) = 20log[H(f)]

Scaling Multipliers

The output of each HDF is passed to a Scaling Multiplier. The Scaling Multipliers are used to compensate for the HDF gain, which is between 1 (inclusive) and 0.5 (non-inclusive), or (0.5, 1.0). The gain through the HDF is dependent on the decimation factor: when the decimation is an even power of two, the HDF gain is equal to 1; otherwise, the gain must be compensated for in the Scaling Multiplier. The HDF gain is given by the equation:

$$\text{HDF Gain} = R^5/2^{\text{CEILING}(5\log_2(R))} \quad \text{(EQ. 8)}$$

where R is the HDF decimation factor. The compensating Scale Factor, which is input to both Scaling Multipliers, is given by the equation:

$$\text{Scale Factor} = 2^{\text{CEILING}(5\log_2(R))}/R^5 \quad \text{(EQ. 9)}$$

where R is the HDF decimation factor.

NOTE: The Scale Factor falls in the interval [1, 2). The output of the scaling multiplier is symmetrically rounded to 17 bits.

The binary formats of the inputs and outputs of the scaling multiplier are as follows:

Input from HDF:	$a_0(-2^0).\ a_1(2^{-1})\ a_2(2^{-2})...\ a_{17}(2^{-17})$
Scale factor:	$a_0(2^0).\ a_1(2^{-1})\ a_2(2^{-2})...\ a_{15}(2^{-15})$
Output:	$a_0(-2^0).\ a_1(2^{-1})\ a_2(2^{-2})...\ a_{16}(2^{-16})$

FIR Filter

The Scaling Multiplier output is passed to the FIR Filter, which performs aliasing attenuation, passband roll off compensation and transition band shaping. The FIR Filter Section is functionally two identical 121 tap lowpass FIR filters, one each for the I and Q channel. The two filters are each implemented as sum of products, each with a single multiplier, with the coefficients stored in ROM. The filters' passbands are precompensated to be the inverse of the response of the HDF. The frequency responses of the HDF, FIR, and Composite HDF/FIR filters are shown in Figure 8. The composite passband of the HDF and FIR filter frequency response is shown in Figure 9. The FIR coefficients are scaled so that the maximum gain of the composite filter is less than or equal to 0dB. The composite passband ripple is less than 0.04dB.

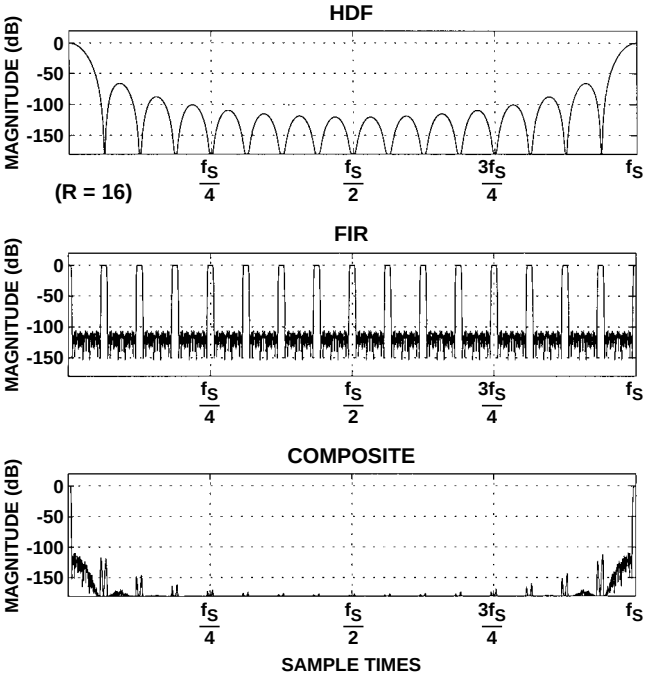


FIGURE 8A. DDC HDF, FIR, AND COMPOSITE FILTER RESPONSE (FOR R = 16)

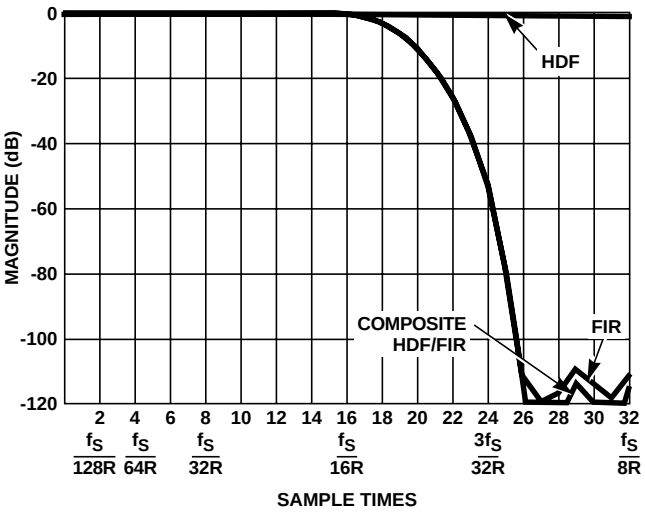


FIGURE 8B. DDC FILTER RESPONSES (FOR R = 16)

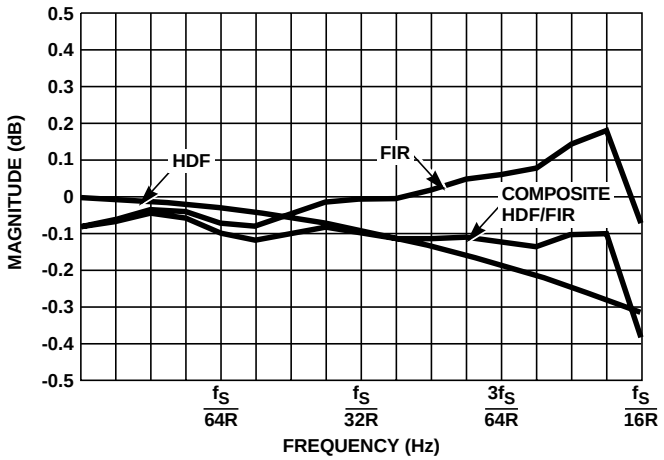


FIGURE 9. FIR COMPENSATION FOR HDF ROLL OFF (FOR R = 16)

The coefficients of the filter are quantized to 22 bits to preserve greater than 106dB of stopband attenuation. The sum of products of each filter output calculation is a 38-bit number with 37 fractional bits.

When a quadrature output is selected, the outputs of the FIR filters are decimated by a factor of four. When real output is selected, only the I output is active. The output is decimated by two in this case. When Filter Only Mode is selected, only the I filter path is active and its output is decimated by four.

The composite filter bandwidths are a function of the HDF decimation rate and the FIR Filter shape. The double sided bandwidths are specified by Equations 10 and 11.

$$-3dB\ BW_{DS} = 0.1375F_S/R \quad 16 < R < 16384 \qquad \text{(EQ. 10)}$$

$$-102dB\ BW_{DS} = 0.2002F_S/R \quad 16 < R < 16384 \qquad \text{(EQ. 11)}$$

where F_S = CLK; R = HDF Decimation Factor.

The single sided bandwidths are specified in Equations 12 and 13.

$$-3dB\ BW_{SS} = 0.06875F_S/R \qquad \text{(EQ. 12)}$$

$$-102dB\ BW_{SS} = 0.100097F_S/R \qquad \text{(EQ. 13)}$$

where F_S = CLK; R = HDF Decimation Factor.

NOTE: The output data rate of the FIR is the HDF output rate divided by either 2 or 4, depending on mode. Recall the HDF output rate is CLK/R. (See Table 1.)

TABLE 1. FIR OUTPUT RATE AND DECIMATION

OUTPUT MODE	FIR OUTPUT RATE	FIR DECIMATION
Real	CLK/2R	2
Complex	CLK/4R	4
Filter Only	CLK/4R	4

R - HDF Decimation Factor

Output Formatter

The circuit has two serial data outputs, I and Q. The timing of the output bits is referenced to IQCLK and IQSTB. There are several modes of operation for the data and control line interface, all of which were designed to be compatible with common microprocessors. These interface modes are selected by loading the appropriate control words (see Tables 3 through 10, with Table 9 containing most interface parameters).

Quadrature data output can occur in one of two ways: simultaneously or sequentially. The simultaneous method clocks out the I and Q data on their respective serial output pins. The I followed by Q method clocks I and Q out sequentially on the I output pin: the entire I word is serially clocked out first, then the entire Q word. In real data Output Mode, the Formatter converts the quadrature data to real and clocks it out serially on the I output pin. In all modes, the I and Q outputs return to the zero state after the last bit is transmitted.

When the “I followed by Q” signal (CW6, bit 35) is low, I data will appear on the I output and Q data will appear on the Q output.

When the “I followed by Q” signal (CW6, bit 35) is asserted, the Q output is inactive and I data, followed by Q data appear on the I output. When in this state, and both the “Test Enable” signal (CW1, Bit 3), and “Q Strobe on Rollover” signal (CW7, Bit 10) signal are asserted, the Phase

Generator Carry Out will appear on the output. Control Word 5 contains fields to set the number of output bits transmitted to the arithmetic representation and interface control of the serial output data. Control Word 4, Bits 31-32, allow selection of baseband centered quadrature on baseband offset quadrature complex outputs. Control Word 4, Bit 0, allows selection of spectral inversion. In addition, the output drivers for I, Q, IQCLK and IQSTB can be individually enabled or placed in a high impedance state using Control Word 6, Bits 20-28. These options are explained below.

When the “Output Spectrum” signal (CW4, bits 31-32) is set to “01”, then the real output data appears on the I output and the Q output in the I/Q separate mode. When in I Mode followed by Q Mode, the Q slot is also real data since the real mode outputs at twice the rate of the complex mode (CW4, bits 31-32 = 00).

When set for fixed point output, the output data can be in two's complement, offset binary or signed magnitude form. Data is converted to offset binary by complementing the most significant bit of a two's complement number. The length of the output data word can be 16, 24, 32 or 38-bits. The first three options are symmetrically rounded to the LSB of the output data; the fourth option represents the full 38-bit width of the accumulator and so represents exact arithmetic.

The output has a saturation option to prevent possible overflow due to a step input at power up. When Overflow Protection is enabled, the output is forced to be either the most positive or most negative number. Saturation is available in all four fixed point output options, and is set via Control Word 7, Bit 0.

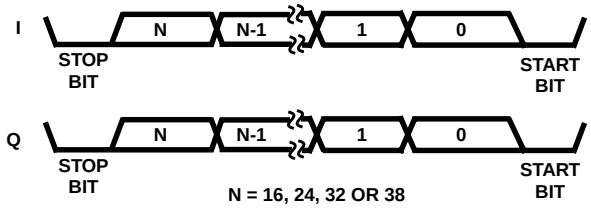
Data can also be output in single precision floating point format (see Table 2). For all output data formats, the internal calculations are performed in exact two’s complement integer arithmetic and the resulting data is converted in the Output Formatter.

