ALEPH 91-142
MINIV 91-05
G. Lütjens et al.
15.10.91

The Sirocco Readout Program

G.Lütjens, H. Seywerd and S. Walther

15 October 1991

Abstract

We describe the programs used in the Sirocco for the readout and calibration of VDET. The readout program performs pedestal and common mode subtraction and cluster finding on the ADC data from a module, and formats the cluster information into output banks. The calibration program calculates pedestals and noises for all channels and puts them into a form where they may be uploaded for later use. These programs were written in Motorola DSP 56001 assembly language and familiarity with the processor is assumed here.

1 Introduction

This note provides detailed documentation for the programs used for reading out the ALEPH silicon vertex detector with the Sirocco module. It is organized as follows. In the introduction we present a brief description of the hardware of the Sirocco module, and a description of the interface between the hardware and the DSP processor. In the following sections we describe the two DSP programs: the event readout program used in datataking, and the program used to calibrate the detector. First a general description of the algorithms are given, then the structure of the assembly language routines are described in detail. In the appendix the data structures used in the DSP are described.

In the first run of the ALEPH vertex detector the readout system was based upon modified Time Projection Digitizers. These modules have little processing capability, and a limited digitization precision. It was decided [1] that they be upgraded to Sirocco modules, which were developed originally for the silicon vertex detector of the DELPHI experiment. Each of these modules [2] has two identical functional units consisting of a 10 bit flash ADC, a Motorola DSP 56001 digital signal processor [3], memory, and a Fastbus interface. The DSP is a Harvard architecture processor with separate instruction and two data memories and paths. In addition there is a front panel connector for ECL signals that trigger the FADC conversion sequence and provides an account number which is used for buffer control and event synchronization. A block diagram of the Sirocco is shown in figure 1.

The sequence for processing one event starts upon the receipt of a trigger on one of the lines of the the front panel bus. This starts the digitization of the analogue input with the bucket clock signal supplied from the front panel connector. The two lowest bits of the account number present on the connector direct the digitized data to be placed in one of

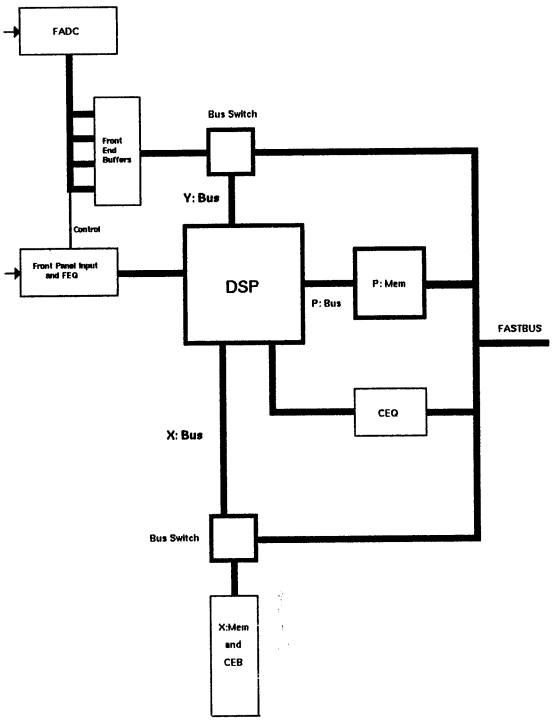

Figure 1: Block diagram of one of the two identical functional units of a Sirocco module. The major parts and the data paths between them are shown. Arrows indicate input from the front panel connectors. The bus switches allow control over the busses to be given to the processor, to the front end control logic, or to external Fastbus masters.

four Front End Buffers (FEB) which may be accessed by the DSP over its Y memory bus. These raw data are also directly accessible from Fastbus. A DSP program can poll a flag register (PCD) to test for the arrival of an event, and then process the data in the buffer. Event numbers ("Account numbers") are stored for every trigger in the Front End Queue (FEQ), a 16 word deep FIFO. They are used to identify events and establish the order in which they are processed.

The DSP also has access to a second data memory on the X bus, this is a 32 bit wide memory, accessible to Fastbus, called the Crate End Buffer (CEB). A special register allows transfers of the upper eight bits of a thirty-two bit word to be made from the twenty-four bit DSP data paths. A DSP program can use part of this memory to store processed data for readout from Fastbus. Parts of it may also be used for the storage of constants and as scratch memory. There is another FIFO, 16×2 words deep, writable from the DSP and readable from Fastbus, which is used to transfer the start and end addresses of processed events.

