

Appendix A

Rx-DSP Notes & Code

Documenting the Rx-DSP codes is a bit of a gong show. The code is slightly modified for every deployment, there's kludges nobody even remembers, and in some cases the code is copyright Texas Instruments, so no bueno. Also, assembly code is absurdly long—just the AGO code is over 5,000 lines. The full codebase is available at the group's GitHub repository, <https://github.com/DartmouthSpacePhys>.

The following will, instead, attempt to go over the history of the code, the basic parts of the modern structure, and compilation, and the parts of the code which are Dartmouth-specific and have unique features. It will also cover a couple of ‘quirks’ that have been found in the design.

A.1 Original Rocket Code & Errata

The original code still used on rockets was written by J. C. Vandiver, based on Iowa code for the hardware the Rx-DSPs themselves were based on. It is a single, monolithic code block that contains the serial monitor code, AD6620 RSP support code, and the main data acquisition code. It takes data in 65 kiloword frames, triggered by a Major Frame Interrupt from the telemetry hardware, and is designed to be used in a two-DSP synchronized setup, in order to get polarization data. It synchronizes the major frames the two DSPs are sending by using the FIFO-reset line wired between, and herein lies an issue.

Design Flaw: FIFO Reset

The FIFO Reset line, as designed, has a flaw when you try to use it as above, while the system is ‘live’ (i.e. the RSP is running and the FIFO is filling with data).

The base problem is that no part of the system pays attention to where the RSP is in its data cycle, or what data has been pulled off of it by the FIFO. If you’re watching the RSP output, the duty cycle is such that it spends a large time idle, then sends the I word of a sample, and then the Q word a short—but non-zero—amount of time later. If the FIFO

Reset line is triggered in that tiny time gap between I and Q, the first word loaded into the now-empty FIFO will be a Q.

The system is designed to account for this using the least significant bit in the first few words of data in each block. There is hardware which is toggled on by the c542 which watches the I/Q line on the RSP output, and sets the bit high or low depending on its I or Q state. In the rocket code, this hardware is set on for the first 200 words of each major frame. In theory it should thus be possible to determine the 'I/Q phase' of a major frame by examining these LSBs.

Unfortunately, this LSB-override hardware is unreliable: it has been observed on the bench to yield incorrect results in a small number of cases, for reasons unknown. A possible more-robust option would be to add hardware which holds the FIFO Reset line high, if the I/Q line is in Q state, for as long as it remains so, thus preventing a Q from ever being read in as the first word.

There are two workarounds, one in post-processing, and the other in firmware operation structure. The post-processing workaround requires that a well-known signal be present in the data, e.g. a beacon or interference line. With this the data can be closely perused, and any major frames which appear to be 'bad' (i.e. discontinuous with respect to neighboring frames) can be tagged as such, and processing attempted with the first word dropped.

The firmware workaround is to change the way the RSP is handled. The only state in which you are guaranteed to start with an I is the startup state, so if you send the RSP into reset at the end of each major frame, then start it fresh just before a new frame, the problem is solved...right? Well, sort of. You also have to discard a chunk of data at the very start—the first few words put out are nonsense, and the filters need a while to kick in. For safety, current codes discard the first 512 words out of the FIFO.

An added benefit of the firmware workaround is that it slightly lowers the power consumption, depending on what fraction of time the RSP spends in the idle state.

The rocket firmware is very barebones, simply reading in a major frame, adding a header, and then putting everything on the output FIFO for telemetry hardware to read out.

Design Quirk: Receive FIFO Clocking Possibly because of its history as a University of Iowa instrument, the receive FIFO hardware is designed such that it is directly clocked by the c542 read action—this makes the FIFO status lines unreliable, and requires that the firmware data-read loop manually time its read operations with `nop`.

While perfectly valid, this method ties the c542 up for an inordinate amount of time. All revised deployments of the Rx-DSP have been hardware-modified to have a more traditional setup, with the FIFO clock line receiving the onboard 40 MHz clock signal. This allows an idle state or other work to take place, until the FIFO status lines reach a state which requires/allows for read-out.

The rocket code contains several parts: interrupt vectors, definition of constants, serial monitor code and utilities, the RSP programming code, and the main acquisition program.

Some of these shall be covered in a general sense by the following sections, which review updated versions.

A.2 South Pole Station PF-ARC

The receiver at South Pole Station was the first ground-based deployment of the Rx-DSP, and is very close to a rocket setup, using an essentially unmodified firmware. The unique part of the setup lies on the ‘telemetry’ side: each of the Rx-DSPs has its parallel high-speed output port wired to a Bitwise Systems QuickUSB module, and both modules are connected to a standard PC running Linux.

The QuickUSB is wired to directly trigger major frames, acting similarly to a rocket telemetry system to pull data from the Tx FIFO. The acquisition code was custom written with assistance from Bitwise, and is available on the repository.

A.3 The Antarctic AGO S-ARC

This section covers codes specifically for an S-ARC designed for use at Antarctic Automatic Geophysical Observatory (AGO) sites. This design takes 512-sample bursts of data, then FFTs the data, and transmits the power in decibels through the c542 buffered serial port.

While these codes are purpose-specific, the general forms are the same as for any single-receiver ARC.

A.3.1 Compilation Support/Toolchain

The codes are compiled and linked from `.asm` to `.obj`, and then linked into a single loadable `.hex`, using a suite of Texas Instruments command line programs. Below are three support files for this process: a GNU `Makefile`, two `.cmd` files, which are used by the `hex500.exe` and `lnk500.exe` programs as sources of additional options, and a short bootloader code.

The files below are designed for a two-stage initialization process, which uses a bootloader code for a more recent DSP, the c549. This bootloader allows more advanced memory management—specifically, having the program code split between multiple memory regions—which is required to fit the program codes and lookup tables into RAM, leaving enough room for data storage, FFTing, and transmission.

The lookup tables, which take over 2 kilowords of storage alone, store pre-calculated values of sine functions for the Hann windowing and FFT functions.

Makefile

```
1 MAINSRC ?= ago_v1.0.asm
2 CFPREFIX ?= twostage
```

```

3
4  ASMFLAG = -v542
5  LNKFLAG = -w -a -r
6
7  CGTDIR = /cygdrive/c/TI54xCGT/bin
8
9  BASESRC = int_table.asm bl549.asm ad6620.asm tablemax.asm
10 FUNC SRC = $(BASESRC) Cbrev32.asm c512.asm log_10.asm hannwin.asm sercook.asm
    cfft_32.asm dpsm.asm scale.asm
11 TABLES = hann_q15.tab
12
13 OBJECTS = $(MAINSRC:.asm=.obj) $(FUNCSRC:.asm=.obj) $(TABLES:.tab=.obj)
14 LSTS = $(FUNCSRC:.asm=.lst) $(MAINSRC:.asm=.lst)
15 ABSS = $(FUNCSRC:.asm=.abs) $(MAINSRC:.asm=.abs)
16
17 OUTMAP = $(MAINSRC:.asm=-link.map)
18 HEXMAP = $(MAINSRC:.asm=-hex.map)
19
20 OUTFILE = $(MAINSRC:.asm=.out)
21 HEXFILE = $(MAINSRC:.asm=.hex)
22
23 .SUFFIXES: .asm .obj .abs .lst .hex .out .tab
24
25 all: $(HEXFILE) $(LSTS)
26
27 $(HEXFILE): $(OUTFILE) $(CFPREFIX)hex.cmd
28     $(CGTDIR)/hex500.exe $(HEXFLAG) $(CFPREFIX)hex.cmd -map $(HEXMAP) -o
        $(HEXFILE) -i $(OUTFILE)
29
30 .abs.lst: $(ABSS)
31     $(CGTDIR)/asm500.exe $(ASMFLAG) -x -a $<
32
33 .obj.abs: $(OUTFILE)
34     $(CGTDIR)/abs500.exe $(OUTFILE)
35
36 $(OUTFILE): $(OBJECTS) $(CFPREFIX)link.cmd rx-dsp.h int_table.h ad6620.asm
37     $(CGTDIR)/lnk500.exe $(LNKFLAG) $(CFPREFIX)link.cmd -m $(OUTMAP) -o
        $(OUTFILE) $(OBJECTS)
38
39 .tab.obj:
40     $(CGTDIR)/asm500.exe $(ASMFLAG) $< $@
41
42 .asm.obj:
43     $(CGTDIR)/asm500.exe $(ASMFLAG) $< $@
44
45 clean:
46     rm -f $(OBJECTS) $(LSTS) $(ABSS) $(OUTFILE) $(HEXFILE) $(OUTMAP)

```

[\\$\(HEXMAP\)](#)

twostagehex.cmd

```
1 /* TMS320C542 DSP Board Boot Rom Generation Command File */
2 /* 19 Dec. 2009- updated for c: drive */
3 /* 01 Nov. 2006 */
4 /* Dartmouth MASTER Rx-DSP Boot PROM Generation */

5
6 -memwidth 8
7 -romwidth 8
8 -boot          /* Convert all COFF sections to hex */
9 -bootorg 0x0000      /* External data memory boot */
10 -swwsr 0x7FFF

11
12 ROMS {
13     EPROM1: origin=0x0000, length=0x8000, memwidth=8, romwidth=8
14 }
15
16 /*SECTIONS {
17     .b1549 = boot
18     .vectors
19     .cbrev_p
20     .cfft_p
21     .log10_p
22     .smon_p
23     .sintab
24 }*/
```

twostagelink.cmd

```
1 /* TMS320C542 DSP Board Boot Rom Linker Command File */
2
3 -e RXDSP_START
4
5 MEMORY {
6 PAGE 0:
7     INTR_TABLE (RWX): origin = 0x0080, length = 0x0080
8     PROG_ANNEX (RIX): origin = 0x0180, length = 0x0680
9     PROG_MAIN (RIX) : origin = 0x0C00, length = 0x1200
10
11 PAGE 1:
12     STACK (RW)      : origin = 0x0100, length = 0x0040
13     TEMP_DATA (RW)  : origin = 0x0140, length = 0x0040
14     SBUFFER (RW)   : origin = 0x0800, length = 0x0400
15     SCALES (RW)    : origin = 0x1E00, length = 0x0200
16     DATA (RW)      : origin = 0x2000, length = 0x0800
```

```

17 }
18
19 SECTIONS {
20     .bl549      : load > PROG_ANNEX, align = 64
21     .vectors    : load > INTR_TABLE
22     .text       : load > (RIX)
23     .cbrev_p    : load > (RIX)
24     .cffft_p   : load > (RIX)
25     .log10_p   : load > (RIX)
26     .smon_p    : load > (RIX)
27     .smon_msg   : load > (RIX)
28     .sine_tab   : load > (RI)
29     .hann_tab   : load > (RI)
30     .hann_p    : load > (RIX)
31     .sercook_p : load > (RIX)
32     .ad6620    : load > (RIX)
33     .transfer_p: load > (RIX)
34     .dpsm_p    : load > (RIX)
35     .scale_p   : load > (RIX)
36
37     /* data sections */
38     .bss        : > TEMP_DATA
39     .stack_v   : > STACK
40     .sbuff_v   : > SBUFFER
41     .scale_v   : > SCALES
42     .data_v    : > DATA
43 }

```

bl549.asm

```

1 ;*****
2 ;*** Bootloader software version NO. : 1.0 ***
3 ;*** Last revision date : 10/23/1996 ***
4 ;*** Author : J. Chyan ***
5 ;*****
6 ;**
7 ;** Boot Loader Program **
8 ;**
9 ;** This code segment sets up and executes boot loader **
10 ;** code based upon data saved in data memory **
11 ;**
12 ;** WRITTEN BY: Jason Chyan **
13 ;** DATE: 06/06/96 **
14 ;**
15 ;** Revision History Omitted **
16 ;*****
17

```

```

18 ;*****
19 ;.title ""bootc54LP
20 ;*****
21 ; symbol definitions
22 ;*****
23 .mnolist
24
25 ; Let's use some scratchpad memory! Woo!
26 brs      .set    60h ; boot routine select (configuration word)
27 xentry   .set    61h ; XPC of entry point
28 entry    .set    62h ; entry point
29 hbyte    .set    63h ; high byte of -8bit serial word
30 p8word   .set    64h ; concatenator for -8bit memory load
31 src      .set    65h ; source address
32 dest     .set    66h ; destination address (dmov from above)
33 length   .set    67h ; code length
34 temp0    .set    68h ; temporary register0
35 temp1    .set    69h ; temporary register1
36 temp2    .set    6ah ; temporary register2
37 temp3    .set    6bh ; temporary register3
38 nmintv   .set    6ch ; -nonmaskable interrupt vector
39 sp_ifr   .set    6dh ; SP IFR temp reg
40 ; MMR definition for c54xlp CPU register
41 ;**
42 ifr      .set    01h
43 st0      .set    06h
44 st1      .set    07h
45 AL       .set    08h
46 AH       .set    09h
47 AG       .set    0Ah
48 brc      .set    1ah
49 pmst     .set    1dh
50 swwsr    .set    28h
51 bscr     .set    29h
52
53
54 ;* * * * * * * * * * * * * * * * * * * * *
55 ;*      Bootload from -8bit memory, MS byte first      *
56 ;* * * * * * * * * * * * * * * * * * * * *
57
58 .global   BOOTLOAD_START, blskipskip, xfr08, par08_1, endboot
59 .ref      RXDSP_START
60 .sect     ".bl549"
61 entry_point .set    RXDSP_START
62 eprom_base  .set    0x8000
63 bl_loadpoint .set   BOOTLOAD_START
64

