IDEX Algorithms Template

Draft 1

Ethan Ayari
Jamey Szalay
Mihály Horányi

Contents

Chapter 1

SDC Prompts

All of the information below was taken from the website: https://lasp.colorado.edu/galaxy/display/IMAP/IMAP+SDC+Software+Developer+Guide. In addition, **from the IMAP SOC CDRL List Applicable Algorithms** is defined in the following sequence:

Level 0 - Level 1A
Level 1A - Level 1B
Level 1B - Level 2
Data culling

1.1 List of Acronyms

1. SDC: Science Data Center
2. SDMP: Science Data Management Plan

1.2 List of Acronyms

- AID
- APID
- CCSDS
- CoDICE: Compact Dual Ion Composition Experiment
- FPGA: Field Programmable Gate Array
- FSW: Flight Software
- MG: Mid Gain
- HG: High Gain
- LG: Low Gain

- SWAPI: Solar Wind and Pickup Ions
- SDC: Science Data Center
- SOC: Science Operations Center
- CDRL
- DAB: Decontamination Actuator Board
- IDEX: Interstellar Dust Experiment
- IMAP: Interstellar Mapping and Acceleration Probe
- ISD: Interstellar Dust
- IDP: Interplanetary Dust Particle
- MSPS: Mega samples per second
- PUI: Pickup ions
- SDMP: Science Data Management Plan
- TOF: Time of Flight

1.3 Definitions

Algorithm: In this text, algorithm refers to a finite sequence of well-defined, computer-implementable instructions needed to generate one or more data products using lower-level data, calibration, and ancillary information.

Code: Computer-executable instructions in a common programming language may be compiled (if needed) and executed to perform designated computations.

Pseudocode: Simplified code used in preliminary code design and layout.

1.4 Preparation Information

Each instrument team shall provide information and data needed to produce applicable science data products identified in the IMAP Science Data Management Plan (SDMP). The intended purpose of this information is to enable members of the IMAP Science Data Center (SDC) to implement, test, and validate executable software that will routinely produce designated data products during the science mission. For IMAP, Science Algorithm Documentation consists of the combination of documentation, instructions, specifications, and any available reference/prototype computer code needed to produce a science data product.

At a minimum, each instrument team shall provide one or more descriptively detailed documents that includes procedural descriptions, flow diagrams, programmable equations, and associated pseudocode. Teams are, however, strongly encouraged to also provide either heritage code used for similar instruments on other missions or prototype/candidate code developed for IMAP (e.g. for instrument

testing or validation). Any code provided should ideally be directly relevant to the corresponding IMAP instrument with extraneous code or unrelated elements omitted (e.g. code from past mission(s) that is not relevant and could create confusion). Provided algorithm descriptions and/or code must be definitively known to work as expected in order to meet mission measurement requirements.

Documentation shall be complete and detailed, including applicable equations and diagrams, such that a third party can generate executable code that correctly implements the algorithms with a reasonable amount of collaborative assistance. This documentation shall include narrative descriptions of the overall sequence of operations, each necessary step, and shall include equations, figures and other supporting information where appropriate, in order to clearly convey the intent and context of the processing process. The documentation shall be written so as to mirror or sufficiently capture the logical structure of the resulting computer source code. The document should include structural information, as needed, such as data flow diagrams, data interface definitions, and pseudocode for suggested or actual routines (e.g. if legacy or prototype code is provided). The documentation shall provide a complete description of all required datasets, such as initial gain and bias terms, detector maps, spacecraft parameters, data sets from other instruments, or other calibration parameters. Provided documentation for legacy instrumentation must directly and unambiguously apply to the specific design, calibration, and operation of the IMAP instrument.

1.5 Suggested Algorithm Document Content

1. Definitions of Terms
2. Description of Overall Approach
3. Background on Heritage Instruments (if applicable)
4. Background on heritage processing implementations (if applicable)
5. Instrument measurement approach and relationship between data collection, telemetry packets, and data products
6. Overall processing flow, e.g. step-by-step process description/diagrams
7. Identification of and detail pertaining to dependencies on S/C or other instrument's data
8. Calibration Data and associated application, including recommended calibration file formats
9. Mathematical description of algorithm steps
10. Equations, constants, and coefficients for science calibrations
11. Pseudocode
12. Examples of in-flight calibration update incorporation
13. Telemetry packet decomposition information (possibly part of SE-005)
14. Spice kernels to map instrument into spacecraft coordinate frame
15. Equations and coefficients for Temperature Conversions (possibly part of SE-005)
16. Equations and coefficients for engineering conversions (possibly part of SE-005)

17. Observation time basis
18. Expected maintenance tasks (e.g. updating calibrations)
19. Approach to Rejection, Culling, or Flagging of Suspect/Bad Data
20. Expected Uncertainties / Uncertainty Analysis
21. Data Validation Steps
22. Recommended Testing Approaches
23. Special considerations and unusual circumstances

Instrument teams are asked to provide actual code in addition to descriptive algorithms, wherever possible, which is intended as a means to optimize overall mission costs and minimize redundant effort. Provided code may be in any programming language used by the institution that generates it and may utilize any conventions or standard practices used by that institution. However, if teams will be developing code that can be provided to the SDC as a basis, they are requested to follow as many of the conventions used at the SDC as possible, as outlined in the SDC Developer Guide document referenced above. The SDC team intends to work in a highly collaborative fashion with each instrument team to construct a streamlined, efficient, and open science data system.

Chapter 2

Introduction

2.1 Document Purpose

This document provides a description of the algorithms used to produce science data products from the IMAP Interstellar Dust Experiment (IDEX) instrument measurements. The data products are documented in the IMAP Science Data Management Plan (SDMP). This document includes a description of the data collection and data products, as well as details of the processing flow, algorithms, and data processing tasks.

2.2 Scope

The intended purpose of this information is to enable members of the IMAP Science Data Center (SDC) to implement, test, and validate executable software that will routinely produce designated data products during the science mission. For IMAP, Science Algorithm Documentation consists of the combination of documentation, instructions, specifications, and any available reference/prototype computer code needed to produce a science data product.