TABLE 2. FLOATING POINT FORMAT

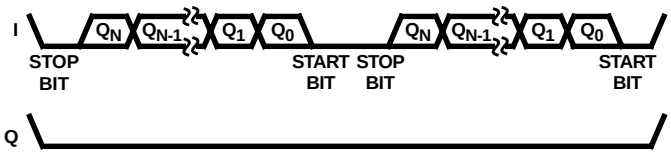
SIGN	EXPONENT	MANTISSA	
-2^0	2^7 to 2^0	Implied 1	0.2^{-1} to 2^{-23}

The I and Q pins can be programmed for either simultaneous or I followed by Q output. In simultaneous mode, the I and Q data appear on the I and Q pins, respectively. Each data sample is preceded by a leading zero bit, followed by the output data, followed by a trailing zero bit. In I followed by Q Mode, the output data appears on the I pin, and consists of a leading zero bit, then the I data, a trailing zero, a leading zero, the Q data, and finally a trailing zero bit. In Figures 10 through 12, the leading and trailing zero bits occur before bit 0 and after bit N, respectively.

$$\text{IQCLK Rate} = \frac{\text{CLK}}{\text{IQCLK Frequency}} - 1 \qquad \text{(EQ. 14)}$$



A. SIMULTANEOUS OUTPUT MODE



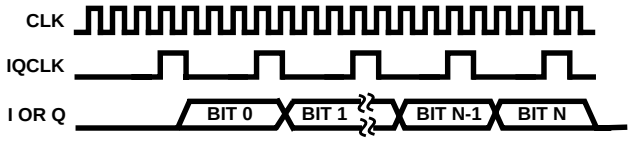
B. I FOLLOWED BY Q OUTPUT MODE

FIGURE 10. DATA OUTPUT MODES

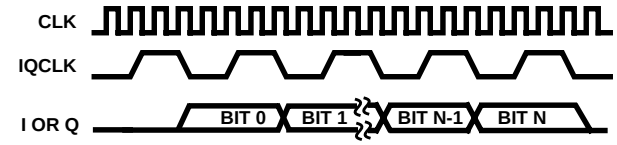
IQCLK is used to delineate the bit or word timing of the I and Q outputs. There are several options on the configuration of IQCLK, which are controlled with Control Word 6 (see Table 8). The frequency of IQCLK is programmed to be a fraction of the CLK frequency, from (CLK rate)/2 to (CLK rate)/8192 (see Equation 14). If IQCLK Rate = 0, then IQCLK remains in its inactive state and the output bits change on the rising edges of CLK.

IQCLK can be programmed to be active continuously, or only during I or Q data output via the IQCLK duration bit. Using the IQCLK Duty Cycle bit, IQCLK is selectable as either 50% duty cycle or to be high for one period of CLK. In addition, the Formatter can be set so that the data bits are clocked on either the positive or negative edges of IQCLK with the IQCLK Polarity bit. Figure 11 shows the various modes of operation with IQCLK Polarity programmed for active high operation.

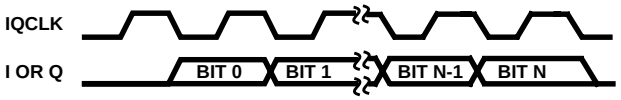
Control Word 6 also configures IQSTB, as shown in Figure 12. When programmed for Active Prior to Data Word, IQSTB is high for one period of IQCLK and terminates simultaneously with the beginning of the first data bit; otherwise it goes active with the beginning of the first bit and inactive with the end of the last bit. IQSTB can be programmed to be either active high or low.



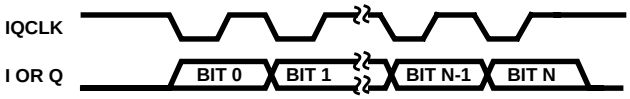
A. IQCLK DUTY CYCLE: ACTIVE TIME = CLK PERIOD (IQCLK POLARITY = 0)



B. IQCLK DUTY CYCLE: 50%

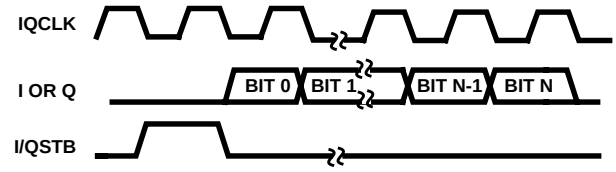


C. IQCLK DURATION: CONTINUOUS

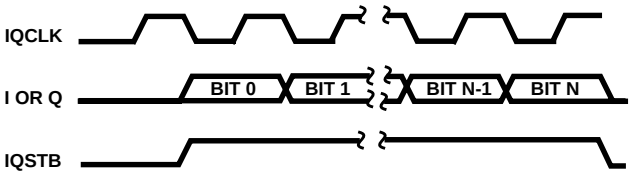


D. IQCLK DURATION: ACTIVE DURING I OR Q ONLY

FIGURE 11. TIMING FOR CLK, IQCLK, IQSTB, I AND Q



A. IQSTB ACTIVE PRIOR TO DATA WORD



B. IQSTB ACTIVE DURING DATA WORD

FIGURE 12. IQSTB TIMING

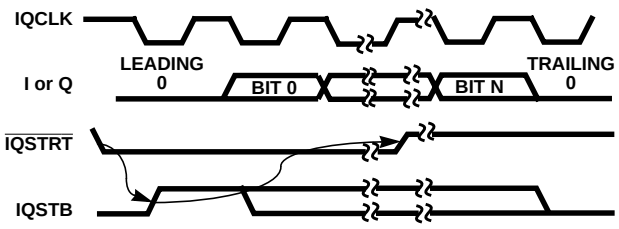


FIGURE 13. REQUESTED DATA OUTPUT TIMING

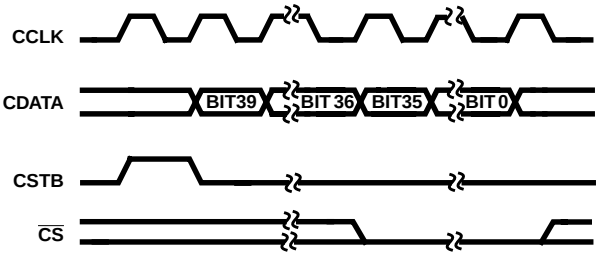


FIGURE 14. CONTROL WORD TIMING DIAGRAM

Data can be read out of the DDC on request through the use of the $\overline{\text{IQSTRT}}$ pin. After passing through the Output Formatter, the I and Q data are stored in output buffers, which are updated at the end of the FIR Filter processing cycle. The $\overline{\text{IQSTRT}}$ and IQSTB lines form a two line handshake as shown in Figure 13. $\overline{\text{IQSTRT}}$ initiates the request. If the buffer has data in it, the DDC will begin an output data sequence on the next edge of IQCLK. The DDC will then put out one bit per IQCLK until the output cycle is complete. In I followed by Q Mode, one $\overline{\text{IQSTRT}}$ will initiate an I output word followed by a Q output word. In real data Output Mode, one IQSTRT will initiate two samples of real data on the I pin.

To avoid the generation of multiple read cycles, IQSTRT must go inactive within 10 cycles of IQCLK after the initiation of IQSTB. The DDC will not update the output buffer again until the current output cycle has completed. When IQSTRT is used in this handshake mode, it must consist of pulses that satisfy the set up and hold requirements listed in the AC Timing Specifications and the pulses must occur at a rate of at least CLK/(HDF Decimation Factor x 4 -1). This mode of operation requires the Time Slot Number in Control Word 6 to be 0.

NOTE: When handshake mode is not used, $\overline{\text{IQSTRT}}$ should be at a logic low.

Auto Three-State Mode for IQCLK, IQSTB, I and Q allows multiple chips to operate using common data and output control lines. Each chip is assigned a Time Slot Number on the bus to use for outputting its data. All outputs programmed for Auto Three-State Mode are active during their time slot and are in a high impedance state at all other times. A time slot starts one CLK period prior to the beginning of the first bit of I or Q and ends (Time Slot Length) CLK periods afterwards. Assignment of a time slot is with reference to the deassertion of $\overline{\text{RESET}}$. The minimum possible Time Slot Length for a given application is:

$$\text{Length}_{MIN} = [(\text{Number of Output Bits} + 2) \times \text{Mode}] + 1; \text{ or} \quad \text{(EQ. 15)}$$

where Mode = 2 if the DDC is in either Real Output or I followed By Q Mode; else Mode = 1.

Note that Equation 15 is useful in all modes for calculating the number of IQCLKs necessary to complete one output data cycle. For a given decimation rate and output word length, the maximum value in the IQCLK Rate field is:

$$IQCLKRate_{MAX} = Floor\left[\frac{(R)\times 4}{Length_{MIN}}\right] - 1; \quad \text{(EQ. 16)}$$

where Floor(X) represents the integer part of X, R is the HDF decimation factor, 4 is the FIR decimation factor.

Example Clock Calculations

Clarification of the use of Equations 14-16, the calculation of the HDF and FIR clocks and the calculation of the IQCLK is best done by example:

The sample clock, CLK, is 10MHz . . . . . CLK = 10MHz
The HDF Decimation Factor, R, is 100 (which makes the decimation counter preload = 99). . . . . R = 100
The Output Mode is I followed by Q . . . . . Mode = 2
Complex output . . . . . FIR Decimation = 4
The desired number of output bits is 32.

1. We begin by identifying the HDF Input Rate:
   HDF Input Rate = CLK = 10MHz . . . . . CLK = 10MHz
2. Next we calculate the HDF Output Rate:
   HDF Output Rate = CLK/R = 10MHz/(100) = 100kHz
   . . . . . .HDF Output Rate = 100kHz
3. Next we calculate the FIR output Rate:
   FIR Output Rate = CLK/4R = 25kHz.
   . . . . . FIR Output Rate = 25kHz
4. Next we calculate the minimum time slot length:
   Equation 15:
   $Length_{MIN}$ = [(Number of Output Bits + 2) x Mode] +1
   where the number of output bits = 32 and the Mode is 2 because of the I followed by Q output selection.
   $Length_{MIN}$ = [(32 + 2) x 2] +1 = 69 IQCLKs
   . . . . . $Length_{MIN}$ = 69 IQCLKs
5. Next we calculate the IQCLK frequency:
   IQCLK frequency = [(F_S)($Length_{MIN}$)/(R)(4)] - 1
   IQCLK frequency = [(10MHz)(69)/(100)(4)] - 1 = 1.725MHz
   The IQCLK frequency can be no slower than 1.725MHz if all of the bits are to be output of the DDC in a time slot.
   . . . . . .Slowest Serial Output Rate = 1.725MHz
6. The Programmed value for the maximum IQCLK Rate, from Equation 16, is:
   $IQCLKRATE_{MAX}$ = Floor[(R) x 4/ $Length_{MIN}$] -1
   $IQCLKRATE_{MAX}$ = Floor[(100 x 4)/69] - 1 = 4
   The IQCLKRATE can be not greater than 4 if all of the bits are to be output of the DDC in a time slot.
   . . . . . Control Word Value for IQCLK Ratemax =
   [00004]H; 0 0000 0000 0100_{LSB}
7. Let's sanity check with Equation 14.
   IQCLK Rate = [(CLK/IQCLKfreq)-1] = [10E6/1.725E6] -1 = 4.
   This checks!