```

```

65 BOOTLOAD_START
66 par08
67     stm    #0x7FFF,swwsr      ; set full wait states
68     stm    #0x0002,bscr       ; bus holder enabled
69     ld     #0, DP
70     nop
71     nop
72
73     st     #entry_point, @entry
74
75     stm    #eprom_base, AR1
76
77 par08_1 ; Main section load loop
78
79     nop
80     ld     *ar1+, 8, a        ; get address of destination
81     and   #0ff00h,a          ; force AG, AH to zero for correct calculation
82                     ; of the -23bit destination address. (10/14/99 BCT)
83     mvdk   *ar1+, ar3        ; ar3    <-- junkbyte.low byte
84     andm   #0ffh, @ar3        ; ar3    <-- low byte
85     or     @ar3, a           ; acc A  <-- high byte.low byte
86     stlm   a,ar2             ; ar2    <-- destination address
87
88     bc     endboot,aeq      ; section dest = 0 indicates boot end
89
90     ld     *ar1+, 8, a        ; get number of 16-bit words
91     and   #0xFF00,a          ; Clear the guard bits and keep low accum (1.92)
92     mvdk   *ar1+, ar3        ; ar3    <-- junkbyte.low byte
93     andm   #0ffh, @ar3        ; ar3    <-- low byte
94     or     @ar3, a           ; acc A  <-- high byte.low byte
95
96     cmpm   AR2, #bl_loadpoint ; check if our dest is the bootloader load
97                     address
98     bc     blskipskip, NTC    ; if not, keep loading
99
100    add    #1, A            ; if it is the bootloader, we want to skip
101    stlm   A, AR0            ; this section, i.e. skip A+1 words
102    nop
103    add    AR0, A            ; but wait,
104    stlm   A, AR0            ; A+1 words = 2(A+1) addresses (8-bit prom)
105    nop
106    bd     par08_1
107    mar   *AR1+0
108    nop
109 blskipskip:
110

```

```

111    stlm    a, brc      ; update block repeat counter register
112    nop
113    rptb    xfr08 - 1   ; block repeat to load section data
114
115    ; load program code word
116    ld      *ar1+, 8, a ; acc A <-- high byte
117    and    #0xFF00, a
118    mvdk    *ar1+, ar3   ; ar3     <-- junkbyte.low byte
119    andm    #0ffh, @ar3  ; ar3     <-- low byte
120    or     @ar3, a       ; acc A <-- high byte.low byte
121    stl     a, @p8word
122
123    ; recover destination address, pause, then write and increment
124    ldu    @ar2, a
125    nop
126    nop
127    writa   @p8word
128    add     #1, a
129    stlm    a, ar2
130
131 xfr08: ; end block repeat
132
133    b      par08_1 ; end section loop
134
135 ;**
136 ;*   End 549 8-bit EEPROM bootloader
137 ;**
138
139 endboot
140    ldu    @entry, a ; branch to the entry point
141    nop
142    nop
143    baccd  a
144    nop
145    nop

```

A.3.2 AD6620 RSP Support Code

The code in this file contains functions which initialize, stop (reset), and start the RSP. Not included are the filter tables which are loaded into memory, though the format is described in a comment.

ad6620.asm

```

1 ;~~~~~
2 ; AD6620 setup functions and tables

```

```

3 ; -----
4
5 .mmregs
6 .def      rsp_clear, rsp_reset, rsp_init, rsp_mstart, rsp_sstart
7 .include  "rx-dsp.h"
8 .sect     ".ad6620"
9
10 ;
11 ;  rsp_reset, rsp_init, rsp_mstart, rsp_sstart
12 ;      shell functions over rsp_setup
13 ;
14
15 rsp_reset:
16     ld      #ad6620_soft_reset, A ; Put AD6620 into reset
17     call    rsp_setup
18
19     retd
20     nop
21     nop
22
23 rsp_init:
24     ld      #ad6620_filter, A ; Set up AD6620 filter
25     call    rsp_setup
26
27     retd
28     nop
29     nop
30
31 rsp_mstart:
32     ld      #ad6620_master_run, A ; Start digitizing as master
33     call    rsp_setup
34
35     retd
36     nop
37     nop
38
39 rsp_sstart:
40     ld      #ad6620_slave_run, A ; Start digitizing as slave
41     call    rsp_setup
42
43     retd
44     nop
45     nop
46
47 ;
48 ;  rsp_clear      Function to clear RCF Data RAM between frames
49 ;

```

```

50
51  rsp_clear:
52      stm      #100000001b, AR3
53      nop
54      nop
55      portw   AR3, (wr_rx+amr) ; Load high and low address registers:
56      stm      #0, AR3
57      nop
58      nop
59      portw   AR3, (wr_rx+lar) ; write to 0x100, auto-increment
60
61      stm      #0xFF-1, BRC
62      nop
63      nop
64      rptb    rsp_clear_loop - 1
65
66      portw   AR3, (wr_rx+dr4)
67      portw   AR3, (wr_rx+dr3)
68      portw   AR3, (wr_rx+dr2)
69      portw   AR3, (wr_rx+dr1)
70      portw   AR3, (wr_rx+dr0)
71      nop
72
73  rsp_clear_loop:
74
75      retd
76      nop
77      nop
78
79 ; Load RSP (AD6620) Registers from Table
80 ;
81 ; 23 Dec 2009 took out most writes to terminal (msgout, dis4hex, asx)
82 ;
83 ; This code was taken directly from the "rspmod" routine used in the
84 ; Dartmouth Monitor. Instead of having the user enter the data words
85 ; or receiving them from a "script" file, this routine looks at a
86 ; table of words in memory, reads them, and transfers them to the
87 ; AD6620. Used to load control bytes and filter coefficients.
88 ;
89 ; Table entry format:
90 ;
91 ; rsp_table:
92 ;     .word    AmLah, r4r3h, r2r1h, r0xxh;
93 ;     .word    (more 4-word entries)
94 ;     .word    0FFFFh    ; End of table
95 ;
96 ; 4 words:

```

```

97 ; AAaah = AD6620 internal address, 0000h to 030Dh,
98 ;      or FFFFh to terminate.
99 ; AMR = Ma Address mode register
100 ; LAR = La Lower address register
101 ; r4r3h, r2r1h, r0xxh = data bytes, packed into words, MS, to LS.
102 ;      Bottom byte of 3rd word not used (xx). Data is treated a 40
103 ;      bits for all AD6620 registers. Not the most compact
104 ;      arrangement for storage, but readable- and it can be edited
105 ;      directly from monitor scripts.
106 ;
107 ; DR4 = r4
108 ; DR3 = r3
109 ; DR2 = r2
110 ; DR1 = r1
111 ; DR0 = r0
112 ;
113 ; Uses:
114 ; A:   Holds table pointer upon entry
115 ; B:   Working register
116 ; AR0: I/O address
117 ; AR2: Table index
118 ; AR3: Holds data to send to or read from I/O port
119
120 table_end .set 0FFFFh           ; End-of-table definition
121
122 rsp_setup:                   ; Enter with table starting address in A
123     stlm  A,AR2    ; Save to AR2 for later use
124     nop          ; Necessary for loop to execute properly (?!&)
125     nop          ; Necessary for loop to execute properly (?!&)
126 ;     ld   #ad6620_msg1,A ; Tell operator what is happening
127 ;     call msgout
128
129 rsp_loop:
130 ;     ldm  AR2,A    ; Retrieve table index
131 ;     call dis4hex  ; Display index of table line
132 ;     ld   #0020h,A  ; Space over on screen
133 ;     call asx
134
135     cmpm  *AR2,#table_end ; Is this the end of the table?
136     bc    rspx,TC        ; Return if at end
137
138     ld   *AR2+,A    ; Get first table word: AD6620 address
139     ld   A,B        ; Save a copy
140
141 ;     call dis4hex  ; Display
142
143 ;

```

```

144 ; Transfer RSP register address bytes to
145 ; AD6620 high and low address registers
146 ;
147     sftl    A,-8,A      ; Shift high byte to low byte
148     and     #0003h,A,A  ; Mask high byte to 2 LSBs (avoid reserved bits
149                           ; and do not auto-increment for now)
150     stlm    A,AR3       ; Move to AR3 for portw
151     portw   AR3,wr_rx+amr ; Write to high address register
152     ld      B,A         ; Get RSP register address
153     and     #00FFh,A,A  ; Mask to low byte only (actually hardware
154                           ; only uses bits 7:0 of data bus, should not
155                           ; need to mask)
156     stlm    A,AR3
157     portw   AR3,wr_rx+lar ; Write to low address register
158
159 ;     ld      #0020h,A  ; Space over
160 ;     call    asx
161
162     ld      *AR2+,A    ; Get next table word (dr4 and dr3 bytes)
163     ld      A,B         ; Save a copy
164
165 ;     call    dis4hex    ; Display
166
167     sftl    A,-8,A      ; Shift high byte to low byte
168     stlm    A,AR3       ; AR3 holds output data
169     portw   AR3,wr_rx+dr4 ; Store to AD6620 MS data byte register
170
171     ld      B,A         ; Get copy
172     and     #00FFh,A,A  ; Mask to low byte only
173     stlm    A,AR3
174     portw   AR3,wr_rx+dr3
175
176 ;     ld      #0020h,A  ; Space over
177 ;     call    asx
178
179     ld      *AR2+,A    ; Get next table word (dr2 and dr1 bytes)
180     ld      A,B         ; Save a copy
181
182 ;     call    dis4hex  ; Display
183
184     sftl    A,-8,A      ; Shift high byte to low byte
185     stlm    A,AR3
186     portw   AR3,wr_rx+dr2
187
188     ld      B,A         ; Get copy
189     and     #00FFh,A,A  ; Mask to low byte only
190     stlm    A,AR3

```

```

191      portw    AR3,wr_rx+dr1
192
193 ;     ld      #0020h,A      ; Space over
194 ;     call    asx
195
196     ld      *AR2+,A      ; Get next table word (dr0 in upper byte)
197
198 ;     call    dis4hex   ; Display
199
200     sftl    A,-8,A  ; Shift high byte to low byte
201     stlm    A, AR3  ; Save for subsequent output port write
202     portw   AR3,wr_rx+dr0 ; Address for RSP LS data byte
203
204 ;     ld      #0020h,A      ; Space over
205 ;     call    asx
206 ;     ld      #000Dh,A  ; Output CR
207 ;     call    asx
208 ;     ld      #000Ah,A  ; LF
209 ;     call    asx
210
211     b      rsp_loop  ; Go back for next table entry
212 rspx:
213 ;     ld      #ad6620_msg2,A
214 ;     call    msgout
215     ret

```

A.3.3 Data Processing Functions

Below lies a subset of the data processing functions used by the AGO code. These are the codes written by in-house, not ones provided in or adapted from the TI DSP library.

scale.asm This file provides three functions: two stages of post-FFT scaling, first before the square-magnitude function, and then again before the logarithm, and then the descaling function for post-logarithm.

```

1 ;~~~~~;
2 ; Prescaling functions by Micah P. Dombrowski
3 ;
4 ; _sqmag_prescale
5 ;
6 ; Used on 32-bit complex number array (stored RIRI), finds the
7 ; largest possible shift applicable to each RI pair using EXP.
8 ; Assumes a zero return equates to EXP(0), and stores the maximum
9 ; shift. Stores 2*shift in scale factor array.
10 ;
11 ; _log_prescale

```

```

12 ;
13 ; For TI DSP Library Logarithm: normalizes each 32-bit value using
14 ; EXP and NORM, adding shift values to existing values in the save
15 ; array, and cutting to 16-bit output.
16 ;
17 ; _descale
18 ;
19 ; Adjusts logarithmic output based on scale factor array, by
20 ; subtracting scale*log10(2).
21 ;-----
22
23 .mmregs
24
25 ; Stack usage
26 ; 0 = ST1, 1 = ST0, 2 = function return pointer
27     .asg    *SP(3), idata
28     .asg    *SP(4), odata
29     .asg    *SP(5), sdata
30
31     .def    _sqmag_prescale, _log_prescale, _descale
32     .sect   .scale_p
33
34 ;~~~~~`~~
35 ; _sqmag_prescale
36 ;
37 ; Inputs:      N, number of values to scale in Acc,
38 ; Top of Stack: data input address (512x2 words present),
39 ;                  data output address (512 words free),
40 ;                  scale save address (512 words free)
41
42 _sqmag_prescale
43
44 ; Set up processor for signed, non-fractional math
45     pshm    ST0
46     pshm    ST1
47     ssbx    CPL
48     rsbx    FRCT
49     ssbx    SXM
50     ssbx    OVM
51     rsbx    C16
52     nop
53     nop
54
55     sub     #1, A ; BRC = N-1
56     stlm   A, BRC
57     stm    #16, ARO    ; max shift value
58     mvdk   idata, AR2  ; input pointer