2.3 Document Organization

The first two chapters of the document provide a brief overview of what IDEX is and how it takes measurements. Chapter 3 Introduces the IDEX instrument characteristics and expected performance. Chapter 4 describes the details of the modes of operation: Booting Mode, Idle Mode, Science Mode, Transmit Mode, Safe Mode, Survival Mode and a Decontamination Mode. Chapter 5 describes the necessary steps to convert data between each of the levels along the pipeline. Detailed information on packets and software is made available in the Appendix.

Chapter 3

IDEX Instrument Description

3.1 Overview

The Interstellar Dust Experiment (IDEX) is a high-resolution dust analyzer that provides the elemental composition and mass distributions of ISD and IDP particles. IDEX links the interstellar gas phase composition, as obtained with IMAP-Lo and through PUIs with CoDICE and SWAPI, with the makeup of ISD and IDP particles.

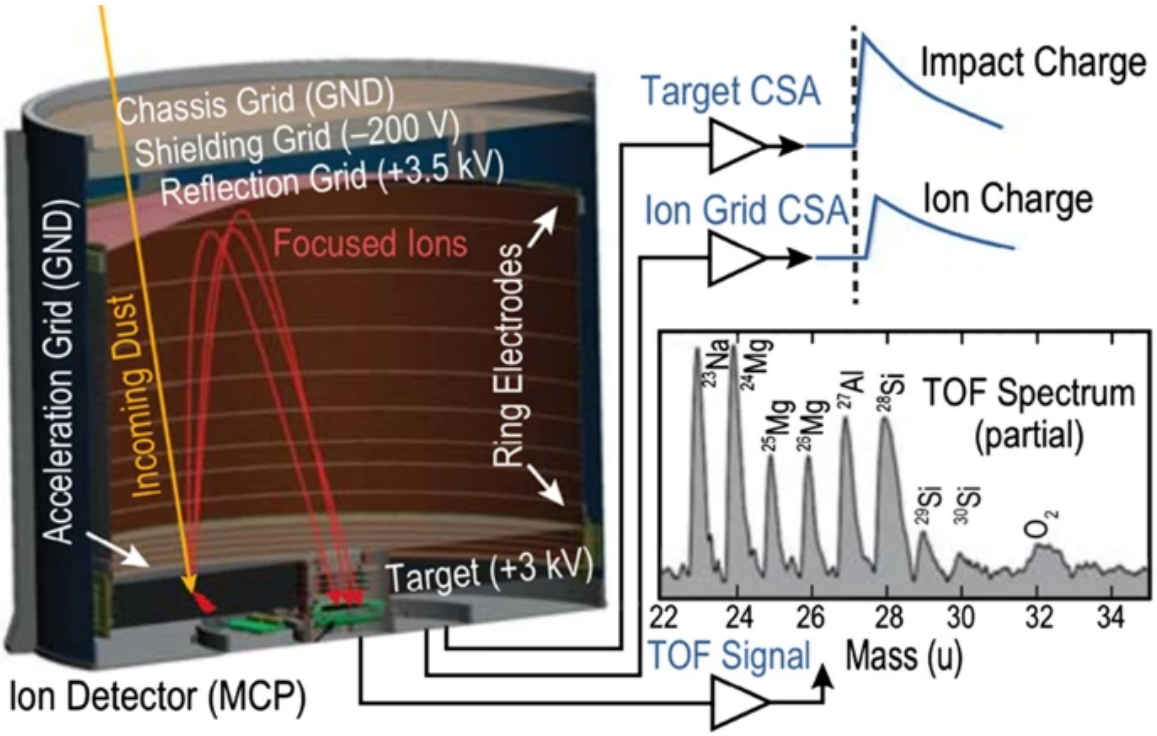

Figure 3.1: An outline of the IDEX instrument and operation.

IDEX is based on the heritage of past flight and laboratory instruments. Its operation principle is based on the impact ionization process.

IDEX has two components, an electronics enclosure and a sensor head with a large effective target area (600 cm^2). The IDEX mechanical design consists of a simple Al shell structure that provides support for the biased electrodes of the TOF ion mass analyzer and the grids over its aperture. The centrally-located ion detector and the front-end charge sensitive amplifier are integrated into the bottom of the instrument.

Individual dust particles entering the instrument pass through a set of grid electrodes and impact the target (3.1). The target is biased at +3 kV to provide positive ion acceleration and the reflectron-type ion optics are used to generate TOF compositional mass spectra. The impact-generated negative charges are collected on the target. An electrostatic field, shaped by a set of biased rings and a curved grid electrode, provides spatial and temporal focusing of the accelerated ions onto the central detector.

Target cleanliness is of high importance to be maintained throughout the fabrication, integration and operation of IDEX. The primary concern is the condensation of volatiles on the target. The mitigate this risk, IDEX has a one-time deployable door and its in-flight operation allows for the regular (once per months) use of heaters to raise the target temperature to 120 C and release any possible condensible contaminants.

3.2 Sensor Head

An overview of the sensor head is shown in Figure 3.2.

