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. Wit this the 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`

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

```

ASMFLAG = -v542
LNKFLAG = -w -a -r

CGTDIR = /cygdrive/c/TI54xCGT/bin

BASESRC = int_table.asm bl549.asm ad6620.asm tablemax.asm
FUNCSRC = $(BASESRC) Cbrev32.asm c512.asm log_10.asm hannwin.asm sercook.asm
    cfft_32.asm dpsm.asm scale.asm
TABLES = hann_q15.tab

OBJECTS = $(MAINSRC:.asm=.obj) $(FUNCSRC:.asm=.obj) $(TABLES:.tab=.obj)
LSTS = $(FUNCSRC:.asm=.lst) $(MAINSRC:.asm=.lst)
ABSS = $(FUNCSRC:.asm=.abs) $(MAINSRC:.asm=.abs)

OUTMAP = $(MAINSRC:.asm=-link.map)
HEXMAP = $(MAINSRC:.asm=-hex.map)

OUTFILE = $(MAINSRC:.asm=.out)
HEXFILE = $(MAINSRC:.asm=.hex)

.SUFFIXES: .asm .obj .abs .lst .hex .out .tab

all: $(HEXFILE) $(LSTS)

$(HEXFILE): $(OUTFILE) $(CFPREFIX)hex.cmd
    $(CGTDIR)/hex500.exe $(HEXFLAG) $(CFPREFIX)hex.cmd -map $(HEXMAP) -o
        $(HEXFILE) -i $(OUTFILE)

.abs.lst: $(ABSS)
    $(CGTDIR)/asm500.exe $(ASMFLAG) -x -a $<

.obj.abs: $(OUTFILE)
    $(CGTDIR)/abs500.exe $(OUTFILE)

$(OUTFILE): $(OBJECTS) $(CFPREFIX)link.cmd rx-dsp.h int_table.h ad6620.asm
    $(CGTDIR)/lnk500.exe $(LNKFLAG) $(CFPREFIX)link.cmd -m $(OUTMAP) -o
        $(OUTFILE) $(OBJECTS)

.tab.obj:
    $(CGTDIR)/asm500.exe $(ASMFLAG) $< $@

.asm.obj:
    $(CGTDIR)/asm500.exe $(ASMFLAG) $< $@

clean:
    rm -f $(OBJECTS) $(LSTS) $(ABSS) $(OUTFILE) $(HEXFILE) $(OUTMAP)
```

```
        $(HEXMAP)
```

twostagehex.cmd

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

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

ROMS {
    EPROM1: origin=0x0000, length=0x8000, memwidth=8, romwidth=8
}

/*SECTIONS {
    .bl549 = boot
    .vectors
    .cbrev_p
    .cfft_p
    .log10_p
    .smon_p
    .sintab
}*/
```

twostagelink.cmd

```
/* TMS320C542 DSP Board Boot Rom Linker Command File */

-e RXDSP_START

MEMORY {
PAGE 0:
    INTR_TABLE (RWX): origin = 0x0080, length = 0x0080
    PROG_ANNEX (RIX): origin = 0x0180, length = 0x0680
    PROG_MAIN (RIX) : origin = 0x0C00, length = 0x1200

PAGE 1:
    STACK (RW)      : origin = 0x0100, length = 0x0040
    TEMP_DATA (RW)  : origin = 0x0140, length = 0x0040
    SBUFFER (RW)    : origin = 0x0800, length = 0x0400
    SCALES (RW)     : origin = 0x1E00, length = 0x0200
    DATA (RW)       : origin = 0x2000, length = 0x0800
}

SECTIONS {
    .bl549       : load > PROG_ANNEX, align = 64
    .vectors     : load > INTR_TABLE
    .text        : load > (RIX)
    .cbrev_p     : load > (RIX)
    .cfft_p      : load > (RIX)
    .log10_p     : load > (RIX)
    .smon_p      : load > (RIX)
    .smon_msg    : load > (RIX)
    .sine_tab    : load > (RI)
    .hann_tab    : load > (RI)
    .hann_p      : load > (RIX)
    .sercook_p   : load > (RIX)
    .ad6620      : load > (RIX)
    .transfer_p  : load > (RIX)
    .dpsm_p      : load > (RIX)
    .scale_p     : load > (RIX)

    /* data sections */
    .bss        : > TEMP_DATA
    .stack_v    : > STACK
    .sbuff_v    : > SBUFFER
    .scale_v    : > SCALES
    .data_v     : > DATA
}
```

bl549.asm

```
;************************************************************
;*** Bootloader software version NO. : 1.0 ***
;*** Last revision date : 10/23/1996 ***
;*** Author : J. Chyan ***
;************************************************************
;** **
;** Boot Loader Program **
;** **
;** This code segment sets up and executes boot loader **
;** code based upon data saved in data memory **
;** **
;** WRITTEN BY: Jason Chyan **
;** DATE: 06/06/96 **
;** **
;** Revision History Omitted **
;************************************************************

```

```
;************************************************************
;.title ""bootc54LP
;**********************************************************
; symbol definitions
;**********************************************************
.mnolist

; Let's use some scratchpad memory!  Woo!
brs       .set    60h ; boot routine select (configuration word)
xentry    .set    61h ; XPC of entry point
entry     .set    62h ; entry point
hbyte     .set    63h ; high byte of -8bit serial word
p8word    .set    64h ; concatenator for -8bit memory load
src       .set    65h ; source address
dest      .set    66h ; destination address (dmov from above)
lngth     .set    67h ; code length
temp0     .set    68h ; temporary register0
temp1     .set    69h ; temporary register1
temp2     .set    6ah ; temporary register2
temp3     .set    6bh ; temporary register3
nmintv    .set    6ch ; -nonmaskable interrupt vector
sp_ifr    .set    6dh ; SP IFR temp reg
; MMR definition for c54xlp CPU register
;**
ifr      .set     01h
st0      .set     06h
st1      .set     07h
AL       .set     08h
AH       .set     09h
AG       .set     0Ah
brc      .set     1ah
pmst     .set     1dh
swwsr    .set     28h
bscr     .set     29h


;* * * * * * * * * * * * * * * * * * * * * * * * * *
;*    Bootload from -8bit memory, MS byte first    *
;* * * * * * * * * * * * * * * * * * * * * * * * * *

    .global    BOOTLOAD_START, blskipskip, xfr08, par08_1, endboot
    .ref       RXDSP_START
    .sect      ".bl549"
entry_point      .set    RXDSP_START
eprom_base       .set    0x8000
bl_loadpoint     .set    BOOTLOAD_START

```

```
BOOTLOAD_START
par08
    stm    #0x7FFF,swwsr    ; set full wait states
    stm    #0x0002,bscr     ; bus holder enabled
    ld     #0, DP
    nop
    nop

    st     #entry_point, @entry

    stm    #eprom_base, AR1

par08_1  ; Main section load loop

    nop
    ld     *ar1+, 8, a  ; get address of destination
    and    #0ff00h,a    ; force AG, AH to zero for correct calculation
                        ; of the -23bit destination address. (10/14/99 BCT)
    mvdk    *ar1+, ar3     ; ar3    <-- junkbyte.low byte
    andm    #0ffh, @ar3    ; ar3    <-- low byte
    or      @ar3, a        ; acc A  <-- high byte.low byte
    stlm    a,ar2          ; ar2    <-- destination address

    bc    endboot,aeq  ; section dest = 0 indicates boot end

    ld      *ar1+, 8, a  ; get number of 16-bit words
    and     #0xFF00,a    ; Clear the guard bits and keep low accum (1.92)
    mvdk    *ar1+, ar3   ; ar3    <-- junkbyte.low byte
    andm    #0ffh, @ar3  ; ar3    <-- low byte
    or      @ar3, a      ; acc A  <-- high byte.low byte

    cmpm    AR2, #bl_loadpoint  ; check if our dest is the bootloader load
       address
    bc      blskipskip, NTC     ; if not, keep loading

    add     #1, A   ; if it is the bootloader, we want to skip
    stlm    A, AR0  ; this section, i.e. skip A+1 words
    nop
    add     AR0, A  ; but wait,
    stlm    A, AR0  ; A+1 words = 2(A+1) addresses (8-bit prom)
    nop
    bd      par08_1
    mar     *AR1+0
    nop

blskipskip:

    stlm    a, brc      ; update block repeat counter register
    nop
    rptb    xfr08 - 1   ; block repeat to load section data

    ; load program code word
    ld      *ar1+, 8, a   ; acc A  <-- high byte
    and     #0xFF00, a
    mvdk    *ar1+, ar3    ; ar3    <-- junkbyte.low byte
    andm    #0ffh, @ar3   ; ar3    <-- low byte
    or      @ar3, a       ; acc A  <-- high byte.low byte
    stl     a, @p8word

    ; recover destination address, pause, then write and increment
    ldu      @ar2, a
    nop
    nop
    writa    @p8word
    add      #1, a
    stlm     a, ar2