Bit Field	Function
0–13	Corrected ADC data (multiplied by four, two bit offset)
14	ADC Overflow if set
15	Data Invalid if set

Table 1: Structure of halfword of the VPLH bank.

2 The Readout Program

The readout program processes the input raw ADC data and produces two types of output banks, one containing start address and sizes of hit clusters, the other the corrected pulse heights of the channels associated with hits.

2.1 The Event Processing Algorithm

The ADC data are stored in the Front End Buffer in a form where the ϕ and z pulseheights of a wafer are interleaved. This is due to the hardware multiplexing performed by the readout electronics sequencer. The data are demultiplexed according to a downloaded table which indicates from which line driver in z and in ϕ the data originated. Each ADC word is multiplied by a factor four to preserve precision in further calculations. The data are checked and invalidated if it is in overflow or originated from a vetoed channel. Pedestals from a previous calibration run are subtracted from the data words. These data are then supplied to the common mode noise calculation routine. For each block of 256 channels (read by one line driver) the contents of all valid channels are histogram. A peak finding routine searches for the most frequent channel and uses the corresponding histogram bin as the common mode noise value for the data block. This value is subtracted from all channels, and saved as part of the output bank. The final pulse heights are determined from the data as follows:

$$PH_i = 4ADC_i - Ped_i - CM_{block},$$

where ADC is the raw data, Ped the pedestal for the channel and CM_{block} is the common mode noise for the line driver. The corrected data are supplied to a clustering routine. This searches for a valid channel above a single hit threshold and forms a cluster. The cluster consists of a fixed number of channels above and below the hit channel. If there are several channels above threshold within the range of a cluster, the cluster is expanded. BOS format output banks are formed, VHLS containing a list of clusters with the start strip address, and length, and VPLH containing the ADC data and validation flags, the structure of the data words in VPLH is given in table 1. The BOS banks are numbered as the readout modules.

After the data from one event is processed the remaining time until the arrival of the next event is used to correct the pedestals, for possible drifts. A correction factor for each

channel is calculated according to the formula:

$$C_i = \frac{Ped_i - 4Chan_i}{\kappa},$$

where Ped_i is the old pedestal, $Chan_i$ is the channel content, and κ is a constant greater than one, which is selected to moderate fluctuations (currently set at 32). Data in overflow is not used, and sanity checks are made to require the correction factor to be "reasonable". This correction factor is then subtracted from Ped_i. This way, if the pedestal in the detector shifts, the pedestal value will converge to the new value exponentially.

2.2 The Program

In the following sections a detailed description of the program is given.

2.2.1 Initialization

After the program is downloaded into Sirocco memory, and the DSP is reset program execution is directed through the block of exception vectors in the routine Interrupt_Vector to the routine Main. The exception vectors also dispatch all other exceptions to the routine FATAL so that a controlled crash of the DSP can be performed. Main then calls the routines Boot, and Clean_Regs.

The boot routine performs some hardware setups on the Sirocco and DSP. It sets the number of memory wait cycles by writing to the DSP BCR register, and the Sirocco front panel lights are switched off. The DSP memory configuration is set up by copying the first 512 words of program memory from the Fastbus accessible, to the DSP internal (on chip) memory, and setting the OMR register such that the internal DSP memories are used. The routine Clean_Regs initializes Sirocco address registers to zero.

After this initialization the program waits for the hex ($) value $AFFE, to be loaded to the location **X**:BootWaitFlag. This allows the externally controlled Sirocco registers to be reset before the DSP program begins processing. After $AFFE is written, the buffer management system is initialized, by calls to FEBInit, and CEBInit, which set up the initial pointers for reading from and writing to the Front End (Input) and Crate End (Output) buffers, respectively. Two further initialization routines are then called Load_P_Data, which initializes constants in the internal program memory, and UP_Init, which sets values for the pointers used by the pedestal updating program.

Upon completion of the initialization, a jump is made into the event processing loop, Event_Loop, which never returns.

2.2.2 Event Processing

The event processing loop waits for new events in the Front End Buffer and when one arrives, does the data reduction and writes banks to the Crate End Buffer.

Upon entry to the routine Event_Loop the program waits for an event through polling the carry flag after calls to the routine FEBOpen. FEBOpen checks the PCD register which contains a flag set when a new event is present in the FEB. Upon receipt of an event

the routine gets the start address and the account number of the event from the FEQ, this is stored to the location **X**:Accnt, and is used to set up the **r0** register to point to the first word of the event in the FEB (i.e. Y memory).