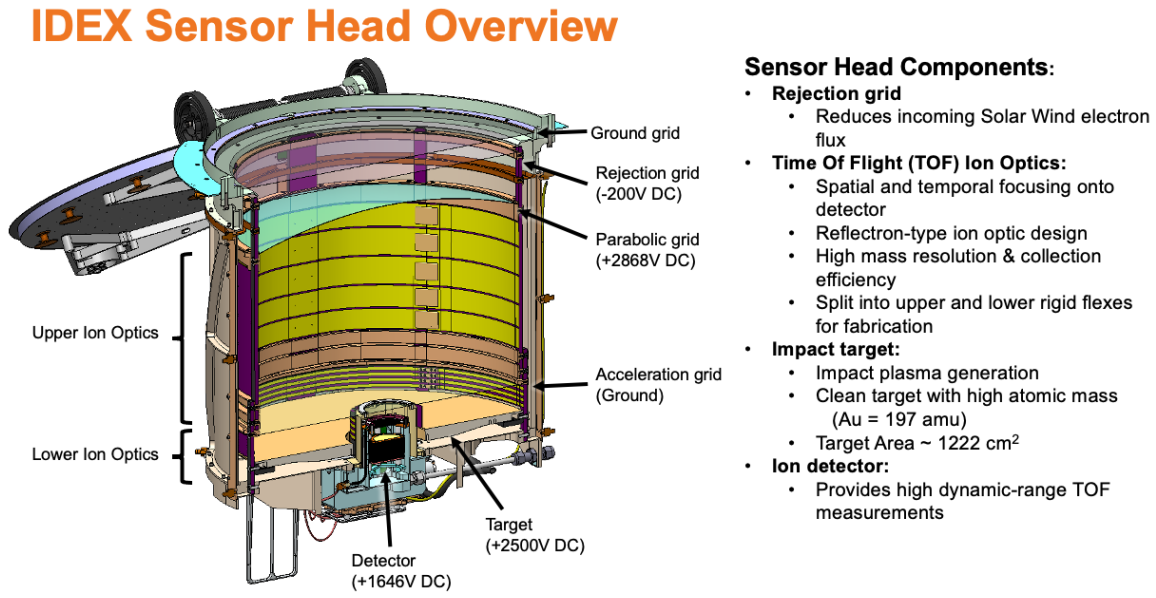

Figure 3.2: An outline of the IDEX instrument sensor head.

3.3 Electronics

The IDEX electronics leverage SUDA flight designs. The Decon/Actuator Board (DAB) is a new design that was added to control the door and decontamination heater. The amplifier boards (CSA, Detector Buffer) were re-designed from the SUDA heritage instrument to meet the IDEX requirements for charge detection ranges. An overview of the IDEX electronics is depicted in Figure 3.3.

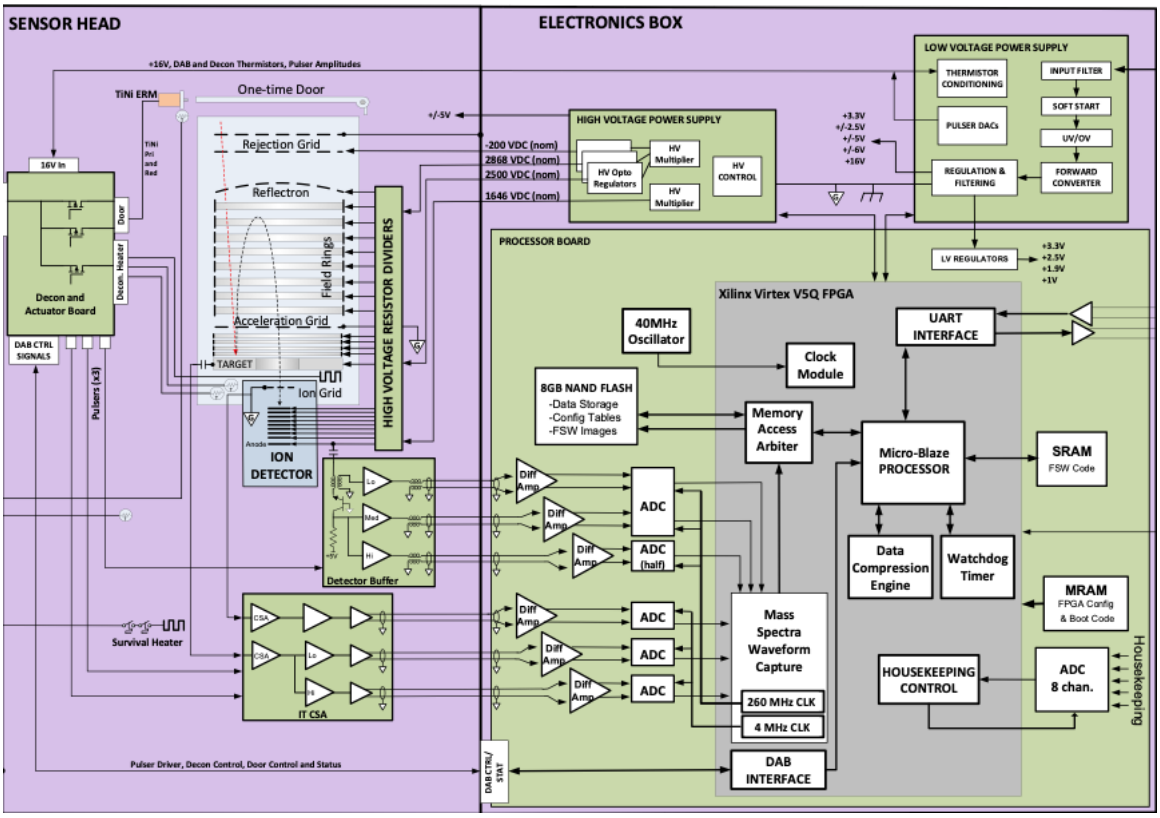

Figure 3.3: An outline of the IDEX instrument electronics system.

The 3 Time-of-Flight (TOF) gain stages, 2 Target CSA and 1 Ion grid CSA waveforms are the signals which the science data products are derived from. These constitute the packetized data accumulated in a fro8 GB NAND flash and form the IDEX level 0 data product.

Chapter 4

Nominal Operation

4.1 Operating Modes

IDEX has seven operating modes: off; boot; and five nominal operations. A flow diagram showing how the instrument transitions through the various modes and a brief description of each is depicted in 4.1.

