Topics in FS station software coding

May 2019 TOW

Ed Himwich, GSFC/NVI

This seminar is intended to be driven by requests from the attendees. As a result, there are no advance notes. Room is left on this page for attendees to make some of their own notes. We intend to provide a write-up of what was covered in the web version of the notes after the meeting. The pages from the write-ups from previous TOW for "Writing Station Specific FS Code" seminars follow, for reference use. Old, but still relevant block diagrams of FS program interaction follow the "Code: Basic" section. At the end of this document is a copy of a presentation from TOW 2015 about how cable-wrap works and interacts with scheduling, it is recommended reading for stations implementing `antcn` for Az/El antennas.

Notes:

Code: Basic

May 2019 TOW

Ed Himwich, NVI/GSFC

1. Overview of FS Architecture

1.1. Diagrams from FS manual, attached at end of write-up
- 1.1.1. Initialization
- 1.1.2. Normal Execution
- 1.1.3. Pointing Programs
- 1.1.4. Module Checking
- 1.1.5. Data logging

1.2. Only `ddout` writes to disk

1.3. Except for `boss` reading the schedule and procedures, there is no disk reading except during initialization (one exception, the `save_file` SNAP command).

2. Control Files for Station Coding

2.1. `stpgm.ctl`
- 2.1.1. Defines station programs to run and stop at FS start and stop
- 2.1.2. Trailing ampersand `&` means it is run in background
- 2.1.3. No ampersand `&` means wait for return before proceeding
- 2.1.4. Programs
  - 2.1.4.1. Most stations only have a subset
  - 2.1.4.2.`stcom`
    - 2.1.4.2.1. Initializes station shared memory
    - 2.1.4.2.2. Reads some control files
    - 2.1.4.2.3. Listed first in control file, so runs first
    - 2.1.4.2.4. Used with no ampersand to "wait"
  - 2.1.4.3. Other programs are run in background, with ampersand
    - 2.1.4.3.1. `stqkr` station SNAP commands
    - 2.1.4.3.2. `antcn` antenna interface
    - 2.1.4.3.3. `cheks` station module checking
    - 2.1.4.3.4. `sterp` station error reporting

2.2. `sterr.ctl`
- 2.2.1. Station program error messages.
- 2.2.2. See Error Messages in Code: Intermediate write-up for more details

2.3. `stcmd.ctl`
   2.3.1. Station SNAP commands access control

2.4. `mdlpo.ctl`
   2.4.1. Pointing model file, usually read by `antcn`

3. Resource allocation

3.1. For FS internals this is done by `fsalloc`: shared memory, semaphores, and message queue
3.2. For station software this can be done with sample `stalloc`. Example provides only shared memory: C and two areas of FORTRAN shared memory

4. Emulation Services

4.1. Class I/O passes binary buffers between programs
4.2. Scheduling allows programs to pass control back and forth between "parent" and "child" processes
4.3. Resource numbers or semaphores allows coordinated access to resources
4.4. "Break" allows a signal to be sent to program, using `brk` program, to initiate some special action, usually aborting some behavior
4.5. Suspending and Resuming - Allows a program to suspend execution until some other action is taken, usually by the operator who must tell the program to resume again, with the `go` program.
4.6. Shared memory, straightforward in C, complicated in FORTRAN because there is no direct support

5. More Information

5.1. Code Intermediate and Advanced Code write-ups
5.2. End of Volume 2 of the FS Manuals has several relevant sections but some out of date
5.3. See examples of use in FS code

6. `antcn`

6.1. Must be coded to not cause delays, no action should take more than a second
6.2. If antenna communication fails because it needs to re-initialized, e.g., open a new socket, after the antenna is restarted, `antcn` should be coded to handle that automatically without having to restart the FS to rerun mode 0

6.3. Modes

0	initialization
1	new source
2	new offsets
3	check onsource status, with error logging
4	`antenna=` command
5	onsource with no error logging
6	reserved for focus control
7	onsource with additional information
8	station specific detectors, see `/usr2/fs/misc/stndet.txt`
9	Satellite tracking, see `/usr2/fs/misc/satellites.txt`
10	Termination mode, for antenna clean-up on FS termination, must return promptly
11-99	Reserved for future use
100-32767	Reserved for station use

6.5. Example antcn

```
/* antcn

6.6.      This is the antcn (ANTenna CoNtrol program) for Tsukuba 32.
*/

#define MINMODE 0
#define MAXMODE 8
#define MDLPO "/usr2/control/mdlpo.ctl"

#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../../fs/include/dpi.h"
#include "../../fs/include/fs_types.h"
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/params.h"
#include "../../fs/include/fscom.h"
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"     /* shared memory pointer */

struct stcom *st;
struct fscom *fs;

void st_setup(), st_command(), st_onsource(), st_antenna();
void setup_ids(), setup_st();
void putpname();
int gmodl();
void corrq(), equn(), tracq(), pmdlq();
void skd_run(), cls_clr();
int nsem_test();
void logit();
long idum[] = {0,0,0,0,0};
long cls_alc();

main()
{
  int ierr;
  int imode;
  long ip[5];
  int iy,id;
  double eqofeq;
  char idev[64],oldlog[8]; /* test */
  char buf [80];
  int imem; /* test */
  int nrec, nrecr;
  long class, classr;
  int i;

/* Set up IDs for shared memory, then assign the pointer to
   "fs", for readability.
 */
  setup_ids();
  fs = shm_addr;
  setup_st();
  st = stm_addr;

/* Put our program name where logit can find it. */

  putpname("antcn");

/* allocate class box for message from trakl */
```

```
    if( -1 == (stm_addr->antbox=cls_alc())) {
      fprintf( stderr," antbox allocation failed\n");
        exit( -1);
    }

/* Return to this point to wait until we are called again */

Continue:
  skd_wait("antcn",ip,(unsigned)0);

  imode = ip[0];
  class = ip[1];
  nrec = ip[2];
  nrecr = 0;
  classr = 0;
  if (imode < MINMODE || imode > MAXMODE) {
    ierr = -1;
    goto End;
  }

  switch (imode) {

    case 0:             /* initialize */
      ierr = 0;
      if (gmodl(MDLPO,&st->pmodel) < 0) {
        ierr = -6;
        goto End;
      }
      st_setup();
      skd_run("trakl",'n',idum);
      break;
    case 1:             /* source= command */
    case 2:             /* offsets         */
      if (nsem_test("trakl") != 1) {
        logit(NULLPTR,-8,"an");
        goto End;
      }
      if (memcmp(st->point.oldlog,fs->LLOG,sizeof(st->point.oldlog))!=0)
        pmdlq(&st->pmodel);   /* log the model  */
      st_command(imode);
      fs->ionsor=0;
      break;

    case 4:            /* direct antenna= command */
      if (class == 0)
        goto End;
      if (nsem_test("trakl") != 1) {
        logit(NULLPTR,-8,"an");
        cls_clr(class);
        goto End;
      }
      st_antenna(class,nrec,&classr,&nrecr,&ierr);
      break;

    case 6:            /* reserved */
      ierr = -1;
      goto End;

    case 3:    /* onsource command with error message    */
    case 5:    /* onsource command with no error logging */
    case 7:    /* onsource command with additional info  */
      if (nsem_test("trakl") != 1) {
        logit(NULLPTR,-8,"an");
        goto End;
      }
      st_onsource();
      if (st->error.konsor)
        fs->ionsor=1;
      else {
        fs->ionsor=0;
        if (imode == 3 && st->point.itype < 5 )
```

```
        logit(NULLPTR,-103,"an");
      }
      if (imode == 7) {
        tracq(&st->pos_old,&st->ercr_old);
        iy = st->pos_old.t[5] - 1900;
        id = st->pos_old.t[4];
        equn(iy,id,&eqofeq);
        corrq(&st->ercr_old,eqofeq);
      }

      break;
  case 8:
    if(strncmp(shm_addr->user_dev1_name,"  ",2)!=0) {
      char *meter;
      for (i=0;i<5;i++)
        ip[i]=0;
      if(strncmp(shm_addr->user_dev1_name,"u5",2)==0) {
        meter="p2";
        ib_req2(ip,meter,"AP");
      } else if(strncmp(shm_addr->user_dev1_name,"u6",2)==0) {
        meter="p1";
        ib_req2(ip,meter,"AP");
      }
      skd_run("ibcon",'w',ip);
      skd_par(ip);
      if(ip[2]<0)
        goto Continue;
    }
    if(strncmp(shm_addr->user_dev2_name,"  ",2)!=0) {
      char *meter;
      for (i=0;i<5;i++)
        ip[i]=0;
      if(strncmp(shm_addr->user_dev2_name,"u5",2)==0) {
        meter="p2";
        ib_req2(ip,meter,"AP");
      } else if(strncmp(shm_addr->user_dev2_name,"u6",2)==0) {
        meter="p1";
        ib_req2(ip,meter,"AP");
      }
      skd_run("ibcon",'w',ip);
      skd_par(ip);
      if(ip[2]<0)
        goto Continue;
    }
    rte_sleep(100);
    if(strncmp(shm_addr->user_dev1_name,"  ",2)!=0) {
      char *meter;
      for (i=0;i<5;i++)
        ip[i]=0;
      if(strncmp(shm_addr->user_dev1_name,"u5",2)==0) {
        meter="p2";
        ib_req5(ip,meter,20);
      } else if(strncmp(shm_addr->user_dev1_name,"u6",2)==0) {
        meter="p1";
        ib_req5(ip,meter,20);
      }
      skd_run("ibcon",'w',ip);
      skd_par(ip);
      if(ip[2]<0)
        goto Continue;

      i=20;
      ib_res_ascii(buf,&i,ip);
      {
        float pwr;
        sscanf(buf,"%f",&pwr);
        shm_addr->user_dev1_value=pwr*1e6;
      }
    }
    if(strncmp(shm_addr->user_dev2_name,"  ",2)!=0) {
      char *meter;
```

```
            for (i=0;i<5;i++)
              ip[i]=0;
            if(strncmp(shm_addr->user_dev2_name,"u5",2)==0) {
              meter="p2";
              ib_req5(ip,meter,20);
            } else if(strncmp(shm_addr->user_dev2_name,"u6",2)==0) {
              meter="p1";
              ib_req5(ip,meter,20);
            }
            skd_run("ibcon",'w',ip);
            skd_par(ip);
            if(ip[2]<0)
              goto Continue;

            i=20;
            ib_res_ascii(buf,&i,ip);
            {
              float pwr;
              sscanf(buf,"%f",&pwr);
              shm_addr->user_dev2_value=pwr*1e6;
            }
          }
          break;
        default:
          ierr = -1;
        }  /* end of switch */

      End:
        ip[0] = classr;
        ip[1] = nrecr;
        ip[2] = ierr;
        memcpy(ip+3,"an",2);

        ip[4] = 0;
        goto Continue;
}
```