xfr08:  ; end block repeat

    b    par08_1  ; end section loop

;**
;*    End 549 8-bit EPROM bootloader
;**

endboot
    ldu    @entry, a   ; branch to the entry point
    nop
    nop
    baccd  a
    nop
    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

```
;```````
; AD6620 setup functions and tables
```

```
;_______

    .mmregs
    .def      rsp_clear, rsp_reset, rsp_init, rsp_mstart, rsp_sstart
    .include  "rx-dsp.h"
    .sect     ".ad6620"

;
; rsp_reset, rsp_init, rsp_mstart, rsp_sstart
;     shell functions over rsp_setup
;

rsp_reset:
    ld      #ad6620_soft_reset, A   ; Put AD6620 into reset
    call    rsp_setup

    retd
    nop
    nop

rsp_init:
    ld      #ad6620_filter, A   ; Set up AD6620 filter
    call    rsp_setup

    retd
    nop
    nop

rsp_mstart:
    ld      #ad6620_master_run, A   ; Start digitizing as master
    call    rsp_setup

    retd
    nop
    nop

rsp_sstart:
    ld      #ad6620_slave_run, A   ; Start digitizing as slave
    call    rsp_setup

    retd
    nop
    nop

;
; rsp_clear    Function to clear RCF Data RAM between frames
;
```

```

rsp_clear:
    stm      #100000001b, AR3
    nop
    nop
    portw    AR3, (wr_rx+amr)  ; Load high and low address registers:
    stm      #0, AR3
    nop
    nop
    portw    AR3, (wr_rx+lar)  ; write to 0x100, auto-increment

    stm      #0xFF-1, BRC
    nop
    nop
    rptb     rsp_clear_loop - 1

    portw    AR3, (wr_rx+dr4)
    portw    AR3, (wr_rx+dr3)
    portw    AR3, (wr_rx+dr2)
    portw    AR3, (wr_rx+dr1)
    portw    AR3, (wr_rx+dr0)
    nop

rsp_clear_loop:

    retd
    nop
    nop

; Load RSP (AD6620) Registers from Table
;
; 23 Dec 2009 took out most writes to terminal (msgout, dis4hex, asx)
;
; This code was taken directly from the "rspmod" routine used in the
; Dartmouth Monitor. Instead of having the user enter the data words
; or receiving them from a "script" file, this routine looks at a
; table of words in memory, reads them, and transfers them to the
; AD6620. Used to load control bytes and filter coefficients.
;
; Table entry format:
;
; rsp_table:
;     .word    AmLah, r4r3h, r2r1h, r0xxh;
;     .word    (more 4-word entries)
;     .word    0FFFFh    ; End of table
;
; 4 words:
```

```
; AAaah = AD6620 internal address, 0000h to 030Dh,
;     or FFFFh to terminate.
; AMR = Ma Address mode register
; LAR = La Lower address register
; r4r3h, r2r1h, r0xxh = data bytes, packed into words, MS, to LS.
;     Bottom byte of 3rd word not used (xx). Data is treated a 40
;     bits for all AD6620 registers. Not the most compact
;     arrangement for storage, but readable- and it can be edited
;     directly from monitor scripts.
;
; DR4 = r4
; DR3 = r3
; DR2 = r2
; DR1 = r1
; DR0 = r0
;
; Uses:
; A:   Holds table pointer upon entry
; B:   Working register
; AR0: I/O address
; AR2: Table index
; AR3: Holds data to send to or read from I/O port

table_end .set 0FFFFh        ; End-of-table definition

rsp_setup:             ; Enter with table starting address in A
     stlm    A,AR2     ; Save to AR2 for later use
     nop               ; Necessary for loop to execute properly (?!&)
     nop               ; Necessary for loop to execute properly (?!&)
;    ld      #ad6620_msg1,A  ; Tell operator what is happening
;    call    msgout

rsp_loop:
;    ldm     AR2,A     ; Retrieve table index
;    call    dis4hex   ; Display index of table line
;    ld      #0020h,A  ; Space over on screen
;    call    asx

    cmpm    *AR2,#table_end  ; Is this the end of the table?
    bc    rspx,TC            ; Return if at end

    ld    *AR2+,A  ; Get first table word: AD6620 address
    ld    A,B      ; Save a copy

;    call    dis4hex  ; Display

;
```

```
; Transfer RSP register address bytes to
; AD6620 high and low address registers
;
    sftl    A,-8,A      ; Shift high byte to low byte
    and     #0003h,A,A  ; Mask high byte to 2 LSBs (avoid reserved bits
                        ; and do not auto-increment for now)
    stlm     A,AR3          ; Move to AR3 for portw
    portw    AR3,wr_rx+amr  ; Write to high address register
    ld       B,A          ; Get RSP register address
    and      #00FFh,A,A  ; Mask to low byte only (actually hardware
                         ; only uses bits 7:0 of data bus, should not
                         ; need to mask)
    stlm     A,AR3
    portw    AR3,wr_rx+lar  ; Write to low address register

;     ld       #0020h,A   ; Space over
;     call     asx

    ld    *AR2+,A   ; Get next table word (dr4 and dr3 bytes)
    ld    A,B       ; Save a copy

;     call     dis4hex        ; Display

    sftl     A,-8,A  ; Shift high byte to low byte
    stlm     A,AR3   ; AR3 holds output data
    portw    AR3,wr_rx+dr4  ; Store to AD6620 MS data byte register

    ld       B,A          ; Get copy
    and      #00FFh,A,A  ; Mask to low byte only
    stlm     A,AR3
    portw    AR3,wr_rx+dr3

;     ld       #0020h,A   ; Space over
;     call     asx

    ld    *AR2+,A   ; Get next table word (dr2 and dr1 bytes)
    ld    A,B       ; Save a copy

;     call     dis4hex  ; Display

    sftl     A,-8,A  ; Shift high byte to low byte
    stlm     A,AR3
    portw    AR3,wr_rx+dr2

    ld       B,A          ; Get copy
    and      #00FFh,A,A  ; Mask to low byte only
    stlm     A,AR3
    portw    AR3,wr_rx+dr1

;    ld      #0020h,A    ; Space over
;    call    asx

    ld    *AR2+,A      ; Get next table word (dr0 in upper byte)

;    call    dis4hex   ; Display

    sftl    A,-8,A  ; Shift high byte to low byte
    stlm    A, AR3  ; Save for subsequent output port write
    portw   AR3,wr_rx+dr0  ; Address for RSP LS data byte

;    ld    #0020h,A    ; Space over
;    call    asx
;    ld      #000Dh,A  ; Output CR
;    call    asx
;    ld      #000Ah,A  ; LF
;    call    asx

    b    rsp_loop  ; Go back for next table entry
rspx:
;    ld      #ad6620_msg2,A
;    call    msgout
    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.

```
;```````
; Prescaling functions by Micah P. Dombrowski
;
; _sqmag_prescale
;
; Used on 32-bit complex number array (stored RIRIRI), finds the
; largest possible shift applicable to each RI pair using EXP.
; Assumes a zero return equates to EXP(0), and stores the maximum
; shift.  Stores 2*shift in scale factor array.
;
; _log_prescale
```

```
;
; For TI DSP Library Logarithm: normalizes each 32-bit value using
; EXP and NORM, adding shift values to existing values in the save
; array, and cutting to 16-bit output.
;
; _descale
;
; Adjusts logarithmic output based on scale factor array, by
; subtracting scale*log10(2).
;_______

    .mmregs

; Stack usage
; 0 = ST1, 1 = ST0, 2 = function return pointer
    .asg    *SP(3), idata
    .asg    *SP(4), odata
    .asg    *SP(5), sdata

    .def    _sqmag_prescale, _log_prescale, _descale
    .sect   .scale_p

;```````
; _sqmag_prescale
;
; Inputs:       N, number of values to scale in Acc,
; Top of Stack: data input address (512x2 words present),
;               data output address (512 words free),
;               scale save address (512 words free)

_sqmag_prescale

; Set up processor for signed, non-fractional math
    pshm    ST0
    pshm    ST1
    ssbx    CPL
    rsbx    FRCT
    ssbx    SXM
    ssbx    OVM
    rsbx    C16
    nop
    nop

    sub     #1, A   ; BRC = N-1
    stlm    A, BRC
    stm     #16, AR0    ; max shift value
    mvdk    idata, AR2  ; input pointer
```

```
    mvdk    odata, AR3  ; output pointer
    mvdk    sdata, AR4  ; scale array pointer

    rptb    sqmag_prescale_loop - 1

    dld     *AR2+, A
    dld     *AR2-, B
    nop
    nop

    exp     A
    nop
    ldm     T, A

    exp     B
    nop
    ldm     T, B

    min     A       ; A = min(A,B)
    nop
    nop

    sub     #4, A   ; 4 guard bits


    stlm    A, T   ; re-store to T
    nop
    nop

    pshm    T   ; save T to stack

    dld     *AR2+, A
    dld     *AR2+, B
    nop
    nop

    norm    A   ; shift
    norm    B

    .global norm_ovm
norm_ovm:

    dst     A, *AR3+   ; save data
    dst     B, *AR3+

    ld      #0, A   ; clear Acc
    popm    AL      ; pop the corrected scale factor into low Acc
    stl     A, 1, *AR4+  ; save with a 1-bit shift (mpy by 2)

sqmag_prescale_loop:

    popm    ST1
    popm    ST0

    retd
    nop
    nop


;```````
; _log_prescale
;
; Inputs:        N, number of values to scale in Acc,
; Top of Stack:  data input address (512x2 words present),
;                data output address (512 words free),
;                scale save address (512 words free)

_log_prescale

; Set up processor for fractional, signed math
    pshm    ST0
    pshm    ST1
    ssbx    CPL
    ssbx    FRCT
    ssbx    SXM
    ssbx    OVM
    rsbx    C16
    nop
    nop

    sub     #1, A       ; BRC = N-1
    stlm    A, BRC
    stm     #16, AR0    ; max shift value
    mvdk    idata, AR2  ; input pointer
    mvdk    odata, AR3  ; output pointer
    mvdk    sdata, AR4  ; scale array pointer

    rptb    log_prescale_loop - 1

    dld     *AR2+, A
    exp     A
    nop

    ldm     T, B      ; load T
```

```
    sub     #4, B     ; guard bits
    stlm    B, T
    add     *AR4, B   ; add any existing scale factor
    stl     B, *AR4+  ; save back to scale array
    nop
    nop

    norm    A  ; shift

    sth     A, *AR3+  ; save data

log_prescale_loop:

    popm    ST1
    popm    ST0

    retd
    nop
    nop


;````````
; _descale
;
; Inputs:       N, number of input points, in Acc
; Top of Stack: data input address (512x2 words present, Q16.15 format)
;               data output address    (512 words free, in-place okay)
;               scale factor array (512 words present)

log10o32767    .set    0x783F  ; ( log10(32767) * 2^15) >> 3
log10o2        .set    0x04D1  ; ( log10(2) * 2^15 ) >> 3


_descale:

    pshm    ST0
    pshm    ST1
    ssbx    CPL
    rsbx    FRCT
    ssbx    SXM
    rsbx    OVM
    rsbx    C16
    nop
    nop

    sub    #1, A    ; BRC = N-1
    stlm    A, BRC
```

```
    mvdk    idata, AR2    ; input pointer
    mvdk    odata, AR3    ; output pointer
    mvdk    sdata, AR4    ; scale array pointer

    rptb    descale_loop - 1

    ; Docs say log10 outputs Q16.15, but this is misleading,
    ; format is S IIII IIII IIII IIII FFFF FFFF FFFF FFF
    dld     *AR2+, A
    sfta    A, #-3

    ld      #log10o32767, B
    add     B, A

    ld      #log10o2, B

    rpt     *AR4+
    sub     B, A
    sfta    A, #8
    sfta    A, #8
    sat     A

    sth     A, *AR3+

descale_loop:

    popm    ST1
    popm    ST0

    retd
    nop
    nop


    .end
```

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

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