After an event has been received, the program updates the status information, and loads constants into the program memory data area (subroutine Load_P_Data), and calls the ProcessData routine. In ProcessData a slot in the output Crate End Buffer is booked by a call to OutBook, this sets up the **r1**, **n1**, and **m1** address registers to point to the next free location in the CEB. These registers are set up to provide a circular buffer. The area used for the temporary storage of the common mode is cleared. In order to provide raw data for offline checking from every DSP, the low part of the account number is compared with the DSP number in **X**:VdspNDSP, and if they are equal the Front End Buffer is copied to the Crate End Buffer as the VFEB bank (routine Make_VFEB). After this preliminary processing index registers are prepared for the data processing. The raw event is processed as eight sections of 256 data words each corresponding to the $t_1, t_2, \phi_1, \phi_2, z_1, z_2, z_3, z_4$ blocks. The t_1 and t_2 blocks, corresponding to the test inputs of the multiplexer module, are not processed. The **r7** register is used to count through these blocks, and is initialized to 2 to point to ϕ_1. The pointer to the temporary list in the X memory of pulse heights produced by the clustering, **r6**, is initialized to the value TempPH. The hit list bank is opened in the CEB by a call to BOSHead, with the bank name, VHLS loaded in the **a2**, and **a1** registers. The loop over the data blocks is started. For each iteration of the loop index registers are set up. The input data start address in the FEB is calculated from the demultiplexing parameters in the **X**:VdspFEBmodSa block and stored in **r0**. Similarly the base address of the corresponding pedestal block is calculated and stored in **r4**. The cluster finding threshold for the block is copied from **X**:VdspThreshold to **P**:PmemThresh.

Pedestal subtraction and data checking is performed in the routine Ped_Corr, this is followed by the common mode subtraction in the routine Calc_CM. For each block the calculated common mode is stored to scratch storage by SaveTempCM. Clustering is performed by the routines Cluster and OutClust, which also fill the VHLS bank in the CEB. This completes the loop over the data blocks.

After the block loop the VHLS bank is closed by a call to BOSClose. The VPLH bank is created in the output buffer by the routine Make_VPLH. The common mode values are written to the CEB in the VOCM bank, by the routine Make_VOCM, and the trailer bank by Make_VTRA. Finally the event is finished by closing the output buffer, OutClose (which also asserts a service request), and the FEB, by a call to FEBClose, making it available for a new event. If pedestal updating has been enabled (this is done by externally writing the hex value $BABE to **X**:PUPEnableFlag) the routine UP_Main is called.

ProcessData returns and the program waits for the next event.

2.2.3 Pedestal Subtraction

The routine Ped_Corr expects the first data word to be pointed to by **r0**, and the first pedestal by **r4**. The data are copied from the FEB, anded (**y0** contains the mask), to retain only the lower ten bits of the data word, and copied to the internal DSP memory.

The data are demultiplexed after this operation. The next phase loops over all the data loading each word to the **a** register multiplying it by four (by two right shifts). It is tested for overflow (the overflow test value is kept in **x0**), and the bit 14 is set if it is in overflow. The pedestal is fetched to **b**, and tested against the flag bits 14 and 15 in **x1** to see if the channel is valid. If the channel is bad a flag bit is set in the data word. The data offset contained in **y1**, is added to the data (this assures that the data will always be positive), and pedestal subtracted from it. The data word is then written back to the internal DSP Y memory, **r2** is used as the index register for internal data.

2.2.4 Common Mode Calculation and Subtraction

This is steered by the routine Calc_CM. It expects that the pedestal corrected data are stored in the **Y**: data memory. A histogram method is used to find the CM. The histogram starting point, bin width, and number of bins are specified by the VDSP bank. The number of bins is copied to **r2**, which is used as the loop limit value in all further subroutines.

Make_Hist sets up the histogram. It first clears the internal X memory where the histogram will be constructed. A loop over the data are performed. Each data word is fetched to **a** and tested against **x0** to see if the invalid channel bit is set. Data from invalid channels are not used in the histogram. The value of the lower edge of the first bin kept in **y1**, is subtracted from the data word. If this drives the data negative the first bin, corresponding to underflows is incremented. To determine the bin number corresponding to a data word, it is divided by the number of bins. To make this operation fast, the bin width is required to be a power of two, and the division is accomplished by right shifts of the data word, this shift value is in **y0**. The shifted data word is loaded to **r4** which is used as the index register pointing to the bin. The previous bin content is loaded to **b**, incremented, and written back to memory.

