ULTRA SHEAR DP↔DA DATA STRUCTURE CHANGES
21 JUNE 1995

This document describes changes in existing data structures for the Ultra Shear system, as well as new data structures. Changes in data structures are in bold type. No seed data structures are transmitted over the DP/DA link, for the descriptions of these data structures see the "QUANTERRA ULTRA SHEAR SEED+ DATA STRUCTURES" document.

1) Components and Streams are now only used internally by the DA, they are not passed to the DP. All data is identified only by the Seed Location and Seed name.

2) There are 4 new spare DA to DP record types defined, as well as 10 new defined record types. ULTRA_PKT is used for sending the "ultra_type" data structure, using the "commo_reply" packet type. DET_AVAIL is used for sending the "det_avail_type" data structure, using the "commo_reply" packet type. CMD_ECHO is used to send a copy of the command sent from the DP to the DA, which is "DP_to_DA_msg_type" using the "commo_reply" packet type. LINK_PKT is used to send link information using the "commo_link" packet type. UPMAP is used to send the "commo_upmap" structure used for file uploading. DOWNLOAD is used for sending the "download_struc" data structure, using the "commo_reply" packet type. CALIBRATION is used to send calibration reports using the "calibration_report_struc" packet type.FLUSH is for GSN/GTSN use. SEED_DATA is for internal DA use. RECORD_HEADER3 is used to send Steim 3 compressed data records.

```
record_type =  (EMPTY, RECORD_HEADER_1, RECORD_HEADER_2,
                CLOCK_CORRECTION, COMMENTS, DETECTION_RESULT,
                RECORD_HEADER3, REC_SPARE4, REC_SPARE3,
                END_OF_DETECTION, CALIBRATION, FLUSH, LINK_PKT,
                SEED_DATA, REC_SPARE2, REC_SPARE1, ULTRA_PKT,
                DET_AVAIL, CMD_ECHO, UPMAP, DOWNLOAD) ;
```

3) New DP to DA commands have been added :

```
type
  DP_TO_DA_CMD_TYPE = byte ;
const {numerical equivalents of DP_TO_DA_CMD_TYPE}
  ACK_MSG = 48 ;
  SHELL_CMD = 50 ;
  NO_CMD = 51 ;
  START_CAL = 54 ;
  STOP_CAL = 55 ;
  AUTO_DAC_CORRECTION = 61 ;
  ACCURATE_DAC_CORRECTION = 68 ;
  COMM_EVENT = 70 ;
  LINK_REQ = 71 ;
  LINK_ADJ = 72 ;
  DOWN_ABT = 73 ;
  ULTRA_REQ = 77 ;
  ULTRA_MASS = 78 ;
  ULTRA_CAL = 79 ;
  ULTRA_STOP = 80 ;
  DET_REQ = 81 ;
  DET_ENABLE = 82 ;
  DET_CHANGE = 83 ;
  REC_ENABLE = 84 ;
  DOWN_REQ = 85 ;
  UPLOAD = 86 ;
```

4) The "microsec" field has been added for more precision. The total offset is millisec / 1000 + microsec / 1000000. "header_revision" was added quite a while ago, the current value is 4. Detection_day and detection_seq are used for NSN VSAT communications, ignored by everyone else. Clock_drift is the amount of clock drift in microseconds and has a range of plus and minus 30000000 (30 seconds). Seed names, the Seed Network ID and Seed Location ID are now put into compressed records. "clkq" is the clock quality (-1 to 5). "frame_count" is the number data frames in the record. Archive is for internal DA use. "nsnchid" is only used for NSN VSAT communications and is a number assigned by them and specified in the configuration file.

```
seed_name_type = array [1..3] of char ;
seed_net_type = array [1..2] of char ;
location_type = array [1..2] of char ;

header_type = record           { first full frame in each record }
  header_flag : longinteger ;  { flag word for header frame }
  frame_type : record_type ;   { standard frame offset for frame type }
  ht_sp1 : byte ;              { Component in SHEAR system }
  ht_sp2 : byte ;              { Stream in SHEAR system }
  soh : byte ;                 { state-of-health bits }
  station : longinteger ;      { 4 byte station name }
  millisec : integer ;         { time mark millisec binary value }
  time_mark : integer ;        { sample number of time tag, usually 1 }
  samp_1 : longinteger ;       { 32-bit first sample in record }
  clock_corr : integer ;       { last clock correction offset in milliseconds }
  number_of_samples : word ;   { number of samples in record }
  rate : shortint ;            { samp rate: + = samp/sec; - = sec/samp }
  blockette_count : byte ;     { Always zero on DA/DP link }
  time_of_sample : time_array { time of the tagged sample }
  packet_seq : longinteger ;
  header_revision : byte ;     { 5 for Ultra Shear beta 2 packets }
  detection_day : byte ;
  detection_seq : integer ;
  clock_drift : longinteger ; { drift in microseconds }
  seedname : seed_name_type ;
  clkq : byte ;                { -1 to 5 }
  seednet : seed_net_type ;
  location : location_type ;
  ht_sp3 : integer ;           { 0 }
  microsec : integer ;         { time mark microsec binary value }
  frame_count : byte ;         { number of 64 byte data frames }
  nsnchid : byte ;             { channel id 1-255 for nsn vsat }
  z : array [1..5] of byte ;   { 0's }
  archive : boolean
end ;                          { total 64 bytes }
```

5) The fields that were previously in compressed records were moved to the header type. Their location has not changed.

```
compressed_record = record    { image of physical record }
  case integer of
    1 : ( w : array [0..((TOTAL_FRAMES_PER_RECORD*WORDS_PER_FRAME)-1)] of lword ) ;
    2 : ( h : header_type ;
          f : compressed_frame_array ) ;
  end ;
```

6) The "commo_reply" packet is used to send variable length data structures from the DA to DP, such as ultra_type and det_avail_type structures. The data structure is sent as a series of segments, between 1 and 348 may be required, allowing for a data structure of up to 65000 bytes in length even when using MM256 format. When using Q512 or QSL format, 65000 bytes will fit into 147 segments. The normal procedure is to wait until you receive any segment of the data structure, allocate storage based on "total_bytes", and after each segment is received, check to make sure you have received all segments between 1 and "total_seg". "header_flag" is there for compatibility with other "commo_record" structures and can be ignored. "frame_type" is currently either ULTRA_PKT or DET_AVAIL. "first_seg" is true if this is segment number one. "total_bytes" is the byte count of the structure being sent, you can use this value to dynamically allocate the storage for the data structure on the DP side. "total_seg" tells you how many segments will be required for transmission. "this_seg" tells you which segment this is. "byte_offset" is where the data from this segment should be stored, relative to the start of the data structure. "byte_count" is how many bytes are transferred in this segment. "bytes" contains the actual data from index 1, through index "byte_count".

```
const
  MAX_REPLY_BYTE_COUNT = 444 ;
type
  commo_reply = record
                  header_flag : longinteger ;
                  frame_type : record_type ;
                  first_seg : boolean ;
                  total_bytes : integer ;
                  total_seg : integer ;
                  this_seg : integer ;
                  byte_offset : integer ;
                  byte_count : integer ;
                  bytes : array[1 .. MAX_REPLY_BYTE_COUNT] of byte
                end ;
```

7) Commo_link is used to send data about the link from the DA to DP as a response to the LINK_REQ command. "header_flag" is there for compatibility with other "commo_record" structures and can be ignored. "frame_type" will be LINK_PKT. "rcecho" is true if the DA will echo DP to DA commands (using CMD_ECHO record type). The DA will never echo the ULTRA_REQ, DET_REQ, or LINK_REQ commands, since they always have their own responses. "seq_modulus" is the number of unique DA to DP sequence numbers available. It is normally 256, so sequence numbers go from 0 to 255. Other values not exceeding 256 are also possible based on the programming of the DA. "window_size" is the maximum number of packets in a transmission window. "total_prio" is the total number of priority levels on this comlink. Remember that the highest priority level is reserved and is not available for user data (such as continuous, event, messages, etc.). "msg_prio" is the priority for messages. "det_prio" is the priority for detection records. "time_prio" is the priority for time records. "cal_prio" is the priority for calibration records. "resendtime" is how many seconds can elapse between sending a packet and getting an acknowledgement packet before dacommo enters resend mode. "synctime" is the sync packet interval. "resendpkts" is how many packets are sent out a time when resending packets. "netdelay" is the number of seconds that dacommo waits in between closing the last network connection and opening a new one. "nettime" is the network connection timeout. "netmax" is how many packets that can be resent on a network connection without getting an acknowledge when using a TCP/IP connection. "groupsize" is the size of packet transmission grouping. "grouptime" is the timeout used to force transmission of packets when less than a full group are available.

```
linkfmt = (QSL, Q512, MM256) ;
commo_link =
  record
    header_flag : longinteger ;
    frame_type : record_type ;
    rcecho : boolean ;
    seq_modulus : integer ;
    window_size : integer ;
    total_prio : byte ;
    msg_prio : byte ;
    det_prio : byte ;
    time_prio : byte ;
    cal_prio : byte ;
    link_format : linkfmt ;
    resendtime : integer ;
    synctime : integer ;
    resendpkts : integer ;
    netdelay : integer ;
    nettime : integer ;
    netmax : integer ;
    groupsize : integer ;
    grouptime : integer ;
  end ;
```

8) The length of a comment string has been reduced to 132 characters for internal consistancy :

```
COMMENT_STRING_LENGTH = 132 ;
comment_format  = (DYNAMIC_LENGTH_STRING) ; {others reserved}
comment_string_type = string[COMMENT_STRING_LENGTH] ;
```

9) The commo comment structure has been changed. cc_pad is always zero. cc_station is the sending station and cc_net is the network the station belongs to. When this record is sent using QSL format, the station and network are moved to immediately following the last byte of the comment string, and must be moved back into correct position by the DP before processing. This increases throughput over the link, since short strings do not take up 132 bytes.

```
commo_comment = record
                  header_flag : longint ;
                  frame_type : record_type ;
                  comment_type : comment_format ;
                  time_of_transmission : time_array ;
                  ct : comment_string_type ;
                  cc_pad : byte ;
                  cc_station : longint ;
                  cc_net : seed_net_type
                end ;
```

10) Two new variant records have been added to send a "commo_reply" packet :

```
commo_record =
  record
    case integer of
      1 : (cr : compressed_record) ;
      2 : (ce : commo_event) ;
      3 : (cc : commo_comment) ;
      4 : (cy : commo_reply) ;
      5 : (cl : commo_link)
      6 : (cu : commo_upmap)
  end ;
```

11) Calibrators can support at most 4 wave form types, sine, step, red-noise (rand), and white-noise (wrand) :

```
waveform_type = (sine, step, rand, wrand) ;
```

12) Sine wave calibrations can be of the following frequencies (excluding Hz_DC). Hz25_0000 is 25.000Hz, Hz0_0005 is 0.0005Hz.

```
cal_freqs = (Hz_DC,    Hz25_000, Hz10_000, Hz8_0000, Hz6_6667,
             Hz5_0000, Hz4_0000, Hz3_3333, Hz2_5000, Hz2_0000,
             Hz1_6667, Hz1_2500, Hz1_0000, Hz0_8000, Hz0_6667,
             Hz0_5000, Hz0_2000, Hz0_1000, Hz0_0667, Hz0_0500,
             Hz0_0400, Hz0_0333, Hz0_0250, Hz0_0200, Hz0_0167,
             Hz0_0125, Hz0_0100, Hz0_0050, Hz0_0020, Hz0_0010,
             Hz0_0005) ;
```

13) Digitizer calibrators are split into "board equivalents", for example, a single Q380 QDP board. To calibrate multiple "boards" you must send multiple calibration commands using the "ULTRA_CAL" command. To determine the capabilities of each "board", the DP should request the information using the "ULTRA_REQ" command, which will send the "ultra_type". "calcmd" must a valid wave form from the above list. "sfrq" must be a valid frequency from the above list, ignored for non-sine calibrations. "plus" is true for a positive step, false for negative step, ignored for non-step calibrations. "capacitor" is true for capacitive calibration coupling, false for resistive coupling. "autoflag" is simply passed into the digitizer (added per GTSN request). "ext_sp1" should be set to zero. "calnum" is the calibrator "board" number. "duration" is the number of seconds the wave form should be generated, a value of zero indicates an infinite duration (or the maximum available length). "amp" is the amplitude in decibels, where maximum amplitude is zero dB, lower amplitude by a negative number. "rmult" is the random noise (red or white) frequency multiplier on the supercal which has a range of 1 to 16, ignored for non-noise calibrations. "map" is a bit map local to this board, the first channel on the board is bit zero. "settle" is the relay settling delay in seconds, supported by qapcal and supercal boards. "filt" is the low pass filter to use on a supercal, between 1 and 4. "ext_sp2" should be set to zero.

```
extcal_type =
  record
    calcmd : waveform_type ;
    sfrq : cal_freqs ; {sine frequency}
    plus : boolean ; {positive step}
    capacitor : boolean ; {resistive = false}
    autoflag : byte ; {non zero for automatic calibration}
    ext_sp1 : byte ; {spare}
    calnum : integer ; {calibrator number}
    duration : longinteger ; {in seconds, 0=infinate}
    amp : integer ; {amplitude, in DB}
    rmult : integer ; {random multiplier}
    map : integer ; {channel map local to this board}
    settle : integer ; {relay settling time, in seconds}
    filt : integer ; {filter to use 1..MAXCAL_FILT}
    ext_sp2 : integer ;
  end ;
```

14) This data structure is not really new, but is now used over the DP↔DA link. This data structure is used internally by the system, and by "det_change_type" and "det_descr". xth1, xth2, xth3, xthx, def_tc, and val_avg are not used for a threshold detector and are zeroes. For other detectors, the use of each of these fields will vary.

```
shortdetload =
  record
    filhi, fillo : longinteger ;
    iwin : longinteger ;
    n_hits : longinteger ;
    xth1, xth2, xth3, xthx : longinteger ;
    def_tc : longinteger ;
    wait_blk : longinteger ;
    val_avg : integer
  end ;
```

15) By sending the "DET_ENABLE" command to the DA, up to 20 detectors can be turned on or off in one command. "count" is how many detectors are to be changed. The detector is not re-initialized as a result of receiving this command, it is simply now either called or not called. "id" is the detector identification number, which is provided by the "det_descr" data structure. "enab" is true to enable the detector, false to disable the detector. "det_sp" should be set to zero.

```
det_enable_type =
  record
    count : integer ; {how many}
    changes : array [1 .. 20] of
      record
        id : integer ; {detector id}
        enab : boolean ;
        det_sp : byte
      end
  end ;
```

16) If the actual operating parameters of a detector need to be changed, then the "DET_CHANGE" command must be used. Only one detector can be changed per command. The detector is completely re-initialized when this command is received. "id" is the detector as provided in the "det_descr" structure. "enab" is true to enable the detector, false to disable it. "dct_sp" should be set to zero. "ucon" are the new detector parameters.

```
det_change_type =
  record
    id : integer ; {detector id}
    enab : boolean ;
    dct_sp : byte ;
    ucon : shortdetload
  end ;
```

17) Recording or transmission of data on up to 8 LCQs can be changed with the "REC_ENABLE" command. "count" is how many LCQs to change with this command. "mask" is the new enable mask, and has the same format as the "enabled" field in the "used_combo_type" definition described later. "c_prio" and "e_prio" are the comlink priorities. "ret_sp1" should be set to zero. "seedname" is the seedname (this field would not normally be changed from what was passed in the "used_combo_type"). "seedloc" has the SEED location and should not need to be changed from what was passed in the "used_combo_type".

```
rec_enable_type =
  record
    count : integer ;
    changes : array [1 .. 8] of
      record
        seedname : seed_name_type ;
        mask : byte ;
        seedloc : location_type ;
        c_prio : byte ;
        e_prio : byte ;
        ret_sp1 : integer ;
      end
  end ;
```

18) This data structure is used by the "AUTO_DAC_CORRECTION" and "ACCURATE_DAC_CORRECTION" commands. For "AUTO_DAC_CORRECTION" "param1" is 1 to enable automatic correction, 0 to disable correction. For "ACCURATE_DAC_CORRECTION" "param2" is the upper 4 bits of the correction value, and "param3" is the lower 8 bits of the correction value.

```
stokely =
  record
    param0 : byte ;
    dummy2 : byte ;
    param1 : integer ;
    param2 : integer ;
    param3 : integer ;
    param4 : integer ;
    param5 : integer ;
    param6 : integer ;
    param7 : integer ;
  end ;
```

19) This new packet allows the DP to change link parameters on the DA side. "window_size" sets the maximum number of packets in a transmission window. "set_msg" sets the priority for messages. "set_det" sets the priority for detection records. "set_time" sets the priority for time records. "set_cal" sets the priority for calibration records. "resendtime" sets how many seconds can elapse between sending a packet and getting an acknowledgement packet before dacommo enters resend mode. "synctime" sets the sync packet interval. "resendpkts" sets how many packets are sent out a time when resending packets. "netdelay" sets the number of seconds that dacommo waits in between closing the last network connection and opening a new one. "nettime" sets the network connection timeout. "netmax" sets how many packets that can be resent on a network connection without getting an acknowledge when using a TCP/IP connection. "groupsize" sets the size of packet transmission grouping. "grouptime" sets the timeout used to force transmission of packets when less than a full group are available. "lasp1" and "lasp2" should be set to zero.

```
link_adj_msg =
  record
    window_size : integer ;
    set_msg : byte ;
    set_det : byte ;
    set_time : byte ;
    set_calp : byte ;
    resendtime : integer ;
    synctime : integer ;
    resendpkts : integer ;
    netdelay : integer ;
    nettime : integer ;
    netmax : integer ;
    groupsize : integer ;
    grouptime : integer ;
    lasp1 : integer ;
    lasp2 : integer ;
  end ;
```

20) The new commands are shown below. Your definition may look a bit different, but should be equivalent. "dp_seq" should be an incrementing number (modulus 256) that changes each time a new command is sent. This is useful if command echo is on to determine which command is being echoed, and is also used in the DA. The DA will not put duplicate command echoes into the transmit queue, a duplicate is defined as having the same command type and the same sequence. "rc_sp1" through "rc_sp6" should be set to zero. For the "COMM_EVENT" command, "mask" is a 32 bit mask for the 32 available remote detection flags. For the "ULTRA_MASS" command, "mbrd" is the calibrator "board" for the mass recentering, and "mdur" is the length of the pulse in milliseconds. For the "ULTRA_STOP" command, "sbrd" is the calibrator "board" to send the stop command to. For the "DET_REQ" command, "dr_name" is the SEED name and "dr_loc" is the SEED location for the LCQ you want to obtain detectors for. For the SHELL_CMD, sc is the command line you want to execute. The DA will redirect the input and stderr paths to /nil, and redirect the output path to /pipe/log.*name* where *name* is the command name. For the dacommstat command, using "@" as the parameter will instruct RCMAN to substitute the name of your comlink data module. LINK_REQ has no parameters. DOWN_REQ is a request to the DA to transfer the contents of the indicated file to you using DOWNLOAD packets. The "download_struc" is described later. UPLOAD is used to upload a file from the DP to the DA, see the "upload_struc" described later.

```
DP_to_DA_msg_type = record  {Different message and command types are send from DP to DA}
  case boolean of
    false : (x: array [0..(DP_TO_DA_MESSAGE_LENGTH-1)] of byte) ;
    true :
      (cmd_type : DP_to_DA_cmd_type ;
       dp_seq : byte ;
       case DP_to_DA_cmd_type of
         AUTO_DAC_CORRECTION, ACCURATE_DAC_CORRECTION : (cmd_parms : stokely) ;
         COMM_EVENT : (rc_sp1 : integer ;
                       mask : longinteger) ;
         ULTRA_MASS : (mbrd : integer ;
                       mdur : integer) ;
         ULTRA_CAL : (rc_sp2 : integer ;
                      xc : extcal_type) ;
         ULTRA_STOP : (sbrd : integer) ;
         DET_REQ : (dr_name : seed_name_type ;
                    rc_sp3 : byte ;
                    dr_loc : location_type) ;
         DET_ENABLE : (rc_sp4 : integer ;
                       de : det_enable_type) ;
         DET_CHANGE : (rc_sp5 : integer ;
                       dc : det_change_type) ;
         REC_ENABLE : (rc_sp6 : integer ;
                       re : rec_enable_type) ;
         SHELL_CMD : (sc : string[80]) ;
```

```
        LINK_ADJ : (la : link_adj_msg)) ;
        DOWN_REQ : (fname : string[63]) ;
        UPLOAD : (ups : upload_struc)) ;
  end;
```

21) This sub-structure (of det_avail_type) defines a single detector. "enabled" is true if the detector is currently running. "remote" indicates that the detector is running on a digitizer, and is probably not important to the DP. "dettype" is either MURDOCK_HUTT (0), THRESHOLD (1), or another detector. "dd_sp1" is currently zero. "cons" are the current detector parameters. "id" is a DA assigned number, and each detector has a unique number, this number is used to identify the detector when sending a DET_ENABLE or DET_CHANGE command. "name" is the name of the detector, such as 'SPWW'. PARAMS[0] is the detector type name, such as "MURDOCK HUTT". PARAMS[1] through PARAMS[11] are the names of each of the fields in "shortdetload".

```
det_descr =
  record
    enabled : boolean ; {detector running}
    remote : boolean ; {is a remote detector}
    dettype : event_detector_type ;
    dd_sp1 : byte ;
    cons : shortdetload ;
    id : integer ;
    name : string[23] ;
    params : array[0..11] of string[15]
  end ;
```

22) This structure is returned as "DET_AVAIL" using the "commo_reply" packet as a result of a "DET_REQ" command. "count" is the number of detectors that will be described. "dat_sp1" is currently zero. "dets" are the detectors using the data structure above.

```
det_avail_type =
  record
    count : integer ; {number of detectors described}
    dat_sp1 : integer ;
    dets : array [1 .. 20] of det_descr
  end ;
```

23) These constants define the maximum number of calibrator "boards", the maximum number of filters, and the maximum number of remote detection flags (comm events).

```
MAXCAL = 8 ; {maximum number of calibrators}
MAXCAL_FILT = 4 ; {maximum number of calibrator filters}
CE_MAX = 32 ; {maximum number of comm events}
COMMLENGTH = 11 ; {number of characters in a comm event name}
```

24) In "ultra_type", all 32 comm names are passed. Various structures use OmegaSoft Pascal "sets" internal to the DA, to avoid compatibility problems, they are represented instead by a 4 byte, bit map, passed as a long integer.

```
stringcme = string[COMMLENGTH] ;
comm_name_type = array[0 .. CE_MAX - 1] of stringcme ;
chan_map = longinteger ;
psuedo_set = longinteger ;
bit_set = set of 0 .. 31 ;
```

25) This structure defines the capabilities of each calibrator. "number" is the number of calibrators available. "mass_ok" is true if any of the calibrators support mass recentering. "ct_sp1" is currently zero. "coupling_option" is true if capacitive/resistive switching is available. "polarity_option" is true if step wave forms can have both positive and negative steps. "board" is the calibrator board that you would pass as "calnum" in an ULTRA_CAL command. "min_settle", "max_settle", and "inc_settle" define the relay settling capabilities, with "max_settle" being zero if this capability is not supported. While they are all defined in seconds, if "inc_settle" is 60, this indicates that only multiples of 60 seconds are supported, and the user should be prompted to enter the time in minutes instead of seconds. This same consideration applies to the "inc_dur" fields later. "min_mass_dur", "max_mass_dur", "inc_mass_dur" , and "def_mass_dur" define the acceptable range for mass duration in milliseconds, with "max_mass_dur" equal to zero if not supported. "def_mass_dur" is a default value to use and an aid in prompting the user for input. "min_filter" and "max_filter" define how many filters are available to use, with "max_filter" equal to zero if filters are not supported. "min_amp", "max_amp", and "amp_step" are the calibration amplitude ranges available and the minimum step between them, all are in decibels. "monitor" defines which physical digitizer channel (1-N) records the output of the calibrator, zero if none. "rand_min_period" and "rand_max_period" define the frequency multiplier range for the random noise wave forms, "rand_max_period" being zero if this feature is not supported. "default_step_filt" is the default filter to use for the step wave form, zero if filters are not supported. "default_rand_filt" is the default filter to use for the random noise wave forms, zero if filters are not supported. "ct_sp2" is currently zero. The "durations" records define the duration range for each wave form using "min_dur", "max_dur", and "inc_dur". "map" is a bit map describing which physical channels are calibrated using this board where channel 1 is bit zero, channel 2 is bit one, etc. "waveforms" is a bit set indicating which waveforms are supported on this calibrator, where sine is bit 0, step is bit 1, etc. "sine_freqs" is a bit set indicating which sine wave frequencies are supported, where Hz_DC is bit 0, Hz25_000 is bit 1, etc. "default_sine_filt" are the frequencies (same format as sine_freqs) that go with each filter, zeroes if filters are not supported. "name" is the name of the calibrator, such as 'QAPCAL'. "filtf" is a string that can be used to prompt the user for the filter to use, for example, a SUPERCAL uses '1=40Hz 2=10Hz 3=1Hz 4=0.1Hz'.

```
cal_type =
  record
    number : integer ; {number of calibrators}
    mass_ok : boolean ;
    ct_sp1 : byte ;
    cals : array[1..MAXCAL] of
      record
        coupling_option : boolean ; {supports capacitor/resistor coupling}
        polarity_option : boolean ; {supports plus and minus step}
        board : integer ; {which board this goes with}
        min_settle : integer ; {minimum settling time in seconds}
        max_settle : integer ; {maximum settling time in seconds}
        inc_settle : integer ; {supports settling time in inc seconds}
        min_mass_dur : integer ; {minimum duration in ms}
        max_mass_dur : integer ; {maximum duration in ms, 0=none}
        inc_mass_dur : integer ; {duration increment}
        def_mass_dur : integer ; {default duration in ms}
        min_filter : integer ; {minimum filter number}
        max_filter : integer ; {maximum filter number}
        min_amp : integer ; {minimum amplitude in db}
        max_amp : integer ; {maximum amplitude in db}
        amp_step : integer ; {amplitude step in db}
        monitor : integer ; {channel that monitors the calibrator}
        rand_min_period : integer ; {minimum random period}
        rand_max_period : integer ; {maximum random period}
        default_step_filt : integer ; {default step filter}
        default_rand_filt : integer ; {default rand filter}
        ct_sp2 : integer ;
        durations : array [waveform_type] of
          record
            min_dur : longinteger ; {minimum duration}
            max_dur : longinteger ; {maximum duration}
            inc_dur : longinteger ; {duration increment}
          end ;
        map : chan_map ; {channels this calibrator calibrates}
        waveforms : psuedo_set ; {supported waveforms}
        sine_freqs : psuedo_set ;
        default_sine_filt : array [1 .. MAXCAL_FILT] of psuedo_set ;
        name : string[23] ;
        filtf : string[59] ; {filter description string}
      end
  end ;
```

26) This data structure provides information that is intended mostly for the DA side, but some fields are useful to a DP side and are in italics. *For example, "set_osc_ok" can be used to know whether sending an "ACCURATE_DAC_CORRECTON" command will actually do anything. "name" is the name of the digitizer, such as 'Q4120'. "version" is the software release version of the server for the digitizer.* "clockmsg" is a prompt to a user regarding sending a command to a clock (normally a GPS clock). "prefilter_ok" indicates that the digitizer can provide logical channels which are pre-filtered data for detectors. "detector_load_ok" indicates that the digitizer can run event detectors in it's processor. "setmap_ok" indicates that the active digitizer polling map can be changed. "clockstring_ok" indicates that the server can provide a clock string to a requester. "int_ext_ok" indicates whether there are internal and external marks, or if false, that external time marks are always used. "send_message_ok" is true if messages can be sent to a digitizer. "message_chan_ok" is true if messages can be directed to an individual channel of a digitizer. *"set_osc_ok" is true if the digitizer has an oscillator that can be changed in frequency by a server command.* "set_clock_ok" is true if the clock can be set to a new time. "wait_for_data" indicates that successful startup of a digitizer should not be assumed until that second of data has been received. "dt_sp1" and "dt_sp2" are currently zero.

```
digi_type =
    record
      name : string[23] ; {digitizer name}
      version : string[23] ; {program version number}
      clockmsg : string[79] ; {available clock messages}
      prefilter_ok : boolean ; {server supports prefiltering}
      detector_load_ok : boolean ; {server supports loadable detectors}
      setmap_ok : boolean ; {server will process a set map command}
      clockstring_ok : boolean ; {server will return clockstring}
      int_ext_ok : boolean ; {two mark sources}
      send_message_ok : boolean ; {can send digitizer messages}
      message_chan_ok : boolean ; {messages can be sent to channels}
      set_osc_ok : boolean ; {can set the master oscillator}
      set_clock_ok : boolean ; {can set the clock}
      wait_for_data : byte ; {number of seconds to wait for data}
      dt_sp1 : byte ;
      dt_sp2 : byte ;
    end ;
```

27) This data structure defines recording status for each LCQ defined. "physical" indicates which physical digitizer channel (1-N) that the data derived from, zero for aux. channels. This field is used to correlate between calibrator channel maps and SEED names. "available" is the bit map (bits 0 - 5) indicating which recording status can be enabled. "enabled" is the bit map indicating which recording status is currently enabled. "det_count" is the number of detectors available (but not necessarily running) on the LCQ. If this field is non-zero, you can use the "DET_REQ" command to obtain information about the detectors. "used_sp1" is currently zero. "c_prio" and "e_prio" are the comlink priorities. "seedloc" is the seed location and "seedname" is the seed name. "rate" is the recording frequency, positive is samples per second, negative is seconds per sample.

```
{
  bit 0 = continous on this comm link
  bit 1 = event on this comm link
  bit 2 = continous on tape
  bit 3 = event on tape
  bit 4 = continous on disk
  bit 5 = event on disk
}
    used_combo_type =
      record
        seedname : seed_name_type ;
        used_sp1 : byte ;
        seedloc : location_type ;
        physical : byte ; {physical channel number, 0 if not QSRVD}
        available : byte ; {can be enabled}
        enabled : byte ; {currently enabled}
        det_count : byte ; {number of detectors}
        c_prio : byte ; {comlink continuous priority}
        e_prio : byte ; {comlink event priority}
        rate : integer ;
      end ;
```

28) This structure is returned as a "ULTRA_PKT" using the "commo_reply" packet as the result of a "ULTRA_REQ" command. "digi" is the "digi_type" described earlier. "vcovalue" is the current VCO value (0 - 4095) and can be used as a starting point for prompting for the "ACCURATE_DAC_CORRECTION" command. "pllon" is true if the VCO is being controlled by the Phase-Lock-Loop, if on, you should not allow the user to send an "ACCURATE_DAC_CORRECTION" command. "umass_ok" is just a copy of the "mass_ok" field in the "cal_type" structure. "comcount" will always be 32. "comoffset" is a byte offset to the start of strings. "usedcount" is the total number of LCQs defined. "usedoffset" is a byte offset from the start of the ultra_type structure where the "used_combo_type" structures start, and will be a multiple of 4. Each "used_combo_type" follows immediately after the previous one. "calcount" is a copy of the "number" field in "cal_type". "caloffset" is the byte offset from the start of the ultra_type structure where the "cal_type" structure starts, and will be a multiple of 4. "pr_levels" is the number of priority levels on this comlink. "ultra_rev" has the revision number to allow for changes in the protocol in the future, the current revision is 1. "comm_mask" contains the bitmask of comm-detectors at this time, 1 being on and 0 being off.

```
ultra_type =
  record
    digi : digi_type ;
    vcovalue : integer ; {current vco value}
    pllon : boolean ; {true if PLL controlling PLL}
    umass_ok : boolean ; {copy of cal.mass_ok}
    comcount : integer ; {number of comm events}
    comoffset : integer ; {offset to start of comm names}
    usedcount : integer ; {number of used combos}
    usedoffset : integer ; {offset to start of used combinations}
    calcount : integer ; {copy of cal.number}
    caloffset : integer ; {offset to start of calibrator defs}
    pr_levels : byte ; {number of comlink priority levels}
    ultra_rev : byte ; {Ultra Shear protocol revision number, currently 1 }
    ut_sp1 : byte ;
    comm_mask : longinteger ; {Current commlink detector bitmask}
  end ;
```

29) An addition has been made to sequence_control_type. SEQUENCE_SPECIAL is used on the LINK_PKT only and signifies that it is not part of the normal transmission queue. It's sequence number is to be used as the starting sequence number for normal packets to receive and acknowledge.

```
sequence_control_type = (SEQUENCE_INCREMENT, SEQUENCE_RESET, SEQUENCE_SPECIAL) ;
```

30) During a Threshold detection, "motion_pick_lookback_quality" and "period_x_100" are zero. "peak_amplitude" is the maximum value above the high or low threshold. "background_amplitude" is used as the threshold value that was crossed. The user defined component name and the detector name have also been added. The seed name of the channel (and location) along with the station and network are also added to completely identify the source of the detection.

```
squeezed_event_detection_report = record    { total 58 bytes }
  jdate : longint ;               { seconds since Jan 1, 1984 00:00:00 }
  millisec : integer ;            { 0..999 fractional part of time }
  sedr_sp2 : byte ;               { component in shear systems }
  sedr_sp3 : byte ;               { stream in shear systems }
  motion_pick_lookback_quality : longint ; { "string" of this gives, eg "CA200999" }
  peak_amplitude : longint ;      { & threshold sample value }
  period_x_100 : longint ;        { 100 times detected period }
  background_amplitude : longint ;{ & threshold limit exceeded }
  detname : string[23]            { detector name }
  seedname : seed_name_type ;     { seed name }
  nsneodid : byte ;               { nsn end of detection channel id }
  location : location_type ;      { seed location }
  ev_station : longint ;          { station name }
  ev_network : seed_net_type ;    { seed network }
end ;
```

31) This new structure is used to send calibration reports.

```
calibration_report_type = (SINE_CAL, RANDOM_CAL, STEP_CAL, ABORT_CAL) ;

calibration_report_struc =              { total 36 bytes }
  record
    cr_duration : longint ;             { duration in seconds }
    cr_period : longint ;               { period in milliseconds }
    cr_amplitude : integer ;            { amplitude in DB }
    cr_time : time_array ;              { time of signal on or abort }
    cr_type : calibration_report_type ;
    cr_sp1 : byte ;                     { component in shear systems }
    cr_sp2 : byte ;                     { stream in shear systems }
    cr_sp3 : byte ;                     { input component in shear systems }
    cr_sp4 : byte ;                     { input stream in shear systems }
    cr_stepnum : byte ;                 { number of steps? }
    cr_flags : byte ;                   { bit 0 = plus, bit 2 = automatic, bit 4 = p-p }
    cr_flags2 : byte ;                  { bit 0 = cap, bit 1 = white noise }
    cr_0dB : single ;                   { 0 = dB, <> 0 = value for 0dB }
    cr_seedname : seed_name_type ;      { seed name }
    cr_sfrq : cal_freqs ;               { calibration frequency if sine }
    cr_location : location_type ;       { seed location }
    cr_input_seedname : seed_name_type;{ seed name of cal input }
    cr_filt : byte ;                    { filter number }
    cr_input_location : location_type ;{ location of cal input }
    cr_station : longint ;              { station name }
    cr_network : seed_net_type ;        { network }
  end ;
```

32) This new structure is used to provide information about the clock. Currently defined clock models are :

0 = unknown clock.	1 = OS9 internal clock.	2 = Kinemetrics Goes	3 = Kinemetrics Omega
4 = Kinemetrics DCF	5 = Meinburg DCF	6 = Quanterra GPS1/QTS	7 = Quanterra GPS1/Goes
8 = Quanterra GPS1/UA31S	9 = Quanterra GPS2/QTS2	10 = Quanterra GPS2/Goes	11 = Quanterra GPS2/UA31S

The sequence number increments each time their is a change in the structure, it folds back to 1 after 255. For the Kinemetrics Omega clock the first status byte indicates the station being received :

"A" = Norway	"B" = Liberia	"C" = Hawaii	"D" = North Dakota
"E" = La Reunion	"F" = Argentina	"G" = Australia	"H" = Japan

Any other status is undefined. For the Quanterra GPS1/QTS and GPS2/QTS2 clocks each byte in the array represents the Signal to Noise ration (in DB) of a satellite, with a 0 meaning no satellite being tracked on that channel.

```
cl_spec_type = record
  model : byte ;                        { model type from above }
  cl_seq : byte ;                       { 1-255 sequence number }
  cl_status : array [0..5] of byte ;    { clock specific status }
end ;
```

33) "usec_correction" has been added to further define when the last timemark occurred. The actual offset is correction_quality.msec_correction * 1E-3 + usec_correction. "clock_drift" has been added and is the amount of clock drift in microseconds. Vco contains the full 12 bit VCO control value (on the most significant 8 bits are in the time_quality_descriptor). Cl_spec has the clock specific information described above. The station and network have also been added. (they are also in squeezed_event_detection_report and calibration_report_struc).

```
eventlog_struc = record
  case frame_type : record_type of     { standard offset for record type }
    EMPTY : ;                          { sync packets }
    CLOCK_CORRECTION :
      ( clock_exception : clock_correction_type ; { type of exception condition }
        time_of_mark : time_array ;    { if clock_exception = EXPECTED, VALID, DAILY, or
                                         UNEXPECTED }
        count_of : record
                   case clock_correction_type of
                     MISSING_TIMEMARK, UNEXPECTED_TIMEMARK :
                       ( seconds_elapsed_since_last_timemark : longint ) ;
                     EXPECTED_TIMEMARK, VALID_TIMEMARK,
                     DAILY_TIME_CORRECTION_REPORT :
                       ( consecutive_expected_timemarks : longint )
                   end ;
        correction_quality : time_quality_descriptor ;
        usec_correction : integer ; { timemark offset, 0 - 999 microseconds}
        clock_drift : longint ;   { clock drift in microseconds }
        vco : integer ;           { full VCO value }
        cl_spec : cl_spec_type ;  { clock specific information }
        cl_station : longint ;    { station name }
        cl_net : seed_net_type) ; { network }
    DETECTION_RESULT :
      ( detection_type : event_detector_type ;
        time_of_report : time_array ;  { time that detection is reported }
        pick : squeezed_event_detection_report ) ;
    CALIBRATION :
      ( word_align : byte ;
        cal_report : calibration_report_struc ) ;
  end ;                                { total 66 bytes }
```

34) When the DP wants to upload a file to the DA, then it sends "upload_struc" packets to the DA. When creating a new file, or after every 10 packets, the DP needs to set the "return_map" flag and wait for the "commo_upmap" packet to determine the status of the upload at the DA end. The first step in a download is to send an "upload_struc" with upload_control = CREATE_UPLOAD, setting in the file_size and file_name and the "return_map" set. The DP should then wait for the commo_up packet. If upload_ok is FALSE, then the DP needs to abort the download, probably due to an invalid file name. At this point segmap should be all zeroes. For each block of 90 bytes to be uploaded, send the "upload_struc" with upload_control set to SEND_UPLOAD. The byte offset is the offset from the start of the file where these "byte_count" bytes start. "seg_num" is a number from 1 to N of this segment. This segment number is set into the "segmap" on the DA side (bit 0 of byte 0 of segmap is segment 1). Every 10 packets, set the "return_map" flag true and then wait for the commo_upmap structure. This makes sure that the DA is keeping up with the data flow. You should have some sort of timeout on the waiting, 30 seconds would probably be a good value. If a timeout occurs without receiving the commo_upmap, send a MAP_ONLY packet with the "return_map" flag set to request the map again. On the last segment, set the "return_map" true, you can then analyze "segmap" to make sure the DA received all the packets. If there are any missing, then resend those packets and repeat checking the map until all packets have been received by the DA. You then send a "UPLOAD_DONE" packet, this tells the DA to write the contents of the buffer into the file. If for any reason, the user wants to abort the upload, then send a "ABORT_UPLOAD" packet, this tells the DA to free the buffer it was using. If you are sending a text file, you must first make sure it is formatted correctly for the DA. Each line is terminated by a carriage return only (hex 0D) and there is no end of file character.

```
seg_map_type = array[0..127] of byte ;
commo_upmap =
  record
    header_flag : longint ;
    frame_type : record_type ;
    upload_ok : boolean ;
    fname : string[63] ;
    segmap : seg_map_type
  end ;
```

```
upload_control_type = (CREATE_UPLOAD, SEND_UPLOAD, ABORT_UPLOAD, UPLOAD_DONE, MAP_ONLY) ;
upload_struc =
  record
    return_map : boolean ;
    case upload_control : upload_control_type of
      SEND_UPLOAD :
        (byte_offset : word ;
         byte_count : word ;
         seg_num : integer ;
         bytes : array [1 .. DP_TO_DA_MESSAGE_LENGTH - 10] of byte) ;
      CREATE_UPLOAD :
        (file_size : word ;
         file_name : string[63])
  end ;
```

35) To download a file from the DA to the DP is simpler than uploading. Send and DOWN_REQ packet with the file name, you will then get a "download_struc" using the "commo_reply" data structure with the "DOWNLOAD" record type. Process as you would any other structure that uses "commo_reply". After completely received you can then process it. If "filefound" is true then the requested file was found. If "toobig" is set, the file was found, but is larger than 65000 bytes and cannot be transferred. If the file was not found, or too big, then inform the user that the file was not transferred. If The file was found and was not too big, then "file_size" has the size in bytes of the file, and "contents" contains an image of the file. If the file you are downloading is a text file, you may need to convert the format before writing on your system. The DA text file format uses only a carriage return (hex 0D) as a line terminator, and there is no end of file character. Should the user want to abort a download in progress, send a DOWN_ABT command, this tells the DA to flush all pending download packets from it's packet queue.

```
download_struc =
  record
    filefound : boolean ;
    toobig : boolean ;
    file_size : word ;
    file_name : string[63] ;
    contents : array[1 .. 65000] of byte
  end ;
```

WORD ALIGNMENT CONSIDERATIONS

The DA is written using OmegaSoft Pascal for the 68000 series. It word aligns all multiple byte data types, but does not force longword alignment. In order to provide maximum compatibility with other systems, efforts have been made to expressly longword align any data structure longer than two bytes. There are exceptions in older data structures, among the new data structures, the only known exception to this rule is the "header_flag" field in the "commo_reply" structure. Since you can ignore this field, you may want to define it as two dummy integers if your system requires longword alignment of longword values. Variant records (unions in C) of course assign the same memory locations to multiple fields, since only one group is used at a time. The following example shows the byte offset from the start of the data structure for each field in angle brackets "<n>" in bold type :

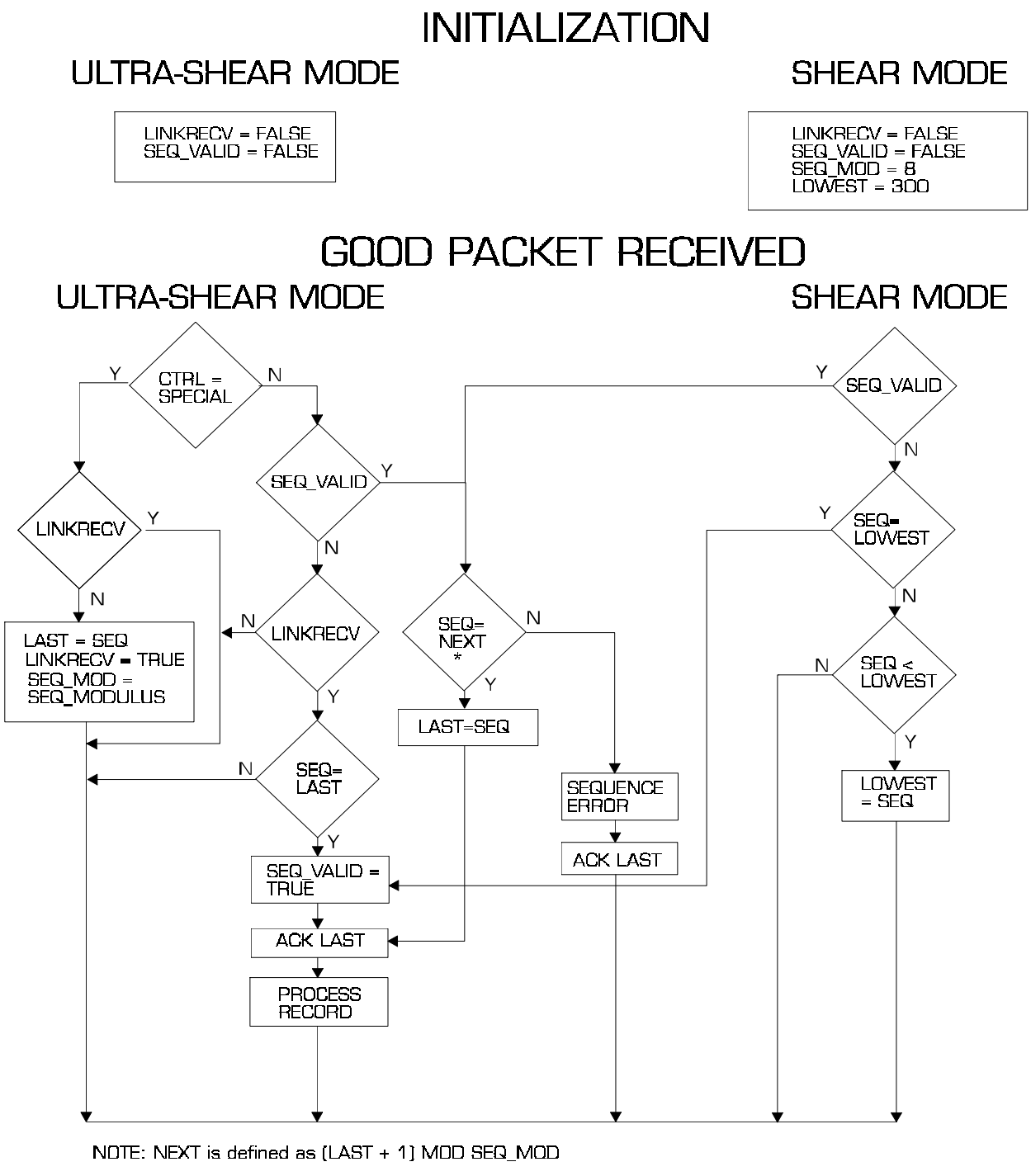
```
DP_to_DA_msg_type = record  {Different message and command types are send from DP to DA}
  case boolean of
    false : (x: array [0..(DP_TO_DA_MESSAGE_LENGTH-1)] of byte) ; <0>
    true :
      (cmd_type : DP_to_DA_cmd_type ; <0>
       dp_seq : byte ; <1>
       case DP_to_DA_cmd_type of
         AUTO_DAC_CORRECTION, ACCURATE_DAC_CORRECTION : (cmd_parms : stokely) ; <2>
         COMM_EVENT : (rc_sp1 : integer ; <2>
                       mask : longinteger) ; <4>
         ULTRA_MASS : (mbrd : integer ; <2>
                       mdur : integer) ; <4>
         ULTRA_CAL : (rc_sp2 : integer ; <2>
                      xc : extcal_type) ; <4>
         ULTRA_STOP : (sbrd : integer) ; <2>
         DET_REQ : (dr_name : seed_name_type ; <2>
                    rc_sp3 : byte ;      <5>
                    dr_loc : location_type) ; <6>
         DET_ENABLE : (rc_sp4 : integer ; <2>
                       de : det_enable_type) ; <4>
         DET_CHANGE : (rc_sp5 : integer ; <2>
                       dc : det_change_type) ; <4>
         REC_ENABLE : (rc_sp6 : integer ; <2>
                       re : rec_enable_type) ; <4>
         SHELL_CMD : (sc : string[80])) ; <2>
         LINK_ADJ : (la : link_adj_msg)) ; <2>
         DOWN_REQ : (fname : string[63]) ; <2>
         UPLOAD : (ups : upload_struc)) ; <2>
  end;
```

Pascal strings are stored with the first byte being the current length of the string, followed by up to the defined number of characters. In the structure below, "name" and "version" occupy 24 bytes, and "clockmsg" occupies 80 bytes.

```
digi_type =
    record
      name : string[23] ; {digitizer name}
      version : string[23] ; {program version number}
      clockmsg : string[79] ; {available clock messages}
      prefilter_ok : boolean ; {server supports prefiltering}
      detector_load_ok : boolean ; {server supports loadable detectors}
      setmap_ok : boolean ; {server will process a set map command}
      clockstring_ok : boolean ; {server will return clockstring}
      int_ext_ok : boolean ; {two mark sources}
      send_message_ok : boolean ; {can send digitizer messages}
      message_chan_ok : boolean ; {messages can be sent to channels}
      set_osc_ok : boolean ; {can set the master oscillator}
      set_clock_ok : boolean ; {can set the clock}
      wait_for_data : byte ; {number of seconds to wait for data}
      dt_sp1 : byte ;
      dt_sp2 : byte ;
    end ;
```

START-UP PROTOCOL

The protocol required when the DP starts depends on whether it is talking to a SHEAR or ULTRA SHEAR system. After the initial startup, the protocol is the same. In either case, the first valid packet to reach the DP may be in the middle of a DA transmission window.

In the non-ultra system, the best technique is to allow packets to be received without acknowledging or processing them, but keeping track of the lowest sequence number you receive. When you receive the lowest sequence number the second time, start acknowledging and processing packets at that packet.

In the ultra system, you must determine what the sequence modulus is. To do this, send a LINK_REQ command to the DA and wait for the LINK_PKT to be received. The LINK_PKT has the SEQUENCE_SPECIAL control type. The packet will contain the sequence modulus required and it's packet sequence will be the first packet you should acknowledge. After you have received the LINK_PKT, wait until the packet with the indicated sequence number is received, at that point start acknowledging and processing packets.