```

```
    .mmregs
    .def    _sqmag
    .sect   .dpsm_p
_sqmag

    pshm    ST0
    pshm    ST1
    ssbx    SXM
    ssbx    FRCT
    ssbx    OVM
    rsbx    C16
    nop
    nop

; Double-precision square magnitude, saving MSB half of result.

    stm     #0, T    ; Multiplication Temp register (for mpy)
    stm     #0, BK   ; Circuluar addressing modulus (do not want)
    sub     #1, B
    stlm    B, BRC
    stm     #2, AR0  ; Increment (jump to next 32-bit datum)
    stlm    A, AR2   ; Load index
    stlm    A, AR3   ; Load index
    stlm    A, AR4   ; Storage index
    rptb    sqmag_loop - 1

    mpy     *AR2+, A              ; a = 0       (1)
    macsu   *AR2-, *AR3+, A       ; a = RL*RH   (1)
    macsu   *AR3-, *AR2, A        ; a += RH*RL  (1)
    ld      A, -16, A             ; a >>= 16    (1)
    mac     *AR2+0%, *AR3+0%, A ; a += RH*RH  (1)
    stm     #0, T  ; (2)
    sat     A      ; (1)

    mpy     *AR2+, B              ; b = 0       (1)
    macsu   *AR2-, *AR3+, B       ; b = IL*IH   (1)
    macsu   *AR3-, *AR2, B        ; b += IH*IL  (1)
    ld      B, -16, B             ; b >>= 16    (1)
    mac     *AR2+0%, *AR3+0%, B ; b += IH*IH  (1)
    stm     #0, T  ; (2)
    sat     B      ; (1)


    add     B, A     ; a += b == R^2 + I^2
    sat     A
    dst     A, *AR4+ ; (1)

```

```
sqmag_loop:

    popm    ST1
    popm    ST0

    nop
    nop

    retd
    nop
    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.

```
;```````
; Frequency selection and averaging function by
; Nathan Utterback and Micah P. Dombrowski
;
; Inputs: start address of data Acc,
;         output address in Bcc

avg_shift_val    .set 3 ; bits to right shift by after summing

    .mmregs
    .def     transfer, transfer_table_sz
    .sect    .transfer_p

    .bss     Delta,1,0,0  ; storage for repeat counter
    .bss     nShift,1,0,0 ; storage for shift value

transfer:

     pshm    ST0
     pshm    ST1

     pshm    AR6

     stlm    A, AR2
     stlm    B, AR3
     stm     transfer_table_start, AR5 ; load the start of the
                                       ; table into memory
     stm     transfer_table_end-1, AR0


transfer_sum_loop:
```

```
        ldm     AR2, A    ; load base address
        add     *AR5+, A  ; add offset from table, inc
        stlm    A, AR4    ; store

        ld      *AR5-, A  ; load next offset, dec
        sub     *AR5+, A  ; subtract current offset to get delta, inc
        sub     #1, A
        stlm    A, BRC    ; store delta-1 for rpt

        ld      #0, B
        rptb    max_loop - 1

        ld      *AR4+, A
        max     B

max_loop:

        stl     B, *AR3+  ; save

        cmpr    LT, AR5
        bc      transfer_sum_loop, TC  ; loop until we finish the table

        popm    AR6

        popm    ST1
        popm    ST0

        retd
        nop
        nop

transfer_table_start:
       .word    25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105

       .word    113, 114, 115, 116, 117, 118, 119, 120, 121
       .word    122, 123, 124, 125, 126, 127, 128, 129, 130
       .word    131, 132, 133, 134, 135, 136, 137, 138, 139
       .word    140, 141, 142, 143, 144, 145, 146, 147, 148
       .word    149, 150, 151, 152, 153, 154, 155, 156, 157
       .word    158, 159, 160, 161, 162, 163, 164, 165

       .word    166, 174, 182, 190, 198, 206, 214, 222, 230
       .word    238, 246, 254, 262, 270, 278, 286, 294, 302
       .word    310, 318, 326, 334, 342, 350, 358, 366, 374
       .word    382, 390, 398, 406, 414, 422, 430, 438, 446
       .word    454, 462, 470, 478
```

```
transfer_table_end:

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.

```
;```````
; Serial data cooking function by Micah P. Dombrowski
;
; Reads N words containing right-aligned bytes,
; bit reverses, and adds start and stop bits.
;
; Inputs: data address in A, number of bytes in B


    .mmregs
    .def    _serial_cook
    .sect    .sercook_p
_serial_cook

    sub    #1, B
    stlm    B, BRC
    stlm    A, AR0
    rptb    bitrev_loop - 1

    ssbx   XF

    ld     #1, B     ; zero result + stop bit

    ld     #001h, A  ; load mask
    and    *AR0, A   ; mask data
    or     A, 8, B   ; OR into result
    ld     #002h, A
    and    *AR0, A
    or     A, 6, B
    ld     #004h, A
    and    *AR0, A
    or     A, 4, B
    ld     #008h, A
    and    *AR0, A
    or     A, 2, B
    ld     #010h, A
    and    *AR0, A
    or     A, 0, B
    ld     #020h, A
    and    *AR0, A
```

```
    or      A, -2, B
    ld      #040h, A
    and     *AR0, A
    or      A, -4, B
    ld      #080h, A
    and     *AR0, A
    or      A, -6, B

    stl     B, *AR0+    ; rewrite to serial buffer

    rsbx    XF

bitrev_loop:

    retd
    nop
    nop

    .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.

```
;```````
;
; Dartmouth College AGO Rx-DSP Program
;
; Written by: Micah P. Dombrowski and Nathan B. Utterback
;     w/ code segments from TI DSP Library
;
;_______

    .mmregs
    .global ZERO, BMAR, PREG, DBMR, INDX, ARCR, TREG1
    .global TREG2, CBSR1, CBER1, CBSR2, CBER2
    .global RXDSP_START
    .ref    _cbrev32, _cfft32_512, _log_10
    .ref    _hann_window, _sqmag, _serial_cook
    .ref    _log_prescale, _sqmag_prescale, _descale
    .ref    rsp_clear, rsp_reset, rsp_init, rsp_mstart, rsp_sstart
    .ref    transfer, transfer_table_sz
    .global bridge_data, buff_clear_loop
    .def    ago_main, int_nmi

```

```
    .include "rx-dsp.h"
    .text

code_version  .string  "v1.0"
band_width    .string  "0300"

; Output constants

output_shift_n    .set  8       ; left shift before 8-bit mask defines
header_freq_mask  .set  0x0FFF  ; bits of 32-bit major frame counter
                                ; that must be zero for a header frame

; Run constants
fft_scaling    .set  0
data_n         .set  512  ; Size of each FFT (# of IQ pairs)
data_discard   .set  512  ; words discarded from Rx FIFO pre-data
data_minor_sz  .set  1    ; acquisitions per half-buffer interrupt
fsync_sz       .set  4    ; # of serial frame sync bytes
                          ; (should be multiple of 4)
abu_buff_sz    .set  214  ; size of serial buffer
                          ; (2x major frame size IN BYTES)
                          ; should be set to
                          ; 2*(transfer_table_sz + fsync_sz)

; Memory allocations
data_addr      .usect  ".data_v", 0x800, 1, 1
scale_addr     .usect  ".scale_v", 0x200, 1, 1
stackres       .usect  ".stack_v", 0x40, 1, 1
abu_buff_loc   .usect  ".sbuff_v", abu_buff_sz, 1, 1
abu_buff_hloc  .set    abu_buff_loc+abu_buff_sz/2  ; half-way

; Memory pointers
iq_data     .set    data_addr     ; 512 * 2 words * I/Q
fft_data    .set    data_addr     ; 512 * 2 words * Re/Im
scale_data  .set    scale_addr    ; 512 words
sqmag_data  .set    data_addr     ; 512 * 2 words
sqsc_data   .set    data_addr+2*data_n    ; 512 words
log_data    .set    data_addr     ; 512 * 2 words
power_data  .set    data_addr     ; 512 words
ebs_data    .set    data_addr+data_n      ; 512 words

; mode flags
mode_std_bit    .set    0001b    ; standard operations
mode_dbg_bit    .set    0010b    ; debug

mode_std_n        .set    transfer_table_sz
mode_dbg_n        .set    512
```

```

; Scratchpad RAM usage
bridge_count    .set    scratch
minor_count     .set    scratch+1
bspce_save      .set    scratch+2
bridge_size     .set    scratch+3
mode_flag       .set    scratch+4
shb_addr        .set    scratch+5
major_count     .set    scratch+6    ; two words!
nco_freq        .set    scratch+8    ; two words!