Control Word Input

The DDC has eight 40-bit control words which are loaded through the four pin control interface. The format and timing of this interface is compatible with the serial interface timing of most common DSP microprocessors (see Figure 14). The words are shifted MSB first, where bit 39 of the control word is the MSB. Bits 39 through 37 are the control word address, i.e., the target control buffer. $\overline{CS}$ must go low before bit 35 is clocked in. All 40 bits of the control word must be loaded. The formats of the control words are shown in Tables 3 through 10.

The control words are double buffered: each control word is initially loaded into one of eight control buffers for subsequent down loading into the corresponding Control Register. The internal circuitry of the DDC uses the Control Registers to regulate its operation. Control buffers can be downloaded in one of two ways. Loading a Buffer Register with bit 36 = 1 causes all Control Registers to be updated from their respective control buffers when the current word is finished loading. If bit 36 = 0, then only that control buffer is updated and the operation of the DDC is not affected. All Control Registers are updated from their respective buffers on the third rising edge of CLK following the deassertion of RESET. ***NOTE: Control Word 0 is unique in that it is only used to update the seven Control Registers, and it is recognized by the DDC regardless of the state of CS.*** In systems with multiple DDCs, this allows the user to update the configuration of all chips simultaneously without using RESET.

To ensure that the control information is properly loaded, the frequency of CLK must be greater than the frequency of CCLK. In addition, $\overline{RESET}$ must remain inactive during the loading of a control word.

TABLE 3. DESTINATION ADDRESS = 0

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	000 = Control Word 0
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-32	Reserved	All Zeroes

TABLE 4. PHASE GENERATOR/TEST ENABLE/OUTPUT REGISTER

DESTINATION ADDRESS = 1		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	001 = Control Word 1
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-4	Minimum Phase Increment	Bits 35-4 = $2^{31}...2^{0}$. Range: 0 < Minimum Phase Increment < π $(1-2^{-32})$ radians. In the CW mode this is the phase increment of the NCO which is added to the NCO intitial phase offset state. The desired Sin/Cos generator (local oscillator) frequency is set by the equation: **f_C = (phase increment)$f_S 2^{-33}$**; where f_C is the desired local oscillator frequency, f_S is the input sampling frequency, and phase increment is the control word value in hexidecimal. To calculate the value to be programmed into this field, use this equation: **phase increment = INT[f_C / f_S)2^{33}]hex** Some examples of phase increments and local oscillator frequencies: 00000000h: f_C = zero frequency 00000001h: $f_C = f_S / 2^{33}$ - lowest frequency (75MHz x 2^{-33} = 8.73mHz) 10000000h: $f_C = f_S /32$ 20000000h: $f_C = f_S /16$ 40000000h: $f_C = f_S /8$ 80000000h: $f_C = f_S /4$ ffffffffh: $f_C = (0.49999) f_S$ - highest frequency (75MHz x 2^{-33} = 37.49MHz) In the CHIRP modes, this is the smallest allowable phase increment. In the Filter Only mode, this parameter should be set to 0.
3	Test Enable	0 = Test Features Disabled 1 = Test Features Enabled
2-0	Phase Generator Mode	000 = Filter Only 001 = Normal Mode (CW) 010 = Reserved 011 = Up Chirp 100 = Reserved 101 = Down Chirp 110 = Reserved 111 = Up/Down Chirp ***Note that the lsb sets the gain through the DDC as follows:*** ***0 = Gain is1*** ***1 = Gain is 2***

TABLE 5. PHASE GENERATOR REGISTER

DESTINATION ADDRESS = 2		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	010 = Control Word 2
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-32	Reserved	All Zeroes
31-0	Maximum Phase Increment	Bits 31-0 = 2^{31}... 2^{0}. Range: is 0 < Maximum Phase Increment < $\pi(1-2^{-32})$ radians. This parameter is only used in the CHIRP modes, and this is the largest allowable phase increment. Set to 0 in the Filter Only and CW modes.

TABLE 6. PHASE GENERATOR/OUTPUT TIME SLOT REGISTER

DESTINATION ADDRESS = 3		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	011 = Control Word 3
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-32	Reserved	All Zeroes
31-18	Time Slot Length	Time Slot Length in IQCLK Periods; Bits 31-18 = 2^{13}... 2^{0}. Range is (19,25, 33, 37, 39, 49, 65 and 77) The equation for calculating the value for this field is: **TSL = [[(Number of Output Bits + 2)Mode] + 1]Hex**; where mode is 2 if the DDC is in either the real or I followed by Q mode. Mode is 1 for all other DDC operational modes. Allowable Minimum Time Slot Lengths: (18)1 + 1 = 19 (13 hexidecimal) (18)2 + 1 = 37 (25 hexidecimal); Real Output or I followed by Q (24)1 + 1 = 25 (19 hexidecimal) (24)2 + 1 = 49 (31 hexidecimal); Real Output or I followed by Q (32)1 + 1 = 33 (21 hexidecimal) (32)2 + 1 = 65 (41 hexidecimal); Real Output or I followed by Q (38)1 + 1 = 39 (27 hexidecimal) (38)2 + 1 = 77 (4d hexidecimal); Real Output or I followed by Q
17-0	Phase Offset	Starting Phase Angle of Phase Accumulator; Range = 0 to 2π. Bits 17-0 = 2^{32}... 2^{15}. Some example phase offset hexidecimal values : `0000` - 0 `1000` - π/2 `2000` - π `3000` - 3π/2 `3fff` - 2π

TABLE 7. PHASE GENERATION/HDF.OUTPUT REGISTER

DESTINATION ADDRESS = 4		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	100 = Control Word 4
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-33	Reserved	All Zeroes
32-31	Output Spectrum	00 = No Up Conversion,Complex Output 01 = Up Convert by f”/4, Real Output 10 = Up Convert by f”/2,Complex Output 11 = Reserved Mode
30-7	Delta Phase Increment	24-Bit Delta Phase Increment. Bits 30-7 = 2^{23}... 2^{0}. Range: 0 < Delta Phase Increment < π (2^{-8}-2^{-32})
6-1	HDF Data Shift (Shift Factor)	16-Bit HDF Gain Compensation Number - the shift portion. HDF Input Data Shift (Towards LSB). Bits 6-1 = 2^{5}...2^{0}. Range: 0 ≤ Shift Factor ≤ 55 decimal; Range: [0 ≤ Shift Factor ≤ 37]hex Calculate the value for this field using this equation: **HDF Data Shift = [75 - Ceiing(5 $\log_2$(R))]hex** Note: $\log_2(x) = (3.32)\log(x)$
0	Spectral Reverse	0 = Normal Output 1 = Spectrally Reversed Output

TABLE 8. HDF/OUTPUT REGISTER

DESTINATION ADDRESS = 5		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	101 = Control Word 5
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-21	HDF Decimation Counter Preload (HDF DCP)	HDF Decimation Counter Preload. Range: 15 < HDF Decimation Counter Preload < 32,767 Calculate the value for this feild using this equation: **HDF DCP = R - 1;** where R is the HDF decimation (rate change) factor. Common HDF decimation (rate change) factors and associated hexidecimal HDF DCP values: `000f`: R = 16 `00ff`: R = 128 `01ff`: R = 512 `03ff`: R = 1,024 `07ff`: R = 2,048 `0fff`: R = 4,096 `1fff`: R = 8,192 `3fff`: R = 16,384 `7fff`: R = 32,768

TABLE 8. HDF/OUTPUT REGISTER (Continued)

<table>
<tr><th colspan="3">DESTINATION ADDRESS = 5</th></tr>
<tr><th>BIT POSITION</th><th>FUNCTION</th><th>DESCRIPTION</th></tr>
<tr><td>20-5</td><td>Scaling Multiplier Gain (Scale Factor)</td><td>16-Bit HDF Gain Compensation Number - the multiplier portion.
Range: 1 ≤ Scale Factor < 2,
Field Format = $2^{0}.2^{-1}...2^{-15}$.
Calculate the value for this field using this equation:
Scale Factor = $2^{CEILING(5log_2(R))}/(R)^5$; where R is the HDF decimation (rate change) factor and CEILING(x) is equal to x for integer values, otherwise is equal to the next higher integer.
Common HDF decimation factors (R), decimation counter preload (DCP) and Scale Factors (SF) values:
<table><tr><th>R</th><th>DCP(dec)</th><th>DCP(hex)</th><th>SF(dec)</th><th>SF(hex)</th></tr><tr><td>16</td><td>15</td><td>000f</td><td>1.000</td><td>8000</td></tr><tr><td>128</td><td>127</td><td>007f</td><td>1.000</td><td>8000</td></tr><tr><td>512</td><td>511</td><td>01ff</td><td>1.00</td><td>8000</td></tr><tr><td>1,024</td><td>1,023</td><td>03ff</td><td>1.000</td><td>8000</td></tr><tr><td>2,048</td><td>2,047</td><td>07ff</td><td>1.000</td><td>8000</td></tr><tr><td>4,096</td><td>4,095</td><td>0fff</td><td>1.000</td><td>8000</td></tr><tr><td>8,192</td><td>8,191</td><td>1fff</td><td>1.00</td><td>8000</td></tr><tr><td>16,384</td><td>16,382</td><td>3fff</td><td>1.00</td><td>8000</td></tr><tr><td>32,768</td><td>32,768</td><td>7fff</td><td>1.000</td><td>8000</td></tr></table>
Note that the Scale Factor is 1 (8000hex) for power of 2 decimation factors.
The compensation for the HDF gain is performed with a shifter and a multiplier. Thus to program the HDF Gain compensation, there is an associated Shift Factor and the Scale Factor. As the Decimation Factor increases, the multiplier moves away from the value 1 and approaches the value 2. When the calculated value for the multiplier equals or exceeds 2, the shifter is incremented and the multiplier returns to 1 and increases towards 2 again as the Decimation Factor increases.
As an example, the table below details the values of Scale Factor for values of R from 16 to 32:
<table><tr><th>R</th><th>DCP(dec)</th><th>DCP(hex)</th><th>SF(dec)</th><th>SF(hex)</th></tr><tr><td>16</td><td>15</td><td>000f</td><td>1.000000000</td><td>8000</td></tr><tr><td>17</td><td>16</td><td>0010</td><td>1.477013647</td><td>BD0E</td></tr><tr><td>18</td><td>17</td><td>0011</td><td>1.109857915</td><td>8E0F</td></tr><tr><td>19</td><td>18</td><td>0012</td><td>1.693916116</td><td>D8D2</td></tr><tr><td>20</td><td>19</td><td>0013</td><td>1.310720000</td><td>A7C5</td></tr><tr><td>21</td><td>20</td><td>0014</td><td>1.026983417</td><td>8374</td></tr><tr><td>22</td><td>21</td><td>0015</td><td>1.627707993</td><td>D058</td></tr><tr><td>23</td><td>22</td><td>0016</td><td>1.303318981</td><td>A6D3</td></tr><tr><td>24</td><td>23</td><td>0017</td><td>1.053497942</td><td>86D9</td></tr><tr><td>25</td><td>24</td><td>0018</td><td>1.717986918</td><td>DBE6</td></tr><tr><td>26</td><td>25</td><td>0019</td><td>1.412060017</td><td>B4BE</td></tr><tr><td>27</td><td>26</td><td>001a</td><td>1.169233029</td><td>95A9</td></tr><tr><td>28</td><td>27</td><td>001b</td><td>1.949663831</td><td>F98E</td></tr><tr><td>29</td><td>28</td><td>001c</td><td>1.635911864</td><td>D165</td></tr><tr><td>30</td><td>29</td><td>001d</td><td>1.380840823</td><td>B0BF</td></tr><tr><td>31</td><td>30</td><td>001e</td><td>1.172037271</td><td>9605</td></tr><tr><td>32</td><td>31</td><td>001f</td><td>1.000000000</td><td>8000</td></tr></table></td></tr>
<tr><td>4-3</td><td>Output Format</td><td>00 = Two’s Complement
01 = Offset Binary
10 = Sign Magnitude
11 = Single Precision Floating Point Format</td></tr>
</table>