Peak finding in the histogram is done by Find_CM_Peak. As a first approximation a loop is made over the histogram to find the bin with the largest number of entries. For each iteration of the loop the bin content is loaded to **b**. If it is larger than the value in **a**, it is copied to **a** and the bin number stored in **r3**. At the end of the loop this leaves the number of the bin containing the most entries in **r3**, and its content in **a**. To get a better approximation of the peak this first value is refined. The width of the peak around the first approximation of the peak value is determined. Then the true peak value is taken as the point half way between the two half maxima of the first approximation. To determine the FWHM two loops are made, one from the peak channel toward the bottom of the histogram, the second from the peak towards the top. In each loop the value of the bin is loaded to **x0**, and compared with half the peak bin content stored in **a**. After the first loop the address of the lower half height is stored in **x1**, and after the second the address of the second half height in **y1**. The difference of these is taken in **a**, and stored to **x0**. The content of **a** is then divided by two, the lower half height bin from **x1** is added to this to get the bin number of the center of the distribution. This is then multiplied by the bin width in **y0**. The lower edge of the histogram is added to this, giving the peak in the correct units. The FWHM is then calculated by multiplying the number of bins stored in **x0**, by the bin width in **y0**. The peak is returned in **b**, and the FWHM in **a**. In

this routine, consideration must be taken that index registers are not changed solely due to pipelining effects in the DSP.

The CM is subtracted from the data by the routine Sub_CM. This loops over the data, indexed by **r0**, loading it to **a**, subtracting the CM in **b**, and returning the data to memory.

2.2.5 Clustering

The cluster finder and bank forming routines are divided into two parts, the routine Cluster which depends only on DSP internal memory, and the routine OutClust which requires access to both the internal and external memories for the production of the output banks. Dividing the processing into two parts like this allows processing to continue inside the DSP even if the bus switches exclude DSP access to the external memories. This situation arises if a new event is triggered or if a Fastbus access is made.

The cluster finding algorithm is executed in internal DSP-memory (Subroutine Cluster). It accesses the pedestal and common mode corrected raw data which are provided in blocks of 256 channels in the internal DSP Y-memory. A threshold, copied from **P**:PmemThresh into the **y0** register, is applied to the data from single channels, indexed by **r0**, after they are copied into the **a** register. Channels marked "invalid" as defined by the constant ValidData stored in **y1**, are not used for opening a cluster nor for expanding an existing cluster.

If the channel content exceeds the threshold a cluster is opened by storing the address of the hit minus an offset (ClustSize = 7) in the internal DSP X-memory, indexed by **r4**. The cluster end address (= hit address + ClustSize), kept in **x1**, is saved in **P**:PmemClEA. If no new hit is observed before the end address is reached, the cluster is closed by storing the end address next to the start address in X-memory, otherwise the end address is updated in **x1**. Clusters spanning the boundaries of a 256 channel block are treated as follows: If a hit is found near the beginning of a block, the start address will be negative and be used to point into a save area (**X**:XmemSvTop), where the last ClustSize channels of the previous block (should there have been one) are stored. If a hit is found near the end of a block, the cluster is closed at the end of the current block and a new cluster is set up for the remaining channels in the next block.

After clustering each block of 256 channels an editing routine (OutClust) is executed. It must be called immediately after the routine Cluster. It first checks the cluster pointer in **r4** against the value zero loaded in the **a** register, to handle the case of no cluster existing. It then loops over the clusters to construct the bank entries for VHLS. The special case of a cluster overlapping the block boundary is checked and specially handled. The bank entry is obtained by or-ing the start strip number with the appropriate HCODE as fetched from the VDSP bank, one of six values in **X**:VdspFEBmodLb. In the normal case with a fully contained cluster the width is determined by subtracting the start from the end address, where the start address is fetched into **y1**, and the end address into **b**. The start address in **a** is or-ed with the HCODE in **x1**. The **a** register must be then shifted left to load the leftmost six bits into **a2**. **a** is then or-ed with the word count in **b**, and complete word transferred back to **a** where it is written to the VHLS bank by a call to OutWrite. The case where the cluster spans a block is handled in a similar although more complicated

fashion. Pulse heights from clusters spanning into the next block must be stored into a temporary buffer for later processing.