3.0 Block Diagrams

This section contains several block diagrams showing the relationships between the programs and physical devices in the Field System. Diagrams are presented for five general areas: initialization, normal execution, pointing programs, module checking, and data logging. The diagrams represent only the general flow of control. Several details are omitted. One omission needs to be specifically noted: The station programs: *antcn*, *sterp*, *stqkr*, and *cheks* are not shown as accessing the bus control programs: *matcn*, *mcbcn*, and *ibcon*. If any station specific hardware, including the antenna interface, is attached to any of these buses, the station programs will need to schedule them in order to access the buses. However since this is often not the case and including the information would have greatly complicated the drawings, it was omitted.

The diagrams show very little about the flow of data between the programs. The connections to external devices and the user are shown. However, none of the class I/O and shared memory accesses are shown. Generally speaking most programs that schedule other programs pass some information to them. If they schedule the programs with wait, there is usually some information returned as well. A less obvious data connection is between programs is that almost any program can send a message to *ddout* for logging and/or display. Also *boss* can potentially accept command input from other sources than *oprin*.

There are several symbols used in the diagrams. Programs are represented by boxes with square corners. Physical devices and the user are represented by boxes with rounded corners. Boxes are always labeled. The boxes are connected with arrows. Programs are always connected by single headed arrows. The tail of the arrow rests on the box for the program that does the scheduling. The head of arrow rests on the box that is scheduled. The arrow is a solid or dashed line depending on whether the scheduling is with or without wait, respectively.

Programs and devices are connected by single or double headed arrows. If the arrow has a single head its orientation depicts the direction of flow for input and output. If the arrow has a double head it indicates that there is both input and output between the program and the device. The programs *cheks*, *sterp*, are *stqkr* are shown as having both input and output with a generic device named "station." The *antcn* program is shown having both input and output with the antenna. This is not a limitation, programs that access the station might access the antenna and *antcn* might access other parts of the station than just the antenna.

The Program Descriptions section of this manual may be helpful when interpreting the roles of the various programs.

3.1 Initialization

Initialization of the Field System is started when the *fs* program loads the on-line programs from disk. The connections between *fs* and the programs are too numerous to show here and not utilize scheduling. *fs* generally runs each program as a normal UNIX program. The *incom*, *boss*, and *fserr* programs read from the disk. *boss* passes some of the information onto *matcn* and *ibcon*, which it schedules in the initialization mode. *antcn* and *mcbcn* are also scheduled in initialization mode. *mcbcn* reads its control information from the disk. *antcn* may read from the disk and access the antenna or other station devices.

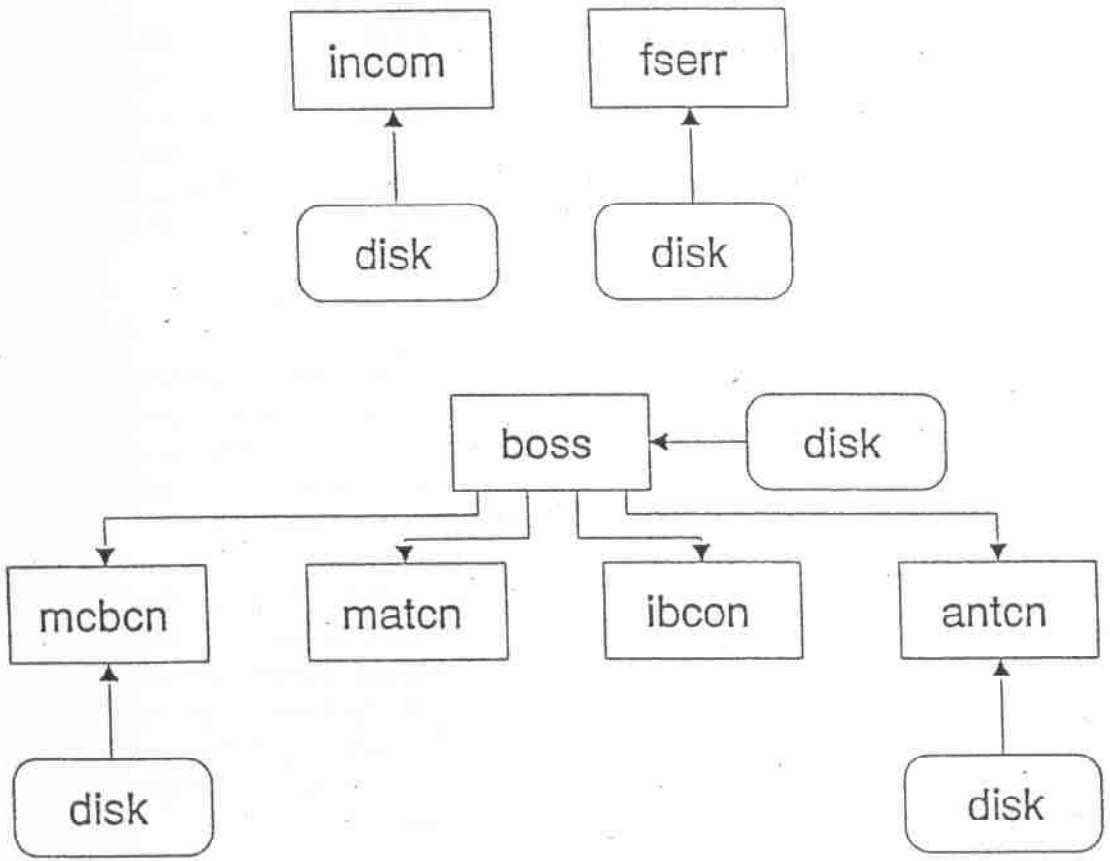

Diagram 3.1 Initialization

3.2 Normal Execution

After initialization is complete, the Field System programs process commands. The source of input are from the user (operator), via *oprin*, or the disk. An input command is parsed by *boss*, which handles it internally or schedules one of *quikr*, *quikv*, or *stqkr* to process the command. The processing programs schedule the bus control programs if necessary to access the communications bus. When processing is complete, control returns to *boss*, which looks for another command to process.