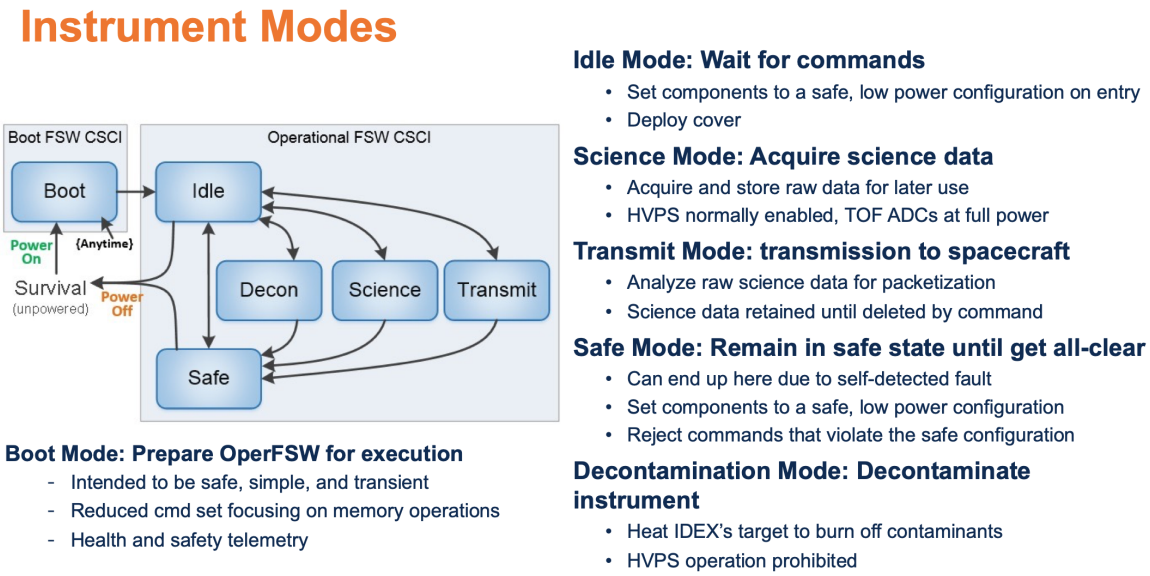

Figure 4.1: An outline of the IDEX instrument operation modes.

The **boot** mode is enabled when the instrument is powered on or any other time the operator indicates. In this mode, while FSW is performing a set of memory operations to prepare for execution, a health and safety telemetry packet is transmitted.

Following boot, the instrument enters its **idle** mode, and the instrument awaits commands. From here, the instrument may enter one of four possible operational modes, of which only two are relevant to the science data processing pipeline.

4.1.1 Science Mode

When the instrument enters the **science** mode, a memory buffer is continuously filled containing each of the 6 instrument signals. Since IDEX is an event-driven instrument, data is only packetized and transmitted when we have a dust impact, which we will refer to as an **event**. The instrument triggers based off of an event, sending the data stored in the memory buffer to the NAND flash.

The amount of data stored in a memory buffer depends on the sampling rate of the corresponding **analog-digital converter (ADC)**. The three time-of-flight gain stages are sampled at 130 MHz (high sampling rate). The two target gain stages and the ion grid, the ADCs are sampled at 4 MHz. Figure 4.2 illustrates how data is packed for these two types of ADCs, as well as how much data is transmitted per event.

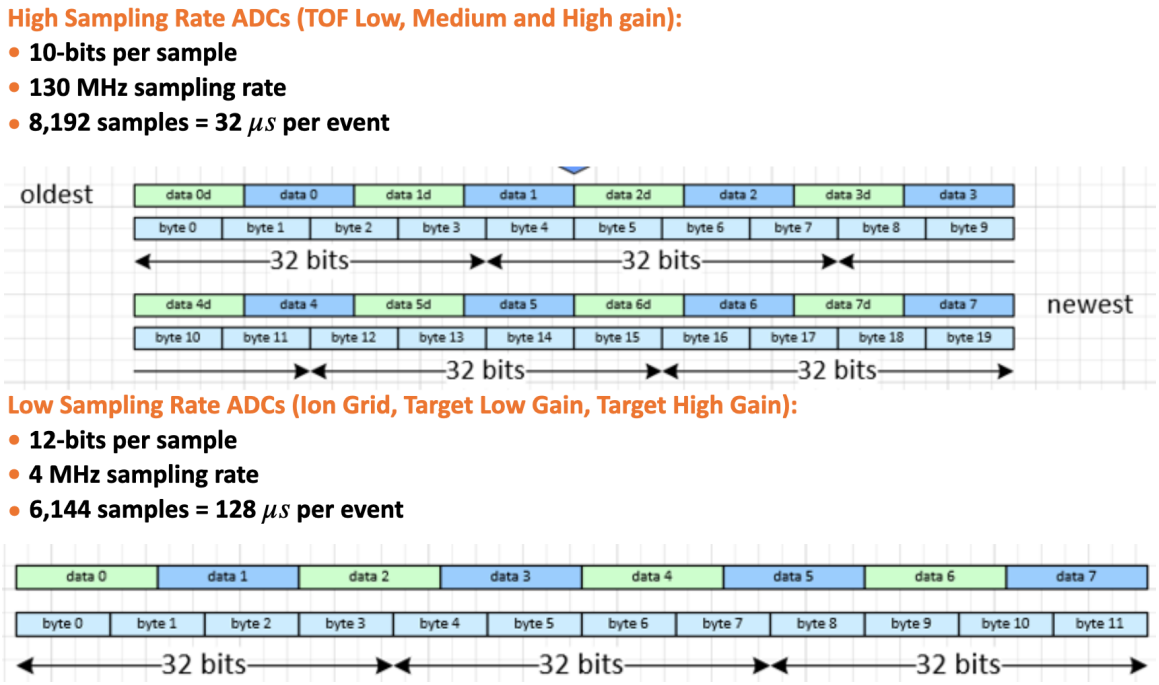

Figure 4.2: Diagram illustrating how data from the two different ADCs is packed in the NAND flash.

4.1.2 Transmit Mode

Once the NAND flash has been populated with data from all 6 signals, it is ready for transmission. There are additional post-trigger blocks based on the sampling rate of the corresponding ADCs, as pictured in Figure 4.3.

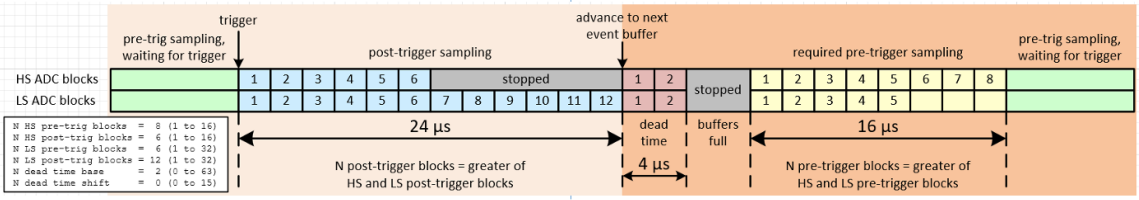

Figure 4.3: Block diagram of the instrument timeline, based on high (HS) and low (LS) sampling rate ADCs.