For every cluster the channel contents are copied from internal Y-memory (or from the save area) into the temporary pulse height (TempPH) area. Finally the last ClustSize channels are moved to the save area to prepare handling of overlapping clusters for the processing of the next block.

2.2.6 Pedestal Updating

To compensate for pedestal drifts over time, the data are used to update the pedestals in the Siroccos. Updating is performed after an event is processed and written to the CEB, and is broken off should a new event arrive. The pedestal is updated by a fraction of the difference between the old pedestal and the current channel content. Each time updating is broken off the break off point is stored, and the next time the update routines are called, the updating is resumed at this point. This ensures that not only the first few channels are updated, when events arrive in quick succession.

As part of the initialization the routine UP_init is called. This loads initial values to the update index block. This block is then used to store the addresses where the pedestal update is broken off for each event, and to restore them when the updating begins again.

Updating is performed by the routine UP_Main, called from ProcessData, this is done only when enabled externally, (eg. after VDET high voltage has ramped up).

UP_Main first restores the stored pointers by calling UP_Restore_Pntrs. This loads **r2** with the number of blocks remaining to be looped over, **r3** with the number of channels remaining to be looped over, **r7** with the number of the first block to be processed, and **r6**, **n6**, and **m6** with the first channel to be processed, the channel increment value (8), and **m6** with 2047 to make the **r6** register index a circular buffer used to determine where in the Front End Buffer processing is going on. After return, the block loop is started. With each iteration of the block loop data and pedestal addresses for that block are recalculated. The pedestal is indexed by **r4**, after being loaded with a value determined from the block number in **r7** and the channel number in **r6**. Furthermore the data start address is determined from the contents of **r6**, and the stored value of **X**:FEBstart, containing the starting address of the event. After the addresses are set up, the loop over the data within a block is performed. For each new data word the Front End Buffer is tested for a new event by a call to FEBTest, the loops are broken off if there is a new event. If a channel is to be updated, the data word is fetched to **a** where it is first masked by the **y0**, to extract the 10 bit ADC value, then multiplied by four by a double right shift, and tested, by the value in **x0**, to see if it is in overflow. The pedestal is fetched to **y1**. If the data are valid the routine UP_calculate_new_peds is called to change the pedestal value. The difference between the old pedestal and the data word is taken in **a**, and this is divided by 32 by shifting it left four times. Several checks are then performed on this value. If its absolute value is too large (greater than the value MaxChange in **y0**), the pedestal is updated by the the value of MaxChange. In this case the code checks the sign of the change and sets it to the correct value. This procedure prevents the pedestal from being changed by a very large one event aberration in pulse height, such as that from a true hit. If the channel has

Bit Field	Function
0–11	Pedestal
13	Not used
14	Data Invalid as flagged in Database.
15	Data Invalid as can be flagged from pedestal program

Table 2: Structure of word in the VOPD bank.

no invalidity bits set it is then updated by adding the correction factor to it in b. The updated pedestal is then returned to memory.

After all channels have been processed, or if a new event arrives and the loops are broken off, the routine UP_Save_Pntrs is called to save the counter registers to the update pointer save block. This allows the updating to be resumed after the next event, at the point where it was broken off.

3 The Calibration Program

The calibration program is used to produce a pedestal and an rms noise value for each channel on a module. The pedestal can then be used in the event processing. The noise values may be used to test for defective channels. A pedestal run is made using the Sirocco hardware in the same way as the event readout. Events are triggered externally (usually about two-hundred of them), after each event is processed the DSP program writes a short trailer bank in the CEB, and signals a service request, allowing the event builder program to synchronize the system. After enough events have been collected, the event builder sets a flag in the DSP memory, signaling the DSP to calculate the the pedestals and noises. Finally when the processing is complete, the DSP sets another service request to inform the event builder that the data may be read out.

3.1 The Algorithm

The calculation of the pedestal is done by summing the pulse heights for each channel from all the events and dividing the sum by the number of events taken:

$$Ped_i = \frac{1}{N}\sum_{j=1}^{N} 4ADC_{ij},$$

where i indexes the channels, and j the events. The ADC content is multiplied by four to preserve accuracy in further operations. The structure of the final pedestal word is give in table 2.