```

```

59      mvdk    odata, AR3 ; output pointer
60      mvdk    sdata, AR4 ; scale array pointer
61
62      rptb    sqmag_prescale_loop - 1
63
64      dld     *AR2+, A
65      dld     *AR2-, B
66      nop
67      nop
68
69      exp    A
70      nop
71      ldm    T, A
72
73      exp    B
74      nop
75      ldm    T, B
76
77      min    A ; A = min(A,B)
78      nop
79      nop
80
81      sub    #4, A ; 4 guard bits
82
83
84      stlm    A, T ; re-store to T
85      nop
86      nop
87
88      pshm    T ; save T to stack
89
90      dld     *AR2+, A
91      dld     *AR2+, B
92      nop
93      nop
94
95      norm   A ; shift
96      norm   B
97
98      .global norm_ovm
99 norm_ovm:
100
101      dst    A, *AR3+ ; save data
102      dst    B, *AR3+
103
104      ld     #0, A ; clear Acc
105      popm   AL ; pop the corrected scale factor into low Acc

```

```

106     stl      A, 1, *AR4+ ; save with a 1-bit shift (mpy by 2)
107
108 sqmag_prescale_loop:
109
110     popm    ST1
111     popm    ST0
112
113     retd
114     nop
115     nop
116
117 ;~~~~~
118 ; _log_prescale
119 ;
120 ; Inputs:      N, number of values to scale in Acc,
121 ; Top of Stack: data input address (512x2 words present),
122 ;                  data output address (512 words free),
123 ;                  scale save address (512 words free)
124 ;
125
126 _log_prescale
127
128 ; Set up processor for fractional, signed math
129     pshm    ST0
130     pshm    ST1
131     ssbx    CPL
132     ssbx    FRCT
133     ssbx    SXM
134     ssbx    OVM
135     rsbx    C16
136     nop
137     nop
138
139     sub     #1, A          ; BRC = N-1
140     stlm   A, BRC
141     stm    #16, ARO      ; max shift value
142     mvdk   idata, AR2    ; input pointer
143     mvdk   odata, AR3    ; output pointer
144     mvdk   sdata, AR4    ; scale array pointer
145
146     rptb   log_prescale_loop - 1
147
148     dld    *AR2+, A
149     exp   A
150     nop
151
152     ldm   T, B          ; load T

```

```

153    sub    #4, B      ; guard bits
154    stlm   B, T
155    add    *AR4, B    ; add any existing scale factor
156    stl    B, *AR4+   ; save back to scale array
157    nop
158    nop
159
160    norm   A  ; shift
161
162    sth    A, *AR3+   ; save data
163
164 log_prescale_loop:
165
166    popm   ST1
167    popm   ST0
168
169    retd
170    nop
171    nop
172
173    ;~~~~~
174    ; _descale
175    ;
176    ;
177    ; Inputs:      N, number of input points, in Acc
178    ; Top of Stack: data input address (512x2 words present, Q16.15 format)
179    ;                  data output address (512 words free, in-place okay)
180    ;                  scale factor array (512 words present)
181
182 log10c32767     .set    0x783F  ; ( log10(32767) * 2^15) >> 3
183 log10c2         .set    0x04D1  ; ( log10(2) * 2^15 ) >> 3
184
185
186 _descale:
187
188    pshm   ST0
189    pshm   ST1
190    ssbx   CPL
191    rsbx   FRCT
192    ssbx   SXM
193    rsbx   OVM
194    rsbx   C16
195    nop
196    nop
197
198    sub    #1, A      ; BRC = N-1
199    stlm   A, BRC

```

```

200    mvdk    idata, AR2      ; input pointer
201    mvdk    odata, AR3      ; output pointer
202    mvdk    sdata, AR4      ; scale array pointer
203
204    rptb    descale_loop - 1
205
206    ; Docs say log10 outputs Q16.15, but this is misleading,
207    ; format is S IIII IIII IIII IIII FFFF FFFF FFFF FFF
208    dld    *AR2+, A
209    sfta   A, #-3
210
211    ld     #log10o32767, B
212    add    B, A
213
214    ld     #log10o2, B
215
216    rpt    *AR4+
217    sub    B, A
218    sfta   A, #8
219    sfta   A, #8
220    sat    A
221
222    sth    A, *AR3+
223
224    descale_loop:
225
226    popm   ST1
227    popm   ST0
228
229    retd
230    nop
231    nop
232
233
234    .end

```

`dpsm.asm` This provides a single function, $|C^2|$.

```

1 ;~~~~~
2 ; Double-precision square magnitude function by Micah P. Dombrowski
3 ;
4 ; Reads n Q.31 numbers arrayed as R[0], I[0], R[1], I[1], ..., R[n-1],
5 ; I[n-1] outputs MSB half of R[0]^2+I[0]^2, R[1]^2+I[1]^2, ...,
6 ; R[n-1]^2+I[n-1]^2 output fills first half of input region.
7 ;
8 ; Inputs: data address in A, number of R/I pairs in B
9

```

```

10      .mmregs
11      .def      _sqmag
12      .sect     .dpsm_p
13      _sqmag
14
15      pshm    ST0
16      pshm    ST1
17      ssbx    SXM
18      ssbx    FRCT
19      ssbx    OVM
20      rsbx    C16
21      nop
22      nop
23
24 ; Double-precision square magnitude, saving MSB half of result.
25
26      stm      #0, T      ; Multiplication Temp register (for mpy)
27      stm      #0, BK     ; Circuluar addressing modulus (do not want)
28      sub      #1, B
29      stlm     B, BRC
30      stm      #2, AR0   ; Increment (jump to next 32-bit datum)
31      stlm     A, AR2   ; Load index
32      stlm     A, AR3   ; Load index
33      stlm     A, AR4   ; Storage index
34      rptb    sqmag_loop - 1
35
36      mpy      *AR2+, A           ; a = 0          (1)
37      macsu   *AR2-, *AR3+, A     ; a = RL*RH    (1)
38      macsu   *AR3-, *AR2, A      ; a += RH*RL   (1)
39      ld       A, -16, A         ; a >= 16      (1)
40      mac      *AR2+0%, *AR3+0%, A ; a += RH*RH   (1)
41      stm      #0, T ; (2)
42      sat      A      ; (1)
43
44      mpy      *AR2+, B           ; b = 0          (1)
45      macsu   *AR2-, *AR3+, B     ; b = IL*IH    (1)
46      macsu   *AR3-, *AR2, B      ; b += IH*IL   (1)
47      ld       B, -16, B         ; b >= 16      (1)
48      mac      *AR2+0%, *AR3+0%, B ; b += IH*IH   (1)
49      stm      #0, T ; (2)
50      sat      B      ; (1)
51
52
53      add      B, A      ; a += b == R^2 + I^2
54      sat      A
55      dst      A, *AR4+ ; (1)
56

```

```

57 sqmag_loop:
58
59     popm    ST1
60     popm    ST0
61
62     nop
63     nop
64
65     retd
66     nop
67     nop

```

tablemax.asm The function provided by this file reduces the final data set by taking the max between a provided set of indices within the 512-bin FFT. The bins in the table can be spaced by 1 to have ranges of complete data transferred.

```

1 ;~~~~~
2 ; Frequency selection and averaging function by
3 ; Nathan Utterback and Micah P. Dombrowski
4 ;
5 ; Inputs: start address of data Acc,
6 ;          output address in Bcc
7
8 avg_shift_val    .set 3 ; bits to right shift by after summing
9
10    .mmregs
11    .def      transfer, transfer_table_sz
12    .sect     .transfer_p
13
14    .bss      Delta,1,0,0 ; storage for repeat counter
15    .bss      nShift,1,0,0 ; storage for shift value
16
17 transfer:
18
19     pshm    ST0
20     pshm    ST1
21
22     pshm    AR6
23
24     stlm    A, AR2
25     stlm    B, AR3
26     stm     transfer_table_start, AR5 ; load the start of the
27                                         ; table into memory
28     stm     transfer_table_end-1, AR0
29
30
31 transfer_sum_loop:

```

```

32
33     ldm    AR2, A      ; load base address
34     add    *AR5+, A    ; add offset from table, inc
35     stlm   A, AR4      ; store
36
37     ld     *AR5-, A    ; load next offset, dec
38     sub    *AR5+, A    ; subtract current offset to get delta, inc
39     sub    #1, A
40     stlm   A, BRC      ; store delta-1 for rpt
41
42     ld     #0, B
43     rptb  max_loop - 1
44
45     ld     *AR4+, A
46     max   B
47
48 max_loop:
49
50     stl   B, *AR3+    ; save
51
52     cmpr  LT, AR5
53     bc    transfer_sum_loop, TC ; loop until we finish the table
54
55     popm  AR6
56
57     popm  ST1
58     popm  ST0
59
60     retd
61     nop
62     nop
63
64 transfer_table_start:
65     .word   25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105
66
67     .word   113, 114, 115, 116, 117, 118, 119, 120, 121
68     .word   122, 123, 124, 125, 126, 127, 128, 129, 130
69     .word   131, 132, 133, 134, 135, 136, 137, 138, 139
70     .word   140, 141, 142, 143, 144, 145, 146, 147, 148
71     .word   149, 150, 151, 152, 153, 154, 155, 156, 157
72     .word   158, 159, 160, 161, 162, 163, 164, 165
73
74     .word   166, 174, 182, 190, 198, 206, 214, 222, 230
75     .word   238, 246, 254, 262, 270, 278, 286, 294, 302
76     .word   310, 318, 326, 334, 342, 350, 358, 366, 374
77     .word   382, 390, 398, 406, 414, 422, 430, 438, 446
78     .word   454, 462, 470, 478

```

```

79 transfer_table_end:
80
81 transfer_table_sz      .set      transfer_table_end-transfer_table_start

```

sercook.asm Finally, this provides a function to change the data into the right form for serial output.

```

1 ;~~~~~
2 ; Serial data cooking function by Micah P. Dombrowski
3 ;
4 ; Reads N words containing right-aligned bytes,
5 ; bit reverses, and adds start and stop bits.
6 ;
7 ; Inputs: data address in A, number of bytes in B
8
9
10 .mmregs
11 .def    _serial_cook
12 .sect   .sercook_p
13 _serial_cook
14
15 sub    #1, B
16 stlm   B, BRC
17 stlm   A, ARO
18 rptb   bitrev_loop - 1
19
20 ssbx   XF
21
22 ld    #1, B      ; zero result + stop bit
23
24 ld    #001h, A   ; load mask
25 and   *ARO, A    ; mask data
26 or    A, 8, B    ; OR into result
27 ld    #002h, A
28 and   *ARO, A
29 or    A, 6, B
30 ld    #004h, A
31 and   *ARO, A
32 or    A, 4, B
33 ld    #008h, A
34 and   *ARO, A
35 or    A, 2, B
36 ld    #010h, A
37 and   *ARO, A
38 or    A, 0, B
39 ld    #020h, A
40 and   *ARO, A

```

```

41      or      A, -2, B
42      ld      #040h, A
43      and    *ARO, A
44      or      A, -4, B
45      ld      #080h, A
46      and    *ARO, A
47      or      A, -6, B
48
49      stl    B, *ARO+ ; rewrite to serial buffer
50
51      rsbx   XF
52
53 bitrev_loop:
54
55      retd
56      nop
57      nop
58
59      .end

```

A.3.4 S-ARC Main Program Code

The main code which ties all of the above—as well as the windowing, FFT, and log10 functions—together.

```

1  ;~~~~~;
2  ;
3  ; Dartmouth College AGO Rx-DSP Program
4  ;
5  ; Written by: Micah P. Dombrowski and Nathan B. Utterback
6  ;      w/ code segments from TI DSP Library
7  ;
8  ;_____
9
10 .mmregs
11 .global ZERO, BMAR, PREG, DBMR, INDX, ARCR, TREG1
12 .global TREG2, CBSR1, CBER1, CBSR2, CBER2
13 .global RXDSP_START
14 .ref   _cbrev32, _cfft32_512, _log_10
15 .ref   _hann_window, _sqmag, _serial_cook
16 .ref   _log_prescale, _sqmag_prescale, _descale
17 .ref   rsp_clear, rsp_reset, rsp_init, rsp_mstart, rsp_sstart
18 .ref   transfer, transfer_table_sz
19 .global bridge_data, buff_clear_loop
20 .def   ago_main, int_nmi
21