TABLE 8. HDF/OUTPUT REGISTER (Continued)

DESTINATION ADDRESS = 5		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
2-1	Number Of Output Bits	00 = 16 Bits 01 = 24 Bits 10 = 32 Bits 11 = 38 Bits
0	Output Sense	0 = LSB First 1 = MSB First

TABLE 9. INPUT AND OUTPUT FORMAT REGISTER

DESTINATION ADDRESS = 6		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	110 = Control Word 6
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35	I followed by Q	0 = I and Q Output Separately 1 = I and Q Data Output on I Pin
34-29	Time Slot Number	Bits 34-29 = $2^5...2^0$. Range: 0 < Time Slot Number < 63. This implies that 64 different channels may be mutliplexed, assigning one time slot per channel.
28	IQCLK Polarity	0 = Output Data Stable On Rising Edge Of IQCLK; IQCLK High between I or Q Bit Periods when IQCLK Duration = 0. 1 = Output Data Stable on Falling Edge of IQCLK; IQCLK Low between I or Q Bit Periods when IQCLK Duration = 0.
27	IQCLK Duty Cycle	0 = IQCLK Active Time = CLK Period. 1 = 50% Duty Cycle
26	IQCLK Duration	0 = Active During I or Q Output Periods Only 1 = Active Continuously
25-24	IQCLK Three-State Control	00 = Three-State IQCLK 01 = Enable IQCLK 1x = Auto-Three-State Enable IQCLK (during time slot)
23	IQSTB Polarity	0 = Active High 1 = Active Low
22	IQSTB Location	0 = IQSTB Prior to the Beginning of the Data Word. 1 = IQSTB During the Data Word.
21-20	IQSTB Three-State Control	00 = Three-State IQSTB 01 = Enable IQSTB 1x = Auto Three-State Enable IQSTB (during time slot)
19	I Polarity	0 = True Data 1 = Inverted Data
18-17	I Three-State Control	00 = Three-State I 01 = Enable I 1x = Auto Three-State Enable I (during time slot)
16	Q Polarity	0 = True Data 1 = Inverted Data

TABLE 9. INPUT AND OUTPUT FORMAT REGISTER (Continued)

DESTINATION ADDRESS = 6		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
15-14	Q Three-State Control	00 = Three-State Q 01 = Enable Q 1x = Auto Three-State Enable Q (during time slot)
13	Input Format	0 = Offset Binary 1 = Two's Complement
12-0	IQCLK Rate Counter Preload	I/QCLK Rate Counter Preload, Bits 12-0 = $2^{12} \cdots 2^{0}$. Range: 2 ≤ IQCLK Rate Counter Preload ≤ 1701. To calculate the value in this field use this equation: **IQCLK Rate Counter Preload = [FLOOR[(HDF Decimation Factor x 4)/TSL] - 1]hex**; where FLOOR(x) represents the integer part of x, and TSL is the decimal value of Control Word 3, bit 31-18.

TABLE 10. PHASE OFFSET REGISTER

DESTINATION ADDRESS = 7		
BIT POSITION	**FUNCTION**	**DESCRIPTION**
39-37	Address	111 = Control Word 7
36	Update	0 = Update Only This Control Register 1 = Update All Control Registers
35-14	Reserved	All Zeroes
13	Data	0 = Normal Data Input 1 = Force Input Data to 8000 Hex.
12-11	FIR Accumulator Control	00 = Normal Accumulation - The accumulator is reset on every FIR cycle. 01 = No Accumulation -The accumulator is disabled. 10 = Continuous Accumulation -The accumulator is not reset on every FIR cycle. This test mode was created to allow the user to perform the equivilent of a check sum test. A very long term test could be run and an accumulated output would yeild a specific numeric value. If the answer differed, the part is not functioning properly. 11 = Reserved
10	Q Strobe on Roll Over	0 = Q carries Normal Data 1 = Q Strobes When Phase Generator Rolls Over
9	Force Outputs	0 = Normal Output Response 1 = Force Outputs
8	IQCLK Forced Data	If Bit 9 = 1, Force IQCLK = Bit 8; Else Normal
7	IQSTB Forced Data	If Bit 9 = 1, Force IQSTB = Bit 7; Else Normal.
6	I Forced Data	If Bit 9 = 1, Force I = Bit 6; Else Normal.
5	Q Forced Data	If Bit 9 = 1, Force Q = Bit 5; Else Normal.
4	Sin/Cos Generator Bypass	0 = Sin Cos Generator Normal, 1 = Bypass Sin Cos Generator; Sin = Cos = 0.fffff (approximately 1)
3	Scaling Multiplier Bypass	0 = Scaling Multiplier Normal, 1 = Scale Factor = 1.
2	Reserved	Must be Zero for Proper Operation while Test Features are Enabled.
1	Wait For RAM Full	If Bit = 0, DDC will Output Data Normally after a Reset, which will Include Unpredictable Data in Data RAMs. If Bit = 1, No Chip Output will Occur until Sufficient Data RAM Locations are Written.
0	Disable Overflow Protection	0 = Normal Operation 1 = Disable Overflow Protection

HSP50016 supports two types of testing. Control Word 7 can be used to verify the operation of the circuit through the divide and conquer method. Setting the Enable Test Bit (Control Word 1, Bit 3) equal to a 1 enables the test features controlled by Control Word 7. (This bit is in Control Word 1 so that Word 7 does not have to be loaded if the test features are not being used.) The functions allowed by Control Word 7 are shown in Table 10.

NOTE: Asserting bits 9 and 13 of Control Word 7 will put all outputs to a static mode. This may remove strobe enables or clocks used to read the data signals. This Test Mode was intended for interface evaluation at the board level.

The DDC also has a Test Access Port (TAP). This port is fully conformant to IEEE Std. 1149.1 - 1990 - IEEE Standard Test Access Port and Boundary-Scan [2]. The TAP supports the following instructions: BYPASS, SAMPLE/PRELOAD, INTEST, EXTEST, RUNBIST and IDCODE. In addition, there are seven instructions called RDCNTLWD1-7, which read the contents of the control words over the TAP. The address bits and bit 36 are only used to determine the destination of data during loading; they are not stored, so they are not read out with the RDCNTLWD1-7 instruction.

Summary

To use the DDC in a down conversion application three items must be considered and designed to compliance. Solutions must satisfy all three items.

1. The Nyquist Sampling Rate for the bandwidth of interest $F_S \geq 2BW$, where BW is the bandwidth of interest.
2. The composite FIR/HDF double sided bandwidth, $BW_{-3dB} = 0.1375F_S/R$.
3. The desired serial output clock rate (total decimation, plus parallel to serial conversion rate increase).

$$\text{IQCLK Frequency} = \left\lceil \frac{f_S \ \text{Length}_{MIN}}{(R_{FIR})\ (R)} \right\rceil - 1$$

NOTE: R_{FIR} = 2 for real mode, 4 for all other modes.

Applications

Down Conversion

The primary spectral operation in the DDC is down conversion of an input signal to base band, see Figure 15. This process is done in two steps: multiplication of the input waveform by an internally generated quadrature sinusoid, i.e., modulation and lowpass filtering to attenuate the unwanted spectral components. The unwanted spectral components have two sources, the input signal and an artifact of the modulation process.

The modulation process can be written as:

$$u(n) = x(n)e^{-j\omega c} = x(n)[\cos(\omega_C) - j\sin(\omega_C)] \quad \text{(EQ. 17)}$$

Where x(n) is the real input data sequence, $\omega = 2\pi f$, and ω_C is the frequency of the signal generated by the SIN/COS Generator.

For demonstration purposes let $x(n) = \cos(\omega_k n)$. The multiplication then becomes:

$$\begin{aligned} u(n) &= \cos(\omega_k n)[\cos(\omega_c n) - j\sin(\omega_c n)] \\ &= 1/2[\cos((\omega_k - \omega_c)n) + \cos((\omega_k + \omega_c)n) \\ &\quad - j(\sin((\omega_k + \omega_c)n) - \sin((\omega_k - \omega_c)n))] \end{aligned} \quad \text{(EQ. 18)}$$

The signal u(n) is passed through a low pass filter; assuming that the filter passes the low frequency terms with no degradation and attenuates the high frequency terms completely, the filtering operation produces the output:

$$\begin{aligned} v(n) &= 1/2(\cos((\omega_k - \omega_c)n) + j\sin((\omega_k - \omega_c)n)) \\ &= 1/2e^{j(\omega_k - \omega_c)n} \end{aligned} \quad \text{(EQ. 19)}$$

When the magnitude of the input signal x(n) is one, the magnitude of v(n) is 1/2. Both the I and Q channels are multiplied by a factor of two to yield:

$$\begin{aligned} w(n) &= \cos((\omega_k - \omega_c)n) + j\sin((\omega_k - \omega_c)n) \\ &= e^{j(\omega_k - \omega_c)n}. \end{aligned} \quad \text{(EQ. 20)}$$

Figure 16 shows an HSP50016 in a single channel down conversion circuit. Notice that the input data is only 12 bits, so it is justified to the MSB of the DDC's input data. If a smaller sample width is used, it is recommended that the MSB of the data is input into DATA15. The unused bits are connected to ground. This alignment makes it easier to locate the position of the MSB in the output data. Note that the input is configured for offset binary arithmetic and the output is set up for I followed by Q, which enables the use of only one serial connection to the output processor. The serial data clock of the processor and the Control Clock of the DDC are driven by a TTL compatible oscillator. (IQCLK cannot be used for this purpose since its frequency is indeterminate until the DDC has been configured). Note that many processors provide a bit clock which eliminates the need for the external oscillator.

