Data Analysis Techniques for Triggerless Data Streams in Nuclear Structure Physics

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy

by

Peter Jonathon Christopher Ikin

August 2005

Abstract

The GREAT spectrometer is a focal plane detector system recently installed in Jyväskylä, Finland. The spectrometer is used to study various topics of interest in nuclear structure physics, in particular the area of superheavy elements.

One feature of the GREAT spectrometer is the triggerless total data readout (TDR) data acquisition system, the unique aspect of which is a lack of a system wide hardware trigger. This is intended to directly compensate for dead time problems of past systems. The TDR data acquisition system reads out the state of each detector independently of any other in the spectrometer when an event occurs. These events are then time-stamped from a central system clock, producing a single stream of time ordered data.

A major challenge, and the primary focus of this thesis is to reconstruct information about time coincident events, and the sequence of events that have been lost by reading out the data as a single stream. The thesis discusses the data analysis techniques used to extract physics information from the time ordered data stream and uses well known examples to demonstrate that the techniques discussed can be applied to real world problems.

Contents

1 Introduction **4**

2 Experimental and Detector Details **10**

2.1 Overview 10

2.2 Experimental Focus 10

2.3 RITU and GREAT 12

2.4 GREAT Spectrometer 13

2.4.1 Multiwire Proportional Counter (MWPC) 15

2.4.2 Cooling Block 16

2.4.3 Double Sided Silicon Strip Detector (DSSD) 17

2.4.4 PIN Diodes 18

2.4.5 Planar Germanium Detector 19

2.4.6 Segmented Clover Detector 19

2.4.7 Silicon-Gas Tac 20

2.4.8 Target Position Arrays 20

2.5 Example Data Set 21

2.6 Summary 23

3 TDR Data Acquisition System and Data Format **24**

3.1 Overview 24

3.2 TDR Electronics System 24

3.3 TDR Data Format 27

3.3.1 Data Items 27

3.3.2 Info Items 28

3.3.3 Data Stream 29

3.3.4 Block Structure 30
3.4 Summary 32

4 Code Architecture and Data Buffering 33
4.1 Overview 33
4.2 High level Code Structure 33
4.2.1 Logical Code Structure 34
4.3 UML Diagrams 35
4.3.1 Class Diagrams 36
4.3.2 TDRSorter Class Diagram 41
4.4 Data Buffering 43
4.4.1 Buffering Methods 44
4.4.2 Time Buffer Operation 47
4.4.3 Buffer Operation Walk Through 53
4.4.4 Example Time Buffer 56
4.5 Summary 57

5 Pixel Definition and Event Construction 58
5.1 Overview 58
5.2 Pixel Definition 60
5.2.1 Time and Energy Condition Statistics 61
5.2.2 Triggering Considerations 65
5.3 Pixel Definition Examples and Problems 68
5.4 Problem Conditions in Pixel Definition 70
5.4.1 Multiple X and Y Strips 70
5.4.2 Double Counting 74
5.4.3 Pixel Construction from Alternate Triggering 77
5.5 Event Packaging 79
5.6 Summary 83

6 Specialised Sorting 85
6.1 Overview 85
6.2 Deriving A Specialised Sorting Class 85
6.3 Event Properties and Data Visualisation 87

6.3.1 Calibration 88
6.3.2 Histograming 89
6.3.3 Basic Spectra 90
6.3.4 Examples of Recoil and Alpha Identification 92
6.4 Tagging 94
6.4.1 The **CTagger** Class 95
6.4.2 **CTagger** Depth Mechanism 96
6.4.3 The Tagger Manager 99
6.4.4 Delayed Coincidences using the **CTagger** Class 101
6.4.5 Recoil Alpha Tagging 102
6.5 Tagger Search Strategies 105
6.6 Lifetime Calculations 110
6.7 Comparison of TDRSorter to GRAIN 112
6.8 Summary 117

7 Conclusions **119**

A TDRSorter Data Analysis Package Operation **125**
A.1 Overview 125
A.2 Requirements 125
A.3 Sort File Creation 126
A.4 Compilation with g++ on linux platform 127
A.5 Compilation with Visual Studio on the Windows Platform . . 128
A.6 Usage 129

B TDRSorter Class Overview **130**
B.1 Classes 130

C Detailed TDRSorter UML Diagram **135**

D TDRSorter Code CD **136**

Chapter 1

Introduction

Nuclear physics experiments collect a lot of data. A typical experimental set-up can have anywhere between ten and five hundred detector channels. In order to make sense of the large quantity of data generated by these systems some sort of triggering mechanism is needed to read out the state of the detectors at appropriate points in the experimental run.

In general past data acquisition systems operate based on the principle of common dead time. One detector group in the system, usually the implantation detector[1], is considered the trigger. When a given reaction or sequence of reactions result in the trigger being activated, the data acquisition is paused whilst the whole state of the system is read out. This set-up introduces dead time into the process of data acquisition that can be especially significant in experiments with a high reaction rate were the rate of incident beam is increased to compensate for the low efficiency of the decay process being studied and the reaction channel of interest is weak.

Another limitation of past data acquisition systems was the fact that the nature of the hardware set-up forced the user to define the experimental conditions in advance. This imposed few limits on traditional in beam γ ray spectroscopy as all the information on the reaction being studied is available

[1]The implantation detector is the component of the spectrometer where the recoiling nuclei being studied initially embed.

in a short time period ($< 1\mu s$). However the situation was not so good for tagging experiments where parts of the information was not available until long after other parts of the reaction sequence. This limitation can be partly mitigated by using hardware delays to postpone information delivery to the data acquisition system. For example, for experiments triggering from recoil implantation, prompt decays at the target position need to be delayed by the flight time (of the order of μs) of the recoil through the the recoil separator (see section 2.3 for details on recoil separators) if they are to be associated correctly.

Again in this case the experimental conditions need to be defined in advance to select the right delay times to allow the data acquisition system to correlate the delayed information to the triggering event. A further problem arises due to hardware limits in the amount of time a signal can be delayed for. If the required delay time exceeds these limits then no data can be gathered from these longer lived events.

Returning to the data loss due to the common dead time in the data acquisition system it is useful to refer to figure 1.1. The middle line on the figure represents a time ordered sequence of data items from the focal plane silicon implant detector. In the GREAT spectrometer the focal plane silicon implant detector is the Double-sided Silicon Strip Detector (DSSD) (see section 2.4 for details). Each of the boxes on this line represents a triggerable event. When the first silicon event on the line triggers, the data acquisition system goes dead for the amount of time indicated by the box on the bottom line. During this dead time no other silicon events can trigger the data acquisition system and cause the state of the detectors to be read out.

When this first event triggers the first three γ rays on the target γ line fall within the trigger gate and are associated with the triggering silicon event. This sequence of events is the ideal circumstance for the data acquisition system to be in i.e. there are no overlapping events within the dead time

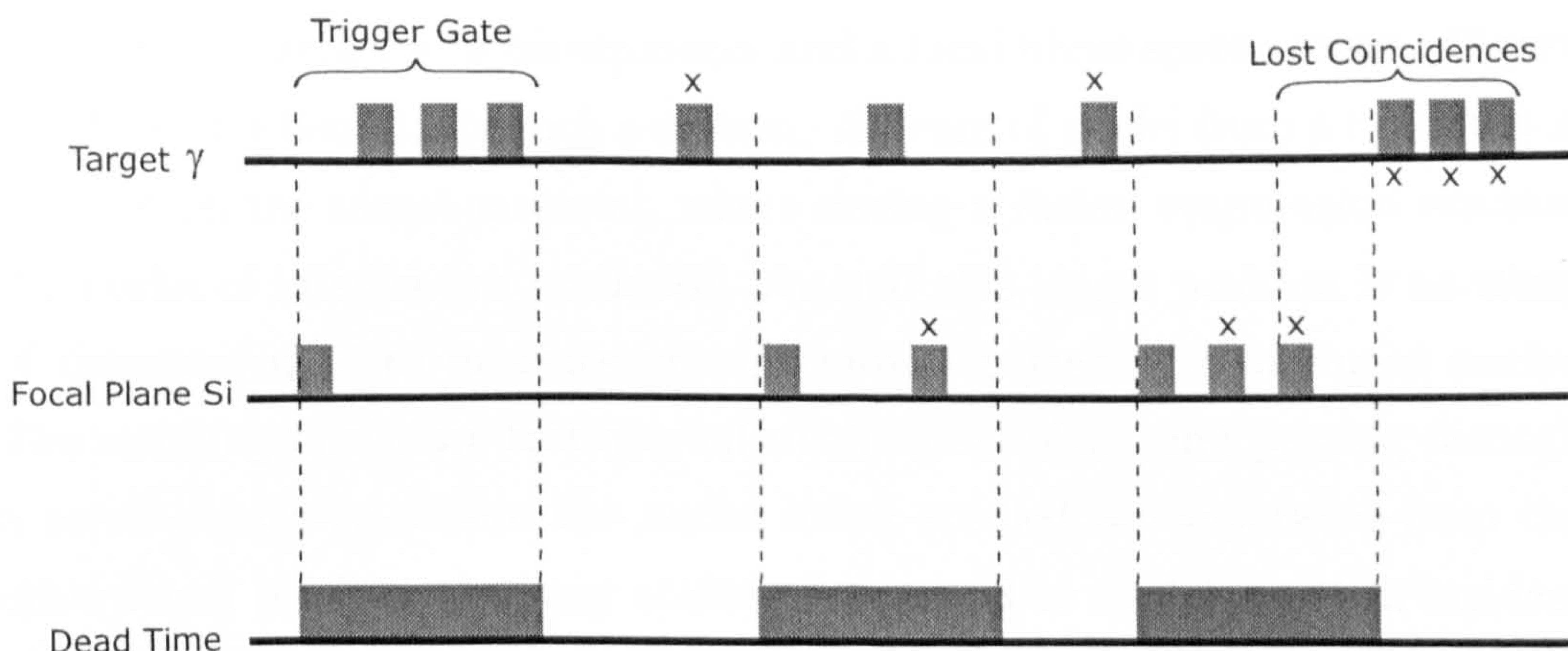


Figure 1.1: Schematic demonstrating how a data acquisition system's triggering conditions can lead to associated dead time. The central line represents a time line with triggers present on it. Any items that are marked with an 'x' are lost due to the dead time shown on the bottom time line.

associated with that trigger. Moving onto the second triggerable event on the focal plane silicon line it can see that this is not the case. This second event triggers and is then associated with the γ ray on the target γ line that falls within the trigger gate. Unfortunately whilst the data acquisition system is dead another triggerable event occurs. This event does not trigger the data acquisition system and this event is simply lost.

The next event on the focal plane silicon line that triggers shows a situation where data other than that on the triggering line can get lost. The first event in this sequence triggers and the data acquisition system goes dead; in this case no gamma rays are in the trigger gate. The two following focal plane triggers occur within the dead time and are lost. The final triggerable event in this sequence if it were triggered would have found several γ rays in coincidence. This shows that using such a common dead time strategy large amounts of good data can be lost.

Past detector systems that have been used with these common dead time data acquisition systems generally consist of three main parts; a target posi-

tion detector array; a recoil separator and a focal plane spectrometer. Figure 1.2 shows a schematic of such a system. A beam of nuclei from a cyclotron is incident on the target material, where during a fusion evaporation reaction the nuclei of interest are produced. Around this target position is an array of detectors used to measure prompt radiation from the produced nuclei. The recoiling nuclei are transported to the focal plane first passing through a recoil separator, where the nuclei being studied are separated from the other recoil products and any scattered beam. The nuclei are then incident on the focal plane spectrometer which usually consists of a position sensitive implantation detector surrounded by arrays of detectors designed to measure specific decay products of the implanted nuclei.

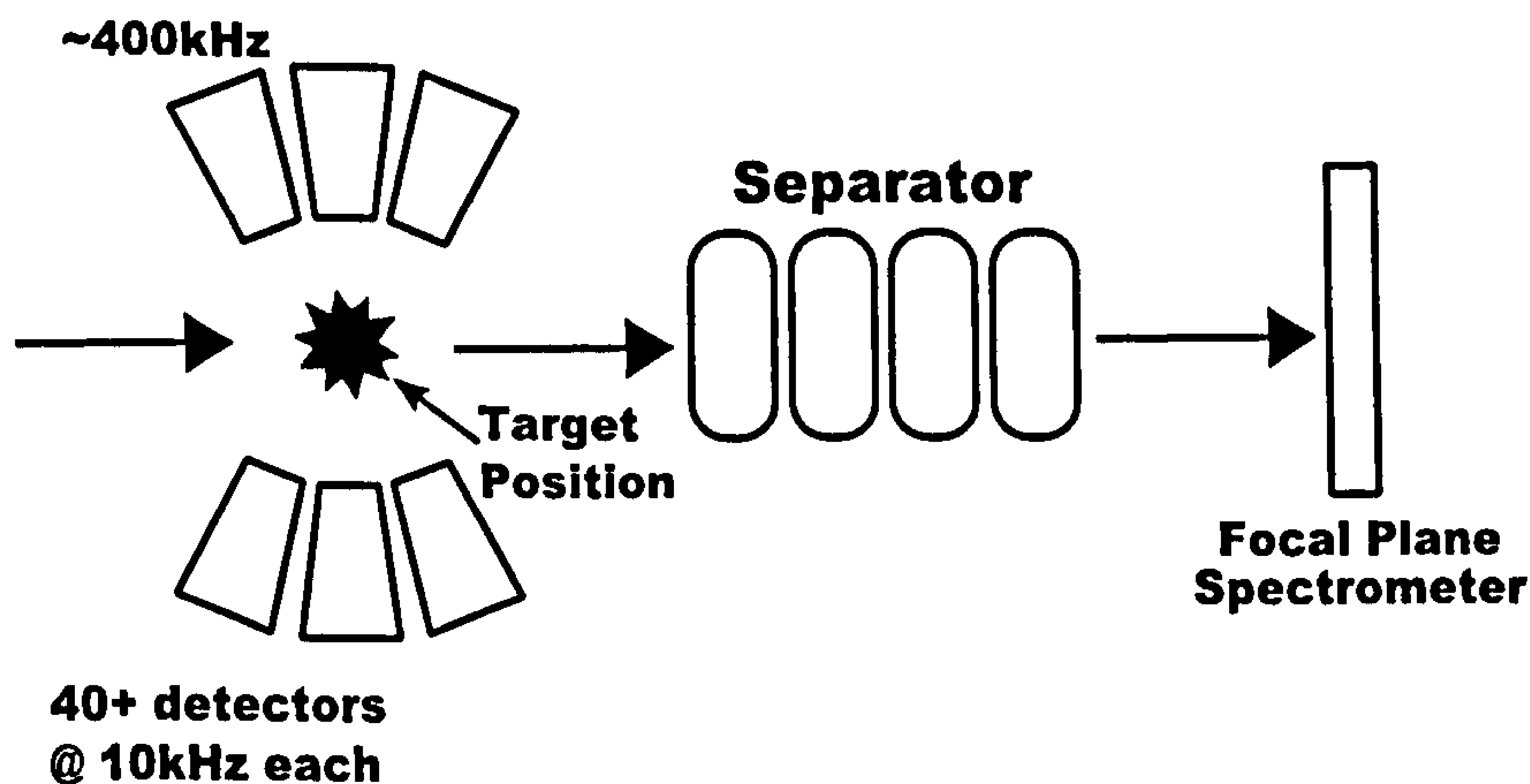


Figure 1.2: Diagram showing a generic detector system that consists of three main parts; a target position array; a recoil separator and a focal plane spectrometer. (The components in the figure are situated relative to each other in the configuration that most detector systems use.)

Considering the generic detector system in figure 1.2 it can clearly be seen how the magnitude of the problem of dead time is compounded when simple tagging is performed. Presuming that the target position array consists of 40 detectors operating at 10kHz each this then equates to a target position count rate of approximately 400kHz. Due to the high efficiency of the recoil separator, the focal plane spectrometer can expect a count rate of around

10Hz. Presuming a common dead time of $10\mu s$ at the target position would lead to a 40% dead time at the focal plane.

Given the above figures it can be presumed that there is an approximate 60% probability of detecting a recoil implanting in the focal plane detector array. By the same token there is a 60% chance of detecting any alpha decay of a previously implanted recoil. This gives us an overall 36% chance of detecting and correlating a recoil alpha pair. It is obvious that the close coupling of all the detectors in the system leads to a large potential loss of data due to the significant dead time in the system. Given the low statistics of superheavy nuclei experiments this loss of data is very significant.

The GREAT (Gamma Recoil Electron Alpha Tagging) [1] spectrometer and TDR (Total Data Readout) [2] data acquisition system was developed both as a 'step up' in sensitivity and also as a means of circumventing the problems of the inherent dead time built in to previous detector and data acquisition systems. The GREAT spectrometer itself was designed to meet the characteristics of nuclear reactions and is discussed in detail in chapter 2. The TDR data acquisition system is a triggerless data acquisition system that is designed to virtually eliminate dead time by decoupling all the detector channels from the system wide trigger. Signals from each detector are read out independently from other channels. The only dead time left in the system is for the period of time that the channel is being read out and this is only applicable to that specific channel.

One left over limitation of the TDR data acquisition system is due to this remaining dead time in the individual detector channels. Fast sequences of events within a single detector channel for example an implantation in a focal plane detector channel followed by a prompt decay would still be lost if the lifetime of the decay was less than dead time associated with the shaping time of the amplification process ($\approx 1 - 3\mu s$). This limitation imposed by the speed of the operating hardware can not be avoided; the only way of mitigating the situation would be to change the data acquisition system to

use digital electronics[2] were the system is independent of the shaping time of the linear amplifiers.

This decoupling of the detectors, although addressing the issues of dead time, introduces a number of other problems as a side effect. The main problem that is presented by such a triggerless data acquisition system is that the signals from the individual detectors are no longer associated. There are no events indicating spatial and temporal coincidences constructed during the data acquisition process as was the case in previous systems.

The bulk of this thesis describes the TDRSorter data analysis code. The analysis code has an essential function to play in using the raw data stream supplied by the TDR data acquisition system and constructing usable physics events. The TDRSorter code takes the raw data stream, sorts it and performs prompt and delayed coincidences. From this the TDRSorter code produces various visualisations in the form of histograms. It is from these visualisations that the physical interpretation of the results can begin.

One of the important functions that is performed by the data acquisition system is the time ordering process. The time ordering process sets an important foundation that the data analysis code is built upon. Essentially this process sorts all the data generated by the detectors into time order. A more detailed discussion of the data acquisition process is given in chapter 3. One of the key challenges that must be overcome by the data analysis code is to take this triggerless, time ordered data stream and reconstruct physics events from it.

[2]As opposed to the current analog based system of acquisition components.

Chapter 2

Experimental and Detector Details

2.1 Overview

This chapter will discuss the experimental motivation as well as the overall structure of the GREAT (Gamma Recoil Electron Alpha Tagging)[1] spectrometer. Further sections will go into some detail about the function and operation of the individual detector components that make up the whole spectrometer. A brief discussion will also be given as to the experimental details of the data set used for all the physics examples depicted in subsequent chapters.

2.2 Experimental Focus

The experimental and data analysis techniques discussed in this and the following chapters apply to systems developed for the study of nuclei far from stability, for example the study of nuclei with high mass and charge i.e. super-heavy elements. One question that has been a focus of research in this area is whether an island of stability exists for nuclei with $Z > 100$. These very heavy nuclei should be unbound against fission. Simple calculations using various parameterisations of the liquid drop model predict that the

limit of stability should occur when the Coulomb repulsion between protons overcomes the attraction due to the strong nuclear force i.e. for nuclei with around $Z = 100$ to $Z = 106$.

The fact that these nuclei exhibit stability is due solely to microscopic shell effects. These shell effects are due primarily to what is called the spin-orbit interaction which occurs between the orbital angular momentum and the intrinsic spin angular momentum of the individual nucleons in an given nuclei. A primary focus of research is to describe this effect by extrapolating the well known mean field for well studied nuclei of around $Z = 92$ up to nuclei with larger masses [3][4]. Performing spectroscopy on these super-heavy elements to gain information about their excited states helps to constrain theoretical model parameters and hence improve understanding of these nuclei at the edge of stability.

A major hurdle to overcome is the difficulty in producing the super heavy elements of interest. Two main approaches have been used to produce elements with $Z = 112$ to $Z = 116$. Nuclei with $Z = 112$ have been produced by using beams of medium mass ions impinging on stable Pb and Bi targets[5]. Elements with $Z = 114$[6] and $Z = 116$[7] have been produced with an alternative method of using beams of lighter ions, in particular ^{48}Ca, on radioactive actinide targets. Both methods have a disadvantage in that they produce fairly neutron deficient nuclei. This limitation can be countered by using neutron rich radioactive beams and neutron rich radioactive targets, details of current work in this area are given in [8].

The approaches mentioned above use fusion evaporation reactions whose primary component to the total cross section of the reaction is fission, leaving only a small part to the fusion channels. The fission products decay via prompt gamma ray emission that masks the weak decays of the fusion products of interest. In order to isolate these transitions a selective way of distinguishing these channels is needed. By using recoil separators and suitably sensitive focal plane detector systems a technique known as Recoil

Decay Tagging (RDT) can be used [9][10]. Details of a recoil separator and focal plane spectrometer at the current forefront of research in the area of super-heavy elements is provided in the following sections.

2.3 RITU and GREAT

The GREAT spectrometer [1] is a focal plane spectrometer that is currently in situation at the RITU [11] gas filled separator in Jyväskylä Finland. Recoiling nuclei produced at the target position by fusion evaporation reactions are transported through the separator where the primary heavy ion beam and fission products are filtered from the fusion products of interest.

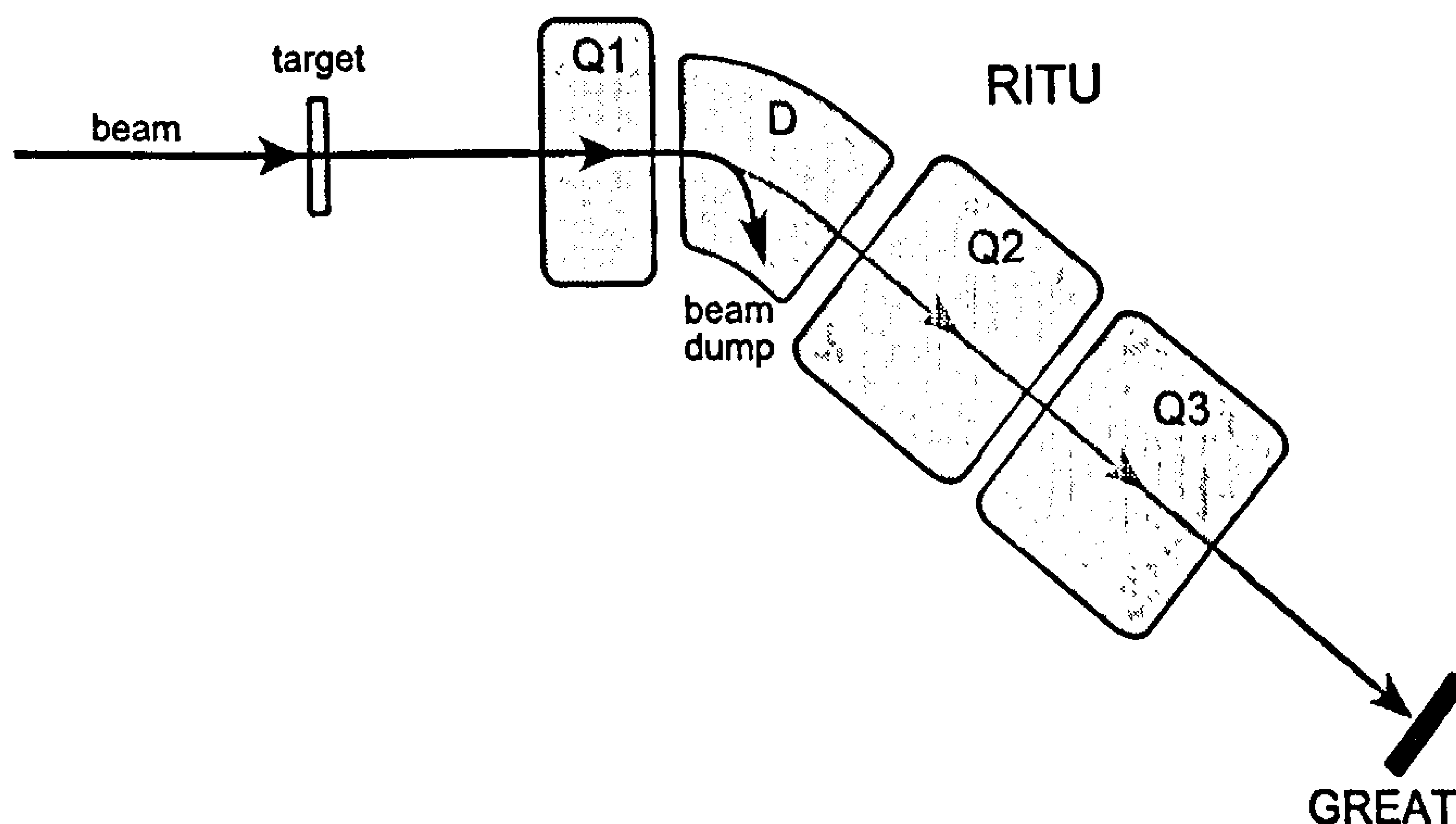


Figure 2.1: Diagram showing RITU and GREAT and their relative positioning. RITU is a recoil separator that consists of four magnets, one dipole magnet for separation and three quadrupole magnets for focusing of the recoil products.

Differences in magnetic rigidity between the fusion, fission and primary beam allow these reaction products to be separated in flight by using a dipole magnet. After passing through the target the reaction products have a wide range of charge states, which after separation results in a wide spread of nuclei incident onto the focal plane. In order to improve focusing of the

recoil products this spread of charge states needs to be evened out. In order to do this a gas filled region is used within the separator's volume. Heavy ions passing into this region undergo many atomic collisions causing the charge state of the ions to change rapidly which has the result of causing the ions to follow an average trajectory through the separator according to the average charge state of the recoil products.

RITU consists of four magnets, three of these are quadrupole magnets and are used for focusing. These are indicated on figure 2.1 by $Q1$,$Q2$ and $Q3$. The fourth magnet is a dipole magnet and is used for separation and is indicated by the D on the figure. Both the beam line and focal plane are kept at a vacuum which is separated from the $1mbar$ of helium gas within the separator by thin windows that allow the transmission of the recoil products. After passing through the separator these products then enter the GREAT focal plane spectrometer.

2.4 GREAT Spectrometer

GREAT is situated on the end of the RITU gas filled separator. Given the set-up of RITU and GREAT as indicated in figure 2.1 it is useful to describe the individual components of the target position detector array. The GREAT focal plane spectrometer consists of five main parts.

- A multi wire proportional counter (MWPC).
- An array of 28 silicon PIN diodes.
- A double sided silicon strip implantation detector (DSSD).
- A double sided planar germanium detector.
- A high efficiency segmented germanium clover detector.

Figure 2.2 shows a schematic diagram of the layout of the above mentioned components and their relative positioning to each other and also within the

whole spectrometer. Referring to the figure it can be seen how the detectors in the spectrometer are arranged around the expected sequence of events produced from the reactions being studied. Recoil products leaving the exit window of the RITU gas filled separator pass through the MWPC and are subsequently embedded in the double sided silicon strip implantation detector. The embedded recoils then decay and the particles and radiation they emit are detected either within the DSSD (i.e the particle or radiation does not escape or pass out of the material making up the detector) or the surrounding detectors in the rest of the spectrometer. The subsequent sections will describe the main purpose of each individual detector component and also give details of how they function.

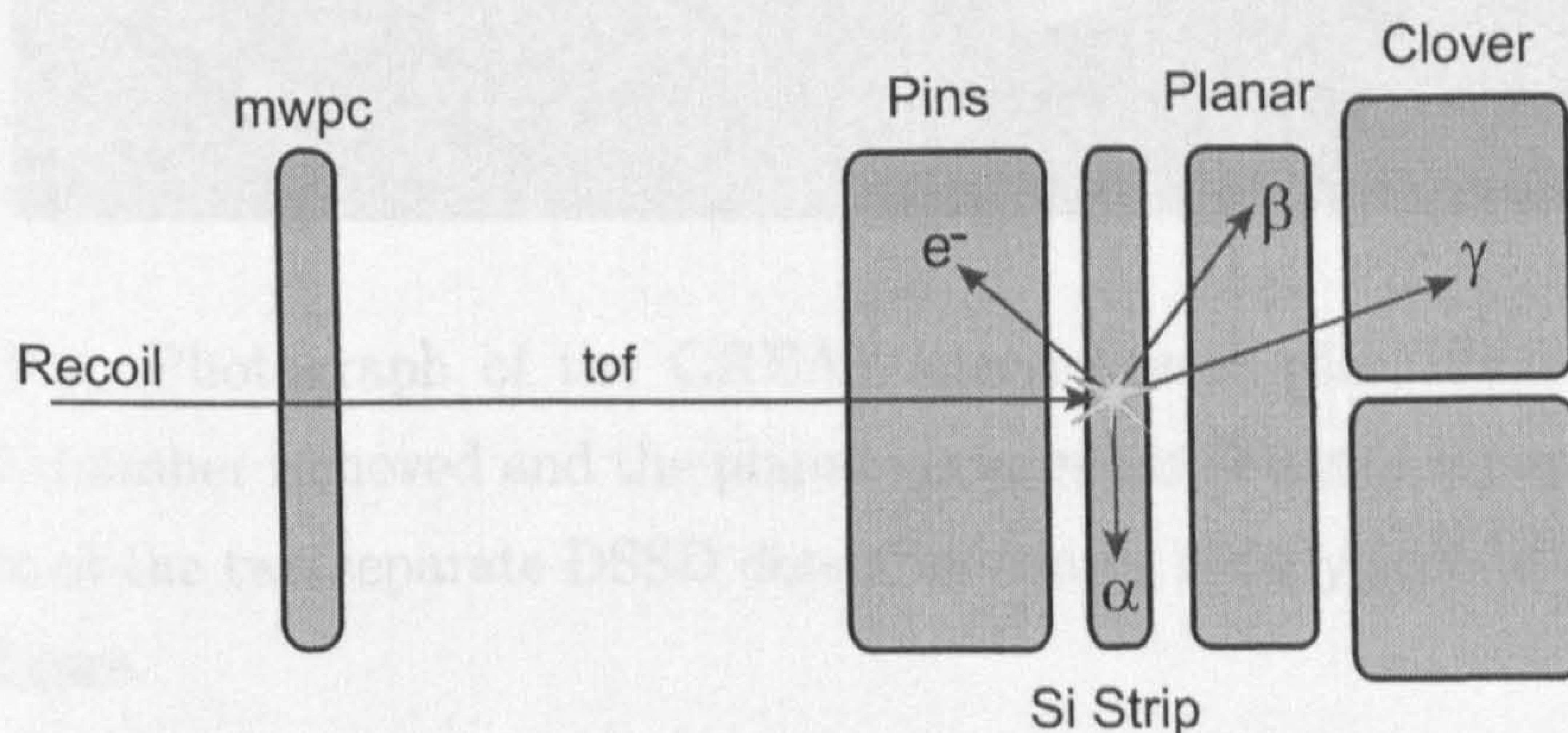


Figure 2.2: Schematic showing the basic layout of the GREAT spectrometer. GREAT consists of five main parts. The multi wire proportional (Gas) counter, a double sided silicon strip implantation detector, an array of PIN diodes, a planar germanium detector and a segmented germanium clover detector.

Figure 2.3 is a photograph showing a partial setup of the GREAT spectrometer situated at the exit window of RITU. In the figure the lid of the vacuum chamber has been removed. The two individually mounted DSSD detectors can be seen at the front of the figure surrounded by the banks of associated preamplifiers. Below the DSSD detectors is the face plate which is removed to allow the planar germanium detector to slide into place. The

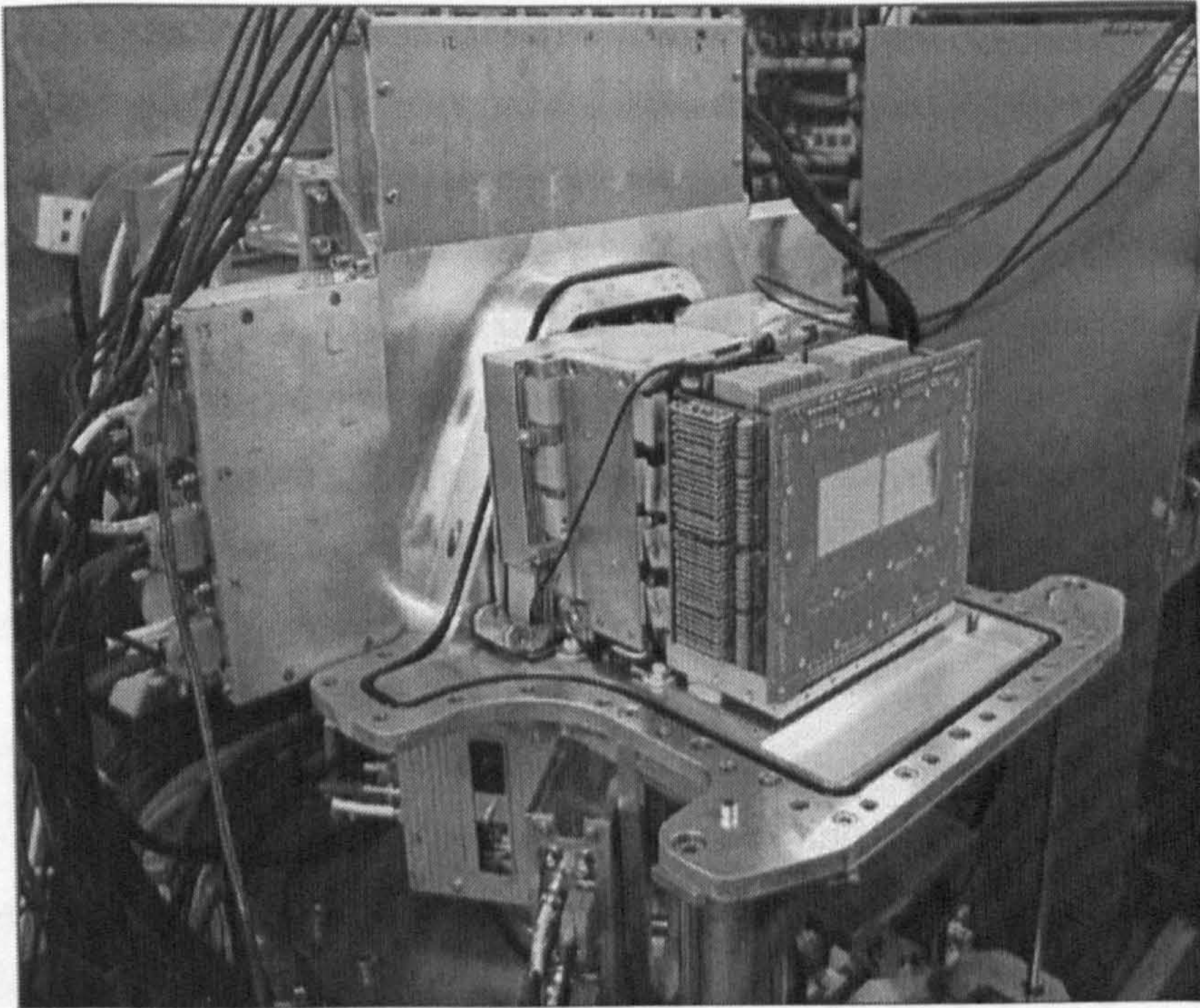


Figure 2.3: Photograph of the GREAT spectrometer with the lid of the vacuum chamber removed and the planar germanium detector removed. The rear face of the two separate DSSD detectors can be clearly seen at the front of the figure.

PIN diode array is situated on the inner face of the detector and therefore cannot be seen in this photograph.

2.4.1 Multiwire Proportional Counter (MWPC)

After the recoil products leave the separator the first part of the GREAT spectrometer they enter is the multiwire proportional counter (MWPC). The multiwire proportional counter is situated at the exit of RITU. The MWPC is filled with low pressure isobutane gas and is separated from the helium of RITU at one end and the vacuum of the rest of the GREAT spectrometer by two thin Mylar windows.

The main function of the MWPC is to act as an active recoil discriminator. Any recoil product passing through the counter deposits energy, if the time-stamped data generated from these events is in prompt coincidence with any signals from the DSSD. It can be inferred that the implantation event must be caused by an object that has passed through the MWPC i.e. it is a recoil product. By the same reasoning any data item generated in the DSSD that is in anti-coincidence with the MWPC must not have passed through it i.e. it is a result of a decay from something already implanted in the DSSD for example an alpha decay from an embedded recoil.

Another purpose that the multiwire proportional counter can be used for is to discriminate between the recoiling nuclei and scattered beam that has not been filtered out by the recoil separator. By combining the timing information generated by the MWPC from nuclei passing through it with the energy deposited in the DSSD implantation detector it is possible to clearly identify the recoils and scattered beam. By selecting only these identified recoils at the data analysis stage it is possible to provide a cleaner recoil signal. Examples of this technique are given later in chapter 5.

2.4.2 Cooling Block

An important part of the GREAT spectrometer is the cooling block. As well as being held in a vacuum, it is useful to cool both the double sided silicon strip detector and the PIN diodes. Although the detectors operate at room temperature, cooling greatly reduces the noise levels and improves the resolution in these set-ups. The cooling block is a hollow metal block through which coolant is pumped. The cooling block reduces the temperature of the mounted DSSD's and PIN diodes to -20^oC. Other detector systems e.g. the planar germanium and focal plane segmented clover detector are also cooled (albeit to much lower liquid nitrogen temperatures of around $63K$) by their integral liquid nitrogen cooling systems.

2.4.3 Double Sided Silicon Strip Detector (DSSD)

The double sided silicon strip detector is the core of the spectrometer. Recoils that have been separated by RITU and have passed through the MWPC implant here. This implantation along with their subsequent decays by α or β particle emission are measured by the DSSD. The DSSD's need to measure both high energy recoils of around $50MeV$ as well as the subsequent α decays of around $5 - 10MeV$. With the energy of β particles and protons falling around $500keV$ the possible energy range that the DSSD must be sensitive for is quite large. To compensate for this a series of Degrader foils can be placed into the path of the recoiling nuclei before they implant to slow them down. This allows the detector to cover the full energy range whilst still maintaining adequate resolution.

Two individual DSSD's are placed at the focal plane of GREAT each having an active area of $60mm$ by $40mm$ with a silicon thickness of $300\mu m$. The strips in both the x and y directions have a width of $1mm$ meaning each DSSD has a total of 60 by 40 strips giving the total number of effective pixels per DSSD as 2400. The two DSSD's are mounted side by side with the respective active areas being $6mm$ apart. The full DSSD therefore has a total number of 4800 pixels that has an estimated recoil collection efficiency of approximately 80%.

As each detector is to be read out individually each strip of the detector is attached to its own charge sensitive pre-amplifier. These pre-amplifiers are mounted inside the vacuum chamber of the GREAT spectrometer on the outer face of the cooling block. The location of the pre-amplifiers allows them to be directly connected to the DSSD's. This minimises the length and number of connections between the signal output of the DSSD and the input stage of the pre-amplifiers. Limiting the length and number of connections of the cabling maximises the energy resolution of the detectors.

2.4.4 PIN Diodes

The array of PIN diodes is primarily used for measuring energies of conversion electrons emitted from the de-exciting nucleus. An implanting nucleus typically embeds close to the surface of the DSSD detector [1] giving a significant chance that any conversion electrons are emitted in the backward direction. It is also possible that any α particles emitted by the nucleus could also escape the DSSD in the backward direction, the array of PIN diodes could therefore also be used for add-back calculations to improve the detection efficiency for α decay.

The PIN diodes are arranged in a "box-like" configuration around the outer perimeter of the DSSD detector. There are a total of 28 PIN diodes in the array, each has an active area of $28mm$ by $28mm$ and a thickness of $500\mu m$. The PIN diodes used were the S3584-06 windowless series manufactured by Hammamatsu. The PIN diodes are mounted in pairs to custom PCBs containing the input stage of the preamplification before being passed onto the external PSC761 preamplifiers manufactured by Eurisys mesures.

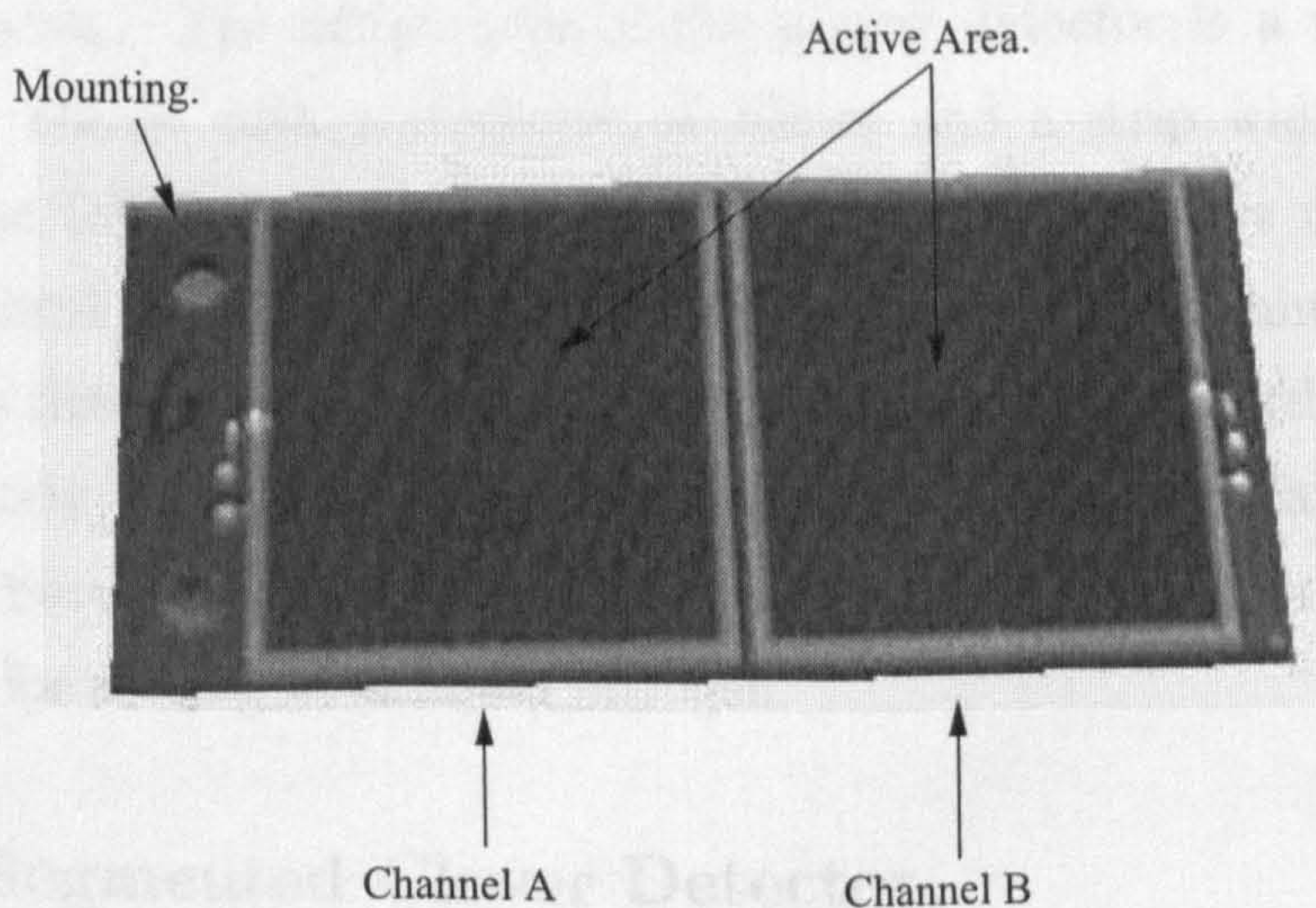


Figure 2.4: Photograph of the Hammamatsu S3584-06 windowless series PIN diodes dual mounted onto their PCBs.

[1]Typically $1 - 10\mu m$

A photograph of the twin mounted PIN diodes are shown in figure 2.4. The PIN diodes and PCBs are mounted on the inside surface of the cooling block so as to cool the PIN diodes to $-20^{o}C$. The energy of conversion electrons produced in the primary reactions being studied are up to a maximum energy of around $500keV$ at which the PIN diodes have an approximate energy resolution of around $5keV$.

2.4.5 Planar Germanium Detector

The planar germanium detector's main purpose is to measure the energy of X rays and low energy γ rays and β particles emitted by the recoiling nuclei embedded in the DSSD detector. The planar germanium detector is placed directly behind the DSSD and inside the vacuum chamber to minimise the attenuation of any photons. Any β particles detected in the planar germanium detector must be of high energy ($\geq 2MeV$) so as to penetrate through the silicon of the DSSD.

The planar germanium detector is segmented into strips similarly to the DSSD detector. The active area of the planar detector is a rectangle of $120mm$ by $60mm$ with a thickness of $15mm$ and a strip width of $5mm$. As with the DSSD implantation detector these strips can be used to provide positional information about any events occurring within the planar germanium detector. The front face of the planar detector itself is situated approximately $10mm$ away from the rear of the DSSD. The planar detector has a thin beryllium entrance window and the whole detector is mounted to a cryostat for cooling with liquid nitrogen.