The calculation of the rms is a bit more complicated. The quantity calculated is not simply the rms, but really the common mode noise subtracted rms. The desired quantity is the true noise of the channel without the component that swings from event to event

affecting all channels equally. The common mode should have no effect on the pedestal because in the course of 200 events it is averaged out. The common mode noise for each event is calculated according to:

$$CM_j = \frac{1}{M_{good}} \sum_{i=1}^{M_{good}} 4ADC_{ij}\delta_{igood}.$$

Here M is the block of channels over which the CM noise is calculated, this is done on a line driver sized block basis of 256 channels. Channels not in the bonding maps, or otherwise labeled as unusable are not used in the calculation, as indicated by the δ-function. The variance for each channel is then calculated as

$$var_i = \frac{1}{N}\sum_{j=i}^{N}\left((4ADC_{ij} - CM_j) - \frac{1}{N}\sum_{k=i}^{N}(4ADC_{ik} - CM_k)\right)^2$$

The variance is calculated for each channel using the ADC value for each event but first subtracting that event's common mode noise contribution.

It should be noted that there are several complications in the implementation of these formulae. The DSP is designed for floating point arithmetic, this is manifested in the way numbers are normalized after arithmetic operations. The program here works only with integers, requiring care after multiply and and divide operations. The second limitation is that the DSP has only only 24 bit wide data paths, this gives insufficient precision for the large values of sums of squares to be calculated in the equations above. This implies that the sums be transferred to and from memory as upper and lower half words of a 48 bit full data word.

3.2 The Program

The program is divided into three parts, initialization: performed after downloading and booting the DSP, event processing: controlled by the routine Ped_Loop, and post-processing: the routine Ped_calculations that produces the final numbers after all events are collected. The main routine is PEDnn.asm (nn is the version number).

3.2.1 Initialization

The processor initialization for the pedestal program is identical to that for the readout program. The program is loaded and copied into internal memory, and the hardware set to use that memory. Data transfer between the front and crate end buffers and the DSP is done by the FEB and CEB handler packages. The main program (section Main) is called after the boot sequence, and the FEB and CEB initializations. It then calls the routine Ped_before_loop before jumping into the event handling routine Ped_Loop.

Ped_before_loop clears the event counter **X**:Number, and clears the memory areas used for scratch storage by calling the routine MEMORYclear.

Block Name	Content
Ped_Block_r2	Sum of CM subtracted data.
Ped_Block_r3	Sum of squares CM subtracted data. (high word).
Ped_Block_r3n3	Sum of squares CM subtracted data. (low word).
Ped_Block_r4	Not currently used.
Ped_Block_r5	Not currently used.
Ped_Block_r6	Sum of all data per channel.
Ped_Block_r7	Number of entries in each channel.
Result_noise_r2n2 (VONS)	Final result of the noise calculation.
Result_ped_r4n4 (VOPD)	Pedestal (or'ed with status bits).

Table 3: Data blocks for the pedestal program. The sums are all sums over all events for a channel.

3.2.2 Event Processing

The control routine for event processing is Ped_Loop. It is similar in structure to the event loop processing routine of the datataking program. The event loop begins with a reset of the run flag, and of the Crate End Buffer wait flag. The routine enters the wait event loop, this differs from the wait event loop for the event readout program in that, on each iteration two conditions are checked: The FEB is tested for the presence of an event indicated by having the carry flag set after a call to FEBOpen. A jump is made to ProcessEvent if a new event has arrived. In addition the location **X**:PedDoneFlag is compared against the hex number $FEED, in the **b** register to see if the event builder is indicating that all events have been triggered. If the last triggered has been received the post-processing is performed by the routine Ped_calculations, after which a trailer bank is created by a call to Make_VPTR, and a service request is sent by calling OutClose.

After the receipt of a trigger, the run flag is set by clearing the RunFlag bit in the register **X**:PCD, the status block is updated, by incrementing the event number, **X**:EventNum, and the event size is set to zero by clearing **X**:EVsize. Space in the CEB for for the trailer is opened by a call to OutBook. The routine Ped_eventloop is called, this is the routine that does all the work. After the event is processed, Make_VPTR is called to create the trailer, and a service request is sent by a call to OutClose. The routine then returns to waiting for the next event.

The trailer bank routine Make_VPTR creates a three word trailer in the CEB for each event. This consists of the event size, the internal event count and the account number from the FEQ.

The event processing routine Ped_eventloop begins by initializing the index registers to point to the blocks of memory used for accumulating the running sums. The blocks are described in tables 3 and 4. The contents of **r0**, which has been set to the start of the event in the FEB, by FEBOpen, is copied to **X**:Pointersav. The number of events is then incremented and stored in **X**:Number.