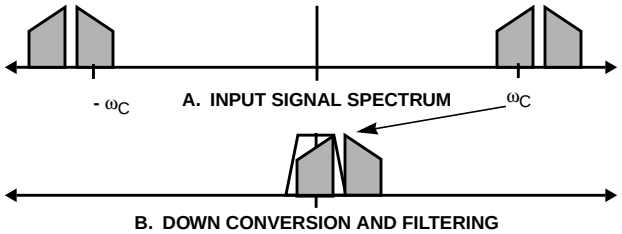


FIGURE 15. DOWN CONVERSION

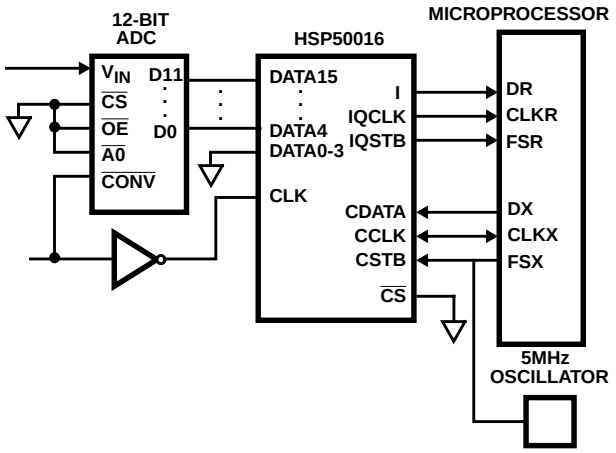


FIGURE 16. CIRCUIT FOR SINGLE CHANNEL OPERATION

An example of the control word contents for this mode of operation is given in Tables 11 through 17. In this setup, the DDC has been configured for a constant down conversion frequency, decimation by 64 and Test Features disabled. Bit fields of three bits or less are in binary notation; longer fields are in hexadecimal. Control Words Zero and Seven are not used.

TABLE 11. SAMPLE FORMAT FOR CONTROL WORD 1 - PHASE GENERATOR/TEST ENABLE

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	001 = Control Word 1.
36	Update	1 = Control Register Update.
35-4	Minimum Phase Increment	Minimum Phase Increment Computed according to $[f_C/f_S]\ 2^{33}$.
3	Test Enable	0 = Test Features Disabled.
2-0	Phase Generator Mode	001 = Normal Mode.

0011 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX 0001

TABLE 12. SAMPLE FORMAT FOR CONTROL WORD 2 - PHASE GENERATOR

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	010 = Control Word 2.
36	Update	1 = Control Register Update.
35-32	Reserved	All Zeroes.
31-0	Maximum Phase Increment	All Zeroes.

0101 0000 0000 0000 0000 0000 0000 0000 0000 0000

TABLE 13. SAMPLE FORMAT FOR CONTROL WORD 3 - PHASE GENERATOR/OUTPUT TIME SLOT

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	011 = Control Word 3.
36	Update	1 = Control Register Update.
35-32	Reserved	All Zeroes.
31-18	Time Slot Length	All Zeroes.
17-0	Phase Offset	All Zeroes.

0111 0000 0000 0000 0000 0000 0000 0000 0000 0000

TABLE 14. SAMPLE FORMAT FOR CONTROL WORD 4 - PHASE GENERATOR/HDF/OUTPUT

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	100 = Control Word 4.
36	Update	1 = Control Register Update.
35-33	Reserved	All Zeroes.
32	Up Convert	0 = Do Not Up convert.
31	Real Mode	0 = Complex Mode.
30-7	Delta Phase Increment	All Zeroes.
6-1	Shift	37 = Decimal 55, the Shift Corresponding to HDF Decimation by 16.
0	Spectral Reverse	0 = No Spectral Reversal

1001 0000 0000 0000 0000 0000 0000 0000 0110 1110

TABLE 15. SAMPLE FORMAT FOR CONTROL WORD 5 HDF/OUTPUT

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	101 = Control Word 5.
36	Update	1 = Control Register Update.
35-21	HDF Decimation Counter Preload	F = Decimation by 16 in HDF.
20-5	Scaling Multiplier Gain	8000 = Scaling Multiplier Gain of 1.
4-3	Output Format	00 = Two's Complement.
2-1	Number of Output Bits	00 = 16-Bits.
0	Output Sense	1 = MSB First.

1011 0000 0000 0001 1111 0000 0000 0000 0000 0000

TABLE 16. SAMPLE FORMAT FOR CONTROL WORD 6 - OUTPUT

BIT POSITION	FUNCTION	DESCRIPTION
39-37	Address	110 = Control Word 6.
36	Update	1 = Control Register Update.
35	I followed by Q	1 = I and Q Data Output on I Pin.
34-29	Time Slot	Time Slot Number = 0.
28	IQCLK Polarity	0 = Data Stable on Rising Edge of IQCLK.
27	IQCLK Duty Cycle	1 = IQCLK Duty Cycle is 50%.
26	IQCLK Duration	1 = Active Continuously.
25-24	IQCLK Three-State Control	01 = Enable IQCLK.
23	IQSTB Polarity	0 = IQSTB Active High.
22	IQSTB Location	0 = IQSTB Active Prior to the Beginning of the Data Word.
21-20	IQSTB Three-State	01 = Enable IQSTB.
19	I Polarity	0 = I Output Active High.
18-17	I Three-State Control	01 = Enable I.
16	Q Polarity	0 = Q Output Active High.
15-14	Q Three-State Control	00 = Disable Q.
13	Input Format	1 = Two's complement.
12-0	IQCLK Rate	All Zeroes = CLK Used to Clock Output Bits.

1101 1000 0000 1101 0001 0010 001X XXXX XXXX XXXX

TABLE 17. SUMMARY OF CONTROL WORDS FOR THE EXAMPLE

CONTROL WORD	HEX VALUE
0	0 0 0 0 0 0 0 0 0 0
1	3 X X X X X X X X 1
2	5 0 0 0 0 0 0 0 0 0
3	7 0 0 0 0 0 0 0 0 0
4	9 0 0 0 0 0 0 0 6 E
5	B 0 0 1 F 0 0 0 0 0
6	D 8 0 D 1 2 X X X X
7	0 0 0 0 0 0 0 0 0 0

Quadrature To Real Conversion

After the input data has been processed by the DDC, the output can be converted into a real signal if desired. In that case, the baseband centered quadrature signal is upcobaseband. The real part of the upconverted signal is taken as the output. To satisfy the Nyquist criteria, the sample rate of the resulting signal must be at least twice the minimum sample rate of the I and Q components of the quadrature signal. This prevents one sideband from aliasing onto the other sideband when the real part of the output signal is taken.

The spectrum of a quadrature signal which has been over sampled by 2 is shown in Figure 17A. This represents the output of the filters. As described in the previous paragraph, the oversampling is a necessary feature of this process, since the final signal will occupy twice the bandwidth of the filter output. To prevent aliasing upon taking the real part of the signal, it is necessary to perform an up conversion by f"/4, where f" is the decimated sample frequency. (Note that f_S is defined as the input sampling frequency, f' is the input sampling frequency divided by the HDF decimation rate R, and f" is f' divided by the FIR decimation rate. f" is the FIR output sampling rate). The up conversion function is:

$$e^{j2\pi nf"/4f"} = e^{j\pi n/2} \quad \text{(EQ. 21)}$$

For n = 0, 1, 2, 3, 4,... the output values of the local oscillator in rectangular representation are: 1 + 0j, 0 + j, -1 + 0j, 0 - j, 1 + 0j,.... Since the real half of the complex multiplication of the local oscillator values by the filtered signal values (the desired output is the real part of the product) require only trivial operations, this up conversion is done in the Formatter. Figure 17B shows the signal spectrum after up conversion. Figure 17C shows the spectrum of the real output signal.

Continuing with the single tone example from the previous section, the quadrature signal output from the FIR filters is:

$$w(n) = \cos((\omega_k - \omega_c)n) + j\sin((\omega_k - \omega_c)n) = e^{j(\omega_k - \omega_c)n} \quad \text{(EQ. 22)}$$

Multiplying w(n) by the up convert function and summing the result is equivalent to the output sequence:

$$\begin{aligned} &y(n) = 1 \times \cos((\omega_k - \omega_c)n), \\ &y(n+1) = j \times j\sin((\omega_k - \omega_c)(n+1)), \\ &y(n+2) = -1 \times \cos((\omega_k - \omega_c)(n+2)), \\ &y(n+3) = -j \times j\sin((\omega_k - \omega_c)(n+3)), \\ &y(n+4) = 1 \times \cos((\omega_k - \omega_c)(n+4)),... \end{aligned} \quad \text{(EQ. 23)}$$

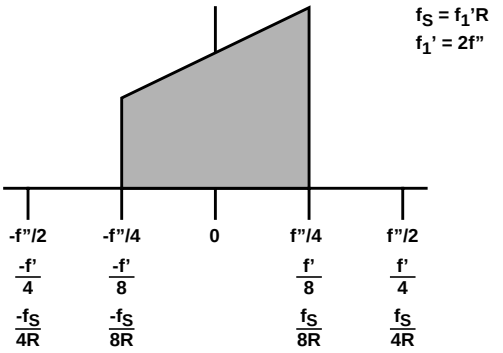
$$y = \cos((\omega_k - \omega_c)n), -\sin((\omega_k - \omega_c)(n+1)), -\cos((\omega_k - \omega_c)(n+2)), \sin((\omega_k - \omega_c)(n+3)), \cos((\omega_k - \omega_c)(n+4)),...$$

Or:

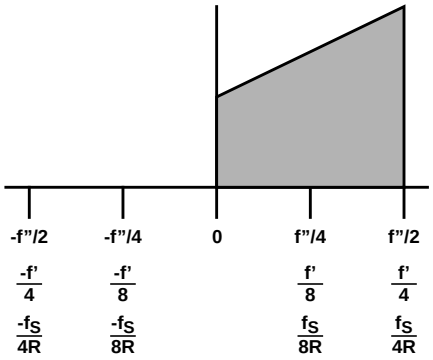
$$y = RE(w(n)), - IM(w(n+1)), - RE(w(n+2)), IM(w(n+3)), RE(w(n+4)),...$$

Since $|e^{j\pi n/2}| = 1$ and |w(n)| = 1, no further magnitude corrections are required.