Once the instrument triggers, there is a 4 μs dead time after the required post-trigger sampling has been completed. Hence, at the very least, there will be a 4 μs gap between events.

Transitions between instrument modes are driven by commands from the spacecraft. For the purposes of error detection and correction (EDAC), the 30 most recently executed commands and their associated timestamps are stored in the instrument log. This instrument log can be telemetered to the ground with the **DumpLog** command. These 30 commands include successful and failing command executions. However, if the command is so badly mangled that the FPGA or FSW is unable to recognize it as a command, then it does not get included in the log.

4.1.3 Spacecraft Time Interface

Each event is organized chronologically by the FPGA trigger time. For each event, the trigger time is stored in the event header. The event header is always specified by **SCI0TYPE=1** (see 4.3.1 for more details). This is given by a linear combination of the **SHFINE** and **SHCOARSE** variables. For each FPGA header, the **SHCOARSE** variable is a 16-bit unsigned integer that delineates the number of seconds since epoch (Midnight, January 1st, 2012)**SHFINE** variable indicates a number of 20 microsecond intervals in addition to the seconds since epoch. Together, these time measurements establish when an event took place with respect to epoch.

Additionally, the time axis for each of IDEX's 6 signals is derived from the sampling rate and the number of samples. The Ion Grid and two Target waveforms sample at 4 MHz, and the TOF sample at 130 MHz. The number of samples can then be multiplied by 4 $[ns]$ and 250 $[ns]$ for the time axis.

4.2 Nominal Science Operations

IDEX is an event-driven instrument, so data collection only takes place in the event of a trigger. Otherwise, when in science mode the instrument will continuously fill and delete the memory buffer.

4.3 Data Products

The IDEX science data products pipeline takes raw instrument CCSDS packets (level 0 data) as inputs and goes through each stage of the mass spectrum calibration process to reveal the composition, impact charge, and ion-grid charge of each measured dust particle (level 3 products). Each science data product level is delineated below in Table 4.1.

PRODUCT	PRODUCT STANDARD	DESCRIPTION
LEVEL 0	Packet	All downlinked data, errors corrected, duplicates removed, time sequence and completeness verified
LEVEL 1A	Raw	Unpacked CSA waveforms (2 target and 1 ion grid) & 3 TOF mass spectra in DN (low, medium and high gain stages), IDEX settings in instrument units (DN), time-tagged event data
LEVEL 1B	Raw	Target, ion grid & TOF mass spectra in V (low, medium, high gain), IDEX settings in physical units (V), time-tagged event data
LEVEL 1C	Reduced	Impact charge from target, ion charge from ion grid, reconstructed TOF waveform (voltage vs time)
LEVEL 2	Calibrated	TOF spectrum converted from time to mass scale of with initially identified major mass lines, dust mass calculated
LEVEL 3	Derived	Composition of individual ISD & IDP particles (relative abundances of most common elements identified in the mass spectra (C, O, Si, Fe, Mg...)
CALIBRATION	Ancillary	Library of laboratory mass spectra of candidate minerals, SPICE kernels for spacecraft state vector and attitude

Table 4.1: An outline of the IDEX instrument science data product levels, their format, and a brief description.

4.3.1 Science Data

IDEX generates three APIDs containing science data:

- **Transmit (APID 0x590):** Waveforms and metadata for science events, in response to the **Transmit** command.
- **Fetch (APID 0x591):** Waveforms and metadata for science events, in response to the **Fetch** and **FetchOne** commands.

The contents of each APID are further described below. All IDEX packets are 32-bit aligned, starting with a CCSDS header.

In general, science data is organized onboard by the **accountability identifier (AID)**. The AID is assigned when the data is acquired, and is used to access the data afterwards. The AID cannot be changed, and duplicate AIDs are not permitted. Each dataset is composed of one or more **"events"**, which is a collection of data collected when the instrument triggers, usually representing a dust particle impact. Each event is given a unique "event number" by the FPGA, so that every event collected by IDEX can be uniquely identified by the combination of AID and event number.

Event data is composed of seven distinct parts:

1. Time of Flight (TOF) waveform data on the high-gain (HG) channel, sampled at 250 megasamples per second (Msps) with 10-bit resolution.
2. TOF waveform data on the mid-gain (MG) channel, sampled at 250 Msps with 10 bit resolution.
3. TOF waveform data on the low-gain (LG) channel, sampled at 250 Msps with 10 bit resolution.
4. Ion (Qi) grid Charge Sensitive Amplifier (CSA) waveform data, sampled at 3.75 Msps with 12 bit resolution.

5. Target Low (Qt) CSA waveform data, sampled at 3.75 Msps with 12 bit resolution.
6. Target High (Qt) CSA waveform data, sampled at 3.75 Msps with 12 bit resolution.
7. FPGA Pre-pended metadata header containing IDEX's configuration and status at the time of the trigger, recorded by the FPGA.