The routine Reset_r0_r5_pointer is then called. This routine sets up the pointers for the

demultiplexing of the eight different data blocks ($t_1, t_2, \phi_1, \phi_2, z_1, z_2, z_3, z_4$). The location **X**:VdspTablePointer is which contains the block number (0–7) is cleared, and the location of the t_1 block is extracted from the **X**:VdspFEBmodSa, and added to the event start address in FEB to give the pointer to the start of the first block in **r0**. The **r5** pointer is initialized, and the demultiplexing step constant (equal to 8) is loaded to **n0**, and **n5**, before returning to Ped_eventloop.

Ped_eventloop loops over the 8 data blocks, for each block the routine Reset_r0_r5_toBlockBeg is called. This calculates the start point for the block in the FEB, and loads it to the **r0** index register. The routine Update_good_ch_rn is called, this performs a loop over the pedestal bank and counts the number of valid channels, i.e. those that do not have a flag set. This is stored to the location **X**:Good_ch_rn_save.

Next a loop over the channels in the block is performed to calculate the mean pulse height for this block for this event. Each data word indexed by the sum of **r0** and **n0** is fetched from the FEB. The data are masked with **x0** to extract the ADC part of the word into **b**. This value is then added to the content of **X**:Sumplace fetched to **a**, and the sum is returned to **X**:Sumplace. Various manipulations are also made with the **r5** block which does not concern us here. The number of channels is loaded to **x0**, the sum of channels from **X**:Sumplace to **a0**, and the routine DividI called to return the mean of the block to **a0**. This is then copied to **X**:Mean.

The common mode of the block is calculated by the routine Calculate_common_mode. The routine loops simultaneously over the data and the downloaded old pedestals in the pedestal bank. The old pedestal indexed by **r4** and **n4** is loaded to **a1**, where it is checked against the dead channel flag mask. If the dead channel flag has been set, the channel is not used. Otherwise the data word is masked to extract the ADC value, and this is summed in the b register. After the data have been summed, **b** is transferred to **a**, the number of good channels in the block loaded from **X**:Good_ch_rn_save to **x0**, and the DividI routine called. The common mode is then copied to **X**:Common_mode.

After these manipulations the main channel processing loop is performed, the pointers are reset by a call to Reset_r0_r5_toBlockbeg. The raw data word is fetched to **a1**, and masked to get the ADC content in **a**. This is added to the running sum of ADC counts for each channel as indexed by **r6**. The common mode is restored to **x0**, from **X**:Common_mode and subtracted from the ADC value. The result is copied to **a**, **x0**, and **y0**. **x0** and **y0** are multiplied to get the square of the common mode subtracted pulse height into **a**, and the result is copied to the **x0**, **x1** register pair. The common mode subtracted pulse height is again calculated from its constituents, to **a**, and copied to **y0**. Leaving in **x0** and **x1** the square, and in **y0** the value of the CM subtracted pulse height. The old channel by channel sum of the squares is loaded from the X memory indexed by **r3**, and **r3**+**n3**, to **a0** and **a1**. The newly calculated square is copied to **b0** **b1** added, and then copied back to the X memory. The CM subtracted value sum is copied from **X**:(**r2**) to **b**, and the value for this channel added to it before being loaded back to the memory. The number of entries for this channel is incremented in the block of memory indexed by **r7**.

At the end of loop the pointers to all of these memory blocks are incremented as necessary, and outside of this loop the pointers updated for the next block by a call

to Setup_r0_r5_to_next_block. This completes the event processing, and the subroutine returns.

3.2.3 Post-Processing

Post event collection processing is performed by the routine Ped_calculations. The sums assembled during event by event processing are used to determine the final pedestal and noise quantities. The routine begins by initializing the index registers as for event processing. A loop is made over all the data blocks, and then through the channels in each block.

The first calculation determines the pedestal. The sum of all the ADC counts for all the events in each channel is fetched from **X**:(**r6**) to **a0**, and shifted left twice in order to multiply by four. The number of collected events is copied from **X**:Number to **x0**, and a division performed to get the pedestal. The old pedestal copied from **X**:(**r4**) to **b** and checked for the presence of the dead channel flag. If this is set, then the flag is then set in the newly calculated pedestal word. The completed pedestal word is then returned to the X memory.