2.4.6 Segmented Clover Detector

The segmented clover detector is used for the measurement of high energy gamma rays emitted from the recoils embedded in the DSSD. These gamma rays must have passed through the thickness of silicon in the DSSD and through the planar germanium detector to be detected by the clover. The clover detector is mounted outside the GREAT spectrometers vacuum

chamber and consists of four germanium crystals each with four-fold segmentation. Each clover crystal is 70mm in diameter and 105mm long with the first 30mm of the crystal tapered at an angle of 15° on the outside surface. Each clover detector is also surrounded by a bismuth germanate suppression shield to help improve the peak to total ratio.

2.4.7 Silicon-Gas Tac

A TAC (Time to Amplitude Converter) is an instrument which converts the time interval between two logic signals into an output pulse. This output pulse has an amplitude that is proportional to the time interval between these two logic signals. The silicon-gas TAC in the GREAT spectrometer consists of two logic gates, one that is triggered when a signal is generated from the MWPC gas detector and another that is generated by any signal in the focal plane silicon implantation detector. The TAC then generates an amplitude pulse that is related to the time interval between these logic signals.

The generated pulse is higher resolution than the time-stamping used in the system metronome, it can therefore be used to generate more accurate time of flight information for recoiling nuclei. This information is useful in the generation of high resolution energy, time of flight matrices that can used to create data selection gates to accurately distinguish recoils from scattered beam particles. As well as using the silicon-gas TAC, the time of flight information can also be estimated using the parameters of the experiment e.g. produced nuclei mass, velocity etc. As an estimate of the order of magnitude of flight times through the spectrometer, presuming that the velocity of nuclei through the 3m long spectrometer has a $\frac{v}{c} \sim 1.5\%$, then the approximate time of flight through the spectrometer is $1\mu s$.

2.4.8 Target Position Arrays

Although not part of the GREAT spectrometer itself an integral part of the setup is the target position detector array. The target position array is used to detect prompt decays (i.e. short lived transitions of $t \lesssim 1ps$) in

the nucleus being studied. As the target position array is independent of the GREAT focal plane spectrometer it can be set up with different detector types depending on the current experimental interests. One such array that is used in Jyväskylä is the JUROGAM gamma ray spectrometer that is used to detect prompt gamma rays at the target position. The JUROGAM array uses 43 Compton escape-suppressed germanium detectors that combined have a gamma ray detection efficiency of 4.2% at $1.3 MeV$.

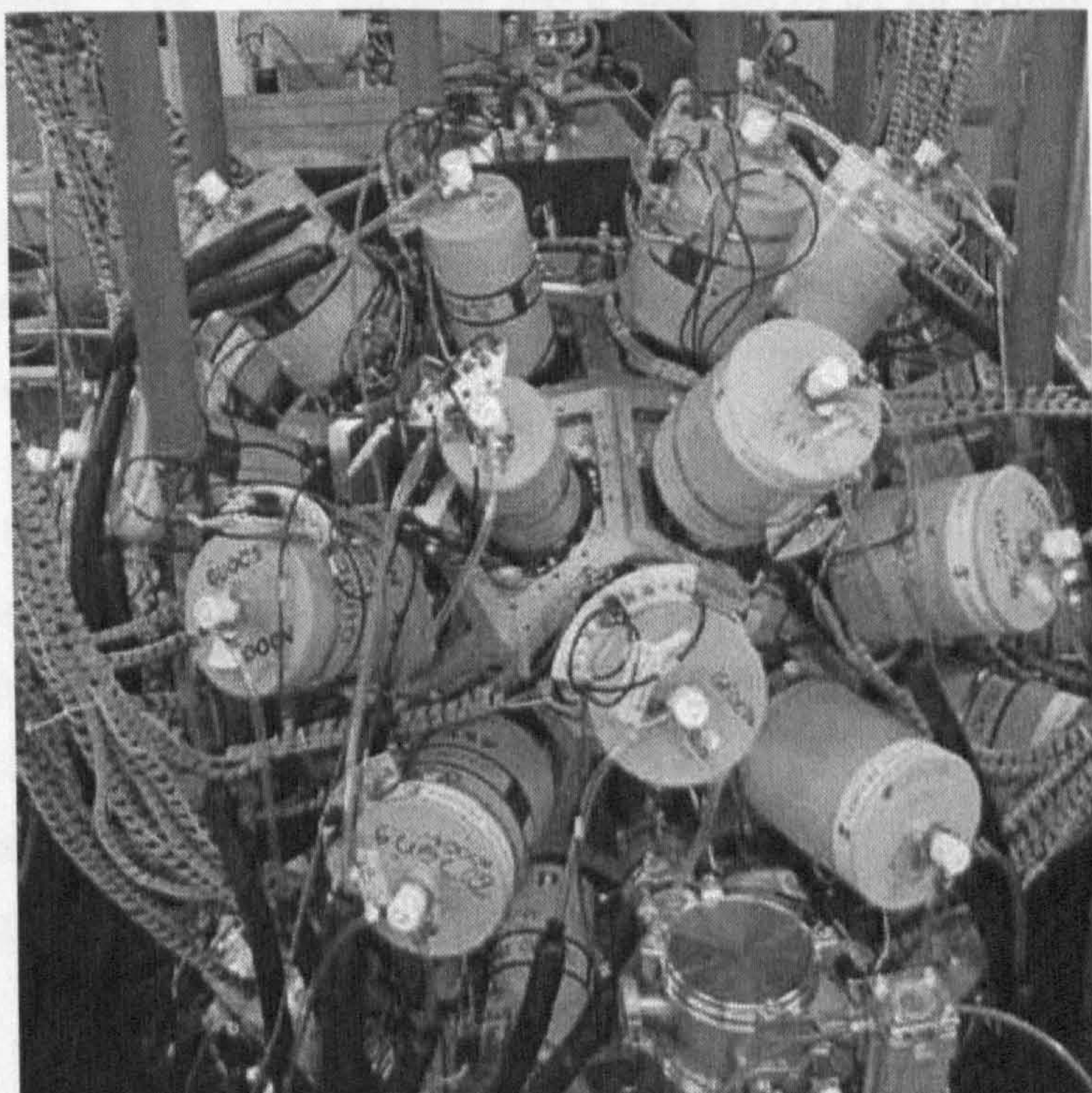


Figure 2.5: Photograph of the JUROGAM gamma ray spectrometer in situation at the laboratory of the university of Jyväskylä.

2.5 Example Data Set

Throughout this thesis several physics examples are used to illustrate that the principles being discussed can be applied to real physical situations. All of these examples are generated from the same data set gathered from an experiment performed at the laboratory in Jyväskylä, Finland. The data

set used was gathered from the fusion evaporation reaction where a beam of ^{48}Ca was incident on a ^{208}Pb target to produce ^{254}No nuclei through the 2 neutron evaporation channel.

This reaction was chosen as ^{254}No has been the focus of much study [12][18] so the basic properties of the nuclei are known to some accuracy. This knowledge is essential in order to infer that any techniques used are producing accurate results. The ^{254}No nucleus has a half life of $51.2s$ which decays primarily by alpha emission to ^{250}Fm with an energy of $8.09MeV$. It is therefore possible to benchmark any results against these well known values to check if the techniques discussed produce meaningful results. One such check that is performed in section 6.6 is to calculate the half-life of the ^{254}No using this α transition.

In using the TDRSorter data analysis code to analyse a given set of data certain running parameters need to be set in order to extract meaningful results from the data stream. The values of these parameters are mentioned here so as to provide a complete description of the example data. Where components of the data analysis code are mentioned that have not yet been discussed in the text, references to the relevant sections of the thesis are given.

In total the example data set used is over $49Gb$ in size and was gathered over an experimental running time of approximately 12 hours. The data set consisted of $\approx 2.1 \times 10^9$ individual data items that where generated by the TDR data acquisition system. From the time ordered data stream output from the system around $320,000$ events (see chapter 5) were reconstructed that belonged to the channel of interest. This experiment was run with an unusually large background due to redundant channels in the high rate clover detector being read out to the data stream. From this high background the important events where extracted.

The time buffer of the data analysis code (CBuffer class see chapter 4) was constructed with a total time window of $20\mu s$ with the constituent forward and backwards time windows both having sizes of $10\mu s$ each. Event construction (chapter 5) was triggered by the presence of a silicon-gas TAC data item at the read position of the time buffer. A total DSSD X and Y strip search time of $1\mu s$ in the forwards and backwards time directions was used for pixel definition.

The search time for any gas data items was set at $\pm 1\mu s$ whereas the time for all other searches for event construction set to $10\mu s$ in the forwards and backwards time direction. No energy gates on the DSSD X and Y data items were set due to different gain ranges used on the X and Y strips of the DSSD. This non standard gain range was used due to the requirements of a different experimental run of which the gathering of this data set was a small part. The final parameter of the TDRSorter data analysis code that was the depth of the tagger (i.e the correlation time as shown in chapter 6). The tagger depth was set to $550s$ which corresponds to approximately ten half lives of the ^{254}No α decay to ^{250}Fm.

2.6 Summary

This chapter has discussed firstly the experimental motivation and also the overall design of the GREAT spectrometer. The discussion went into some detail describing the structure and function of the individual detector components within the spectrometer. A brief overview of the example data set used in subsequent physics examples was also given as well as a brief discussion of the TDRSorter parameters used to analyse the data gathered from the experimental run.

Chapter 3

TDR Data Acquisition System and Data Format

3.1 Overview

In this chapter the TDR data acquisition system will be discussed. Coverage of the high level design of the electronics system is also provided as well as a discussion on the process of data acquisition. Descriptions of the TDR data format and the TDR data stream will also be given. The details of these are important in understanding the design decisions taken in the development of the TDRSorter data analysis code. Most of the information in this chapter is available in the form of electronic documents at the following web address [13]. Relevant specific documents from the website are referenced individually. The discussion in this section provides information about the details of the relevant data formats and structures that is important for future discussions on data analysis techniques.

3.2 TDR Electronics System

The GREAT spectrometer has been implemented with a new triggerless data acquisition system called TDR (Total Data Readout)[2]. The main feature of this system is the lack of a system wide hardware trigger. Each

individual detector channel is read out independently of the other channels in the spectrometer. In the context of this thesis a detector channel is essentially an individual detector element such as a single strip of the DSSD or an individual PIN Diode detector.

Each detector channel is mapped to its own linear shaping amplifier which in turn is mapped to its own Constant Fraction Discriminator(CFD) which is then assigned its own channel in the Analogue to Digital Converter (ADC). Essentially within the TDR system each detector has its own acquisition path i.e. there are as many linear amplifiers, CFD's and ADC's as there are individual detectors. This has the effect of decoupling all the detectors from each other allowing each detector to gather as much (or as little) data as it is capable of gathering regardless of the state of other detectors in the spectrometer.

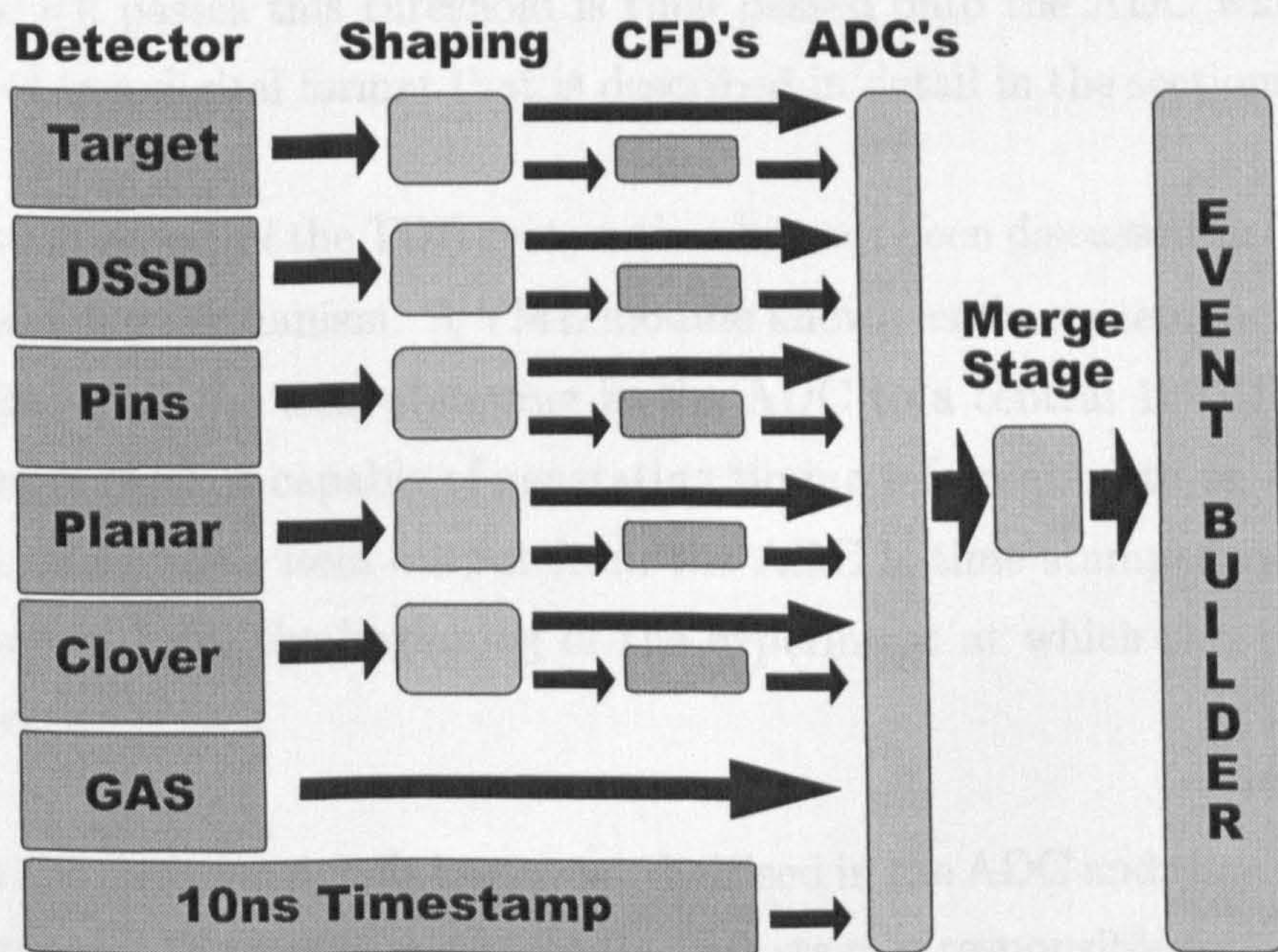


Figure 3.1: Schematic of the TDR Electronics System. The figure shows how the individual detector channels are mapped to their own individual linear shaping amplifiers, constant fraction discriminators (CFD) and analogue to digital converter (ADC).

Figure 3.1 shows diagrammatically the layout of the major subsystems of the TDR electronics system. In the figure, detectors are divided into groups representing each individual detector channel. As mentioned before each detector is mapped individually onto an amplifier, CFD and ADC. When a charge pulse is generated from incident particles or radiation in a detector the signal is first pre-amplified and then passed onto the linear shaping amplifier. The shaping time of the linear amplifier accounts for the majority of the dead time in the data acquisition system. As the amplification from this stage takes a finite time (shaping time) to complete the fast output from the amplifier (where no amplification is performed) is passed into the CFD's input.

The constant fraction discriminator essentially sets a threshold for which the magnitude of any signal from the detector must be greater than. Any signal which passes this threshold is then passed onto the ADC where it is converted to a digital format that is described in detail in the sections below.

One vital aspect of the TDR system that has not been discussed so far is the time-stamping mechanism. A VME module known as the system metronome synchronises all the time-stamping in the ADC to a central 100MHz clock. The system clock is capable of generating timing information to an accuracy of $10ns$. Each data item output from the ADC is time stamped indicating a time offset from the beginning of the experiment at which this data was gathered.

After the detector signals have been digitised in the ADC and time stamped they are passed on to the merge stage. This stage is responsible for arranging the data output from the ADC into time order. This stage is important as data can arrive at slightly different intervals from the separate ADC cards and it is an important assumption of all further data analysis that all the data is in strict time order. The details of the produced data stream are given in the section 3.3.3 below.

The final step in figure 3.1 is the event builder. The event builder is responsible for performing basic checks (e.g. time ordering) on the data and marshalling the data to various output sources. The event builder can write the raw data stream to a tape source as well as to various online sorting software where basic checks on the performance of the experiment can be monitored. Most of the units mentioned here, particularly the ADC's and metronome were custom built at Daresbury laboratory, more information about most of the TDR electronics is available in reference [2]

3.3 TDR Data Format

Each ADC card outputs both data items and information items. The data items essentially contain the ADC data; the channel number of the detector that fired and the time that the detector fired. The information items contain all the other information i.e. piled up data items, pause and resume functions etc. The detailed format of these data and information items[14] is described below.

3.3.1 Data Items

Each data item consists of 64 bits of information that is output from the ADC as two 32 bit words. Figure 3.2 shows the structure of each data item. Bits 0-15 in the first word contain the data from the ADC. This is essentially the digitised signal from the analogue linear amplifiers. Bits 16-27 contain the channel identity information. The final four bits 28-31 contain a fail and veto flag at bits 28 and 29 respectively. Bits 30 and 31 are always set to 1 and can then be used to distinguish data items from other information items (discussed next) as they have a different bit pattern.

The second word contains the time stamping information. Bits 0-27 contain the lower portion of the total time stamp which is 48 bits in length. The higher order more slowly changing 19 bits are transmitted in the information items every $64\mu s$. The top four bits of the second word, bits 28-31 are used

for padding the time-stamp out to 32 bits in size. These padding bits are assigned a value of zero.

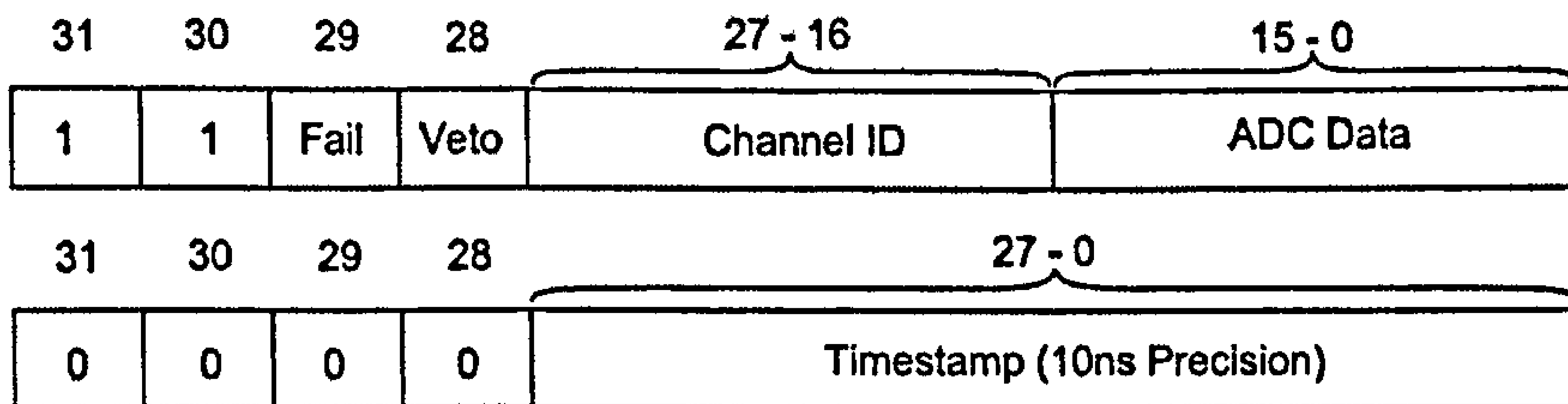


Figure 3.2: Schematic of the TDR data item format. The arrangement of the various components of the data item including the channel of the detector, the actual ADC data and the time-stamp showing the time that the data item was generated.

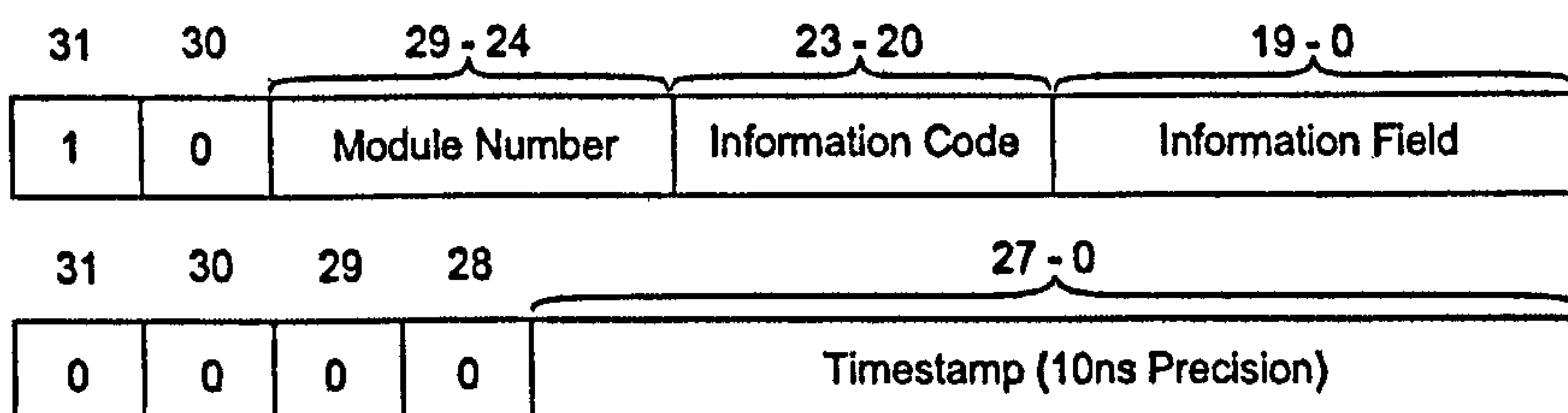


Figure 3.3: Schematic of the TDR Info Item Format. The arrangements of the various components used in specifying information important to the sorting process is shown. The data carried by the info items relates to any information other than detector event information, e.g. time stream sync pulses.

3.3.2 Info Items

All other information is passed in the form of info items [14], see figure 3.3. As with the data items the info items consist of two 32 bit words making up 64 bits of information. Bits 0-19 in the first word contain the information field, this field contains data that relates to the information code given in bits 20-23. Table 3.1 taken from [14] shows what data is contained in the

information field as it corresponds to the information code. Bits 24-29 contain the module number. The module number identifies the number of the ADC VXI card that sent the information. The final two bits 30 and 31 contain the values 0 and 1 respectively that distinguishes the information items from the data items described earlier (data items contain a value of 1 for both bits). The second word contains the time stamping information and has an identical format to the Data Items time stamping word.

Code	Information Type	Information Field Definition
0	Undefined Data	
1	ADC Channel Pile-up	Channel Number
2	Pause Time-stamp	Time-stamp bits 28-47
3	Resume Time-stamp	Time-stamp bits 28-47
4	SYNC100 Time-stamp	Time-stamp bits 28-47
15	SHARC Link number	Link Number.

Table 3.1: Table showing the information code and information field definitions that the respective components in the info item data structure can take.

The total length of the time-stamp associated with each data item is 48 bits. Only the bottom 27 bits are transmitted with each data item. The top 19 bits are transmitted every $64\mu sec$ in the information field of the SYNC100 Information Items (Code 4 in table 3.1).

3.3.3 Data Stream

All the data and information items output from the ADC cards are arranged into strict time order. Figure 3.4 shows a high level view of the structure of the outputted data stream. This time ordered list of data is what is ultimately stored and analysed. As the unaltered data stream is stored, the data can be analysed multiple times using any software created triggers that the user cares to invent.

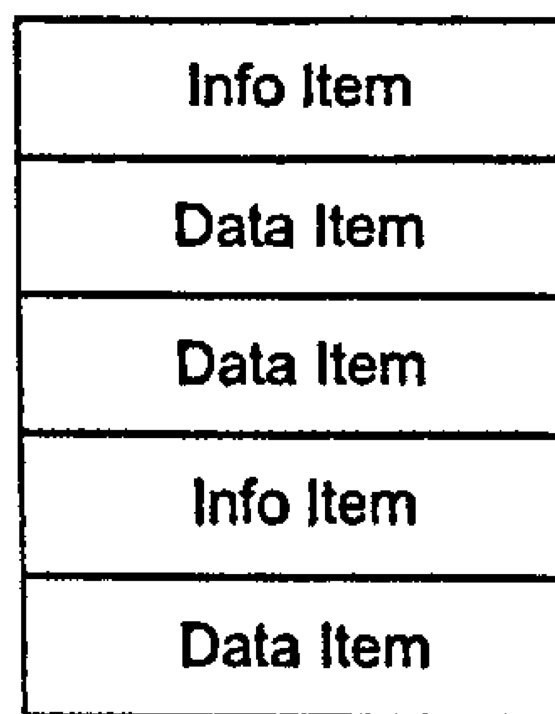


Figure 3.4: Schematic showing the general data stream structure. i.e. a time ordered series of data and information items.

3.3.4 Block Structure

The data stream is split up into 16kb blocks for transmission and storage [15]. This process does not alter the time ordered nature of the data stream but is simply a convenience for storage and analysis. Figure 3.5 shows the structure of each 16kb block.

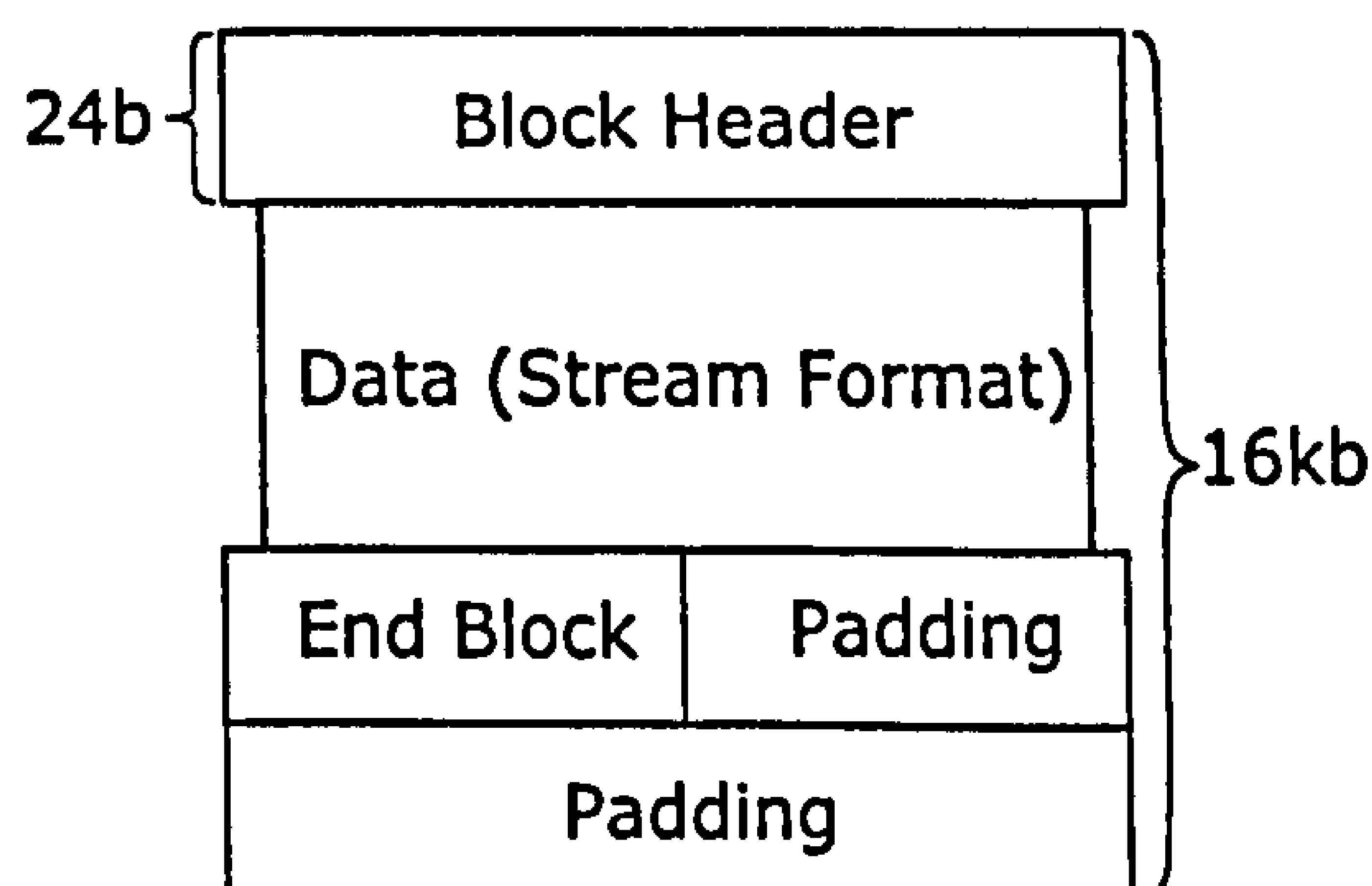


Figure 3.5: Schematic showing the 16kb data block structure that the time ordered data stream is segmented into for serialisation onto hardware storage systems such as tape or hard drives.

Each block consists of four main parts; The header; The Data part; The end of block statement and the padding. The header Figure 3.6 consists of 24 bytes (192 bits) of information that describe the data in the data part of the block. The first 8 bytes are a simple identifier that designates the type

of block. In this case the bits represent the arbitrary ASCII character string 'EBYEDATA'. The next 2 bytes of the header is the block sequence, this is simply the numerical order of the block as it is transmitted. The next 2 bytes is the stream number, this number simply defines which data stream in the data acquisition system the data block originated from. The 2 bytes proceeding this contain a variable called 'Tape'. The value of this variable is always set to 1.

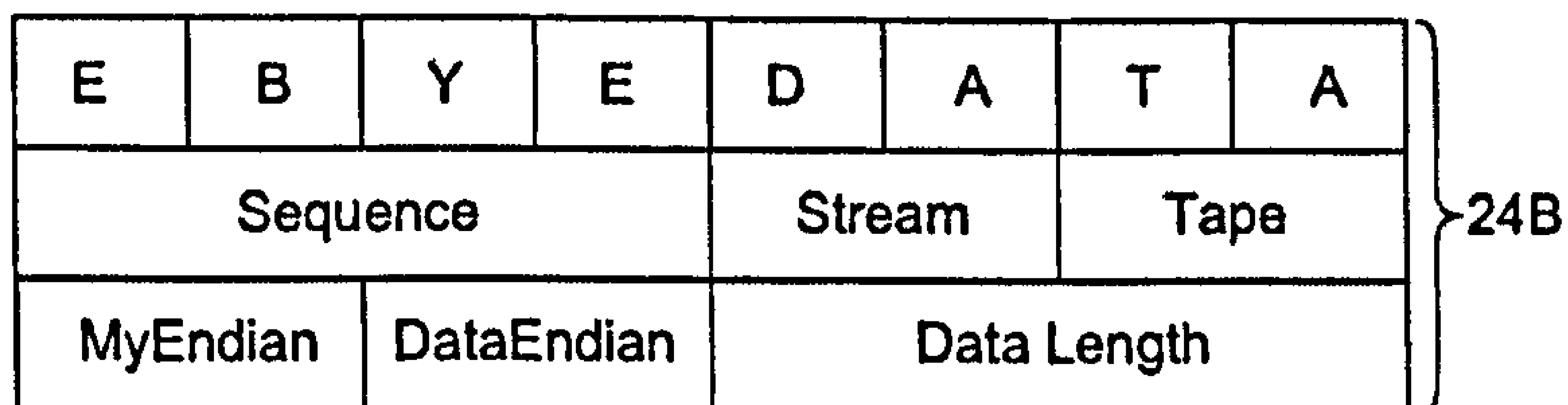


Figure 3.6: Schematic showing the detailed structure of the 24 byte block header portion of the data block structure shown in figure 3.5. The block header contains various parameters for describing the position of the block within the serialised data file and also the endianness of the binary data contained within it.

The next 4 bytes of the header contain two variables called MyEndian and DataEndian, respectively. These two variables allow the hardware architecture of the data source to be determined. The variable MyEndian is written as a native 1 on the Tape Server (i.e. the computer that writes the data to tape or to disk). The second value DataEndian is written as a native 1 in the hardware architecture of the machine where the data originated (i.e. The data acquisition system). By examining these two values it is possible to determine the endianness of the various systems involved. This determination is critical if the data is to be interpreted correctly. The final 4 Bytes in the header contain the data length. This variable simply states the length in bytes of data that follows the header.

The Data part of the block is simply the data stream of Info items and data items that was described earlier. The third part of the block is the end of

block sequence, this sequence is simply 4 bytes that contain the hexadecimal value (0xFFFFFFFF). This value simply declares that the length of data declared in the block header has finished. The fourth and final part of the block is the optional padding. If the total length of header, data stream and end of block statement is less than the total block length of 16 kilobytes then padding bytes containing the hexadecimal value (0x5E) are inserted at the end of the block until the total length is achieved.

3.4 Summary

This chapter briefly described at a high level the operation of the TDR data acquisition system, specifically how charge pulses generated by the detectors are converted into time-stamped digital data. A detailed description was also given of the data format output from the ADC's and the structure of the resultant data stream.

Chapter 4

Code Architecture and Data Buffering

4.1 Overview

This chapter explains both the high level design choices and the detailed implementation of the data buffering process. The chapter is started with a simple flow diagram that highlights the general structure of the data analysis code. Following this is a general UML (Unified Modeling Language, see section 4.3) diagram indicating the various classes used throughout the data analysis code. Following on from this is a detailed discussion of the various data buffering methods and also a detailed implementation of the data analysis codes time buffer and the various solutions to various buffering issues.

4.2 High level Code Structure

Using a UML (see section 4.3) class diagram as a starting point, each major section of the code is given a more in depth treatment. Details of the algorithms and data structures used and the decisions behind their choice are also covered in the individual sections below. Initially the chapter is started with a high level overview of the TDRSorter data analysis code which gives a useful outline of how all the components fit together. Following on from

this a more detailed discussion of the operation of each area of code is given.

4.2.1 Logical Code Structure

Figure 4.1 shows the logical code structure of the TDR sorter data analysis program. The functioning of the TDRSorter code can be broken down roughly into the following sections; data input; data buffering; event packaging and specialised sorting. The first section, data input, is fairly elementary in its construction and operation. It is mainly concerned with getting the raw binary data output from the ADC's, into memory, and in a form that is usable.

The data buffering section of the TDRSorter code (see section 4.4) poses one of the major challenges in reconstructing useful physics information from the triggerless data stream. Here the time ordered data is buffered in such a way that given a particular data item in the data stream, all data items that have a time-stamp that fall within a specified time period (both forwards and backwards in time) can be accessed.

The event packaging section of the TDRSorter code (see section 5.5) makes heavy use of the time buffer constructed in the data buffering portion of the code. Given a user defined software trigger the time buffer is searched for other data items from any other detector that lie within specified time periods of the trigger data item[1]. The event packaging mechanism is essentially a way of reconstructing prompt coincidences from detectors.

The final section of the TDRSorter code is the specialised sorter section (see chapter 6). This portion of the code is essentially where the sorted data is tested against the expected physical outcomes of the experiment. Delayed coincidences can be stored and analysed with a tagging framework and one and two dimensional histograms can be generated.

[1] Up to the maximum time period of data items stored in the time buffer.

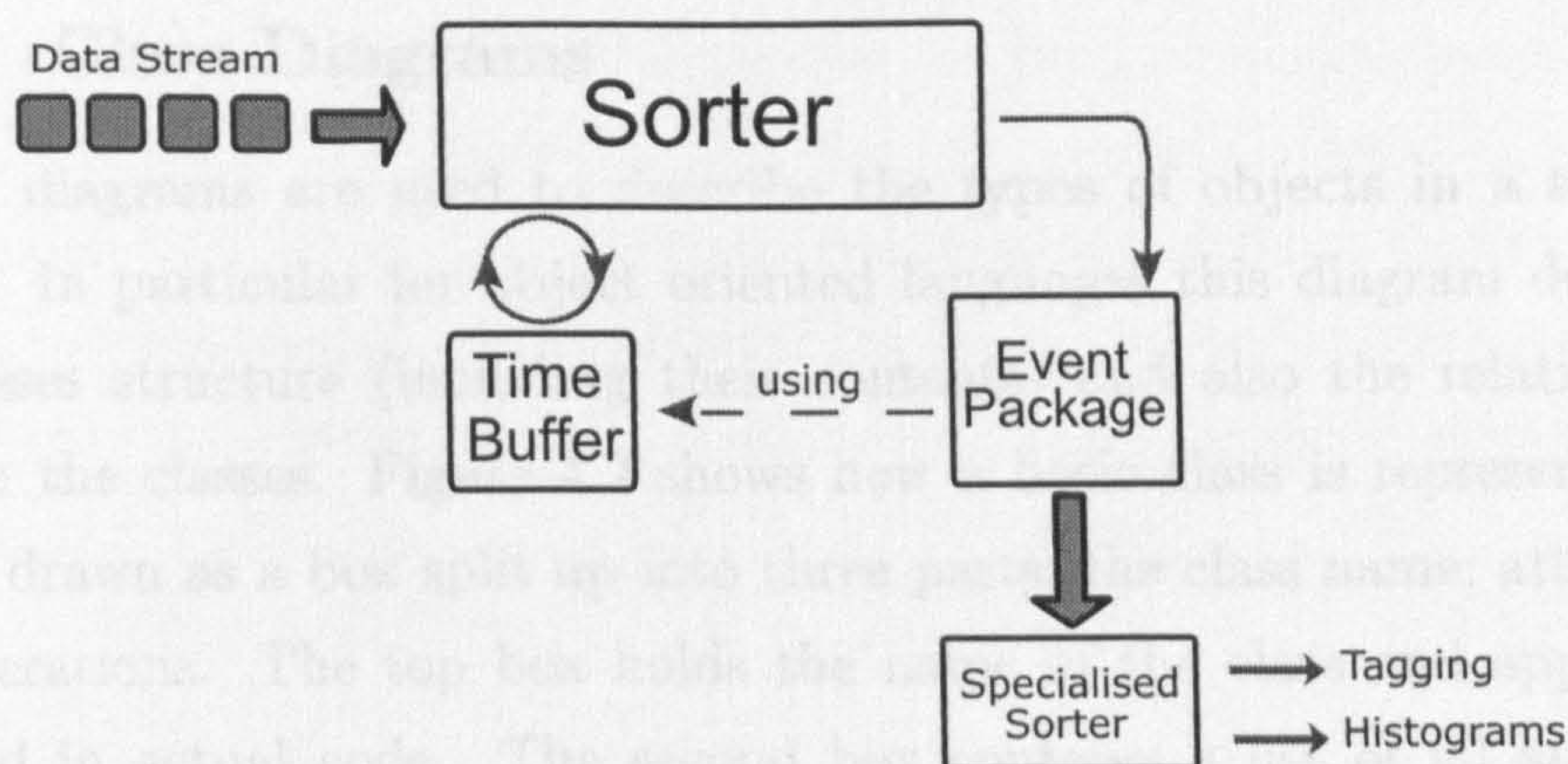


Figure 4.1: Diagram showing the logical structure of the TDRSorter data analysis code. The figure shows how the code is split into four main sections; data input from the data stream; Data buffering using a time buffer; Event packaging and specialised sorting.

4.3 UML Diagrams

Throughout this thesis much of the discussion of the data analysis techniques used are backed up using examples of code from the TDRSorter program. In order to help in the understanding of these portions of code and how they relate to each other within the overall structure of the program it is useful to have some sort of diagrammatic visualisation.

An international standard notation called the Unified Modeling Language[2] or UML has been defined to help in the visualisation and construction of object oriented software. UML uses several different types of diagrams to document the components of a software system using engineering best practices. Throughout the thesis constant reference will be made to standard UML class diagrams. The following section attempts to provide a brief introduction to the various notations used in these types of diagram.

[2]The UML standard is maintained by the Object Management group `http://www.omg.org`

4.3.1 Class Diagrams

Class diagrams are used to describe the types of objects in a software system. In particular for object oriented languages this diagram describes the classes structure (including their contents) and also the relationships between the classes. Figure 4.2 shows how a basic class is represented. A class is drawn as a box split up into three parts; the class name; attributes; and operations. The top box holds the name of the class and appears as it would in actual code. The second box contains a list of all attributes contained in the class. An attribute is essentially the data that the class contains for example in the **CEvent** class an attribute would be the **CPixel** pixel object. The third box contains a list of all operations contained in the class, each operation is a method (or function) that can be called by users of the class. Referring again to the **CEvent** class an example of an operation would be the **GetProperties()** method.

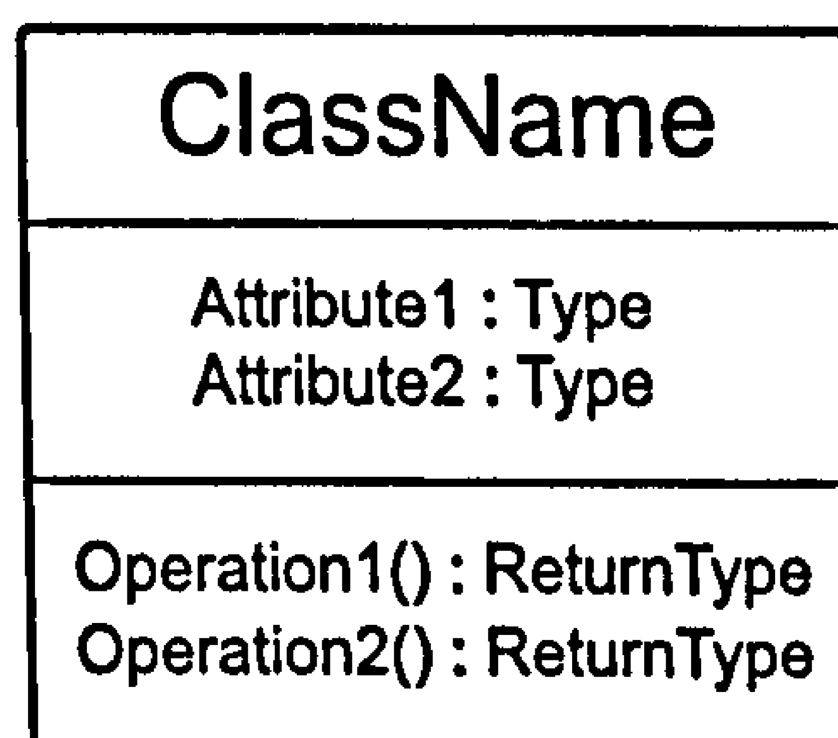


Figure 4.2: Figure showing the basic notation for describing a class in UML. The two core types shown are attributes and operations which correspond to the class member data and methods, respectively.

When declaring attributes in the class diagram the name is given first. Following the name is a colon and the type of the attribute being declared. For example, in many modern programming languages an integer variable would be declared **int x**, in UML notation the variable would be declared as an attribute as x : **int**. A similar convention applies when describing operations in UML. Firstly the name of the operation is given then separated

by a colon the return type of the operation is given. For example a generic function **int main()** would be declared in UML as **main():int**. Figure 4.3 shows an example UML class diagram, alongside of which is a section of c++ code that it represents. One additional point to note is that the parameters of methods are described using the attribute syntax described earlier.

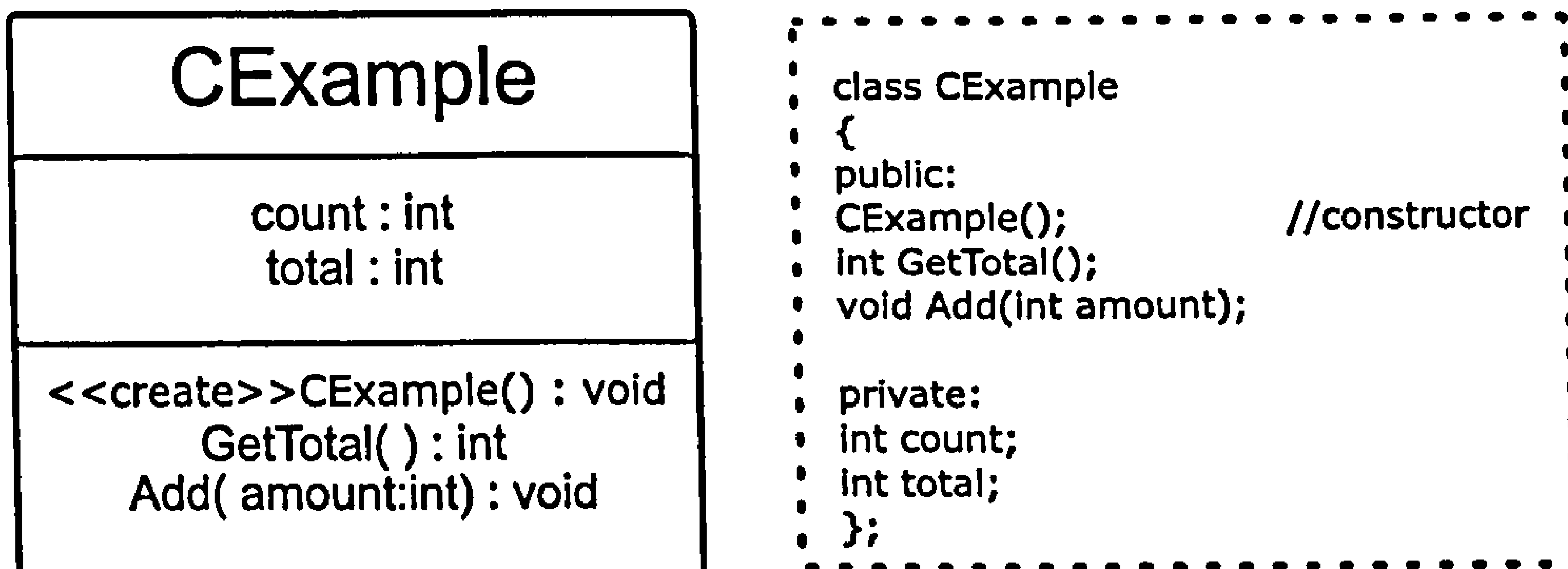


Figure 4.3: Figure showing an example UML class diagram and the corresponding c++ code that implements the diagram.