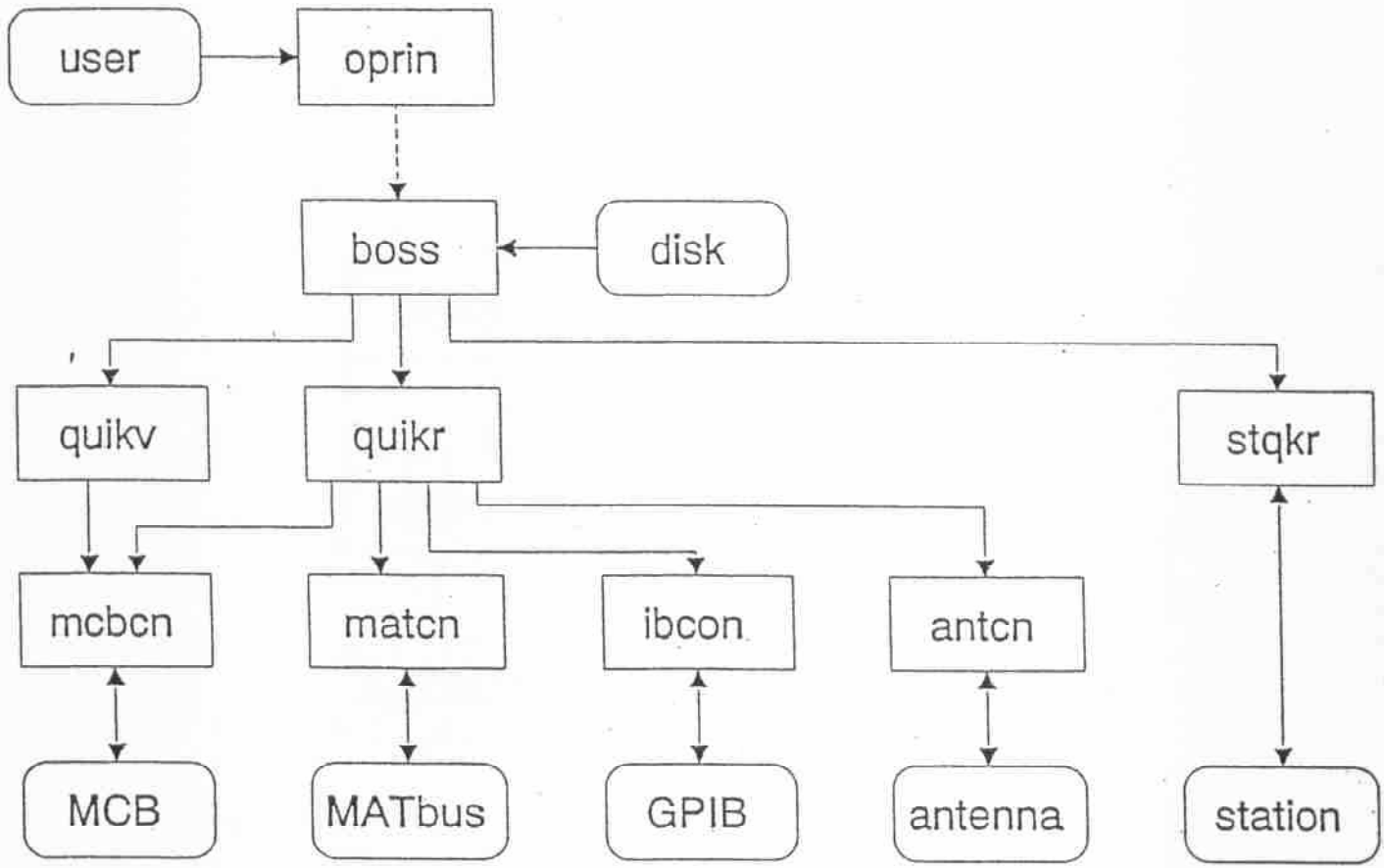

Diagram 3.2 Normal Command Processing

3.3 Pointing Programs

The pointing programs normally execute outside the normal command processing for the Field System. The *fivpt* and *onoff* programs may be started by the `fivept` and `onoff` SNAP commands or by the *aquir* program. These programs are always scheduled without wait. These programs invoke SNAP procedures by sending commands to the operator input stream in *boss*. *aquir*, *fivpt* and *onoff* use *antcn* to change the pointing offsets and/or to check whether the antenna is onsource. *fivpt* and *onoff* access the MCB and MAT-bus to sample power detectors in the racks.

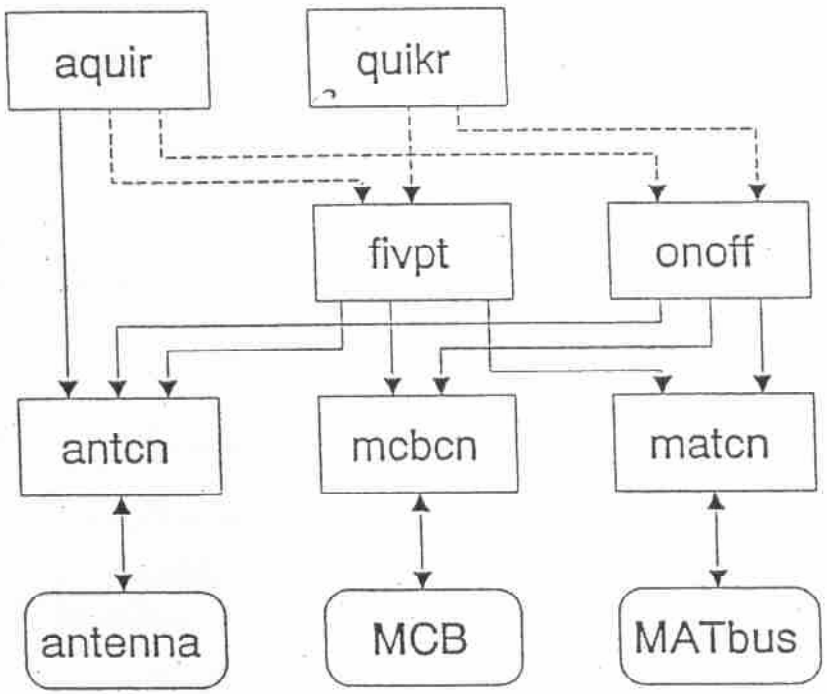

Diagram 3.3 Pointing Programs

3.4 Module Checking

Module checking occurs asynchronously from normal command execution. The program *chekr* time schedule itself every 20 seconds. It may also be scheduled by the *run* program sending a schedule request to it. Once *boss* starts executing it utilizes the bus programs to interrogate the modules. If any modules are found in a state different than the last commanded, an error message is sent to *ddout*. If a station checking program has been defined, *chekr* will run this program with wait at the end of each cycle.

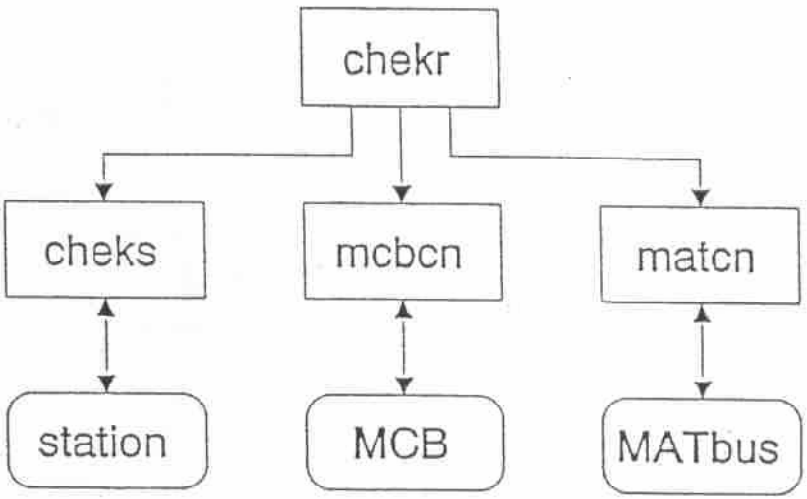

Diagram 3.4 Module Checking

3.5 Data Logging

ddout handles the display and writing of the log file to disk. It schedules the *fserr* program to retrieve the message associated with any error codes. The *sterp* program is an optional station program that may be used to report errors using some other device, perhaps a buzzer, or some other display.