```

```

22     .include "rx-dsp.h"
23     .text
24
25 code_version    .string  "v1.0"
26 band_width      .string  "0300"
27
28 ; Output constants
29
30 output_shift_n   .set 8          ; left shift before 8-bit mask defines
31 header_freq_mask .set 0x0FFF    ; bits of 32-bit major frame counter
32                                     ; that must be zero for a header frame
33
34 ; Run constants
35 fft_scaling       .set 0
36 data_n            .set 512        ; Size of each FFT (# of IQ pairs)
37 data_discard       .set 512        ; words discarded from Rx FIFO pre-data
38 data_minor_sz     .set 1          ; acquisitions per half-buffer interrupt
39 fsync_sz          .set 4          ; # of serial frame sync bytes
40                   ; (should be multiple of 4)
41 abu_buff_sz       .set 214        ; size of serial buffer
42                   ; (2x major frame size IN BYTES)
43                   ; should be set to
44                   ; 2*(transfer_table_sz + fsync_sz)
45
46 ; Memory allocations
47 data_addr         .usect ".data_v", 0x800, 1, 1
48 scale_addr        .usect ".scale_v", 0x200, 1, 1
49 stackres          .usect ".stack_v", 0x40, 1, 1
50 abu_buff_loc      .usect ".sbuff_v", abu_buff_sz, 1, 1
51 abu_buff_hloc    .set    abu_buff_loc+abu_buff_sz/2 ; half-way
52
53 ; Memory pointers
54 iq_data           .set    data_addr      ; 512 * 2 words * I/Q
55 fft_data          .set    data_addr      ; 512 * 2 words * Re/Im
56 scale_data        .set    scale_addr     ; 512 words
57 sqmag_data        .set    data_addr      ; 512 * 2 words
58 sqsc_data         .set    data_addr+2*data_n ; 512 words
59 log_data          .set    data_addr      ; 512 * 2 words
60 power_data        .set    data_addr      ; 512 words
61 ebs_data          .set    data_addr+data_n ; 512 words
62
63 ; mode flags
64 mode_std_bit      .set    0001b        ; standard operations
65 mode_dbg_bit      .set    0010b        ; debug
66
67 mode_std_n        .set    transfer_table_sz
68 mode_dbg_n        .set    512

```

```

69
70 ; Scratchpad RAM usage
71 bridge_count      .set    scratch
72 minor_count       .set    scratch+1
73 bspce_save        .set    scratch+2
74 bridge_size       .set    scratch+3
75 mode_flag         .set    scratch+4
76 shb_addr          .set    scratch+5
77 major_count       .set    scratch+6      ; two words!
78 nco_freq          .set    scratch+8      ; two words!

79
80     .bss TempLmem,1*2,0,0 ;temporary dword
81
82
83 RXDSP_START
84 ago_main:
85
86     rsbx    XF
87
88 ; Processor setup
89     ssbx    INTM      ; Disable interrupts
90     stm     #(stackres+0x40), SP ; set Stack Pointer
91     stm     #npmst,PMST   ; Set processor mode/status
92 ;     stm     #defst0, ST0
93 ;     stm     #defst1, ST1
94     rsbx    SXM       ; Suppress sign extension
95 ;     rsbx    XF
96     nop     ; Space for branch to app
97     nop
98
99 appcode:
100 ;     stm     #0,state  ; Clear interrupt routine state
101     stm     #0,AR0     ; Clear auxilliary register 0
102
103     portw   ARO,wr_disc ; Enable parallel TLM drivers, I_Q out
104     portw   ARO,wr_dog   ; Strobe watchdog timer
105
106     stm     #0,AR0     ; Clear all auxiliary registers
107     stm     #0,AR1
108     stm     #0,AR2
109     stm     #0,AR3
110     stm     #0,AR4
111     stm     #0,AR5
112     stm     #0,AR6
113     stm     #0,AR7
114
115     stm     #0FFh,IFR   ; Clear any pending interrupts

```

```

116     stm    #ntss,TCR ; Stop timer, if running
117
118 ; Main data code start
119 read_init:
120     call    rsp_reset
121     nop
122     nop
123     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
124     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
125     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
126     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
127     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
128     portw  ARO,wr_rs_rx ; Hardware reset of AD6620 RSP
129     nop
130     nop
131     call    rsp_reset
132     call    rsp_init
133     call    rsp_clear
134     call    rsp_mstart
135
136     rpt    #4444 ; Let the AD6620 do its first initialization in peace
137     nop
138
139     call    rsp_reset
140
141 ; store permanent NCO Frequency
142     ld     #0x4420, A
143     stl   A, @nco_freq
144     ld     #0x01FA, A
145     stl   A, @(nco_freq+1)
146
147     portr rs_rx_fifo, ARO ; Reset RxFIFO - also wired to Slave
148     nop
149     nop
150
151     stm    #lsb_sel, ARO ; Reset acq_seq
152     portw ARO, wr_disc
153     nop
154
155 ; BSP prep
156
157     stm    #(bspc_Free+bspc_fsm), BSPCO ; reset BSP
158     stm    #(int_bx), IMR ; unmask serial tx interrupt
159     stm    #(bspce_fe+bspce_bxe), BSPCEO ; 10-bit words,
160                                ; enable tx autobuffer
161
162 ; where in the 2 kw of buffer RAM does the transmit buffer start?

```

```

163     stm      #(abu_buff_loc-0x800), AXR
164     stm      #(abu_buff_sz), BKK ; buffer size
165
166 ; Clear entire serial buffer
167     stm      #abu_buff_sz-1, BRC
168     stm      #abu_buff_loc, AR4
169     rptb    buff_init_loop - 1
170
171     st      #0h, *AR4+
172
173 buff_init_loop:
174     .global buff_init_loop, head_ramp, major_loop
175
176 ; Write out header to first buffer half
177
178     stm      #abu_buff_loc, AR4
179     ; two-byte frame sync 0xEB90
180 ;     stm      #(abu_buff_loc+abu_buff_sz/2-2), AR4
181     st      #0xFE, *AR4+ ; 4-byte initialization sync
182     st      #0x6B, *AR4+
183     st      #0x28, *AR4+
184     st      #0x40, *AR4+
185
186     stm      #file_header, ARO ; Point to static header words
187 header_loop:
188     ld      *ARO+, A           ; Get a word, point to next
189     bc      header_loop_x, AEQ ; If terminator, end static header
190     stl    A, *AR4+           ; Write to serial buffer
191     b      header_loop
192 header_loop_x:
193
194     ld      #abu_buff_loc, A
195     ld      #(abu_buff_sz/2), B
196
197     call    _serial_cook
198
199 ; Set initial bspce_save such that the first acquisition will
200 ; write to the second half of the serial buffer
201     st      #bspce_xh, @bspce_save
202
203 ; Start BSP transmits
204     ; have to hold fsm bit
205     stm      #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0
206
207 ;     rsbx    INTM ; global interrupt enable
208
209 ; All Aux Registers are fungible in the main loop: values which must

```

```

210 ; be preserved over time are stored in the scratchpad RAM as defined
211 ; above. Note that only AR6 and AR7 are required to be preserved by
212 ; the DSP Library functions (and most others), all other ARx may be
213 ; modified within function calls.

214
215     ld      #0, A
216     dst    A, @major_count
217
218 major_loop:
219
220     portr   rs_rx_fifo, ARO ; Reset Rx FIFO - also wired to Slave
221     nop
222     nop
223
224     stm     #(acq_seq_out+lsb_sel), ARO ; send acq_seq, set lsb_sel
225     portw   ARO, wr_disc
226
227     call    rsp_clear      ; clear NCO RAM
228     call    rsp_mstart     ; and start digitizing
229
230 ;
231 ; Reads
232 ;
233 data_acq_start:
234
235     st      #0, @bridge_count ; reset bridged data counter
236     st      #0, @minor_count  ; set minor frame counter
237
238 ;
239 ; Set wait states for Rx FIFO
240 ;
241     ldm    SWWSR, A
242     ld     #65535, B
243     xor    #7, #swwsr_is, B ; (0b111<<Nset XOR 0d65535) creates bitmask
244     and    B, A            ; mask out bits of interest
245     or     #0, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
246     ; Nwait = 0 means no additional waits
247     stlm   A, SWWSR
248
249     ssbx   XF
250     rpt    #100
251     nop
252     rsbx   XF
253
254     .global  pre_disc, pre_read
255 pre_disc:
256

```

```

257 ; loop to read and discard first data out of AD6620
258     stm    #(data_discard), AR2
259     nop
260 rx_discard_loop:
261     ; read only if rx fifo is nonempty
262     portr   rd_disc, AR0
263     nop
264     nop
265     bitf   AR0, #rx_ef0
266
267     bc    discard_fifo_empty, NTC
268     portr   rd_rx_out, AR1 ; read data into AR1
269     mar    *AR2-           ; decrement word counter
270 discard_fifo_empty:
271
272     banz   rx_discard_loop, *AR2
273
274
275 ; loop to read data from Rx FIFO into RAM
276     stm    #2, AR0
277     stm    #iq_data, AR1 ; set address for first data word
278     stm    #(data_n*2-1), AR2
279 pre_read:
280     nop
281 rx_read_loop:
282     ; read only if rx fifo is nonempty
283     portr   rd_disc, AR3
284     nop
285     nop
286     bitf   AR3, #rx_ef0
287
288     bc    read_fifo_empty, NTC
289
290     portr   rd_rx_out, *AR1+
291     st     #0, *AR1+ ; zero out second word
292     mar    *AR2-           ; decrement word counter
293
294 read_fifo_empty:
295
296     banzd   rx_read_loop, *AR2
297     nop
298     nop
299
300 ;
301 ; Set wait states for other stuff (full 7)
302 ;
303     ldm    SWWSR, A

```

```

304     or      #7, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
305     stlm   A, SWWSR
306     nop
307     nop
308
309     ssbx   XF
310     rpt    #50
311     nop
312     rsbx   XF
313
314 ;
315 ; End data acquisition, begin data processing
316 ;
317
318 data_process:
319
320
321     .global pre_window
322 pre_window:
323
324     ld      #iq_data, A
325     ld      #data_n, B
326
327     call    _hann_window
328
329     .global pre_bit_rev
330 pre_bit_rev:
331
332 ; Bit reversal
333     stm    #data_n, ARO
334     pshm  ARO
335     stm    #iq_data, ARO
336     pshm  ARO
337     ld     #iq_data, A
338
339     call    _cbrev32
340
341     frame  2
342
343
344     .global pre_fft
345 pre_fft:
346     stm    #fft_scaling, ARO
347     pshm  ARO
348     ld     #fft_data, A
349
350     call    _cfft32_512

```

```

351
352     frame    1
353
354
355     .global    pre_move
356 pre_move:
357 ; flip things into proper power spectra order (swap halves)
358 ; remove zeroes here, too
359
360     stm      #(data_n-1), BRC
361     stm      #fft_data, AR2
362     stm      #(fft_data + 2*data_n), AR3
363
364     rptb    move_loop - 1
365
366     dld      *AR2, A
367     dld      *AR3, B
368
369     nop
370     nop
371     xc      2, AEQ
372     add     #1, A
373     xc      2, BEQ
374     add     #1, B
375
376     dst      A, *AR3+
377     dst      B, *AR2+
378
379 move_loop:
380
381     .global    pre_sqscale
382 pre_sqscale:
383
384     stm      #scale_addr, ARO ; scale saves
385     pshm    ARO
386     stm      #fft_data, ARO ; output
387     pshm    ARO
388     stm      #fft_data, ARO ; input
389     pshm    ARO
390     ld      #data_n, A ; N
391
392     call     _sqmag_prescale
393
394     frame    3 ; free stack
395
396
397     .global pre_abs

```

```

398 pre_abs:
399     ssbx    SXM
400     ssbx    OVM
401     nop
402     nop
403
404     stm    #(2*data_n-1), BRC
405     stm    #fft_data, ARO
406     rptb   abs_loop - 1
407
408     dld    *ARO, A
409     abs    A
410     dst    A, *ARO+
411
412 abs_loop:
413
414     .global  pre_sqmag, pre_log, pre_db
415 pre_sqmag:
416 ; |FFT|^2
417     ld     #sqmag_data, A
418     ld     #data_n, B
419
420     call   _sqmag
421
422     .global pre_logps
423 pre_logps:
424
425     stm    #scale_data, ARO ; scale saves
426     pshm   ARO
427     stm    #sqsc_data, ARO ; output
428     pshm   ARO
429     stm    #sqmag_data, ARO ; input
430     pshm   ARO
431     ld     #data_n, A ; N
432
433     call   _log_prescale
434
435     frame  3
436
437
438 pre_scale:
439
440
441 pre_log:
442 ; log_10(|FFT|^2) (outputs 32-bit Q16.15)
443
444     stm    #data_n, ARO