    .bss TempLmem,1*2,0,0  ;temporary dword


RXDSP_START
ago_main:

    rsbx    XF

; Processor setup
    ssbx    INTM          ; Disable interrupts
    stm     #(stackres+0x40), SP  ; set Stack Pointer
    stm     #npmst,PMST   ; Set processor mode/status
;    stm    #defst0, ST0
;    stm    #defst1, ST1
    rsbx    SXM           ; Suppress sign extension
;    rsbx   XF
    nop     ; Space for branch to app
    nop

appcode:
;    stm    #0,state  ; Clear interrupt routine state
    stm    #0,AR0     ; Clear auxilliary register 0

    portw    AR0,wr_disc  ; Enable parallel TLM drivers, I_Q out
    portw    AR0,wr_dog   ; Strobe watchdog timer

    stm    #0,AR0  ; Clear all auxiliary registers
    stm    #0,AR1
    stm    #0,AR2
    stm    #0,AR3
    stm    #0,AR4
    stm    #0,AR5
    stm    #0,AR6
    stm    #0,AR7

    stm    #0FFh,IFR  ; Clear any pending interrupts
    stm     #ntss,TCR  ; Stop timer, if running

; Main data code start
read_init:
    call    rsp_reset
    nop
    nop
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    portw     AR0,wr_rs_rx  ; Hardware reset of AD6620 RSP
    nop
    nop
    call    rsp_reset
    call    rsp_init
    call    rsp_clear
    call    rsp_mstart

    rpt     #4444  ; Let the AD6620 do its first initialization in peace
    nop

    call    rsp_reset

; store permanent NCO Frequency
    ld      #0x4420, A
    stl     A, @nco_freq
    ld      #0x01FA, A
    stl     A, @(nco_freq+1)

    portr     rs_rx_fifo, AR0  ; Reset RxFIFO - also wired to Slave
    nop
    nop

    stm       #lsb_sel, AR0  ; Reset acq_seq
    portw     AR0, wr_disc
    nop

    ; BSP prep

    stm     #(bspc_Free+bspc_fsm), BSPC0   ; reset BSP
    stm     #(int_bx), IMR                 ; unmask serial tx interrupt
    stm     #(bspce_fe+bspce_bxe), BSPCE0  ; 10-bit words,
                                           ; enable tx autobuffer

    ; where in the 2 kw of buffer RAM does the transmit buffer start?
    stm    #(abu_buff_loc-0x800), AXR
    stm    #(abu_buff_sz), BKX  ; buffer size

; Clear entire serial buffer
    stm    #abu_buff_sz-1, BRC
    stm    #abu_buff_loc, AR4
    rptb   buff_init_loop - 1

    st     #0h, *AR4+

buff_init_loop:
    .global buff_init_loop, head_ramp, major_loop

; Write out header to first buffer half

    stm    #abu_buff_loc, AR4
    ; two-byte frame sync 0xEB90
;    stm    #(abu_buff_loc+abu_buff_sz/2-2), AR4
    st    #0xFE, *AR4+  ; 4-byte initialization sync
    st    #0x6B, *AR4+
    st    #0x28, *AR4+
    st    #0x40, *AR4+

    stm    #file_header, AR0  ; Point to static header words
header_loop:
    ld     *AR0+, A            ; Get a word, point to next
    bc     header_loop_x, AEQ  ; If terminator, end static header
    stl    A, *AR4+            ; Write to serial buffer
    b      header_loop
header_loop_x:

    ld    #abu_buff_loc, A
    ld    #(abu_buff_sz/2), B

    call    _serial_cook

; Set initial bspce_save such that the first acquisition will
; write to the second half of the serial buffer
    st    #bspce_xh, @bspce_save

; Start BSP transmits
    ; have to hold fsm bit
    stm    #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0

;    rsbx    INTM  ; global interrupt enable

; All Aux Registers are fungible in the main loop: values which must
; be preserved over time are stored in the scratchpad RAM as defined
; above.  Note that only AR6 and AR7 are required to be preserved by
; the DSP Library functions (and most others), all other ARx may be
; modified within function calls.

    ld     #0, A
    dst    A, @major_count

major_loop:

    portr    rs_rx_fifo, AR0  ; Reset Rx FIFO - also wired to Slave
    nop
    nop

    stm      #(acq_seq_out+lsb_sel), AR0  ; send acq_seq, set lsb_sel
    portw    AR0, wr_disc

    call    rsp_clear    ; clear NCO RAM
    call    rsp_mstart    ; and start digitizing

;
; Reads
;
data_acq_start:

    st    #0, @bridge_count  ; reset bridged data counter
    st    #0, @minor_count   ; set minor frame counter

;
; Set wait states for Rx FIFO
;
    ldm   SWWSR, A
    ld    #65535, B
    xor   #7, #swwsr_is, B ; (0b111<<Nset XOR 0d65535) creates bitmask
    and   B, A             ; mask out bits of interest
    or    #0, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
    ; Nwait = 0 means no additional waits
    stlm    A, SWWSR

    ssbx    XF
    rpt     #100
    nop
    rsbx    XF

    .global    pre_disc, pre_read
pre_disc:

```

```
; loop to read and discard first data out of AD6620
    stm    #(data_discard), AR2
    nop
rx_discard_loop:
    ; read only if rx fifo is nonempty
    portr    rd_disc, AR0
    nop
    nop
    bitf    AR0, #rx_efo

    bc    discard_fifo_empty, NTC
    portr    rd_rx_out, AR1  ; read data into AR1
    mar    *AR2-             ; decrement word counter
discard_fifo_empty:

    banz    rx_discard_loop, *AR2


; loop to read data from Rx FIFO into RAM
    stm    #2, AR0
    stm    #iq_data, AR1  ; set address for first data word
    stm    #(data_n*2-1), AR2
pre_read:
    nop
rx_read_loop:
    ; read only if rx fifo is nonempty
    portr    rd_disc, AR3
    nop
    nop
    bitf     AR3, #rx_efo

    bc    read_fifo_empty, NTC

    portr    rd_rx_out, *AR1+
    st     #0, *AR1+  ; zero out second word
    mar    *AR2-      ; decrement word counter

read_fifo_empty:

    banzd    rx_read_loop, *AR2
    nop
    nop

;
; Set wait states for other stuff (full 7)
;
    ldm    SWWSR, A
    or      #7, #swwsr_is, A   ; (A or Nwait<<Nset) to set Nwait to Nset
    stlm    A, SWWSR
    nop
    nop

    ssbx     XF
    rpt      #50
    nop
    rsbx     XF

;
; End data acquisition, begin data processing
;

data_process:


    .global pre_window
pre_window:

    ld     #iq_data, A
    ld     #data_n, B

    call     _hann_window

    .global     pre_bit_rev
pre_bit_rev:

; Bit reversal
    stm      #data_n, AR0
    pshm     AR0
    stm      #iq_data, AR0
    pshm     AR0
    ld       #iq_data, A

    call     _cbrev32

    frame    2


    .global pre_fft
pre_fft:
    stm      #fft_scaling, AR0
    pshm     AR0
    ld       #fft_data, A

    call     _cfft32_512

    frame   1


    .global    pre_move
pre_move:
; flip things into proper power spectra order (swap halves)
; remove zeroes here, too

    stm     #(data_n-1), BRC
    stm     #fft_data, AR2
    stm     #(fft_data + 2*data_n), AR3

    rptb    move_loop - 1

    dld     *AR2, A
    dld     *AR3, B

    nop
    nop
    xc      2, AEQ
    add     #1, A
    xc      2, BEQ
    add     #1, B

    dst     A, *AR3+
    dst     B, *AR2+

move_loop:

    .global    pre_sqscale
pre_sqscale:

    stm     #scale_addr, AR0  ; scale saves
    pshm    AR0
    stm     #fft_data, AR0    ; output
    pshm    AR0
    stm     #fft_data, AR0    ; input
    pshm    AR0
    ld      #data_n, A  ; N

    call    _sqmag_prescale

    frame   3  ; free stack


    .global pre_abs
```

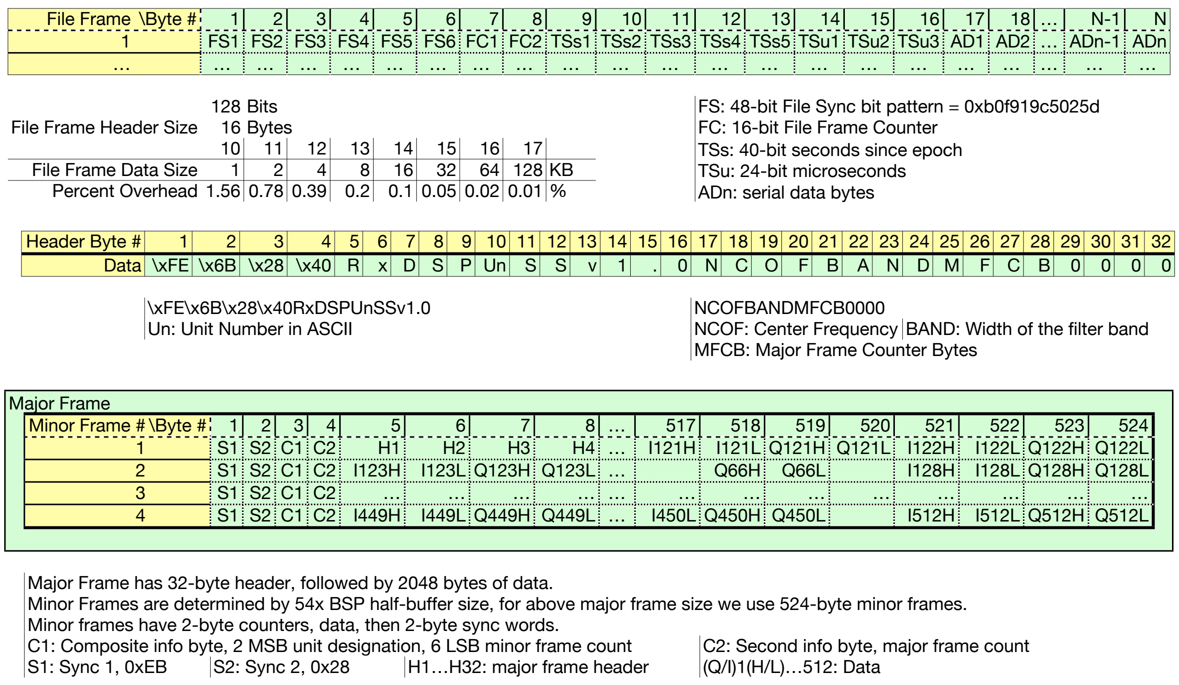
```
pre_abs:
    ssbx    SXM
    ssbx    OVM
    nop
    nop

    stm     #(2*data_n-1), BRC
    stm     #fft_data, AR0
    rptb    abs_loop - 1

    dld     *AR0, A
    abs     A
    dst     A, *AR0+

abs_loop:

    .global    pre_sqmag, pre_log, pre_db
pre_sqmag:
; |FFT|^2
    ld    #sqmag_data, A
    ld    #data_n, B

    call    _sqmag

    .global pre_logps
pre_logps:

    stm     #scale_data, AR0   ; scale saves
    pshm    AR0
    stm     #sqsc_data, AR0    ; output
    pshm    AR0
    stm     #sqmag_data, AR0   ; input
    pshm    AR0
    ld      #data_n, A   ; N

    call    _log_prescale

    frame   3


pre_scale:


pre_log:
; log_10(|FFT|^2) (outputs 32-bit Q16.15)

    stm     #data_n, AR0
    pshm    AR0
    stm     #log_data, AR0  ; write to beginning of data buffer
    pshm    AR0
    ld      #sqsc_data, A   ; read from halfway point of data buffer

    call _log_10

    frame    2


    .global pre_descale
pre_descale:

    stm     #scale_data, AR0  ; scale saves
    pshm    AR0
    stm     #power_data, AR0  ; output
    pshm    AR0
    stm     #log_data, AR0    ; input
    pshm    AR0
    ld      #data_n, A  ; N

    call    _descale

    frame   3  ; free stack


    .global post_descale
post_descale:
; multiply by output_scale_factor,
; shift right by output_shift_n, re-store

    pshm    ST0
    pshm    ST1
    ssbx    FRCT
    ssbx    SXM
    ssbx    OVM
    rsbx    C16