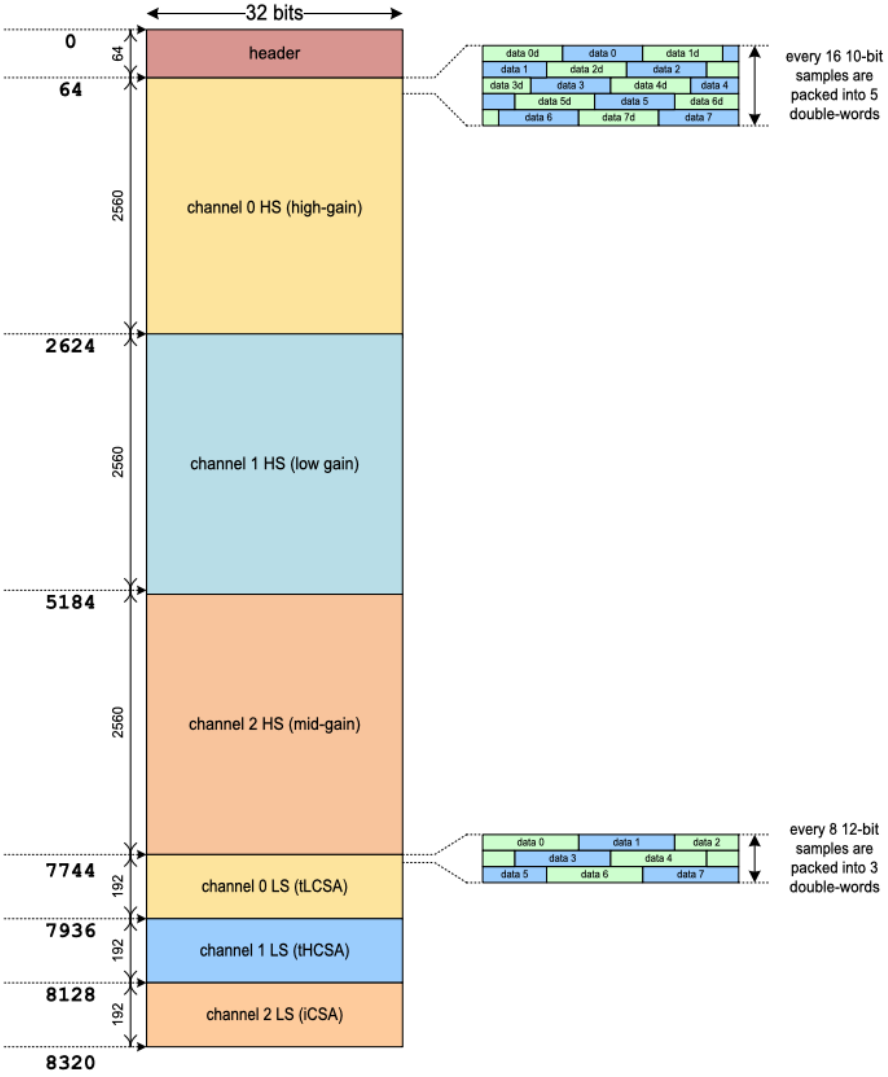

Figure 4.4: A bitmap outline of the science data constituting each event

.

At the end of science data acquisition, a catalog representing the new dataset is stored along with the raw science data in nonvolatile memory. The catalog is mostly empty at this point but can be downlinked with the **SendCatalog** command. If no science data is acquired during acquisition, then no catalog is created either. A high-level diagram of how the high and low sampling rate ADC

data is transmitted once packetized is shown in Figure 4.4, describing how different blocks of data are stored. Following science data acquisition, FSW must process the data before it can be sent to the ground. Processing is started with the **Process** command, which has the FSW evaluating waveform data to determine characteristics (e.g. peak count) of each event. The characteristics are then used to assign each individual event to a category. As each event in a dataset is processed, a new entry is added to the catalog (in chronological order of acquisition). The FSW includes a setting where event data (all seven parts) are transmitted (APID 0x590) as soon as it is processed – which was used for some early ground testing but never afterward.
A dataset only needs to be processed once but can be processed multiple times (with different results if parameter table entriesare changed). However only the most recent processing results are saved to the catalog.
Once a dataset is processed, it can be downlinked as any of the three packet types. Normally the catalog is sent right away since it is considered "feed-forward data" used to configure instrument parameters for subsequent data acquisition opportunities. "Transmitted" science data comes next, and are used as a "first pass" at evaluating the science data and downlinking a subset of acquired events. Afterward, scientists will evaluate the catalog and request specific events to be downlinked as "fetched" science data. Another option is that the dataset can be "transmitted" again but with different parameter table entries resulting in a different selection of events transmitted.

Transmitted and Fetched Science Packets

Transmitted and fetched science data share the same decommutation, the only real difference being the APID. When transmitting or fetching science data, one event is handled at a time. Each event may spawn between 1 and 10 packets, depending on size and requested contents.
As mentioned above, event data is composed of seven distinct parts but not all are sent for every event. An argument of the **FetchOne** command specifies which parts to send, and similarly the Fetch Table includes this info for "Fetch" commands (see **AddToFetch** command for more details). Parameter table entries are used to determine what parts of an event are sent for transmitted data (using the event's category which themselves are grouped into quality factors). Each part of the event is packaged into its own packet, while still sharing the same APID. To fit within the IICD-required maximum CCSDS packet length of 4095 bytes, one packet will contain no more than 4032 bytes of science data (the rest dedicated to headers and such). Each part normally fits in one packet, but the TOF waveforms can span two-three packets if the dataset is large and compression is poor (or disabled). In this case, the maximum amount of science data is sent in the first packet and any leftover sent in the next packet. The content of packets containing the metadata header are very different from the waveform data, so it is discussed separately.

Packets Containing Waveforms

Science packets containing any of the six waveforms, whether transmitted or fetched, start with a brief header and are followed by a variable amount of science data. The header is never compressed, and provides identifying information such as AID, event number, and which waveform.