```

```

445     pshm    ARO
446     stm     #log_data, ARO ; write to beginning of data buffer
447     pshm    ARO
448     ld      #sqsc_data, A   ; read from halfway point of data buffer
449
450     call _log_10
451
452     frame   2
453
454
455     .global pre_descale
456 pre_descale:
457
458     stm     #scale_data, ARO ; scale saves
459     pshm    ARO
460     stm     #power_data, ARO ; output
461     pshm    ARO
462     stm     #log_data, ARO   ; input
463     pshm    ARO
464     ld      #data_n, A     ; N
465
466     call    _descale
467
468     frame   3 ; free stack
469
470
471     .global post_descale
472 post_descale:
473 ; multiply by output_scale_factor,
474 ; shift right by output_shift_n, re-store
475
476     pshm    ST0
477     pshm    ST1
478     ssbx    FRCT
479     ssbx    SXM
480     ssbx    OVM
481     rsbx    C16
482
483 ; Scale and shift, save 8-bit data
484
485     stm     #data_n-1, BRC
486     stm     #power_data, ARO
487     stm     #(data_addr + 2*data_n), AR1
488     stm     #ebs_data, AR2
489 ;     stm     #output_scale_factor, T
490     rptb    ebs_loop - 1
491

```

```

492 ;      mpy      *AR0+, A ; multiply by scale factor in T
493 ;      sfta     A, #0-output_shift_n
494 ;      and      #0xFF0000, A
495 ;      dst      A, *AR1+
496
497 ;      mpy      *AR0+, A ; multiply by scale factor in T
498      ld       *AR0+, A
499      sfta     A, #0-output_shift_n
500      add      #128, A
501      and      #0xFF, A
502      stl      A, *AR2+
503 ;      dadd     output_shift_n, A ; shift
504 ;      sat      A
505 ;      and      #0xFF, #16, A ; mask to A(23-16)
506 ;      sth      A, *AR2+      ; store A(23-16)
507
508 ebs_loop:
509
510      nop
511
512      .global    dp_end
513 dp_end:
514
515 ; Debug code, writes a 512-byte ramp-up-ramp-down
516 ;      stm      #data_n/2-1, BRC
517 ;      stm      #0, AR0
518 ;      stm      #ebs_data, AR2
519 ;      stm      #ebs_data+data_n-1, AR3
520 ;      rptb     dummy_data - 1
521 ;
522 ;      mvkd     AR0, *AR2+
523 ;      mvkd     AR0, *AR3-
524 ;      mar      *AR0+
525 ;
526 ;dummy_data:
527 ;      .global    dummy_data
528
529      popm     ST1
530      popm     ST0
531
532      nop
533
534 ;
535 ; End data processing, begin serial data handling
536 ;
537
538      stm      #(lsb_sel), AR0

```

```

539     portw    ARO, wr_disc
540
541 ; Standard mode (#buff_size bytes) or
542 ; debug mode (#debug_size) depending on trm_28 state.
543     portr    rd_disc, ARO      ; Get discrete bits
544
545 ;     stm    #0, ARO      ; DEBUG !!
546 ;     nop
547 ;     nop
548
549     bitf    ARO, #trm_28 ; Test for high terminal input
550     bc      standard_mode, NTC ; If trm_28 is low (NTC), standard
551 ;                                ; data settings to #debug_size
552
553     st      #mode_dbg_n, @bridge_size
554     st      #mode_dbg_bit, @mode_flag
555
556     b      debug_mode_skip
557
558 standard_mode:
559
560     st      #mode_std_n, @bridge_size
561     st      #mode_std_bit, @mode_flag
562
563 debug_mode_skip:
564
565 ; entry point for bridging data transfers
566 ; over multiple serial half-buffers
567 bridge_data:
568     stm      #(acq_test2+lsb_sel), ARO ; send acq_seq, set lsb_sel
569     portw   ARO, wr_disc
570
571     nop
572     nop
573 ; Determine serial buffer position
574     ld      #abu_buff_loc, A ; load buffer base
575
576     bitf    @bspce_save, #bspce_xh ; read XH out of
577 ;                                ; stored BSPCE register
578     nop
579     nop
580     bc      buff_skip, NTC ; if first half _finished_
581 ;                                ; (XH=0, NTC), do nothing
582
583     add     #(abu_buff_sz/2), A
584     stm      #(acq_test2+acq_test3+lsb_sel), ARO
585     portw   ARO, wr_disc

```

```

586
587  buff_skip:
588      nop
589      nop
590      stl  A, @shb_addr ; scratch storage for serial half-buffer address
591      nop
592      nop
593
594  abu_first_half:
595      .global abu_first_half
596
597      ssbx    INTM
598
599 ; Clear serial buffer half
600      mvdm  @shb_addr, AR4
601      nop
602      rpt   #(abu_buff_sz/2-1)
603      st    #0xFF, *AR4+
604
605 ; reset AR4 for data copy
606      mvdm  @shb_addr, AR4
607      nop
608      nop
609
610  abu_fill_start:
611      .global abu_fill_start
612
613      ; two-byte frame sync 0xEB90
614      st    #0xEB, *AR4+
615      st    #0x90, *AR4+
616
617      ; two-byte infofooop
618      ld    @minor_count, A ; byte 1, minor frame number
619      and   #0xFF, A
620      stl   A, *AR4+
621      dld   @major_count, A ; byte 2, major frame number
622      and   #0xFF, A, B
623      stl   B, *AR4+
624
625  post_sync_write:
626      .global post_sync_write
627
628 ; Transfer in selected data mode
629      bitf  @mode_flag, #mode_dbg_bit
630      bc    dbg_transfer, TC
631
632 ; In standard mode, check if 8-bit major_count (still in A) == 0,

```

```

633 ; if so, transfer a header instead of data.
634
635 ; Std header: spit out a header frame every 4096th
636     and    #header_freq_mask, A
637     bc     header_skip, ANEQ ; A != 0, skip header
638
639     call    hwrite
640     st     #mode_std_n, @bridge_count ; fake it out
641     nop
642     nop
643
644     b      end_transfer
645
646 header_skip:
647 ; Std transfer: selected bins in a 1-frame major frame
648     ; transfer selected data to serial buffer
649     ld     #ebs_data, A ; input addr in A
650     ldm    AR4, B        ; output addr in B
651
652     call    transfer
653
654     st     #mode_std_n, @bridge_count ; fake it out
655
656     b      end_transfer
657
658 ; Debug transfer: entire 512-bin fft
659 ; spread over multiple minor frames.
660 dbg_transfer:
661
662     ; copy raw data (words) into serial buffer (bytes)
663     ld     #ebs_data, A
664     add   @bridge_count, A
665     stlm  A, AR2
666
667     mvdm  @bridge_size, ARO      ; goal bridge size
668     mvdm  @bridge_count, AR1      ; current count
669
670     stm   #((abu_buff_sz/2-fsync_sz)-1), BRC
671     rptb  rawdata_loop - 1
672
673 ;     ld     *AR2+, A ; load (data word) to Acc
674 ;     and   #0xFF, A ; mask to low-byte
675 ;     stl   A, *AR4+ ; save to serial buffer
676
677     mvdd  *AR2+, *AR4+
678
679     mar   *AR1+

```

```

680     cmpr    LT, AR1 ; If we're not done with a bridged data sequence,
681     nop
682     nop
683     xc      2, NTC
684     rsbx    BRAF
685     nop
686     nop
687     nop
688     nop
689     nop
690     nop
691     .global rawdata_loop, dbg_transfer_skip
692 rawdata_loop:
693
694     mvmd    AR1, @bridge_count
695     nop
696
697 end_transfer:
698     .global    end_transfer
699
700     nop
701
702 serial_transfer_end:
703     .global    serial_transfer_end
704     nop
705
706 ; Bit reverse and add start/stop bits
707     ld      @shb_addr, A
708     ld      #(abu_buff_sz/2), B
709
710     call    _serial_cook
711
712 ;     rsbx    INTM
713
714     addm    #1, @minor_count
715
716 ; If a major frame is complete, shut it down
717
718 ; unset acq_seq, keep lsb_sel
719     stm    #lsb_sel, ARO
720     portw  ARO, wr_disc
721
722 ; Strobe watchdog- once per acquisition
723     stm    #0, ARO ; Data is not used- just the wr_dog strobe
724     portw  ARO,wr_dog ; Strobe the watchdog
725
726 ; Stop acquisition, clear interrupts, then idle until an interrupt.

```

```

727     call    rsp_reset
728
729     nop
730     nop
731
732     .global pre_sleep
733 pre_sleep:
734
735     stm    #(acq_test4+lsb_sel), ARO
736     portw ARO, wr_disc
737
738     ssbx  INTM
739     stm    #OFFh, IFR ; Clear any pending interrupts
740
741     idle   2 ; and now...we wait.
742
743     .global post_sleep
744 post_sleep:
745
746 ; check for aux int -> serial monitor
747 ;     bitf    IFR, #int_3
748 ;     cc      inth_3, TC
749
750 ; make sure we had a serial interrupt
751     bitf    IFR, #int_bx
752     bc     pre_sleep, NTC ; stray interrupt, go back to IDLE
753
754     nop
755
756     stm    #int_bx, IFR ; clear int flag
757
758     mvmd  BSPCEO, @bspce_save ; store control extension register in AR6
759
760     bitf    BSPCEO, #bspce_xh
761     bc     xh_skip, NTC
762
763     stm    #(lsb_sel), ARO
764     portw ARO, wr_disc
765
766 xh_skip:
767
768     rpt    #100
769     nop
770
771     stm    #lsb_sel, ARO
772     portw ARO, wr_disc
773

```

```

774     mvdm    @bridge_size, AR0      ; need to copy these
775     mvdm    @bridge_count, AR1     ; to use CMPR
776     nop
777     nop
778     cmpr    LT, AR1      ; If we're not done with a bridged data sequence,
779     bc      bridge_data, TC      ; jump to bridge_data to
780                      ; continue transfers, otherwise...
781     dld     @major_count, A      ; increment major frame counter
782     add     #1, A
783     dst     A, @major_count
784
785     b      major_loop  ; new data acquisition
786
787 ;
788 ; Main acquisiton ('appcode') branch done
789 ;
790
791 ;
792 ; Interrupts
793 ;
794
795 ; Non-Maskable Interrupt
796 ;      this is hit by the watchdog
797 int_nmi:
798     nop
799     nop
800
801     stm     #0, AR0
802     portw   AR0,wr_dog  ; Strobe watchdog timer
803
804     b      0xF800
805
806 ; Setup: IPTR=0x1FF, OVLY=1, all else =0
807 ; This should set things up to completely
808 ; reload the program from the EPROM on reset.
809     stm     #0xFFA0, PMST
810     nop
811     nop
812
813 ;      reset  ; I don't have to take this. ...I'm going home.
814
815     ret  ; should never get here!
816
817 int_h_3:
818     ssbx    INTM
819     stm     #int_3, IFR  ; clear int flag
820

```

```

821      ; call serial monitor
822
823 ;     calld  _RxDSP_Monitor
824 ;     stm    #bspce_haltx, BSPCE ; tell serial transmit to halt
825 ;                           ; after completing this half-buffer
826
827     retd
828     nop
829     nop
830
831 file_header:
832     .string  "Dartmouth College Rx-DSP, AGO Site 3 Unit 0."
833     .word    0000h ; Null terminator
834
835 ;~~~~~
836 ; hwrite
837 ;
838 ; Writes a header:
839 ;
840 ; <0xFE6B2840><RxDSP><Unit #><Ver #><NCOF><MFBC><00000000>
841 ;-----
842
843 hwrite:
844     .global   hwrite
845
846     pshm     AR3
847
848 ; 4-byte sync
849     stm      #static_header, AR3 ; Point to static header words
850     rpt      #static_header_len ; <SYNC><RxDSP>
851     mvdd    *AR3+, *AR4+
852
853     stm      #code_version, AR3
854     rpt      #3
855     mvdd    *AR3+, *AR4+
856
857     stm      #spec_header, AR3 ; Point to static header words
858     rpt      #spec_header_len ; <skipNS><fbstFBSN><fbenFBEN>
859     mvdd    *AR3+, *AR4+
860
861     stm      nco_freq, AR3
862     ld       *AR3, #-8, A
863     and     #0xFF, A
864     stl      A, *AR4+
865     ld       *AR3+, A ; inc to second word
866     and     #0xFF, A
867     stl      A, *AR4+

```

```

868    ld      *AR3, #-8, A
869    and    #0xFF, A
870    stl    A, *AR4+
871    ld      *AR3, A
872    and    #0xFF, A
873    stl    A, *AR4+
874
875    stm    #band_width, AR3
876    rpt    #3
877    mvdd  *AR3+, *AR4+
878
879    stm    major_count, AR3
880    ld     *AR3, #-8, A
881    and    #0xFF, A
882    stl    A, *AR4+
883    ld     *AR3+, A ; inc to second word
884    and    #0xFF, A
885    stl    A, *AR4+
886    ld     *AR3, #-8, A
887    and    #0xFF, A
888    stl    A, *AR4+
889    ld     *AR3, A
890    and    #0xFF, A
891    stl    A, *AR4+
892
893    popm  AR3
894
895    retd
896    nop
897    nop
898
899 static_header:
900     .word    0xFE, 0x6B, 0x28, 0x40
901     .string "AGORxDSP" ; 12 bytes
902 static_header_end:
903     static_header_len    .set    static_header_end-static_header-1
904
905 spec_header:
906     .string  "stride08"
907     .string  "cbst0113"
908     .string  "cben0166"
909     .string  "00000000" ; pad to 32 bytes
910 spec_header_end:
911     spec_header_len    .set    spec_header_end-spec_header-1
912
913     .end