Relationships Between Classes

As well as describing the structure and content of classes the UML class diagram is used to describe how these classes relate to one another. Figure 4.4 shows some of the most commonly used relationships in UML class diagrams. The list of relationships shown here is not exhaustive and only notation used in the thesis is given here. Further information about UML relationships [3] can be found in [16].

The first relationship shown is aggregation. Aggregation is used to model classes that contain references to other classes but do not own them i.e. they are not ultimately responsible for the creation or destruction [4] of the

[3] And most other UML information.

[4] Classes that possess an aggregation relationship with another class can play a part in the creation of objects. A good example would be a class that creates an object but passes ownership to another class.

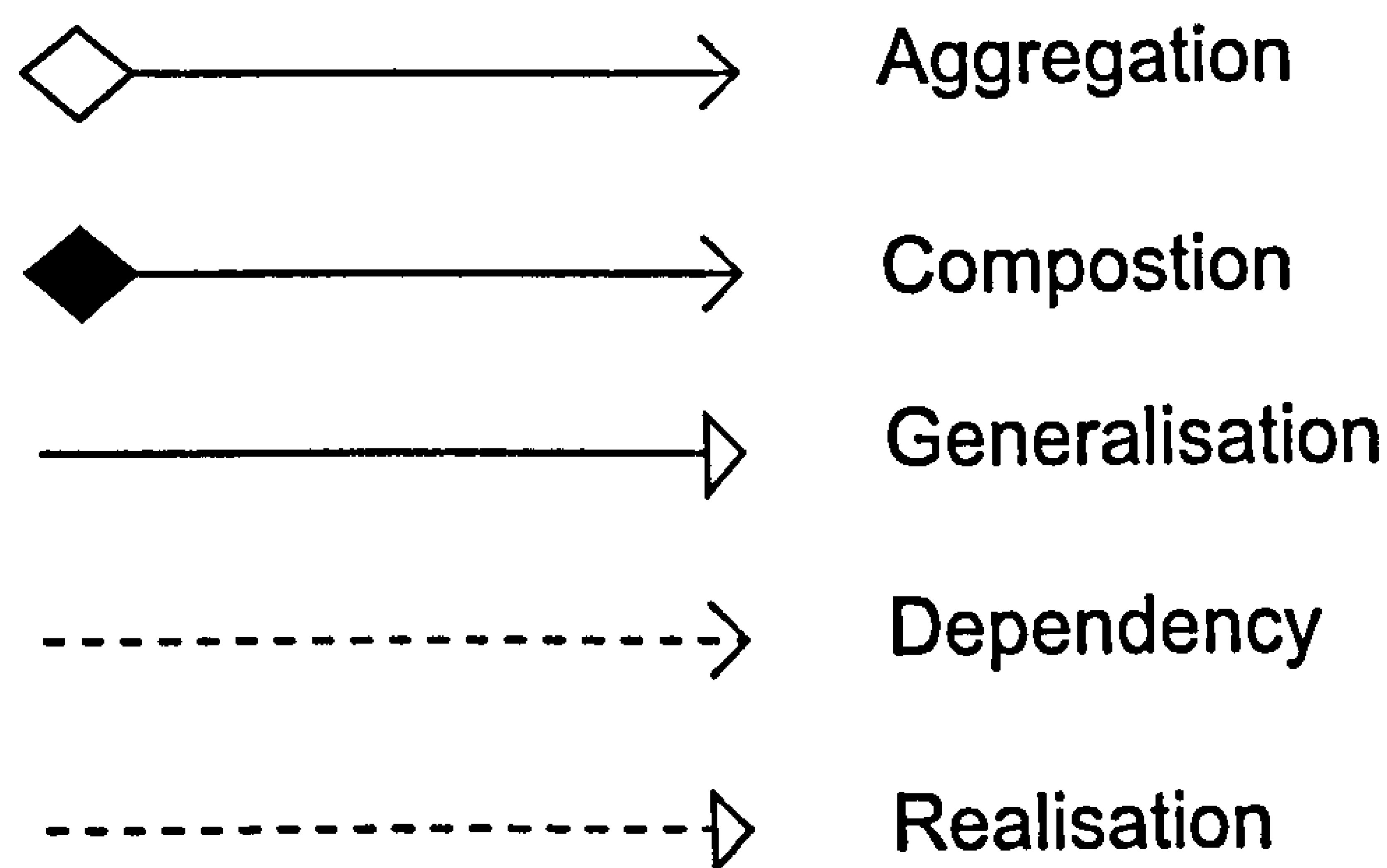


Figure 4.4: Figure showing the most commonly used Relationships used in UML.

aggregated objects. Another way of looking at the aggregation relationship is to say that it represents a 'uses a' condition for example where class A uses an instance of class B.

The composition relationship is similar to aggregation in that it contains references to other classes. The main difference is that in this case the class owns the other classes in that it is ultimately responsible for the creation and destruction of the contained classes. Composition is also said to model a 'has a' relationship for example Class A 'has a' Class B contained within it.

The next relationship in figure 4.4 is the generalisation. This represents the common object oriented concept of inheritance. Inheritance represents an 'is a' relationship e.g. Class B 'is a' type of Class A. In this case the derived class is considered to be a specialisation of the base class. One key point to be aware of is that in UML the relationship is called a generalisation so the arrow in the relationship points from the derived class to the base class as shown in figure 4.5

The fourth relationship shown in figure 4.4 is the dependency. The dependency relationship is fairly self explanatory, it indicates that any two

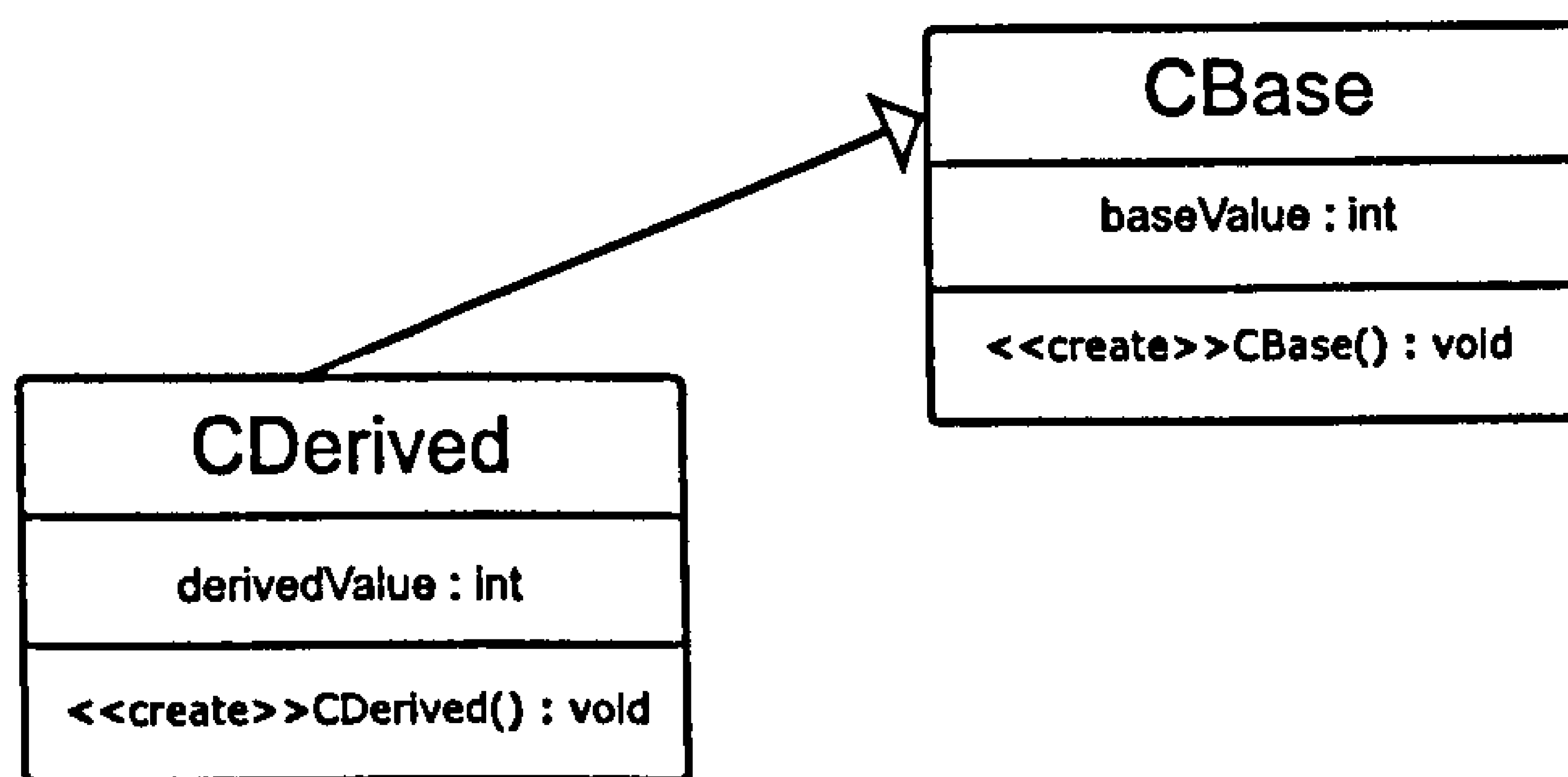


Figure 4.5: Figure showing a basic inheritance relationship between two classes.

classes are dependent on one another for some aspect of their functioning. The nature of dependencies are often indicated on UML diagrams by the use of so called stereotypes that help to further define the relationship. Stereotypes are enclosed in angled brackets e.g. $\ll$ *stereotype* $\gg$. two of the main stereotypes used in the diagrams throughout the thesis are $\ll$ *create* $\gg$ and $\ll$ *uses* $\gg$ which denote that one class creates or uses the other as indicated.

The final relationship shown is the realisation. The realisation relationship is used in cases where one class realises another, this situation occurs where a class implements a specific abstract interface that cannot be instantiated in its own right. The realisation in this situation indicates that the realising class will support the methods outlined in the interface.

One further feature of the UML diagrams used throughout the thesis is cardinality. e.g. 1..1 denotes a one to one relationship between any two classes. This is the default cardinality if no other is explicitly indicated, and hence 1..1 relationships are usually never depicted on class diagrams. Other commonly used cardinalities are 1..$*$ and $*$..1 which represent 'one to many' and 'many to one' relationships, respectively.

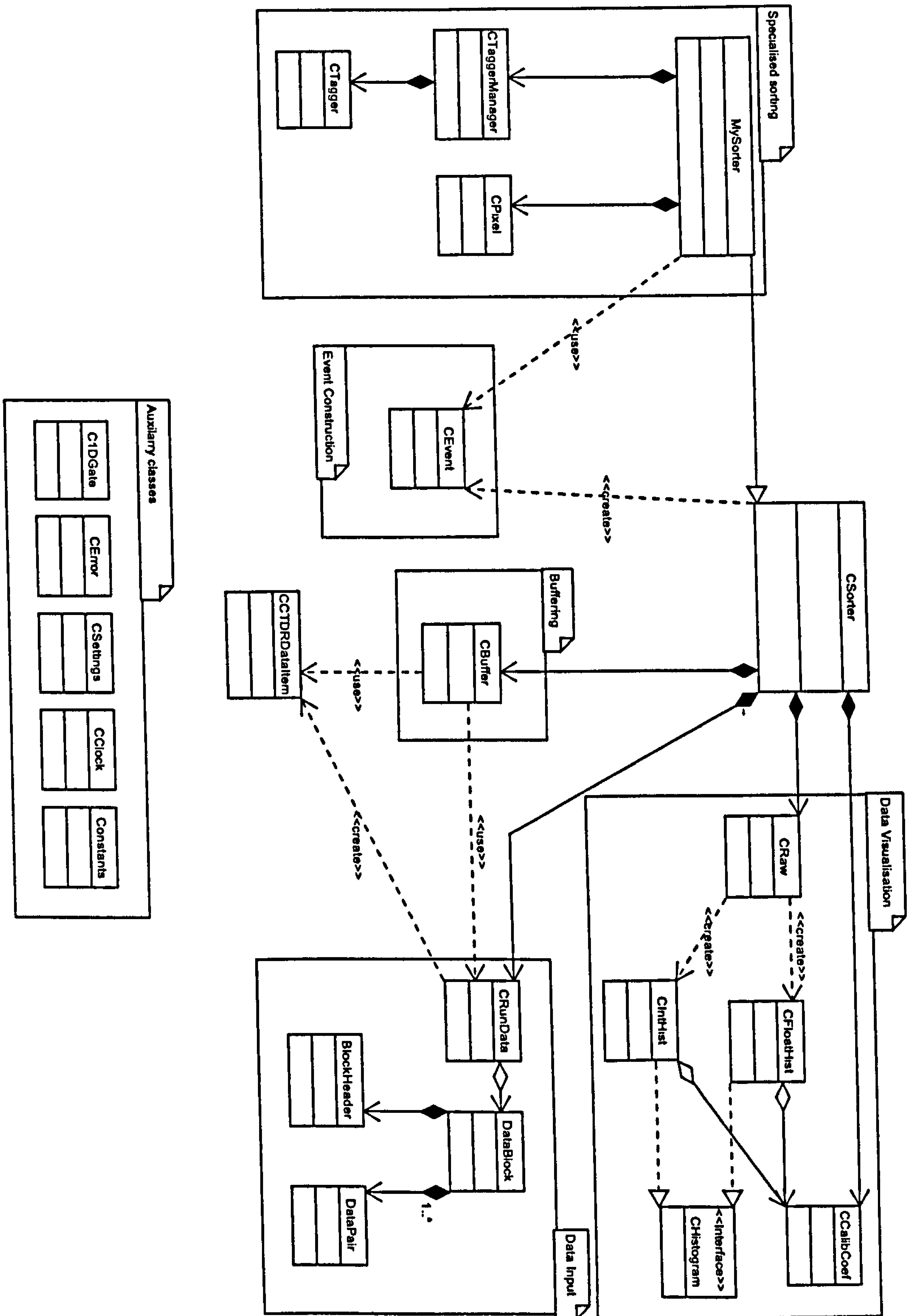


Figure 4.6: UML Class Diagram of the TDRSorter Data Analysis Code. The diagram shows how the different classes used in the code implementation are organised into the general categories illustrated in figure 4.1.

4.3.2 TDRSorter Class Diagram

Figure 4.6 is a UML class diagram showing all of the classes in the TDR sorter analysis code. The central class in the TDRSorter code is the **CSorter** class. This class is responsible for managing all the other classes in the code. i.e. it instantiates and owns all the other controlling classes for various sections of the code. As well as owning, either directly or indirectly, all of the other classes it also contains the 'main loop' of the program. The method containing the main loop is **CSorter->Run()**. Printed in figure 4.7 is an abbreviated version of the code in the main loop.

Main Loop

After the **CSorter** class has been set up the **Run()** method is called, passing in a list of run files for the program to iterate through. For every run file a **CRunData** object is created passing in the file name of the run file. As shown in figure 4.1 the code can be divided roughly into data input; data buffering; event packaging and specialised sorting sections. The created **CRunData** object encapsulates the Data Input section of the logical code diagram.

Data Input

Figure 4.8 shows a UML class diagram of the **CRunData** class. This class on creation opens the specified run file for processing. The main loop then enters a while loop that executes continuously on the condition that the method **IsData()** returns true. i.e. that there is still data remaining in the runfile. The main loop then executes a call to the **CRunData** method **ProcessFile()** which returns a **CTDRDataItem** object that can be used in the buffering stage. The **ProcessFile()** method takes care of the conversion of the raw data contained in the run file into these **CTDRDataItem** objects.

The **CRunData ProcessFile()** method reads in 16kb of data into a **dataBlock** structure that comprises of a block header and an array of 2045 data pairs as described in section 3.3.4. Once this structure has been read

```
1    for(int j=1;j<argc;j++)
2    {
3        myrunfile = new CRunData(filelist[j]);
4        while ( myrunfile->IsData() )
5        {
6            if(mybuffer->isFull())
7            {
8                this->process(mybuffer);
9                mybuffer->increment();
10               continue;
11           }
12
13           dataItem=myrunfile->ProcessFile();
14           calibration->calibrate(dataItem);
15
16           BUFFER_STATE bs = mybuffer->add(dataItem);
17           if(bs==TIME_ERR)
18           {
19               if(Tagger)
20               Tagger->clearAll();
21           }
22       }
23       mybuffer->flushBuffer();
24       delete myrunfile;
25   }
```

Figure 4.7: Figure showing the main loop sorter code. The main loop is the core of the running data analysis code. Whilst there is valid data the main loop continues to execute, managing and using the various classes of the TDRSorter program to accomplish the data analysis process.

into memory, a list of **CTDRDataItem** objects is created from the data pairs. This list contains both data and information items as described in sections 3.3.1 and 3.3.2, respectively. For each call to **ProcessFile()** the next **CTDRDataItem** is returned to the main loop. The method keeps returning **CTDRDataItem's** until the end of the list is reached, at which point it loads the next block in from the run file to create the next list of **CTDRDataItem** objects. This process continues until no more data is present in the run file at which point a flag is set so the call to **IsData()** returns false and the while loop exits to load in the next relevant run file.

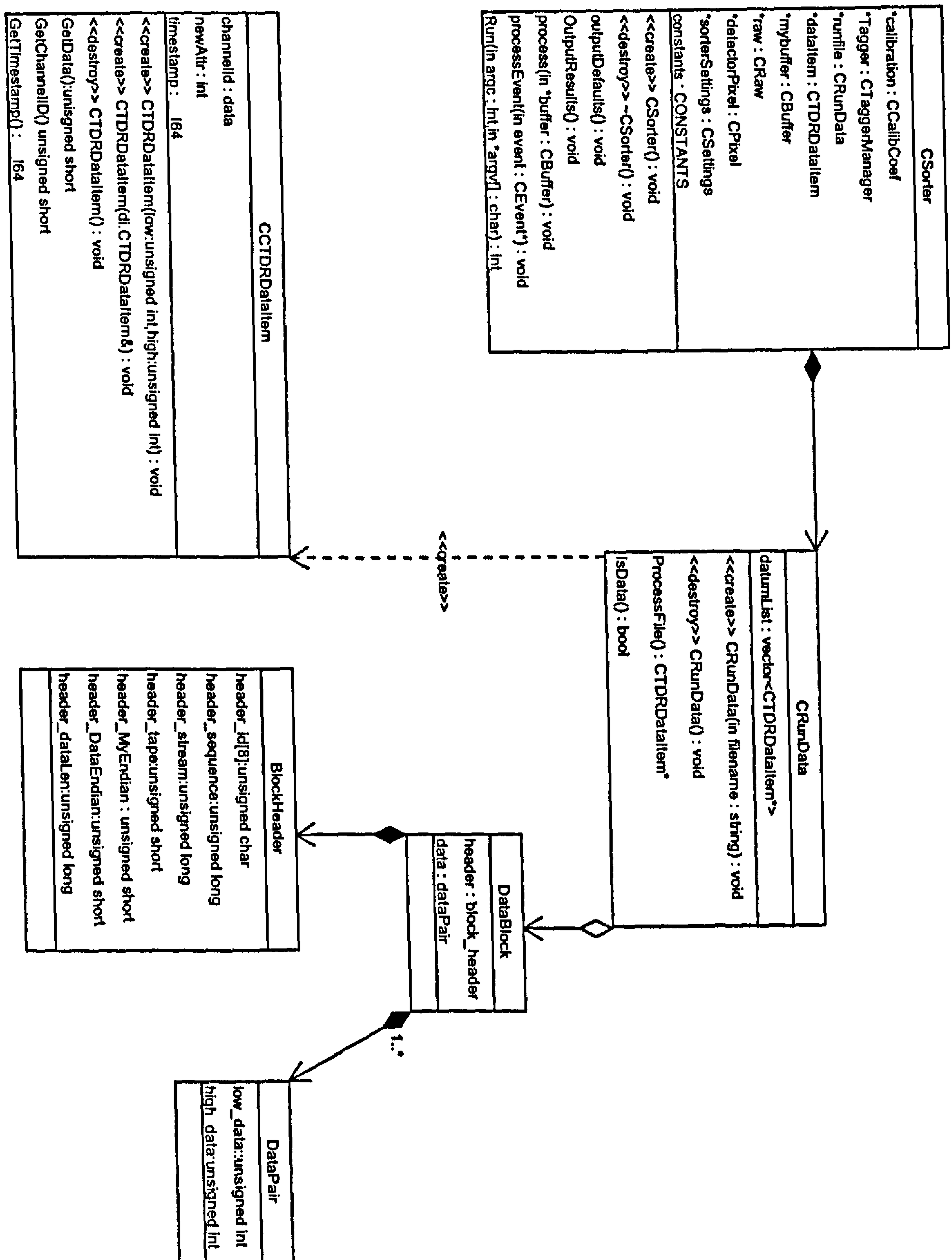


Figure 4.8: Class diagram associated with the data input section of the TDRSorter analysis code. The relationship between the **CRunData** class (which manages the data input process) and the various data block structures is shown.

4.4 Data Buffering

As indicated in figure 4.1 the next stage in the data analysis process is the data buffering section. This section is a crucial part in getting rele-

vant physics information from the time ordered data stream. The underlying buffering method needs to be able to implement the features indicated in figure 4.9. A read position needs to be maintained (i.e. the current **CTDRDataItem** being processed) as well as two buffers, one containing all **CTDRDataItems** in the specified forward time window and one containing all **CTDRDataItems** in the backward time window.

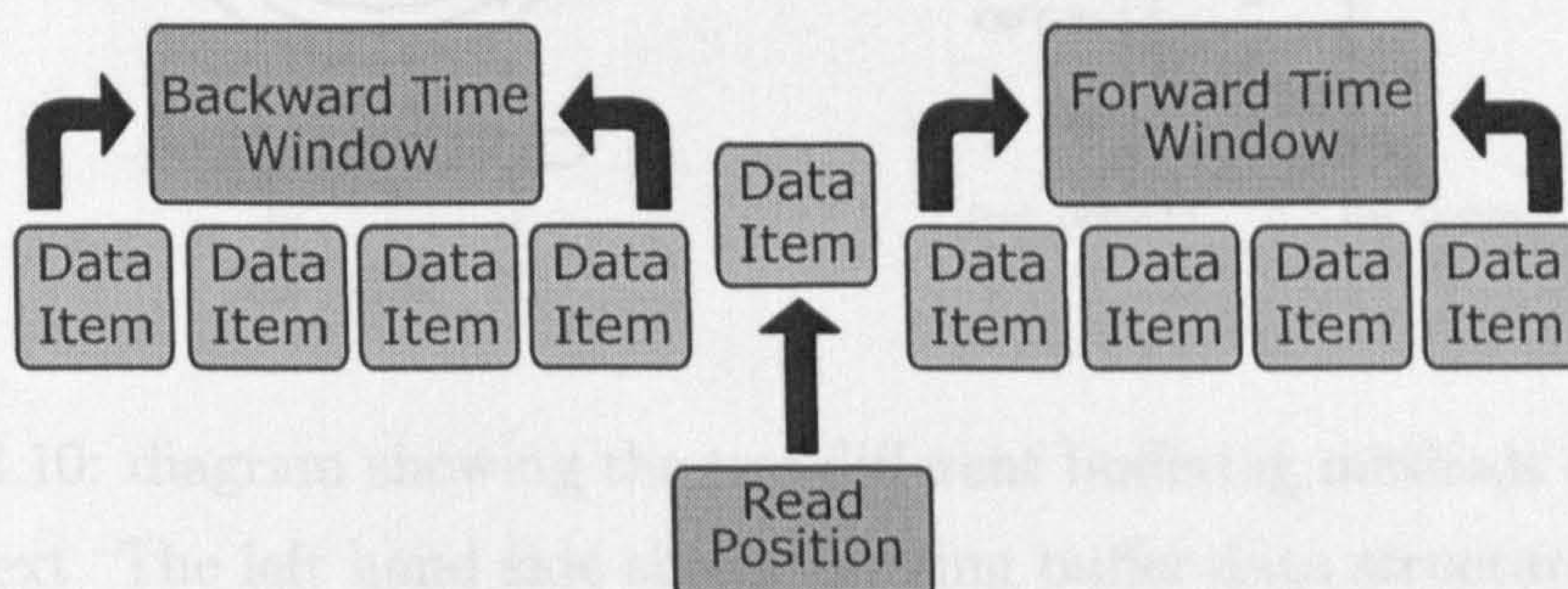


Figure 4.9: Schematic showing the general operational principles of the TDRSorter time buffer. The read position (the current data item of interest) is associated with all data items that lie within a specified time period (The forwards and backwards time windows). The time buffer is used to locate detector channels that have fired in prompt coincidence with each other.

4.4.1 Buffering Methods

Given the requirements discussed above two different underlying buffering methods were considered for the data analysis programs time buffer. The two structures considered were the static ring buffer and the double-ended queue (DE-Queue) otherwise known as a dynamic ring buffer.

Ring Buffer

The left hand side of Figure 4.10 shows the structure of a ring buffer. The ring buffer is constructed from a simple array and 3 pointers; the add data pointer the remove data pointer and the read position. The read position is the point at which data is retrieved from the buffer. When the data at

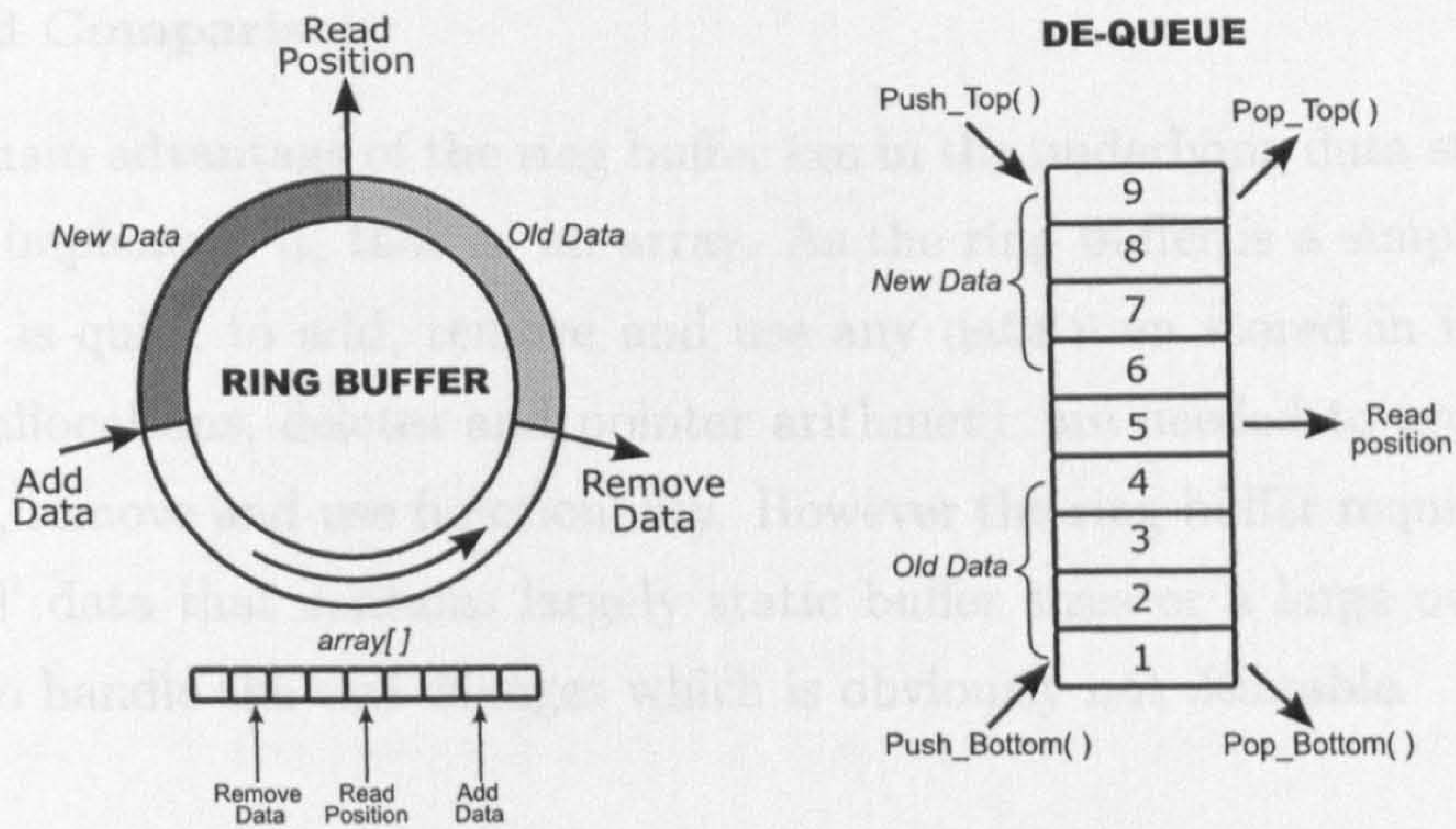


Figure 4.10: diagram showing the two different buffering methods compared in the text. The left hand side shows the ring buffer data structure whereas the right hand side shows the double-ended queue data structure.

the read position is finished with the read position is incremented i.e. the pointer is incremented to point at the next point in the array.

After an increment, any new data is added on at the position in the array assigned to the add pointer, whereas any data that is no longer required is removed from the array at the remove pointer. After any addition or removal process the associated pointers are either incremented or decremented as appropriate. Data can be added to the ring buffer up to the maximum size of the underlying array.

Double Ended Queue

The right hand side of Figure 4.10 shows the structure of the double-ended queue. The DE-Queue has four main operations that can be performed on it, **push_bottom()**, **pop_bottom()**, **push_top()** and **pop_top()**. The push and pop bottom methods are used to add or remove data items from the tail of the structure whereas the push and pop top methods are used to add or remove data items from the head of the structure.

Method Comparison

The main advantage of the ring buffer lies in the underlying data structure used to implement it, that is, an array. As the ring buffer is a simple static array it is quick to add, remove and use any data item stored in it. Only simple allocations, deletes and pointer arithmetic are needed to implement the add, remove and use functionality. However the ring buffer requires 'well behaved' data that contains largely static buffer sizes or a large over sized buffer to handle the size changes which is obviously not desirable.

Assuming that a time buffer is to be constructed that can search for $10\mu s$ forwards and backwards from the current data item an array must be constructed that has enough space for one data item per $10ns$ time-stamping 'slot' i.e. $10\mu s$ equals 1000 $10ns$ therefore an array capable of holding 2000 **CTDRDataItem** objects needs to be created. The obvious shortcoming here is that the amount of memory used remains the same regardless of whether 1 or 1000 data items are currently being stored in the ring buffer.

The major disadvantage of the ring buffer is that it has no capability to adjust in size if an increase in capacity is needed. This is an important capability because there is a possibility of high multiplicity events occurring. i.e. there may be more than one **CTDRDataitem** object for any given $10ns$ time-stamp slot in the buffer. This leads to the situation where there may be more data that falls within the required time window than it is possible to store in the array.

In contrast to the ring buffer the double-ended queues main advantage is its capability to resize itself. As data items are added or removed the data structure automatically resizes itself to take into account current requirements. This resizing reflects in the memory allocation aspects of the data structure also. As the data is added or removed, memory is allocated or deallocated, respectively. This dynamic allocation / deallocation ensures that the memory footprint of the double-ended queue accurately reflects the number of data items currently in the data structure.

The main drawback of the double-ended queue is its very ability to resize itself according to the current storage requirements. Each memory allocation or deallocation takes a specific period of time to complete. For most data sets the number of data items currently required will vary frequently and hence memory will need to be allocated and deallocated frequently. This frequent memory activity will cause a slow down in the operation of the buffer. This constant memory activity can be mitigated to a certain extent by requiring that the size of the double-ended queue is given a minimum value so that small changes in size does not result in extraneous memory activity.

Given the associated strengths and weaknesses of the data structures given above and that the main disadvantage of the double-ended queue can be compensated for to a certain extent, it was decided that the double-ended queue would be used to implement the time buffer outlined in figure 4.9.

4.4.2 Time Buffer Operation

This section deals with how the time buffer, implemented with a double-ended queue, operates. Figure 4.11 shows a UML class diagram of the **CBuffer** class and how it relates to the other classes in the TDRSorter data analysis code. The **CBuffer** class implements the functionality of the time buffer as outlined in figure 4.9.

Referring to the main loop in figure 4.7 there are two sections of code relevant to the operation of the **CBuffer** class. The first section is the line **BUFFER_STATE bs = mybuffer->add(dataItem)**. This line of code adds the CTDRDataItem returned from the CRunData object as described earlier to the buffer.

Figure 4.12 shows the operation of the add method of the **CBuffer** class. The add method passes in a pointer to a **CTDRDataItem** object that is to be added to the buffer. Before any processing is performed a check is performed to make sure that the time-stamp of the data item being added

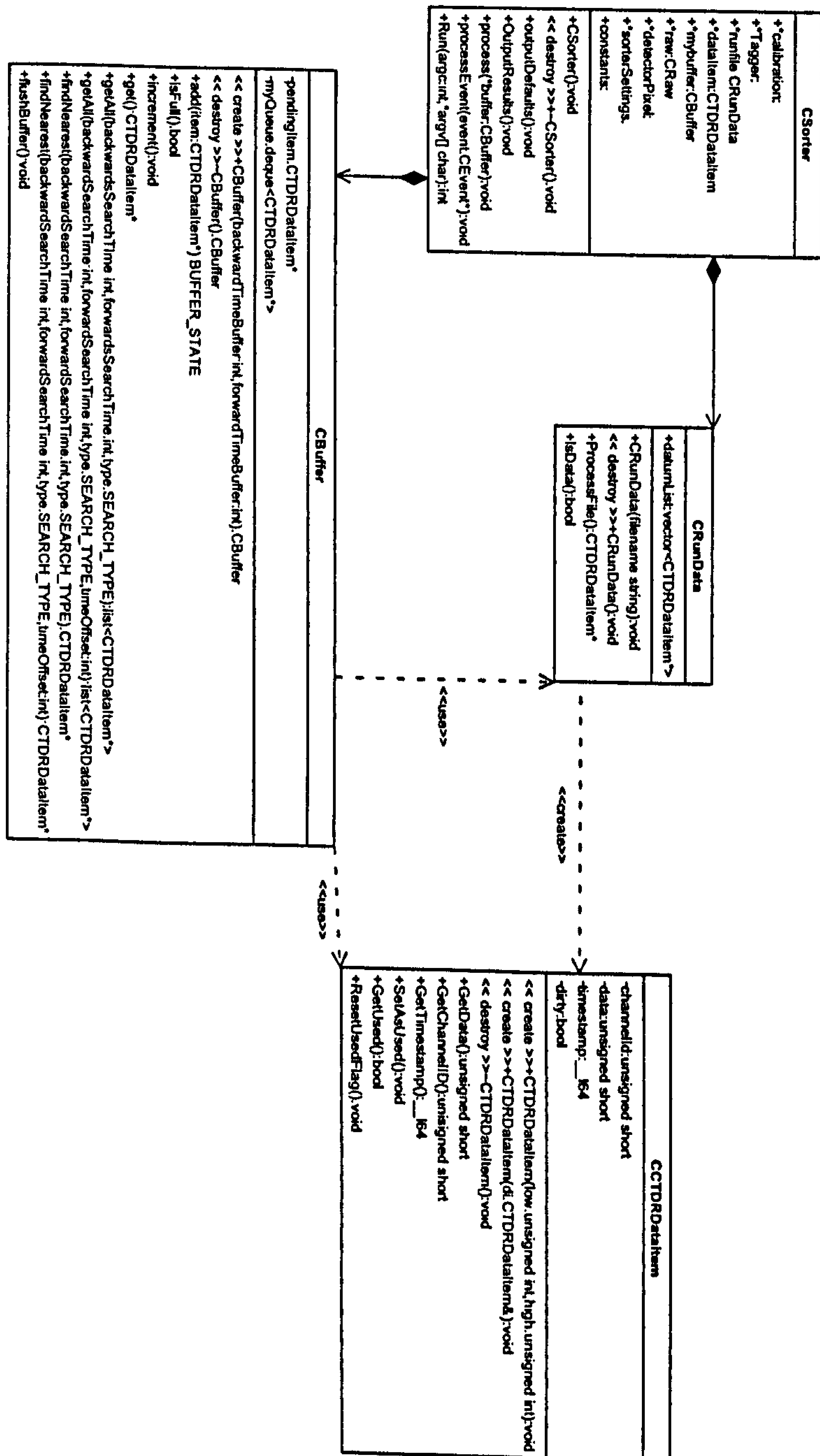


Figure 4.11: UML Class Diagram showing the **CBuffer** class and the relationships between the various supporting classes.

makes sense (i.e. is in correct time order). The basic requirement for the time-stamp of the added item is that it is equal to or greater than the time-stamp of the last data item added to the buffer. This check is necessary as

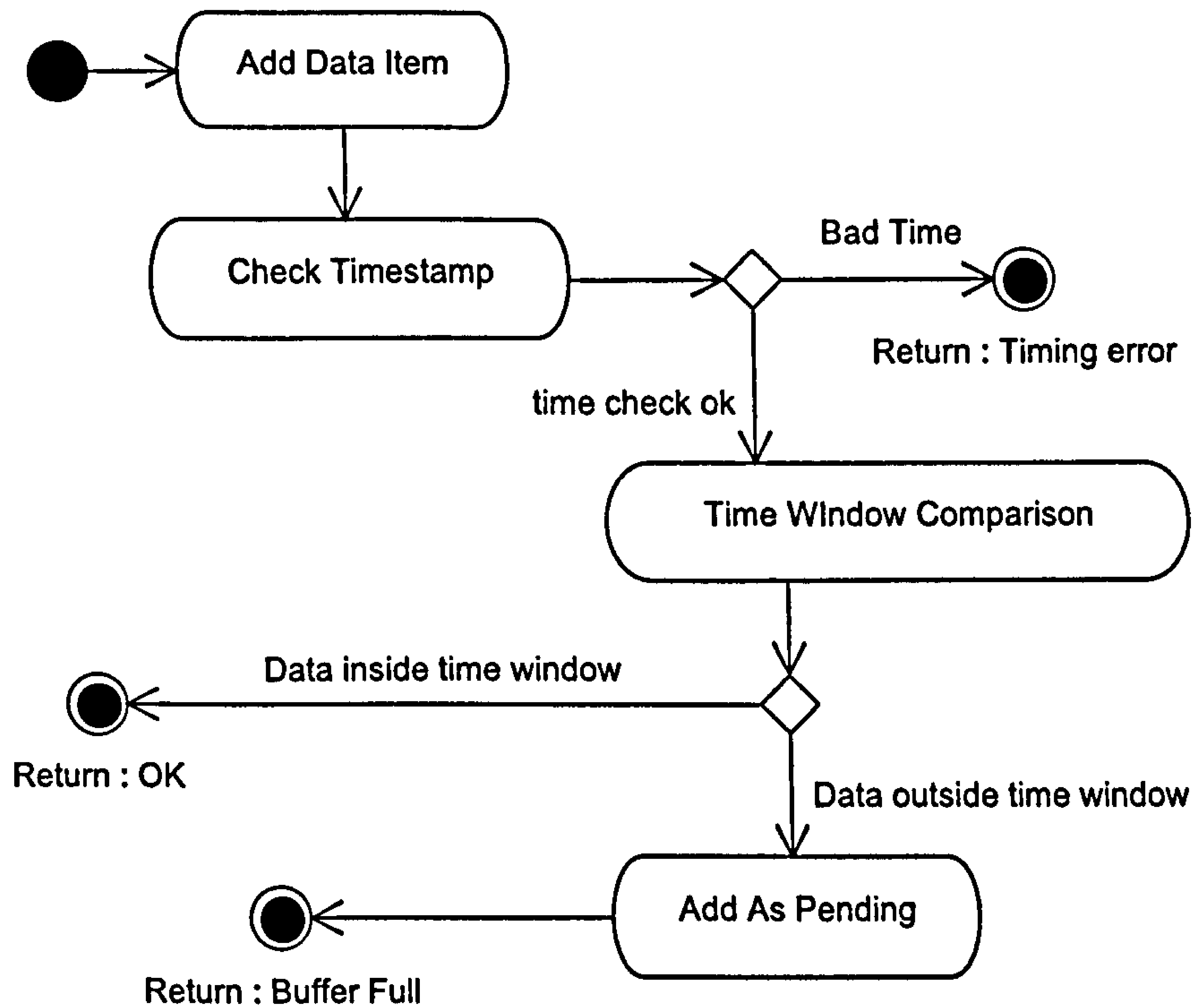


Figure 4.12: Figure showing the operation of the add method of **CBuffer** class. The figure shows the logic of the process indicating how the data item added alters the internal state of the buffer i.e. when the buffer is fully constructed and contains valid data the buffer state is set to full, indicating that the **CBuffer** object is ready to use.

it ensures that the time buffer remains in strict time order. The data being in time order is a fundamental assumption of all further processing.

If the time-stamp fails the time-stamp test described above, the add function returns a buffer state variable indicating that a timing error has occurred. This then allows the calling code to handle the condition and finish any processing necessary and to also clean up and release any necessary memory including flushing the time buffer. Timing errors in the buffer are not a fatal condition, flushing the time buffer and then rebuilding it from the next item in the data stream is all that is necessary to recover and continue

processing. At any point where a timing error[5] occurs the user is notified by an entry being placed in a program global log file that is created whenever the TDRSorter code is run.

If the time check is passed then the next stage of the add method is to compare the time-stamp of the data item being added to see if it fits within the backwards time window of the buffer. If the time-stamp of the data item does fall within the range specified by the buffer it is added to the end of the DE-Queue structure by using the **push_bottom( )** operation it defines. For example if a buffer is specified with a data item at the read position with an arbitrary time-stamp of $1000\mu s$ and a backwards time window of $100\mu s$. Any data item that has a time-stamp that falls in the range of $900\mu s$ to $1000\mu s$ inclusive is considered to be within the backwards time window of the buffer. If the data item is added the add method returns with a buffer state of 'OK' indicating that more data can be added.

If the data item being added to the buffer falls outside the backwards time window then the buffer is now considered to be full. When the buffer is full it means that it is fully constructed and ready to be used i.e. it contains valid data. It does not mean that the capacity of the buffer has been used and no more data can be added. Carrying on with the example above if the data item being added had a time-stamp that had a time $< 900\mu s$ it would be considered to have fallen outside the backwards time window of the buffer. This data item is now classified as the buffers pending item and essentially is the next item to be added when the buffer is no longer considered to be full. At this point the add method returns with a buffer state of 'FULL' indicating that the time buffer is read for use in any future analysis.

The second section of code from the main loop that is relevant to the time buffer is the line starting '**if(mybuffer->isFull())**'. This line, executed before another data item is added to the buffer, checks to see whether the buffer contains valid data. If the buffer does contain valid data it is passed

[5] Or indeed any other error the user needs to be made aware of.

as an argument to the process method of the **CSorter** class where event construction takes place. Event construction is detailed in the next chapter.

Following the event processing the buffer is incremented. This incrementation moves the read position of the time buffer to point at the next data item. The incrementation process results in a resizing of the buffer with any data items that now lie outside the time window being removed from the buffer. The call to the process method and subsequent incrementation continues whilst the buffer contains valid data. At any time the incrementation process and subsequent removal of data items from the buffer can alter the state of the buffer to be no longer full i.e. it no longer contains valid data, at which point data items are added again until the time buffer is full again.

Figure 4.13 shows the detailed operation of the increment method of the buffer class. The first stage is to perform the initial increment of the read position of the buffer. This operation simply makes the read position point to the next item in the time buffer i.e. the next youngest data item. This incrementation now invalidates the rest of the buffer and all remaining data items need to be retested to see if they are still within the specified front and backward time windows.

The next stage is to check the time-stamp of the oldest data item in the time buffer. If the data item's time-stamp indicates that the data item falls outside the backwards time window i.e. the data item is too old, then the data item is removed from the time buffer and the memory allocated to this data item is freed as it is no longer of interest. This process of checking the oldest items time-stamp and removing old items is continued until one of the checks finds a data item that lies within the backwards time window. This data item is the oldest data item in the data stream that is still of interest.

At this point the backwards portion of the buffer is now valid, but the front portion is not. In order to see if more data is needed to fill the buffer, the time-stamp of the pending item is checked to see if it lies inside or outside the

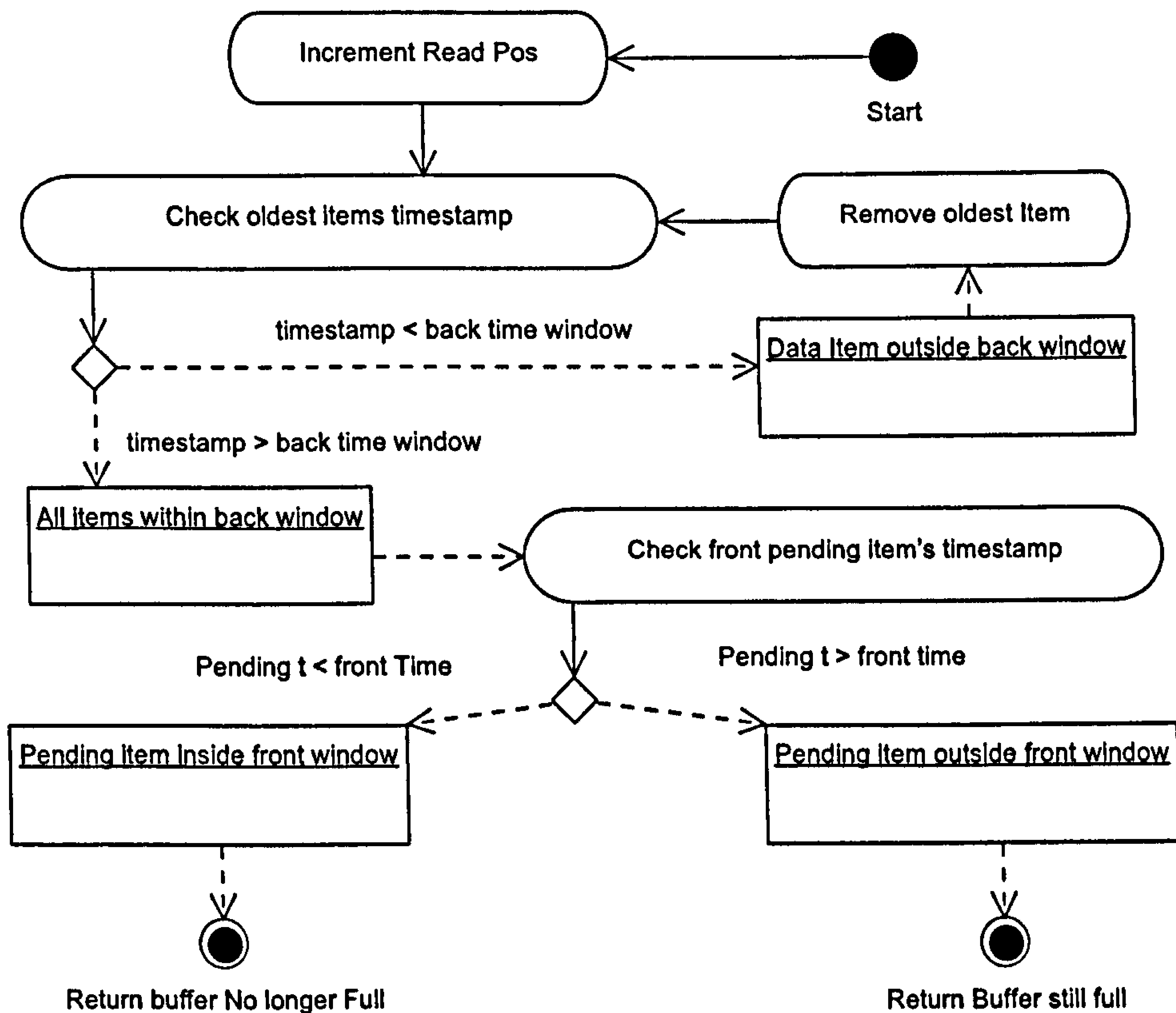


Figure 4.13: Figure showing the logical operation of the increment method of **CBuffer** class. This operation increments the read position of the time buffer and forces the entire buffer to be revalidated. This revalidation can cause the buffer state to be changed.

forward time window. If the time-stamp of the pending item is greater than the front time i.e. the pending item is outside the forward time window, then the buffer still contains valid data. If the pending items time-stamp indicates that the data item lies within the forward time window then the buffer is no longer full and more data items need to be added before it can be used. The increment method returns a buffer state variable indicating whether the buffer is still full and can be used, or if it is not full and more data needs to be added.

4.4.3 Buffer Operation Walk Through

As the individual stages of data buffering has been discussed in some detail it is now useful to look at how all of these sections fit together in relation to the buffering of a real data stream. Figure 4.14 shows how the increment and add operations work on an example data stream. Along the top of the diagram, increasing in time from left to right is a time ordered sequence of data items as they appear in the data stream and is essentially the order of data items as they are read from the data source.

This walk through of the buffering process is based on building a time buffer with a forward and backwards time window of $20ns$. The buffering starts at the read position indicated in figure 4.14 (at the $30ns$ position) on the first line where there is already a complete time buffer in place with a single pending data item containing information about a γ ray. After this buffer has been passed on for further analysis an increment operation is needed to revalidate the buffer. The result of this incrementation is given in the second line of the figure.

As described in section 4.4.2 the read position is first moved to the following data item in the data stream which in this case is the data item at the $40ns$ position. Firstly the back window is checked and any items that lie outside it are removed which in this case is the γ ray data item at the $10ns$ position, secondly the time-stamp of the pending data item is checked against the forward time window. In this stage the pending data item still lies outside the time window so the time buffer is still valid and is passed on for subsequent analysis.

Following this as the buffer still contains valid data another increment is performed with the results showing in part three of figure 4.14.In this line the data items out of time scope in the backwards time window are again cleared up. The time-stamp of the pending item is checked and as in the previous stage the pending item is still outside the forward time window. This data

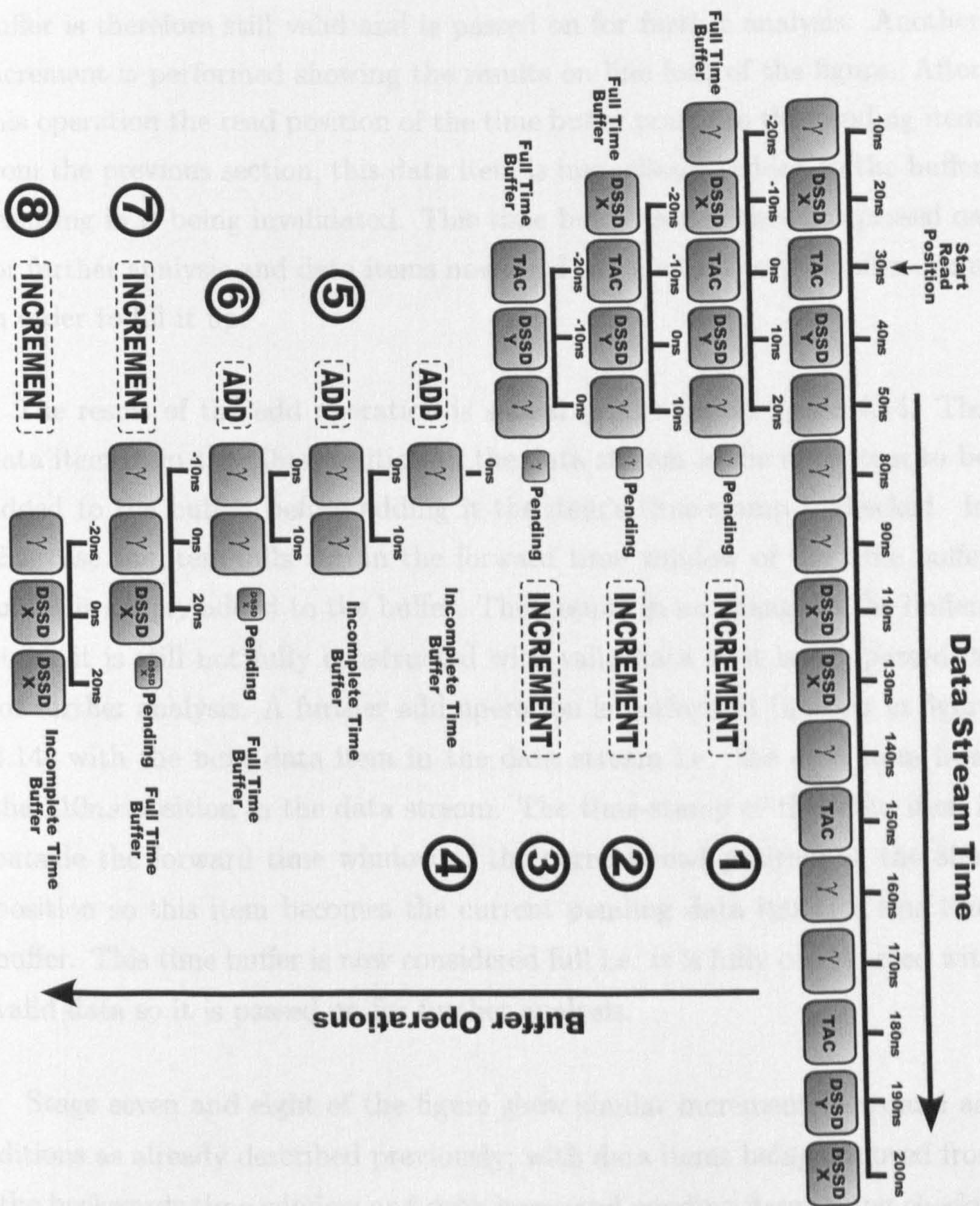


Figure 4.14: Figure showing how the increment and add buffer operations work on an example section of data stream. The resultant time buffers for successive iterations of the operations are shown, indicating the internal state of the time buffer. i.e. indicating whether the buffer is fully constructed and can be used for further data analysis.

buffer is therefore still valid and is passed on for further analysis. Another increment is performed showing the results on line four of the figure. After this operation the read position of the time buffer points to the pending item from the previous section, this data item is immediately added to the buffer resulting in it being invalidated. This time buffer is therefore not passed on for further analysis and data items now need to be added to the buffer again in order to fill it up.

The result of the add operation is shown in line five in figure 4.14. The data item from the $90ns$ position in the data stream is the data item to be added to the buffer, before adding it the item's time-stamp is checked. In this case the item falls within the forward time window of the time buffer and it is simply added to the buffer. This results in no change in the buffers state, it is still not fully constructed with valid data so it is not passed on for further analysis. A further add operation is performed (line six in figure 4.14) with the next data item in the data stream i.e. the data item from the $110ns$ position in the data stream. The time-stamp of this data item is outside the forward time window of the current read position at the $80ns$ position so this item becomes the current pending data item for this time buffer. This time buffer is now considered full i.e. it is fully constructed with valid data so it is passed on for further analysis.

Stage seven and eight of the figure show similar incrementations and additions as already described previously; with data items being removed from the backwards time window and data items and pending items being checked against the forward time windows range. This process essentially continues throughout the data stream constructed from all of the runfiles passed into the TDRSorter data analysis code. As valid time buffers are constructed they are passed into the **process()** method of the **CSorter** class where pixels and events are constructed. The process of pixel definition and event construction are described in detail in chapter 5

4.4.4 Example Time Buffer

One final illustration that is useful at this point is to present an example of a complete, fully constructed time buffer as it would appear in the TDRSorter data analysis code. The time buffer presented here is an actual example of one that was created during a run through the example data set. All of the data items shown are simplified versions (containing only key information) of the real data items used in the construction of the time buffer. Figure 4.15 shows this outline of the fully constructed CBuffer object.

```
START BUFFER TIME DUMP
 INDEX 0  CHN 344  TIME 20739355984741  DATA 16364  ENERGY 15030.88  CLOVER
 INDEX 1  CHN 336  TIME 20739355984777  DATA 16287  ENERGY 14982.04  CLOVER
 INDEX 2  CHN 320  TIME 20739355984778  DATA 16299  ENERGY 14971.08  CLOVER
 INDEX 3  CHN 416  TIME 20739355985197  DATA 3343   ENERGY -1        GAS_X1
 INDEX 4  CHN 417  TIME 20739355985197  DATA 11732  ENERGY -1        GAS_X2
 INDEX 5  CHN 418  TIME 20739355985197  DATA 10840  ENERGY -1        GAS_Y1
 INDEX 6  CHN 419  TIME 20739355985197  DATA 9232   ENERGY -1        GAS_Y2
 INDEX 7  CHN 420  TIME 20739355985197  DATA 4387   ENERGY -1        GAS_E
 INDEX 8  CHN 128  TIME 20739355985199  DATA 16246  ENERGY 16985.3   DSSD_X
 INDEX 9  CHN 59   TIME 20739355985199  DATA 16255  ENERGY 4242.9    DSSD_Y
 INDEX 10 CHN 432  TIME 20739355985209  DATA 7672   ENERGY -1        SI_GAS_TAC

       PENDING 336 DATA 16238 TIME 20739355999552 CLOVER

END BUFFER TIME DUMP
```

Figure 4.15: Outline showing a real fully constructed time buffer (CBuffer object) created during a run through the example data set.

It can be seen from the data dump of the buffer that the time buffer contains three clover events; the x,y and energy outputs of the MWPC; a DSSD X and Y data item; and the triggering silicon-gas TAC. The quoted energy value for the gas and the TAC data item values are indicated as -1 meaning that there is no energy value because there is no calibration supplied for these channels. There is no energy calibration supplied as an energy value for these detector channels is either meaningless or unneeded. Another point to note is the energy values of the DSSD X and the DSSD Y data items. In the data set used the X and Y sides of the DSSD had different gain ranges with the Y side being approximately one quarter of the X side. Even though at first glance the X and Y data items appear to not be caused by the same

event, in reality the energy values are quite close and the data items would probably pass any required gates to define a real pixel if the gain ranges were the same.

4.5 Summary

This chapter covered the high level structure of the TDRSorter data analysis code and detailed how it can be broken down into four general sections. The first two sections; data input and data buffering were covered in this chapter, the remaining sections are covered in later chapters. Two different methods of buffering were also compared and an outline of the operation of the time buffer was given. A walk through of the entire process was also given at the end of this chapter.

Chapter 5

Pixel Definition and Event Construction

5.1 Overview

This chapter discusses two important sections of the TDRSorter data analysis code; pixel definition and event construction. Pixel definition is an important aspect of the analysis code as it is effectively the central building block from which all other data structures are formed. A discussion is given of how pixels are defined from fully constructed time buffers and some issues that arise during this process. The next major section discussed is that of Event construction. This section will detail how **CEvent** objects are constructed using the **CPixel** class previously discussed. The **CEvent** object encapsulates the fundamental 'physics' data that is used in particular in the specialised sorting portion of the data analysis code as discussed in chapter 6.

The process of pixel definition and event construction takes place entirely within the **process()** method of the **CSorter** class as outlined in figure 5.1. The process method is called from the main loop line **this->process(mybuffer)**. The **mybuffer** object passed as a parameter is the fully constructed **CBuffer** class as described in section 4.4. The read posi-

tion points to the current triggered data item and the forward and backward portions are filled with all **CTDRDataItem** objects from the data stream that have timestamps that lie within these given time periods

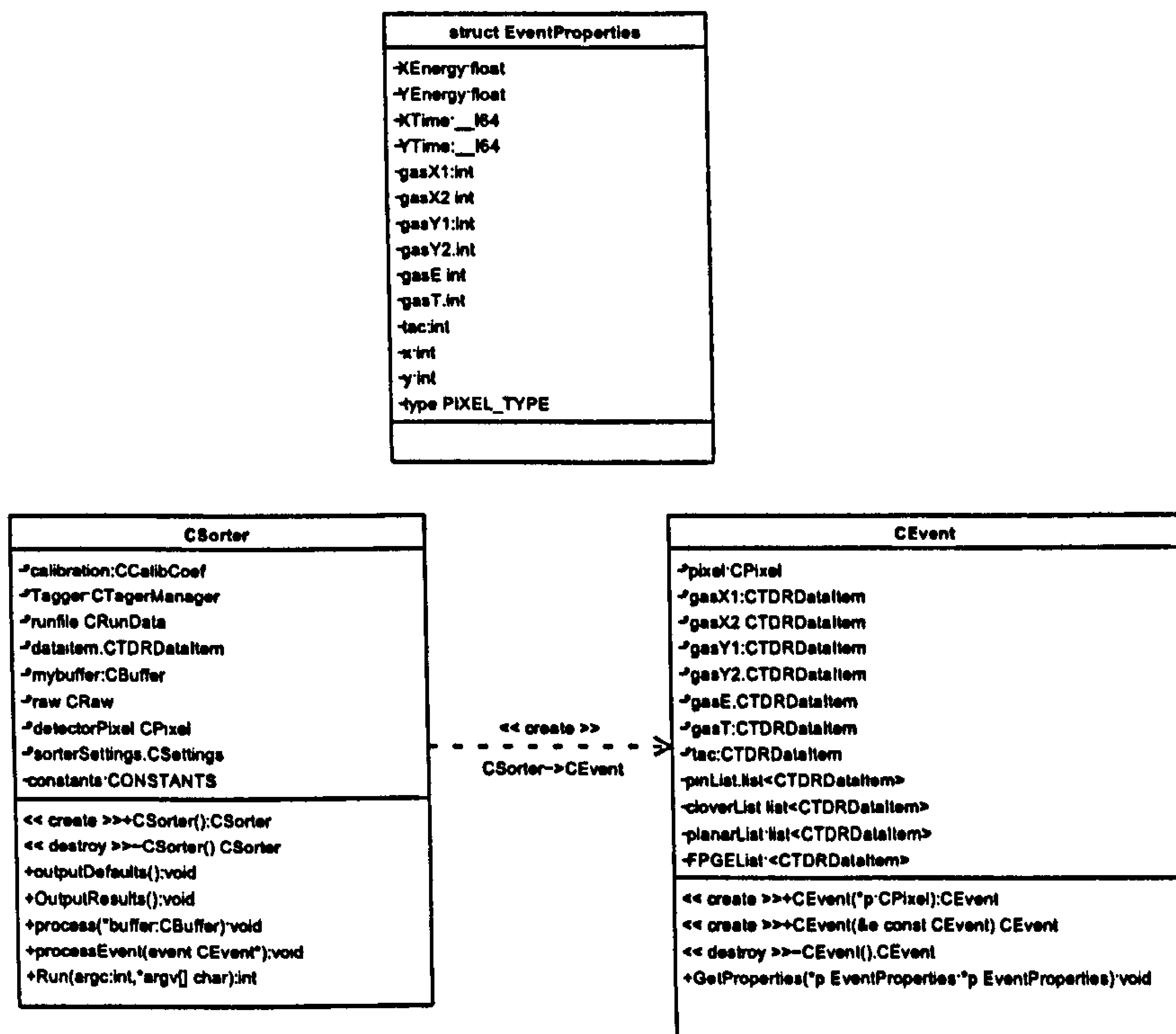


Figure 5.1: UML Class Diagram showing the **CSorter**, **CEvent** and **EventProperties** classes.

The main purpose of the event construction process is to build valid **CEvent** objects that can be passed onto the specialised sorting section of the code. The **CEvent** class contains basic information about data items that are in prompt coincidence; pixel information; and detector energy information. The following sections will discuss how these **CEvent** objects are built and some of the most important issues that arise.

5.2 Pixel Definition

The core of constructing a **CEvent** is the definition of a CPixel object that must be passed as an argument in the constructor of the **CEvent** object. As outlined in section 2.4.3 the double sided silicon strip detector (Or DSSD) consists of sets of individual detector strips positioned orthogonally to each other. These sets of strips are termed the x and y strips of the DSSD. The same principles discussed in the following sections are applicable to the planar germanium detector which is segmented into strips in a similar way to the DSSD. Other detectors in the spectrometer such as the PINs which are not segmented into strips do not need the following processes to be used effectively.

The DSSD detector has both x and y oriented strips to gain spatial information from any signals produced from recoil implantations or subsequent decays. Using the example of a recoiling nucleus implanting in the DSSD detector, energy will be deposited in the nearest x and y strips. As the position of these strips within the overall detector is known an (x, y) coordinate for this implantation can be inferred.

Given this information it can be seen that in order for a pixel to be defined a **CTDRDataItem** corresponding to a DSSD X strip and a **CTDRDataItem** corresponding to a DSSD Y strip need to occur in the data stream. Figure 5.2 shows diagrammatically the process of defining a pixel using the x and y **CTDRDataItem** objects.

Apart from the presence of these data items in the data stream there are also time and energy constraints to consider. In order for a pixel to be considered valid both x and y data items need to lie within a short time period of one another. Section 5.3 provides detailed information about timing relationships between strips. If however it is assumed that all detector and data acquisition paths should operate at approximately the same speed, and that the energy is deposited from a single implantation event it is highly

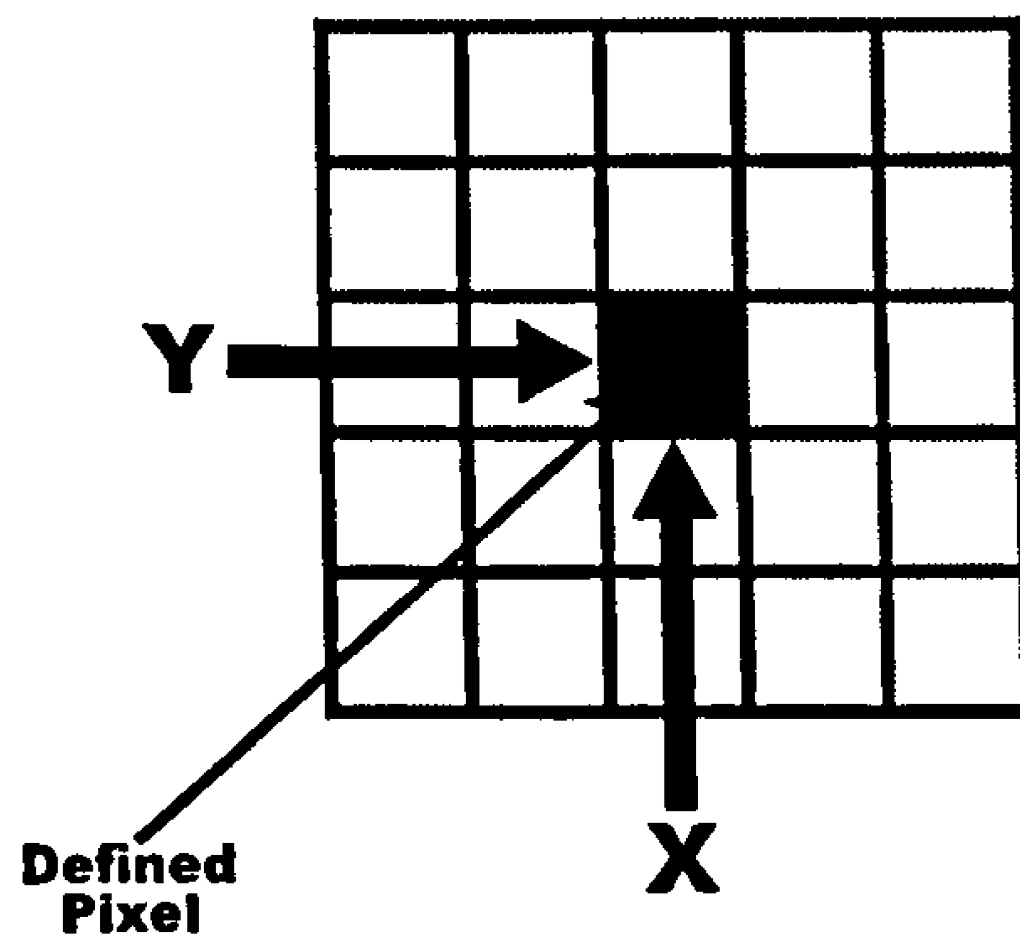


Figure 5.2: Diagram showing the principles of pixel definition. The diagram shows how a pixel is constructed from both a DSSD X data item and a DSSD Y data item.

likely that any x and y data items in close temporal proximity in the data stream could define a valid pixel.

As well as lying in close temporal proximity, any data item should also pass certain energy conditions to be considered a true pixel. The energy condition is that the magnitude of the energy deposited in a given x or y strip should closely match that of its respective x or y channel. This condition is put in place as the energy deposited in each strip should vary around a mean value. If the energy of one strip in the pixel was $4000keV$ and its corresponding channel was $40keV$ then there is the possibility that the smaller energy strip could be a false coincidence as the result of noise or background radiation in the detector.

5.2.1 Time and Energy Condition Statistics

Figure 5.3 shows the timing relationship between DSSD X and DSSD Y strips. This figure was generated by searching through the data stream and triggering from either a DSSD Y or a DSSD X strip. When one of these strips was found a search was performed to locate the nearest complimentary x or y strip. When this strip was found a difference was taken between the timestamps of the respective strips data items and the result was plotted

onto the figure.

The x axis of figure 5.3 represents this time difference in nanoseconds. As can be seen the distribution lies around the zero position and quickly falls off at either side. This indicates that most x and y strips in the data stream that are near to each other probably were generated from the same event. In order to reduce any background or stray signals on future analysis it is possible to set a time gate on strips that are likely to be correlated. It can be seen that if the time difference between strips is greater than $100ns$ then they are unlikely to be valid strips for defining pixels. In reality a tighter gate can be used to ensure only good is used (i.e. a time gate of $\pm 20ns$). It must be noted that such gates must be recalculated for different experiments and different set-ups as timing relationships are likely to vary.

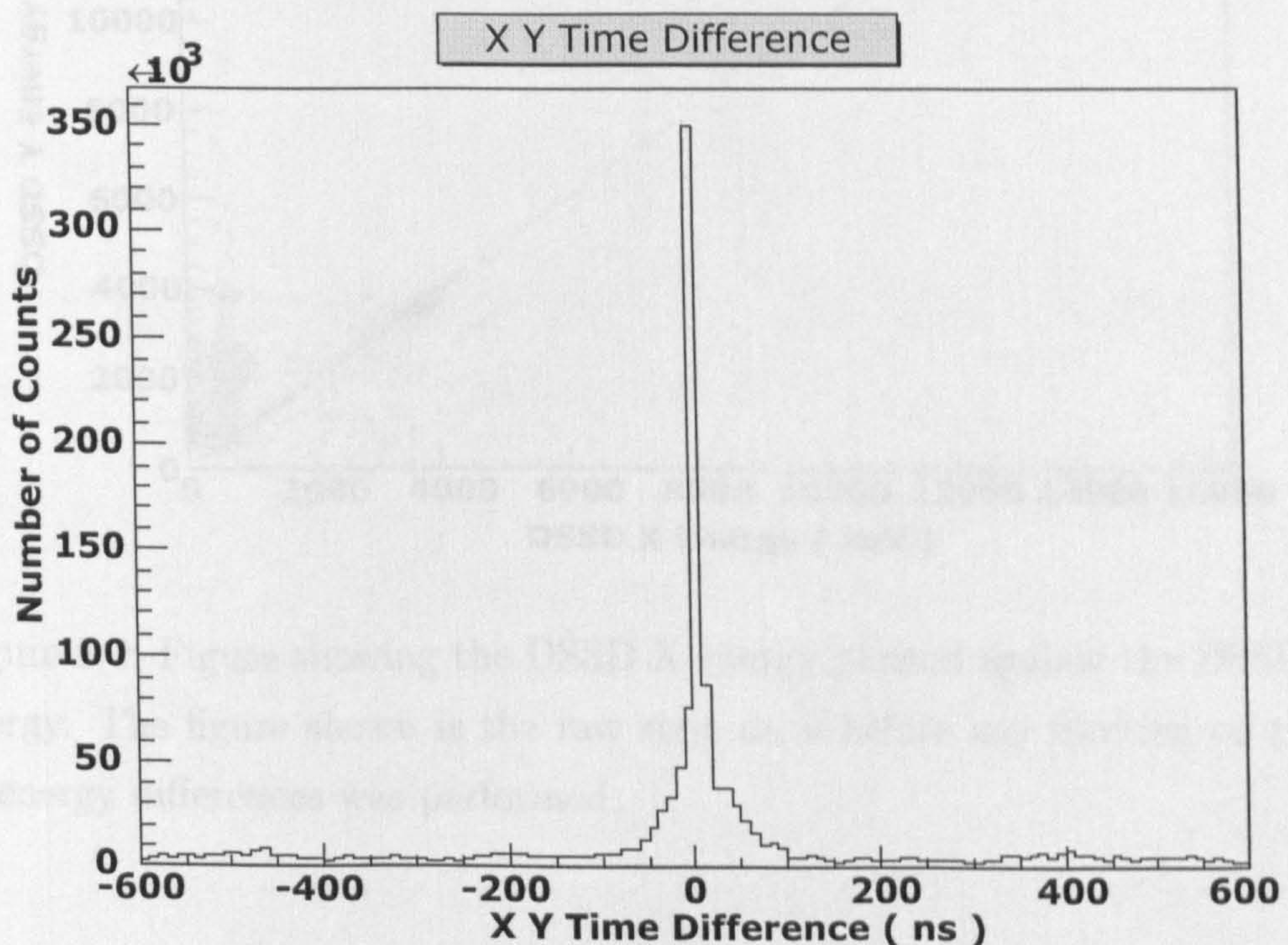


Figure 5.3: Figure showing time difference between x and y strips in the data stream.

As well as filtering out widely different timestamps it is also useful to filter out located DSSD X and Y strips that have large energy differences. These energy differences could be caused by various problems in the system e.g. malfunctioning strips in the DSSD detector, or false coincidences with noise generated by a DSSD strip. Other sources of large energy differences could be caused by recoils embedding only partially on a strip of the DSSD. Figure 5.4 below shows a plot of the energy of the DSSD X strips against the energy of the DSSD Y strips.

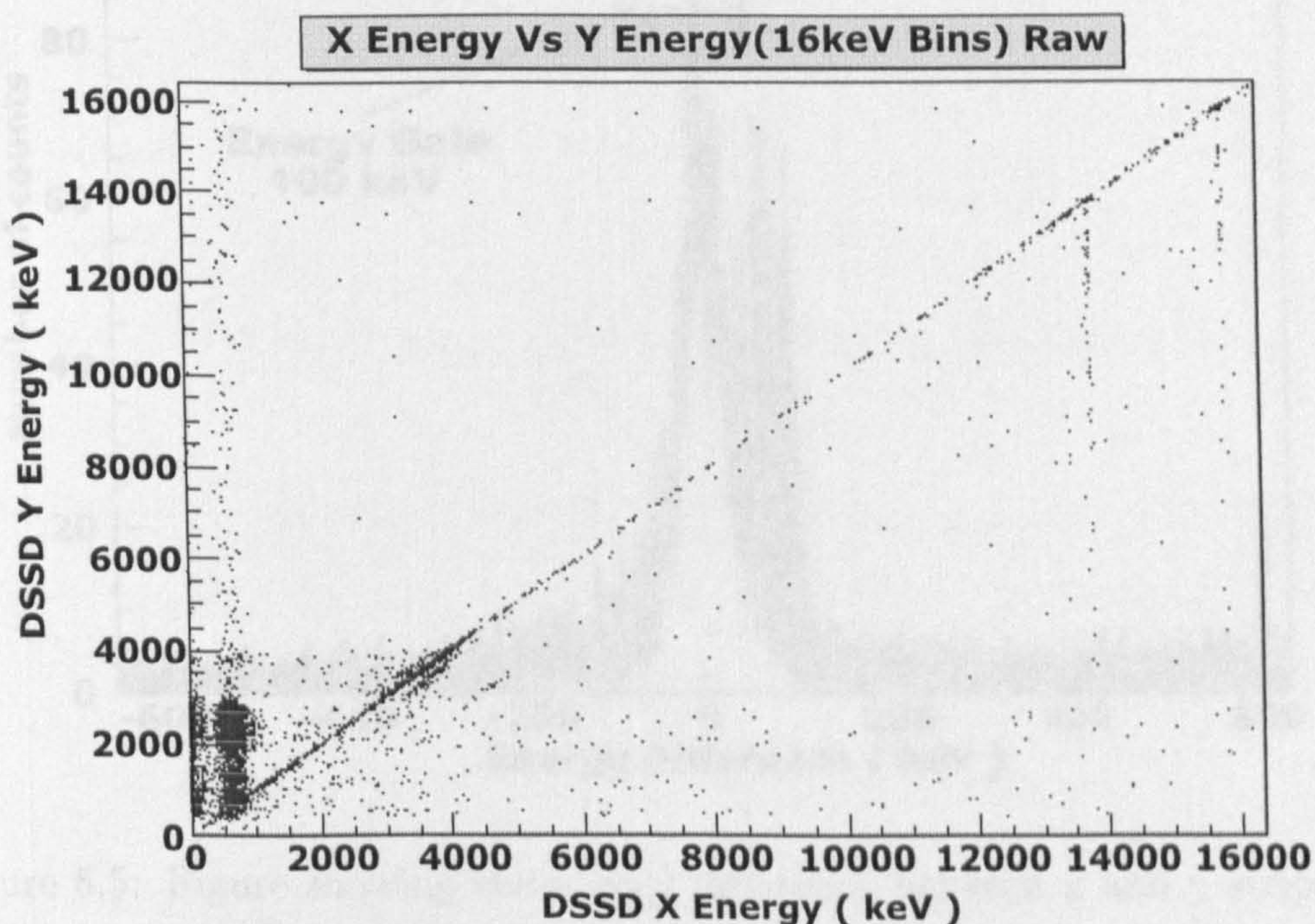


Figure 5.4: Figure showing the DSSD X energy plotted against the DSSD Y energy. The figure shown is the raw strip data before any filtering on time or energy differences was performed.

On figure 5.4 a straight line corresponding to DSSD X and Y data items that have identical or nearly identical values can be clearly seen. This main line represents the data items that need to be selected, all other data items represent less than ideal data items for pixel construction. Other structures on this plot are also visible. What appear to be horizontal or vertical lines

on this plot correspond to areas where strips are dead in one axis. The large dense patches near the zero position correspond to low end noise generated in the DSSD strips. In order to decide on how to filter out these unwanted data items it is useful to look at the energy difference between the x and y data items.

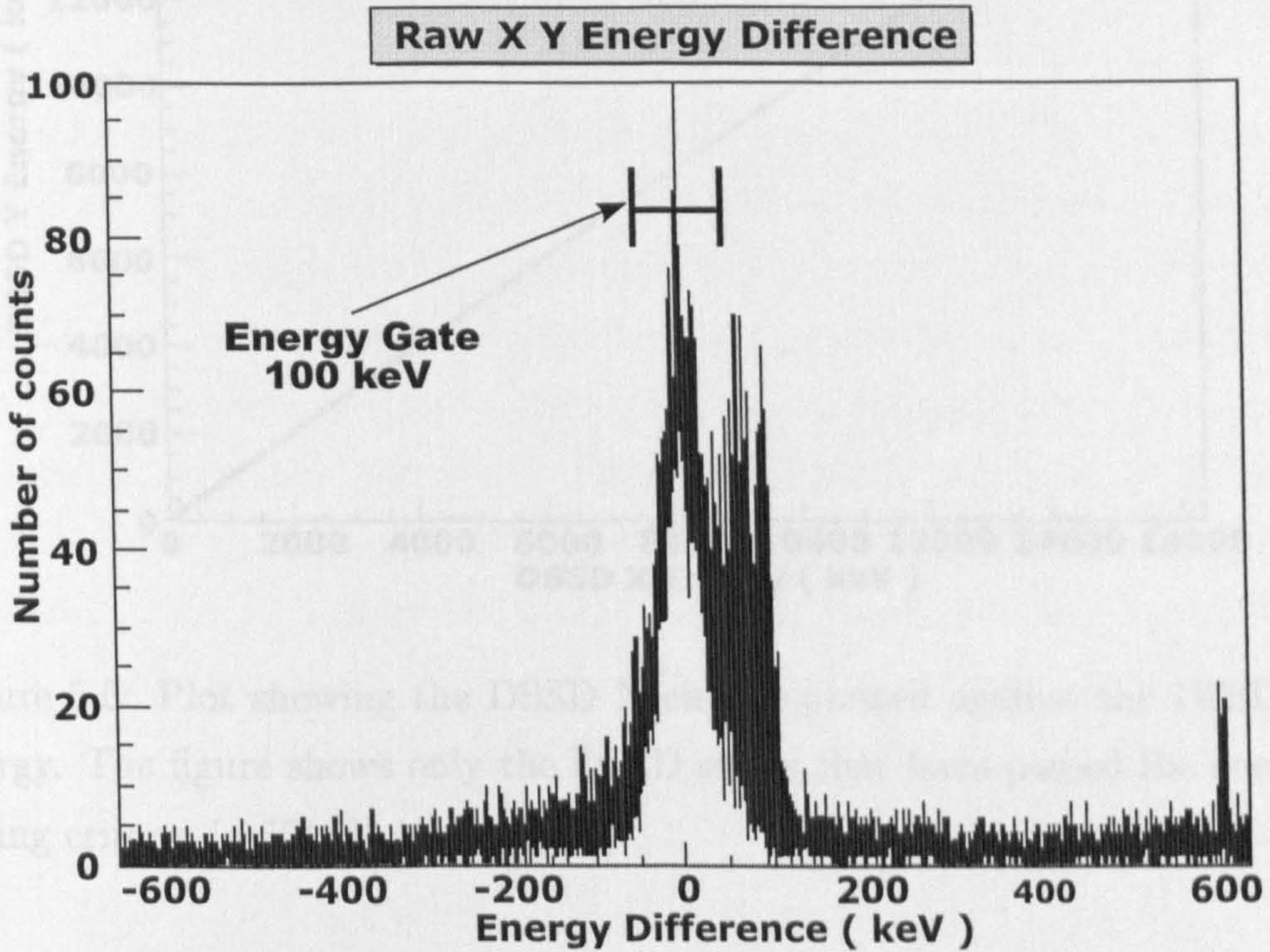


Figure 5.5: Figure showing the energy difference between x and y strips in the data stream. A 100 keV energy gate is indicated, the results of this gate on the x vs y energy plot can be seen in figure 5.6.

Figure 5.5 shows a plot of the energy differences between the DSSD X and Y strips. A spike can be seen around the zero position which corresponds to data items that have very similar x and y energies. By setting a gate of $\pm 50keV$ and again plotting the DSSD X energy versus the DSSD Y energy as in figure 5.6 it can clearly be seen that the only feature that remains is the straight line that corresponds to the data items used to build pixels from. All other artifacts have been removed by gating on only small energy differences

between the DSSD X and Y strips.

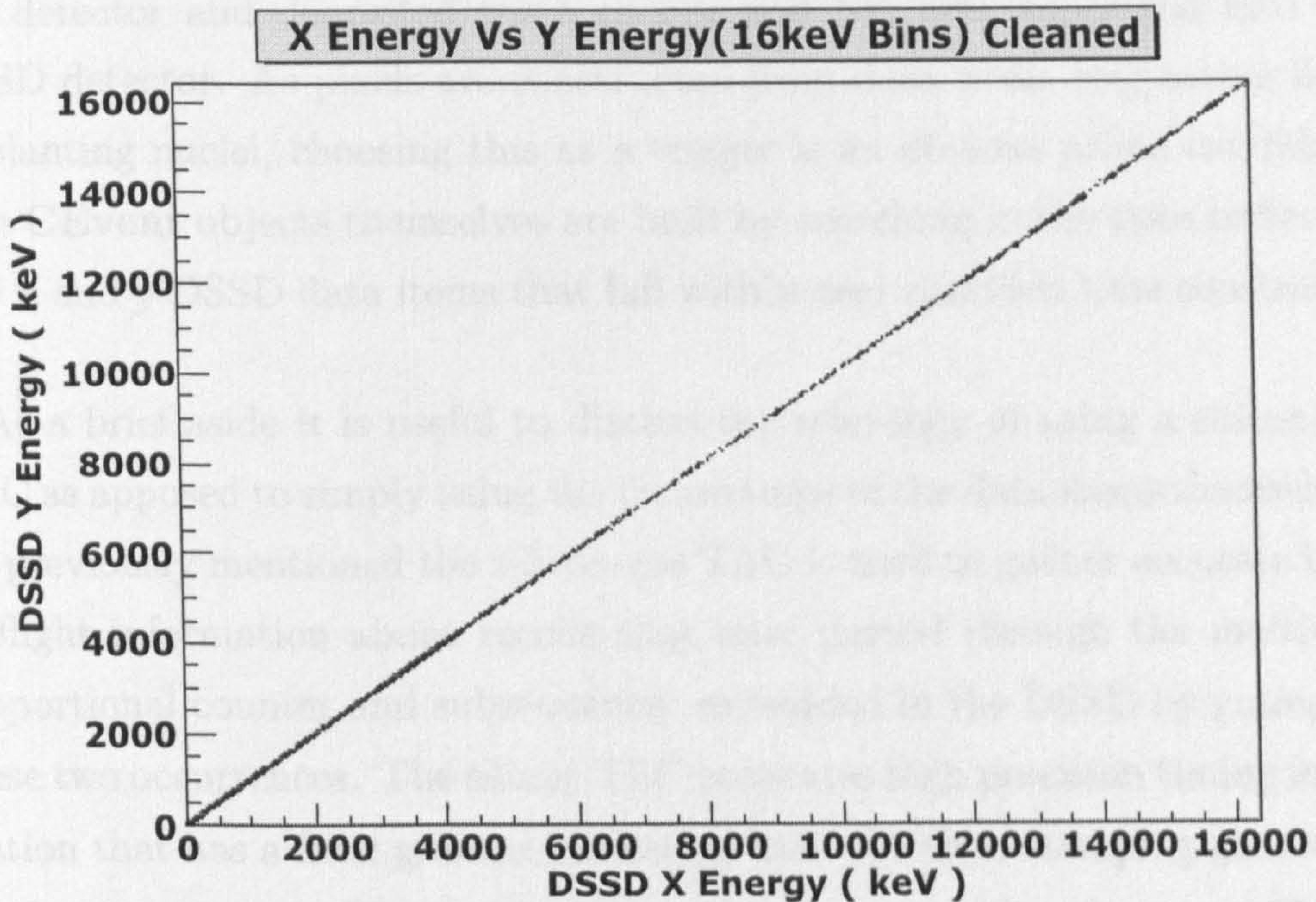


Figure 5.6: Plot showing the DSSD X energy plotted against the DSSD Y energy. The figure shows only the DSSD strips that have passed the energy gating criteria ($\pm 50keV$ energy gate).

5.2.2 Triggering Considerations

Before giving some examples of how pixels are defined from different data sets it is necessary to briefly discuss triggering issues and how the pixel definition process can vary depending on what data items are triggered from. As mentioned in previous sections the TDR data analysis system is triggerless, this enables all the data to be read out without being dependent on any particular hardware trigger.

In order to sort the time ordered data stream it is useful to have a user defined software trigger from which to build the **CEvent** objects necessary. One obvious choice is to trigger from the Silicon-Gas TAC as described in section 2.4.7. When a silicon-gas TAC data item is found in the data stream

it is an indication that a recoiling nucleus has passed through the MWPC gas detector and deposited some energy and has also implanted into the DSSD detector. As pixels are constructed from data items originating from implanting nuclei, choosing this as a trigger is an obvious prime candidate. The **CEvent** objects themselves are built by searching in the time buffer for any x and y DSSD data items that fall within user specified time constraints

As a brief aside it is useful to discuss the relevancy of using a silicon-gas TAC as apposed to simply using the timestamps of the data items themselves. As previously mentioned the silicon-gas TAC is used to gather accurate time of flight information about recoils that have passed through the multiwire proportional counter and subsequently embedded in the DSSD by gating on these two occurrences. The silicon TAC generates high precision timing information that has a finer grained resolution than the time-stamping generated by the metronome. Figure 5.7 shows a histogram of the silicon-gas TAC's high resolution data.

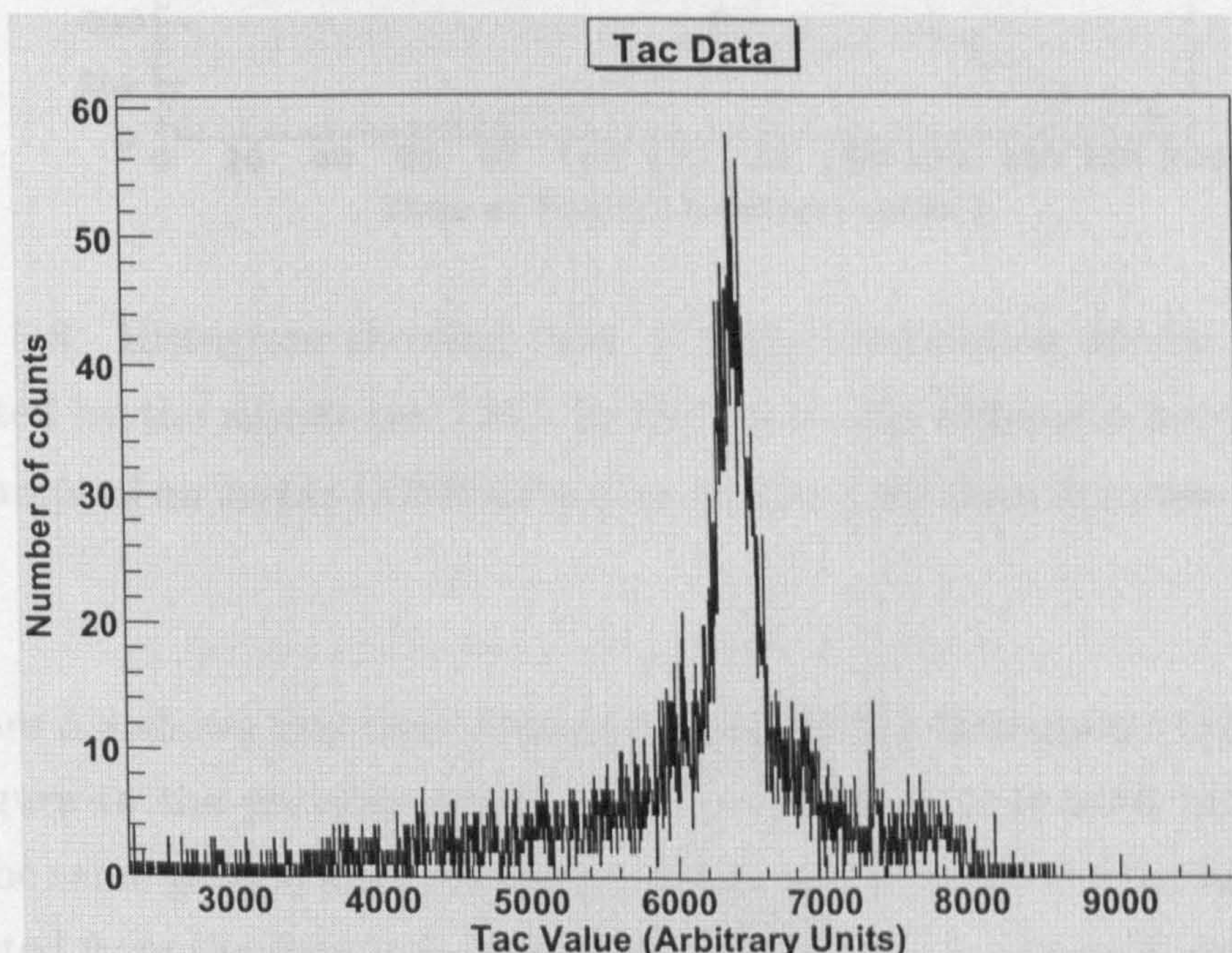


Figure 5.7: Histogram showing the high resolution timing information generated by the silicon-gas TAC.

In addition to the above method of gathering time of flight information about recoiling nuclei from the silicon-gas TAC. It is possible to gather the same information, albeit of lower precision, by using the time-stamping information of data items in the data stream. By identifying a **CEvent** object that has both a defined pixel (or a DSSD X or DSSD Y event on its own) and a defined gas event a similar function can be performed to the TAC by simply taking a difference between the relative data items timestamps.

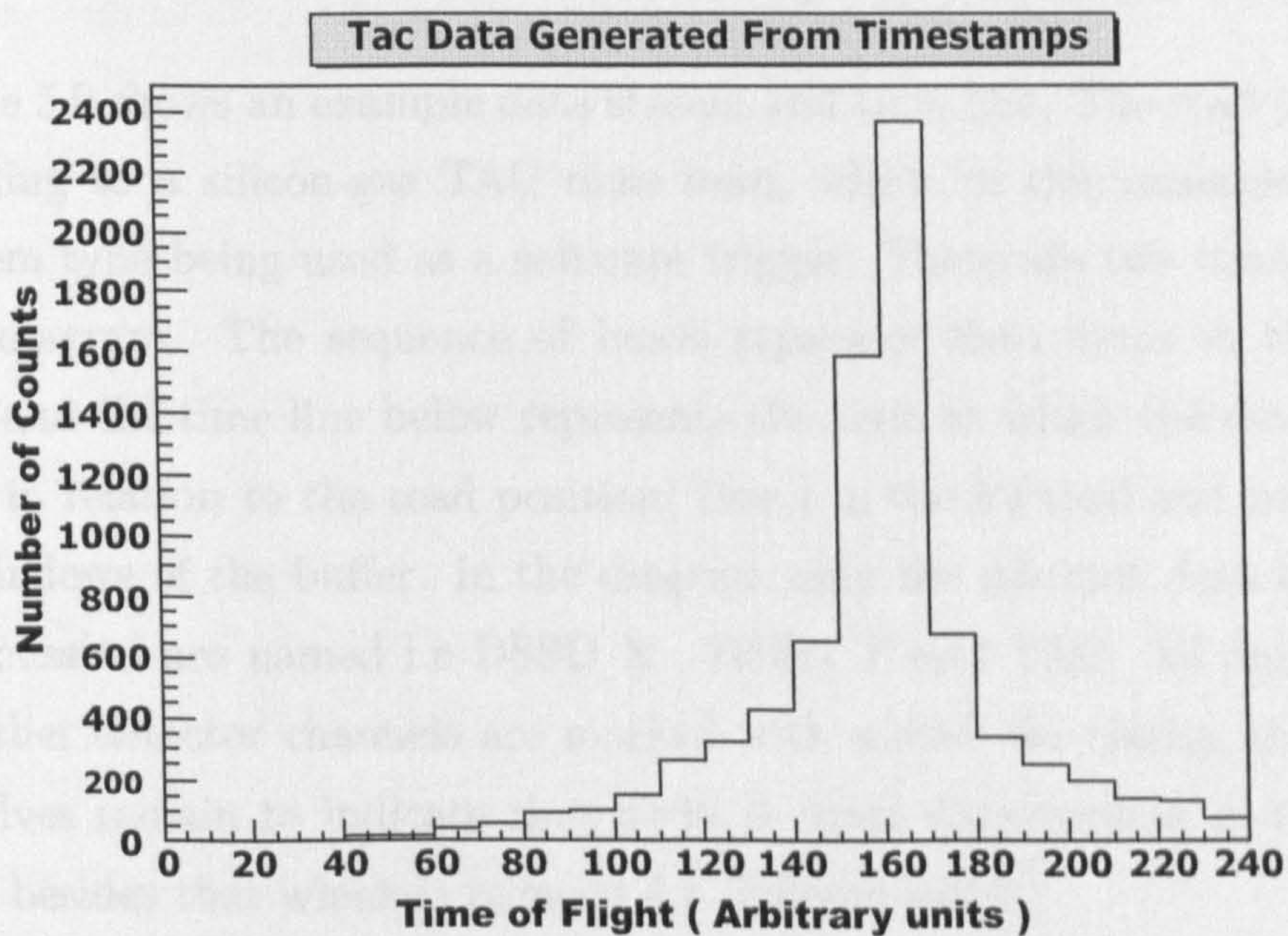


Figure 5.8: Histogram showing time of flight information similar to that generated by the silicon-gas TAC. In this figure the difference between the timestamps of an events DSSD data item and its GAS data item was plotted.

Figure 5.8 shows this time difference plotted in a histogram. Comparing this figure to the previous figure 5.7 it can be seen that both histograms have the same general shape. However it can also be seen that the histogram generated from the data item time-stamp differences is of significantly lower resolution than the histogram generated from the silicon-gas TAC.

Another possibility for a good software trigger is to use **CTDRDataItem** objects that are generated from the individual DSSD detector channels them-

selves. In this case rather than the **CEvent** objects being built from the Silicon-gas TAC they are constructed from either an x or y DSSD data item. When a DSSD x or y data item is triggered from, a corresponding y or x DSSD data item is searched for in the time buffer. If the found data item falls within specified time constraints it can be used to define a pixel.

5.3 Pixel Definition Examples and Problems

Figure 5.9 shows an example data stream and time line. The read position is pointing to a silicon-gas TAC data item, which in this example is the data item type being used as a software trigger. There are two components to the diagram. The sequence of boxes represent data items in the data stream and the time line below represents the time at which the data items appear in relation to the read position($0ns$) in the forward and backward time windows of the buffer. In the diagram only the relevant data items to the discussion are named i.e DSSD X , DSSD Y and TAC. All data items from other detector channels are marked with a dash for clarity, the boxes themselves remain to indicate that there is other data present in the data stream besides that which is relevant for defining pixels.

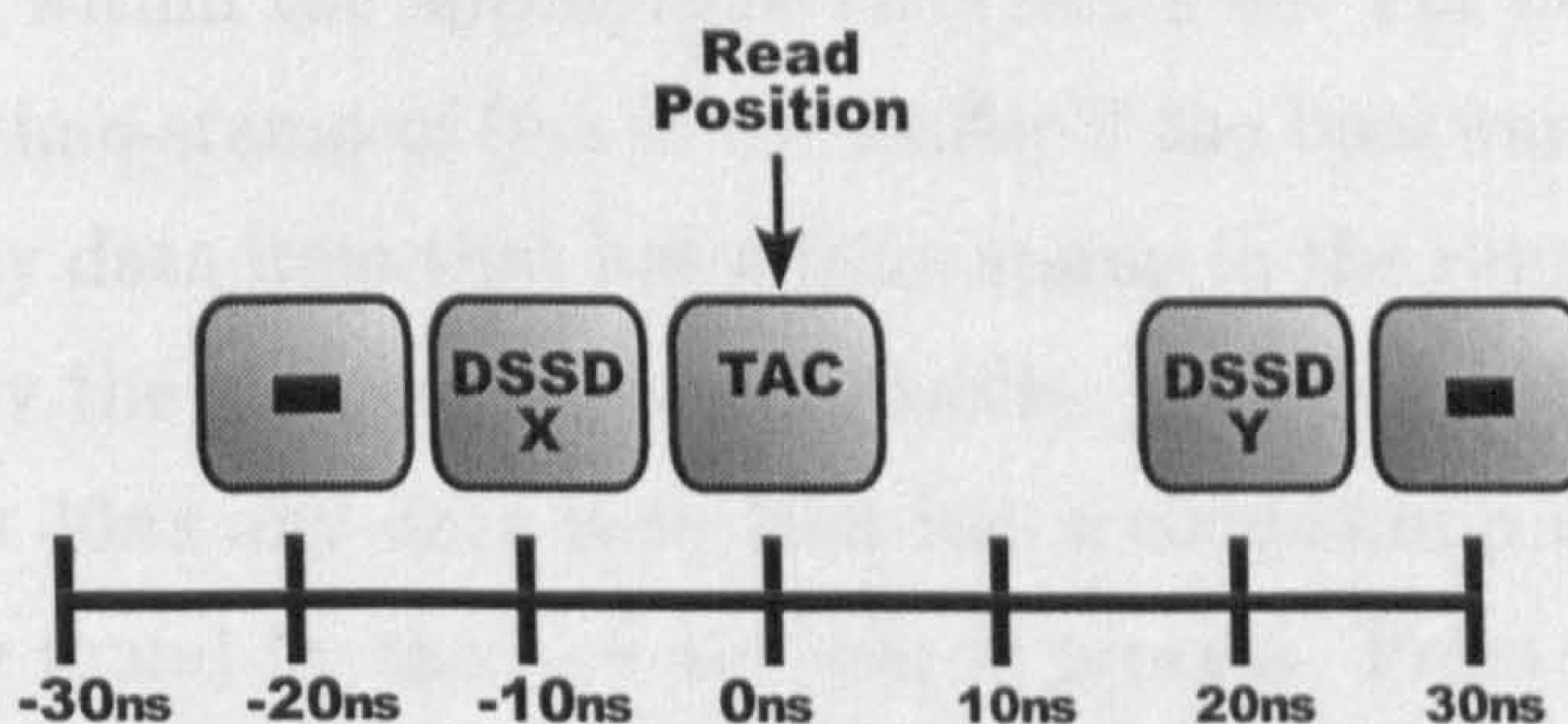


Figure 5.9: Figure showing a sample data stream and timeline from which a pixel can be defined.

Figure 5.9 shows a portion of data stream that is in an ideal state to define pixels. Suppose that a silicon-gas TAC data item is being used to trigger from and that the user has defined a time buffer with a forward and

backwards time window of $30ns$. The sequence of data items in the diagram represents all of the data items in the data stream that lie within $\pm 30ns$ of the time-stamp of the read position. The time line shown indicates the time in tens of nanoseconds starting from zero at the read position.

As described in previous sections in order to define a pixel both a DSSD X strip and a DSSD Y strip need to be found in the buffer within close temporal proximity to the triggered silicon-gas TAC data item. When using the TAC as a trigger, the process to find these x or y strips is to simply search within the buffer and extract the appropriate data items when they are found. Referring to figure 5.9 it can be seen that within this portion of the data stream amongst the other events that are not relevant for pixel definition there are two data items, one from a DSSD X channel and one from a DSSD Y channel.

The logic of the event construction process is as follows. The time-stamp of the data item at the read position is stored (the read time). Starting from the read time the buffer is then searched in both forwards and backwards time directions for any DSSD X or DSSD Y data items that have a time-stamp that fall within the appropriate time windows. For example the read position has a time-stamp of $0ns$ in the buffer if the backwards time window is $30ns$ then any data item that has a time-stamp in the range $-30ns$ to $0ns$ will be found by the backwards search process. Also given that the forwards time window is $30ns$ any data item that has a time-stamp in the range $0ns$ to $30ns$ will be found by the forward search process. From figure 5.9 it can be seen that a DSSD X strip is at the $-10ns$ position in the data stream and a DSSD Y lies at the $20ns$ position, both of which lie within the required range.

Searching the time buffer is performed by calling the **find()** or **findAll()** methods of the **CBuffer** object. Two variables are passed in as parameters that specify the amount of time to search within the buffer in a forwards and backwards direction. A third parameter is used to set the **SEARCH_TYPE**

of the find method that specifies what data item to look for in the buffer. This parameter essentially maps the channels in the GREAT spectrometer to a detector group defined in one of the TDRSorter's header files making it easy to search for specific groups without worrying about the channel numbers [1] of the detector. The **SEARCH_TYPE** parameter is passed to the **find()** method which returns the first data item that it finds of the specified type that falls within the search criteria. The **findAll()** method returns a list of all data items that are found within the search criteria.

In the example illustrated in figure 5.9 the **findAll()** method is being used to locate and return a list of all data items that fall within the specified search criteria. In the example the list of found data contains two items, one DSSD X strip at $-10ns$ position and one DSSD Y strip at the $20ns$ position. These two data items are used to define the pixel. As mentioned earlier this example illustrates the simplest pixel definition case, the following sections describe other likely situations and how the complications they introduce are solved.

5.4 Problem Conditions in Pixel Definition

This section forms the bulk of this chapter. The various problems that can arise when defining pixels are discussed in some depth. In particular how pixels can still be successfully defined when there are multiple DSSD data item candidates within the time buffer. Attention is also given to the issue of double counting and how it can be avoided.

5.4.1 Multiple X and Y Strips

Figure 5.10 shows a different section of time buffer and its accompanying time line. This figure shows a more complicated situation. As before the diagram represents a complete time buffer triggered from a silicon-gas TAC

[1] The channel numbers of the data acquisition system can be reordered so having the mapping of 'channels to detector groups' in one place makes changing this mapping easy.

data item at the read position but this time there are two data items that correspond to DSSD X channels, one at the $-10ns$ position and one at the $-20ns$ position. There is also one data item corresponding to a DSSD Y channel at the $10ns$ position. Using the same time buffer parameters as before all three of these data items fall within the buffer's time window for defining pixels. This situation creates a problem in that a decision now has to be made as to which of the data items in the data stream actually corresponds to the true x and y parameters of the pixel.

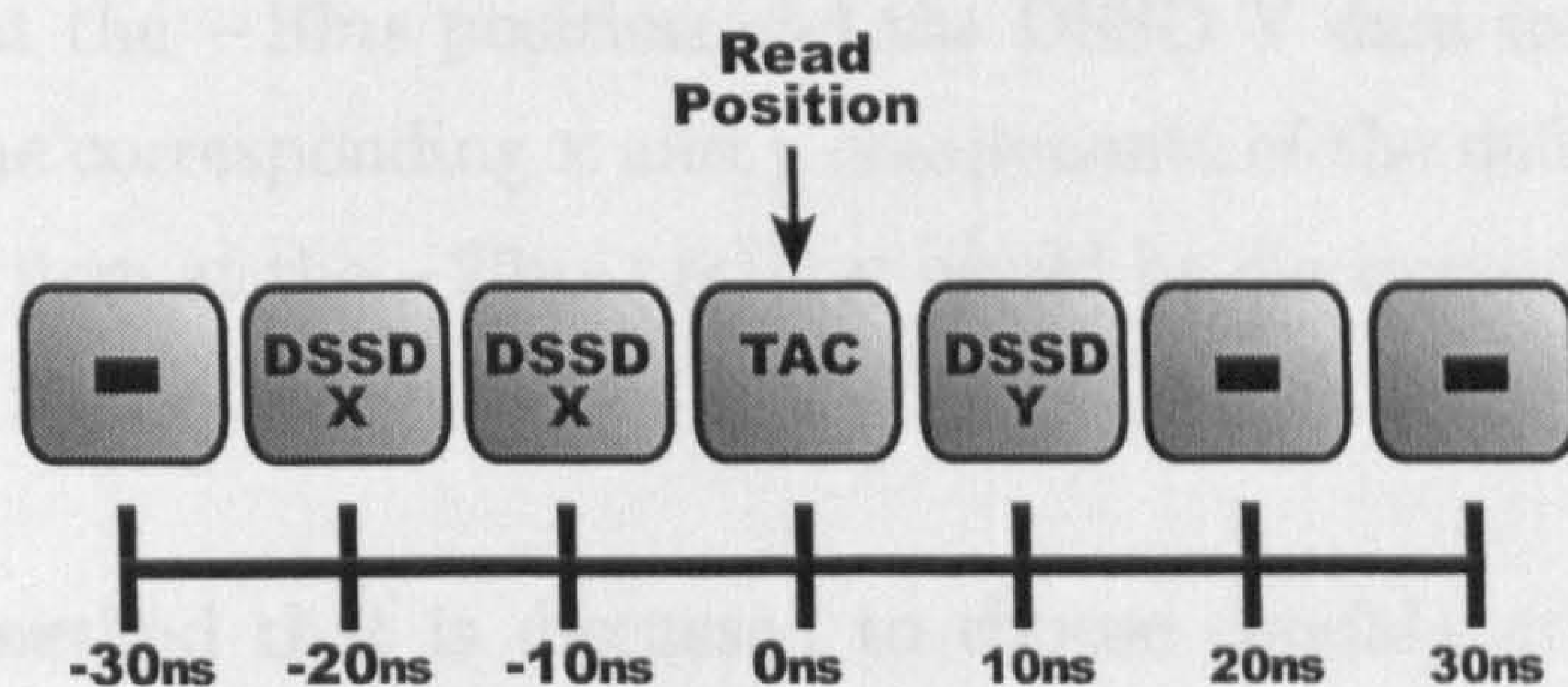


Figure 5.10: Figure showing a sample data stream and timeline with multiple x strips.

Given the above situation it is necessary to develop a method for finding the most likely DSSD data items in the data stream to constitute a pixel. In dealing with the time and energy of the detector channels only, it is not possible to say that any given data items definitely correspond to a valid pixel and therefore have to manage with defining pixels from the most probable data items to constitute them. As before the **findAll()** method of the buffer is used to return a list of all DSSD X and Y data items that lie within the search time window of the buffer.

The returned list of data items can be dealt with in various ways, the most trivial of which is to simply discount the data if there are multiple DSSD channels associated with a single silicon-gas TAC event. Obviously this is not the most desirable way of dealing with the data as with a little effort meaningful data can still be extracted from such a time buffer.

Another option is to select candidates from the list of DSSD data items based upon the difference in time from the time-stamp of the read position data item. The list of DSSD data items is searched and each DSSD X and Y data items time-stamp is checked against the time-stamp of the data item at the read position and a difference is taken. Using these calculated differences the DSSD X data item and the DSSD Y item that lies closest in time to the silicon-gas TAC are selected as the corresponding x and y items of the pixel. Referring to figure 5.10 using the method described would select the DSSD X data item at the $-10ns$ position and the DSSD Y data item at the $10ns$ position for the corresponding x and y components of the defined pixel. The DSSD X data item at the $-20ns$ position would be discounted from the pixel definition.

The final method that is discussed to choose candidates from the data stream is to select based on the difference in energy. The list of DSSD data items would be searched and the energy difference between each DSSD X and DSSD Y data item is taken. Referring to section 5.2.1 it can be seen that the energy difference between corresponding x and y strips lies within known limits. It can therefore be assumed that given a set of DSSD X and Y strips, those data items that lie closest in energy to one another are likely candidates for defining the components of the pixel.

Referring to figure 5.10 the energy of the first DSSD X data item at the $-20ns$ position would be read and the value compared to all other possible Y data items, which in the example is the DSSD Y data item at the $10ns$ position. The difference in energy between these two data items is calculated and then stored. The energy difference between the next DSSD X item at the $10ns$ position and the DSSD Y item is also taken and stored. Out of these two energy differences the pair of x and y data items with the smallest energy difference are selected as the most likely data items to constitute the x and y components of the pixel.

These two methods of pixel component selection; selecting by time difference and selecting by energy difference are not mutually exclusive. They can be used together to filter out unlikely candidates. One such combination would be to first select based on time i.e. search through the data stream and select the two data items (DSSD X and Y) that lie closest to the triggering silicon-gas TAC's time-stamp. The time difference between the DSSD X and DSSD Y data items would then be calculated, if this difference was within a certain range then they could be pixel candidates if they passed the next test.

Next the two data items energy values would then be checked. Only if the two data items energy difference fell within a certain range would they be considered to define a pixel. These user defined time and energy differences are effectively being used as filters to remove data items with very different values. For example if the DSSD X data item's energy was $100keV$ and the DSSD Y data item's energy was $4000keV$ they are unlikely to be from the same event in the detector. The user could specify that only data items with energy differences of $200keV$ should be considered for pixel definition which would effectively discount the data items in this example.

One caveat to be aware of when using the energy difference to select pixel components is if non-standard gain ranges are used. It is possible to apply different gains to the DSSD X and DSSD Y strips of the detector, which is useful if the DSSD is not only used to detect embedding nuclei and their subsequent alpha decays, but also to detect any conversion electron decay. As conversion electrons have a much lower energy range (maximum of $500keV$) the amplifier gain range can be set differently for one side of the DSSD to provide this ability. Clearly in circumstances such as these using the energy difference to identify pixels is misguided.

5.4.2 Double Counting

An issue that is of prime concern in any experiment and in particular those with low statistics is not only making sure that all data in the data source is accounted for but also that the data is only used once. In low statistic experiments this is vitally important as small variations in the number of counts can make a vast difference in any calculations carried out based on these counts. This section describes how situations can arise where double counting is an issue and also explains how this problem is overcome.

Figure 5.11 shows a continuous section of the data stream starting from the $0ns$ position and finishing at the $80ns$ position as shown on the time line at the bottom of the diagram. The section of data stream has had two individual time buffers, A and B constructed from it. Each time buffer has been triggered by two separate silicon-gas TAC data items indicated on the diagram as read position trigger A and B. Read position trigger A occurs at the $30ns$ position in the whole data stream section and read position trigger B occurs at the $50ns$ position of the section of data stream.

Time Buffer A and Time Buffer B each have a forward and backwards time window of $30ns$. The read position of each separate time buffer has been set at the $0ns$ position with the forward and backward portions of the time buffer indicated as $\pm 30ns$ respectively. It is important to remember in the following discussion that these two separate time buffers overlap and are constructed from the one continuous portion of the data stream.

Starting with the read position trigger of time buffer A it can be seen that constructing a pixel by searching for all possible DSSD X and DSSD Y data items would return two data items one DSSD X at the $-10ns$ position of time buffer A and one DSSD Y at the $+10ns$ position of time buffer A. Assuming that these data items pass any user specified time and energy constraints a valid pixel can be constructed.

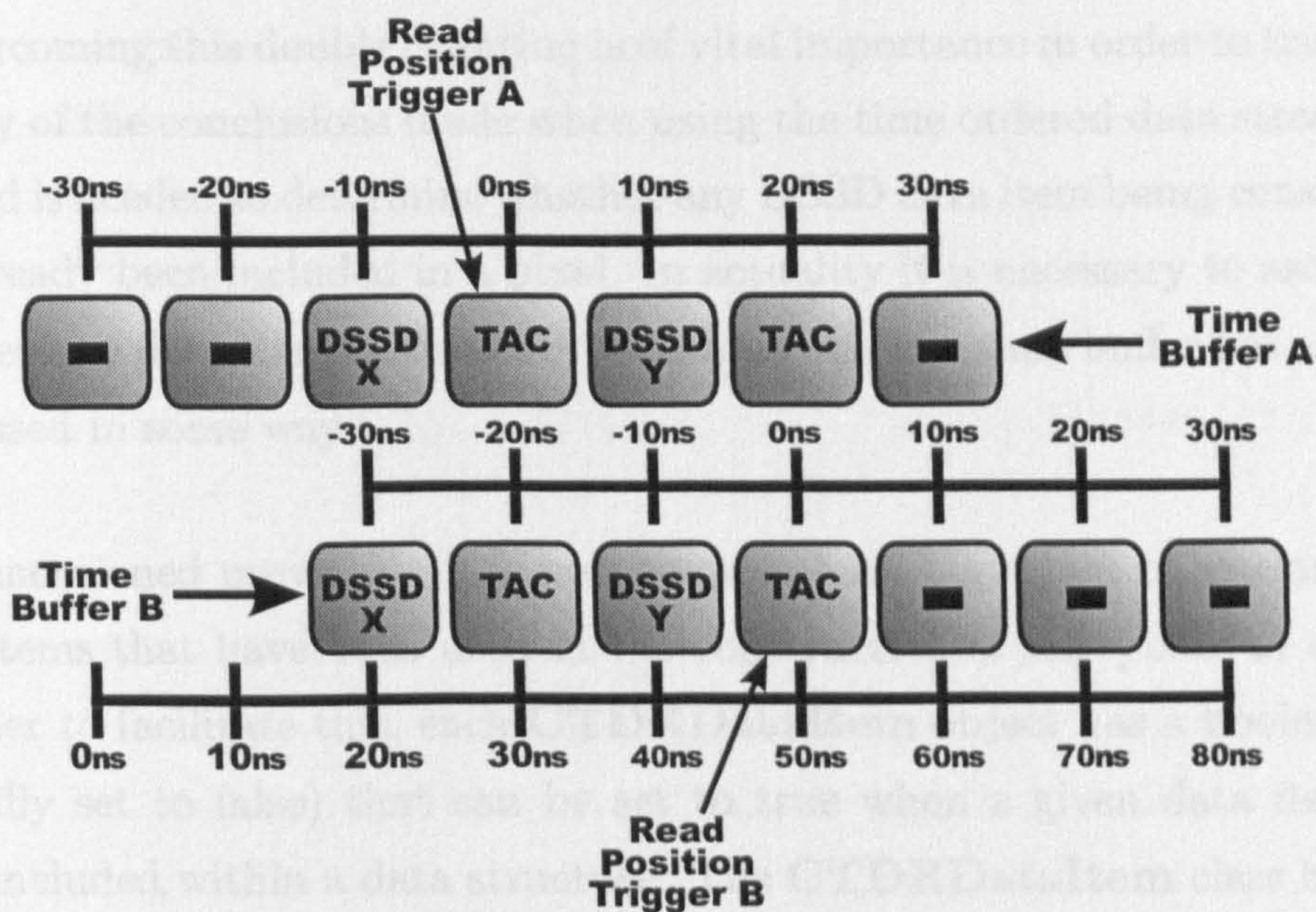


Figure 5.11: Figure showing a sample data stream and time line with the possibility of double counting.

Moving on to time buffer B a search for all possible DSSD X and DSSD Y data items in relation to the read position trigger would return two candidates. The first a DSSD X data item at the $-30ns$ position of time buffer B and the second a DSSD Y data item at the $-10ns$ position of time buffer B. Again assuming that these two data items pass any user defined time and energy conditions another valid pixel can be constructed.

Individually the pixels constructed from both time buffer A and time buffer B are perfectly valid. The problem is only observed if the data items involved are considered in their relation to their overall position in the data stream. The DSSD X and Y data items at the $20ns$ and $40ns$ position in the data stream are used as the X and Y component of both defined pixels. A situation has now arose were if both of these pixels are used to construct an event the DSSD data items in the data stream have been counted twice i.e. double counted.

Overcoming this double counting is of vital importance in order to trust the validity of the conclusions made when using the time ordered data stream. A method is needed to determine whether any DSSD data item being considered has already been included in a pixel. In actuality it is necessary to ascertain whether *any* data items returned from any search of a time buffer has already been used in some way.

As mentioned previously it is necessary to have a method of marking any data items that have been used in the construction of any pixels or events. In order to facilitate this, each **CTDRDataItem** object has a boolean flag (initially set to false) that can be set to true when a given data item has been included within a data structure. The **CTDRDataItem** class has two methods **SetAsUsed()** and **GetUsed()** that sets and retrieves the state of this flag. Essentially whenever a data item is being considered for being included in a pixel the state of the flag is checked and only if it is unset i.e. false can the data item be used.

Considering the data given in figure 5.11 then the 'used' flag prevents double counting in the following manner. Starting at the read position of time buffer A, a search for all DSSD data items returns a DSSD X data item at the $20ns$ position in the overall data stream and a DSSD Y data item at the $40ns$ position. Assuming the time and energy constraints set by the user are passed the data items are used to construct a pixel. At this point the **SetAsUsed()** method is called on each data item which sets the flag indicating that these data items have been used.

As the time buffer is incremented the data item at read position B is eventually reached i.e. the silicon-gas TAC data item at the $50ns$ position in the overall time buffer. At this point a search for DSSD items is performed which returns the DSSD X data item at the $20ns$ position in the overall data stream and also the DSSD Y data item at the $40ns$ position. At this point before any checks are performed to see if the data items pass any user defined energy constraints the **GetUsed()** method is called on each data

item to see if they have already been included in any other data structure. In the example both data items have already been included in the previously defined pixel so the calls to this method return true indicating they have been used.

At this point the construction of the pixel is abandoned as there are no valid data items left that could be used to construct a valid pixel. This method effectively removes any double counting issues by effectively disallowing any data item to be used more than once. The only remaining issue that can not be avoided is that it is impossible to judge which triggering data item the DSSD strips actually belong to. Simply being the first data item to be triggered in the data stream does not guarantee that the assignment of the DSSD data items to this event is the correct one. Unfortunately any false assignments due to this issue cannot be avoided, but as a consolation this is a lesser problem than double counting data. Any false assignments would probably generate events that would later be filtered out by further refining processes carried out. It is of course up to the user to decide if any of these events are used or ignored altogether.

5.4.3 Pixel Construction from Alternate Triggering

As mentioned previously in section 5.2.2 it is possible to use other types of data items apart from the silicon-gas TAC as a trigger. Figure 5.12 shows a portion of the data stream that could be used to define a pixel if any DSSD X or DSSD Y data item was chosen as a trigger. Using this triggering method the DSSD X data item at the $0ns$ position would be the first data item in this section of the data stream to be triggered from.

Similarly to previous methods the initial step when triggering is to perform a search in the time buffer for other DSSD data items that could be used to define a pixel. Previously the **findAll()** method was used to return a list of all DSSD data items in the data stream. As a different triggering method is being used a slightly different search strategy is necessary, but overall the

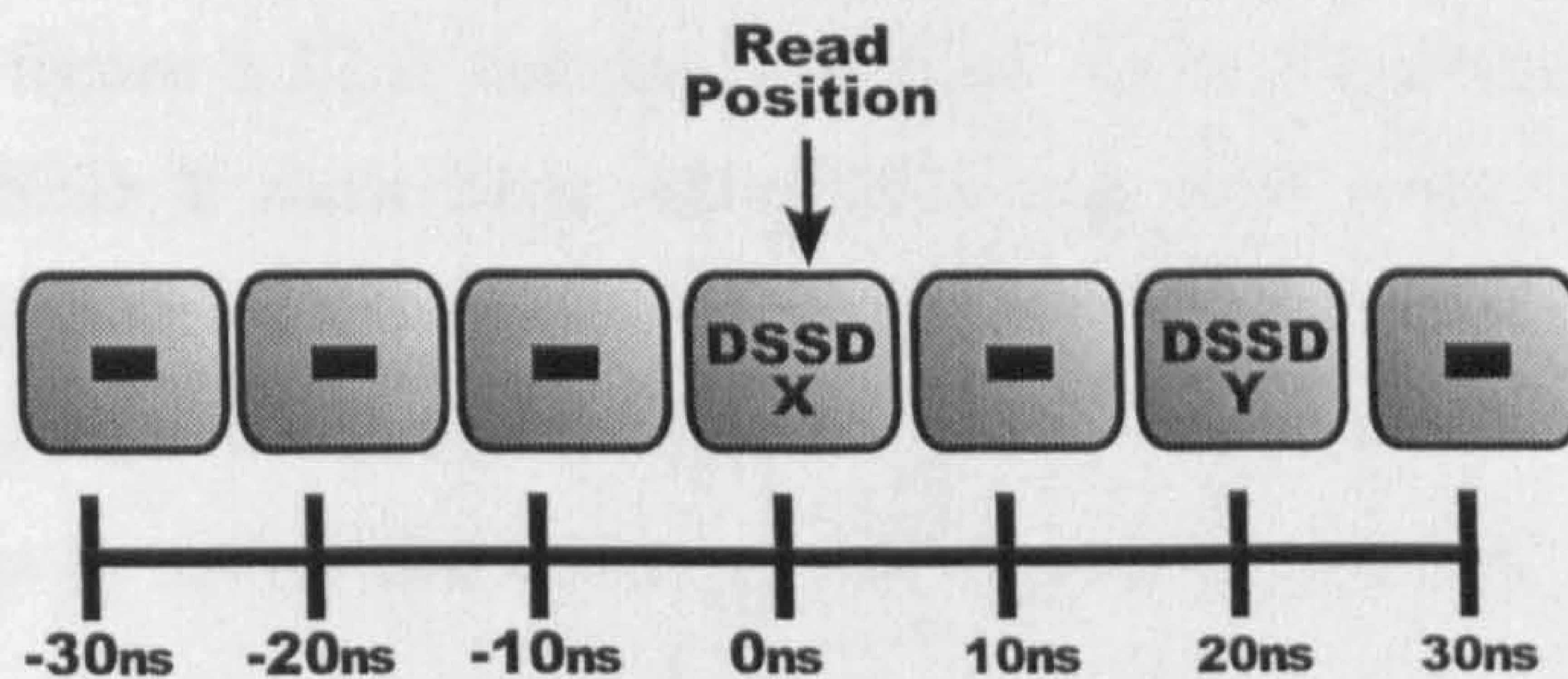


Figure 5.12: Figure showing a sample data stream and timeline to be used with alternate triggering mechanisms.

principle remains the same.

As individual DSSD data items are being triggered from as opposed to a silicon-gas TAC, simply searching the buffer for all DSSD data items will return all the data items present in the buffer *including* the original triggering data item. It is obviously necessary to be more selective in the searches performed. One method for returning only relevant data items is to selectively search for the complementary DSSD type to the triggering data item. For example if when triggering from a DSSD X data item a search would be performed for all DSSD Y data items in the time buffer. When triggering from a DSSD Y data item a search would be performed for all DSSD X data items.

Referring to figure 5.12 it can see that the data item being triggered from i.e. the data item at the read position is a DSSD X data item. In order to build a valid pixel a corresponding DSSD Y data item is needed, therefore it is necessary to perform a search in the time buffer for all DSSD Y data items. In the example this returns a single DSSD Y data item from the $20ns$ position in the data stream. As described in previous sections this point of the process could be used to perform time and energy checks to see if the data items do define a pixel within the user defined constraints.

Considering figure 5.12 it can be seen that when triggering from either a DSSD X or DSSD Y data item, situations can arise were double counting can become an issue. This occurs as from any defined pixel both the x and y component would be used to trigger from. Fortunately it is possible to circumvent this by using the strategies discussed in previous sections.

5.5 Event Packaging

As mentioned previously at the beginning of this chapter the pixel is the central data structure needed in the construction of **CEvent** objects. Now that methods for defining a pixel have been discussed it is possible to progress onto a explaining how they are used to build these events.

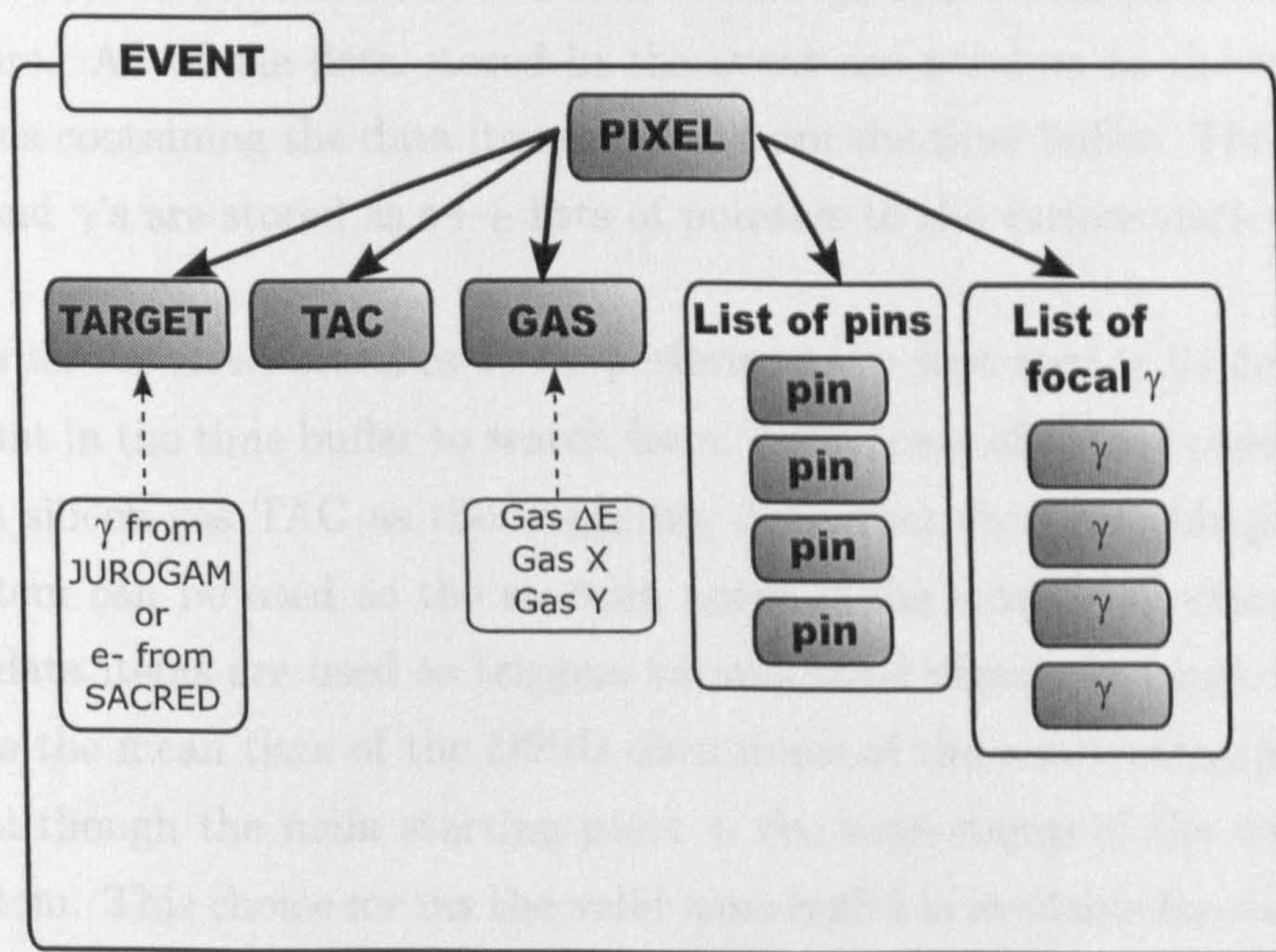


Figure 5.13: Schematic showing how events are packaged within a **CEvent** object. The constructed pixel is used as the starting position to search for target, TAC, gas, PINs and focal plane gammas that are in prompt coincidence.

Figure 5.13 shows a schematic of how the **CEvent** object is composed once it has been constructed. As previously mentioned the pixel is the central data structure of the **CEvent** class. Alongside this is the related target data items; the associated silicon-gas TAC; the data items associated with the multi wire proportional counter (or gas detector); a list of all PIN diode data items and finally a list of all focal plane gamma data items that can include gamma rays detected from the planar germanium detector or other gamma detectors such as a clover detector.

The various components of an event are built up using similar methods used in the construction of a pixel. Various searches are performed on the time buffer that has already had a valid pixel constructed from it and relevant data items are extracted and formed into groups within the event data structure. All of the data stored in the event are pointers to the memory locations containing the data items copied from the time buffer. The lists of PINs and γ's are stored as c++ lists of pointers to the various data items.

In order for these searches to be performed the first step is to decide on the point in the time buffer to search from. In the case of a pixel constructed from a silicon-gas TAC as the triggering data item the time-stamp of this data item can be used as the starting point of the search. In cases where other data items are used as triggers various other strategies could be used such as the mean time of the DSSD data items of the constructed pixel. In general though the main starting point is the time-stamp of the triggering data item. This choice means the valid time buffer is available for searching.

Once the starting position has been decided the next stage is to perform various searches of the time buffer looking for the various other data item components that compose an event. The first stage would be to search for the silicon-gas TAC data item that is associated with the pixel. In cases where this is the triggering data item, no search is necessary and the data item is simply assigned to the event. In cases where the trigger is something other than the silicon-gas TAC a simple search is performed to find the silicon-

gas TAC that lies closest in time to the triggering data item. One thing to take note of is that the pixel is the only compulsory constituent of the event object, all other data items are optional so if any search returns zero data items the corresponding component of the event is simply left blank. The filtering of events that do not contain components that the user might want to require is performed in the specialised sorting section in which the user can define custom filtering code.

The next search performed is for any multi wire proportional counter (Gas) data items that may be in the time buffer. As with the previous search in this case the data items that appear closest in time in the time buffer are selected as the gas components of the event object. It can also be appropriate at this point to perform the following sanity check on the values of any gas data items i.e. any gas data items time-stamp must be offset from the implantations time-stamp by the flight time ($\approx 1\mu s$ see section 2.4.7) of the recoil through the spectrometer. If a gate is set around this time range then it is possible to positively identify if the gas data items found in the data stream indicate if this pixel is a true recoil candidate. Using this technique it is possible to eliminate any recoil misidentifications due to random noise in the gas detector being wrongly used to identify a recoil event.

The next two searches performed, the one for the associated PIN diode detectors data items and the one for the associated focal plane gamma data items require that all the data items in the associated time buffer be passed into the event. In these two cases a call to the **FindAll()** method of the buffer is used which returns a list of all the data item types selected. This returned list is than passed into the event. The last type of data item that can be encapsulated in an event is a list of all associated target position data items. A target position search differs slightly from the normal search in that a time offset is required to correctly identify data items that are in prompt coincidence with the triggering data items time-stamp. This offset is necessary because of the fact that the recoiling nucleus takes a finite amount of time (Time of flight) to travel from the target position where it is produced

to the focal plane where it is implanted and the resultant triggering data item is generated.

The time of flight of a given nucleus through the recoil separator can be calculated ($\approx 1\mu s$ see section 2.4.7) and this can be used to apply a time offset into any buffer search. Searches in the time buffer that use an offset essentially set the starting position of the search to some other point that does not correspond to the time-stamp of the triggering data item i.e. the read position. When any offset into the time buffer is used the user must ensure that sufficient data is being buffered by setting the time window of the buffer to an appropriate value so as to cover the size of the offset. Shorter searches are possible as each search method specifies a parameter that can be passed in to set the length of time to search for within the time buffer.

When all the searches have been performed and any data items that have been found to be in coincidence with the triggering pixel have been added to the event it is possible to proceed onto the next stage, that of specialised sorting. Before moving onto this it is useful to refer to figure 5.14 which shows a complete time buffer that can be used to construct an event. The following discussion steps through briefly the stages discussed in detail above and provides a concrete example of how an event is constructed from a 'real' time buffer.

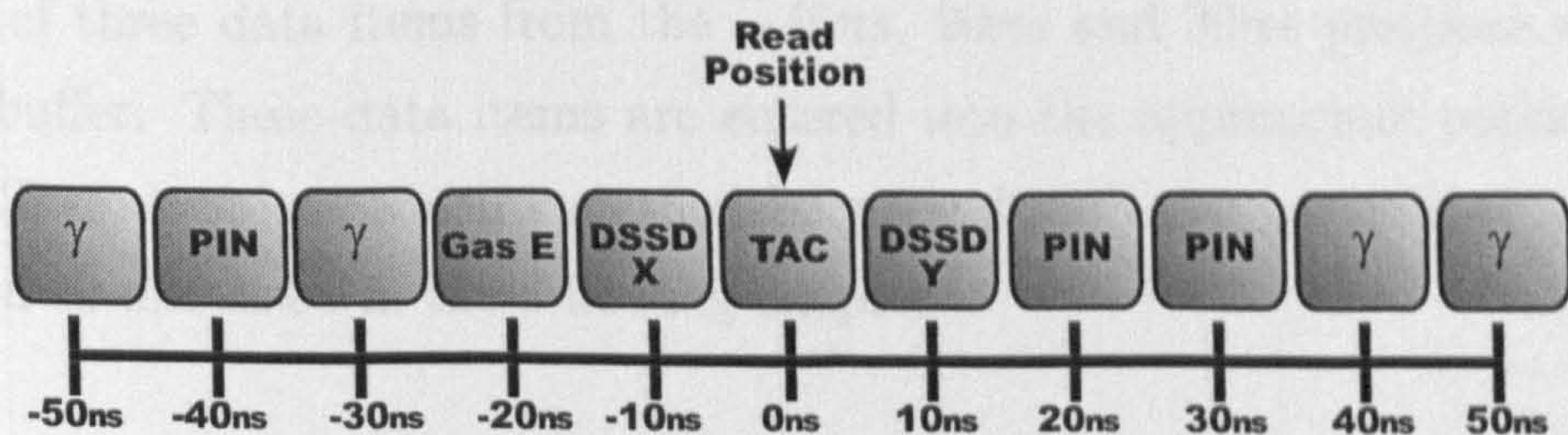


Figure 5.14: Figure showing a complete time buffer that an event can be constructed from.

The sample time buffer shown in 5.14 illustrates a 100ns continuous portion of the data stream. In this case the silicon-gas TAC data item is being used to trigger from. As indicated, a forward and backward time window of 50ns has been set with the TAC data item at the zero position. The user has set a time constraint on the pixel that any DSSD data items must have a time difference of at most 30ns.

The first stage of event construction involves defining the pixel. A search of all DSSD data items returns a DSSD X data item at the $-10ns$ position in the time buffer and a DSSD Y data item at the $+10ns$ position. The time difference between these two DSSD data items is 20ns which is less than the user defined criteria for valid pixels. Mow there is a valid pixel which can be used to build up the rest of the event. The silicon-gas TAC data item used for triggering is assigned to the TAC parameter of the event.

A series of searches is now performed to identify the rest of the data items (if any) that are in prompt coincidence with the pixel. Examining the time buffer it can be seen that the searches that will return results are the searches for PIN diodes, focal plane γ and gas data items. The gas search will return the gas energy data item at the $-20ns$ position of the time buffer. The focal plane γ search returns a list of four data items from the $-50ns$, $-30ns$, 40ns and 50ns positions of the time buffer. The final search for PIN diodes returns a list of three data items from the $-40ns$, 20ns and 30ns positions of the time buffer. These data items are entered into the appropriate portions of the CEvent object which can then be passed onto the specialised sorting section as discussed in the following chapter.

5.6 Summary

This chapter covered the areas of event construction and pixel definition. An important stage of constructing an event was the definition of a pixel. The discussion went into some depth about how a pixel is defined and examples were given as solutions to common problems such as double counting. The

discussion briefly went on to mention how the remaining constituents such as the associated gas and focal plane gamma data items are searched for and added to the **CEvent** object.

Chapter 6

Specialised Sorting

6.1 Overview

This chapter will discuss how the **CEvent** objects discussed in the previous chapter are used. Specifically how the user specifies detailed parameters and code to perform delayed coincidences using the tagging methodology. A section is also devoted to how physics information can finally be built up using the properties of the **CEvent** objects.

6.2 Deriving A Specialised Sorting Class

The main function of all previous code is to essentially extract relevant portions of the time ordered data stream and package them in a form (**CEvent** objects) that can be more effectively and easily utilised by the user. The user does this by specifying a section of code that organises the events and extracts physics information from them.

In order to perform this 'specialised sorting' the user first needs to derive their own user class from the **CSorter** class. Figure 6.1 shows how the user defined class **MySorter** derives from the base **CSorter** class. As described earlier the arrow denotes a generalisation, the derived **MySorter** class is a specialisation of the **CSorter** base class. One feature of derived classes that

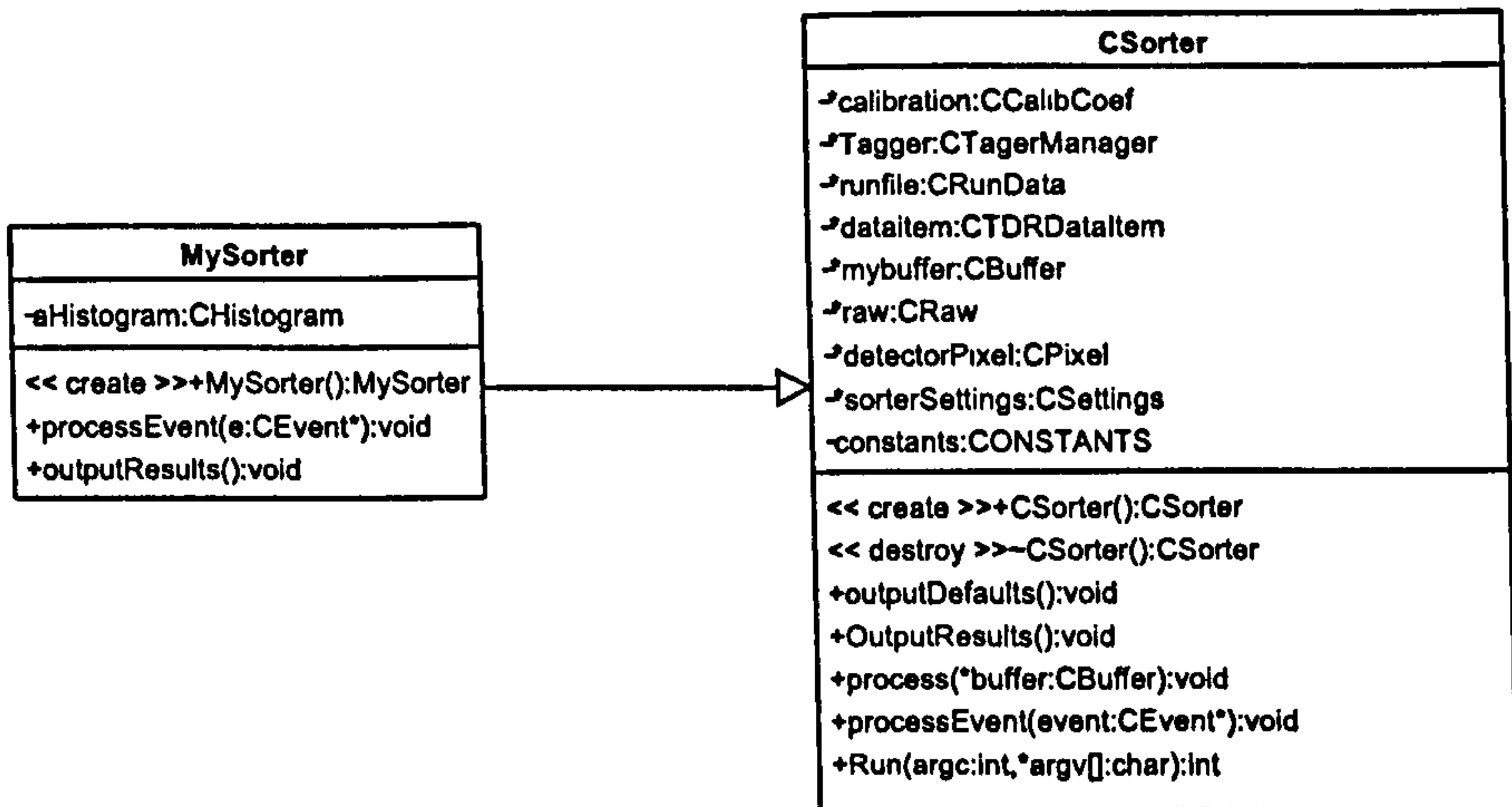


Figure 6.1: UML class diagram showing how a user specified sort class 'MySorter' derives from the base CSorter class.

it is important to note is that they inherit all of the methods and data of their parent classes. In this case the **MySorter** class gains all of the methods defined in the operations section of the **CSorter** UML class diagram as well as all the data defined in the attributes section of the class diagram. The details of how to write a class that derives from the **CSorter** is not relevant to the immediate discussion, more information on how to do this is given in Appendix A.

As described in the previous chapter, pixel definition and event construction takes place entirely in the call to **process()** in the **CSorter** class. The **CBuffer** object that encapsulates a valid time buffer is passed in as a parameter. As the **process()** method finishes it performs a call to the **processEvent()** method passing in the newly constructed **CEvent** object. This method call is where all [1] of the specialised sorting, including tagging, takes place.

[1]Apart from data output which is handled in the **outputResults()** method of the class.

Referring to Figure 6.1 it can be seen that there is a method called **processEvent()** present in both the **CSorter** base class and the **MySorter** derived class. Defining a method in such a way is called method overriding and is an important concept in object oriented programming. Declaring a method in a derived class that has the same name automatically overrides the base method. Any implementation specified in the overridden method effectively takes the place of any implementation that was specified in the base class.

To ensure that the specialised sorting class that the user derives from **CSorter** behaves correctly i.e. is capable of receiving the **CEvent** objects constructed from the time buffer then the specialised sorting class *must* override the **processEvent()** method. Any particular data manipulation that the user requires is then placed into this method where it is called once for each **CEvent** object that is created and passed to it.

6.3 Event Properties and Data Visualisation

Every time the **processEvent()** method is called a new **CEvent** object is passed in as a parameter. In order to extract information from the **CEvent** object various methods can be called that return lists of data items of a specified type. These data items can then be queried individually for information such as their energy. This process of selecting and querying can be time consuming when frequent access to the same information is needed. To limit this a summary of the most accessed data is calculated at event construction and stored for easy access. The data in the event properties structure is an exact mirror of the data contained in the main **CEvent** class and is simply present as a convenience for the user.

This data structure called an event properties structure is shown as a UML class diagram in Figure 6.2. Information contained in the data structure include items such as the energy and time of the x and y components of the pixel; the various energies, times and data of the gas detector as well as the value of the silicon-gas TAC. At any point the user can pass in an event

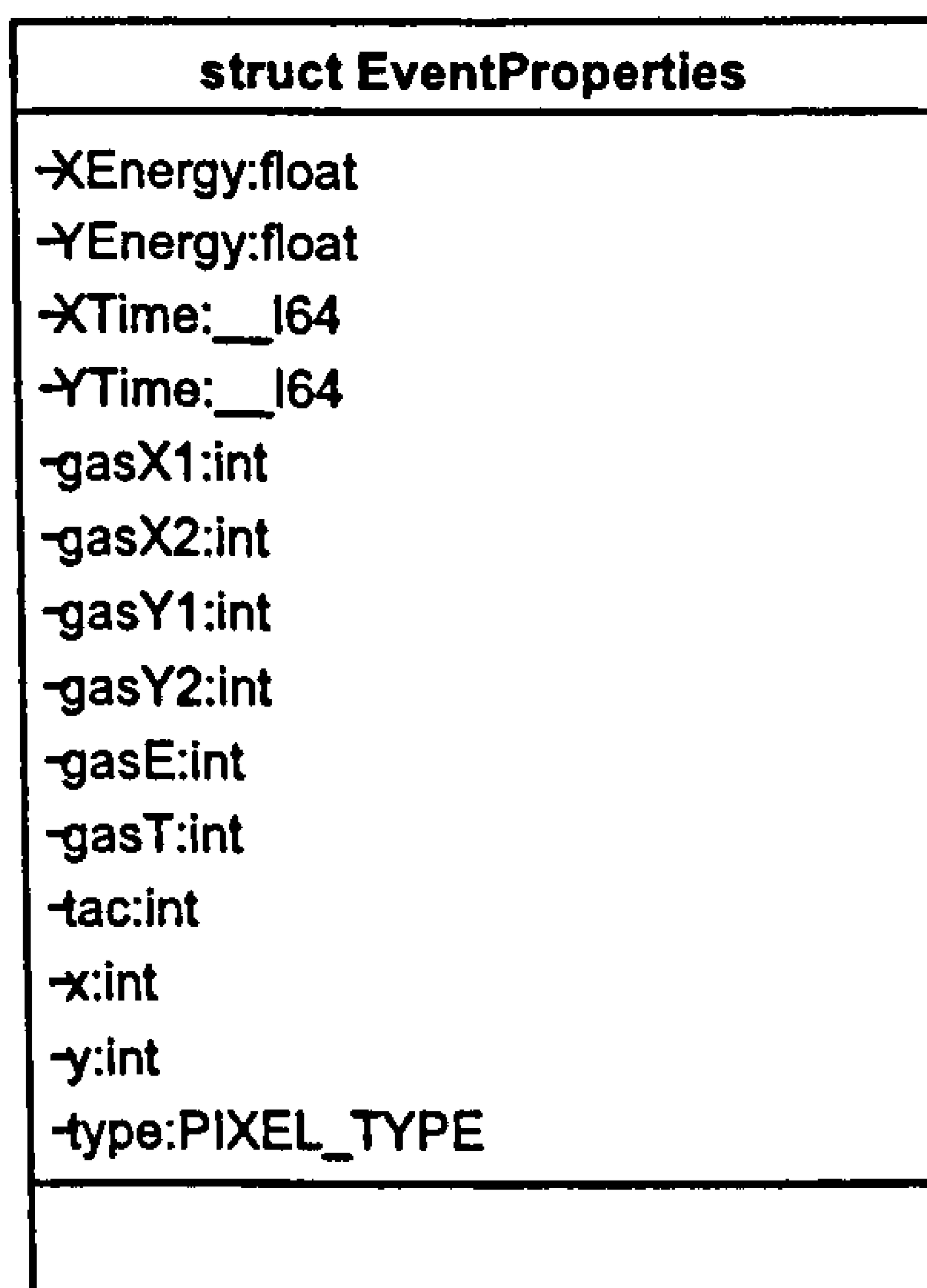


Figure 6.2: UML Class Diagram showing the event properties structure.

properties structure to the **CEvent** object currently being accessed where it will be populated by the event to contain the current values for its most used values. All of the values populated in the event properties structure can also be found by using methods directly on the **CEvent** object itself.

Before proceeding with the discussion of how the event properties are used along with the data in the **CEvent** object it is useful to briefly discuss calibrations and also how data is accumulated using one and two dimensional histograms.

6.3.1 Calibration

Before any meaningful energy information can be extracted from the **CEvent** objects the detector system must be calibrated. As all the detectors are run through separate ADC's a separate energy calibration must be calculated for each individual detector channel in the spectrometer. These calibrations are usually performed externally to the code and provided to the TDRSorter program as text files made up of a large table containing all the relevant calibration data on a per channel basis.

These calibration text files are read in once and passed in to the **CCalibration** class where it stores the parameters. The TDRSorter program usually applies the specified calibration of the users choosing e.g. linear or quadratic to the individual data items in the event construction stage. The energy for any given data item is calculated according to the calibration data for that channel by accessing methods on the **CCalibration** class. The energy value is then stored in the **CTDRDataItem** class for later retrieval. As the raw data value is always perpetuated throughout all stages of the sort code a different calibration could be applied at any time i.e. in the specialised sorting stage.

6.3.2 Histograming

In order to help use the data it is usual to provide some sort of visualisation to help in extracting information from the data set. It is useful to use both one and two dimensional histograms to help present the data gathered from the **CEvent** objects passed in as parameters to the **processEvent()** method where the current processing is taking place. In the next two sections two different types of histogram, integer and floating point will be briefly discussed.

Integer Histograms

Both the integer histogram represented by the **CIntHist** class and the floating point histogram represented by the **CFloatHist** class are derived from the **CHistogram** base class. This base class specifies basic functionality that a basic histogram should have e.g. having the capacity to increment a specified 'bin' of the histogram. The specialised integer and floating point classes are derived from this class so that collections of generic 'histograms' can be created that actually contain objects of the specialised derived classes.

The most basic type of histogram included in the TDRSorter code is the integer histogram. The integer histogram class **CIntHist** implements the functionality as prescribed by the **CHistogram** base class. Apart from

methods that provide output of the data the most important method is the **increment()** function that is used to add data to the histogram. The integer histogram class adds whole counts to the relevant bin of the histogram, which can produce 'staggering' in the data as no account is taken of how close a given count lies to the boundary of another bin. The data is either in one bin or another.

Floating Point Histograms

Floating point histograms serve to address some of the limitations of integer histograms. The main way that they do this is to allow the bins in the histogram to have fractional counts and aim to take into account the proximity of a given value to the histogram bin boundaries. For example if a given value was to lie exactly on the boundary between two bins 0.5 counts would be placed into each of the bins.

As well as using the built in **CIntHist** and **CFloatHist** classes to help visualise the data it is also possible to use a third party visualisation library. Using another library that has c++ bindings is simply a matter of including the appropriate header files and linking the library to the **MySorter** class. In most of the examples shown in the following chapter the ROOT system framework [17] is used in a purely visual capacity to display and print the data as it is sorted.

6.3.3 Basic Spectra

It is useful at this point in the discussion to show some examples of information that could be extracted from the data stream. All of the examples that follow are performed using only the information currently contained in the **CEvent** object passed into the **processEvent()** method and the **EventProperties** structure retrieved from the event.

DSSD Total Spectrum

Figure 6.3 shows the total energy spectrum of the double sided silicon strip detector. This histogram has been created by plotting the energy of the pixel retrieved from the **EventProperties** structure. The spectrum has a number of interesting features including the labeled recoils and alpha decay lines from the ^{254}No decay and the subsequent decay of its daughter nuclei.

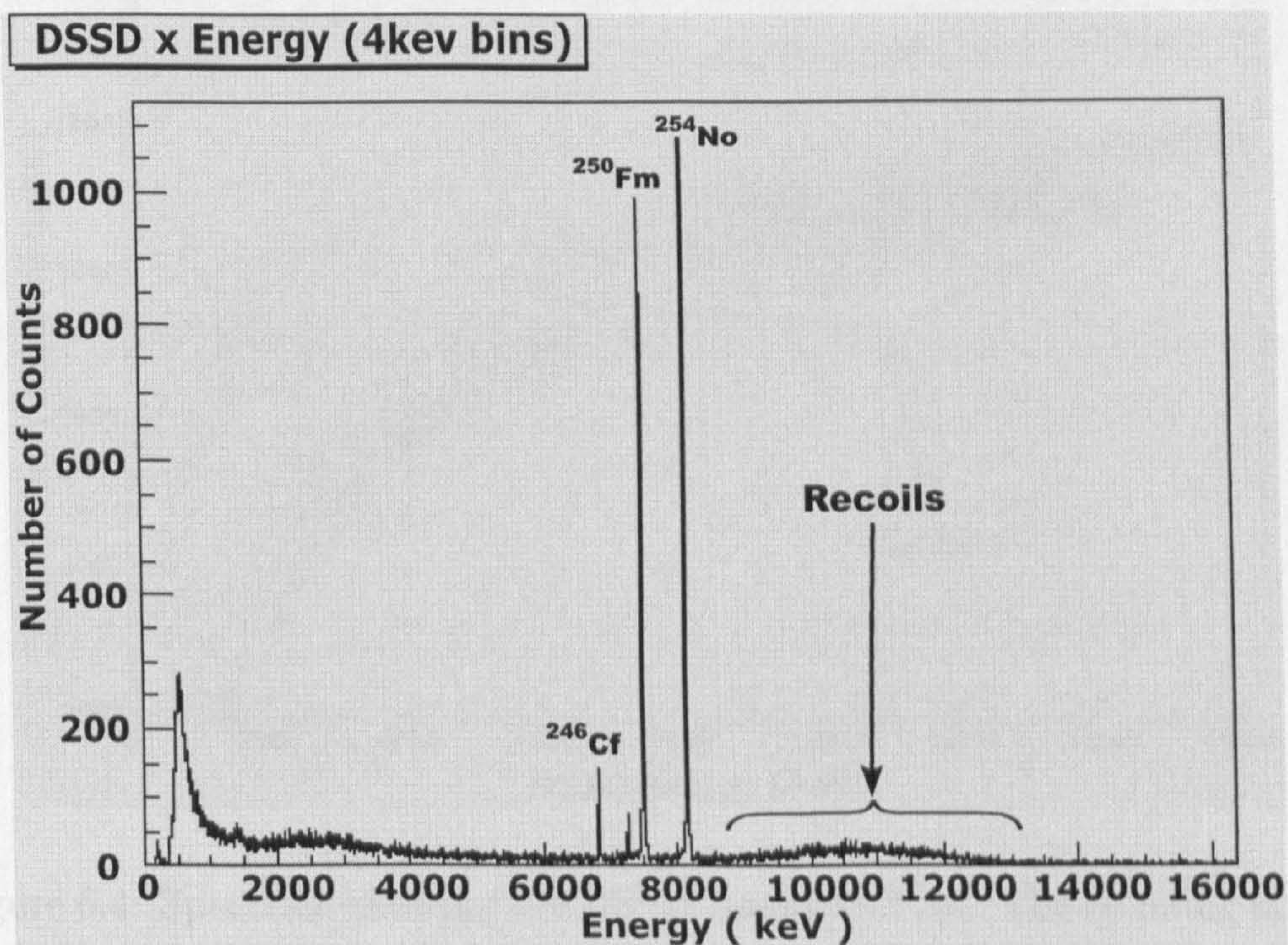


Figure 6.3: Total DSSD energy spectrum. The recoil nuclei and alpha decays of the embedded recoils can be clearly seen on the figure.

DSSD Energy vs TAC

Figure 6.4 shows a two dimensional histogram that is used to help distinguish recoiling nuclei from other decay products and scattered beam. This is created by taking the pixel energy value of the event and plotting this against the value of the silicon-gas TAC from the same event. Various structures can be observed in this plot. The structure that is of primary importance is that of the recoils (indicated by the arrow). The identification of these recoils

allows a gate to be set on specific TAC and energy values. These gates can then be used to clean up further spectra that need positive recoil identification to give accurate results. Other structures on the figure are the result of other transfer products not completely filtered out by the recoil separator.

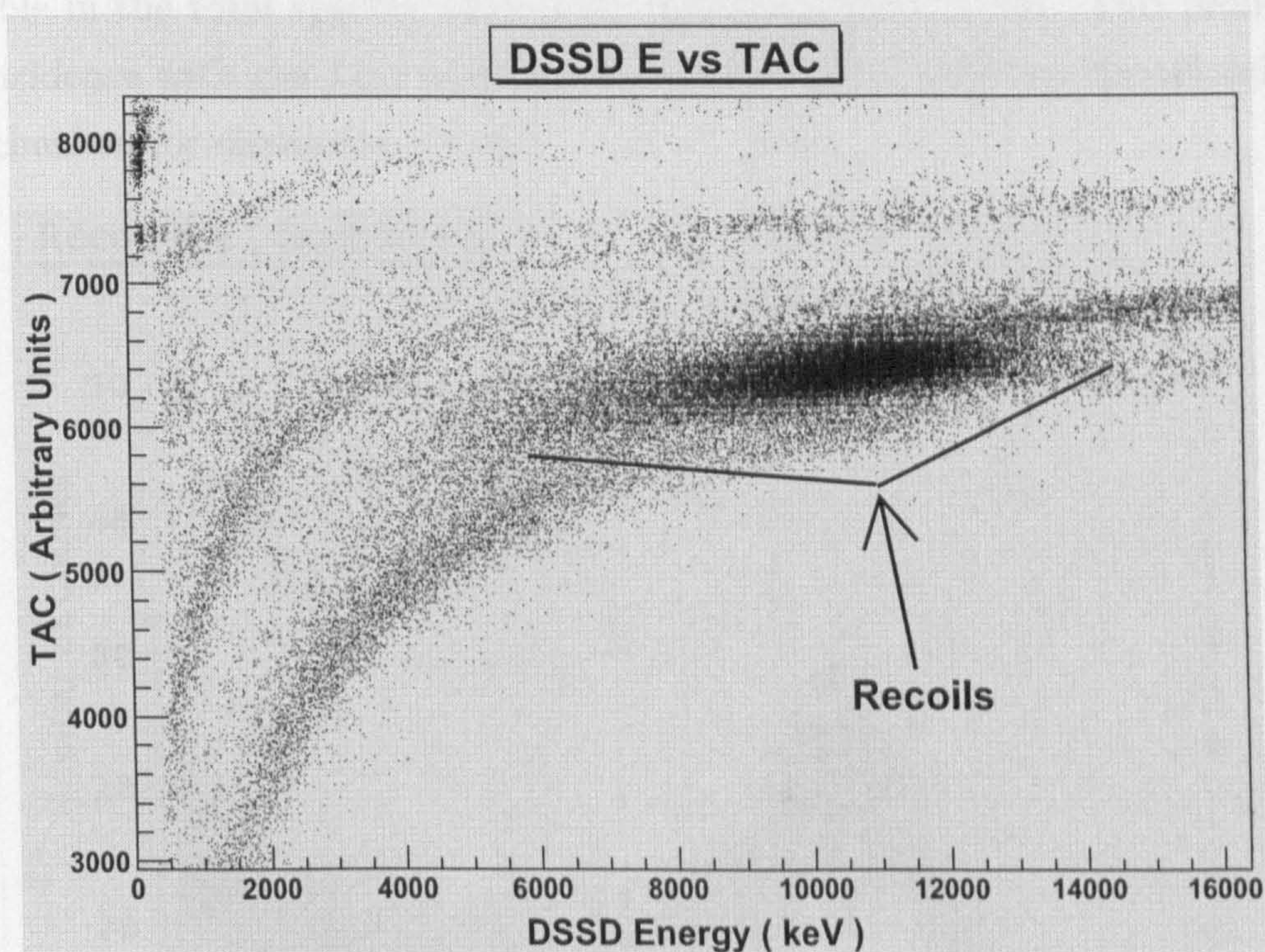


Figure 6.4: Spectrum showing the DSSD energy vs TAC. The recoiling nuclei can be clearly identified.

6.3.4 Examples of Recoil and Alpha Identification

The following sections give examples of how both recoils and alphas are correctly identified using the methods previously discussed. Each histogram shown is generated from the same example data set as before.

Recoils

By taking the event object and plotting the pixel energy only if the event contains a non zero value for any gas data item produces the result shown

in figure 6.5. This histogram essentially shows only events that have passed through the gas detector i.e. are recoiling nuclei embedding themselves in the DSSD detector. Comparing this plot to the total DSSD energy spectrum shown in figure 6.3 it can be seen that the alpha events that are clearly visible in the total spectra have been completely filtered out. This prompt coincidence with gas data items can be seen to be an effective identification mechanism for discerning recoils.

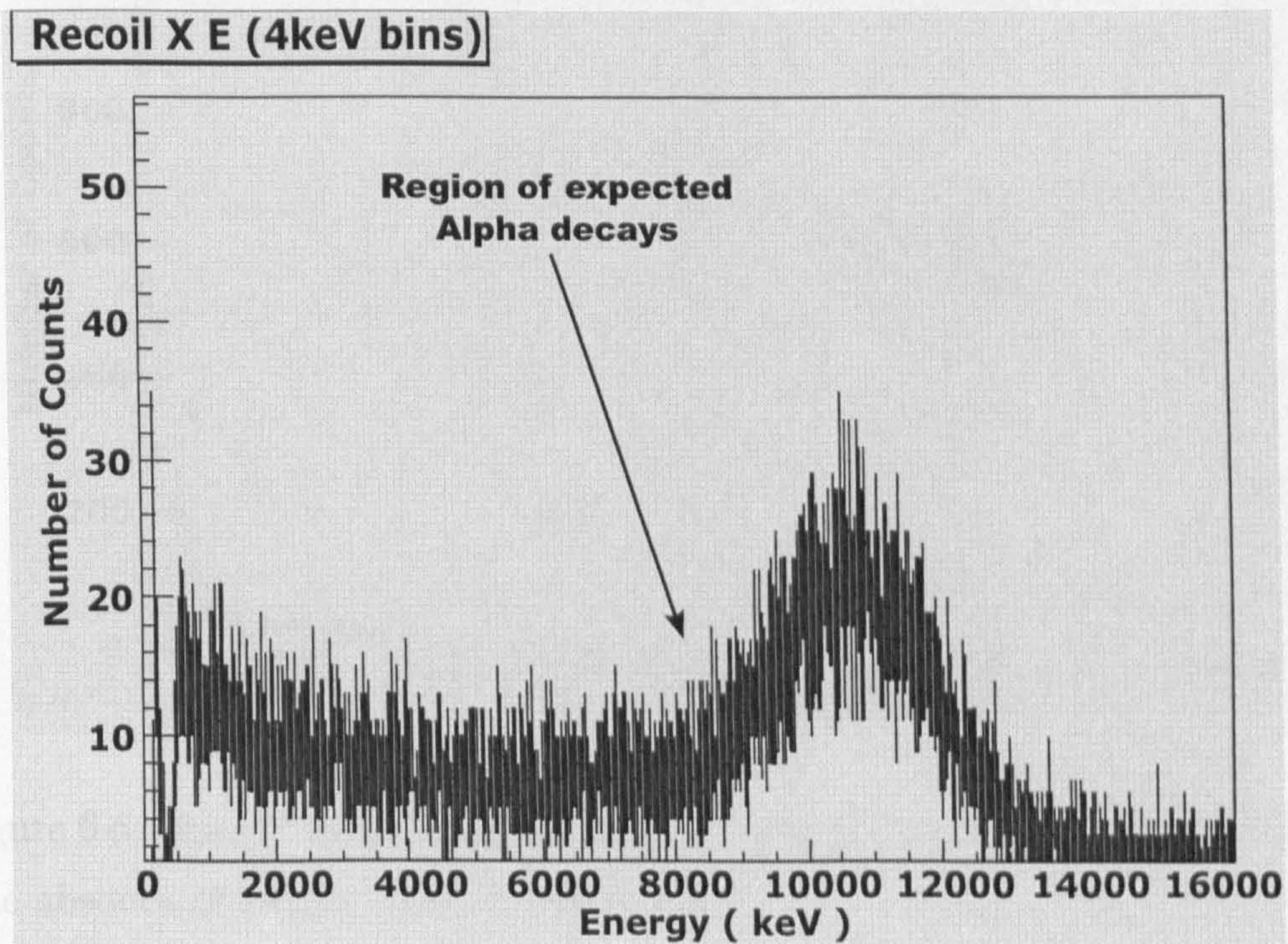


Figure 6.5: Spectrum showing the gas coincidence filtered Recoil Energy. The absence of alpha particles can be clearly seen.

Alphas

By using a similar method it is possible to clearly identify events that correspond to decays of nuclei previously embedded in the double sided silicon strip implantation detector. By plotting the pixel energy only of events that contain no gas data items figure 6.6 is produced. This histogram shows the energy of events that have not passed through the MWPC i.e. decays of

embedded recoils. Comparing this histogram to figure 6.3 it can be seen that the clearly defined area caused by recoiling nuclei has been completely filtered from the spectra, which shows that the above is a good method for distinguishing alpha decays from recoils in real experimental data.

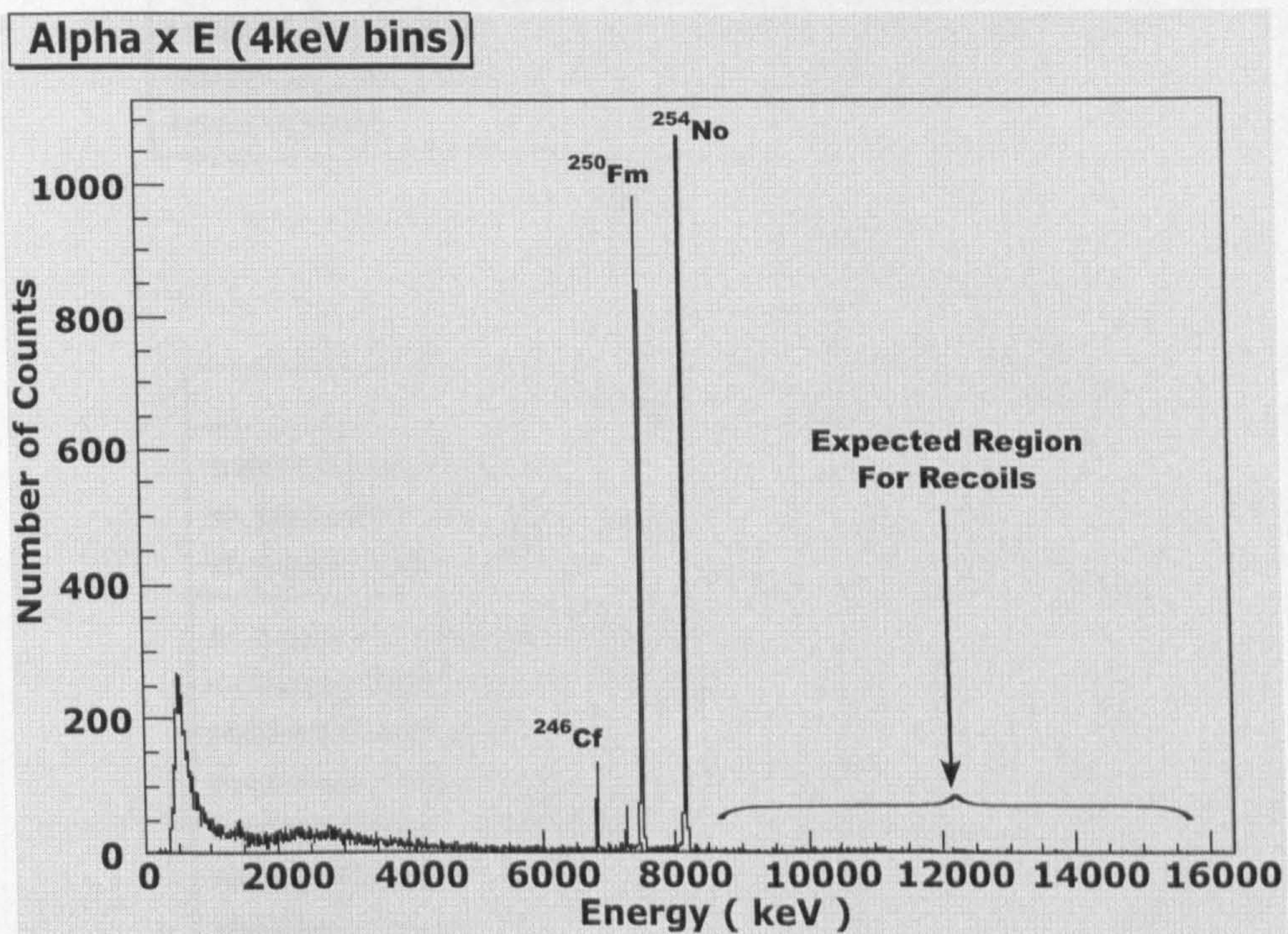


Figure 6.6: Spectrum showing the gas anti-coincidence filtered alpha Energy. The absence of recoils can be clearly seen.

6.4 Tagging

This following section discusses the tagging methodology and how it is used to perform spatial and temporal correlation for delayed coincidences. The tagger is one of the key components of the data analysis process for extracting meaningful physics information from the data stream.

Figure 6.7 shows a UML diagram for the tagger. The tagger framework is split into two parts; the tagger manager represented by the **CTaggerManager** class and the individual tagger object represented by the **CTagger**

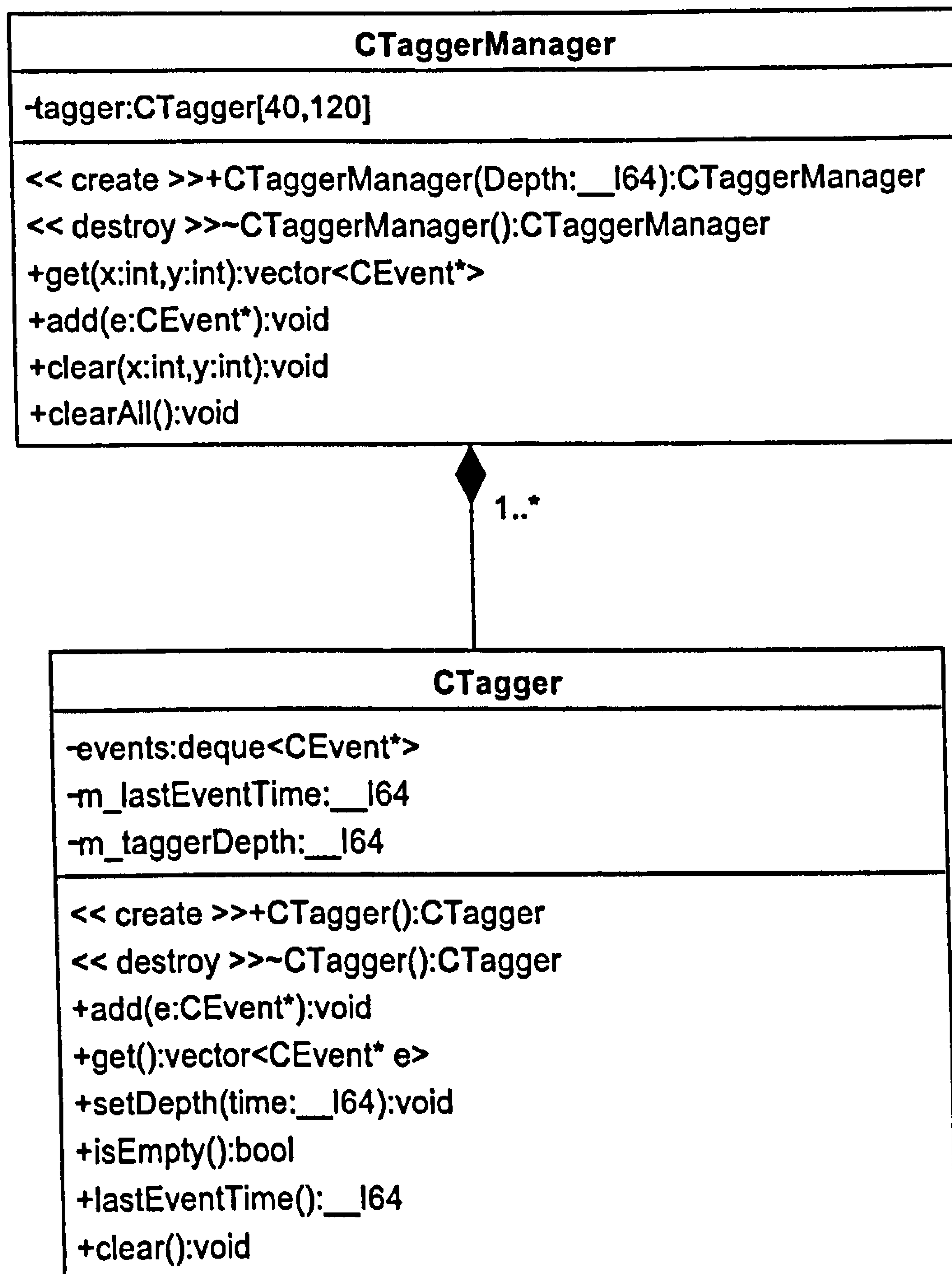


Figure 6.7: UML Class Diagram showing The TaggerManager and Tagger classes.

class. Both of these classes are closely related and provide functionality that complements the operation of both classes. Before moving onto the tagger manager the purpose and operation of the **CTagger** class is first discussed.

6.4.1 The CTagger Class

The **CTagger** class is responsible for the sequencing of events that originate from a single pixel. A **CEvent** object that is passed into the **processEvent()** method of the specialised **MySorter** class corresponds to a specific **CPixel** object. As previously discussed the **CPixel** is made from a specific

DSSD X and DSSD Y data item. Any events generated from a pixel that has the same DSSD X and Y components are handled by the same **CTagger** object.

The **CTagger** class has a number of methods used to manipulate the queuing of events within it. The first relevant method is the **void add(CEvent* e)** function that is used to add the **CEvent** object to the queue of the **CTagger**. Figure 6.8 shows how the **CEvent** objects are ordered within a given **CTagger** object. With each **processEvent** call of the specialised sorter relevant events are added to the **CTagger** object and subsequently appended to the end of the internal queue.

As the tagger object is member data of the specialised sorting class it lives as long as this class does i.e. for the duration that the entire data analysis program is running. The tagger is essentially persisted between each call to the specialised sorters **processEvent** method. This means that as time progresses and more calls are made all the events that originated from the same DSSD X and Y data items are built into the tagger's list with the oldest event at the back of the queue and the most recent (or youngest) events at the front.

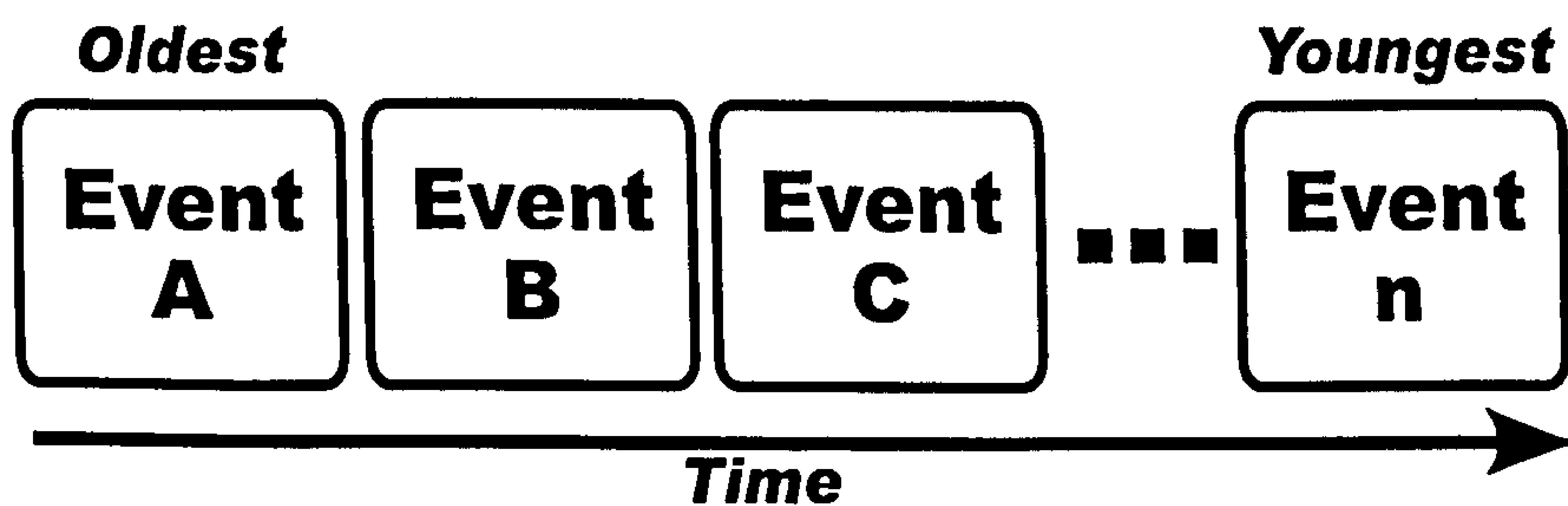


Figure 6.8: Figure showing how **CEvent** objects are added to the tagger.

6.4.2 CTagger Depth Mechanism

Another important functional aspect of the **CTagger** object is the tagger depth. The depth of the **CTagger** object is essentially the period of time that

the tagger is to store **CEvent** objects for. The tagger depth is set by calling the **setDepth()** method passing in a 64*bit* integer specifying the length in time, in terms of numbers of timestamps, in which to buffer **CEvent** objects for.

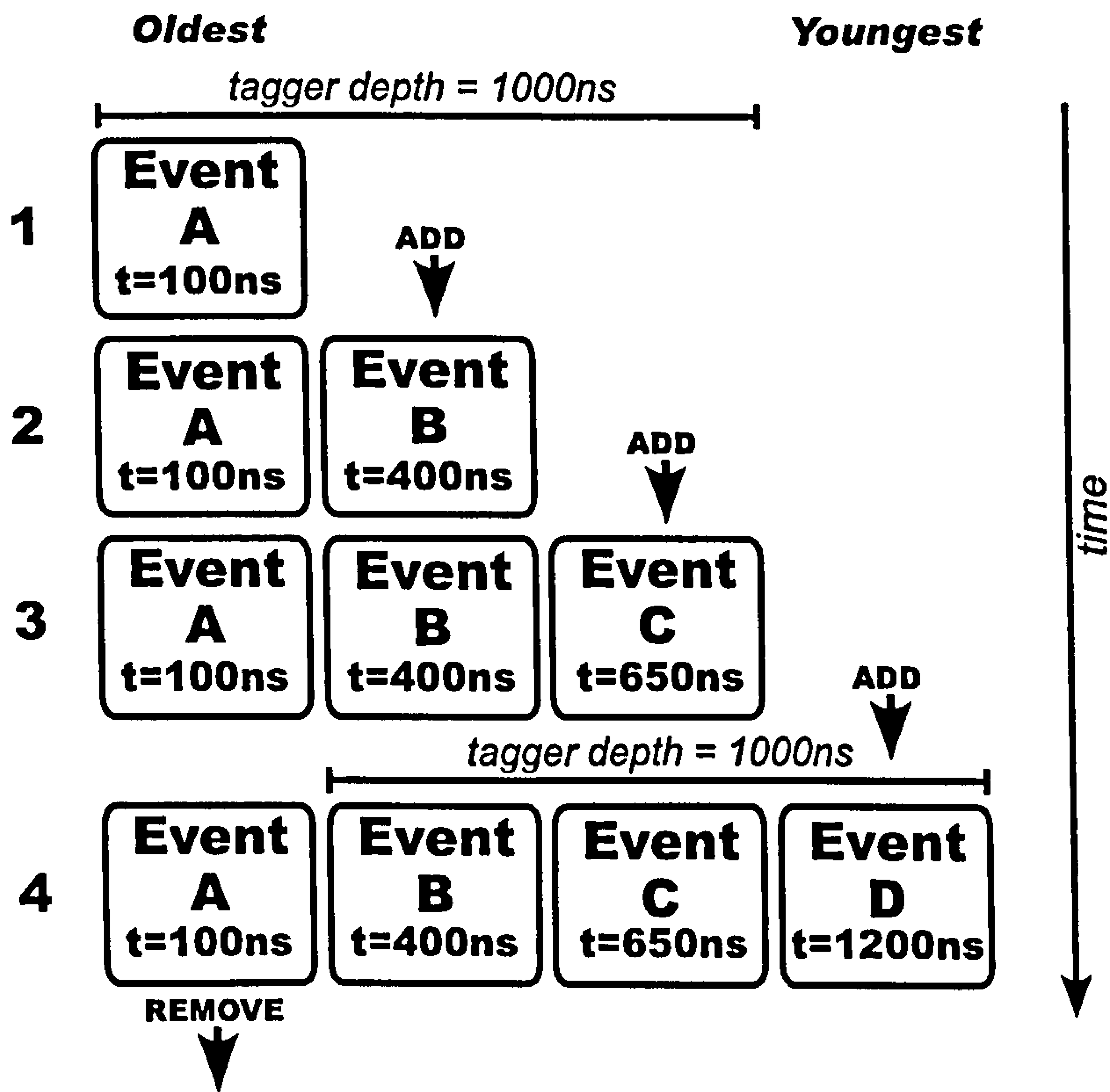


Figure 6.9: Figure showing how the depth mechanism operates in the tagger.

Figure 6.9 shows how the size of the event queue within the **CTagger** object is controlled using tagger depth checking. Whenever a **CEvent** object is added to the **CTagger** event queue the time-stamp of that event is recorded. The depth of the tagger is then subtracted from this time. The result is the earliest time that an event can have to remain on the queue. Any events that have a time earlier than this value should be removed as they are no longer of any interest in the analysis.

Referring to figure 6.9 it can be seen how this process works. Starting with a **CTagger** object that has had its depth set to $1000ns$. Each of the indiviual stages represents a separate call to the specialised sorters **procesEvent()** method that is not necessarily consecutive to the previous call. The **CEvent** object passed in as a parameter corresponds to a pixel with the same DSSD X and Y as the previous tagger addition.

At stage 1, event A is added to the tagger's queue. As the tagger was previously empty i.e. there was no items in the queue no tagger depth check is performed. Stage 2 has another **CEvent**, event B, being added. At this point a tagger depth check is performed. The last event added has a time-stamp of $400ns$, the tagger depth of $1000ns$ is subtracted from this value leaving the earliest time that an event should be queued for as $-600ns$. The event queue is then iterated starting at the earliest item and the time is checked against the earliest time allowed. The first event in the queue, event A has a time of $100ns$ which is greater than this earliest time. All of the events currently on the queue are still of interest.

Stage 3 corresponds to another addition of an event, event C that has a time of $650ns$. As before a event depth check is performed, the earliest allowed event must have a time greater than $-350ns$ to remain on the queue. The iteration shows that all events are still of interest. Stage 4 corresponds to the addition of event D with a time of $1200ns$. The tagger depth check reveals that the earliest time an event can have to remain on the queue is $200ns$. As the queue is iterated starting at the earliest item it can be seen that the first event on the queue, event A has a time of $100ns$. This time is less than the minimum value required so this event is removed from the tagger.

The process of tagger depth checking is performed for all events in all **CTagger** objects as they arrive and are added. This mechanism ensures that only events that are deemed to be relevant for the user are stored in memory at any given time. The magnitude of the tagger depth must be

selected carefully as the depth is essentially the period of time that any delayed coincidences are to be searched for. A good rule of thumb for setting the depth of the tagger would be to select a time that is multiples of the lifetime of any decay being studied. (e.g. $3xT\frac{1}{2}$.)

6.4.3 The Tagger Manager

As discussed in the previous section each individual pixel is associated with its own individual **CTagger** object. As each pixel is defined by a unique combination of DSSD X and DSSD Y channels it can be inferred that one **CTagger** object is needed for each pixel of the DSSD Detector. As discussed in Chapter 2 the DSSD detector is divided into two halves consisting of 60 X channels and 40 Y channels making a total of 4800 pixels. Therefore a total of 4800 **CTagger** objects are needed to accurately store and queue all of the **CEvent** objects for use in finding delayed coincidences.

Trying to manage 4800 **CTagger** objects could rapidly become unwieldy so the **CTaggerManager** class was created to help in keeping all the individual objects in one place with a single interface used for adding, deleting and retrieving events as they are required. The **CTaggerManager** class is detailed in the UML diagram in figure 6.7 and provides several methods for manipulating the contained data. The central structure in the **CTaggerManager** class is the two dimensional array of **CTagger** objects called tagger which is defined as **CTagger tagger[120][40]**. This two dimensional array essentially maps the DSSD detectors physical layout with one **CTagger** object per pixel.

Figure 6.10 shows how this arrangement can be visualised. The grid of squares in the figure is a representation of the two dimensional array of **CTagger** objects within the tagger manager class **CTaggerManager**. Each square of the grid is referred to as a tagger cell of the tagger manager and is essentially an instance of the **CTagger** class within the array. The boxes within each tagger cell correspond to a single **CEvent** object within the

event queue stored in the **CTagger** object. Multiple stacked boxes represent the sequence of events in the queue as outlined in figures 6.8 and 6.9.

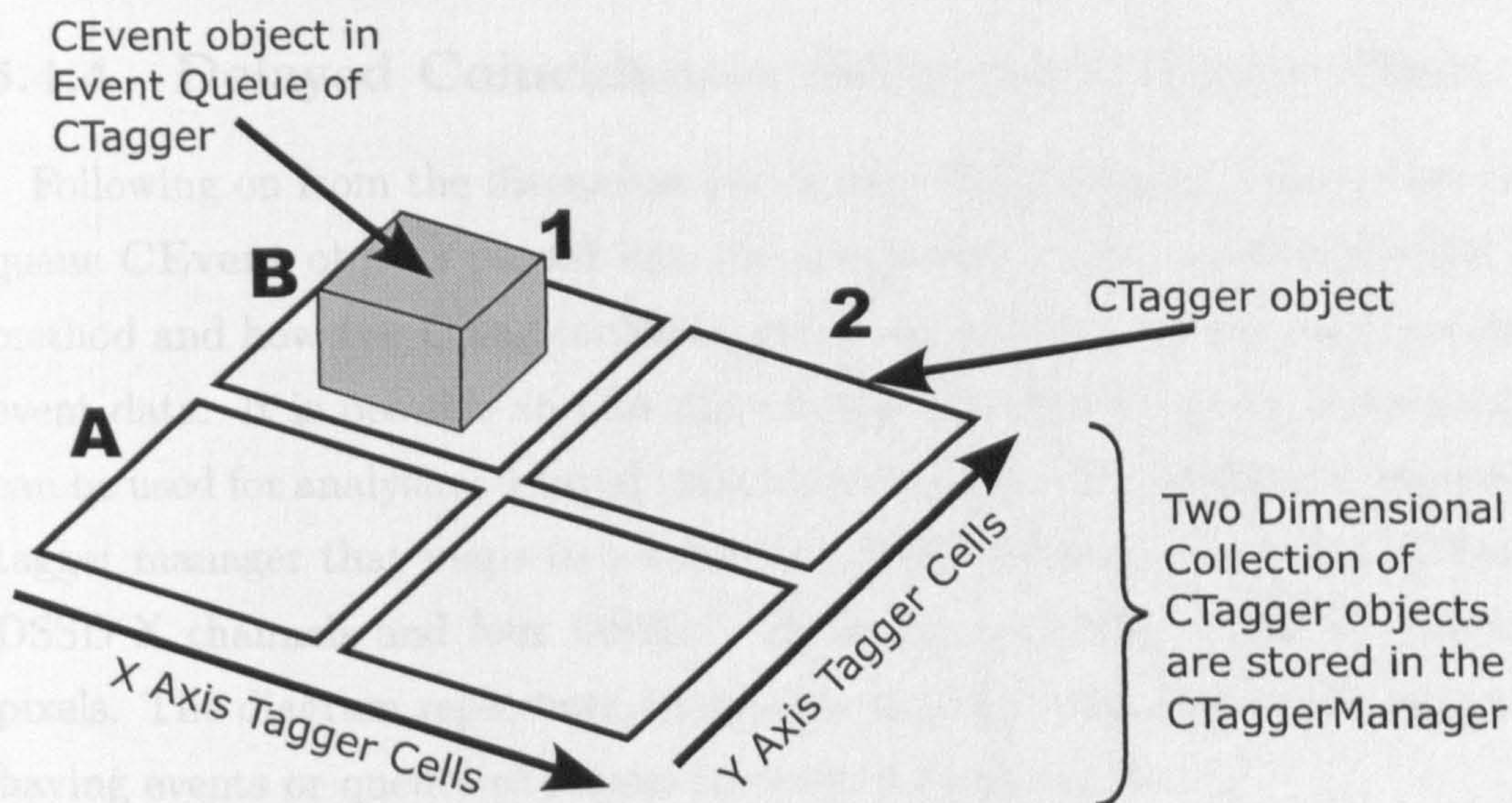


Figure 6.10: Figure showing a simplified tagger manager.

The **CTaggerManager** class has several methods as outlined in 6.7 for accessing and modifying the underlying array of **CTagger** objects. When the tagger manager is first created the required depth is passed in as a parameter. The tagger manager calls the **setDepth()** method of each **CTagger** object in the contained array to set the depths of the individual taggers. The tagger manager also specifies a method for adding **CEvent** objects to it. To do this a call to the **add()** method of the manager is made passing in the event object to be added. Within this method the X and Y parameters of the pixel is queried and the event is added to the appropriate tagger in the array.

The final method that is of interest is the get method which is specified as **vector<CEvent*> get(int x,int y)**. When executing this method the x and y parameters of the pixel of interest must be specified, which ensures that only the list of events that of are interest are returned. The **CEvent** objects are returned as a list of pointers that point to the events contained in

the underlying tagger objects within the managers array. These events are returned in the same order as they have been added as detailed in figure 6.8.

6.4.4 Delayed Coincidences using the CTagger Class

Following on from the discussion about how the **CTagger** class is used to queue **CEvent** objects passed into the specialised sorters **processEvent()** method and how the **CTaggerManager** class is used to access and process event data. It is possible to now discuss how the whole tagging framework can be used for analysing delayed coincidences. Figure 6.11 shows an example tagger manager that maps to a simplified DSSD detector consisting of four DSSD X channels and four DSSD Y channels producing a total of sixteen pixels. The diagram represents a snapshot in time with some of the taggers having events or queues of events currently in time scope.

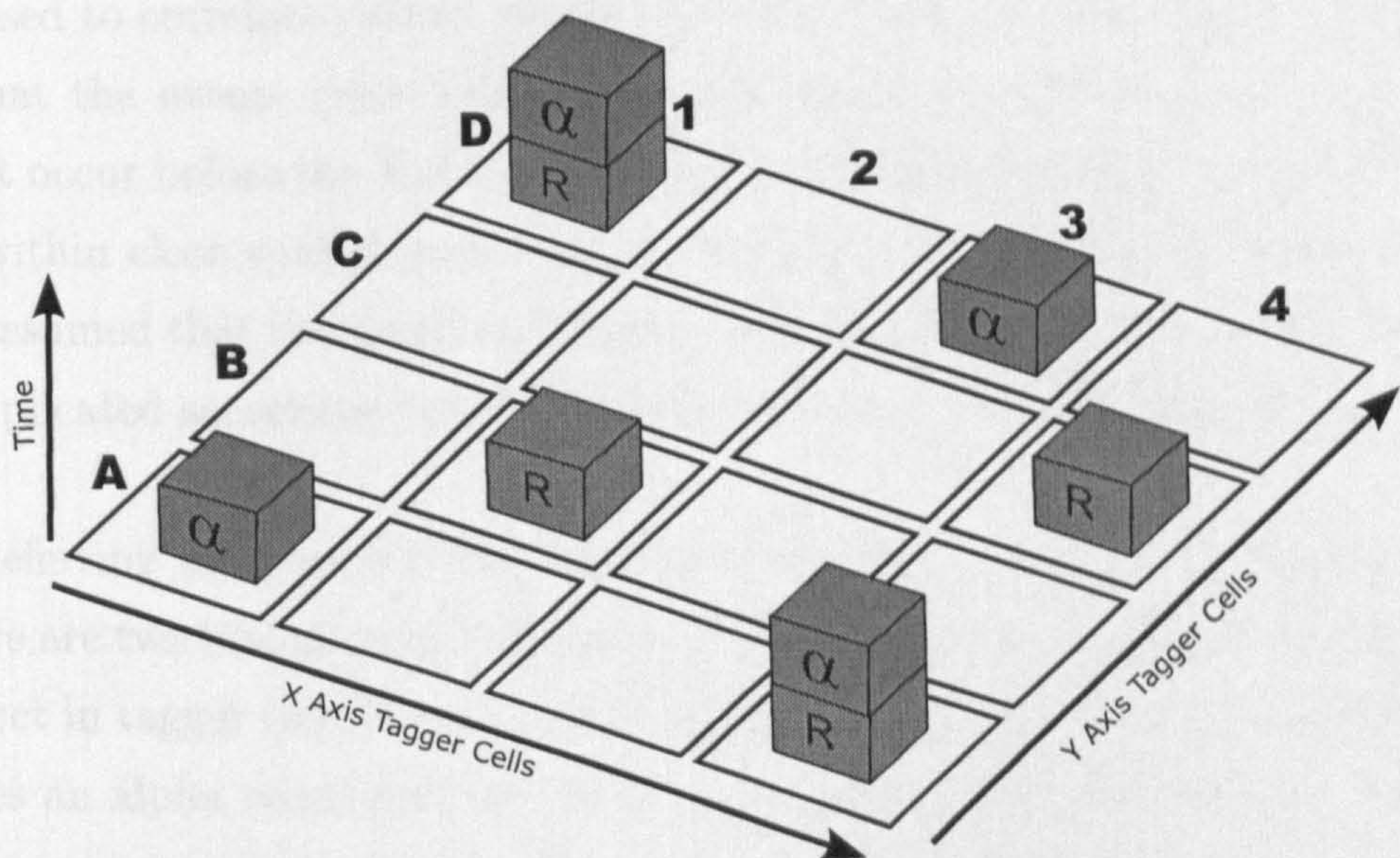


Figure 6.11: Figure showing a generic tagger with multiple cells.

One of the key delayed coincidences that is usually required is that of recoil alpha tagging. In order to perform this tagging it is necessary to be able to distinguish events that correspond to an alpha decay and events that correspond to recoil implantation. A simple way of doing this is use

the gas detector (MWPC) as a recoil discriminator. Essentially a recoil must pass through the gas detector, therefore any event that has a valid gas energy value i.e. has deposited energy in the gas detector is potentially a recoil. It is possible to now apply certain energy conditions dictated by the experimental parameters of the reaction being studied and positively identify a recoil event. Conversely if an event does not contain a gas energy event it has not passed through the gas detector and has therefore originated from a nucleus already embedded in the DSSD. Again energy restriction are imposed based on the expected energy of the alpha particle.

6.4.5 Recoil Alpha Tagging

Now that a method to distinguish recoil generated events from alpha generated events has been identified it is possible to discuss how the tagger can be used to correlate related events. The first property that can be identified is that the events must follow a specific sequence in that the recoil event must occur before the alpha event. Another property is that the events must lie within close spatial proximity of each other. In this first instance it can be assumed that the recoil and alpha events must be in the same pixel, more complicated searching strategies are discussed in the following section.

Referring to figure 6.11 and keeping the above in mind it can be seen that there are two instances in the diagram that meet these criteria, the **CTagger** object in tagger cell $D1$ and the **CTagger** object in cell $A4$. In both of these cases an alpha recoil pair has been identified that can be used as a basis to extract physics information. One possible use is to look for gamma rays that are only related to this specific decay by extracting the list of gamma rays associated with the alpha event and hopefully build up information about the level structure of the nucleus.

In identifying these transitions the specialised sort code needs to be structured in a certain way. As an event is received in the **processEvent()** method it needs to be tested to see if it is a potential alpha event or a recoil

event. The test is based on the presence or absence of a gas energy event as well as specified energy criteria as discussed above. If the event is identified as a recoil a flag is set in the **CEvent** class specifying that this event is a recoil event. This event is then added to the tagger in the appropriate pixel coordinates. If the event object is identified as being neither a recoil event nor an alpha event i.e. an 'other' event it is appropriately flagged 'other' and added to the appropriate tagger.

If the event object is identified as an alpha event it is first appropriately flagged as an alpha event. After retrieving the relevant DSSD X and Y coordinates from the **CEvent** object it is then possible to retrieve a list of all events from that tagger cell by calling the `get()` method of the tagger manager. This method returns a list of all events that have been added to the specified cell with respect to the time of the last event added and the tagger depth. This list is then iterated, if any recoil events are found within it then a valid recoil alpha pair has been identified. At this point further filtering and data extraction can be performed according to the remit of the experiment.

Recoil-Alpha Tagging Examples

Given the method that has just been described, the code in figure 6.12 gives an example sort file showing how a recoil alpha tag can be performed. The source code is an extract from a real sort file, extraneous lines of code relating to graphing and other tests not relevant to the current discussion have been omitted.

The code starts by first checking if the event is an alpha particle, in this example alphas are identified by any event that has a pixel energy between $8100keV$ and $8220keV$ and has no gas data associated with it. The next stage is to retrieve a list of all events in the tagger manager for this pixel and search back through them looking for all recoil events. In the example, recoils are identified by events that have a gas data item value and also pass

```
1   void MySorter::processEvent(CEvent* e)
2   {
3       EventProperties EventData;
4       e->GetProperties(&EventData);
5
6       //if its an alpha particle search for recoils
7       if( (EventData.XEnergy > 8100) && (EventData.XEnergy < 8220)&&
8         (EventData.gasE < 10) )
9       {
10          vector<CEvent*> eventList=Tagger->get( e->getX(),e->getY() );
11
12          //loop over the arraylist from the tagger and check for recoils
13          for(int i =0 ; i<eventList.size() ; i++)
14          {
15              CEvent* taggedEvent=eventList[i];
16              EventProperties taggedEventData;
17              taggedEvent->GetProperties(&taggedEventData);
18
19              //check to see if this is a recoil
20              if ((taggedEventData.gasE>1500)&&(taggedEventData.gasE<5000)&&
21                (taggedEventData.XEnergy>5000)&&(taggedEventData.XEnergy<15000)
22                &&(taggedEventData.tac>5600)&&(taggedEventData.tac<7200) )
23              {
24                  //this is a recoil
25                  recoilsx->Fill(taggedEventData.XEnergy);
26                  tagalphax->Fill(EventData.XEnergy);
27              }
28          }
29      }
30      else
31      {
32          //put everything else into the tagger
33          Tagger->add(e);
34      }
35
36  }//end of processEvent method
```

Figure 6.12: Figure showing the **processEvent()** specialised sorting code.

a basic DSSD and TAC gate defined by the values in the if statement. If any event in the list meets this criteria then a recoil alpha pair has been correctly identified.

Now the relevant histograms are incremented with the DSSD energy values indicated by their appropriate **EventProperties** structure values. Figure 6.13 shows the identified recoils of the correlated pair and Figure 6.14 shows the identified alphas of the correlated pair. Comparing these two figures with previously generated histograms shows that using the tagging method provides a much cleaner identification of the alpha and recoil components than simply relying on the presence or absence of a gas data item.

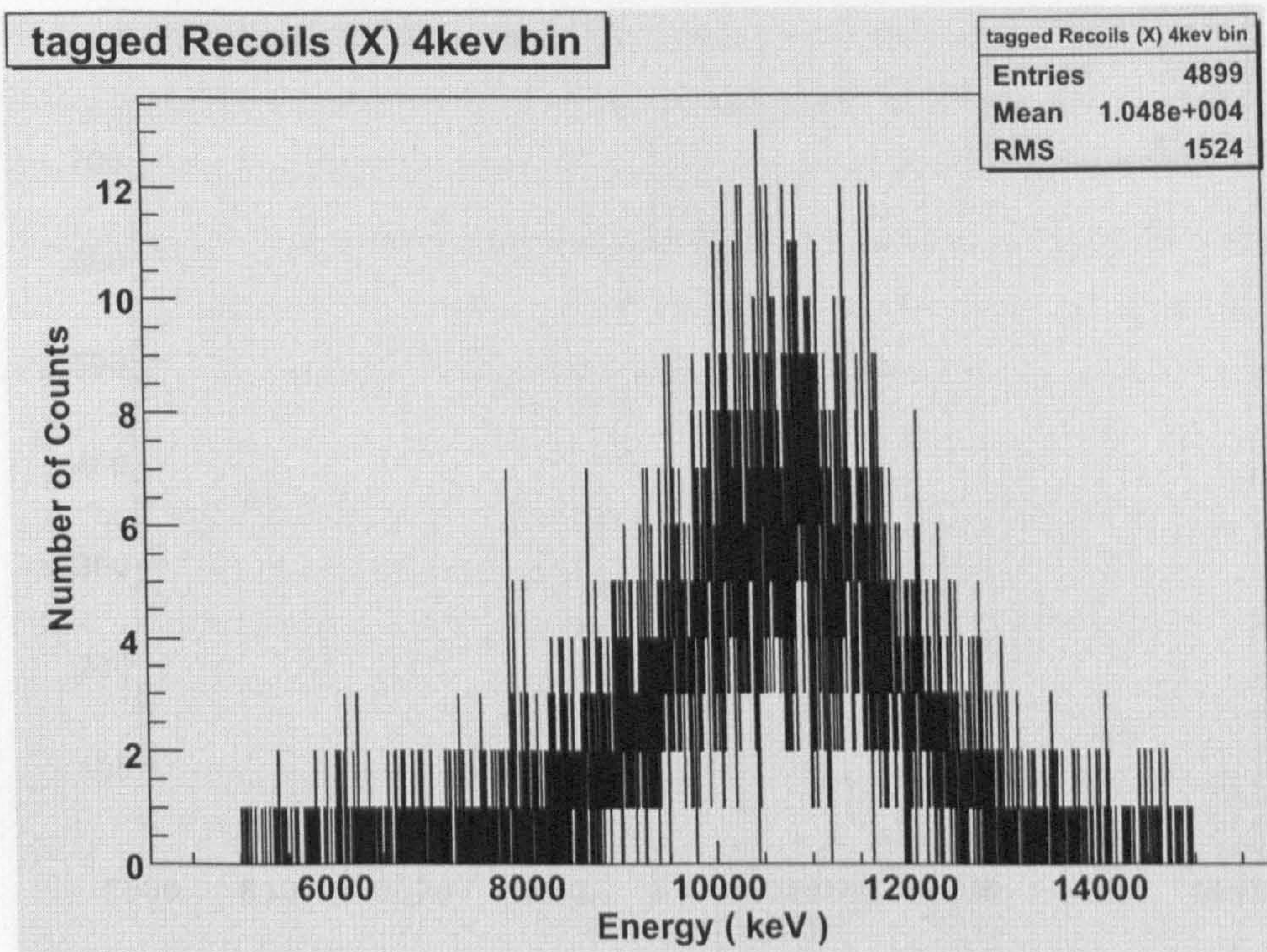


Figure 6.13: Figure showing the alpha tagged recoil energy.

Example Event

It is useful at this point to provide an example of a real tagged event constructed from the data stream. Figure 6.15 shows an event that has been completely constructed and passed onto the **processEvent()** method during a run through the example data set and identified as an alpha. It also shows an event found from the tagger that meets the user defined criteria of a recoil. The figure shows a simplified version of the data items contained in the **CEvent** object, showing only the information relevant to the illustration.

6.5 Tagger Search Strategies

In the example discussed above delayed coincidences were only considered to be valid if the sequence of events required lay in the same cell of the tagger, however it is possible that the events may not be constrained to this one place. A possible cause would be if a recoil implanted in a pixel and the

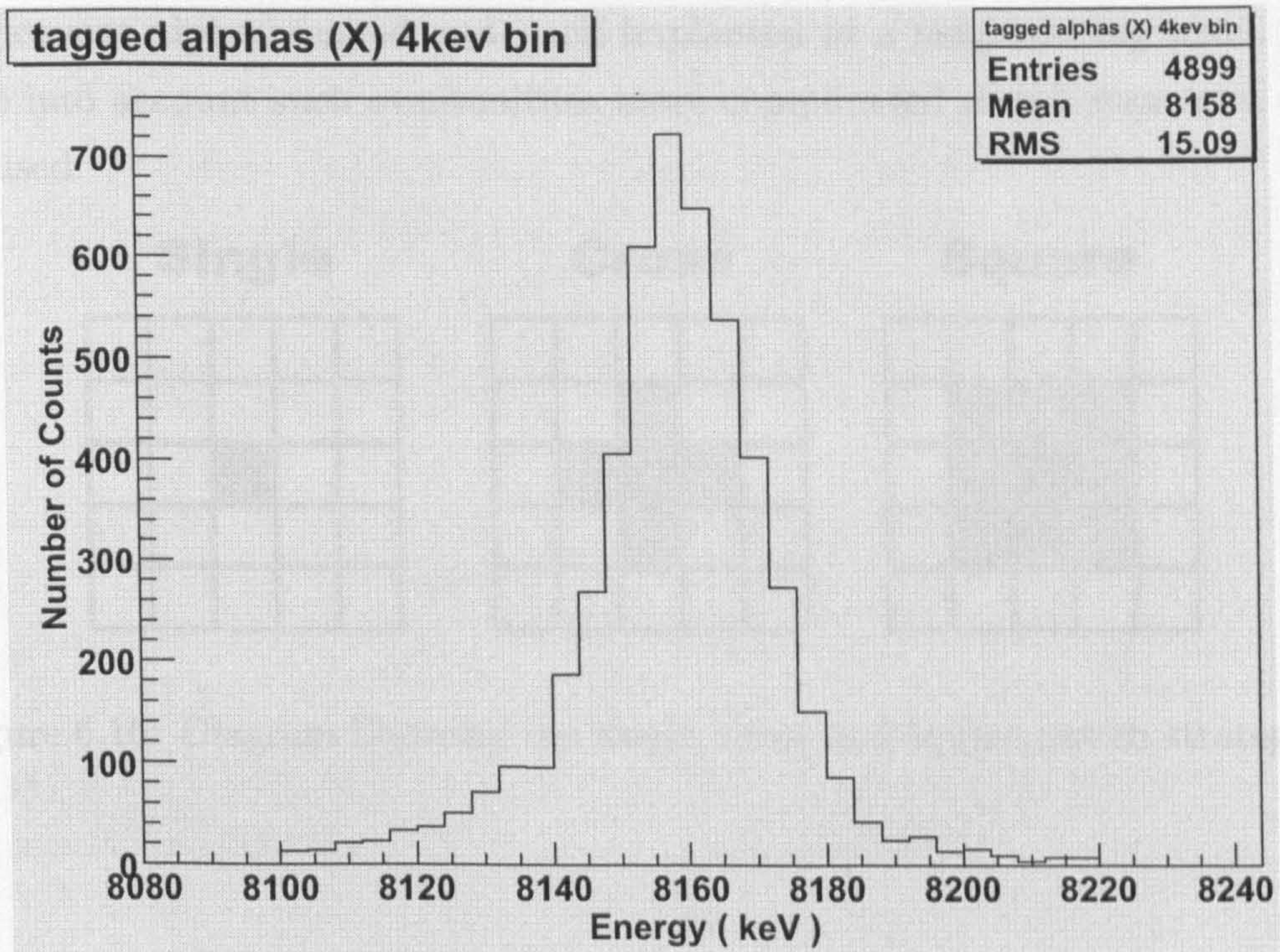


Figure 6.14: Figure showing the recoils tagged alpha energy.

ALPHA EVENT
XEnergy : 8140.41
YEnergy: 4100.16
XTime: 20749797510850
YTime: 20749797510850
gasX1: -1
gasX2: -1
gasY1: -1
gasY2: -1
gasE: -1
gasT: -1
x: 92
y: 0
PLANAR E : 158.971

RECOIL EVENT
XEnergy : 9638.06
YEnergy: 4100.67
XTime: 20749062929448
YTime: 20749062929448
gasX1: 11226
gasX2: 3889
gasY1: 10721
gasY2: 9737
gasE: 2466
gasT: -1
x: 92
y: 0
CLOVER E : 6096.91
CLOVER E : -1
PLANAR E : 3.57883
PLANAR E : 3.17846
PLANAR E : 424.08

Correlation Time : 734581402 (ticks) = 7.35s

Figure 6.15: Outline showing the components of a real **CEvent** object constructed from the data stream during a run through the example data set.

subsequent alpha escaped and itself implanted in a neighbouring pixel. To take into account such eventualities more complicated search strategies can be used.

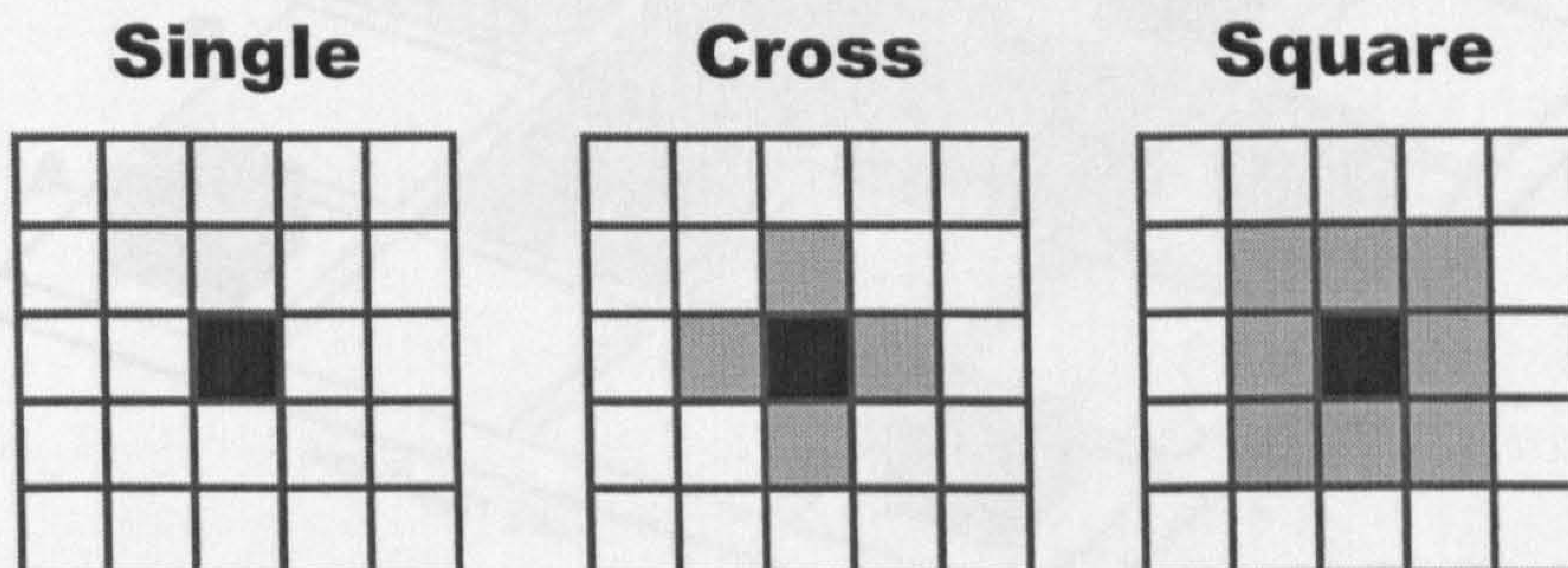


Figure 6.16: Diagram Showing the single, cross and square search strategies.

Figure 6.16 shows three possible search strategies that may be used to identify event sequences for delayed coincidences. The first strategy is the single cell (or pixel) strategy where the events must fall within the same pixel to be considered. The details of this method have been discussed previously. The next search strategy is the cross, in this case events are considered if they fall within the cross like pattern shown in the diagram, this pattern represents the most likely adjacent cells where an escaping alpha could be embedded.

Referring to figure 6.17 and considering the following sequence of events; the recoil event at tagger cell $B2$ is generated due to a recoiling nucleus embedding at that pixel of the DSSD at time $t = 100ns$. In the **processEvent()** method of the specialised sorting class this event is identified as a recoil type event and then entered into the tagger manager at the $B2$ coordinate position. At a time $t = 300ns$ another event is passed into the **processEvent()** method. This event is identified as an alpha type event so it is now possible to start searching for delayed coincidences. This is acheived by calling the **get()** method of the tagger manager supplying the coordinates of the cell of interest.

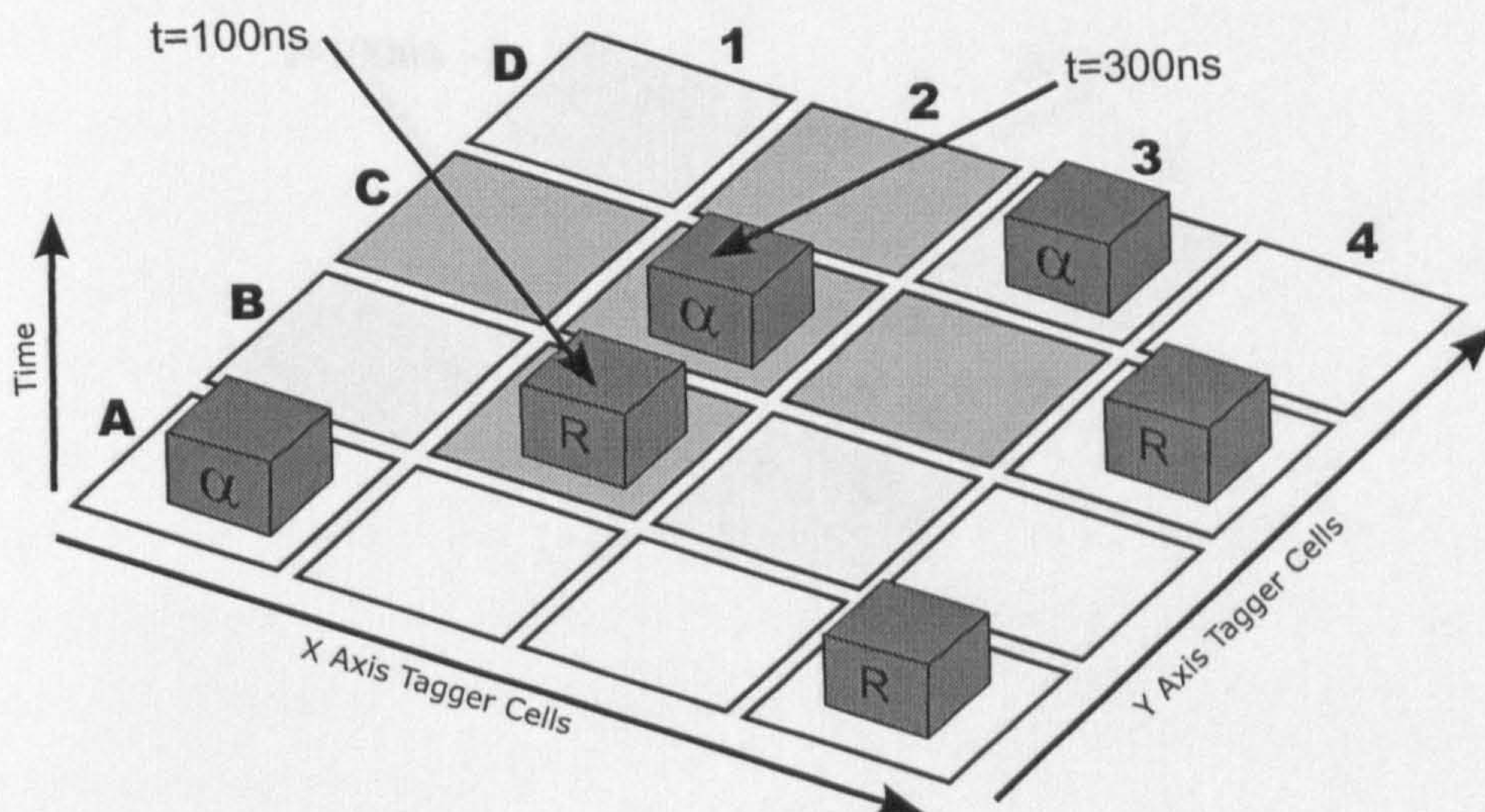


Figure 6.17: Figure showing a generic tagger with multiple cells and using a cross search strategy.

The call to the **get()** method returns a list of all the relevant events from the tagger manager. As the search strategy is set to cross this returns a time ordered list of events from the cells shaded in figure 6.17 i.e. cells $C1$, $C2$, $C3$, $D2$ and $B2$. As can be seen from the figure this will result in the recoil occurring at time $t = 100ns$ at cell $B2$. This has now established these two events as a recoil alpha pair that can now be used for further analysis.

The final search type discussed here is the square search strategy. As the name implies this method searches for all possible delayed coincidences in a square shaped pattern around the originating pixel. The square search strategy is shown in figure 6.18 which shows how events can be correlated. Considering that the recoil event at time $t = 100ns$ is added to tagger cell $B1$ and in a subsequent call to the **processEvent()** method the event at cell $C2$ with time $t = 200ns$ is added then these two events can be correlate using the square search strategy. The call to the tagger manager that retrieves all relevant events will return events from the following cells $D1 - D3$, $C1 - C3$ and cells $B1 - B3$. As can be seen this will result in the return of the recoil event at cell $B2$ which can be used to define a delayed coincidence pair for

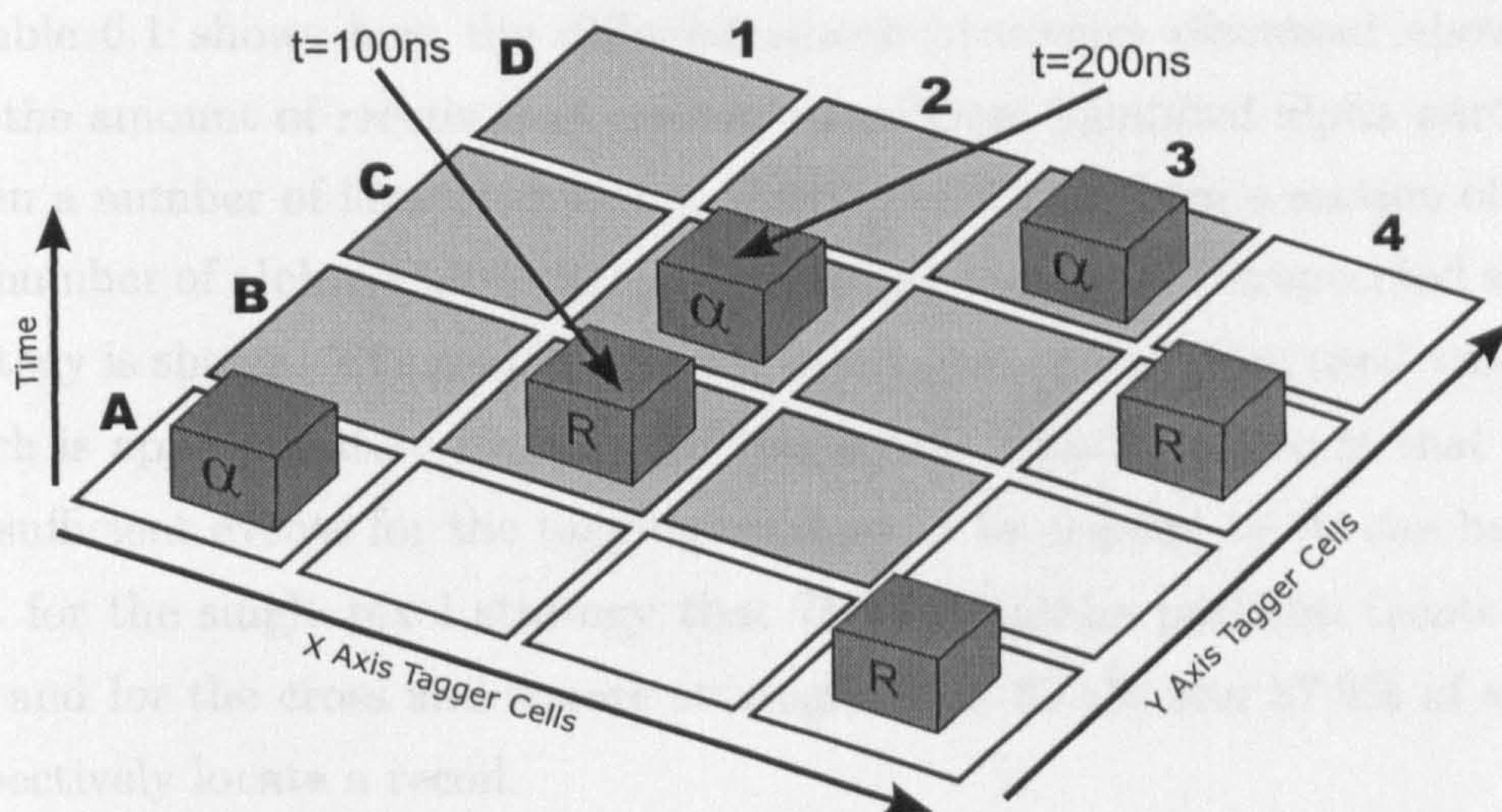


Figure 6.18: Figure showing a generic tagger with multiple cells and using a square search strategy.

further analysis.

Referring again to figure 6.17 and 6.18 a situation can be seen were a given alpha event would not find a corresponding recoil in the cross strategy but would find one if the square strategy was used. The alpha event in cell $D3$ when using the cross search strategy finds no recoil event, however the same alpha event when using a cross search strategy would find the recoil event in cell $C4$. From this it can be reasonably expected that the numbers of correlated recoil alpha pairs will be different for different strategies.

Raw α	Single	Cross	Square	
1523	1203	1304	1339	Number of α particles
n/a	78.9%	85.6%	87.9%	% of correlated recoils

Table 6.1: Table showing the number and percentage of recoils correlated from the detected alpha particles for single, square and cross tagging search strategies.

Table 6.1 shows how the different search strategies discussed above affect the amount of recoils that are correlated from identified alpha particles. Given a number of identified alpha particles ($1523\alpha s$) from a section of data the number of alphas that locate a recoil in the tagger for the specified search strategy is shown. The search time for recoil alpha correlation used was $550s$ which is approximately $10 \times T\frac{1}{2}$ for the decay. This is to ensure that there are sufficient events for the tagging method to be applied to. It can be seen that for the single pixel strategy that 78.9% of alpha particles locate a recoil and for the cross and square strategies that 85.6% and 87.9% of alphas respectively locate a recoil.

Given the values above it is useful to mention the relative benefits of each of the strategies. The single strategy which correlates 78.9% of recoils is the fastest in terms of execution speed. Compared to the square strategy which identifies 87.9% of recoils but is the slowest to execute due to the eight fold increase in tagger searches. The cross strategy is an excellent compromise however as it only has a four fold increase in tagger searches but more importantly only correlates 2.3% less recoils than the comprehensive square search strategy.

6.6 Lifetime Calculations

In this section the data gathered from the recoil alpha tagging section is used to calculate a real physical property of a nucleus. In the following the lifetime and half life of an alpha decaying nucleus will be calculated. In the example data set used throughout this thesis a prime candidate for this calculation is the alpha decay of ^{254}No to ^{250}Fm. The values of the lifetime and half life are well known [18] i.e. the the half life $T\frac{1}{2} = (51.2 \pm 0.4)s$

To calculate the lifetime or half life of an alpha decaying nucleus it is first necessary to be able to identify a recoil alpha pair and gather information about their timing relationship. A segment of specialised sorting code similar to the code section given in figure 6.12 is used to identify the recoil alpha

pairs needed. Once they have been identified a difference of the two events timestamps is taken. This is accomplished in the specialised sorting code by inserting the following lines (figure 6.19) at line 26 of the code outlined in figure 6.12

```
1    __l64 dt = (EventData.XTime - taggedEventData.XTime);
2    recoilalphatime->Fill((dt/(10*Constants.Seconds)));
```

Figure 6.19: Section of code inserted into the specialised sorting class to calculate and plot the time difference between a correlated recoil alpha pair.

Referring to figure 6.19 it can be seen that the additional lines of code needed in the specialised sorting class are fairly straightforward, line 1 simply takes the time-stamp of the DSSD X strip of the current event (the identified alpha) and subtracts the time-stamp of the DSSD X strip of the recoil event located by a tagger search. The calculated differences are then plotted into a histogram. It can be seen from line 2 of the code that the time difference calculated in tens of nanoseconds is scaled to fit into $10sec$ bins which is of the same order of magnitude as the expected half life. The results of this process can be seen in figure 6.20.

As radioactive decay is governed by the well known law [19] given in equation (6.1) fitting an exponential curve to the generated data can yield a value for λ. By using equations (6.2) and (6.3) values for the half life and lifetime can then be calculated from the decay constant λ.

$$N(t) = N_0 e^{-\frac{t}{\tau}} \tag{6.1}$$

$$T_{\frac{1}{2}} = \frac{ln(2)}{\lambda} \tag{6.2}$$

$$\lambda = \frac{1}{\tau} \tag{6.3}$$

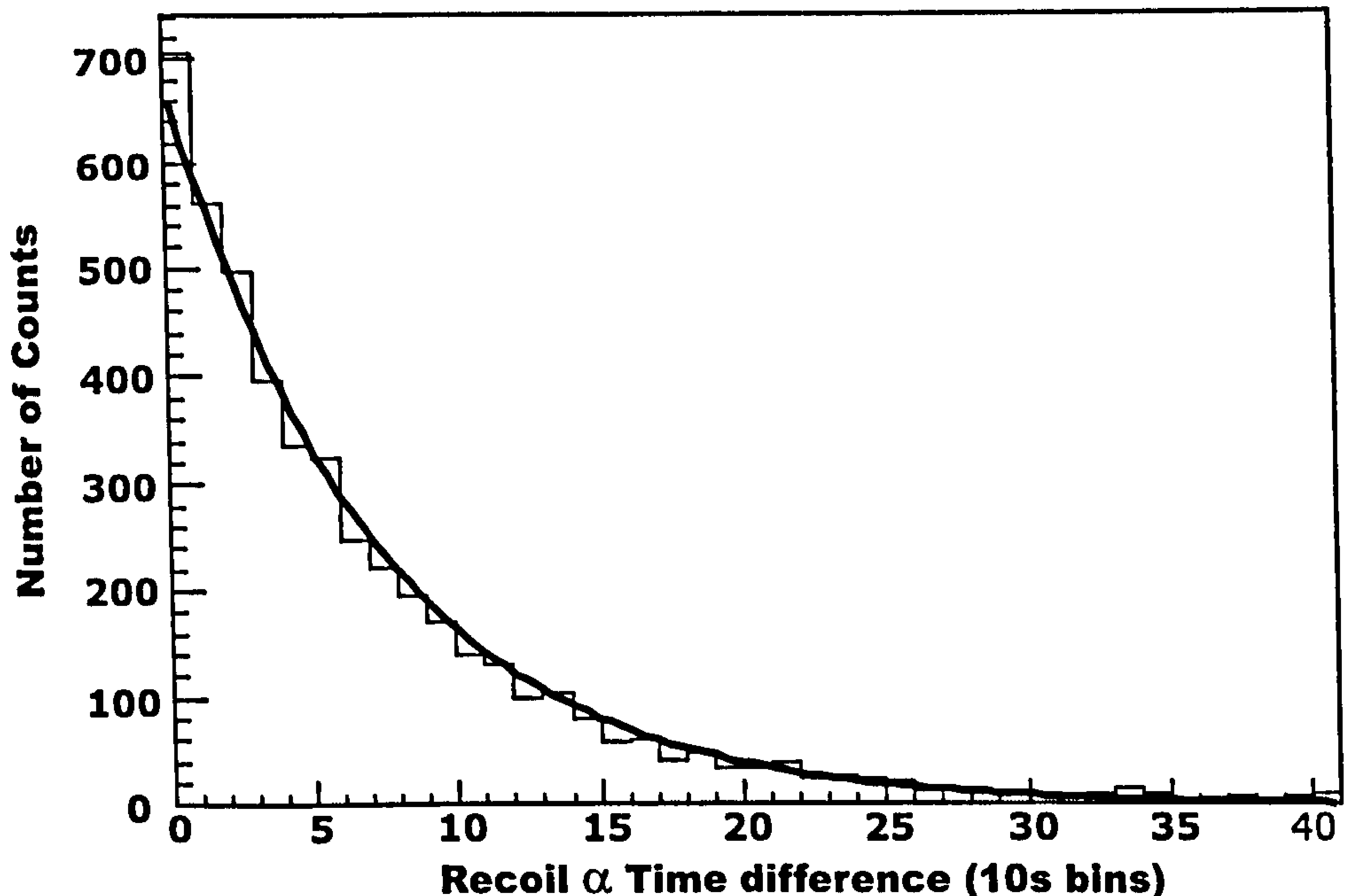


Figure 6.20: Figure showing a decay curve for the ^{254}No alpha decay. A exponential fit of the curve is shown.

The result of an exponential fit to the data is shown in figure 6.20. The result of this fit yields a value for λ as $(0.0130 \pm 0.0002)s^{-1}$. Using equation (6.2) and equation (6.3) gives a half life ($T_{\frac{1}{2}}$) of $(53.5 \pm 0.8)s$ and a lifetime τ of $(77.2 \pm 1.1)s$. The calculated half life compares well with the expected value of $(51.2 \pm 0.4)s$.

6.7 Comparison of TDRSorter to GRAIN

During the process of developing the TDRSorter data analysis code and the subsequent writing of this thesis other data analysis tools to work with the TDR data acquisition system have been developed in other institutions. One of these systems known as GRAIN [21] was developed by Panu Rahkila at the university of Jyväskylä. GRAIN has been developed in Java to handle the same time ordered data stream produced by the TDR data acquisition system that is analysed by the TDRSorter code.

It is beneficial to compare the TDRSorter program to GRAIN as it provides a useful cross check on the validity of any results generated from either program. In the comparisons shown below a separate set of data was used than in the analysis in previous sections. This separate data set was used due to changes made in the experimental setup; the channel mappings of individual detectors was altered before the initial release of GRAIN. This means that processing the data set used previously (containing the old detector mappings) GRAIN would incorrectly assign channel numbers in the data stream to their respective detectors. These detector mappings can not be changed due to the fact that GRAIN is a closed system.

The solution to this issue is to use a more recent data set that so that GRAIN has the correct detector channel mappings. It is easy to update the mappings in the TDRSorter data analysis program due mainly to the fact that access to the source code is not restricted. An additional point to note is that this more recent data set is from an experiment designed to investigate the same nuclei as used in previous examples i.e. ^{254}No.

A comparison of each of the main stages passed through in order to obtain the half-life and lifetime values for the alpha decay of ^{254}No was chosen to show that both TDRSorter and GRAIN are consistent with each other and established values. The first fundamental step in being able to calculate the half-life and lifetime is to correctly identify and distinguish recoiling nuclei and alpha particles. This is also the first step in comparing the two data analysis programs.

Figure 6.21 shows the identified recoils and alphas for both the TDRSorter and GRAIN data analysis programs. The method of identification is the same as described in section 6.3.4 i.e. that the recoils are identified by DSSD events in coincidence with energy loss events in the gas detector (MWPC) and alphas are identified by DSSD events in anti-coincidence with events in the MWPC. Figure 6.21(a) and 6.21(b) show the identified recoils for the TDRSorter and GRAIN data analysis programs, respectively. The results

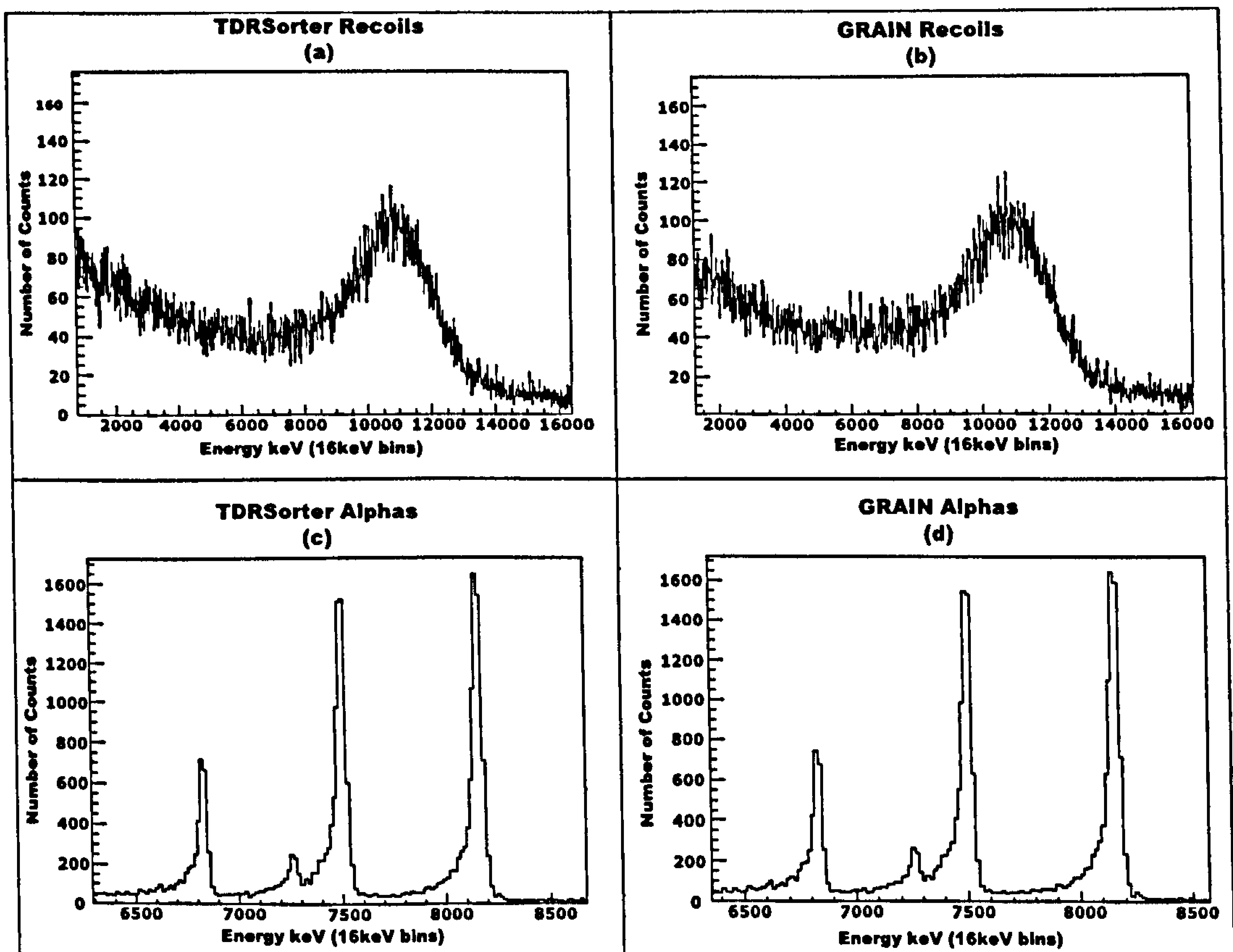


Figure 6.21: Comparison of recoil and alpha spectra for both GRAIN and TDRSorter data analysis programs. In both cases recoils are identified by a gas coincidence and alphas are identified by a gas anti-coincidence.(a) shows the recoils identified by the TDRSorter program, (b) shows the recoils identified by GRAIN, (c) shows the alphas identified by TDRSorter and (d) shows the alphas identified by GRAIN.

from both programs can be seen to be the same and it can also be seen that in both cases the alpha particles that should lie in the energy range of $6500keV$ to $8500keV$ have been correctly filtered from the recoils. Figure 6.21(c) and 6.21(d) show the identified alphas for the TDRSorter and GRAIN data analysis programs, respectively. In both cases the alpha peaks can be seen to lie at the same energy and with similar peak heights. It can also be seen that the alphas have been correctly separated from the recoils by the process described above.

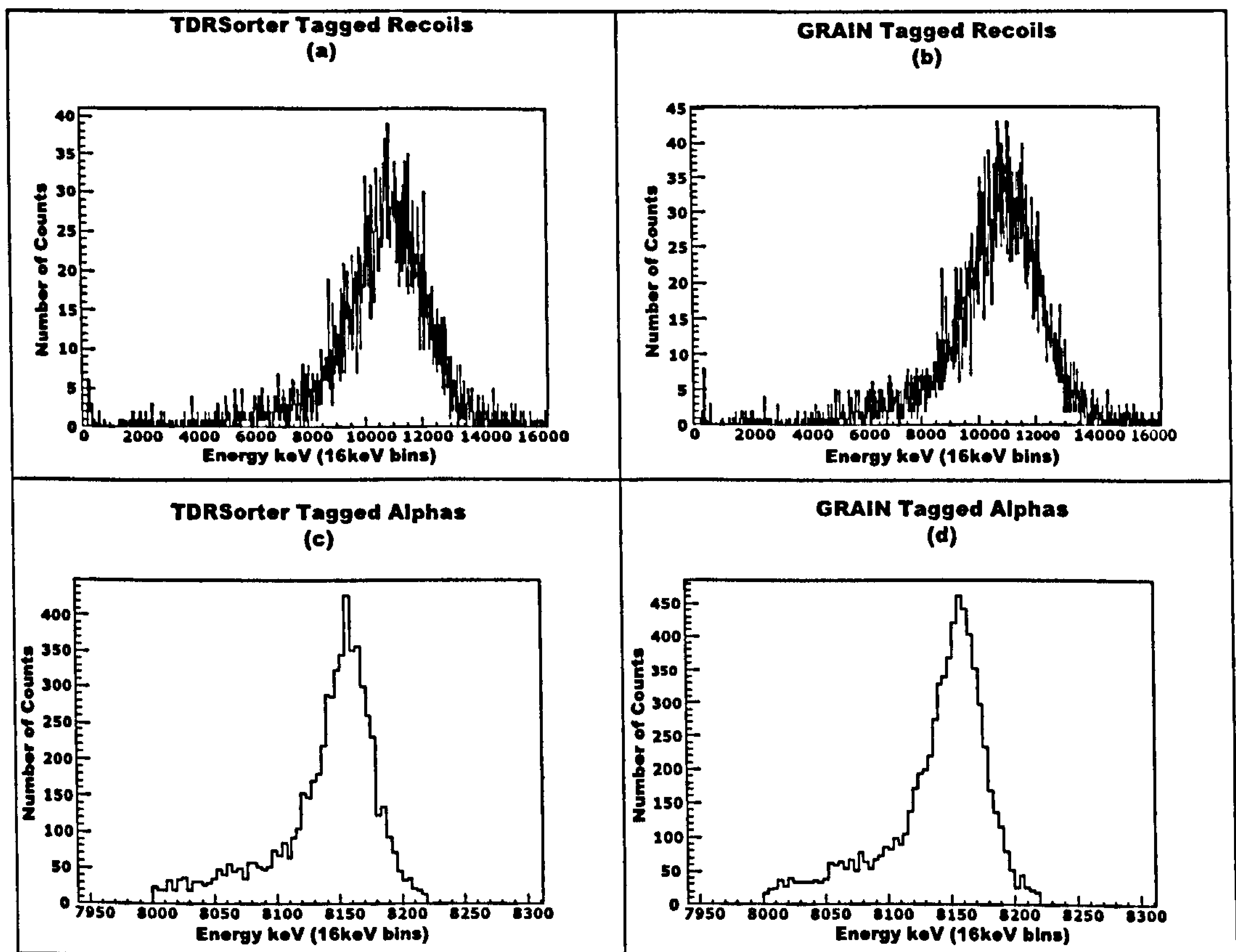


Figure 6.22: Comparison of tagged recoil and alpha spectra for both GRAIN and TDRSorter data analysis programs. (a) shows the alpha tagged recoils identified by the TDRSorter program, (b) shows the alpha tagged recoils identified by GRAIN, (c) shows the recoil tagged alphas identified by TDRSorter and (d) shows the recoil tagged alphas identified by GRAIN.

The next step to be able to calculate the half-life and life time is to correctly correlate recoil and alpha pairs. This is performed by recoil alpha tagging as described in section 6.4.5. Figure 6.22 shows the correlated alpha and recoil pairs identified by the TDRSorter and GRAIN data analysis programs. Figure 6.22(a) and 6.22(b) show the identified alpha tagged recoils for the TDRSorter and GRAIN data analysis programs, respectively. Figure 6.22(c) and 6.22(d) show the identified recoil tagged alphas for the TDRSorter and GRAIN data analysis programs, respectively. In both cases the figures show that the identified recoils and alphas are similar to each other.

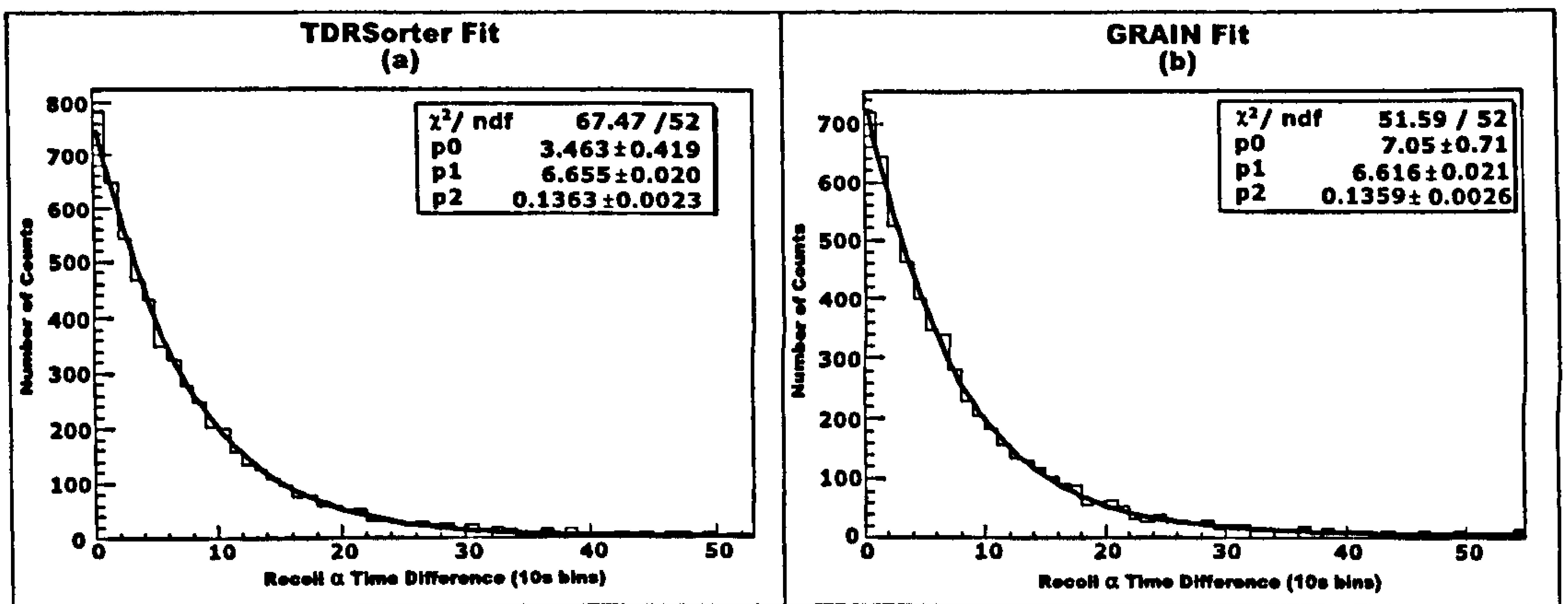


Figure 6.23: Comparison of decay curves for the ^{254}No alpha decay. (a) shows the fit results for the data sorted using the TDRSorter analysis program. (b) shows the fit results for the same data sorted with the GRAIN data analysis program.

The final step is to use these identified correlated alphas and recoils and calculate the time difference between them. Plotting this time difference for all recoil-alpha pairs produce a decay curve that can be used to calculate the half-life and lifetime of the ^{254}No alpha decay. The process and calculations performed are identical to those in section 6.6. Figure 6.23 compares the fitted decay curve for the TDRSorter data analysis program (Fig.6.6(a)) and for the GRAIN data analysis program (Fig.6.6(b)). For both cases the same fit is performed using the same fitting program i.e. ROOT. The fit used is an exponential curve with a linear background, which provides the best fit for the data. In both cases the coefficients of the fit are shown as well as the χ^2/ndf (number of degrees of freedom). It can be seen from these χ^2 values that the fit performed with the data gathered from GRAIN 6.23(b) is 'better' than the fit for the TDRSorter package. Despite this difference in goodness of fit, the calculated values of half-life and lifetime are still very consistent.

Using these fits it is now possible, using the formula given in section 6.6 to calculate the half-life and lifetime of the ^{254}No alpha decay. Table 6.2 shows the calculated half-life and lifetime along with errors for both the

Program	$T_{\frac{1}{2}}(s)$	err (s)	λ s^{-1}	err s^{-1}
TDRSorter	50.9	0.9	73.4	0.2
GRAIN	51.0	1.0	73.6	0.2

Table 6.2: Table showing the calculated values of half-life and lifetime. The table compares values for the GRAIN and TDRSorter data analysis programs.

TDRSorter and GRAIN data analysis programs. The values shown in the table are consistent both with each other and with the expected value of $(51.2 \pm 0.4)s$.

This comparison has served as a useful cross check as to the validity of both the TDRSorter and GRAIN data analysis programs. As can be seen by the previous examples both packages produce results that are consistent with each other. In most cases it cannot be expected that both programs produce identical results. Differences in design primarily in the algorithm used for data buffering and tagging will cause differences in the produced spectra. Without careful analysis at the source code level it would be impossible to attribute differences to any given design decision and therefore be able to make decisions about which method is the most acceptable or accurate. Given that both programs produce consistent results it shows that the methods used are suitable for extracting useful information from the triggerless data generated by GREAT and the TDR data acquisition system.

6.8 Summary

This chapter has discussed how a user defined class was derived from the base **CSorter** class and how **CEvent** objects passed to it where then used in the specialised sorting section to extract physics information. The tagging methodology used to perform delayed coincidences was discussed as well as different tagging search strategies that were used. Also briefly discussed was how data gathered from the specialised sorting section can be visualised using

histograms or other third party visualisation libraries.

This chapter has also provided some examples of how the techniques discussed are used to produce meaningful and useful results. Basic tagged spectra have been shown indicating how recoiling nuclei can be discerned from alpha decays in the DSSD. A calculation of the half life and lifetime of ^{254}No was also shown, proving that this data analysis method based on triggerless data streams is viable for performing real world analysis.

Results generated from the TDRSorter data analysis program were compared to results generated from the GRAIN data analysis program. In both cases the same data set was used along with comparable sort programs. This comparison provided a useful cross check to validate the results generated from both programs.

Chapter 7

Conclusions

This chapter will briefly discuss whether the TDRSorter data analysis code has met the goals set out in the introduction. An overview of the advantages, disadvantages and limitations of the TDRSorter data analysis code implementation is also given. A comparison of the TDRSorter as it relates to other software solutions available is discussed and finally a brief overview of possible future improvements of the code base is also given.

Throughout this thesis, real world examples of the TDRSorter data analysis code being used on data gathered from a real experiment have been given. In all these instances the results produced from the code have been borne out by what is expected from the physics of the various specific situations. It can be seen through this that the TDRSorter class can be seen to produce accurate results.

The TDRSorter data analysis code was designed with flexibility and control in mind. As the code is split off into separate classes that handle individual areas of responsibility it is easy to customise specific components to perform specialist functions that are required by any given experiment. As long as the interface that the component shares with classes that it uses and are used by remains the same then a given component can be rewritten and used with the rest of the framework of the application without needing to alter other components. This ability can be seen as one of the advantages of

the TDRSorter data analysis code. If a specific experiment requires that a specialised trigger is needed it can be rewritten easily as all the code for the implementation is freely available. For example a custom trigger could be required that does not build events from a silicon-gas TAC or an x or y pixel in the DSSD but instead triggers from a focal plane gamma ray or electron.

In addition to the ability to rewrite core components is the amount of customisation that is available whilst using the inbuilt settings mechanism. Most of the variables used in the set-up of the data analysis code are read in from a settings file at run time, it is therefore easy for a user to tweak the parameters of event construction to be in line with the requirements of any given experiment. For example the values of the forward and backward time windows of the **CBuffer** class are read in from the settings file.

One of the main design choices taken in building the TDRSorter data analysis code was to integrate with ROOT to provide a presentation layer for visualising histograms and other data. This decision meant that the data analysis was somewhat decoupled from the presentation of data although this is compensated by the fact that is easy to integrate with other third party libraries. As ROOT is cross platform the presentation of TDRSorter data will work on both windows and Linux systems with little or no change. Lack of availability of a cross platform graphical user interface toolkit for c++ that was robust enough to meet the requirements meant that a custom visualisation layer that was integrated closely to the data analysis code was not implemented.

As briefly discussed in section 6.7 during the development of the TDRSorter data analysis code another data analysis tool called GRAIN was developed. GRAIN was developed with a different focus in mind and differs in design in several areas. It is useful at this point to discuss the two tools and compare their various strengths and weaknesses.

GRAIN was developed to provide easy online or nearline sorting to perform online checks on the running experiments. As such it has been used by a wide user base and has been continually expanded to the point where it has become the main analysis tool used in many institutions. Written in Java, GRAIN can be ran from any machine that has the Java virtual machine installed, this differs from the TDRSorter data analysis code which needs to be recompiled to work on different platforms.

GRAIN is designed to handle most main stream experiments, it however requires expert intervention if a non standard experiment is to be analysed e.g. angular correlation experiments. The process of event construction is hidden from the user. If specialised buffering or triggering is needed it is impossible for the user to write a customised component to perform the operations as required by the parameters of the experiment. In essence the only information the user is given is a pre-packaged event that is similar to the **CEvent** class provided in the **processEvent** method of the specialised sorting code. In general use this limitation may not be great, but in more specialised circumstances it becomes easier to use the TDRSorter data analysis code.

Having two data analysis codes that can be used to analyse the same set of data gives us the ability to perform cross checks on the veracity of the results gathered from running data through either system. The cross check performed in section 6.7 showed that the two data analysis programs produced visually similar histograms. The results of the compared half-life and lifetime calculations were both consistent with each other and with the accepted value.

Any system can be improved upon and during the development of the TDRSorter data analysis code some areas have presented themselves where future development may improve the overall system. Apart from general improvement of the efficiency of the sorting algorithms one area that could be improved upon is the area of integration. An ideal solution would be to

provide a complete tool that included an inbuilt visualisation layer that could be used to show histograms and other data without resorting to other third party solutions. Improving on this area would significantly improve the user friendliness of the data analysis code.

Additional improvements could also be made in the sorter set-up area, a graphical user interface could be supplied to alter the various values used internally by the sorting classes. Currently this functionality is provided by having to manipulate text based configuration files. Other areas of improvement could be to provide inbuilt functionality to perform calibrations on the data, thus preventing the need for users to resort to other third party solutions to perform these necessary tasks.

The use of either data analysis system is a viable solution for the data analysis requirements of any experiment using the GREAT spectrometer and the TDR data acquisition system. In certain areas the TDRSorter code presents some advantages over GRAIN hopefully the code will be used in these more specialised areas were more control over the construction and triggering of events is needed.

Previous versions of the TDRSorter data analysis code and preliminary results produced from the code have previously been presented at the IOP conferences in 2003 and 2004 and also in poster format at the University of Liverpool.

Bibliography

[1] Page R D et al 2003 Nucl. Instrum. Methods B 204 634

[2] Lazarus I H et al 2001 IEEE Trans. Nucl. Sci. 48 567

[3] Nilsson S G et al 1968 Nucl. Phys. A 115 545

[4] Nilsson S G et al 1969 Nucl. Phys. A 131 1

[5] Hofman S et al 1996 Z. Phys. A 354 229

[6] Oganessian Yu Ts et al 1999 Phys. Rev. Lett. 83 3154

[7] Oganessian Yu Ts et al 2001 Phys. Rev. C 63 11301R

[8] Hofmann S and Münzenberg G 2000 Rev. Mod. Phys. 72 733

[9] Simon R S et al 1986 Z. Phys. A 325 197

[10] Paul E S et al 1995 Phys. Rev. C 51 78

[11] M. Leino et al., Nucl. Instr. and Meth.. B 99 (1999) 653.

[12] Herzberg R-D J. Phys. G: Nucl. Part. Phys. 30 (2004) R123-R141

[13] `http://npg.dl.ac.uk/documents/edoc000/#GREAT`

[14] `http://npg.dl.ac.uk/documents/edoc504/edoc504.htm`

[15] `http://nnsa.dl.ac.uk/MIDAS/DataAcq/TSformat.html`

[16] M Fowler, K Scott: UML Distilled: A Brief Guide to the Standard Object Modeling Language (Object Technology S.). Addison Wesley. ISBN:0321193687

[17] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also `http://root.cern.ch/`.

[18] R-D Herzberg et al. Private communication and Physica Scripta, accepted for publication.

[19] K Heyde: Basic Ideas and Concepts in Nuclear Physics. (Second Edition) Institute of Physics Publishing. ISBN:0750305355

[20] B. Stroustrup: The C++ Programming Language (Special Edition). Addison Wesley.(February 11, 2000) ISBN:0201700735

[21] Rahkila P `http://phys157.phys.ad.jyu.fi/grain/`

[22] `http://www.gnu.org/software/gcc/gcc.html`

[23] `http://msdn.microsoft.com/vstudio/`

Appendix A

TDRSorter Data Analysis Package Operation

A.1 Overview

This appendix is essentially a tutorial on how to setup, compile and use the TDRSorter data analysis code. A step by step guide on how to use the code as provided on the cd in appendix D is provided along with information on how to create specialised sorting classes and run the TDRSorter program.

A.2 Requirements

The TDRSorter data analysis code can be compiled to run on both linux and windows platforms. Any available c++ compiler should be able to build the code on either platform. Throughout the development of the data analysis code one compiler was tested on each system. On linux g++ [22] was used whereas on windows, various versions[1] of visual studio [23] was used as the compiler. In order to build TDRSorter on either windows or linux the system must meet the requirements imposed by the relevant compiler.

[1]Versions 6.0, .net 2002 and .net 2003 have been tested.

An important requirement for compiling the TDRSorter code as it is presented on the CD is the presence of a fully installed and working copy of the ROOT [17] framework. The current implementation of the data analysis code uses ROOT histograms as a means of visualising the analysed data. The ROOT framework is currently available on both windows and linux. The TDRSorter has been tested with Version 4.04.02 on windows and 4.02.00 on linux systems.

A.3 Sort File Creation

Before compiling and using the TDRSorter data analysis code a specialised sorting class must be created. This class is absolutely essential as the main **CSorter** class of the analysis code defines two pure virtual functions to be overridden in a class derived from the sorting classes. These functions essentially prevent the code from being compiled unless they are implemented in a class derived from the sorting class. Printed below is the minimal specialised sorting class that can be compiled.

```
1   #include "CSorter.h"
2   class MySorter : public CSorter
3   {
4   public:
5      MySorter();
6      void processEvent(CEvent* e);
7      void outputResults();
8   };//end of sort class
9
10  MySorter::MySorter() //Empty C++ constructor
11  {
12  }
13  void MySorter::processEvent(CEvent* e)
14  {
15  }
16  void MySorter::outputResults()
17  {
18  }
19  // entry point for TDRSorter Data Analysis program
20  int main(int argc,char *argv[])
```

```
21 {
22 MySorter* mysorter = new MySorter();
23 mysorter->Run(argc,argv);
24 return 0;
25 }
```

Line 1 of the specialised sorting class specifies the main data analysis code header (**CSorter.h**). This file is the only include file needed to get the code to compile. This header includes all the files necessary for the compilation of the rest of the components of the data analysis code as well as including the headers for some components of the ROOT framework. The next stage of declaring a specialised sorting class is to derive a class from the **CSorter** base class as indicated on line 2. This class must inherit and implement the two methods **processEvent()** and **outputResults()** from the base class. These methods are declared on lines 6 and lines 7 and there subsequent empty implementations are found on later lines.

The last section of this sort file that is of importance is on lines 20 to 25. This is essentially the entry point of the data analysis code. Line 20 is the standard c++ main function, where execution of the program begins. Line 22 creates an instance of the derived sorter class declared above and calls the run method. This starts the execution of the sorting code. An important point is that the command line arguments argc and argv contain the names and locations of the runfiles that the data analysis code is to be run on. It is important to pass the command line arguments from the main function to this run method of the sorting class.

A.4 Compilation with g++ on linux platform

In order to compile the **TDRSorter** data analysis code on the linux platform the g++ compiler is used. The commands below must be executed from the shell to successfully build the code. It is assumed in the following that all the source and header files are in the same directory or somewhere in the users path.

```
g++ -I /usr/local/root_v4.02.00/include/ -c *.cpp
g++ -L /usr/local/root_v4.02.00/lib/ -lCint -lHist
 -lCore -lMatrix -ldl -o TDR *.o
```

The first line executes the compilation stage of the g++ compiler which is specified by the $-c$ flag. The g++ compilation produces object code files indicated by the ".o" file extension. These object files are used by the linking stage of the process. The $-I$ flag indicates that the following directory path specifies a location where necessary include files can be found. In this case the path points to the include directory of the ROOT distribution, indicating where the necessary classes used by the TDRSorter data analysis code can be found. The final section "*.cpp" indicates that all the c++ code files present in the current directory are to be compiled.

The second line tells g++ to perform a link operation, the "-o" flag and produce an executable file called TDR. The link is performed on all "*.o" object files in the current directory. The $-L$ flag specifies the directory where any library files needed can be located. In this case it specifies the lib directory of the installed ROOT distribution. Any libraries to be linked with are specified by their name preceded with a $-l$ e.g. the Hist library is specified as $-lHist$.

A.5 Compilation with Visual Studio on the Windows Platform

Compilation on the windows platform is specific to the version of visual studio used. As the process differs, step by step instructions are not given here, instead a broad overview is given. In general a visual studio workspace or solution is created and the TDRSorter include and source files are added into it. The specialised sorting class is usually created in the "main.cpp" file and is also added. The library and header file directories of the installed ROOT distribution need to be added to the list of searched directories within

the visual studio environment. Assuming all of this is done it should be possible to compile the code by following the standard steps of the given environment.

A.6 Usage

Regardless of whether the code was compiled in a linux or windows environment the operation of the TDRSorter data analysis code is the same. The only caveat is that in the linux environment the executing program may not be able to use the ROOT dynamic libraries. To fix this program the following line is executed in the shell.

```
setenv LD_LIBRARY_PATH /usr/local/root_v4.02.00/lib/
```

This line sets the LD_LIBRARY_PATH environment variable which gives ld.so, the run-time shared library loader, an extra set of directories to look for a required shared library in.

To run the TDRSorter data analysis program the following command is used.

```
TDR runfile1 runfile2 ...
```

This command simply runs the TDRSorter data analysis code on the specified runfiles. All of the external files such as the calibration file "calib.dat", the veto file[2] "veto.dat", the threshold file[3] "threshold.dat" and the sorter configuration file [4] "sorter.cfg" by default are searched for within the current directory that the data analysis code is executed from. Examples of all of these files are present on the included CD.

[2]The veto.dat file specifies any channels to be excluded from the analysis

[3]The threshold.dat file specifies an energy threshold for a given detector channel that a calibrated energy signal for that detector must exceed to be used in the analysis

[4]The sorter.cfg file contains all of the run parameters of the TDSorter data analysis code e.g. tagger search depth, the use of a veto file etc.

Appendix B

TDRSorter Class Overview

B.1 Classes

This appendix contains a quick overview of the main functionality of each class. Although the names of each class where chosen such that their operation could be inferred from their title this section serves as a glossary to allow the main function of a class to be known before it may have received treatment in the main body of the text.

- C1DGate

  This class serves as a simple one dimensional gate. It is constructed with an upper and lower limit floating point value. Various methods are provided for setting and retrieving these pre-set limits. The main method is **bool passes(float value)** which checks whether the value passed in as a parameter lies inside or outside of the limits specified. The method returns a boolean value of true if the parameter value is inside the limits or false if it lies outside.

- CBuffer

  The **CBuffer** class is responsible for taking data items from the time ordered data stream and organising them into a time buffer. The

time buffer is constructed based upon user entered time durations i.e. a specified forward search time and backwards search time. The **CBuffer** class is discussed extensively in chapter 4.4.

- CCalibCoeff

The **CCalibCoeff** class is responsible for managing and applying the calibration coefficients for all the detector channels defined in the system. The calibrations are stored in a calibration file called 'calib.dat'. This file is a simple text file containing all the appropriate coefficients that is read in by the class when it is constructed. This class also contains the function that performs the calculation i.e. takes a channel ID and raw data and returns an energy value. This function can be expanded to perform different kinds of calibrations besides the standard linear calibration. For example various non linear polynomial calibrations could be implemented.

- CClock

The **CClock** class is used internally by the **CTDRDataItem** class to keep track of the synchronisation data items in the data stream. This class can also be used to ensure that the data stream is in time order.

- CError

This **CError** class provides simple logging functionality for the TDRSorter data analysis code. Any relevant warnings or errors are output to the same log file. The user may also make use of this class in the specialised sorting section to allow any logging necessary to be kept in one centralised log file.

- CEvent

The **CEvent** class is essentially the package of data items that are in prompt coincidence with each other. The central data structure used to build this class is the **CPixel**. All data items in prompt coincidence with the time-stamp of the pixel are placed into the event structure. The packaged **CEvent** is the primary parameter passed into the specialised sorting class. This class is discussed extensively in section 5.5.

- **CFloatHist**

  The **CFloatHist** class implements the **CHistogram** class and provides a histogram class that provides floating point precision counting i.e allows bins of the histogram to be incremented with fractional counts. The **CFloatHist** class is described in more detail in section 6.3.2.

- **CHistogram**

  This class is an abstract interface that defines the behaviour a histogram should exhibit. e.g. incrementation, output etc. The concrete histogram classes inherit from this to enable any type of histogram to be treated generically i.e. to allow a common histogram container. This class is discussed in more detail in section 6.3.2.

- **CIntHist**

  This class **CIntHist** implements the **CHistogram** class and provides a histogram class that provides integer precision counting. The **CIntHist** class is described in more detail in section 6.3.2.

- **CPixel**

  The **CPixel** class represents a DSSD X channel and a DSSD Y channel that are in prompt time coincidence with a triggering data item.

The two defining DSSD channels are also within specified time and energy constraints. The pixel encapsulates an event in the DSSD detector that has a defined position. The **CPixel** class is described in detail in section 5.2.

- **CRaw**

  The **CRaw** class is a collection class that contains a histogram for each individual detector channel in the spectrometer. The class increments each histogram with the data value of each data item before any further filtering is performed. The **CRaw** class histograms provide a record of the total counts in each channel of the detector system.

- **CRunData**

  The **CRunData** class is responsible for reading the raw binary data from the data source. The class takes this data and reads it into memory in a series of structures that can later be accessed in the construction of the time buffer. The **CRunData** class is covered in more detail in chapter 4.

- **CSettings**

  The **CSettings** class is a utility class that is used to set various parameters in the data analysis code. This class reads in data from a file called 'sorter.cfg' that contains various parameters that the user can alter to customize the triggering and event construction processes to fit requirements. Some of these settings are discussed in appendix A.

- **CSorter**

  The **CSorter** class is responsible for managing all the other classes in the data analysis system and also controlling the overall flow of the

data analysis system. The various components of the class are discussed throughout the thesis.

- CTagger

  The **CTagger** class is responsible for the sequencing of events that originate from a single pixel. This class along with the tagger manager forms a crucial part in identifying delayed coincidences between events such as recoil and alpha decays.

- CTaggerManager

  The **CTaggerManager** class is used to manage the two dimensional array of **CTagger** objects that map to the corresponding physical pixels of the DSSD detector. This class contains several methods for adding, retrieving and clearing data in the associated taggers. This class is discussed in section 6.4.3.

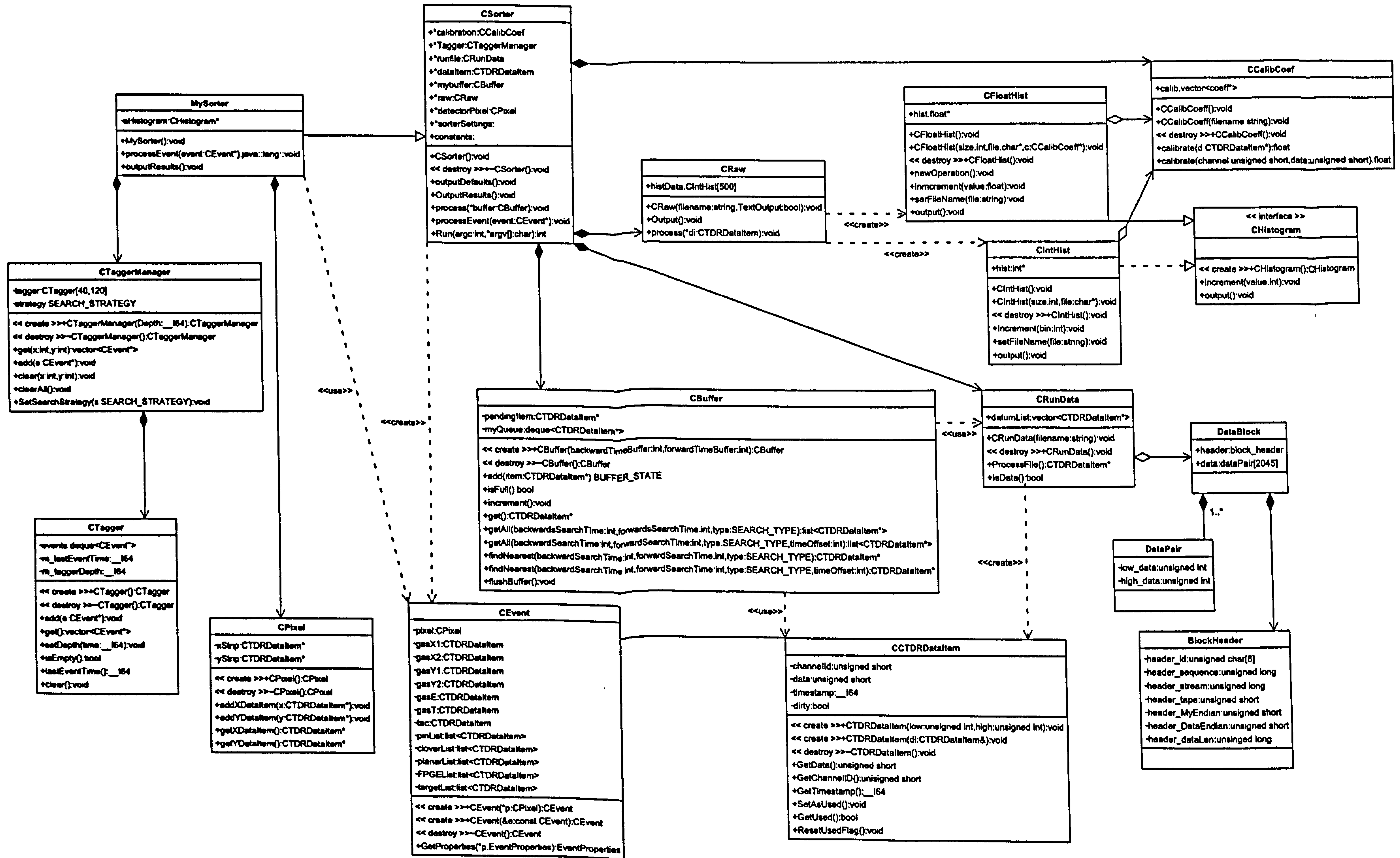
- CTDRDataItem

  The **CTDRDataItem** class is the fundamental unit of data that is extracted from the time ordered data stream. This class is used within most other classes within the data analysis code. The types of data encapsulated within this class are discussed in detail in chapter 3.

The final major class that is used in the TDRSorter data analysis code is the custom, specialised sorting class that is derived by the user from the **CSorter** class. This class is named by the user and contains all the sorting code specific to the experiment being performed. The creation of the specialised sorting class is discussed in appendix A and the methods used within it are discussed in detail in chapter 6.

Appendix C

Detailed TDRSorter UML Diagram



CSorter
+*calibration:CCalibCoef
+*Tagger:CTaggerManager
+*runfile:CRunData
+*dataItem:CTDRDataItem
+*mybuffer:CBuffer
+*raw:CRaw
+*detectorPixel:CPixel
+*sorterSettings:
+constants:
+CSorter():void
<< destroy >>+~CSorter():void
+outputDefaults():void
+OutputResults():void
+process(*buffer:CBuffer):void
+processEvent(event:CEvent*):void
+Run(argc:int,*argv[]:char):int
MySorter
-aHistogram:CHistogram*
+MySorter():void
+processEvent(event:CEvent*).java::lang::void
+outputResults():void
CCalibCoef
+calib.vector<coeff*>
+CCalibCoeff():void
+CCalibCoeff(filename string):void
<< destroy >>+CCalibCoeff():void
+calibrate(d CTDRDataItem*):float
+calibrate(channel unsigned short,data:unsigned short).float
CFloatHist
+hist.float*
+CFloatHist():void
+CFloatHist(size.int,file.char*,c:CCalibCoeff*):void
<< destroy >>+CFloatHist():void
+newOperation():void
+inmcrement(value:float):void
+serFileName(file:string):void
+output():void
CRaw
+histData.CIntHist[500]
+CRaw(filename:string,TextOutput:bool):void
+Output():void
+process(*di:CTDRDataItem):void
<<create>>
<<create>>
<< interface >>
CHistogram
<< create >>+CHistogram():CHistogram
+increment(value.int):void
+output():void
CIntHist
+hist:int*
+CIntHist():void
+CIntHist(size.int,file:char*):void
<< destroy >>+CIntHist():void
+Increment(bin:int):void
+setFileName(file:string):void
+output():void
CTaggerManager
-tagger:CTagger[40,120]
-strategy SEARCH_STRATEGY
<< create >>+CTaggerManager(Depth:__I64):CTaggerManager
<< destroy >>~CTaggerManager():CTaggerManager
+get(x:int,y:int):vector<CEvent*>
+add(e CEvent*):void
+clear(x:int,y:int):void
+clearAll():void
+SetSearchStrategy(s SEARCH_STRATEGY):void
<<use>>
<<create>>
CBuffer
-pendingItem:CTDRDataItem*
-myQueue:deque<CTDRDataItem*>
<< create >>+CBuffer(backwardTimeBuffer:int,forwardTimeBuffer:int):CBuffer
<< destroy >>~CBuffer():CBuffer
+add(item:CTDRDataItem*) BUFFER_STATE
+isFull() bool
+increment():void
+get():CTDRDataItem*
+getAll(backwardsSearchTime:int,forwardsSearchTime.int,type:SEARCH_TYPE):list<CTDRDataItem*>
+getAll(backwardSearchTime:int,forwardSearchTime.int,type.SEARCH_TYPE,timeOffset:int):list<CTDRDataItem*>
+findNearest(backwardSearchTime:int,forwardSearchTime.int,type:SEARCH_TYPE):CTDRDataItem*
+findNearest(backwardSearchTime int,forwardSearchTime:int,type:SEARCH_TYPE,timeOffset:int):CTDRDataItem*
+flushBuffer():void
<<use>>
CRunData
+datumList:vector<CTDRDataItem*>
+CRunData(filename:string):void
<< destroy >>+CRunData():void
+ProcessFile():CTDRDataItem*
+IsData():bool
DataBlock
+header:block_header
+data:dataPair[2045]
1..*
<<create>>
DataPair
-low_data:unsigned int
-high_data:unsigned int
CTagger
-events deque<CEvent*>
-m_lastEventTime:__I64
-m_taggerDepth:__I64
<< create >>+CTagger():CTagger
<< destroy >>~CTagger():CTagger
+add(e:CEvent*):void
+get():vector<CEvent*>
+setDepth(time:__I64):void
+isEmpty().bool
+lastEventTime():__I64
+clear():void
CPixel
-xStrip:CTDRDataItem*
-yStrip:CTDRDataItem*
<< create >>+CPixel():CPixel
<< destroy >>~CPixel():CPixel
+addXDataItem(x:CTDRDataItem*):void
+addYDataItem(y:CTDRDataItem*):void
+getXDataItem():CTDRDataItem*
+getYDataItem():CTDRDataItem*
CEvent
-pixel:CPixel
-gasX1:CTDRDataItem
-gasX2:CTDRDataItem
-gasY1:CTDRDataItem
-gasY2:CTDRDataItem
-gasE:CTDRDataItem
-gasT:CTDRDataItem
-tac:CTDRDataItem
-pinList:list<CTDRDataItem>
-cloverList:list<CTDRDataItem>
-planarList:list<CTDRDataItem>
-FPGEList:list<CTDRDataItem>
-targetList:list<CTDRDataItem>
<< create >>+CEvent(*p:CPixel):CEvent
<< create >>+CEvent(&e:const CEvent):CEvent
<< destroy >>~CEvent():CEvent
+GetProperties(*p:EventProperties):EventProperties
<<use>>
CCTDRDataItem
-channelId:unsigned short
-data:unsigned short
-timestamp:__I64
-dirty:bool
<< create >>+CTDRDataItem(low:unsigned int,high:unsigned int):void
<< create >>+CTDRDataItem(di:CTDRDataItem&):void
<< destroy >>~CTDRDataItem():void
+GetData():unsigned short
+GetChannelID():unisigned short
+GetTimestamp():__I64
+SetAsUsed():void
+GetUsed():bool
+ResetUsedFlag():void
BlockHeader
-header_id:unsigned char[8]
-header_sequence:unsigned long
-header_stream:unsigned long
-header_tape:unsigned short
-header_MyEndian:unsigned short
-header_DataEndian:unsigned short
-header_dataLen:unsinged long

Appendix D

TDRSorter Code CD