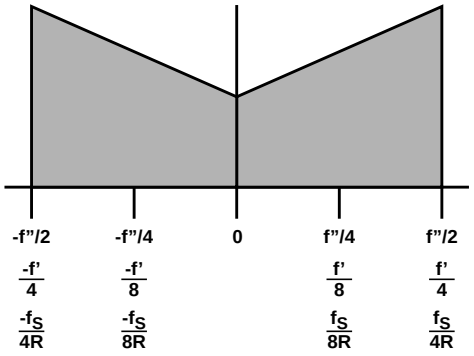
The setup for this application is similar to that of the down conversion circuit given above, except the Output Formatter is set for Real Mode (Bit 31 in Control Word 4). This bit configures the part for up conversion by f"/4 and summing of the real and imaginary parts of the filter output.



A. OUTPUT OF FIR FILTERS: SIGNAL OVERSAMPLED BY 2



B. FIRST OPERATION IN FORMATTER: UP CONVERT BY SAMPLE FREQUENCY / 4



C. SECOND OPERATION IN FORMATTER: I OUTPUT = I + Q

FIGURE 17. QUADRATURE TO REAL CONVERSION OF AN OUTPUT SIGNAL

Up Conversion by f"/2

This operation allows the user to exchange the positions of the upper and lower halves of a down converted signal while leaving each half unchanged. Quadrature up conversion by f"/2 is performed by multiplying the output signal by $e^{j2\pi nf"/2} = \cos(2pnf"/2) + j \sin(2pnf"/2)$. When sampled at a rate of f", cos(2pnf"/2) takes on the values 1, -1, 1, -1,... and sin(2pnf"/2) always = 0. Thus, the up convert LO sequence is:

$$e^{j\pi n} = 1 + j0, -1 + j0, 1 + j0, \ldots \quad \text{(EQ. 24)}$$

This sequence is multiplied by the output of the I and Q branches of the filter:

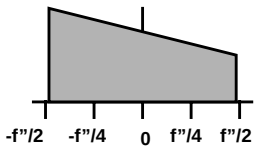
$$w(n) = \cos((\omega_k - \omega_c)n) + j\sin((\omega_k - \omega_c)n) = e^{j(\omega_k - \omega_c)n}, \quad \text{(EQ. 25)}$$

yielding an output sequence:

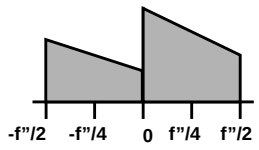
$$y = (RE(w(n)), IM(w(n))), \quad \text{(EQ. 26)}$$
$$-RE(w(n+1)), - IM(w(n+1))),$$
$$RE(w(n+2)), IM(w(n+2))), \ldots$$

The Formatter contains the circuitry to shift the quadrature output spectrum up by one half of the output sample frequency f". This operation is independent of the function of the Phase Generator and Mixer. The spectra of the outputs of the Filter and Formatter are shown in Figure 18.

The setup is identical to the down conversion configuration, except that the Up Convert Bit is set in Control Word 4.



A. OUTPUT OF FIR FILTERS



B. FILTER OUTPUT UP CONVERTED BY OUTPUT SAMPLE FREQUENCY / 2

FIGURE 18. UP CONVERSION BY F" / 2

Quadrature Spectral Reversal

Spectral reversal is often used to negate a spectral reversal which has occurred due to a previous operation in the processing chain. Examples of this are spectral reversal in an analog down conversion or in a constructive aliasing operation. The DDC gives the user the ability to convert the signal to baseband in either forward or reverse fashion. Quadrature spectral reversal is achieved by translating the lower sideband of the input to baseband rather than the upper sideband. This is implemented in the DDC by mixing the input signal with $e^{j2\pi f_c n}$ - that is, up converting the input rather than down converting it. The resulting signal is:

$$u(n)= x(n)e^{j2\pi f_c n} = x(n)[\cos(\omega_c n) + j\sin(\omega_c n)] \quad \text{(EQ. 27)}$$

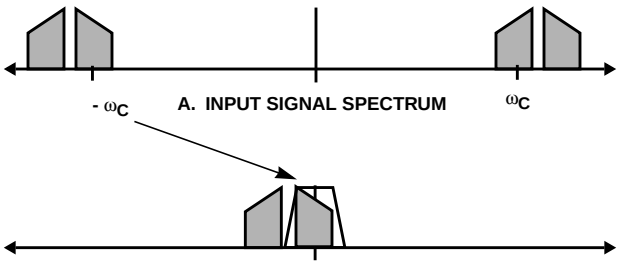
Assuming $x(n) = \cos(\omega_k n)$,

$$\begin{aligned} u(n) &= \cos(\omega_k n)[\cos(\omega_c n) + j\sin(\omega_c n)] \quad \text{(EQ. 28)} \\ &= [\cos((\omega_k - \omega_c)n) + \cos((\omega_k + \omega_c)n) \\ &\quad + j(\sin((\omega_k + \omega_c)n) - \sin((\omega_k - \omega_c)n))] \end{aligned}$$

After quadrature filtering and correcting for the gain of 1/2, we have:

$$\begin{aligned} w(n) &= \cos((\omega_k - \omega_c)n) - j\sin((\omega_k - \omega_c)n) \quad \text{(EQ. 29)} \\ &= \cos(-(\omega_k - \omega_c)n) + j\sin(-(\omega_k - \omega_c)n) \\ &= \cos((\omega_c - \omega_k)n) + j\sin((\omega_c - \omega_k)n) \\ &= e^{j(\omega_c - \omega_k)n} \end{aligned}$$

The appropriate spectral plots are shown in Figure 19. In up conversion, the sine output of the SIN/COS Generator is negated so that the vector output of the Local Oscillator rotates counter clockwise. This is implemented by setting the Spectral Reverse bit in Control Word 4 to a one. Otherwise, the setup for this mode is the same as the one for down conversion.

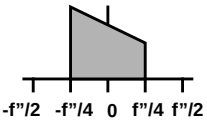


B. UP CONVERSION AND FILTERING

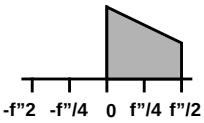
FIGURE 19. UP CONVERSION OF FILTER OUTPUT SIGNAL

Real Spectral Reversal

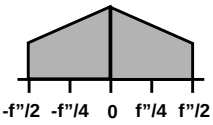
Real spectral reversal is simply quadrature spectral reversal with quadrature to real conversion in the Formatter. The up converted and filtered signal w(n) is upconverted again by f”/4 in the Formatter. Each sideband of the result is spectrally reversed from the sidebands that would have been produced by down conversion with quadrature to real conversion. The output spectrum is shown in Figure 20.



A. OUTPUT OF FILTERS: SIGNAL OVERSAMPLED BY 2



B. FIRST OPERATION IN FORMATTER: UP CONVERT BY SAMPLE FREQUENCY / 4



C. SECOND OPERATION IN FORMATTER: I OUTPUT = I + Q

FIGURE 20. QUADRATURE TO REAL CONVERSION OF AN OUTPUT SIGNAL

The setup for this application is similar to that of down conversion, except in Control Word 4, where the Spectral Reverse and Real Output bits are set to one.

High Decimation Filter Only

The DDC can be operated as a single high decimation filter. This is done by setting the Phase Generator to Filter Only and the Minimum Phase Increment and Phase Offset to 0. This multiplies the incoming data stream by a constant hexadecimal 3FFFF in the I channel and 0 in the Q channel. The HDF Section of the circuit requires a minimum decimation rate of 16 to allow sufficient time for the FIR to compute its response. This mode of operation implements a filter which has a decimation rate from 64 to 131,072. The frequency response is shown in Figures 7, 8 and 9. Only the I output has valid data in this mode; the Q output should be set to high impedance state to reduce circuit noise.

Multichannel Operation

Several DDCs can be placed in parallel with each one operating on a different frequency band. To minimize wiring, their outputs can be configured so that they are connected over a common serial bus. Each DDC is assigned a time slot number (Control Word 6) and a time slot length (Control Word 3). Each DDC in turn controls the bus for long enough to output its data, then relinquishes the bus. The time slot assignment and length are programmed at configuration time. Up to 64 chips can be multiplexed in this manner.

Figure 21 shows a Block Diagram of this configuration. The DDCs are configured by the microprocessor by first writing a logical 0 to its Chip Select line. The control words are written to that part in any order. When the part has been configured, $\overline{CS}$ is written high again, and the next part is configured in the same manner. Collisions are prevented by programming each DDC with a unique Time Slot number, which holds its output from 0 to 63 output word times before transmission. Each part also has a Time Slot Length, whose minimum value is given in Equation 8. Note that a value greater than the minimum can be used to give the processor time to operate on the data

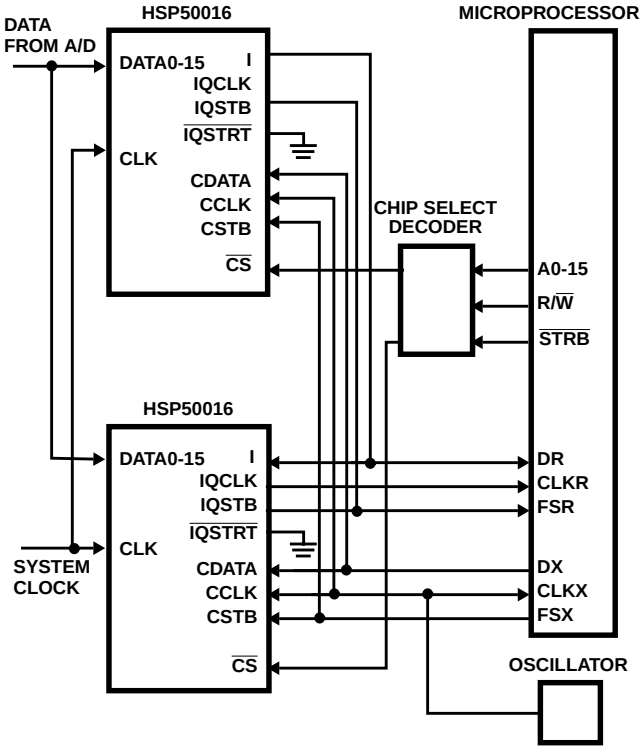


FIGURE 21. CIRCUIT FOR MULTIPLE CHANNEL OPERATION (AUTO THREE-STATE)

The corresponding Configuration Register setup is similar to that of single channel down conversion, except for the Auto Three-State fields. In this example, the first DDC in the chain is set to drive IQCLK; the others have this output set for high impedance. (It makes no difference which DDC is chosen to be the one to drive IQCLK, but it must be active continuously). The unused outputs are put in their high impedance condition on the other DDCs to minimize power consumption. Note also that this example shows all DDCs in I followed by Q Mode so that only one data line to the microprocessor is necessary. Figure 22 gives the timing of the output data.