```

File Frame \Byte #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	N-1	N												
1	FS1	FS2	FS3	FS4	FS5	FS6	FC1	FC2	TSs1	TSs2	TSs3	TSs4	TSs5	TSu1	TSu2	TSu3	AD1	AD2	...	ADn-1	ADn												
...													
File Frame Header Size	128 Bits																	FS: 48-bit File Sync bit pattern = 0xb0f919c5025d															
File Frame Data Size	10	11	12	13	14	15	16	17												FC: 16-bit File Frame Counter													
Percent Overhead	1.56	0.78	0.39	0.2	0.1	0.05	0.02	0.01	%											TSs: 40-bit seconds since epoch													
																				TSu: 24-bit microseconds													
																				ADn: serial data bytes													
Header Byte #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	
Data	\xFE	\x6B	\x28	\x40	R	x	D	S	P	Un	S	S	v	1	.	0	N	C	O	F	B	A	N	D	M	F	C	B	0	0	0	0	
	\xFE\x6B\x28\x40RxDSPUnSSv1.0																	NCOFBANDMFCB0000															
	Un: Unit Number in ASCII																	NCOF: Center Frequency BAND: Width of the filter band															
	MFCB: Major Frame Counter Bytes																																
Major Frame																																	
Minor Frame # \Byte #	1	2	3	4	5	6	7	8	...	517	518	519	520	521	522	523	524																
1	S1	S2	C1	C2	H1	H2	H3	H4	...	I121H	I121L	Q121H	Q121L	I122H	I122L	Q122H	Q122L																
2	S1	S2	C1	C2	I123H	I123L	Q123H	Q123L	...		Q66H	Q66L			I128H	I128L	Q128H	Q128L															
3	S1	S2	C1	C2				
4	S1	S2	C1	C2	I449H	I449L	Q449H	Q449L	...	I450L	Q450H	Q450L		I512H	I512L	Q512H	Q512L																

Major Frame has 32-byte header, followed by 2048 bytes of data.
 Minor Frames are determined by 54x BSP half-buffer size, for above major frame size we use 524-byte minor frames.
 Minor frames have 2-byte counters, data, then 2-byte sync words.
 C1: Composite info byte, 2 MSB unit designation, 6 LSB minor frame count
 S1: Sync 1, 0xEB S2: Sync 2, 0x28 |H1...H32: major frame header C2: Second info byte, major frame count
 (Q/I)1(H/L)...512: Data

Figure A.1: Maps of the Sondrestrom MI-ARC data structure, with file structure top, headers middle, and major/minor frame structure bottom.

A.4 Sondrestrom MI-ARC

The Sondrestrom MI-ARC has three sample-synchronized Rx-DSPs. It cycles through a table of center frequencies, returning small frames of data through a single serial line. The data from the three units is combined on the line by a custom hardware multiplexer. The two unique parts of this deployment’s code are an additional RSP function, and the main program code.

A.4.1 Data Structure

Figure A.1 shows the structure of the Sondrestrom data. The data file structure is on top, and file frames include sync words, counters, time data, and the serial data stream (which may be asynchronous to the file’s frame structure). Middle is the structure of the header, containing the major frame sync words, unit number, and frequency data. Finally, on the bottom is the internal structure of the major frames, each composed of four minor frames.

A.4.2 RSP One-Shot

This function modifies one memory location in the AD6620's RAM. The most common use for this is to change the center frequency.

```
1 ;~~~~~  
2 ; rsp_os      Receive Processor One-Shot  
3 ;           Quickly loads one 36-bit value to an address on the AD6620  
4  
5 rsp_os:  
6     ; 36-bit value in A  
7     ; 10-bit address in B  
8  
9     ; write address 0x0303 to address registers  
10    stl    B, #-8, AR4  
11    andm  #0x03, AR4 ; mask to lower two bytes  
12    nop  
13    nop  
14    nop  
15    nop  
16    portw AR4, wr_rx+amr  
17  
18    stl    B, AR4  
19    andm  #0xFF, AR4  
20    nop  
21    nop  
22    nop  
23    nop  
24    portw AR4, wr_rx+lar  
25  
26    ; write 36-bit value into five 8-bit data registers  
27    mvdm  @AG, AR4  
28    andm  #0x0F, AR4 ; 0x0F 0000 0000  
29    nop  
30    nop  
31    nop  
32    nop  
33    portw AR4, wr_rx+dr4  
34  
35    sth    A, #-8, AR4 ; 0x00 FF00 0000  
36    andm  #0xFF, AR4  
37    nop  
38    nop  
39    nop  
40    nop  
41    portw AR4, wr_rx+dr3  
42  
43    sth    A, AR4 ; 0x00 00FF 0000
```

```

44      andm    #0xFF, AR4
45      nop
46      nop
47      nop
48      nop
49      portw  AR4, wr_rx+dr2
50
51      stl     A, #-8, AR4 ; 0x00 0000 FF00
52      andm    #0xFF, AR4
53      nop
54      nop
55      nop
56      nop
57      portw  AR4, wr_rx+dr1
58
59      stl     A, AR4 ; 0x00 0000 00FF
60      andm    #0xFF, AR4
61      nop
62      nop
63      nop
64      nop
65      portw  AR4, wr_rx+dr0 ; writing to dr0 commits
66
67      retd
68      nop
69      nop

```

A.4.3 MI-ARC Main Program Code

This main program code is unique in that it was designed to do away with separate codes and .hex files for the Master and Slave units. Instead, the code's role is entirely determined by the `unit_designation` uword, which can be targeted by the PROM burner, and auto-incremented when PROMs for a number of units are burned in-sequence.

The code contains two helper functions, `cfreq_walk` and `sync_units`, which implement the frequency switching and frame synchronization, respectively. This setup uses the TCLKR and TCLKX lines of the TDM Serial Port (TSP) as feedback inputs from the two Slave units, to help ensure synchronization.

```

1 ; v1.1    2012.06.01    Added 1-frequency debug mode
2 ;                                controlled by trm_28 jumper
3
4 .mmregs
5 .global ZERO, BMAR, PREG, DBMR, INDX, ARCR, TREG1
6 .global TREG2, CBSR1, CBER1, CBSR2, CBER2
7 .global RXDSP_START

```

```

8      .ref      _serial_cook
9      .ref      rsp_clear, rsp_reset, rsp_init
10     .ref      rsp_mstart, rsp_sstart, rsp_freq
11     .ref      transfer, transfer_table_sz
12     .global   bridge_data, buff_clear_loop
13     .def      ago_main, int_nmi
14
15     .include "rx-dsp.h"
16     .text
17
18 find_me           .ulong    0x6B28FE40
19 unit_designation .uword    0 ; Unit number (Master = 0)
20 code_version     .string   "v1.0"
21 station_code    .string   "SS"
22
23 band_width      .string   "0750"
24
25 ; Rotating center frequency table
26 ; 32-bit values derived by mapping the sampling frequency (S) to
27 ; the 0:2^32 range, then taking the ratio of center frequency
28 ; (C) to S, i.e. C/S*2^32
29
30 cfreq_table:
31     .word    0x01d2, 0xf1c9 ; 475 kHz
32     .word    0x04b4, 0x39a7 ; 1225 kHz
33     .word    0x0795, 0x8185 ; 1975 kHz
34     .word    0xa076, 0xc964 ; 2725 kHz
35 cfreq_table_end:
36 cfreq_table_sz   .set      cfreq_table_end-cfreq_table
37
38 cfreq_test      .word    0x0e66, 0x6758 ; 3750 KHz, used when
39                                     ; trm_28 jumper is on
40
41 cfreq_toggle1   .set      0x0999 ; 2500 kHz
42 cfreq_toggle2   .set      0x9A3B
43
44
45 ; Run constants
46 data_n          .set      512 ; Size of each FFT (# of IQ pairs)
47 data_discard    .set      512 ; number of words to discard from
48                                     ; Rx FIFO before taking data
49 data_minor_sz   .set      1  ; number of acquisitions per
50                                     ; half-buffer interrupt
51 abu_buff_sz    .set      1048 ; size of serial buffer
52                                     ; (2x major frame size IN BYTES)
53 fsync_sz        .set      4  ; # of serial frame sync bytes
54                                     ; (should be multiple of 4)

```

```

55
56 ; Memory allocations
57 data_addr      .usect  ".data_v", 0x800, 1, 1
58 stackres       .usect  ".stack_v", 0x40, 1, 1
59 abu_buff_loc   .usect  ".sbuff_v", abu_buff_sz, 1, 1
60 abu_buff_hloc  .set    abu_buff_loc+abu_buff_sz/2 ; half-way
61
62 ; Scratchpad RAM usage
63 bridge_count   .set    scratch
64 minor_count    .set    scratch+1
65 bspce_save     .set    scratch+2
66 bridge_size    .set    scratch+3
67 major_count    .set    scratch+4 ; two words!
68 nco_freq       .set    scratch+6 ; two words!
69 cfreq_tp       .set    scratch+8
70
71     .bss     TempLmem,1*2,0,0 ;temporary dword
72
73
74 RXDSP_START
75 ago_main:
76
77     rsbx    XF
78
79 ; Processor setup
80     ssbx    INTM          ; Disable interrupts
81     stm     #(stackres+0x40), SP ; set Stack Pointer
82     stm     #npmst,PMST ; Set processor mode/status
83 ;     stm #defst0, ST0
84 ;     stm #defst1, ST1
85     rsbx    SXM ; Suppress sign extension
86 ;     rsbx    XF
87     nop      ; Space for branch to app
88     nop
89
90     ssbx    XF
91     rpt     #64
92     nop
93     rsbx    XF
94
95 appcode:
96 ;     stm #0,state      ; Clear interrupt routine state
97     stm #0,AR0          ; Clear auxilliary register 0
98     portw ARO,wr_rs_rx  ; Reset AD6620 RSP
99     portw ARO,wr_disc   ; Enable parallel TLM drivers, I_Q out
100    portw ARO,wr_dog    ; Strobe watchdog timer
101

```

```

102    stm    #0,AR0 ; Clear all auxiliary registers
103    stm    #0,AR1
104    stm    #0,AR2
105    stm    #0,AR3
106    stm    #0,AR4
107    stm    #0,AR5
108    stm    #0,AR6
109    stm    #0,AR7
110
111    stm    #OFFh,IFR ; Clear any pending interrupts
112    stm    #ntss,TCR ; Stop timer, if running
113    .global read_init
114 ; Main data code start
115 read_init:
116    call   rsp_reset
117    call   rsp_init
118    call   rsp_clear
119    call   rsp_mstart
120
121    rpt   #4444 ; Let the AD6620 do its first initialization in peace
122    nop
123
124    call   rsp_reset
125
126    portr  rs_rx_fifo, AR0 ; Reset Rx FIFO - also wired to Slave
127    nop
128    nop
129
130    stm    #lsb_sel, AR0 ; Reset acq_seq
131    portw AR0, wr_disc
132    nop
133
134
135    ; TSP shutoff
136
137 ;    stm    #(tspc_Free+tspc_fsm+tspc_nXrst+tspc_nRrst), TSPC
138    stm    #(tspc_Free+tspc_fsm), TSPC
139
140    ; BSP prep
141
142    stm    #(bspc_Free+bspc_fsm), BSPCO ; reset BSP
143    stm    #(int_bx), IMR ; unmask serial transmit interrupt
144    ; 10-bit words, enable tx autobuffer, halt after first half-buffer
145    stm    #(bspce_fe+bspce_bxe+bspce_haltx), BSPCEO
146    stm    #(abu_buff_loc-0x800), AXR ; where in RAM does
147                                ; the tx buffer start?
148    stm    #(abu_buff_sz), BKX ; buffer size

```

```

149
150 ; Clear entire serial buffer
151     stm      #abu_buff_loc, AR4
152     stm      #abu_buff_sz-1, BRC
153     rptb    buff_init_loop - 1
154
155     st      #0h, *AR4+
156     .global buff_init_loop, head_ramp, major_loop
157 buff_init_loop:
158
159 ; set initial frequency table position
160
161     st      #cfreq_table_sz-2, @cfreq_tp
162
163 ;     rsbx    INTM ; global interrupt enable
164
165 ; All Aux Registers are fungible in the main loop: values which must
166 ; be preserved over time are stored in the scratchpad RAM as defined
167 ; above. Note that only AR6 and AR7 are required to be preserved by
168 ; the DSP Library functions (and most others), all other ARx may be
169 ; modified within function calls.
170
171     ld      #0, A
172     dst     A, @major_count
173
174 ; Set initial bspce_save such that the first acquisition will
175 ; write to the second half of the serial buffer
176     st      #bspce_xh, @bspce_save
177
178 ; boot lag: insert >50 ms delay to allow
179 ; everyone plenty of time to boot up
180     stm      #16384, BRC
181     nop
182     nop
183     rptbd   boot_delay_loop - 1
184     nop
185     nop
186
187     nop
188
189     rpt      #4096
190     nop
191
192     nop
193     nop
194
195 boot_delay_loop:

```

```

196
197     nop
198     nop
199
200 ; now synchronize units for the first time
201     call    sync_units
202
203 ; Start BSP transmits
204     stm    #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0 ; hold fsm bit
205     nop
206     nop
207
208 ;~~~~~
209 ; Main loop
210 ;-----
211
212 major_loop:
213     .global    major_loop
214
215 ; Method:
216 ;
217 ; Master waits for acq_seq_rdy from Slaves, then raises
218 ; acq_seq_out, starting synced acquisition. Note any previous
219 ; acquisition will still be transferring its last half-buffer,
220 ; and the ABU will be set to halt transmissions when that half
221 ; is done. The acq_seq_rdy & out lines are both held high on
222 ; All, until the end of this new acquisition and half-buffer
223 ; fill. Then Master waits for its own ABU haltx, and then for
224 ; !acq_seq_rdy (signaling Slaves have hit ABU haltx), before
225 ; dropping acq_seq_out, signaling time for the next
226 ; synchronized ABU startup.
227
228     call    cfreq_walk
229
230 ; acquisition sync
231     call    sync_units
232
233     portr   rs_rx_fifo, ARO ; Reset Rx FIFO - also wired to Slave
234
235     cmpm    *(unit_designation), #0
236     bc      slave_startup, NTC
237
238 ; if Master (unit 0) insert delay time to allow
239 ; Slaves time to detect acq_seq_out and start up
240     rpt    #128
241     nop
242

```

```

243     call    rsp_mstart ; start digitizing as Master
244
245     b      data_acq_start
246
247 slave_startup:
248
249     call    rsp_sstart ; start digitizing as Slave
250
251 ;
252 ; Reads
253 ;
254 data_acq_start:
255
256     st     #0, @bridge_count ; reset bridged data counter
257     st     #0, @minor_count  ; reset minor frame counter
258
259 ;
260 ; Set wait states for Rx FIFO
261 ;
262     ldm   SWWSR, A
263     ld    #65535, B
264     xor   #7, #swwsr_is, B ; (0b111<<Nset XOR 0d65535) creates bitmask
265     and   B, A           ; mask out bits of interest
266     or    #0, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
267     ; Nwait = 0 means no additional waits
268     stlm  A, SWWSR
269
270     .global  pre_disc, pre_read
271 pre_disc:
272
273 ; loop to read and discard first data out of AD6620
274     stm   #(data_discard), AR2
275     nop
276 rx_discard_loop:
277     ; read only if rx fifo is nonempty
278     portr rd_disc, ARO
279     nop
280     nop
281     bitf   ARO, rx_ef0
282
283     bc    discard_fifo_empty, NTC
284     portr rd_rx_out, AR1 ; read data into AR1
285     mar   *AR2-           ; decrement word counter
286 discard_fifo_empty:
287
288     banz   rx_discard_loop, *AR2
289

```

```

290
291 ; loop to read data from Rx FIFO into RAM
292     stm    #2, ARO
293     stm    #data_addr, AR1 ; set address for first data word
294     stm    #(data_n*2-1), AR2
295 pre_read:
296     nop
297 rx_read_loop:
298     ; read only if rx fifo is nonempty
299     portr   rd_disc, AR3
300     nop
301     nop
302     bitf   AR3, rx_efo
303
304     bc     read_fifo_empty, NTC
305
306     portr   rd_rx_out, *AR1+
307     mar    *AR2- ; decrement word counter
308
309 read_fifo_empty:
310
311     banzd   rx_read_loop, *AR2
312     nop
313     nop
314
315 ;
316 ; Set wait states for other stuff (full 7)
317 ;
318     ldm    SWWSR, A
319     or     #7, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
320     stlm   A, SWWSR
321     nop
322     nop
323
324 ; Stop acquisition
325     call    rsp_reset
326
327     ; drop acq flags
328     stm    #lsb_sel, ARO
329     portw  ARO, wr_disc
330
331 ;
332 ; End data collection, begin serial data handling
333 ;
334     st    #4*data_n, @bridge_size
335
336 ; entry point for bridging data transfers

```

```

337 ; over multiple serial half-buffers
338 bridge_data:
339     .global    bridge_data
340
341     stm      #(lsb_sel), ARO ; send acq_seq, set lsb_sel
342     portw   ARO, wr_disc
343
344     nop
345     nop
346     ; Determine serial buffer position
347     stm      #abu_buff_loc, AR3 ; load buffer base
348
349     bitf    @bspce_save, #bspce_xh ; read XH out of
350                           ; stored BSPCE register
351     nop
352     nop
353     bc      buff_skip, NTC ; if first half _finished_
354                           ; (XH=0, NTC), do nothing
355
356     addm    #(abu_buff_sz/2), @AR3
357     stm      #(lsb_sel), ARO
358     portw   ARO, wr_disc
359
360 buff_skip:
361     nop
362     nop
363     nop
364     nop
365
366 abu_first_half:
367     .global    abu_first_half
368
369     ssbx    INTM
370
371 ; Clear serial buffer half
372     mvmm    AR3, AR4
373     nop
374     rpt     #(abu_buff_sz/2-1)
375     st      #0x00, *AR4+
376
377
378 ; reset AR4 for data copy
379     mvmm    AR3, AR4
380     nop
381     nop
382
383 abu_fill_start:

```

```

384     .global    abu_fill_start
385
386
387 ; two-byte frame sync 0xEB90
388 st    #0xEB, *AR4+
389 st    #0x90, *AR4+
390
391 ; two-byte infofooop
392 ld    *(unit_designation), #6, A ; byte 1, 2 MSB, unit number
393 or    @minor_count, A ; byte 1, 6 LSB, minor frame number
394 stl   A, *AR4+
395 dld   @major_count, A ; byte 2, 8 bit, major frame number
396 and   #0xFF, A
397 stl   A, *AR4+
398
399 dinner_is_ready:
400     .global dinner_is_ready
401
402 ld    #((abu_buff_sz/2-fsync_sz)/2-1), B ; let's stick the
403                                     ; BRC value here...
404
405 ; If haltx is true, we are starting a
406 ; new major frame, so write the header
407 bitf  BSPCEO, #bspce_haltx
408 bc    header_skip, NTC
409
410 call   hwrite ; write out header, uses &
411           ; modified write address in AR4
412 sub    #16, B ; oops, 32 bytes less space in this half buffer
413
414 header_skip:
415     ; copy raw data (words) into serial buffer (bytes)
416 ld    #data_addr, A
417 add   @bridge_count, #-1, A ; div by 2 to increment
418           ; input word-wise
419 stlm  A, AR2 ; store a copy of the data to AR2
420 stlm  B, BRC ; and here's our BRC setup
421
422 mvdm  @bridge_size, AR0 ; goal bridge size
423 mvdm  @bridge_count, AR1 ; current count
424
425 rptb  rawdata_loop - 1
426
427 ld    *AR2, -8, A ; load (data word) >> 8 to Acc
428 and   #0xFF, A     ; mask to low-byte
429 stl   A, *AR4+     ; save to serial buffer
430 mar   *AR1+         ; count bytes

```

```

431
432     ld      *AR2+, A      ; reload and increment
433     and    #0xFF, A      ; mask
434     stl    A, *AR4+      ; save
435     mar    *AR1+        ; count bytes
436
437     cmpr   LT, AR1    ; If we're not done with a bridged data sequence
438     nop
439     nop
440     xc     1, NTC
441     rsbx   BRAF
442     nop
443     nop
444     nop
445     nop
446     nop
447     .global   rawdata_loop, dbg_transfer_skip
448 rawdata_loop:
449
450     nop
451     nop
452
453     mvmd   AR1, @bridge_count
454
455 ; Bit reverse and add start/stop bits
456     ldm    AR3, A
457     ld     #(abu_buff_sz/2), B
458
459     call   _serial_cook
460
461
462 ;     rsbx   INTM
463
464     addm   #1, @minor_count
465     nop
466     nop
467     stm    #(lsb_sel), ARO
468     portw ARO, wr_disc
469
470 ; buffer is now loaded
471     call   rsp_clear ; clear NCO RAM, do it here
472                     ; since we have some free time
473
474 ; Strobe watchdog- once per minor frame cycle
475     stm    #0, ARO      ; Data is not used- just the wr_dog strobe
476     portw ARO,wr_dog  ; Strobe the watchdog
477

```

```

478 ; clear interrupt flags, then idle until an interrupt.
479     .global    pre_sleep
480 pre_sleep:
481
482     ssbx      INTM
483     stm      #0FFh, IFR ; Clear any pending interrupts
484
485     idle     2 ; and now...we wait.
486
487     .global    post_sleep
488 post_sleep:
489
490 ; check for aux int -> serial monitor
491 ;     bitf    IFR, #int_3
492 ;     cc      inth_3, TC
493
494 ; make sure we had a serial interrupt
495     bitf    IFR, #int_bx
496     bc      pre_sleep, NTC ; stray interrupt, go back to IDLE
497     nop
498
499     stm      #int_bx, IFR      ; clear int flag
500
501     mvmd    BSPCEO, @bspce_save ; store control extension register
502
503 ; if this is a new major frame, we need to sync
504 ; everyone up by waiting for all ABU haltx.
505
506     bitf    BSPCEO, #bspce_haltx
507     bc      abu_restart_skip, NTC
508
509 abu_haltx_wait:
510     .global    abu_haltx_wait
511
512 ; ABU has halted. Reset ABU, and synchronize
513     stm      #(bspce_fe+bspce_bxe), BSPCEO ; 10-bit words, enable
514                                     ; tx ABU, disable haltx
515
516     call     sync_units
517
518 ; Start BSP transmits
519     stm      #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0 ; hold fsm bit
520     nop
521     nop
522
523     rpt     #16383
524     nop

```

```

525
526 ; unset acq_seq, keep lsb_sel
527     stm      #lsb_sel, ARO
528     portw    ARO, wr_disc
529
530 abu_restart_skip:
531     .global   abu_restart_skip
532
533     mvdm    @bridge_size, ARO    ; need to copy these
534     mvdm    @bridge_count, AR1   ; to use CMPR
535     nop
536     nop
537     cmpr    LT, AR1           ; If we're not done with
538                           ; a bridged data sequence,
539     bc      bridge_data, TC   ; jump to bridge_data to
540                           ; continue transfers, otherwise...
541
542     dld     @major_count, A    ; increment major frame counter
543     add     #1, A
544     dst     A, @major_count
545
546 ; The final half is transmitting, we want to halt when it finishes.
547     stm     #(lsb_sel), ARO
548     portw  ARO, wr_disc
549     orm    #bspce_haltx, BSPCE0
550     nop
551     nop
552 ;     stm     #(lsb_sel), ARO
553 ;     portw  ARO, wr_disc
554
555     b      major_loop ; new data acquisition
556
557 ;~~~~~
558 ; Main acquisiton ('appcode') branch done
559 ;-----
560
561 ;
562 ; Interrupts
563 ;
564
565 ; Non-Maskable Interrupt
566 ;      this is hit by the watchdog
567 int_nmi:
568
569     nop
570     stm     #npmst, PMST   ; Reset PMST to be sure IPTR -> 0x80
571

```

```

572 ; Alternative: IPTR=0x1FF, OVLY=1, all else =0
573 ; This should set things up to completely reload
574 ; the program from the EPROM on reset
575 ;     stm    #0xFFA0, PMST
576
577     reset ; I don't have to take this. ...I'm going home.
578
579     ret ; should never get here!
580
581
582 int_3:
583     ssbx    INTM
584     stm     #int_3, IFR ; clear int flag
585
586     retd
587     nop
588     nop
589
590 ;~~~~~
591 ; cfreq_walk
592 ;
593 ; walks through the table of center frequencies
594 ; at label #cfreq_table
595 ;
596 ; stores table position @cfreq_tp and current NCO value
597 ; as a 32-bit number @nco_freq
598
599 cfreq_walk:
600
601     portr   rd_disc, ARO
602     nop
603     nop
604     bitf    ARO, #trm_28
605     ; If trm_28 is low (NTC, jumper on), skip the walk.
606     bc      walk_skip, NTC
607
608     addm   #2, @cfreq_tp
609     nop
610     nop
611     cmpm   @cfreq_tp, #cfreq_table_sz
612     nop
613     nop
614     xc    2, TC
615     st    #0, @cfreq_tp
616
617     nop
618     nop