Packets Containing Waveforms

Item Name	Type	Description
VERSION	U3	CCSDS version number, fixed 0b000
TYPE	U1	CSDS type, fixed 0b0 for telemetry
SEC_HDR_FLG	U1	CCSDS secondary header flag, fixed 0b1 for presence of secondary header
PKT_APID	U11	**CCSDS Application Process Identifier (APID), either 0x590 or 0x591**
SEQ_FLGS	U2	CCSDS grouping flags, fixed 0xb11 for standalone packet
SRC_SEQ_CTR	U14	CCSDS source sequence counter, increments per packet (separate counters per APID)
PKT_LEN	U16	CCSDS packet length, excludes UMS and CCSDS primary header (minus one)
SHCOARSE	U32	Time of packet generation (not data acquisition), as integer seconds since epoch
SHFINE	U16	Time of packet generation, each DN represents 20usec within current second
SCI0AID	U32	Accountability Identifier for this event
SCI0TYPE	**U8**	**Packet data content type:** **1: FPGA pre-pended metadata header** **2: TOF HG.** **4: TOF LG.** **8: TOF MG.** **10: Target LG (Q_T) CSA** **20: Target HG (Q_T) CSA** **40: Ion grid (Q_I) CSA**
SCI0CONT	U8	Channels being downlinked (same decomposition as above)
SCI0SPARE1	U13	Spare
SCI0PACK	U1	Always 0b1, showing data is bit-packed
SCI0FRAG	U1	If 0b1, then this channel data spans multiple packets and this is not the last packet (use SCI0FRAGOFF to reconstruct)
SCI0COMP	U1	Data is compressed if 0b1, otherwise not
SCI0EVTNUM	U16	Event number for this event
SCI0CAT	U8	Category assigned to this event (for most recent processing operation)
SCI0QUAL	U8	Quality factor assigned to this category
SCI0FRAGOFF	U16	Starting offset for this data when reconstructing data that spans multiple packets, in 32-bit “dwords” not bytes. Either 0x000 or multiple of 0x3F0
SCI0VER	U16	Science CSC Version number for this header
SCI0TIMECOARSE	U16	Lowest 16 bits of integer seconds since epoch for time of this event
SCI0TIMEFINE	U16	Sub-second time of first event, each DN represents 20usec within current second
SCI0SPARE2	U32	Spare
SCI0SPARE3	U32	Spare
SCI0SPARE4	U32	Spare
SCI0RAW	**Varies**	**Padded with zeroes for 32-bit alignment, if needed**
SYNCSCI0PKT	U16	Synchronization marker, fixed 0x3333
CRCSCI0PKT	U16	CRC CCITT-16

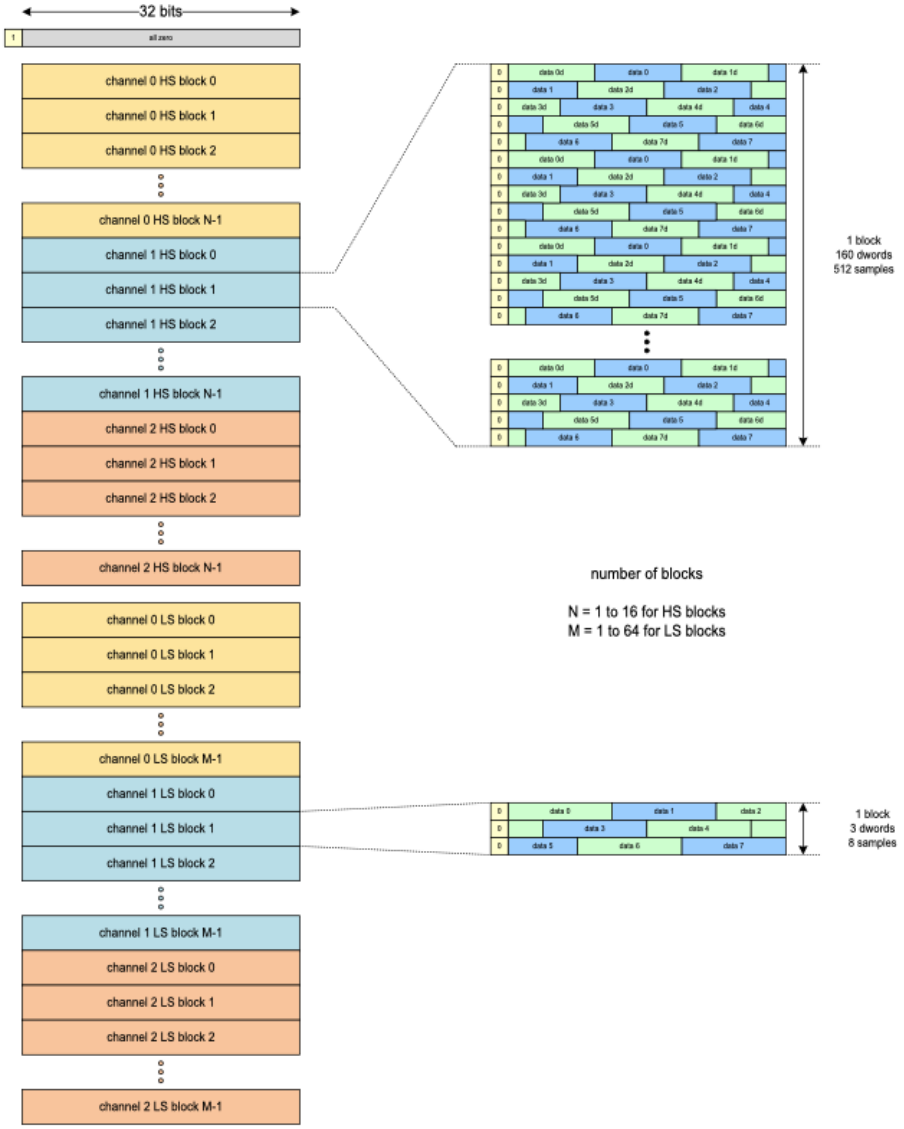

Figure 4.5: A bitmap outline of what is contained in each set of waveform data.

The FSW makes use of the FPGA's RICE compression core, with details discussed in the FPGA specification (167130). **If the resulting compressed data is not 32-bit aligned, then the remainder (up to 31 bits) are padded with zeroes**. To decompress a TOF waveform, a RICE decompressor should be used and the output interpreted as a chronological series of 10-bit samples. For SCA waveforms, the output is 12-bit samples.
After the data is decompressed on the ground, the resulting data represents 6 time-ordered waveforms corresponding to each AID.

The packet is summarized in the table below, using a format similar to the SUDA Command and Telemetry handbook:

- Item name: Mnemonic for the named item
- Bytes: Position of the item within the element, show bytes with packet and bits within byte
- Type of data: Everything is "U" for unsigned data number, followed by bit length
- Description: Briefly describes the item, giving the conversion if appropriate.

The **SCI0RAW** field contains the instrument waveforms in each packet. How the header, low sampling and high sampling rate ADC data appears in each packet is described in figure 4.5. While each channel appears in 3 packets for most events, figure 4.5 describes the general case for N high sampling and M low sampling blocks.