; Scale and shift, save 8-bit data

    stm     #data_n-1, BRC
    stm     #power_data, AR0
    stm     #(data_addr + 2*data_n), AR1
    stm     #ebs_data, AR2
;    stm     #output_scale_factor, T
    rptb    ebs_loop - 1

```

```
;    mpy    *AR0+, A  ; multiply by scale factor in T
;    sfta   A, #0-output_shift_n
;    and    #0xFF0000, A
;    dst    A, *AR1+

;    mpy     *AR0+, A  ; multiply by scale factor in T
    ld      *AR0+, A
    sfta    A, #0-output_shift_n
    add     #128, A
    and     #0xFF, A
    stl     A, *AR2+
;    dadd    output_shift_n, A  ; shift
;    sat     A
;    and     #0xFF, #16, A  ; mask to A(23-16)
;    sth     A, *AR2+       ; store A(23-16)

ebs_loop:

    nop

    .global    dp_end
dp_end:

; Debug code, writes a 512-byte ramp-up-ramp-down
;    stm     #data_n/2-1, BRC
;    stm     #0, AR0
;    stm     #ebs_data, AR2
;    stm     #ebs_data+data_n-1, AR3
;    rptb    dummy_data - 1
;
;    mvkd    AR0, *AR2+
;    mvkd    AR0, *AR3-
;    mar        *AR0+
;
;dummy_data:
;    .global dummy_data

    popm    ST1
    popm    ST0

    nop

;
; End data processing, begin serial data handling
;

    stm      #(lsb_sel), AR0
    portw    AR0, wr_disc

; Standard mode (#buff_size bytes) or
; debug mode (#debug_size) depending on trm_28 state.
    portr    rd_disc, AR0    ; Get discrete bits

;    stm    #0, AR0    ; DEBUG !!
;    nop
;    nop

    bitf    AR0, #trm_28  ; Test for high terminal input
    bc      standard_mode, NTC  ; If trm_28 is low (NTC), standard
                                ; data settings to #debug_size

    st    #mode_dbg_n, @bridge_size
    st    #mode_dbg_bit, @mode_flag

    b     debug_mode_skip

standard_mode:

    st    #mode_std_n, @bridge_size
    st    #mode_std_bit, @mode_flag

debug_mode_skip:

; entry point for bridging data transfers
; over multiple serial half-buffers
bridge_data:
    stm      #(acq_test2+lsb_sel), AR0  ; send acq_seq, set lsb_sel
    portw    AR0, wr_disc

    nop
    nop
    ; Determine serial buffer position
    ld    #abu_buff_loc, A  ; load buffer base

    bitf    @bspce_save, #bspce_xh  ; read XH out of
                                    ; stored BSPCE register
    nop
    nop
    bc      buff_skip, NTC  ; if first half _finished_
                            ; (XH=0, NTC), do nothing

    add      #(abu_buff_sz/2), A
    stm      #(acq_test2+acq_test3+lsb_sel), AR0
    portw    AR0, wr_disc
```

```

buff_skip:
    nop
    nop
    stl  A, @shb_addr ; scratch storage for serial half-buffer address
    nop
    nop

abu_first_half:
    .global abu_first_half

    ssbx    INTM

; Clear serial buffer half
    mvdm    @shb_addr, AR4
    nop
    rpt     #(abu_buff_sz/2-1)
    st      #0xFF, *AR4+

; reset AR4 for data copy
    mvdm    @shb_addr, AR4
    nop
    nop

abu_fill_start:
    .global abu_fill_start

    ; two-byte frame sync 0xEB90
    st    #0xEB, *AR4+
    st    #0x90, *AR4+

    ; two-byte infofoop
    ld     @minor_count, A   ; byte 1, minor frame number
    and    #0xFF, A
    stl    A, *AR4+
    dld    @major_count, A   ; byte 2, major frame number
    and    #0xFF, A, B
    stl    B, *AR4+

post_sync_write:
    .global post_sync_write

; Transfer in selected data mode
    bitf    @mode_flag, #mode_dbg_bit
    bc      dbg_transfer, TC

; In standard mode, check if 8-bit major_count (still in A) == 0,
; if so, transfer a header instead of data.

; Std header: spit out a header frame every 4096th
    and    #header_freq_mask, A
    bc     header_skip, ANEQ  ; A != 0, skip header

    call    hwrite
    st      #mode_std_n, @bridge_count  ; fake it out
    nop
    nop

    b    end_transfer

header_skip:
; Std transfer: selected bins in a 1-frame major frame
    ; transfer selected data to serial buffer
    ld     #ebs_data, A  ; input addr in A
    ldm    AR4, B        ; output addr in B

    call   transfer

    st    #mode_std_n, @bridge_count  ; fake it out

    b     end_transfer

; Debug transfer: entire 512-bin fft
; spread over multiple minor frames.
dbg_transfer:

    ; copy raw data (words) into serial buffer (bytes)
    ld      #ebs_data, A
    add     @bridge_count, A
    stlm    A, AR2

    mvdm    @bridge_size, AR0    ; goal bridge size
    mvdm    @bridge_count, AR1    ; current count

    stm     #((abu_buff_sz/2-fsync_sz)-1), BRC
    rptb    rawdata_loop - 1

;    ld      *AR2+, A  ; load (data word) to Acc
;    and     #0xFF, A  ; mask to low-byte
;    stl     A, *AR4+  ; save to serial buffer

    mvdd    *AR2+, *AR4+

    mar     *AR1+
    cmpr    LT, AR1  ; If we're not done with a bridged data sequence,
    nop
    nop
    xc      2, NTC
    rsbx    BRAF
    nop
    nop
    nop
    nop
    nop
    nop
    .global rawdata_loop, dbg_transfer_skip
rawdata_loop:

    mvmd    AR1, @bridge_count
    nop

end_transfer:
    .global    end_transfer

    nop

serial_transfer_end:
    .global    serial_transfer_end
    nop

; Bit reverse and add start/stop bits
    ld    @shb_addr, A
    ld    #(abu_buff_sz/2), B

    call    _serial_cook

;    rsbx    INTM

    addm    #1, @minor_count

; If a major frame is complete, shut it down

; unset acq_seq, keep lsb_sel
    stm      #lsb_sel, AR0
    portw    AR0, wr_disc

; Strobe watchdog- once per acquisition
    stm      #0, AR0     ; Data is not used- just the wr_dog strobe
    portw    AR0,wr_dog  ; Strobe the watchdog

; Stop acquisition, clear interrupts, then idle until an interrupt.
```

```
    call    rsp_reset

    nop
    nop

    .global pre_sleep
pre_sleep:

    stm      #(acq_test4+lsb_sel), AR0
    portw    AR0, wr_disc

    ssbx    INTM
    stm     #0FFh,IFR  ; Clear any pending interrupts

    idle    2  ; and now...we wait.

    .global post_sleep
post_sleep:

; check for aux int -> serial monitor
;    bitf    IFR, #int_3
;    cc      inth_3, TC

; make sure we had a serial interrupt
    bitf    IFR, #int_bx
    bc      pre_sleep, NTC  ; stray interrupt, go back to IDLE

    nop

    stm    #int_bx, IFR  ; clear int flag

    mvmd  BSPCE0, @bspce_save ; store control extension register in AR6

    bitf    BSPCE0, #bspce_xh
    bc      xh_skip, NTC

    stm      #(lsb_sel), AR0
    portw    AR0, wr_disc

xh_skip:

    rpt    #100
    nop

    stm      #lsb_sel, AR0
    portw    AR0, wr_disc

    mvdm    @bridge_size, AR0   ; need to copy these
    mvdm    @bridge_count, AR1  ; to use CMPR
    nop
    nop
    cmpr    LT, AR1  ; If we're not done with a bridged data sequence,
    bc      bridge_data, TC  ; jump to bridge_data to
                             ; continue transfers, otherwise...
    dld     @major_count, A  ; increment major frame counter
    add     #1, A
    dst     A, @major_count

    b    major_loop  ; new data acquisition

;
; Main acquisiton ('appcode') branch done
;

;
; Interrupts
;

; Non-Maskable Interrupt
;     this is hit by the watchdog
int_nmi:
    nop
    nop

    stm      #0, AR0
    portw    AR0,wr_dog  ; Strobe watchdog timer

    b    0xF800

; Setup: IPTR=0x1FF, OVLY=1, all else =0
; This should set things up to completely
; reload the program from the EPROM on reset.
    stm    #0xFFA0, PMST
    nop
    nop

;    reset  ; I don't have to take this.  ...I'm going home.

    ret  ; should never get here!

inth_3:
    ssbx    INTM
    stm     #int_3, IFR  ; clear int flag

    ; call serial monitor

;    calld  _RxDSP_Monitor
;    stm    #bspce_haltx, BSPCE  ; tell serial transmit to halt
                                 ; after completing this half-buffer

    retd
    nop
    nop

file_header:
    .string     "Dartmouth College Rx-DSP, AGO Site 3 Unit 0."
    .word       0000h  ; Null terminator

;```````
; hwrite
;
; Writes a header:
;
; <0xFE6B2840><RxDSP><Unit #><Ver #><NCOF><MFCB><00000000>
;_______

hwrite:
    .global     hwrite

    pshm        AR3

; 4-byte sync
    stm     #static_header, AR3  ; Point to static header words
    rpt     #static_header_len   ; <SYNC><RxDSP>
    mvdd    *AR3+, *AR4+

    stm     #code_version, AR3
    rpt     #3
    mvdd    *AR3+, *AR4+

    stm     #spec_header, AR3  ; Point to static header words
    rpt     #spec_header_len   ; <skipNS><fbstFBSN><fbenFBEN>
    mvdd    *AR3+, *AR4+

    stm     nco_freq, AR3
    ld      *AR3, #-8, A
    and     #0xFF, A
    stl     A, *AR4+
    ld      *AR3+, A  ; inc to second word
    and     #0xFF, A
    stl     A, *AR4+
```