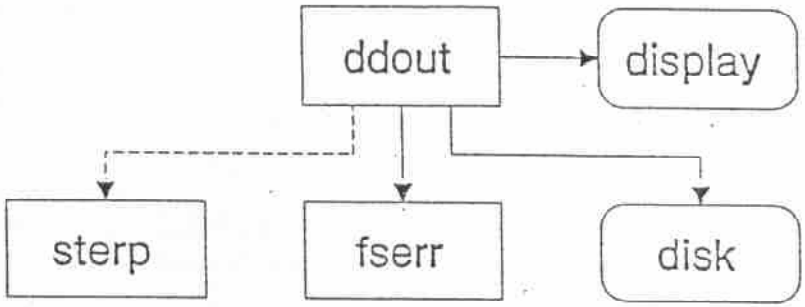

Diagram 3.5 Data Logging

Code: Intermediate

May 2019 TOW

Ed Himwich, NVI/GSFC

1. Connecting to FS resources

1.1. Necessary for FS libraries and utilities to work

1.2. The first thing a program should do is

1.2.1. In C:

1.2.1.1. Use `setup_ids()`

Call this routine only **once** per program execution, do not call for every program scheduling (as opposed to program execution, see section **2**. below), do not call in a loop and do not call in each subroutine. Calling more than once per execution should be benign, but apparently there is a memory leak in kernel support for shared memory.

1.2.2. In FORTRAN:

1.2.2.1. call `setup_fscom`

Call this routine only **once** per program execution, do not call for every program schedule (as opposed to program execution, see section **2**. below), do not call in a loop and do not call in each subroutine. Calling more than once per execution should be benign, but apparently there is a memory leak in kernel support for shared memory.

1.2.2.2. call `read_fscom`

1.2.2.3. When waking up must call `read_quikr` to refresh `fscom_quik`

2. Program Scheduling

2.1. This is a form of inter-program communication. It is used pass control from "parent" to "child" programs. The "child" programs do not terminate, but can return control to "parent", also known as "co-routines." The "child" can also run asynchronously. Most FS programs run persistently rather than starting and stopping as normal UNIX programs would.

2.2. Flow

2.2.1. All programs initialize by connecting to FS resources (see above)

2.2.2. Most wait to be scheduled, `skd_wait()`.

2.3. In C:

2.3.1.	`skd_run()`	schedule a program with run parameters
2.3.2.	`skd_par()`	retrieve run parameters from a returning child
2.3.3.	`skd_run_arg()`	schedule program with an ASCII string
2.3.4.	`skd_wait()`	wait for someone to schedule me with optional time-out and return run parameters
2.3.5.	`skd_arg()`	retrieve n-th ASCII blank delimited argument from father
2.3.6.	`skd_chk()`	check whether I've been scheduled
2.3.7.	`skd_end()`	wake up my father if I have one, send run parameters
2.3.8.	`dad_pid()`	get my father's pid, `0` if no father

2.4. In FORTRAN:

2.4.1.	`run_prog()`	accesses `skd_run`
2.4.2.	`wait_prog()`	`skd_wait` with no time-out
2.4.3.	`wait_abst()`	`skd_wait` with time-out at absolute time
2.4.4.	`wait_abstd()`	`skd_wait` with time-out at absolute time including day of year
2.4.5.	`wait_relt()`	`skd_wait` with relative wait time
2.4.6.	`get_arg()`	`skd_arg`
2.4.7.	`rmpar()`	`skd_par`

3. Error handling and class buffer exchange

3.1. Structured around long (`integer*4`) five element interprogram communication array

3.1.1.	First Element	- class number holding messages
3.1.2.	Second	- number of class records
3.1.3.	Third	- Error number if non-zero
3.1.4.	Fourth	- first two characters of error type
3.1.5.	Fifth	- first two characters of additional error type or additional binary information

3.2. Errors are detected at the lowest level and the routines and process return until the highest-level routine, usually `boss` or `chekr` logs the error

3.3. Asynchronous - errors exist without context, this has good and bad points

3.4. If there is an error don't leave data in class numbers, unless you want it logged, it is ASCII and `boss` is the top-level scheduler.

3.5. Normal path

3.5.1. `boss` passes command to `quikr` in a class buffer with save bits on

3.5.2. `quikr` parses command, generates class message for `matcn`

3.5.3. `matcn` processes buffers, generating responses which go into class buffers, and or an error

3.5.4. `quikr` parses response from `matcn` and generates log entries in class buffers and sends them or `matcn` errors to boss

3.5.5. `boss` accepts log entries or errors and sends them to the log entry system, an error automatically removes a command for time list

4. Setting up station help pages and error messages

4.1 Text files go in `/usr2/st/help` and are in the form *xxx*`.___` (three underscores), where *xxx* corresponds to the *xxx* used in `help=`*xxx*, could be a command name or other token

4.1. Error messages go in `/usr2/control/sterr.ctl`, compare to `/usr2/fs/control/fserr.ctl`

4.2. Three lines per error message, where *XX* is the two-letter error mnemonic (in caps), *nnn* is the numeric error:

```
""
XX -nnn
message
```

5. Station SNAP Commands

5.1. Linked to the station specific `stqkr` program

5.2. See example code in `/usr2/fs/st.default/st-0.0.0./stqkr`

5.3. FORTRAN should be avoided

5.4. Selection of code in `stqkr` for each command is determined by values in `stcmd.ctl` file

5.5. Coding of sample C based SNAP VLBA rack command `bbc`*nn* in `/usr2/fs/quikv/bbc.c` provides an example of how to implement

5.6. `stqkr` example

`stqkr.c`

```
/* stqkr - C version of station command controller
 */
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

#include "../../fs/include/fs_types.h"
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/params.h"
#include "../../fs/include/fscom.h"
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"     /* shared memory pointer */

struct stcom *st;
struct fscom *fs;

#define MAX_BUF   257

main()
{
    long ip[5];
    int isub,itask,idum,ierr,nchars,i;
    char buf[MAX_BUF];
    struct cmd_ds command;
    int cls_rcv(), cmd_parse();
    void skd_wait();

/* Set up IDs for shared memory, then assign the pointer to
   "fs", for readability.
*/
  setup_ids();
  fs = shm_addr;
  setup_st();
  st = stm_addr;

loop:
      skd_wait("stqkr",ip,(unsigned) 0);
      if(ip[0]==0) {
        ierr=-1;
        goto error;
      }
      ierr=0;

      nchars=cls_rcv(ip[0],buf,MAX_BUF,&idum,&idum,0,0);
      if(nchars==MAX_BUF && buf[nchars-1] != '\0' ) { /*does it fit?*/
        ierr=-2;
        goto error;
      }
                                  /* null terminate to be sure */
      if(nchars < MAX_BUF && buf[nchars-1] != '\0') buf[nchars]='\0';

      if(0 != (ierr = cmd_parse(buf,&command))) { /* parse it */
        ierr=-3;
        goto error;
      }


      isub = ip[1]/100;
      itask = ip[1] - 100*isub;
```

```
      switch (isub) {
         case 1:
/*                                  antenna echo function */
            ierr=0;
            aecho(&command,ip);
            break;

         case 2:
/*                                  WX function */
            ierr=0;
            wx(&command,ip);
            break;

         case 3:
/*                                  IF Attenuator function */
            ierr=0;
            ifatt(&command,itask,ip);
            break;

         default:
            ierr=-4;
            goto error;
      }
      goto loop;

error:
      for (i=0;i<5;i++) ip[i]=0;
      ip[2]=ierr;
      memcpy(ip+3,"st",2);
      goto loop;
}
```

ifatt.c

```
/* Tsukuba if att snap command */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#include "../../fs/include/params.h"
#include "../../fs/include/fs_types.h"
#include "../../fs/include/fscom.h"         /* shared memory definition */
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"

void ifatt(command,itask,ip)
struct cmd_ds *command;               /* parsed command structure */
int itask;
long ip[5];                           /* ipc parameters */
{
      int ilast, ierr, ichold, i, count, type;
      char *ptr;
      struct ifatt_cmd lcl;

      char *arg_next();

      void ifatt_dis();

      void skd_run(), skd_par();      /* program scheduling utilities */

      ichold= -99;                    /* check vlaue holder */

      ip[0]=ip[1]=0;

      if (command->equal != '=') {            /* read module */
        ifatt_req_q(ip);
        goto k4con;
      }
      else if (command->argv[0]==NULL) goto parse;  /* simple equals */
      else if (command->argv[1]==NULL) /* special cases */
        if (*command->argv[0]=='?') {
          ifatt_dis(command,itask,ip);
          return;
         }

/* if we get this far it is a set-up command so parse it */

parse:
      ilast=0;                                      /* last argv examined */
      memcpy(&lcl,&stm_addr->ifatt,sizeof(lcl));

      count=1;
      while( count>= 0) {
        ptr=arg_next(command,&ilast);
        ierr=ifatt_dec(&lcl,&count, ptr);
        if(ierr !=0 ) goto error;
      }

/* all parameters parsed okay, update common */
      /*
      ichold=shm_addr->check.k4rec.check;
      shm_addr->check.k4rec.check=0;
      */
      memcpy(&stm_addr->ifatt,&lcl,sizeof(lcl));

/* format buffers for k4con */
```

```
      ifatt_req_c(ip,&lcl);

k4con:
      skd_run("ibcon",'w',ip);
      skd_par(ip);
      /*
      if (ichold != -99) {
       shm_addr->check.k4rec.state=TRUE;
       if (ichold >= 0)
         ichold=ichold % 1000 + 1;
       shm_addr->check.k4rec.check=ichold;
      }
      */
      if(ip[2]<0) {
       cls_clr(ip[0]);
       ip[0]=ip[1]=0;
       return;
      }

      ifatt_dis(command,itask,ip);
      return;

error:
      ip[0]=0;
      ip[1]=0;
      ip[2]=ierr;
      memcpy(ip+3,"st",2);
      return;
}
```