When operating a set of HSP50016s in the Multiple Channel Operation Mode, the two control signals that ensure proper time slot operation are: $\overline{CS}$ and $\overline{RESET}$. The $\overline{CS}$ allows unique Control Word loading of each DDC. The $\overline{RESET}$ synchronizes all of the DDC's to the start of the first time slot.

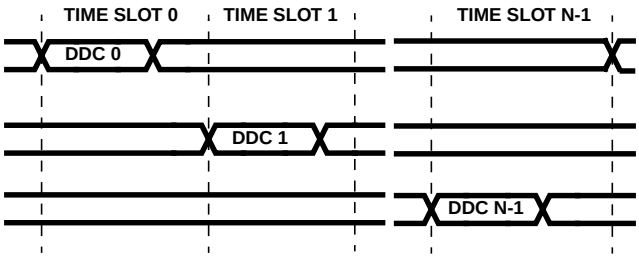


FIGURE 22. TIMING FOR MULTIPLE CHANNEL OPERATION (AUTO THREE-STATE)

NOTE: In this mode, it was anticipated that parallel DDC's would be programmed identically, except for the NCO (L.O.) frequency and Time Slot Number. This implies identical HDF Decimation factors, Time Slot Length's, Number of Output Bits, Output Mode are identical in all DDCs. This means that the output rate of all DDC HDFs, FIRs and Parallel to Serial Converters are identical.

The DDC keeps an internal count of the number of IQCLK periods that have transpired since the rising edge of $\overline{RESET}$. The internal counter of each DDC is set to enable the serial output at the time slot number assigned to that DDC. The count is based on the time slot length, number of output bits, HDF decimation, and Output Mode programmed to that part.

NOTE: In the Multiple Channel Operation Mode, all the time slot lengths should be set to the same value to avoid output signal contention. The time slot length should be equal to the largest "minimum time slot length" as calculated by Equation 15, for every DDC in the multichannel arrangement. Note that Equation 8 is in IQCLK periods, not CLK periods. Equations 7 and 9 will be helpful in making the translation to CLK periods.

This mode does not require that all of the time slots be used (assigned to a DDC). If only two parts were used and the maximum time signal isolation was desired, one could assign the first signal time slot 31 and the second signal time slot 63. This would ensure that the maximum separation in time occurred. It is the designers responsibility to ensure that the output rate, including the decimation is consistent with the time allotted to output each signal.

Let's return to the clock calculation example found in the Output Formatter, and add the requirement of 4 time slots. The calculations in the Example Clock Calculation Section remain true, but step 8 must be added:

8. Now lets consider the multichannel timing. Each channel must output the data at no less than 1.725MHz to get all the I/Q data out in the allocated time for the assigned time slot. Time margin is created when the output is clocked out at a higher rate. Because each channel is outputting data in only one of four channels, the effective output rate for each channel is 1.725MHz /4 = 431.25kHz, even though the part is outputting data 1.7MHz in every time slot.
. . . . . Effective Channel Output Rate = 431.25kHz

Alternatively, the processor can request data from each of the DDCs asynchronously. In this setup, Requested Output Mode is used. The Data Concentrator polls each channel individually and is responsible for ensuring that each channel is polled before the output data is lost. The Data Concentrator is a custom circuit designed by the user. A Block Diagram of such a system is shown in Figure 23. The interface between the controller and the DDCs has been omitted for the sake of clarity.

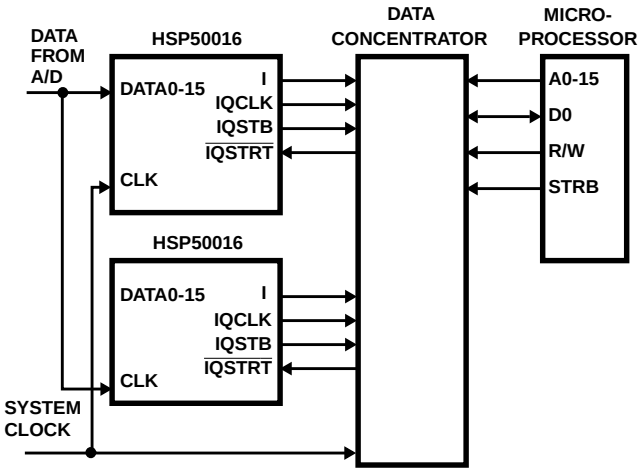


FIGURE 23. CIRCUIT FOR MULTIPLE CHANNEL OPERATION (REQUESTED OUTPUT)

References

[1] Hogenauer, Eugene V., An Economical Class of Digital Filters for Decimation and Interpolation, IEEE Transactions on Acoustics, Speech and Signal Processing, April 1981.

[2] IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1 - 1990.

Absolute Maximum Ratings

Supply Voltage . . . . . +7.0V
Input, Output or I/O Voltage . . . . . GND -0.5V to V_{CC} +0.5V
ESD Classification . . . . . Class 1

Operating Conditions

Operating Voltage Range . . . . . +4.75V to +5.25V
Operating Temperature Range . . . . . 0^oC to 70^oC

Thermal Information

Thermal Resistance (Typical, Note 1)	θ_{JA} ($^oC/W$)	θ_{JC} ($^oC/W$)
CPGA Package	45	7
PLCC Package	35	N/A

Maximum Junction Temperature
CPGA Package . . . . . 150^oC
PLCC Package . . . . . 150^oC
Maximum Storage Temperature Range . . . . . -65^oC to 150^oC
Maximum Lead Temperature (Soldering 10s) . . . . . 300^oC

Die Characteristics

Gate Count . . . . . 65,000 Gates

CAUTION: Stresses above those listed in "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress only rating and operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.

NOTE:

1. θ_{JA} is measured with the component mounted on an evaluation PC board in free air.

DC Electrical Specifications

V_{CC} = 5.0V ±5%, T_A = 0^o to 70^oC

PARAMETER	SYMBOL	TEST CONDITIONS	MIN	MAX	UNITS
Power Supply Current	I_{CCOP}	V_{CC} = Max, CLK Frequency 52.6MHz Notes 2, 3	-	394	mA
		V_{CC} = Max, CLK Frequency 76.9MHz Notes 2, 3	-	577	mA
Standby Power Supply Current	I_{CCSB}	V_{CC} = Max, Outputs Not Loaded	-	500	µA
Input Leakage Current	I_I	V_{CC} = Max, Input = 0V or V_{CC} TMS, TDI, $\overline{TRST}$	-500	10	µA
		V_{CC} = Max, Input = 0V or V_{CC} All other inputs	-10	10	µA
Output Leakage Current	I_O	V_{CC} = Max, Input = 0V or V_{CC}	-10	10	µA
Logical One Input Voltage	V_{IH}	V_{CC} = Max	2.0	-	V
Logical Zero Input Voltage	V_{IL}	V_{CC} = Min	-	0.8	V
Logical One Input Voltage: CLK, $\overline{TRST}$	V_{IHC}	V_{CC} = Max	3.0	-	V
Logical One Output Voltage	V_{OH}	I_{OH} = -5mA, V_{CC} = Min	2.6	-	V
Logical Zero Output Voltage	V_{OL}	I_{OL} = 5mA, V_{CC} = Min	-	0.4	V
Input Capacitance	C_{IN}	CLK Frequency 1MHz All measurements referenced to GND. T_A = 25^oC, Note 4	-	10	pF
Output Capacitance	C_{OUT}		-	10	pF

NOTES:

2. Power supply current is proportional to frequency. Typical rating is 7.5mA/MHz. Note that operation at maximum clock frequency will exceed maximum junction temperature of device. Use of a heat sink and/or air flow is required under these conditions: Recommended heat sink is EG&G Wakefield D10650-40.
3. Output load per test circuit and C_L = 40pF.
4. Not tested, but characterized at initial design and at major process/design changes.

AC Electrical Specifications

V_{CC} = 5.0V ±5%, T_A = 0^o to 70^oC, (Note 5)

PARAMETER	SYMBOL	NOTES	-52(52.6)MHz		-75(76.9)MHz		UNITS
			MIN	MAX	MIN	MAX	
CLK Period	t_{CP}		19	-	13	-	ns
CLK High	t_{CH}		7	-	5	-	ns
CLK Low	t_{CL}		7	-	5	-	ns
Setup Time DATA0-15 to CLK	t_{DS}		10	-	7	-	ns
Hold Time DATA0-15 from CLK	t_{DH}		1	-	1	-	ns
$\overline{RESET}$ Pulse Width	t_{RL}		t_{CP}+11	-	t_{CP}+8	-	ns

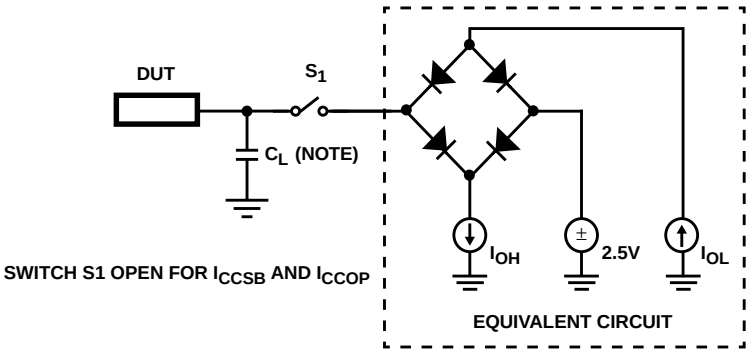
AC Electrical Specifications V_{CC} = 5.0V ±5%, T_A = 0° to 70°C, (Note 5) **(Continued)**

PARAMETER	SYMBOL	NOTES	-52(52.6)MHz		-75(76.9)MHz		UNITS
			MIN	MAX	MIN	MAX	
$\overline{RESET}$, $\overline{IQSTRT}$ Setup Time from CLK	t_{RS}	Note 6	10	-	7	-	ns
RESET, $\overline{IQSTRT}$ Hold Time to CLK	t_{RH}	Note 6	1	-	1	-	ns
CLK to I, Q, IQSTB, IQCLK Delay	t_{DO}		-	15	-	12	ns
CCLK Period	t_{CCP}		100	-	100	-	ns
CCLK High	t_{CCH}		40	-	40	-	ns
CCLK Low	t_{CCL}		40	-	40	-	ns
CDATA, CSTB, $\overline{CS}$ Setup to CCLK	t_{CDS}		30	-	30	-	ns
CDATA, CSTB, $\overline{CS}$ Hold from CCLK	t_{CDH}		30	-	30	-	ns
CCLK Low Setup to CLK	t_{CLS}	Notes 6, 7	30	-	30	-	ns
CCLK High Hold from CLK	t_{CHH}	Notes 6, 7, 10	30	-	30	-	ns
TCK Period	t_{TCP}	Note 8	100	-	100	-	ns
TCK High	t_{TH}		40	-	40	-	ns
TCK Low	t_{TL}		40	-	40	-	ns
$\overline{TRST}$ Pulse Width	t_{TRL}		100	-	100	-	ns
TCK to TDO, Data Delay	t_{TDO}		-	30	-	30	ns
Setup Time On All Inputs to TCK	t_{ATS}	Note 9	30	-	30	-	ns
Hold Time On All Inputs from TCK	t_{ATH}	Note 9	30	-	30	-	ns
TCK Setup Time to CLK	t_{TCS}	Note 8	30	-	30	-	ns
TCK Hold Time from CLK	t_{TCH}	Note 8	30	-	30	-	ns
Output Enable Time from CLK	t_{OE}	Note 10	-	18	-	12	ns
Output Disable Time from CLK	t_{OD}	Note 10	-	18	-	12	ns
Output Enable Time from TCK	t_{TOE}	Note 10	-	32	-	32	ns
Output Disable Time from TCK	t_{TOD}	Note 10	-	32	-	32	ns
Output Rise, Fall Time	t_{RF}	Note 10	-	5	-	5	ns