```
    ld      *AR3, #-8, A
    and     #0xFF, A
    stl     A, *AR4+
    ld      *AR3, A
    and     #0xFF, A
    stl     A, *AR4+

    stm     #band_width, AR3
    rpt     #3
    mvdd    *AR3+, *AR4+

    stm     major_count, AR3
    ld      *AR3, #-8, A
    and     #0xFF, A
    stl     A, *AR4+
    ld      *AR3+, A   ; inc to second word
    and     #0xFF, A
    stl     A, *AR4+
    ld      *AR3, #-8, A
    and     #0xFF, A
    stl     A, *AR4+
    ld      *AR3, A
    and     #0xFF, A
    stl     A, *AR4+

    popm    AR3

    retd
    nop
    nop

static_header:
    .word      0xFE, 0x6B, 0x28, 0x40
    .string    "AGORxDSP"  ; 12 bytes
static_header_end:
static_header_len    .set    static_header_end-static_header-1

spec_header:
    .string    "stride08"
    .string    "cbst0113"
    .string    "cben0166"
    .string    "00000000"  ; pad to 32 bytes
spec_header_end:
spec_header_len    .set    spec_header_end-spec_header-1

    .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
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

	128 Bits								
File Frame Header Size	16 Bytes								
	10	11	12	13	14	15	16	17	
File Frame Data Size	1	2	4	8	16	32	64	128	KB
Percent Overhead	1.56	0.78	0.39	0.2	0.1	0.05	0.02	0.01	%

FS: 48-bit File Sync bit pattern = 0xb0f919c5025d
FC: 16-bit File Frame Counter
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
Un: Unit Number in ASCII

NCOFBANDMFCB0000
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 | C2: Second info byte, major frame count
S1: Sync 1, 0xEB | S2: Sync 2, 0x28 | H1…H32: major frame header | (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.

```
;```````
; rsp_os    Receive Processor One-Shot
;            Quickly loads one 36-bit value to an address on the AD6620

rsp_os:
    ; 36-bit value in A
    ; 10-bit address in B

    ; write address 0x0303 to address registers
    stl     B, #-8, AR4
    andm    #0x03, AR4  ; mask to lower two bytes
    nop
    nop
    nop
    nop
    portw    AR4, wr_rx+amr

    stl     B, AR4
    andm    #0xFF, AR4
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+lar

    ; write 36-bit value into five 8-bit data registers
    mvdm    @AG, AR4
    andm    #0x0F, AR4  ; 0x0F 0000 0000
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+dr4

    sth     A, #-8, AR4  ; 0x00 FF00 0000
    andm    #0xFF, AR4
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+dr3

    sth     A, AR4  ; 0x00 00FF 0000
```

```
    andm    #0xFF, AR4
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+dr2

    stl     A, #-8, AR4  ; 0x00 0000 FF00
    andm    #0xFF, AR4
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+dr1

    stl     A, AR4  ; 0x00 0000 00FF
    andm    #0xFF, AR4
    nop
    nop
    nop
    nop
    portw   AR4, wr_rx+dr0  ; writing to dr0 commits

    retd
    nop
    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.

```
; v1.1    2012.06.01    Added 1-frequency debug mode
;                       controlled by trm_28 jumper

    .mmregs
    .global ZERO, BMAR, PREG, DBMR, INDX, ARCR, TREG1
    .global TREG2, CBSR1, CBER1, CBSR2, CBER2
    .global RXDSP_START
    .ref    _serial_cook
    .ref    rsp_clear, rsp_reset, rsp_init
    .ref    rsp_mstart, rsp_sstart, rsp_freq
    .ref    transfer, transfer_table_sz
    .global bridge_data, buff_clear_loop
    .def    ago_main, int_nmi

    .include "rx-dsp.h"
    .text

find_me             .ulong     0x6B28FE40
unit_designation    .uword     0  ; Unit number (Master = 0)
code_version        .string    "v1.0"
station_code        .string    "SS"

band_width    .string    "0750"

; Rotating center frequency table
; 32-bit values derived by mapping the sampling frequency (S) to
; the 0:2^32 range, then taking the ratio of center frequency
; (C) to S, i.e. C/S*2^32

cfreq_table:
    .word    0x01d2, 0xf1c9  ;  475 kHz
    .word    0x04b4, 0x39a7  ; 1225 kHz
    .word    0x0795, 0x8185  ; 1975 kHz
    .word    0x0a76, 0xc964  ; 2725 kHz
cfreq_table_end:
cfreq_table_sz    .set    cfreq_table_end-cfreq_table

cfreq_test    .word    0x0e66, 0x6758 ; 3750 KHz, used when
                                      ; trm_28 jumper is on

cfreq_toggle1    .set    0x0999  ; 2500 kHz
cfreq_toggle2    .set    0x9A3B


; Run constants
data_n         .set 512  ; Size of each FFT (# of IQ pairs)
data_discard   .set 512  ; number of words to discard from
                         ; Rx FIFO before taking data
data_minor_sz  .set 1    ; number of acquisitions per
                         ; half-buffer interrupt
abu_buff_sz    .set 1048 ; size of serial buffer
                         ; (2x major frame size IN BYTES)
fsync_sz       .set 4    ; # of serial frame sync bytes
                         ; (should be multiple of 4)

; Memory allocations
data_addr      .usect  ".data_v", 0x800, 1, 1
stackres       .usect  ".stack_v", 0x40, 1, 1
abu_buff_loc   .usect  ".sbuff_v", abu_buff_sz, 1, 1
abu_buff_hloc  .set    abu_buff_loc+abu_buff_sz/2  ; half-way

; Scratchpad RAM usage
bridge_count    .set    scratch
minor_count     .set    scratch+1
bspce_save      .set    scratch+2
bridge_size     .set    scratch+3
major_count     .set    scratch+4  ; two words!
nco_freq        .set    scratch+6  ; two words!
cfreq_tp        .set    scratch+8

    .bss    TempLmem,1*2,0,0  ;temporary dword


RXDSP_START
ago_main:

    rsbx    XF

; Processor setup
    ssbx    INTM         ; Disable interrupts
    stm     #(stackres+0x40), SP  ; set Stack Pointer
    stm     #npmst,PMST  ; Set processor mode/status
;    stm #defst0, ST0
;    stm    #defst1, ST1
    rsbx    SXM  ; Suppress sign extension
;    rsbx    XF
    nop          ; Space for branch to app
    nop

    ssbx    XF
    rpt     #64
    nop
    rsbx    XF

appcode:
;    stm    #0,state      ; Clear interrupt routine state
    stm     #0,AR0        ; Clear auxilliary register 0
    portw   AR0,wr_rs_rx  ; Reset AD6620 RSP
    portw   AR0,wr_disc   ; Enable parallel TLM drivers, I_Q out
    portw   AR0,wr_dog    ; Strobe watchdog timer

    stm    #0,AR0  ; Clear all auxiliary registers
    stm    #0,AR1
    stm    #0,AR2
    stm    #0,AR3
    stm    #0,AR4
    stm    #0,AR5
    stm    #0,AR6
    stm    #0,AR7

    stm    #0FFh,IFR  ; Clear any pending interrupts
    stm    #ntss,TCR  ; Stop timer, if running
    .global read_init
; Main data code start
read_init:
    call    rsp_reset
    call    rsp_init
    call    rsp_clear
    call    rsp_mstart

    rpt    #4444  ; Let the AD6620 do its first initialization in peace
    nop

    call    rsp_reset

    portr    rs_rx_fifo, AR0   ; Reset Rx FIFO - also wired to Slave
    nop
    nop

    stm    #lsb_sel, AR0   ; Reset acq_seq
    portw  AR0, wr_disc
    nop


    ; TSP shutoff

;    stm    #(tspc_Free+tspc_fsm+tspc_nXrst+tspc_nRrst), TSPC
    stm    #(tspc_Free+tspc_fsm), TSPC

    ; BSP prep

    stm    #(bspc_Free+bspc_fsm), BSPC0  ; reset BSP
    stm    #(int_bx), IMR  ; unmask serial transmit interrupt
    ; 10-bit words, enable tx autobuffer, halt after first half-buffer
    stm    #(bspce_fe+bspce_bxe+bspce_haltx), BSPCE0
    stm    #(abu_buff_loc-0x800), AXR  ; where in RAM does
                                       ; the tx buffer start?
    stm    #(abu_buff_sz), BKX  ; buffer size
```

```

; Clear entire serial buffer
    stm     #abu_buff_loc, AR4
    stm     #abu_buff_sz-1, BRC
    rptb    buff_init_loop - 1

    st    #0h, *AR4+
    .global buff_init_loop, head_ramp, major_loop
buff_init_loop:

; set initial frequency table position

    st    #cfreq_table_sz-2, @cfreq_tp

;    rsbx    INTM ; global interrupt enable

; All Aux Registers are fungible in the main loop: values which must
; be preserved over time are stored in the scratchpad RAM as defined
; above.  Note that only AR6 and AR7 are required to be preserved by
; the DSP Library functions (and most others), all other ARx may be
; modified within function calls.

    ld     #0, A
    dst    A, @major_count

; Set initial bspce_save such that the first acquisition will
; write to the second half of the serial buffer
    st    #bspce_xh, @bspce_save

    ; boot lag: insert >50 ms delay to allow
    ; everyone plenty of time to boot up
    stm    #16384, BRC
    nop
    nop
    rptbd    boot_delay_loop - 1
    nop
    nop

    nop

    rpt    #4096
    nop

    nop
    nop

boot_delay_loop:

    nop
    nop

    ; now synchronize units for the first time
    call    sync_units

    ; Start BSP transmits
    stm    #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0  ; hold fsm bit
    nop
    nop

;```````
; Main loop
;_______

major_loop:
    .global    major_loop

    ; Method:
    ;
    ; Master waits for acq_seq_rdy from Slaves, then raises
    ; acq_seq_out, starting synced acquisition.  Note any previous
    ; acquisition will still be transferring its last half-buffer,
    ; and the ABU will be set to halt transmissions when that half
    ; is done.  The acq_seq_rdy & out lines are both held high on
    ; All, until the end of this new acquisition and half-buffer
    ; fill.  Then Master waits for its own ABU haltx, and then for
    ; !acq_seq_rdy (signaling Slaves have hit ABU haltx), before
    ; dropping acq_seq_out, signaling time for the next
    ; synchronized ABU startup.