ifatt_dis.c

```
/* Tsukuba if att display */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#include "../../fs/include/params.h"
#include "../../fs/include/fs_types.h"
#include "../../fs/include/fscom.h"
#include "../../fs/include/shm_addr.h"
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"

#define MAX_OUT 256

void ifatt_dis(command,itask,ip)
struct cmd_ds *command;
int itask;
long ip[5];
{
  struct ifatt_cmd lclc;
  int kcom, i, ierr, count;
  char output[MAX_OUT];

  kcom= command->argv[0] != NULL &&
    *command->argv[0] == '?' && command->argv[1] == NULL;

  if ((!kcom) && command->equal == '=') {
    if(ip[0]!=0) {
      cls_clr(ip[0]);
      ip[0]=0;
    }
    ip[1]=0;
    return;
  } else if (kcom){
    memcpy(&lclc,&stm_addr->ifatt,sizeof(lclc));
  } else {
    ifatt_res_q(&lclc,ip);
    if(ip[1]!=0) {
      cls_clr(ip[0]);
      ip[0]=ip[1]=0;
    }
    if(ip[2]!=0) {
      ierr=ip[2];
      goto error;
    }
  }

   /* format output buffer */

  strcpy(output,command->name);
  strcat(output,"/");

  count=0;
  while( count>= 0) {
    if (count > 0) strcat(output,",");
    count++;
    ifatt_enc(output,&count,&lclc);
  }
  if(strlen(output)>0) output[strlen(output)-1]='\0';

  for (i=0;i<5;i++) ip[i]=0;
  cls_snd(&ip[0],output,strlen(output),0,0);
  ip[1]=1;
```

```
  return;

 error:
  ip[0]=0;
  ip[1]=0;
  ip[2]=ierr;
  memcpy(ip+3,"ki",2);
  return;
}
```

ifatt_util.c

```
/* Tsukuba IF ATT buffer parsing utilities */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <limits.h>
#include <math.h>

#include "../../fs/include/macro.h"
#include "../../fs/include/params.h"
#include "../../fs/include/fs_types.h"
#include "../../fs/include/fscom.h"         /* shared memory definition */
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"

static char device[]={"if"};           /* device menemonics */

#define MAX_BUF 512

int ifatt_dec(lcl,count,ptr)
struct ifatt_cmd *lcl;
int *count;
char *ptr;
{
    int ierr, arg_int();

    ierr=0;
    if(ptr == NULL) ptr="";

    if (*count >0 && *count <5) {
      ierr=arg_int(ptr,&lcl->atten[*count-1],0,FALSE);
      if(ierr==0 && (lcl->atten[*count-1]<0 || lcl->atten[*count-1]>81))
        ierr=-200;
    } else
      *count=-1;

    if(ierr!=0) ierr-=*count;
    if(*count>0) (*count)++;
    return ierr;
}

void ifatt_enc(output,count,lcl)
char *output;
int *count;
struct ifatt_cmd *lcl;
{
  int ivalue, type;

  output=output+strlen(output);

  if(*count > 0 && *count < 5 ) {
    sprintf(output,"%02d",lcl->atten[*count-1]);
  } else
    *count=-1;

  return;
}

ifatt_req_q(ip)
long ip[5];
{
 ib_req7(ip,device,31*2-1+2,"rout:clos? (@1(0:15),2(0:14))");

}
```

```
ifatt_req_c(ip,lclc)
long ip[5];
struct ifatt_cmd *lclc;
{
  char buffer[120];
  unsigned short word[2];
  int i;

  ib_req2(ip,device,"ROUT:DRIV:ON:ALL");

  word[0]=word[1]=0;

  for(i=0;i<4;i++) {
    int value=lclc->atten[i];
    int low,up;
    if(value<80) {
      if(value%10 <8)
        low=value%10;
      else
        low=0x8 | (value%10-4);
      up=value/10;
    } else if(value == 80){
      low=0xE;
      up=0x7;
    }  else if(value == 81) {
      low=0xF;
      up=0x7;
    }
    word[0]|=low << (i*4);
    word[1]|=up  << (i*4);
  }


  if(word[0]!=0xFFFF || word[1] !=0x7777) {
    strcpy(buffer,"ROUT:OPEN (@");
    if(word[0]!=0xFFFF) {
      strcat(buffer,"1(");
      for (i=0;i<16;i++)
        if((word[0] & (1<<i)) == 0)
          sprintf(buffer+strlen(buffer),"%d,",i);
      strcpy(buffer+strlen(buffer)-1,")");
      if(word[1]!=0x7777)
        strcat(buffer,",");
    }

    if(word[1]!=0x7777) {
      strcat(buffer,"2(");
      for (i=0;i<16;i++)
        if((word[1] & (1<<i)) == 0)
          sprintf(buffer+strlen(buffer),"%d,",i);
      strcpy(buffer+strlen(buffer)-1,")");
    }
    strcat(buffer,")");

    ib_req2(ip,device,buffer);
  }

  if(word[0]!=0 || word[1] !=0) {
    strcpy(buffer,"ROUT:CLOS (@");
    if(word[0]!=0) {
      strcat(buffer,"1(");
      for (i=0;i<16;i++)
        if((word[0]& (1<<i)) != 0)
          sprintf(buffer+strlen(buffer),"%d,",i);
      strcpy(buffer+strlen(buffer)-1,")");
      if(word[1]!=0)
        strcat(buffer,",");
    }

    if(word[1]!=0) {
      strcat(buffer,"2(");
```

```
      for (i=0;i<16;i++)
        if((word[1]& (1<<i)) != 0)
          sprintf(buffer+strlen(buffer),"%d,",i);
      strcpy(buffer+strlen(buffer)-1,")");
    }
    strcat(buffer,")");
    ib_req2(ip,device,buffer);
  }

}

ifatt_res_q(lclc,ip)
struct ifatt_cmd *lclc;
long ip[5];
{
  char buffer[MAX_BUF];
  int i,max;
  unsigned short word[2];

  max=sizeof(buffer);
  ib_res_ascii(buffer,&max,ip);
  if(max < 0) {
    ip[2]=-1;
    return;
  }

  word[0]=word[1]=0;

  for(i=0;i<31;i++) {
    int ibit;
    sscanf(buffer+i*2,"%d",&ibit);
    if(ibit!=0)
      word[i/16]|=1<<(i%16);
  }

  for (i=0;i<4;i++) {
    unsigned char x;
    x=(word[0]>>(i*4)) & 0xf;
    lclc->atten[i]=x &0x7;
    if((x&0x8)!=0)
      lclc->atten[i]+=4;
    x=(word[1]>>(i*4)) & 0x7;
    lclc->atten[i]+=x*10;
  }

  return;
}
```

wx.c

```
/* wx command
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#include "../../fs/include/fs_types.h"
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/params.h"
#include "../../fs/include/fscom.h"
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"      /* shared memory pointer */

extern struct stcom *st;
extern struct fscom *fs;

#define MAX_OUT 256

void wx(command,ip)
struct cmd_ds *command;
long ip[5];

{
  char  output[MAX_OUT];
  int which,i;
  long time;

  which=stm_addr->wx.which;
  if(which <0 || 1 < which) {
    ip[2]=-96;
    goto error;
  }

  rte_rawt(&time);
  if(stm_addr->wx.time[which]< time-6001) {
    ip[2]=-95;
    goto error;
  }
  strcpy(output,command->name);
  strcat(output,"/");

  sprintf(output+strlen(output),"%.1f,%.1f,%.1f",
          stm_addr->wx.temp[which],
          stm_addr->wx.pres[which],
          stm_addr->wx.humi[which]);

  shm_addr->tempwx=stm_addr->wx.temp[which];
  shm_addr->humiwx=stm_addr->wx.humi[which];
  shm_addr->preswx=stm_addr->wx.pres[which];

  for (i=0;i<5;i++) ip[i]=0;
  cls_snd(&ip[0],output,strlen(output),0,0);
  ip[1]=1;

  return;

error:
      ip[0]=0;
      ip[1]=0;
      memcpy(ip+3,"st",2);
      return;

}
```

Code: Advanced

May 2019 TOW

Ed Himwich, NVI/GSFC

1. Class I/O

1.1. Emulation of HP system feature with System V messages
1.2. Class numbers are FIFO mailboxes,
1.3. In C:
  1.3.1. `cls_snd()` send a class message
  1.3.2. `cls_rcv()` receive a class message
  1.3.3. `cls_clr()` clear a class number

1.4. In FORTRAN:
  1.4.1. `put_buf()` send a class message
  1.4.2. `get_buf()` receive a class message
  1.4.3. `clrcl()` clear a class number

1.5. Class number word:
  1.5.1. long (`integer*4`) variable
  1.5.2.

Bits	Value
31-16	zero
15	1=no-wait
14	1=save buffer
13	1=save class
12-0	class number

2. LOGIT routines

2.1. Used for reporting errors, send text to log
2.2. Not normally needed since BOSS normally logs these errors
2.3. See examples in existing code
2.4. In C:
  2.4.1. `putpname()` insert external program name for external messages
  2.4.2. `logit()` ASCII message OR Error number, mnemonic
  2.4.3. `logita()` ASCII message OR Error number, mnemonic, plus additional character code
  2.4.4. `logite()` ASCII message OR Error number, mnemonic, plus additional text for S2 errors

2.4.5. `logit_nd()` log information without operator display

2.5. In FORTRAN:

2.5.1. `pname()` Insert external program name for external messages

2.5.2. `logit2()` log text message from external program

2.5.3. `logit2_ch()` log text message from external program input is character variable

2.5.4. `logit3()` log ASCII text, used primarily by `newlg.f` for logging header information

2.5.5. `logit4()` log ASCII message with calling procedure name

2.5.6. `logit4d()` log ASCII message with date information

2.5.7. `logit4_ch()` log character type message

2.5.8. `logit5()` log the opening line of a new log

2.5.9. `logit6()` log error, all args except the last two are zero.

The last two arguments are:

2.5.9.1. error integer
2.5.9.2. error mnemonic: ASCII

2.5.10. `logit6c()` log error, all args except the last two are zero.

The last two arguments are:

2.5.10.1. error integer
2.5.10.2. error mnemonic, character value

2.5.11. `logit7()` log error, all arguments except the last four are zero.

The last four arguments are:

2.5.11.1. + or -/0 for last argument is 2 characters
2.5.11.2. error number
2.5.11.3. ASCII mnemonic
2.5.11.4. last argument depends on pervious arguments, but `0` is ignored)

2.5.12. `logit7cc()` same as `logit7()`, but last two args are character

2.5.13. `logit7ci()` same as `logit7()`, but next to last argument is character

2.5.14. `logit7ic()` same as `logit7()`, but last argument is

character

3. shared memory for the FS

3.1. FORTRAN

Defined in `/usr2/fs/include/fscom.i`, includes:

3.1.1. `fscom_init.i`

3.1.1.1. initialization values from `sincom`

3.1.1.2. set only once and is read in by `read_fscom`

3.1.2. `fscom_quik.i`

3.1.2.1. `quikr` defined values

3.1.2.2. may be changed every time `quikr` runs and needs to be refreshed with `read_quikr` every time a program is scheduled

3.1.3. `fscom_dum.i`

3.1.3.1. FORTRAN copy of C stored data

3.1.3.2. C data accessed by `fs_set/fs_get` routines in `/usr2/fs/newlb/prog.c`

3.2. In C:

The shared memory area is available through the pointer

3.2.1. `extern struct fscom *shm_addr;`

defined in `/usr2/fs/include/shm_addr.h`

4. Shared memory for the station software

4.1. Example in `/usr2/fs/st.default/stlib/stm_util.c`

4.1.1. Needs initialization by `stalloc` program (analog of `fsalloc`)

4.2. Supports one C area and two FORTRAN areas

4.3. In C:

The shared memory area is available through the pointer

```
extern struct stcom *stm_addr;
```

defined in `/usr2/fs/st.default/st-1.0.0/include/stm_addr.h`

4.4. FORTRAN
4.5. Please avoid using FORTRAN shared memory
4.6. For FORTRAN primitive C-based functions are provided
  4.6.1. `stm_map()` defines up to two FORTRAN areas to managed
  4.6.2. `stm_read()` refreshes from C area
  4.6.3. `stm_write()` copies to C area
  4.6.4. These can be used to build up higher level function like those for the FS found in `/usr2/fs/flib/`: `setup_fscom.f`, `read_*.f`, `write_*.f`

5. Semaphores

5.1. Provide a means for controlling access to resource, literal or "virtual".
5.2. In C there are two levels of semaphores:
  5.2.1. Numbered
    5.2.1.1. Fixed number `SEM_NUM` (32)
    5.2.1.2. Access functions
      5.2.1.2.1. `sem_take()` take a semaphore, wait until available if necessary
      5.2.1.2.2. `sem_put()` release a semaphore
      5.2.1.2.3. `sem_nb()` take if available (non blocking), return `-1` if not available
      5.2.1.2.4. `sem_value()` return value

  5.2.2. Named
    5.2.2.1. Fixed number `SEM_NUM` (32), disjoint from Numbered semaphores uses a five-character name to identify (blank pad to right)
    5.2.2.2. once a name is used, it is defined until next boot
    5.2.2.3. Access functions
      5.2.2.3.1. `nsem_take()` take a semaphore, wait until available if necessary
      5.2.2.3.2. `nsem_put()` release a semaphore
      5.2.2.3.3. `sem_test()` return state, `1` = taken

5.3. In FORTRAN
  5.3.1. only the named semaphores are supported
  5.3.2. Access functions:
    5.3.2.1. `rn_take()` take a semaphore, wait until available if necessary
    5.3.2.2. `rn_put()` release a semaphore

5.3.2.3. `rn_test()` return state, `.true.` = taken

5.4. Defined named semaphores (blank pad to right to get five characters)

5.4.1. `fs` FS is active

5.4.2. `fsctl` schedule needs control with the next 2 seconds, coordinates hardware access of `boss`, `chekr`, `setcl`, and `fmset`

5.4.3. `fivpt` `fivpt` is active

5.4.4. `onoff` `onoff` is active

5.4.5. `pfmed` `pfmed` is active

5.4.6. `pcalr` `pcalr` is active

5.4.7. `lvdt` access to the LVDT tape head motion controller

6. `pcald`

6.1. Used to collect phase-cal data

6.2. Sample stub in `/usr2/fs/st.default/pcald`

6.3. Should use `fsctl` named semaphore before accessing hardware

7. Asynchronous programs

7.1. Three options:

7.1.1. Started at boot time

7.1.2. Started from `stpgm.ctl` at FS start time

7.1.3. Started by `antcn`

7.2. Option (2) is preferred because if the FS is restarted the system is completely re-initialized

7.3. Option (1) is necessary if the program needs to be running even when the FS isn't

7.4. Option (3) can simulate option (2) if it runs periodically, say every second, by checking the `fs` named semaphores to see if the FS is running

7.5. Asynchronous WX data Retrieval Example

```
/* wxget - retrieve wx data asynchronously
 */

#include <string.h>
#include <stdio.h>
#include <math.h>

#include "../../fs/include/dpi.h"
#include "../../fs/include/fs_types.h"
#include "../../fs/include/shm_addr.h"      /* shared memory pointer */
#include "../../fs/include/params.h"
#include "../../fs/include/fscom.h"
#include "../../fs/include/pmodel.h"

#include "../include/stparams.h"
#include "../include/stcom.h"
#include "../include/stm_addr.h"      /* shared memory pointer */

main(argc,argv)
int argc;
char **argv;
{
  int max,which;
  long ip[5],time;
  char buffer[28], bufs[10];
  float temp,pres,humi;

  setup_ids();
  setup_st();

  stm_addr->wx.which=-1;

  while(TRUE) {
    ip[0]=ip[1]=0;
    ib_req12(ip,"wx");
    skd_run("ibcon",'w',ip);
    skd_par(ip);
    if(ip[2] < 0) {
       if(ip[0]!=0)
        cls_clr(ip[0]);
      logita(NULL,ip[2],ip+3,ip+4);
      continue;
    }
    rte_sleep(3);

    ip[0]=ip[1]=0;
    ib_req2(ip,"wx","S1D000X0/0*");
    skd_run("ibcon",'w',ip);
    skd_par(ip);
    if(ip[2] < 0) {
      if(ip[0]!=0)
       cls_clr(ip[0]);
      logita(NULL,ip[2],ip+3,ip+4);
      continue;
    }
    rte_sleep(1001);

    ip[0]=ip[1]=0;
    ib_req5(ip,"wx",28);
    skd_run("ibcon",'w',ip);
    skd_par(ip);
    if(ip[2] < 0) {
      if(ip[0]!=0)
       cls_clr(ip[0]);
      logita(NULL,ip[2],ip+3,ip+4);
      continue;
```

```
    }
    rte_rawt(&time);

    max=sizeof(buffer);
    ib_res_ascii(buffer,&max,ip);

    memcpy(bufs,buffer+11,6);
    bufs[6]=0;
    if(1!=sscanf(bufs,"%f",&temp)) {
      logita(NULL,-99,"st","  ");
      continue;
    }
    memcpy(bufs,buffer+17,3);
    bufs[3]=0;
    if(1!=sscanf(bufs,"%f",&humi)) {
      logita(NULL,-98,"st","  ");
      continue;
    }
    memcpy(bufs,buffer+20,6);
    bufs[6]=0;
    if(1!=sscanf(bufs,"%f",&pres)) {
      logita(NULL,-97,"st","  ");
      continue;
    }

    if(stm_addr->wx.which < 0 ||1 < stm_addr->wx.which )
      which=0;
    else
      which=1-stm_addr->wx.which;

    stm_addr->wx.temp[which]=temp;
    stm_addr->wx.pres[which]=pres;
    stm_addr->wx.humi[which]=humi;
    stm_addr->wx.time[which]=time;
    stm_addr->wx.which=which;
    /*
    printf(" max %d buffer %s time %d which %d \n",max,buffer,time,which);

    printf(" temp %f pres %f humi %f\n",temp,pres,humi);
    */
  }
}
```

Cable-wrap:
What it is and how to deal with it

Ed Himwich, John Gipson

May 2015 TOW

Definitions

- Azimuth range of motion consists of two regions:
  - Overlapped region, which can be reached two ways
  - Neutral region, which can be reached only one way
- Overlapped region has two “wraps” (viewed from above)
  - Counter-clockwise (CCW or W)
  - Clockwise (CW or C)
- Neutral region is also conventionally called a “wrap”
- Limited to less than two full turns of travel
- Minimum travel is one full turn (plus 4°)
- SKED specifies azimuth limits in positive angles only
  - Therefore a range of -90° -> +450° is represented as 270° -> +810°

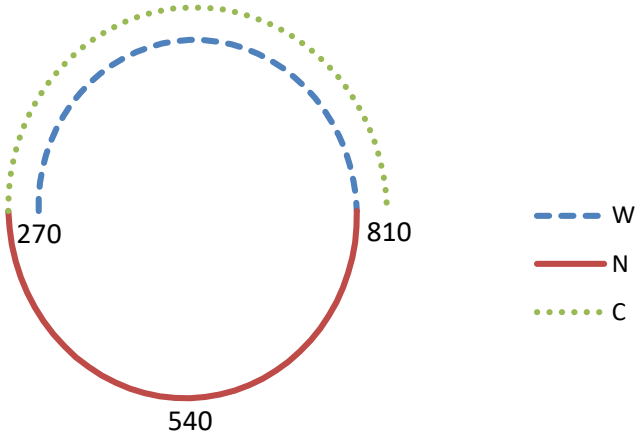

Figure 1. Example of cable-wrap. This is viewed from above. The azimith range runs from 270° (-90°) to 810° (+450°). The center of travel is at 540° (+180°). The W arc is the CCW wrap. The C arc is the CW wrap. The N arc is the Neutral "wrap".

SKED Scheduling I

- SKED always assumes the antenna goes the shortest way when there is more than one choice
- This is more complicated than expected
  - Wrap limits are not accurately know
  - Exclusions zones (±2°) at each limit/transition zone are used to eliminate ambiguous cases
- Sources may move beyond a limit while being observed
  - Exclude sources that would enter exclusion zone inside wrap limit before observation ends

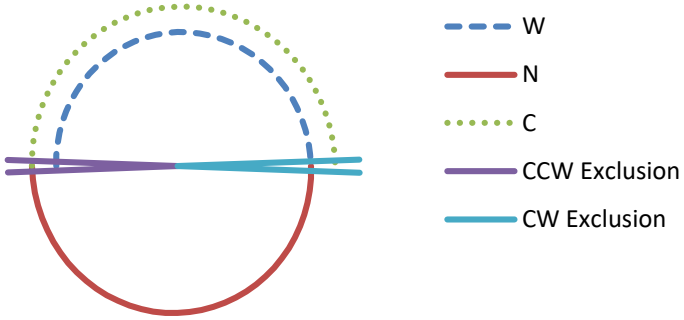

Figure 2. CCW and CW Scheduling Exclusion Zones. The zones are ±2° around the CCW and CW limits/transition points, respectively.

SKED Scheduling II

- We would like to minimize exclusion zone impact
  - Zones cover about 2% of the sky total
- They are needed because they eliminate uncertainty about which direction the antenna should go
- Don’t need to avoid a zone if:
  - More than 180° from the corresponding limit
- Three examples:
  - (1) Slewing from CCW wrap
  - (2) Slewing from Neutral wrap within 180° CCW limit
  - (3) Slewing from Neutral wrap more 180° from both limits
  - Cases (1) and (2) have corresponding cases for CW

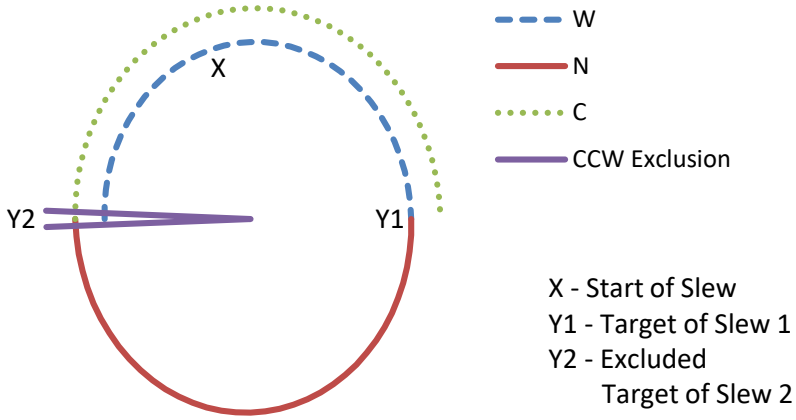

Figure 3. Two example slews from the CCW wrap. These examples illustrate that only the CCW exclusion needs to be considered in this case. There is no opportunity for confusion about the CW limit/transition point.

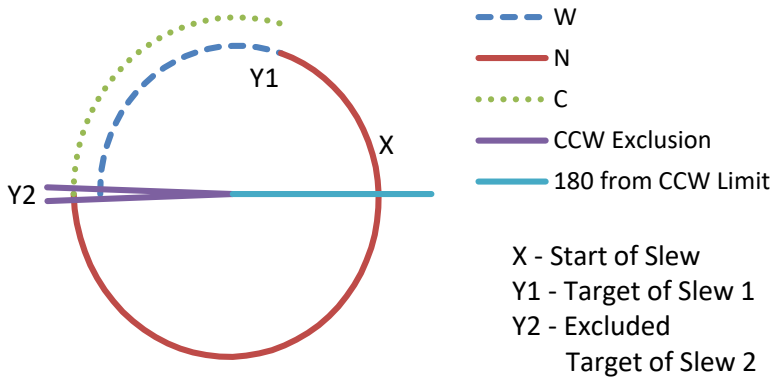

Figure 4. Two example slews from the Neutral wrap. From a point in Neutral wrap that is close to the CCW limit (within 180°), only the CCW exclusion needs to be considered.

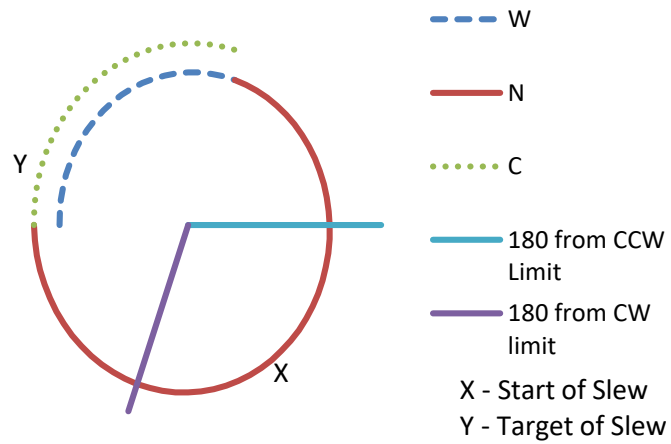

Figure 5. Slew from Neutral wrap. From a point in Neutral wrap that far from both limits (more than 180°), neither exclusion zone needs to be considered.