NOTES:

5. AC tests performed with C_L= 40pF, I_{OL} = 5mA, and I_{OH} = -5mA. Input reference level for CLK, $\overline{TRST}$ is 2.0V, all other inputs 1.5V. Test V_{IH} = 3.0V, V_{IHC} = 4.0V, V_{IL} = 0V; V_{OH} = V_{OL} = 2.5V.
6. These are asynchronous inputs; setup and hold times must only be maintained in order to predict which clock cycle they take effect internally.
7. Timing must only be maintained when Update bit is active in control word data being loaded.
8. Special Timing relationship between TCK and CLK is required for Test Instructions RUNBIST, EXTEST and INTEST.
9. All inputs except $\overline{TRST}$, and only when TCK is driving internal clock.
10. Controlled via design or process parameters and not directly tested. Characterized upon initial design and after major process and/or design changes.

AC Test Load Circuit



NOTE: Test head capacitance.

Waveforms

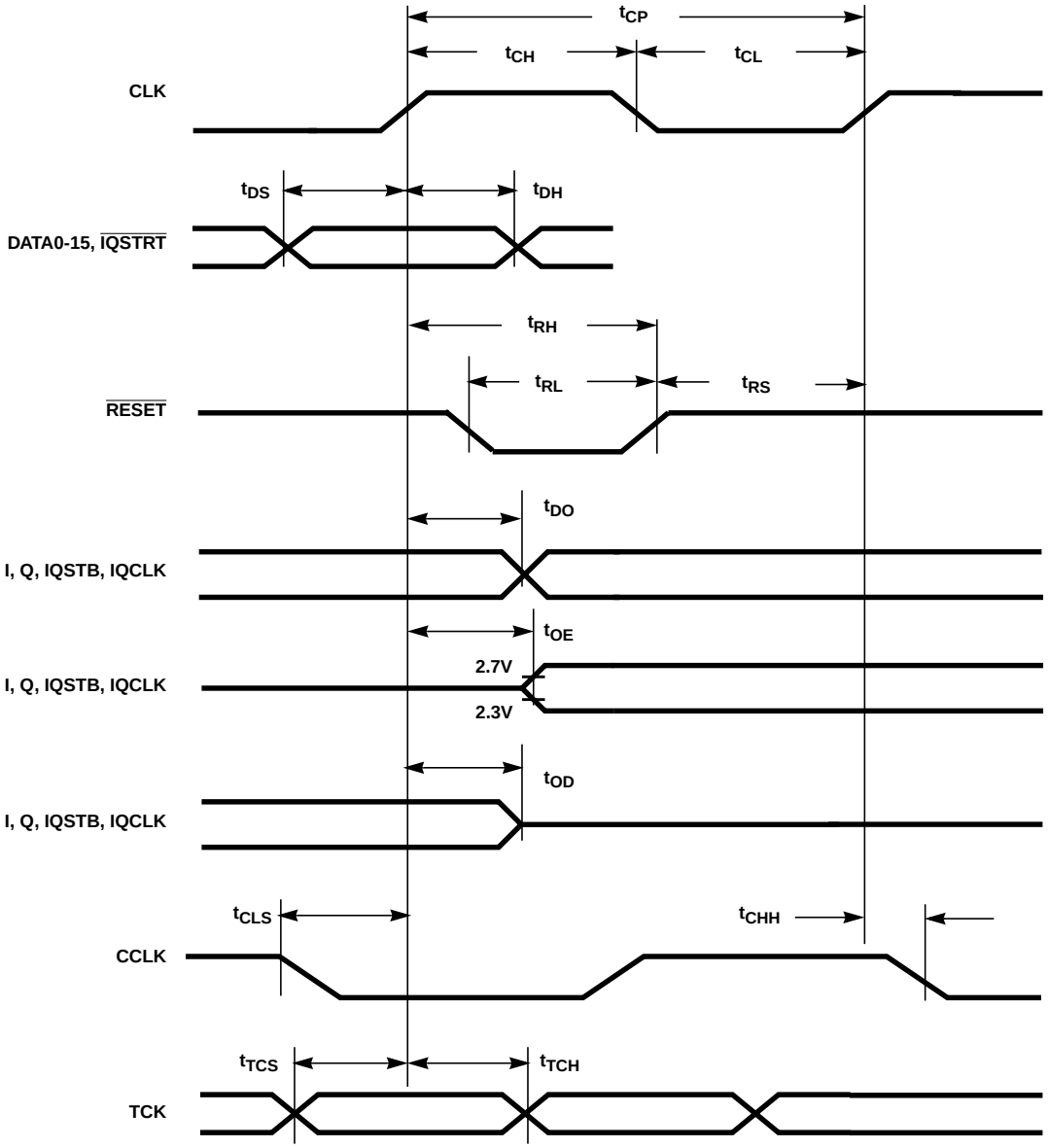


FIGURE 24. TIMING RELATIVE TO CLK

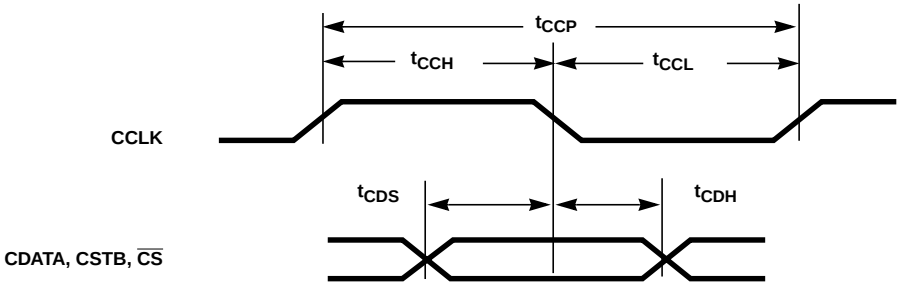


FIGURE 25. TIMING RELATIVE TO CCLK

Waveforms (Continued)

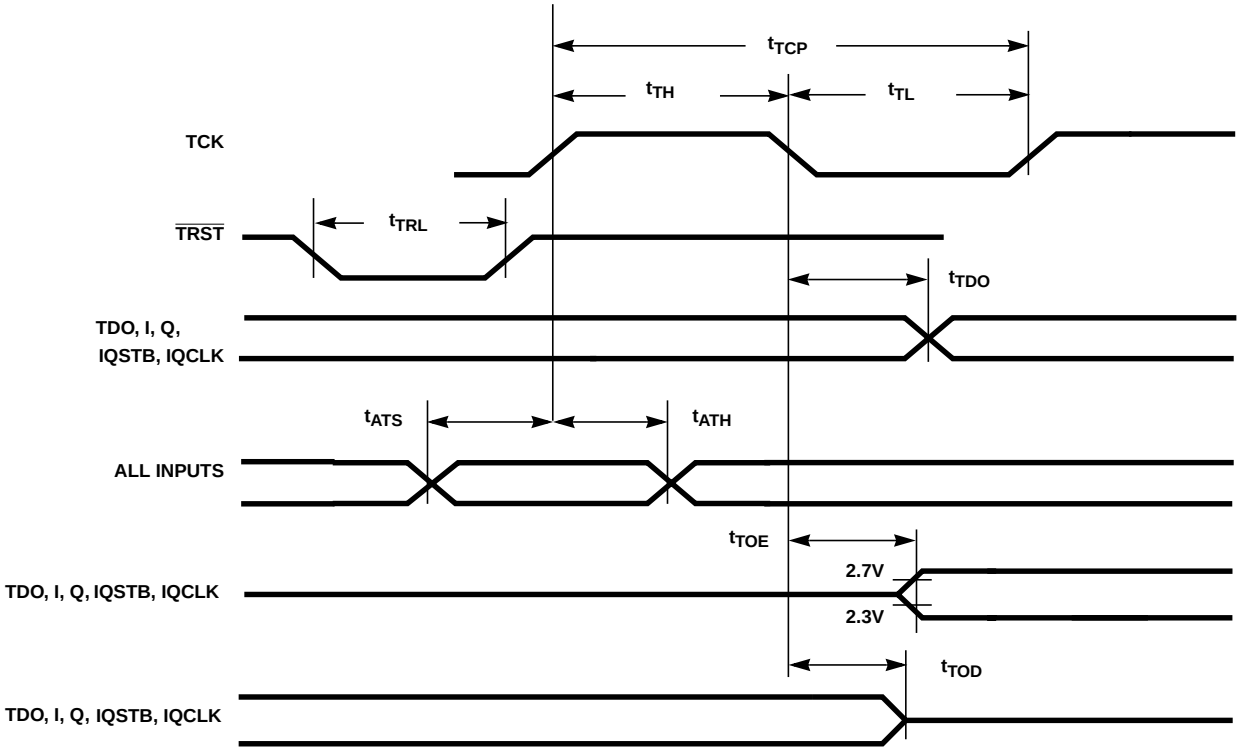


FIGURE 26. TIMING RELATIVE TO TCK

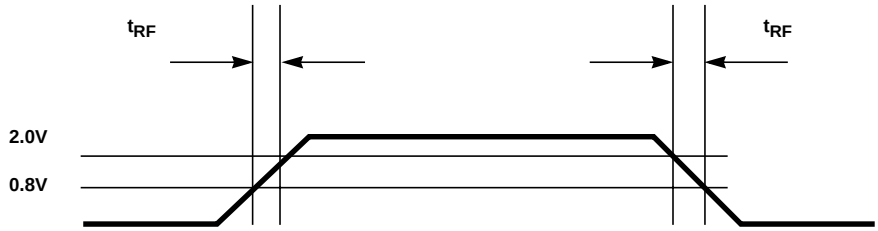


FIGURE 27. OUTPUT RISE AND FALL TIMES

This page is intentionally blank

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP