EDS-ASM SCIENCE MODES IMPLEMENTATION STUDY

Ilya Garelik

November 5, 1991

1 Position Histogram mode.

1.1 Back End Processing

In the back-end CPU this mode is implemented by three independent threads: science processing, rotation manager, and housekeeper. The science processing and housekeeper are synchronized to the rotation control. All three threads have similar command interfaces in which major commands are "initiate" and "terminate". The science processing and rotation manager threads also respond to "abort" command. The science processing thread can also receive "suspend" and "resume" commands.

In the ground specified commands to the rotation manager the rotation parameters (the dwell sequence) must be specified. Any ASM movement might be expressed in terms of dwell sequences (ref. "EDS - ASM Science Requirements" document). A sequence with non-zero dwell time and zero rotation angle (staring mode) also might be specified. Rotation time should be specified as at least 1 major frame in that case. It is necessary to have time to copy the data accumulated during the dwell time from the DSP buffer memory to the CPU memory. This kind of a dwell sequence is called "staring". Another special case of a dwell sequence is an individual rotation which may be commanded as a dwell sequence with 1 period and zero dwell time.

The ground is able to place the ASM rotation mechanism into the initial ("home") position. That position is defined as one of the soft limits. The ASM may be sent to that position by rotating towards it through the angle which is bigger than the whole range of the ASM rotation. Just before that the ASM must be rotated into the opposite direction through the angle bigger than that between soft limit and the mechanical stop. It is necessary in case when on powering up of the ASM the rotation mechanism is positioned between the mechanical stop and the soft limit.

The Ground supplies the following information to the rotation manager thread for the "initiate" command:

-time when to start rotation

-rotation steps number and sign (direction) - <rotation_steps>; rotation and settling phase duration in major frames - <rotation_time> can be calculated on board, this time must be not less then 1 to allow time to copy the data block from the DSP to the CPU memory.

-number of dwell periods - <number_of_dwells>.

-dwell phase duration in major frames - <dwell_time>.

-NOTE: data block is a position histogram accumulated during one dwell phase.

Superintervals are never used. Dwell time and rotation time are specified in major frames. Dwell time and rotation time each constitute the whole interval. This will result in alternating intervals of different lengths. During the dwell time the interval constant must be 1, during rotation - 0. The DSP code for this mode is organized in such a way that the events which contain interval constant 0 are discarded. The interval sequence is generated by the rotation manager thread. It also notifies the science processing thread of the event EV_ROTATION_BEGIN and the housekeeping thread of the event EV_ROTATION_END.

The science processing thread performs the following procedure: Just after every dwell (at the beginning of the rotation phase) the accumulated scientific data is copied from the DSP buffer memory to the CPU memory. Then this data is downlinked at a steady rate. This means that number of packets downlinked during one major frame must not exceed a ground-specified value.

APPLICATION NOTE: Every major frame the data is downlinked at as fast a rate as possible until the counter of packets downlinked during current major frame reaches a predefined value obtained from the science processing thread's parameter block. That counter is reset every major frame by the deferred major frame client function.

The housekeeper thread is designed to read cyclically some specified sensors and to send the obtained information to the ground. The thread starts its new cycle at the beginning of every dwell. Every minor frame during the cycle it activates the next sensor and reads the one activated at the previous minor frame. The information read (called engineering data block) is stored in the CPU memory. The science processing thread downlinks these data right before science data block.

Bit 9 of the EA command register (FIFO_CONTROL) should be deasserted because no FIRST_FLAG is needed in this mode. Reminder: FIRST_FLAG indicates the first accumulated event of the interval.

When the SAA (South atlantic anomaly) signal becomes active it is latched in the <SAA_flag>. This flag can be reset by the software only after the SAA signal becomes not active.

When the South Atlantic Anomaly is detected all three SSCs are shut off for 5 to 15 minutes. During the SAA time the DSP will not accumulate any new data. Data accumulation will resume only when the SSCs are turned back on. The DSP code for this mode is organized in such a way that it copies the accumulated data block to the CPU memory only when interval constant of the events in the input stream switches from 1 to 0. Therefore there will be no copying while the SSCs are shut off. To avoid hanging the CPU must not try to copy data blocks from the DSP buffer during that time. The first data block copied to the CPU memory after the SSCs are turned back on will be invalid because it may consist partly of the data collected during the dwell when SAA was detected before the SSCs were turned off and partly of the data collected during the dwell when the SSC were turned back on. This data block should be discarded. It is accomplished using <SAAD_flag>. At the beginning of the rotation phase <SAAD_flag> is tested. If it is set it means the SAA signal either is active or has ceased being active not earlier than at the previous rotation. Then the CPU tries to reset <SAA_flag>. If it fails (SAA signal is still active) no data block copying is performed by the CPU. If it succeeds the data block will be copied and downlinked but marked as "corrupted". There is only one "pathological" case when the SAA signal starts and ends during rotation phases. In that case a valid data block might be marked as "corrupted". It is up to the ground to find such blocks given the time when the data was copied from DSP buffer and time of the start and the end of the SAA signal.

Position Histogram mode.

If a dwell sequence is aborted (by the "abort" command to the rotation manager) the first data block downlinked at the next dwell sequence will be corrupted by the data accumulated in the DSP buffer memory at the last aborted dwell of the previous sequence and not flushed to the CPU memory. Such a block will be downlinked marked as "corrupted".

Nothing like that needs to be done if a dwell sequence is stopped by the "terminate" command to the rotation manager because in that case the sequence is exited in a "clean" way. If a "terminate" command is received during the dwell phase the sequence will be stopped after the consequent rotation phase. Otherwise (if a "terminate" commannd is received during the rotation phase) it will be stopped after the next rotaion phase.

Correct working of the DSP software depends on getting at least one event during rotation phase. It will always happen due to the background events rate.

The following is pseudocode that illustrates the ASM position histogram mode. Almost all the error checking code is omitted.

```
CPU THREAD <rotation_manager>

  FOREVER
  {
    wait for EV_NEW_UPLINK or EV_NEW_SC

    read the command

    IF "abort" or "terminate"
      ack NONE

    ELSE
    {
      ack OK

      IF <dwell_time> is zero -- an individual rotation
        CALL <perform_rotation>

      ELSE
        CALL <perform_dwell_sequence>
    }
  }

END <rotation_manager>



CPU SUBROUTINE <perform_rotation>

  given <rotation_steps> calculate <rotation_time>

  given the rotation start time from the parameter block
  calculate <time_before_rotation> - number of major frames
  before the rotation should start

  WHILE number of major frames passed is less then
```

```
        <time_before_rotation>
  {
    wait for EV_MAJOR or EV_NEW_UPLINK or EV_NEW_SC

    IF EV_NEW_UPLINK or EV_NEW_SC
    {
      read the command

      IF "initiate"
        ack CLOBBER

      ELSE -- the command is "abort" or "terminate"
      {
        ack OK

        RETURN from <perform_rotation>
      }
    }
  }

  rotate the ASM in correspondance with <rotation_steps>

  wait for <rotation_time> major frames

END <perform_rotation>



CPU SUBROUTINE <perform_dwell_sequence>

  given <rotation_steps> calculate <rotation_time>

  <dwell_seq_count> := <number_of_dwells>

  <dwell_in_progress> := FALSE

  <abort> =: FALSE

  <term> := FALSE

  <exit> =: FALSE

  CALL <dwell_sequence_setup>

  WHILE <end_of_dwell_sequence> is FALSE
  {
    wait for EV_END_INTERVAL or EV_NEW_UPLINK or EV_NEW_SC

    IF EV_NEW_UPLINK or EV_NEW_SC
    {
      read the command

      IF "initiate"
```

```
        ack CLOBBER

      IF "abort"
      {
        ack OK

        <abort> := TRUE

        IF <dwell_in_progress> is TRUE
          <mark_corrupted> := TRUE  -- if the "abort"command was received
                                       during a dwell phase the first data
                                       block of the next dwell sequence will
                                       be marked as "corrupted"; this is a
                                       global

      }

      IF "terminate"
      {
        ack OK

        <term> := TRUE
      }
    }

    ELSE -- EV_END_INTERVAL
    {
      IF <exit>                               -- it was the last interval
      {
         RETURN from <perform_dwell_sequence>
      }
    }
  }

END <perform_dwell_sequence>



CPU SUBROUTINE <dwell_sequence_setup>

  deassert FIFO control in the CPU command register -- in this mode
                                                       FIRST_FLAG
                                                       marker in the DSP
                                                       input FIFO is not
                                                       needed

  schedule the first interval marker according to the
  commanded dwell sequence start time

  schedule the first interval's duration <dwell_time> major frames

  <next_seq_active> := TRUE -- allow the data collection during the interval
                               sequence
```

```
  make interval constant for the first interval 1

END <dwell_sequence_setup>



CPU SUBROUTINE <interval_handler>

  invert <dwell_in_progress>

  invert <next_interval_const>

  IF <abort> is FALSE and <dwell_seq_count> is not 0 -- this interval marker
                                                         is not the last one
                                                         and the sequence is
                                                         not to be aborted
  {
    IF <dwell_in_progress>           -- dwell interval's starting marker
    {
      make next interval's duration <rotation_time>
      notify EV_ROTATION_END to the housekeeper thread
    }

    ELSE                             -- rotation interval's starting marker
    {
      decrement <dwell_seq_count>
      IF <term> is TRUE
       <dwell_seq_count> := 0
      IF <dwell_seq_count> is 0
        CALL TIMINGintervals_done()
      ELSE
        make next interval's duration <dwell_time>

      perform rotation in correspondance with <rotation_steps>
      notify EV_ROTATION_BEGIN to the science processing thread
    }
  }
  ELSE -- either <abort> is set or <dwell_seq_count> is 0 on
          <interval_handler> entry
  {
    IF <abort> is TRUE               -- interval sequence must be stopped
                                        immediately
    {
        CALL TIMINGstop()
        <dwell_seq_count> := 0       -- make sure next marker is the last
        <abort> := FALSE
    }
    ELSE                             -- the last interval marker's interrupt
    {
      <exit> := TRUE
    }
```

```
  }

END <interval_handler>



CPU THREAD <science_processing>

  FOREVER
  {
    wait for EV_NEW_UPLINK or EV_NEW_SC

    read the command

    IF "abort" or "terminate" or "suspend" or "resume"
      ack NONE

    ELSE -- the command was "initiate"
    {
        ack OK

        CALL <perform_pos_hist_mode> -- this function will be called only
                                        if the parameter block says to run
                                        the position histogram mode; the code
                                        checking this is not illustrated
    }
  }

END <science_processing>



CPU SUBROUTINE <perform_pos_hist_mode>

  setup DSP according to observation parameters

  CALL <BUFPsetup_pool(current downlink packet contents size)>

  <ret_ok> := TRUE -- this flag is set to FALSE if there was not enough
                      time to downlink all previously accumulated data
                      before copying the next block

  <mark_corrupted> := FALSE -- when TRUE this flag forces the next downlinked
                               data block to be marked as "corrupted"; this
                               is a global


  <SAA_inhibit_downlink> := FALSE -- set to TRUE to inhibit downlinking the
                                     data while SAA signal is active

  CALL <asm_pos_hist_bytes> -- returns <pos_hist_size>
```

Position Histogram mode.

```
allocate <pos_hist_size> bytes of heap memory -- temporary storage
                                                 for the data block
                                                 copied from the DSP
                                                 buffer memory

<data_block_number> := 0

given the data processing thread start time from the parameter block
calculate <time_before_data_processing> - number of major frames
before the data processing thread should start

WHILE number of major frames passed is less then
      <time_before_data_processing>
{
  wait for EV_MAJOR or EV_NEW_UPLINK or EV_NEW_SC

  IF EV_NEW_UPLINK or EV_NEW_SC
  {
    read the command

    IF "initiate"
      ack CLOBBER

    IF "abort" or "terminate"
    {
      ack OK

      RETURN from <perform_pos_hist_mode>
    }

    IF "suspend" then set <suspended>, ack OK

    IF "resume"  then clear <suspended>, ack OK
  }
}

FOREVER -- this loop executes until one of the following occurs:
           a terminate command is received; an abort command is
           received
{
A: wait for EV_NEW_UPLINK or EV_NEW_SC or EV_ROTATION_BEGIN

  IF EV_NEW_UPLINK or EV_NEW_SC
  {
    read the command

    IF "initiate" then ack CLOBBER

    IF "suspend" then set <suspended>, ack OK

    IF "resume" then clear <suspended>, ack OK

    IF "terminate" then set <terminate>, ack OK, exit FOREVER loop
```

```
      IF "abort" then set <abort>, ack OK, exit FOREVER loop
    }

    GOTO A -- continue FOREVER loop

     -- if EV_ROTATION_BEGIN

B:  IF <SAAD_flag> is TRUE -- active SAA signal was latched
    {
      reset <SAAD_flag>

      IF <SAAD_flag> is TRUE -- reset failed; it means SAA signal
                                is still active
        <SAA_inhibit_downlink> := TRUE -- the CPU will not get a new
                                          data block, so the data
                                          downlink should be inhibited

      ELSE                     -- SAA signal is no longer active
      {
        <SAA_inhibit_downlink> := FALSE

        <mark_corrupted> := TRUE -- global flag is set to mark the first
                                    data block downlinked after SAA as
                                    corrupted

        CALL <copy>
      }
    }

    ELSE -- no active SAA signal was latched
      CALL <copy>

    CALL <process_data_block>

    IF <abort> is set exit FOREVER loop

    IF <terminate> is set and <ret_ok> is TRUE
      exit FOREVER loop

    IF <ret_ok> is FALSE -- downlink of the data block finished prematurely
    {
      send error message

      IF <terminate> is set exit FOREVER loop

      <ret_ok> := TRUE

      GOTO B
    }

  } -- FOREVER loop
```

```
  IF <abort> is not set
    CALL <coda>

  CALL <asm_pos_hist_cleanup>

  deallocate storage, etc. used by this procedure

END <perform_hist_mode>



CPU SUBROUTINE <asm_pos_hist_bytes>

  return the <pos_hist_size> - number of bytes in the
  position histogram; this value depends on whether co-
  adding is used

END <asm_pos_hist_bytes>



CPU SUBROUTINE <copy>

  busy wait until the DSP output FIFO is not empty

  copy <pos_hist_size> bytes of data
  from the DSP buffer memory to the
  CPU memory                        -- before reading every byte from
                                       the FIFO check if it is empty;
                                       if it is that indicates an error
                                       because the DSP is writing to the
                                       FIFO much faster then the CPU is
                                       reading it

  <copy_time> := current major frame number -- this should be a part of data
                                               block header information

END <copy>



CPU SUBROUTINE <process_data_block>

  Do whatever necessary to downlink the engineering data block

  CHAR <buffer[]>

  <inhibited> := <suspended> is set or <SAA_inhibit_downlink> is TRUE

  P := <pos_hist _size>/<current_downlink_packet_size> + 1 -- number of
                                                              packets needed
                                                              to telemeter
                                                              the data block
```

Position Histogram mode.

```
<buffer> := BUFPget() -- also supplies a "done" flag

IF <buffer> was obtained
{
  -- build first data block header

  place <observation_id> from parameter block into <buffer>

  place major frame of observation start time (from initiate command)
  into <buffer>

  place <data_block_number> into <buffer>

  increment <data_block_number>

  place <pos_hist_size> into buffer

  place <overflow_flag> into <buffer>  -- <overflow_flag> is the flag set
                                          by the <interval_handler> routine
                                          in case of the DSP input FIFO
                                          overflow during this dwell phase
                                          (the code is not illustrated)

  place <mark_corrupted> into <buffer>

  place <copy_time> into <buffer>

   -- first data block header constructed

  CALL <asm_pos_hist_spool>(<buffer>, number of bytes needed for the first
  data block packet)

  IF not <inhibited>
  {
    post <buffer> via DOWNLINKpost(), using "done" flag supplied by the
    BUFPget that yielded buffer. This flag will be set to TRUE by the
    Downlink Packet module once this packet has been posted to the downlink
    FIFO.

    WHILE the post is not OK -- if the downlink queue was full wait until
                                the <buffer> can be posted
      wait for EV_DOWNLINK_AGAIN
  }

  ELSE -- telemetry is inhibited
    set the <buffer>'s "done" flag
}

ELSE -- a buffer was not obtained
{
 -- record this occurrence as housekeeping
 -- discard an appropriate amount of data
```

```
  }

  <packet_number> := 1

  LOOP BLOCK  -- this loop executes until all the data block is posted
                 or time to downlink this data block runs out
                 (EV_ROTATION_BEGIN is notified) or an  "abort" command
                 is received
  {
    <buffer> := BUFPget() -- also supplies a "done" flag

    IF <buffer> was obtained
    {
      IF this is not the last packet in the data block
        place <packet_number> into <buffer>

      ELSE -- final packet
        place -1 into buffer

      increment <packet_number>

      IF <packet_count> is less than <packets_per_major_frame>

       -- this condition must be checked before reading any new packet-sized
          chunk of information from the data block is read to ensure the data
          block is not downlinked at too fast a
rate;<packets_per_major_frame>
          is supplied in the parameter block, <packet_count> is reset every
          major frame
      {
        CALL <asm_pos_hist_spool>(<buffer>, number of bytes needed for this
        data block packet)
      }

      ELSE
      {
C:        wait for EV_NEW_UPLINK or EV_NEW_SC or EV_MAJOR or
EV_ROTATION_BEGIN

        IF EV_NEW_UPLINK or EV_NEW_SC
        {
          read the command

          IF "initiate" then ack "CLOBBER"

          IF "suspend" then set <suspended>, ack OK

          IF "resume" then clear <suspended>, ack OK

          IF "terminate" then set <terminate>, ack OK

          IF "abort" then
```

```
    {
      set <abort>, ack OK

      IF the data block is not completed
        post an empty downlink packet with a packet number of -2;
        details omitted

        RETURN from <process_data_block>
    }
    GOTO C
  }

  IF EV_ROTATION_BEGIN -- it means that the data block downlink
                          should be finished prematurely bacause
                          it is time to copy the next data block
  {
    <ret_ok> := FALSE

    RETURN from <process data block>
  }

  IF EV_MAJOR -- then <packet_count> has been reset
    CALL <asm_pos_hist_spool>(<buffer>, number of bytes needed for this
    data block packet)
}
IF not <inhibited>
{
D:  post <buffer> via DOWNLINKpost(), using "done" flag supplied by
    the BUFPget() that yielded <buffer>

    IF the post was not OK
    {
E:    wait for EV_NEW_UPLINK or EV_NEW_SC or EV_DOWNLINK_AGAIN or
               EV_ROTATION_BEGIN

      IF EV_NEW_UPLINK or EV_NEW_SC
      {
        read the command

        IF "initiate" then ack "CLOBBER"

        IF "suspend" then set <suspended>, ack OK

        IF "resume" then clear <suspended>, ack OK

        IF "terminate" then set <terminate>, ack OK

        IF "abort" then
        {
          set <abort>, ack OK

          IF the data block is not completed
            post an empty downlink packet with a packet number of -2;
```

```
                details omitted

              RETURN from <process_data_block>
            }
          GOTO E
          }

          IF EV_ROTATION_BEGIN -- it means that the data block downlink
                                  should be finished prematurely bacause
                                  it is time to copy the next data block
          {
            <ret_ok> := FALSE

            RETURN from <process data block>
          }

          IF EV_DOWNLINK_AGAIN
            GOTO D
        }
      }

      ELSE -- telemetry is inhibited
        set the <buffer>'s "done" flag
    }

    ELSE -- a buffer was not obtained
    {
      -- this should never happen because the buffer pool should be bigger
         then downlink queue and in this mode when the downlink queue is full
         no more data is read from the data block until a buffer is released
         from the downlink queue
      generate fault -- send error message and reset the CPU
    }
  } -- LOOP BLOCK

END <process_data_block>



CPU SUBROUTINE <coda>

  -- ref. to "Flight Software Global Ddesign", section PCA science
     processing modes, page 64

END <coda>



CPU SUBROUTINE <asm_pos_hist_coda>(<dest>)

  place TBD post-mortem data into the <dest> storage area,
  returning the number of bytes placed there
```

```
END <asm_pos_hist_coda>



CPU SUBROUTINE <asm_pos_hist_cleanup>

  deallocate the heap memory used as temporary storage
  for data blocks; perform all other necessary cleanup
  after an observation has ended

END <asm_pos_hist_cleanup>



CPU SUBROUTINE <asm_pos_hist_spool>(<dest>, <num_bytes>)

  read next num_bytes of data from the heap copy of
  the data block into the dest storage area

END <asm_pos_hist_spool>



CPU SUBROUTINE <count_reset>

   -- this subroutine is called every major frame at the deferred
      interrupt priority level

  <packet_count> := 0

END <count_reset>



CPU THREAD <housekeeper>

  FOREVER
  {
    wait for EV_NEW_UPLINK or EV_NEW_SC

    read the command

    IF "terminate"
      ack NONE

    ELSE -- the command was "initiate"
    {
        ack OK

        CALL <perform_housekeeping>
    }
  }
```

```
END <housekeeper>



CPU SUBROUTINE <perform_housekeeping>

  given the housekeeper thread start time from the parameter block,
  calculate <time_before_housekeeping> - number of major frames be-
  fore the housekeeper thread should start

  WHILE number of major frames passed is less then
        <time_before_housekeeping>
  {
    wait for EV_MAJOR or EV_NEW_UPLINK or EV_NEW_SC

    IF EV_NEW_UPLINK or EV_NEW_SC
    {
      read the command

      IF "initiate"
        ack CLOBBER

      IF "terminate"
      {
        ack OK

        RETURN from <perform_housekeeping>
      }
    }
    ELSE -- EV_MAJOR
      increment the count of major frames passed
  }

  <data_acq_active> := FALSE --this flag indicates whether <housekeep_minor>
                              routine should perform data acquizition

  FOREVER
  {
    wait for EV_ROTATION_END or EV_NEW_UPLINK or EV_NEW_SC

    IF EV_NEW_UPLINK or EV_NEW_SC
    {
      read the command

      IF "initiate" then ack "CLOBBER"

      IF "terminate"
      {
        ack OK

        RETURN from <perform_housekeeping>
      }
```

```
      continue the FOREVER loop
    }

    -- otherwise EV_ROTATION_END has been notified

    <data_acq_active> := TRUE

    <sensor_count> := <number_sensors>

  }

END  <perform_housekeeping>


CPU SUBROUTINE <housekeep_minor> --installed as a minor frame client function

  IF <data_acq_active> is TRUE -- data acquisition allowed
  {

    IF <sensor_count> is not equal to <number_sensors> -- not first sensor
      read ASM Command/Housekeeping register and store it into
      engineering data block

    ELSE -- first sensor
      store current major frame number into engineering data block

    decrement <sensor_count>

    IF <sensor_count> is not 0 -- not last sensor
      write ASM Command/Housekeeping register to activate next sensor

    ELSE -- last sensor has been read
      <data_acq_active> := FALSE -- inhibit data acquisition until next dwell
  }

END <housekeeper_minor>
```

Position Histogram mode.

DSP event processing.

Most of the time the DSP is polling its input FIFO. When the FIFO flag indicates that it is not empty 3 (TBD) event-words describing the event are read. If the interval constant is 1 which means that the event occured during rotation phase, the event-words are discarded. Otherwise they are processed and accumulated as a position histogram. When the interval constants of two consequent events is seen to change from 0 to 1 which corresponds to the transition from a dwell phase to a rotation phase, the accumulated histogram is posted to the CPU memory via the output FIFO. This approach introduces some latency between the beginning of the rotation time and the start of posting of the histogram because the beginning of the rotation phase will be "noticed" by the DSP software only after the appearance of the first event of the rotation phase. With the minimal ASM event rate of 300 events/second this latency could be about 3 msec. Copying of the histogram to the CPU memory takes about 0.25 sec and a rotation phase duration is not less than 1 sec. So the copying process will be over by the beginning of the next dwell phase i.e. by the beginning of the next histogram accumulation.

Note:

Ground-supplied parameters are passed to the DSP via the parameter block named block which is loaded into the DSP scratch memory by the DSP monitor program.

The following is pseudocode that illustrates the DSP working in the ASM position histogram mode.

```
global <num_anodes> -- received through the parameter block (1 if co-adding,
                       9 if not)

global <num_SSCs> = 3

global <num_energies> -- received through the parameter block

global <num_positions> = 256

global <size_of_position_histogram> -- calculated using the data received
                                       through the parameter block

global <lo_energy>[3], <hi_energy>[3] -- high and low raw_energy limits over
                                         and below which the "good" events
are
                                         discarded these values are received
                                         through the parameter block; 2
values
                                         are specified for every SSC

global array <break_points>[3][<num_energies>-1] -- received through the
                                                    parameter block; sepa-
                                                    rate set of breakpoints
                                                    is supplied for every SSC

global <the_event> -- this event describing structure is derived from the
                      event describing event-words as they appear on the
                      event bus
{
```

Position Histogram mode.

```
  interval_const

  phA

  phB

  SSC_id

  anode_code

  position -- derived from phA and phB

  raw_energy -- derived from phA and phB

  pulse_height -- derived from raw_energy
}

 -- the following array is a part of the DSP buffer memory
    used for the position histogram accumulation

global array of 16 bit integers
<position_histogram>[256*5*9*3] -- 256 (TBR) is a max number of positoin bins
                                   5 (TBR) is a max number of pulse height
bins
                                   9 is a number of anodes
                                   3 is a number of detectors




DSP PROGRAM <pos_hist_mode>

   -- initialize the DSP for the position histogram mode

  zero <position_histogram>

  LOOP FOREVER -- this position histogram accumulation loop is exited
                  only when the DSP is reset
  {
    wait for the next event-word to become available at the
    DSP input FIFO;

    IF <the_event.interval_constant> is zero
    {
      discard this event-word

      read and discard next two (TBD) event-words from the input FIFO

      IF <old_interval_const> is not zero
        CALL <copy>

      <old_interval_const> := 0
```

```
      CONTINUE LOOP FOREVER
    }

    <old_interval_const> := 1

    store this event-word

    read and store next two (TBD) event-words from the input FIFO

    CALL <event_extract>(the input FIFO words)

    IF <event_check>() returns ERROR
      continue the loop -- discard the event

    <the_event.pulse_height> := <pulse_height>(<the_event.raw_energy>,
                                 <num_energies>,
<break_points>[the_event.SSC_id])

    CALL <pos_hist_accumulate>
  }

END <pos_hist_mode>



DSP SUBROUTINE <event_extract>(the input words) -- the interval_constant and
                                                   time_tag are not used in
                                                   the position histogram
mode

  <the_event.phA> := the appropriate bits from the input words

  <the_event.phB> := the appropriate bits from the input words

  <the_event.SSC_id> := the appropriate bits from the input words

  <the_event.anode_code> := the appropriate bits from the input words

  <the_event.raw_energy> := <the_event.phA>+<the_event.phB>

  <the_event.position> := (<num_positions>*<the_event.phA>)/
                          (<the_event.raw_energy>+1)

   -- +1 in the denominator causes the position result to be in the range
      0..255; also prevents divide error when phA=phB=0.

END <event_extract>



DSP FUNCTION <pulse_height>(<energy_value>, <number_of_energy_bins>,
                            <array_of_breakpoints>)
```

```
   -- <array_of_breakpoints> should be sorted from low to high

  16 bit integer pointer <array_of_breakpoints>

  IF <energy_value> is greater then the last member of <array_of_breakpoints>
    RETURN (index to the last member of <array_of_breakpoints> + 1)

  ELSE
    RETURN index to the lowest  member of the <array_of_breakpoints>
           which is greater than <energy_value>

END <pulse_height>



DSP FUNCTION <event_check>

   -- this function returns ERROR if the event should be discarded

  IF <the_event.anode_code> is not 0..8 RETURN ERROR

  IF <the_event.phA> is zero RETURN ERROR

  IF <the_event.phB> is zero RETURN ERROR

  IF <the_event.raw_energy> is less than <lo_energy>[<the_event.SSC_id>]
    RETURN ERROR

  IF <the_event.raw_energy> is greater than <hi_energy>[<the_event.SSC_id>]
    RETURN ERROR

END <event_check>



DSP SUBROUTINE <pos_hist_accumulate>

  index := (<the_event.position> shifted left 7 times) +
           (<the_event.pulse_height> * 27) +
           (<the_event.anode_code> * 3) +
           (<the_event.SSC_id>)

  increment <position_histogram>[index]

END <pos_hist_accumulate>



DSP SUBROUTINE <copy>

   -- post the accumulated position histogram to the CPU

  UNTIL <size_of_position_histogram> 16 bit words are posted
```

```
  {
    IF output FIFO is not full
    {
      stuff the next 16 bit word from the <position_histogram>
      into the output FIFO

      zero corresponding word of the <position_histogram>
  }

END <copy>
```

2 Multiple Time Series mode.

2.1 Back End Processing

The CPU PCA Science Processing Skeleton (ref. to the "Flight Software Global Design" docoment) is entirely applicable for this ASM mode. There are only two differences:

1. The interval interrupt ISR should keep track of "big superintervals" and switch the second interval constant at the "big superinterval" marker.

2. The <mode_partition_bytes> function for this mode should normally return <size_of_time_series> + <size_of_non_X_ray_background>, but for the "big superinterval it should return <size_of_time_series> + <size_of_non_X_ray_background> + <size_of_pulse_height_spectrum>.

2.2 DSP Event Processing

The main challenge of this mode is the requirement to accumulate 3 histograms simultaneously: Time Series with time bin duration 0.001 to 1 sec, Non-X-ray Background with time bin duration 1 sec and Pulse Height Spectrum with bin duration 10 sec to 1600 sec. The event processing should also be performed on the continuous basis. It means that the the DSP must transfer the previously accumulated data to the CPU while it is accumulating new data. So only half of the DSP buffer memory can be used for the data accumulation while the contents of the second half (partition) is being posted to the CPU. It leaves us only 16K of 16 bit words for the three histograms. Simple calculations show that it is much less than needed to store the three histograms' data collected for the Pulse Hight Spectrum time bin duration.

The solution is to use the DSP buffer memory only for accumulation of the Time Series and Non-X-ray Background histograms' data switching between two partitions every major frame (which corresponds to the Non-X-ray Background histogram's time bin). Pulse Height Spectra will be accumulated in the DSP scratch RAM area. That area's magnitude must be:

(No. of pulse height bins)*(No. of anode bins)*(No. of detectors)*(No. of partitions), where:

No. of pulse height bins <= 48, No. of anode bins = 9, No. of detectors = 3, No. of partitions = 2.

The DSP data accumulation program will switch between these two partitions every Pulse Height Spectrum time bin.

Bit 9 of the EA command register (FIFO_CONTROL) should be deasserted because no FIRST_FLAG is needed in this mode.

Both the CPU science processing and the DSP software for this mode resemble the PCA ASM working in the binned mode (ref. to the "PCA Science Modes Implementation Study"). The superunterval duration is equal to the Non-X-ray background timebin (which always equals to 1 major frame), and time bin duration is equal to the Time Series time bin. The major differences are:

-Because no look-up tables are employed much more of event processing is performed by the DSP.

-Both interval constants are used. The first is used to switch the DSP event accumulation for the Time Series and Non-X-ray background between two buffer memory partitions, the second is employed to switch for the Height Spectra between two scratch memory partitions. The first switch occurs every superinterval, the second happens every "big superinterval" which is equal to the Pulse Height spectrum time bin and should consist of an integer number of superintervals.

-Both the CPU science processing and the DSP software keep track of "big superintervals", transferring from the DSP memory to the CPU and then downlinking not only Time Series and Non-X-ray backgroung, but also Pulse Height spectrum at every "big superinterval boundary"..

-Two values of <back_quantum> are passed by the ground to the DSP. <back_quantum1> is used for the superintervals when only Time Series and Non-X-ray Background are transferred to the CPU. <back_quantum2> is used for "big superintervals" when Pulse Height Spectrum is also transferred.

-The PCA Science Processing Skeleton (ref. to the "Flight Software Global De sign" document is fully applicable for this ASM mode. The <mode_partition_bytes> mode specific function should in this case return (<size of time series histogram> + <size of non-x-ray background histogram> + <size of pulse height spectrum>) for "big superintervals" and that value less <size of pulse height spectrum> for all the other superintervals.

The following pseudocode illustrates the DSP software for the ASM EA Multiple Time Series Mode:

```
global <num_anodes> = 9

global <num_energies_time_series> -- number of energy bins for the Time
                                     Series; it is received through the
                                     parameter block

global <num_energies_pulse_height>[3] -- number of energy bins for the Pulse
                                         Height Spectra; received through the
                                         parameter block; separate value is
                                         specified for every detector

global <types_background_data> = 4

global <num_SSCs> = 3

global <SSCs_in_use> -- 3 least significant bits of this byte indicate which
                        detectors' data is used for the Time Series accumula-
                        tion

global <time_series_time_bins_per_major>

global <time_bin_pulse_height>
```

Multiple Time Series mode.

```
 -- the following are low and high raw energy limits below and over which the
    "good" events are discarded; separate values are supplied for every SSC
    for the both time series and pulse height spectra

global <lo_energy_time_series>[3]

global <hi_energy_time_series>[3]

global <lo_energy_pulse_height>[3]

global <hi_energy_pulse_height>[3]

global array <break_points>[3][<num_energies_time_series>-1] -- received
thro-
                                                                ugh the para-
                                                                meter block;
                                                                separate set
                                                                of
breakpoints
                                                                is supplied
                                                                for every SSC

global <the_event> -- this event describing structure is derived from the
                      event describing event-words as they appear on the
                      event bus
{
  interval_const_1

  interval_const_2

  time_tag

  phA

  phB

  SSC_id

  anode_code

  raw_energy -- derived from phA and phB

  type -- is meaningful only for non-X-ray background data
}

 -- the following array is a part of the DSP buffer memory
    used for the time series accumulation

global array of 16 bit integers
<time_series>[2][5*1000*3]       -- 2 is a number of partitions
                                    5 (TBR) is a max number of energy bins
                                    1000 (TBR) is a max number of time series
                                    time bins in 1 sec
```

Multiple Time Series mode.

```
                                3 is a max number of detectors

 -- the following array is a part of the DSP buffer memory
    used for the non-X-ray background accumulation

global array of 16 bit integers
<non_X_ray>[2][4*3]            -- 2 is a number of partitions
                                  4 is a number of types of background data
                                  3 is a number of detectors

 -- the following array is a part of the DSP scratch memory
    used for pulse height spectra

global array of 16 bit integers
<pulse_height_spectrum>[2][48*9*3] -- 2 is anumber of partitions
                                     48 (TBR) is a max number of energy bins
                                     9 is a number of anodes
                                     3 is a number of detectors

global array of 16 bit integers
<size_of_array_to_send>[3]     -- contains the sizes of above arrays



DSP PROGRAM <multiple_time_series>(<back_quantum>)

 -- the <back_quantum> value depends on the compression accomplished by the
    partition map employed and on the telemetry allocated for this EA (for
mo-
    re information ref. to the "PCA Science Modes Implementation Study" p.
19)
    this value shell be calculated by the ground and passed through the para-
    meter block.

  CALL <prepare>

  wait for initial interval interrupt

  LOOP forever -- the CPU will reset the DSP when the observation is done
  {
    -- call <front> until superinterval ends

    DO
      CALL <front>
    UNTIL status register indicates a superinterval marker

    -- now all the just ended superinterval's events still left in the DSP
       input FIFO should be read; to do that call <front> until the FIFO is
       empty or the interval constant changes

    LOOP clean-up
    {
      CALL <front>
```

```
      IF <front> was exited because the FIFO is empty
        exit LOOP clean-up

      IF <the_event.interval_const_1> for the last event read by the <front>
      is not equal to <interval_const_1>
      {
        exit LOOP clean-up
      }
    }

    CALL <superinterval_boundary>

    LOOP back_and_front -- this loop executes until back task is finished
    {
      CALL <back>(back_quantum)

      IF a flag indicates that the back task has finished
      {
        IF status register indicates a superinterval marker -- superinterval
        {                                                        already ended
          -- this should never happen!

          set superinterval sync error flag for the CPU to see
          after the observation is finished
        }

        EXIT LOOP back_and_front
      }

      CALL <front>
    }
  }



DSP SUBROUTINE <prepare>

<interval_const_1> := 0   -- this variable always has value of the first
                             interval constant for the current superinterval;
                             it is used to find the superinterval boundary in
                             the event stream; attention: FIRST_FLAG is not
                             employed in this mode

<interval_const_2> := 0   -- this variable always has the value of the second
                             interval constant for the current superinterval

<size_of_time_series> :=
<num_energies_time_series>*<time_series_time_bins_per
                         _major>*<num_detectors> -- <num detectors> is
derived
                                                    from the <SSC_in_use>
value
```

Multiple Time Series mode.

```
<size_of_non_X_ray_background> := <types_background_data>*<num_SSCs>

<max_energies_pulse_height> := value of the biggest member of the array
                               <num_energies_pulse_height>[3]

<size_of_pulse_height_spectrum> := <max_energies_pulse_height>*<num_anodes>*
                                   <num_SSCs>

<supintervals_per_big_supinterval> := <time_bin_pulse_height>

<size_of_array_to_send>[0] := <size_of_non_X_ray_background>

<size_of_array_to_send>[1] := <size_of_time_series>

<size_of_array_to_send>[2] := <size_of_pulse_spectrum>

zero both partitions of <time_series>, <non_X_ray>, <pulse_height_spectrum>

END <prepare>



DSP SUBROUTINE <front>

DO 100 (TBR) times
{
  IF input FIFO is empty
    RETURN from <front>

  read and store 3 (TBR) event-words from the input FIFO

  CALL <event_extract>(the input FIFO words)

  CALL <event_check>(<the_event>)

  IF <ts> is TRUE
  {
    <pulse_height_time_series> := <pulse_height>(<the_event.raw_energy>,
    <num_energies_time_series>, <break_points>[<the_event.SSC_id])

    -- accumulate this event in the time series

    <index> := <the_event_SSC_id>*<num_energies_time_series>*<time_series_
               time_bins_per_major>+
               <pulse_height_time_series>*<time_series_time_bins_per_major>+
               <the_event.time_tag>

    increment <time_series>[<the_event.interval_const_1>][index]
  }

  IF <phs> is TRUE
  {
```

```
    <pulse_height_height_spectrum> := <num_energies_pulse_height>[<the_event.
    SSC_id>]*(<the_event.raw_energy>-<lo_energy_pulse_height>[<the_event.
    SSC_id>])/(<hi_energy_pulse_height>[<the_event.SSC_id>]-
    <lo_energy_pulse_height>[<the_event.SSC_id>])

    -- accumulate this event in the pulse height spectrum

    index := <the_event.SSC_id>*<num_anodes>*<num_energies_pulse_height>[
             <the_event.SSC_id>]+<the_event.anode_code>*
             <num_energies_pulse_height>[<the_event.SSC_id>]+
             <pulse_height_height_spectrum>

    increment <pulse_height_spectrum>[<the_event.interval_const_2>][index]
  }

  IF <nxb> is TRUE
  {
    -- accumulate this event in the non X-ray background

    index := <the_event.SSC_id>*<types_background_data>+
             <the_event_type>

    increment <non_X_ray>[<the_event.interval_const_1>][index]
  }
}

END <front>



DSP SUBROUTINE <event_extract>(3 event-words)

given 3 event words fill in <the_event> structure

END <event_extract>



DSP <event_check>(the_event)

given <the_event> structure classify the event:
set 3 flags: <ts> := TRUE if it is a good event for time series (only
             (SSC_in_use> specified detector's events are considered)
             <phs> := TRUE if it is a good event for pulse height spectrum
             <nxb> := TRUE if it is a good event for non X-ray background
in the latter case also set: <type> := type of the background event

END <event_check>



DSP FUNCTION <pulse_height>(<energy_value>, <number_of_energy_bins>,
                            <array_of_breakpoints>)
```

```
   -- <array_of_breakpoints> should be sorted from low to high

  16 bit integer pointer <array_of_breakpoints>

  IF <energy_value> is greater then the last member of <array_of_breakpoints>
    RETURN (index to the last member of <array_of_breakpoints> + 1)

  ELSE
    RETURN index to the lowest  member of the <array_of_breakpoints>
           which is greater than <energy_value>

END <pulse_height>



DSP SUBROUTINE <superinterval_boundary>

increment the superinterval count and, comparing it with the value of
<superintervals_per_big_superinterval> find out if the next superinterval
will be the "big" one

<part_to_send_buf> := 1 - <interval_const_1> -- is used to indicate the pa-
                                                rtition to be sent to the
                                                CPU for <non_X_ray> and
                                                <time_series>

<interval_const_1> := 1 - <interval_const_1>

IF next is a "big superinterval"
{
  <num_arrays_to_send> := 3

  <part_to_send_scratch> := 1 - <interval_const_2> -- is used to indicate the
                                                    partition to be sent to
                                                    the CPU for <pulse_hei-
                                                    ght_spectrum>

  <interval_const_2> := 1 - <interval_const_2>
}

ELSE
  <num_arrays_to_send> := 2

<part_to_send>[0] := <part_to_send_buf> -- the <partition_to_send> is an
array
                                           of 3 integers (its members
indicate
<part_to_send>[1] := <part_to_send_buf>    the numbers of partitions to send
                                           to the CPU during the next
superin-
                                           terval for all three arrays)
<part_to_send>[2] := <part_to_send_scratch>
```

Multiple Time Series mode.

```
<ind> := 0 -- this  indicates which array to send; 0 means: begin sending
              the accumulated data to the CPU beginning with <non_X_ray>

<transfer_count> := <size_of_array_to_send>[ind]

<partition> := <part_to_send>[ind]

<transfer_pointer> := 0 -- this variable indicates the 16 bit bin to be sent
                           to the CPU

set a flag showing we are on the first pass

END  <superinterval_boundary>



DSP SUBROUTINE <back>(<back_quantum>)

DO <back_quantum> times
{
  IF <transfer_count> <= 0 then EXIT DO -- back task already finished

  IF output FIFO is full then EXIT DO -- do not overrun CPU

  SWITCH <ind>
  {
  CASE 0:

    write <non_X_ray>[<partition>][<transfer_pointer>] to output FIFO

    IF not sending twice or we are on second pass
      <non_X_RAY>[<partition>][<transfer_pointer>] := 0

    BREAK from SWITCH

  CASE 1:

    write <time_series>[<partition>][<transfer_pointer>] to output FIFO

    IF not sending twice or we are on second pass
      <time_series>[<partition>][<transfer_pointer>] := 0

    BREAK from SWITCH

  CASE 2:

    write <pulse_height_spectrum>[<partition>][<transfer_pointer>] to output
    FIFO

    IF not sending twice or we are on second pass
      <pulse_height_spectrum>[<partition>][<transfer_pointer>] := 0
  }
```

```
  increment <transfer_pointer>

  decrement <transfer_count>

  IF <transfer_count> just decremented to 0
  {
    IF sending twice and we were on the first pass
    {
      -- setup a second pass

      <transfer_pointer> := 0

      <transfer_count> := <size_of_array_to_send>[ind]

      set a flag noting that we are on the second pass
    }

    ELSE -- either not sending twice or finished second pass
    {
      -- setup for next accumulation

      increment <ind>

      IF <ind> >= <num_arrays_to_send> -- all the accumulated data is sent to
      {                                   the CPU
        set flag indicating that the
        back task has finished

        EXIT DO
      }

      ELSE
      {
        <transfer_count> := <size_of_array_to_send>[ind]

        <partition> := <part_to_send>[ind]

        <transfer_pointer> := 0
      }
    }
  }
}

END <back>
```