SKED Scheduling III

- Need to exclude slews that are 180°±2°
  - Variations in timing can make direction unpredictable
  - If source enters the exclusion, before the slew is over it should be excluded well

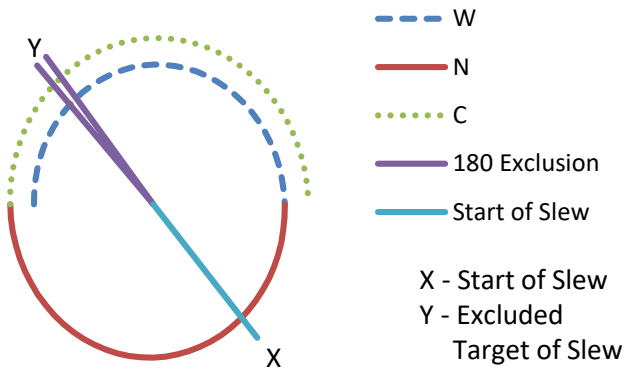

Figure 6. Scheduling exclusion zone to prevent slew of 180° (±2°) when there are two ways to reach the target.

Field System I

- source=… command has a cable-wrap parameter for Az-El antennas
  - Fifth position, values: CCW, CW, Neutral, and null
  - Each value specifies the wrap to use
  - Null (empty) means “fastest” if two choices exist
    - Normally not used in schedules
  - Information for the operator
- FS just passes this parameter to *antcn*
  - *antcn* implements this parameter for the antenna, if it can
  - It can be used to make schedule execution more robust

Field System II

- Problems arise if commanded wrap does not agree with current source wrap, this may occur because:
  - Limits in schedule don't agree with the antenna
  - source=… command is not executed at the expected time
- Choice of direction of slew does not depend on where the antenna is, only where the target source is

Field System III

- Four examples:
  - (1) Commanded wrap is Neutral but source is in overlap near CCW end of Neutral
    - Move to source on CCW wrap
  - (2) Commanded wrap is Neutral but source is in overlap near CW end of Neutral
    - Move to source on CW wrap
  - (3) Commanded wrap is CCW but source is in Neutral region near CW end of Neutral
    - Move to CCW limit on CCW wrap
  - (4) Commanded wrap is CCW but source is in Neutral region near CCW end of Neutral
    - Move to source in Neutral wrap
  - Cases (3) and (4) have mirror images for CW command wrap
    - Invert all CWs and CCWs in conditions and actions

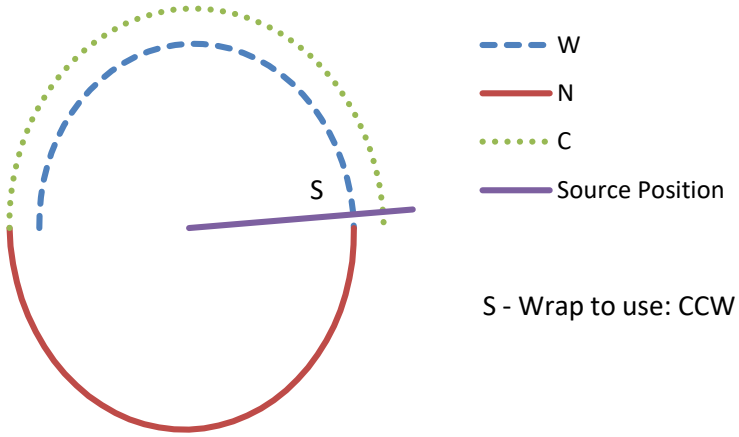

Figure 7. Commanded wrap is Neutral, I. In this case, when the source is command it is in the Overlap region near the CCW end of the Neutral region. The correct action is move to the source on the CCW wrap, regardless of the start position.

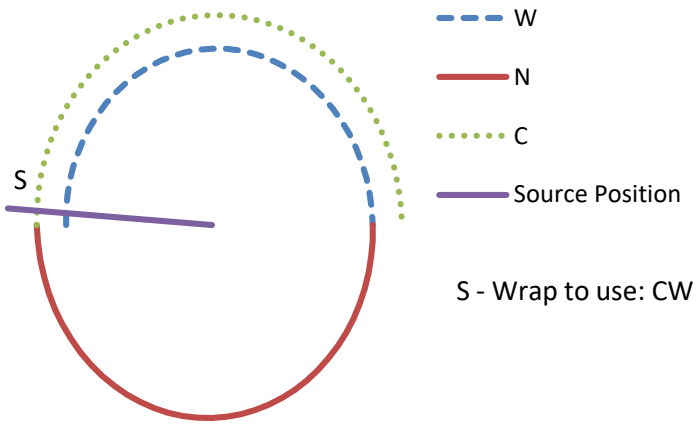

Figure 8. Commanded wrap is Neutral, II. In this case, when the source is command it is in the Overlap region near the CW end of the Neutral region. The correct action is move to the source on the CW wrap, regardless of the start position.

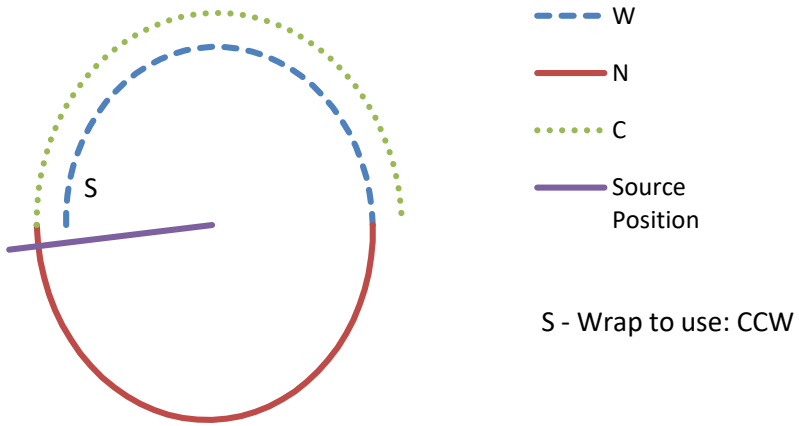

Figure 9. Commanded wrap is CCW, I. In this case, when the source is command it is in the Neutral region near the CW end of the Neutral region. The correct action is move to the CCW limit on the CCW wrap, regardless of the start position.

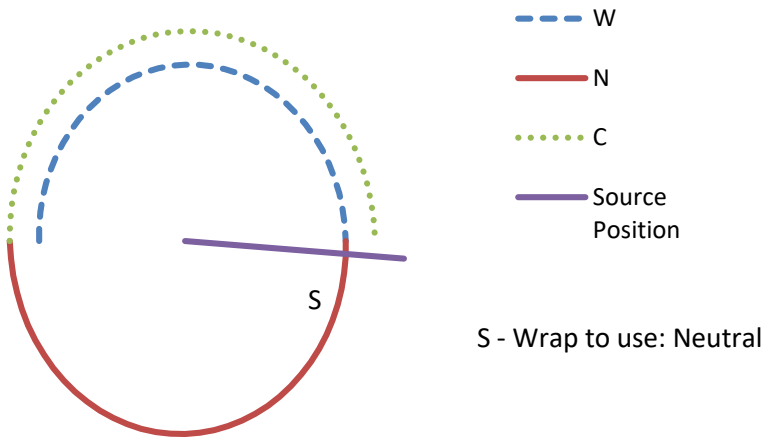

Figure 10. Commanded wrap is CCW, II. In this case, when the source is command it is in the Neutral region near the CCW end of the Neutral region. The correct action is move to the source in the Neutral region, regardless of the start position.

Operations

- When starting or restarting schedule make sure the antenna has time to reach source on requested wrap
  - If source command cable-wrap parameter is not implemented it may be necessary to “walk” the antenna to the correct wrap
  - Use schedule listing or source=… command to determine correct wrap
- Once on the correct wrap, no wrap errors should occur, if they do:
  - Typically visible as the antenna “unwrapping” while trying to get to a source
  - Action may help correct the problem if it is noticed soon enough, but otherwise it may “self heal” in a scan or two, this depends on the schedule
  - Report problem with:
    - log comments
    - Stop message
    - Message to ivscc@ivscc.gsfc.nasa.gov
- If a source is command a long time (hours) before being observed and will change wraps before being observed, intervention may (probably) be require to reach the correct wrap.