```

```

619
620     ld      #(cfreq_table), A
621     add    @cfreq_tp, A
622     stlm   A, ARO
623
624     b      cfreq_commit
625
626 walk_skip:
627     ; load a single frequency instead
628     stm    #(cfreq_test), ARO
629
630 cfreq_commit:
631
632     nop
633     nop
634
635     ld      *ARO+, B
636     stl    B, @nco_freq
637     sftl   B, #8
638     sftl   B, #8
639     ld      *ARO, A
640     stl    A, @(nco_freq+1)
641     or     B, A
642
643     nop
644     nop
645
646     stm    #acq_ant_1, ARO ; enable antenna 1
647
648 ; frequency is in A. subtract toggle freq and save result to B
649     sub    #cfreq_toggle1, #16, A, B
650     sub    #cfreq_toggle2, B
651
652     xc    2, BGEQ          ; if B>=0, cfreq >= toggle freq
653     stm    #acq_ant_2, ARO ; so enable antenna 2
654     nop
655
656     portw ARO, wr_disc    ; write out antenna toggle lines
657
658     call   rsp_freq
659
660     retd
661     nop
662     nop
663
664 ;~~~~~
665 ; sync_units

```

```

666 ;
667 ; synchronizes master/slave units by
668 ; toggling and waiting for latched lines
669
670 sync_units:
671
672 ; Raise test2 line. On the Master this should do nothing (NC),
673 ; on the Slaves it signals the Master they are ready.
674
675     stm      #(acq_seq_rdy+lsb_sel), ARO
676     portw    ARO, wr_disc
677     nop
678     nop
679
680 ; Check status of TSP, wait for IN1 & IN2 high
681 ready_loop:
682
683     nop
684
685     bitf    TSPC, #tspc_in0
686     bc      ready_loop, NTC
687
688     bitf    TSPC, #tspc_in1
689     bc      ready_loop, NTC
690
691 ; Raise acq_seq_out--on the Master this signals the Slaves
692 ; to start, on the Slaves it does nothing (NC).
693     stm      #(acq_seq_out+acq_seq_rdy+lsb_sel), ARO
694     portw    ARO, wr_disc
695
696     retd
697     nop
698     nop
699
700 static_header:
701     .word    0xFE, 0x6B, 0x28, 0x40
702     .string   "RxDSP"
703 static_header_end:
704
705 static_header_len    .set      static_header_end-static_header-1

```

A.4.4 Sondrestrom Utilities

Below are two short Python utility scripts: the first acquires MI-ARC data from the serial port, saves it to disk, and will parse out major frames and save them separately for real-time

display, if desired. The second is a mostly functional real-time display script which makes use of the Gnuplot.py module.

tridsp-acq.py

```
1 #!/usr/bin/env python
2
3 # MCB 27 Sept. 2013. This code is the same as tridsp-acq.py except
4 # there are new lines (41-43) that power cycle the receiver by
5 # changing the state of the serial DTR line.
6
7 import serial,signal,sys,time
8 from struct import unpack,pack
9 from datetime import datetime
10 import numpy as np
11 #import Gnuplot
12 #from matplotlib import pyplot as plt
13 from optparse import OptionParser
14
15 parser = OptionParser(usage="""tridsp-acq.py: read in triple-DSP multiplexed
   serial data.""")
16
17 parser.set_defaults(verbose=False,port="/dev/ttyS0", ofp="SStridsp",
   nboards=3, majfsync=0xFE6B2840, majfsz=4, minfsz=524, limit=242936,
   filesync=0xb0f919c5025d, acqsz=4080, rtd=True, rtddfile="/tmp/trirtd.data")
18
19 parser.add_option("-o", "--outfile", type="str", dest="ofp", help="Output
   file prefix. [default: %default]")
20 parser.add_option("-p", "--port", type="str", dest="port", help="Serial port
   number. [%default]")
21 parser.add_option("-r", "--rtdd", action="store_true", dest="rtd",
   help="Write out major frames for RTD. [%default]")
22 parser.add_option("-R", "--rtddfile", type="str", dest="rtddfile", help="File
   for RTD data. [%default]")
23 parser.add_option("-n", "--nboards", type="int", dest="nboards",
   help="Number of multiplexed DSP boards. [%default]")
24 parser.add_option("-s", "--maj-sync", type="int", dest="majfsync",
   help="Major frame sync pattern. [%default]")
25 parser.add_option("-z", "--maj-size", type="int", dest="majfsz", help="# of
   Minor frames per Major frame. [%default]")
26 parser.add_option("-Z", "--min-size", type="int", dest="minfsz", help="# of
   bytes per Minor frame. [%default]")
27 parser.add_option("-X", "--limit", type="int", dest="limit", help="Number of
   acquisitions to record. [%default]")
28 parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
   help="print status messages to stdout.")
29
30
31 (o, args) = parser.parse_args()
```

```

32
33 o.majfsync = pack("4B", *[o.majfsync>>i)&0xFF for i in [24,16,8,0]])
34 o.filesync = pack("6B", *[o.filesync>>i)&0xFF for i in [40,32,24,16,8,0]])
35
36 try:
37     inp = serial.Serial(port=o.port, baudrate=115200, bytesize=8,
38                           parity='N', stopbits=1)
39     except serial.SerialException:
40         print "Unable to open serial port {0}.".format(o.port)
41         sys.exit(1)
42
43 #Restore power to receiver by toggling the DTR line
44 time.sleep(5.0)
45 inp.setDTR(False)
46
47 print("Reading from serial port {0}...".format(o.port))
48
49 dtstr = datetime.today().strftime("%Y%m%d-%H%M%S")
50
51 ofn = "{0}-{1}.data".format(o.ofp, dtstr)
52 ofile = open(ofn, "w")
53
54 print("Taking {0} {1}-byte acquisitions ({2} hours).".format(o.limit,
55           o.acqsz, o.limit*o.acqsz/11520/3600))
56
57 if o.rtd:
58     rfile = open(o.rtdfile, 'w')
59
60 print("Writing RTD data to {0}.".format(o.rtdfile))
61
62 framecount = 0
63 bytestr = ""
64 mfbsync = "".join([o.majfsync[i]*o.nboards for i in range(len(o.majfsync))])
65 mfbsize = o.nboards*o.majfsz*o.minsz
66
67 running = True
68
69 acqcount = 0
70
71 while running and acqcount < o.limit:
72     data = inp.read(o.acqsz)
73
74     # build timestamp: 40-bit uint seconds since epoch, 24-bit uint
75     # microseconds
76     timefl = time.time()

```

```

76     timeint = int(timefl)
77     timefrac = int((timefl-timeint)*(1e6))
78     timestr = ( pack(">Q", timeint&0xFFFFFFFFFFFF) [3:] ) + ( pack(">I",
79         timefrac)[1:] )
80
80     ofile.write(o.filesync)
81     ofile.write(pack('>H', framecount&0xFFFF))      # 16-bit counter
82     ofile.write(timestr)
83     ofile.write(data)
84
85     framecount += 1
86
87     # if we want rtd, add to bytestream, and if there's a major frame in
88     # there, write it
89
90     if o.rtd:
91         bytestr += data
92
93         loc = bytestr.find(mfbsync)
94         nloc = bytestr.find(mfbsync, loc+1)
95
96         if (loc >= 0) and (nloc >= loc):
97             rfile.seek(0)
98             rfile.write(bytestr[loc-4 * o.nboards:nloc-4 *
99                 o.nboards])
100            rfile.flush()
101            prelen = len(bytestr)
102            bytestr = bytestr[nloc-4*o.nboards:]
103            postlen = len(bytestr)
104            print("Wrote RTD file. {0} ->
105                  {1}".format(prelen,postlen))
106
106     acqcount += 1
107
108 if o.rtd:
109     rfile.close()
110
111 ofile.close()

```

tridsp-rtd.py

```

1  #!/usr/bin/env python
2  from os.path import getmtime
3  import sys
4  from struct import unpack,pack
5  from datetime import datetime
6  import numpy as np
7  import Gnuplot

```

```

8 #from matplotlib import pyplot as plt
9 from optparse import OptionParser
10 from math import ceil, floor
11
12 parser = OptionParser(usage="""tridsp-acq.py: read in triple-DSP multiplexed
   serial data.""")
13
14 parser.set_defaults(verbose=False, rfile="/tmp/trirtd.data", nboards=3,
15                         majfsync=0xFE6B2840,
16                         majfsz=4,
17                         minfsync=0xEB90, minfsz=524,
18                         dataplot=True)
19
20 parser.add_option("-r", "--rfile", type="str", dest="rfile", help="RTD data
   file. [default: %default]")
21 parser.add_option("-n", "--nboards", type="int", dest="nboards",
   help="Number of multiplexed DSP boards. [%default]")
22 parser.add_option("-s", "--maj-sync", type="int", dest="majfsync",
   help="Major frame sync pattern. [%default]")
23 parser.add_option("-z", "--maj-size", type="int", dest="majfsz", help="# of
   Minor frames per Major frame. [%default]")
24 parser.add_option("-S", "--min-sync", type="int", dest="minfsync",
   help="Minor frame sync pattern. [%default]")
25 parser.add_option("-Z", "--min-size", type="int", dest="minfsz", help="# of
   bytes per Minor frame. [%default]")
26 parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
   help="print status messages to stdout.")
27 parser.add_option("-c", "--channel", action = "store", type = 'int',
   dest="chan_num", help = "channel number to display")
28 parser.add_option("-f", "--freq", action = "store_true", dest = "spectra")
29 parser.add_option("-b", "--band", action = "store", dest = "bw",
   type="float")
30 (o, args) = parser.parse_args()
31 def b2iq(indat):
32     # input : binary string containing 16-bit big-endian signed I/Q data
33     # output: [[I array], [Q array]]
34
35     num = unpack(">"+str(len(indat)/2)+"h", indat)
36     return np.reshape(num, (2,-1), "F")
37
38 cplotd = []
39 blist = range(o.nboards)
40
41 infile = open(o.rfile, 'r')
42 freqplot=Gnuplot.Gnuplot()
43 #freqplot("set title \"Unit {0}\\"".format(k,cfreq))
44 freqplot("set term x11 noraise")

```

```

44 freqplot("set yrang [0:140]")
45 freqplot("set xrange [100:3100]")
46 freqplot("set style data lines")
47 #freqplot("set title \"\{0\} KHz\"".format(cfreq))
48
49 oldtime = 0
50 first_loop = 0
51 plots= []
52
53 running = True
54
55 window = np.hanning(512)
56
57 while running:
58
59     while True:
60         newtime = getmtime(o.rf)
61         if newtime != oldtime:
62             break
63
64     oldtime = newtime
65     badness = False
66
67     infile.seek(0)
68     data = infile.read()
69
70     # parse major frames and plot
71     unitdata = [[]]*o.nboards
72
73     for i in blist:
74         bdata = data[i::o.nboards]
75         majfr = "".join([bdata[j*o.majfsz:(j+1)*o.majfsz] for j in
76                         range(o.majfsz)]) # extract major frame
77         minsyncs = "".join([bdata[j*o.minfSz:(j+1)*o.minfSz] for j in
78                           range(o.majfsz)]) #extract minor frame syncs
79
80         minunit = [(x & 0xC0)>>6 for x in unpack(">4B",
81                                                       minsyncs[2::4])]
82         if len(np.unique(minunit)) != 1:
83             print("Minor frame Unit number mismatch.")
84             badness = True
85
86         minmajN = unpack(">4B", minsyncs[3::4])
87         if len(np.unique(minmajN)) != 1:
88             print("Minor frames' major frame # mismatched.")
89             badness = True

```

```

88     unit = unpack("B", majfr[9])[0]-0x30
89     if unit != np.unique(minunit)[0]:
90         print("Major frame doesn't match minor frame Unit
91             #. ")
92         badness = True
93
94     majN = unpack(">I", majfr[24:28])[0]&0xFF
95     if majN&0xFF != np.unique(minmajN)[0]: # mask to one-byte
96         to test against minor frame #
97         print("Major frame # doesn't match minor frames'
98             major frame #. ")
99         badness = True
100
101     # looks like a good major frame
102     cfreq = int(round(unpack(">I",
103         majfr[16:20])[0]/2.0**32*66666.6))
104     nums = unpack(">" + str(len(majfr)/2-16) + "h", majfr[32:])
105
106     unitdata[unit] = { 'cfreq' : cfreq,
107                         'eyes' : nums[::-2],
108                         'ques' : nums[1::2] }
109
110     if len(np.unique([ x['cfreq'] for x in unitdata ])) != 1:
111         print("Center frequency mismatch!")
112
113     if badness:
114         continue
115
116     cfreq = unitdata[0]['cfreq']
117
118     if o.spectra:
119         k = o.chan_num
120         dfdb = o.bw/512.0
121         freqlist = [cfreq-(o.bw/2.0) + dfdb * i for i in range(512)]
122         eyes = unitdata[k]['eyes']
123         ques = unitdata[k]['ques']
124         fftdata = [eyes[j] + ques[j] * 1j for j in range(len(eyes))]
125
126         spec = [10.0*y for y in np.log10([abs(x)**2 for x in
127             np.fft.fft(window*fftdata,n=512,)])] # power spectra
128         spec = spec[len(spec)/2:] + spec[:len(spec)/2] # swap from
129             normal order
130
131         if ((cfreq == 475) or (cfreq == 3750)) and (len(plots) > 0):
132             freqplot.plot(*plots)
133             plots = []
134             plots.append(Gnuplot.Data(freqlist, spec))

```