    call    cfreq_walk

    ; acquisition sync
    call    sync_units

    portr    rs_rx_fifo, AR0  ; Reset Rx FIFO - also wired to Slave

    cmpm    *(unit_designation), #0
    bc    slave_startup, NTC

    ; if Master (unit 0) insert delay time to allow
    ; Slaves time to detect acq_seq_out and start up
    rpt    #128
    nop

```

```
    call    rsp_mstart  ; start digitizing as Master

    b    data_acq_start

slave_startup:

    call    rsp_sstart  ; start digitizing as Slave

;
; Reads
;
data_acq_start:

    st    #0, @bridge_count  ; reset bridged data counter
    st    #0, @minor_count   ; reset minor frame counter

;
; Set wait states for Rx FIFO
;
    ldm   SWWSR, A
    ld    #65535, B
    xor   #7, #swwsr_is, B  ; (0b111<<Nset XOR 0d65535) creates bitmask
    and   B, A              ; mask out bits of interest
    or    #0, #swwsr_is, A  ; (A or Nwait<<Nset) to set Nwait to Nset
    ; Nwait = 0 means no additional waits
    stlm    A, SWWSR

    .global    pre_disc, pre_read
pre_disc:

; loop to read and discard first data out of AD6620
    stm    #(data_discard), AR2
    nop
rx_discard_loop:
    ; read only if rx fifo is nonempty
    portr    rd_disc, AR0
    nop
    nop
    bitf     AR0, rx_efo

    bc     discard_fifo_empty, NTC
    portr  rd_rx_out, AR1  ; read data into AR1
    mar    *AR2-           ; decrement word counter
discard_fifo_empty:

    banz    rx_discard_loop, *AR2

```

```

; loop to read data from Rx FIFO into RAM
    stm    #2, AR0
    stm    #data_addr, AR1  ; set address for first data word
    stm    #(data_n*2-1), AR2
pre_read:
    nop
rx_read_loop:
    ; read only if rx fifo is nonempty
    portr    rd_disc, AR3
    nop
    nop
    bitf     AR3, rx_efo

    bc    read_fifo_empty, NTC

    portr    rd_rx_out, *AR1+
    mar      *AR2-  ; decrement word counter

read_fifo_empty:

    banzd    rx_read_loop, *AR2
    nop
    nop

;
; Set wait states for other stuff (full 7)
;
    ldm    SWWSR, A
    or     #7, #swwsr_is, A ; (A or Nwait<<Nset) to set Nwait to Nset
    stlm   A, SWWSR
    nop
    nop

; Stop acquisition
    call    rsp_reset

    ; drop acq flags
    stm      #lsb_sel, AR0
    portw    AR0, wr_disc

;
; End data collection, begin serial data handling
;
    st    #4*data_n, @bridge_size

; entry point for bridging data transfers
; over multiple serial half-buffers
bridge_data:
    .global    bridge_data

    stm     #(lsb_sel), AR0  ; send acq_seq, set lsb_sel
    portw   AR0, wr_disc

    nop
    nop
    ; Determine serial buffer position
    stm     #abu_buff_loc, AR3  ; load buffer base

    bitf    @bspce_save, #bspce_xh  ; read XH out of
                                    ; stored BSPCE register
    nop
    nop
    bc      buff_skip, NTC  ; if first half _finished_
                            ; (XH=0, NTC), do nothing

    addm     #(abu_buff_sz/2), @AR3
    stm      #(lsb_sel), AR0
    portw    AR0, wr_disc

buff_skip:
    nop
    nop
    nop
    nop

abu_first_half:
    .global    abu_first_half

    ssbx    INTM

; Clear serial buffer half
    mvmm    AR3, AR4
    nop
    rpt     #(abu_buff_sz/2-1)
    st      #0x00, *AR4+


; reset AR4 for data copy
    mvmm    AR3, AR4
    nop
    nop

abu_fill_start:
    .global    abu_fill_start


    ; two-byte frame sync 0xEB90
    st    #0xEB, *AR4+
    st    #0x90, *AR4+

    ; two-byte infofoop
    ld     *(unit_designation), #6, A   ; byte 1, 2 MSB, unit number
    or     @minor_count, A   ; byte 1, 6 LSB, minor frame number
    stl    A, *AR4+
    dld    @major_count, A   ; byte 2, 8 bit, major frame number
    and    #0xFF, A
    stl    A, *AR4+

dinner_is_ready:
    .global dinner_is_ready

    ld    #((abu_buff_sz/2-fsync_sz)/2-1), B   ; let's stick the
                                               ; BRC value here...

    ; If haltx is true, we are starting a
    ; new major frame, so write the header
    bitf    BSPCE0, #bspce_haltx
    bc    header_skip, NTC

    call    hwrite   ; write out header, uses &
                     ; modified write address in AR4
    sub     #16, B   ; oops, 32 bytes less space in this half buffer

header_skip:
    ; copy raw data (words) into serial buffer (bytes)
    ld     #data_addr, A
    add    @bridge_count, #-1, A   ; div by 2 to increment
                                   ; input word-wise
    stlm    A, AR2   ; store a copy of the data to AR2
    stlm    B, BRC   ; and here's our BRC setup

    mvdm    @bridge_size, AR0    ; goal bridge size
    mvdm    @bridge_count, AR1   ; current count

    rptb    rawdata_loop - 1

    ld     *AR2, -8, A   ; load (data word) >> 8 to Acc
    and    #0xFF, A      ; mask to low-byte
    stl    A, *AR4+      ; save to serial buffer
    mar    *AR1+         ; count bytes

    ld      *AR2+, A      ; reload and increment
    and     #0xFF, A      ; mask
    stl     A, *AR4+      ; save
    mar     *AR1+         ; count bytes

    cmpr    LT, AR1   ; If we're not done with a bridged data sequence
    nop
    nop
    xc      1, NTC
    rsbx    BRAF
    nop
    nop
    nop
    nop
    nop
    .global    rawdata_loop, dbg_transfer_skip
rawdata_loop:

    nop
    nop

    mvmd    AR1, @bridge_count

; Bit reverse and add start/stop bits
    ldm     AR3, A
    ld      #(abu_buff_sz/2), B

    call    _serial_cook


;     rsbx     INTM

    addm     #1, @minor_count
    nop
    nop
    stm      #(lsb_sel), AR0
    portw    AR0, wr_disc

; buffer is now loaded
    call     rsp_clear  ; clear NCO RAM, do it here
                        ; since we have some free time

; Strobe watchdog- once per minor frame cycle
    stm       #0, AR0     ; Data is not used- just the wr_dog strobe
    portw     AR0,wr_dog  ; Strobe the watchdog

```

```
; clear interrupt flags, then idle until an interrupt.
    .global    pre_sleep
pre_sleep:

    ssbx    INTM
    stm     #0FFh,IFR   ; Clear any pending interrupts

    idle    2  ; and now...we wait.

    .global    post_sleep
post_sleep:

; check for aux int -> serial monitor
;    bitf    IFR, #int_3
;    cc      inth_3, TC

; make sure we had a serial interrupt
    bitf    IFR, #int_bx
    bc      pre_sleep, NTC   ; stray interrupt, go back to IDLE
    nop

    stm    #int_bx, IFR     ; clear int flag

    mvmd    BSPCE0, @bspce_save ; store control extension register

; if this is a new major frame, we need to sync
; everyone up by waiting for all ABU haltx.

    bitf    BSPCE0, #bspce_haltx
    bc      abu_restart_skip, NTC

abu_haltx_wait:
    .global    abu_haltx_wait

; ABU has halted.  Reset ABU, and synchronize
    stm    #(bspce_fe+bspce_bxe), BSPCE0  ; 10-bit words, enable
                                          ; tx ABU, disable haltx

    call    sync_units

; Start BSP transmits
    stm    #(bspc_Free + bspc_fsm + bspc_nXrst), BSPC0  ; hold fsm bit
    nop
    nop

    rpt    #16383
    nop
```

```

; unset acq_seq, keep lsb_sel
    stm      #lsb_sel, AR0
    portw    AR0, wr_disc

abu_restart_skip:
    .global    abu_restart_skip

    mvdm    @bridge_size, AR0   ; need to copy these
    mvdm    @bridge_count, AR1  ; to use CMPR
    nop
    nop
    cmpr    LT, AR1           ; If we're not done with
                              ; a bridged data sequence,
    bc      bridge_data, TC   ; jump to bridge_data to
                              ; continue transfers, otherwise...

    dld    @major_count, A    ; increment major frame counter
    add    #1, A
    dst    A, @major_count

; The final half is transmitting, we want to halt when it finishes.
    stm      #(lsb_sel), AR0
    portw    AR0, wr_disc
    orm      #bspce_haltx, BSPCE0
    nop
    nop
;     stm     #(lsb_sel), AR0
;     portw   AR0, wr_disc

    b    major_loop   ; new data acquisition

;```````
; Main acquisiton ('appcode') branch done
;_______

;
; Interrupts
;

; Non-Maskable Interrupt
;      this is hit by the watchdog
int_nmi:

    nop
    stm    #npmst, PMST   ; Reset PMST to be sure IPTR -> 0x80

```

```
; Alternative: IPTR=0x1FF, OVLY=1, all else =0
; This should set things up to completely reload
; the program from the EPROM on reset
;     stm    #0xFFA0, PMST

    reset  ; I don't have to take this.  ...I'm going home.

    ret  ; should never get here!


inth_3:
    ssbx    INTM
    stm     #int_3, IFR  ; clear int flag

    retd
    nop
    nop

;```````
; cfreq_walk
;
; walks through the table of center frequencies
; at label #cfreq_table
;
; stores table position @cfreq_tp and current NCO value
; as a 32-bit number @nco_freq

cfreq_walk:

    portr    rd_disc, AR0
    nop
    nop
    bitf     AR0, #trm_28
    ; If trm_28 is low (NTC, jumper on), skip the walk.
    bc       walk_skip, NTC

    addm    #2, @cfreq_tp
    nop
    nop
    cmpm    @cfreq_tp, #cfreq_table_sz
    nop
    nop
    xc    2, TC
    st    #0, @cfreq_tp

    nop
    nop
```

```

    ld      #(cfreq_table), A
    add     @cfreq_tp, A
    stlm    A, AR0

    b    cfreq_commit

walk_skip:
    ; load a single frequency instead
    stm    #(cfreq_test), AR0

cfreq_commit:

    nop
    nop

    ld      *AR0+, B
    stl     B, @nco_freq
    sftl    B, #8
    sftl    B, #8
    ld      *AR0, A
    stl     A, @(nco_freq+1)
    or      B, A

    nop
    nop

    stm    #acq_ant_1, AR0  ; enable antenna 1

    ; frequency is in A.  subtract toggle freq and save result to B
    sub    #cfreq_toggle1, #16, A, B
    sub    #cfreq_toggle2, B

    xc     2, BGEQ          ; if B>=0, cfreq >= toggle freq
    stm    #acq_ant_2, AR0  ; so enable antenna 2
    nop

    portw  AR0, wr_disc     ; write out antenna toggle lines

    call   rsp_freq

    retd
    nop
    nop

;```````
; sync_units
```

```
;
; synchronizes master/slave units by
; toggling and waiting for latched lines

sync_units:

; Raise test2 line.  On the Master this should do nothing (NC),
; on the Slaves it signals the Master they are ready.

    stm      #(acq_seq_rdy+lsb_sel), AR0
    portw    AR0, wr_disc
    nop
    nop

; Check status of TSP, wait for IN1 & IN2 high
ready_loop:

    nop

    bitf    TSPC, #tspc_in0
    bc      ready_loop, NTC

    bitf    TSPC, #tspc_in1
    bc      ready_loop, NTC

; Raise acq_seq_out--on the Master this signals the Slaves
; to start, on the Slaves it does nothing (NC).
    stm    #(acq_seq_out+acq_seq_rdy+lsb_sel), AR0
    portw    AR0, wr_disc

    retd
    nop
    nop

static_header:
    .word    0xFE, 0x6B, 0x28, 0x40
    .string    "RxDSP"
static_header_end:

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, rtdfile="/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", "--rtd", action="store_true", dest="rtd",
       help="Write out major frames for RTD. [%default]")
22 parser.add_option("-R", "--rtdfile", type="str", dest="rtdfile", 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",
29                 help="print status messages to stdout.")
30
31 (o, args) = parser.parse_args()
```

```

o.majfsync = pack("4B", *[(o.majfsync>>i)&0xFF for i in [24,16,8,0]])
o.filesync = pack("6B", *[(o.filesync>>i)&0xFF for i in [40,32,24,16,8,0]])

try:
    inp = serial.Serial(port=o.port, baudrate=115200, bytesize=8,
        parity='N', stopbits=1)
except serial.SerialException:
        print "Unable to open serial port {0}.".format(o.port)
        sys.exit(1)

#Restore power to receiver by toggling the DTR line
time.sleep(5.0)
inp.setDTR(False)

print("Reading from serial port {0}...".format(o.port))

dtstr = datetime.today().strftime("%Y%m%d-%H%M%S")

ofn = "{0}-{1}.data".format(o.ofp, dtstr)
ofile = open(ofn, "w")

print("Taking {0} {1}-byte acquisitions ({2} hours).".format(o.limit,
    o.acqsz, o.limit*o.acqsz/11520/3600))

print("Writing data to {0}...".format(ofn))

if o.rtd:
        rfile = open(o.rtdfile, 'w')

print("Writing RTD data to {0}.".format(o.rtdfile))

framecount = 0
bytestr = ""
mfbsync = "".join([o.majfsync[i]*o.nboards for i in range(len(o.majfsync))])
mfbsize = o.nboards*o.majfsz*o.minfsz

running = True

acqcount = 0

while running and acqcount < o.limit:
        data = inp.read(o.acqsz)

        # build timestamp: 40-bit uint seconds since epoch, 24-bit uint
            microseconds
        timefl = time.time()
```

```python
        timeint = int(timefl)
        timefrac = int((timefl-timeint)*(1e6))
        timestr = ( pack(">Q", timeint&0xFFFFFFFFFF)[3:] ) + ( pack(">I", timefrac)[1:] )

        ofile.write(o.filesync)
        ofile.write(pack('>H', framecount&0xFFFF))      # 16-bit counter
        ofile.write(timestr)
        ofile.write(data)

        framecount += 1

        # if we want rtd, add to bytestream, and if there's a major frame in there, write it

        if o.rtd:
                bytestr += data

                loc = bytestr.find(mfbsync)
                nloc = bytestr.find(mfbsync, loc+1)

                if (loc >= 0) and (nloc >= loc):
                        rfile.seek(0)
                        rfile.write(bytestr[loc-4 * o.nboards:nloc-4 * o.nboards])
                        rfile.flush()
#                       prelen = len(bytestr)
                        bytestr = bytestr[nloc-4*o.nboards:]
#                       postlen = len(bytestr)
#                       print("Wrote RTD file. {0} -> {1}".format(prelen,postlen))

        acqcount += 1

if o.rtd:
        rfile.close()
ofile.close()
```

tridsp-rtd.py

```python
#!/usr/bin/env python
from os.path import getmtime
import sys
from struct import unpack,pack
from datetime import datetime
import numpy as np
import Gnuplot
```

```
#from matplotlib import pyplot as plt
from optparse import OptionParser
from math import ceil, floor

parser = OptionParser(usage="""tridsp-acq.py: read in triple-DSP multiplexed
    serial data.""")

parser.set_defaults(verbose=False, rfile="/tmp/trirtd.data", nboards=3,
                                                  majfsync=0xFE6B2840,
                                                      majfsz=4,
                                                  minfsync=0xEB90, minfsz=524,
                                                      dataplot=True)

parser.add_option("-r", "--rfile", type="str", dest="rfile", help="RTD data
    file.  [default: %default]")
parser.add_option("-n", "--nboards", type="int", dest="nboards",
    help="Number of multiplexed DSP boards.  [%default]")
parser.add_option("-s", "--maj-sync", type="int", dest="majfsync",
    help="Major frame sync pattern.  [%default]")
parser.add_option("-z", "--maj-size", type="int", dest="majfsz", help="# of
    Minor frames per Major frame.  [%default]")
parser.add_option("-S", "--min-sync", type="int", dest="minfsync",
    help="Minor frame sync pattern.  [%default]")
parser.add_option("-Z", "--min-size", type="int", dest="minfsz", help="# of
    bytes per Minor frame.  [%default]")
parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
                 help="print status messages to stdout.")
parser.add_option("-c", "--channel", action = "store",type = 'int',
    dest="chan_num", help = "channel number to display")
parser.add_option("-f", "--freq", action = "store_true", dest = "spectra")
parser.add_option("-b", "--band", action = "store", dest = "bw",
    type="float")
(o, args) = parser.parse_args()
def     b2iq(indat):
        # input : binary string containing 16-bit big-endian signed I/Q data
        # output: [[I array], [Q array]]

        num = unpack(">"+str(len(indat)/2)+"h", indat)
        return np.reshape(num, (2,-1), "F")

cplotd = {}
blist = range(o.nboards)

infile = open(o.rfile, 'r')
freqplot=Gnuplot.Gnuplot()
#freqplot("set title \"Unit {0}\"".format(k,cfreq))
freqplot("set term x11 noraise")
freqplot("set yrange [0:140]")
freqplot("set xrange [100:3100]")
freqplot("set style data lines")
#freqplot("set title \"{0} KHz\"".format(cfreq))

oldtime = 0
first_loop = 0
plots= []

running = True

window = np.hanning(512)

while running:

        while True:
                newtime = getmtime(o.rfile)
                if newtime != oldtime:
                        break

        oldtime = newtime
        badness = False

        infile.seek(0)
        data = infile.read()

        # parse major frames and plot
        unitdata = [[]]*o.nboards

        for i in blist:
                bdata = data[i::o.nboards]
                majfr = "".join([bdata[j*o.minfsz+4:(j+1)*o.minfsz] for j in
                    range(o.majfsz)]) # extract major frame
                minsyncs = "".join([bdata[j*o.minfsz:j*o.minfsz+4] for j in
                    range(o.majfsz)])  #extract minor frame syncs

                minunit = [(x & 0xC0)>>6 for x in unpack(">4B",
                    minsyncs[2::4])]
                if len(np.unique(minunit)) != 1:
                        print("Minor frame Unit number mismatch.")
                        badness = True

                minmajN = unpack(">4B", minsyncs[3::4])
                if len(np.unique(minmajN)) != 1:
                        print("Minor frames' major frame # mismatched.")
                        badness = True

                unit = unpack("B", majfr[9])[0]-0x30
                if unit != np.unique(minunit)[0]:
                        print("Major frame doesn't match minor frame Unit #.")
                        badness = True

                majN = unpack(">I", majfr[24:28])[0]&0xFF
                if majN&0xFF != np.unique(minmajN)[0]:  # mask to one-byte to test against minor frame #
                        print("Major frame # doesn't match minor frames' major frame #.")
                        badness = True

                # looks like a good major frame
                cfreq = int(round(unpack(">I", majfr[16:20])[0]/2.0**32*66666.6))
                nums = unpack(">"+str(len(majfr)/2-16)+"h", majfr[32:])

                unitdata[unit] = {      'cfreq' : cfreq,
                                                'eyes' : nums[::2],
                                                'ques' : nums[1::2] }

        if len(np.unique([ x['cfreq'] for x in unitdata ])) != 1:
                print("Center frequency mismatch!")

        if badness:
                continue

        cfreq = unitdata[0]['cfreq']

        if o.spectra:
                k = o.chan_num
                dfdb = o.bw/512.0
                freqlist = [cfreq-(o.bw/2.0) + dfdb * i for i in range(512)]
                eyes =  unitdata[k]['eyes']
                ques =   unitdata[k]['ques']
                fftdata = [eyes[j] + ques[j] * 1j for j in range(len(eyes))]

                spec = [10.0*y for y in np.log10([abs(x)**2 for x in np.fft.fft(window*fftdata,n=512,)])]       # power spectra
                spec = spec[len(spec)/2:]+spec[:len(spec)/2]    # swap from normal order

                if ((cfreq == 475) or (cfreq == 3750)) and (len(plots) > 0):
                        freqplot.plot(*plots)
                        plots = []
                plots.append(Gnuplot.Data(freqlist, spec))
```