The pedestal value also in **x1** is then checked against the value **X**:Null_cut_rn. If it is less than this value, the sick channel flag is also set in the pedestal word, by the routine Set_dead_channel. This routine copies the word pointed to by **X**:(**r4**+**n4**) and sets the bit specified by Set_dead_ch_mask before restoring the word back to memory.

The noise is now calculated for the channel. The sum of the CM subtracted pulse heights is fetched from the X-memory location pointed to by **r2**, to **y0**. The value is squared, with the result appearing in the **a** register. The number of entries for this channel is copied from **X**:(**r7**) to **x0**, and the the two numbers divided, with the result left in **a**. The sum of the squares of the CM-subtracted pulse heights is loaded from **X**:(**r3**), and **X**:(**r3**+**n3**) to **b**, and subtracted from **a**. This is then divided by one less than the number of entries. Finally the square root of this value of is calculated. Notice that much magic (some of it black) happens in the divide and square-root code. The rms value is then copied back to **X**:(**r2**+**n2**).

After theses quantities have been calculated the code makes it possible to flag channels based on values outside preset limits. The noise is compared against the value in **X**:Raw_nse_cut and the channel flagged if its noise is greater than this value.

The pointers are then set to deal with the next channel, and the loop is terminated. Outside the channel loop, a call is made to Setup_r0_r5_to_next_block to set the pointers for the next data block.

This completes the pedestal processing, and the completed blocks are available for uploading. To inform the event builder that processing is complete, a trailer bank is created and a service request set.

Appendix: Allocation of the X: Memory

Address Range		Function
$0000	$00ff	CM histogram, Clustering algorithms
$0100	$01ff	Free
$0200	$07ff	Temporary Pulse Heights
$0800	$0fff	Ped_Block_r2 – PEDS only
$1000	$17ff	Ped_Block_r3 – PEDS only
$1800	$1fff	Ped_Block_r3n3 – PEDS only
$2000	$27ff	Ped_Block_r4 – PEDS only
$2800	$2fff	Ped_Block_r5 – PEDS only
$3000	$37ff	Ped_Block_r6 – PEDS only
$3800	$3fff	Ped_Block_r7 – PEDS only
$4000	$5fff	CEB
$6000	$60ff	Free
$6100	$6160	Fatalbank
$6161	$6eff	Free
$6f00	$76ff	VONS (Result_noise_r2n2 for PEDS)
$7700	$76ff	VOPD (Result_ped_r4n4 for PEDS)
$7f00	$7f9f	Free
$7fa0	$7faf	Xmem – See below
$7fb0	$7fcf	Free
$7fd0	$7fef	VDSP – See below
$7ff0	$7ff5	DSPCONT – See below
$7ff6	$7ffd	DSPSTAT – See below
$7ffe		CEBWrite
$7fff		CEBRead

Table 4: Allocated Blocks of the **X**:memory.

Address Range		Function
$7fa0		XmemBosTop
$7fa1	$7fa7	XmemSvBot
$7fa8	$7fae	XmemSvTop
$7faf	$7fbe	XmemTempCM Common Mode storage

Table 5: XMEM Block.

Address Range		Function
$7fd0		VdspVersion
$7fd1		VdspNDSP
$7fd2		VdspDACValue
$7fd3		VdspOverflow
$7fd4	$7fdb	VdspFEBmodLb
$7fdc	$7fe3	VdspFEBmodSa
$7fe4	$7feb	VdspThreshold
$7fec		VdspCMHistMin
$7fed		VdspCMHistSize
$7fee		VdspCMHistBinS
$7fef		VdspCMHistBinB

Table 6: VDSP Block.

Address Range		Function
$7ff0		PedDoneFlag
$7ff1		Affe flag
$7ff2	$7ff5	Free

Table 7: DSPCONT Block.

Address	Function
$7ff6	NumClus
$7ff7	NumStrip
$7ff8	EventNum
$7ff9	Accnt
$7ffa	EVsize
$7ffb	FEBStart
$7ffc	TOPhit
$7ffd	Prstat

Table 8: DSPSTAT Block.

References

[1] H.G. Moser *et al Upgrade of the Data Aquisition Hardware for the ALEPH Minivertex Detector*
ALEPH 90-159, MINIV 90-015 (October 1990).

[2] Nils Bingefors and Mike Burns *Sirocco IV - Hardware and Software Manual*
DELPHI 88-48 Track 48 (July 1988).

[3] Motorola Inc. *DSP56000/DSP56001 Digital Signal Processor User's Manual*
DSP56000UM/AD (1990).