Packets Containing FPGA Metadata Headers

The packet containing the metadata header holds status and configuration information at the time when this event was acquired. The first 32 bytes after the CCSDS header are identical to the preceding waveform data, so that the SCI0TYPE field can be used as a key to differentiate packets containing this metadata header versus waveforms.

After the common header is the **FPGA prepended metadata header** that the FPGA captures for every event. These data are fixed size and never compressed. Many of these fields are the same for all events within the same dataset. A diagram describing the components of the FPGA header is shown in Figure 4.6.

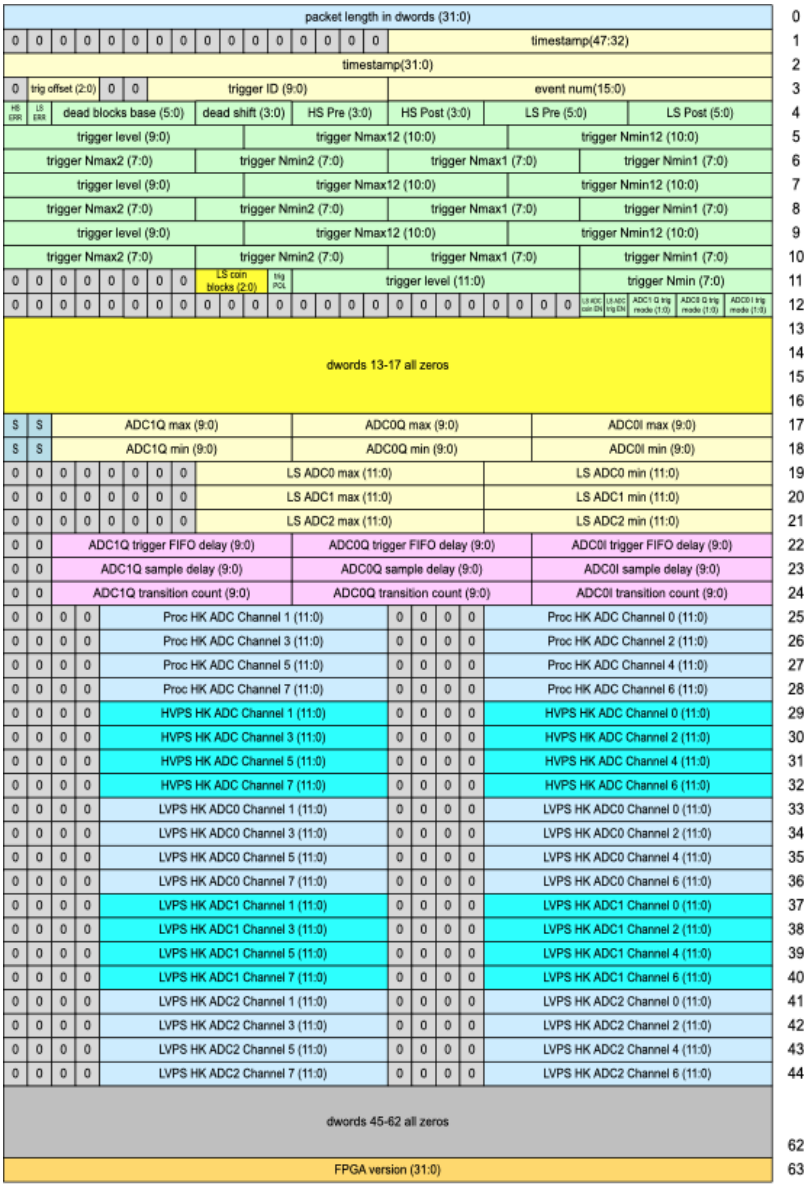

Figure 4.6: A bitmap outline of what is contained in each FPGA metadata event header.

The fields in the FPGA header pertinent to the level 0 to 1A science data processing pipeline include the **timestamp**, **event num**, and various **trigger** fields. A full outline of all the available fields including their length, dtype, and description is shown below.

FPGA Header

Item Name	Type	Description
CCSDS header	12 bytes	See section 2.1 for details
SCI0AID	**U32**	See above (same as waveform packets)
SCI0TYPE	**U8**	
SCI0CONT	U8	
SCI0SPARE1	U13	
SCI0PACK	U1	
SCI0FRAG	U1	
SCI0COMP	U1	
SCI0EVTNUM	U16	
SCI0CAT	U8	
SCI0QUAL	U8	
SCI0FRAGOFF	U16	
SCI0VER	U16	
SCI0TIMECOARSE	U16	
SCI0TIMEFINE	U16	
SCI0SPARE2	U32	
SCI0SPARE3	U32	
SCI0SPARE4	U32	
SCIDATALENGTH	U32	Length of raw science data in memory including this FPGA pre-pended metadata header but excludes the sync words at the start of all Flash pages.
TIMESTAMP1	**U16**	**Padded with 0's**
	U16	**Time of event trigger, in integer seconds (bits 31:16)**
TIMESTAMP2	**U16**	**Time of event trigger, in integer seconds (bits 15:0)**
	U16	**Time of event trigger, in sub-seconds (20usec per DN)** **event time, sub-seconds**
EVENTNUMBER	U1	Padded with 0
	U3	Trigger offset: number of WCLK cycles that the start of post-trigger data collection is delayed from the trigger position to align the HS and LS sampling periods. See FPGA's ADC Trigger Offset" register.
	U2	Padded with 0's
	U10	Identifies origin of this trigger (multiple concurrent triggers are possible) upper 4 bits are spares (unused) bit 5 – external trigger bit 4 – SW trigger bit 3 – LS ADC1 trigger (target CSA) bit 2 – HS ADC1Q trigger (mid-gain channel) bit 1 – HS ADC0Q trigger (low-gain channel) bit 0 – HS ADC0I trigger (high-gain channel) See FPGA's "ADC Trigger ID" register.
	U16	Event number assigned by FPGA. Resets to 0 at power on and incremented thereafter (rolling over from 0xFFFF to 0).
NBLOCKS, see FPGA's "ADC N Blocks" register	U6	LS Post Blocks - programmed number of post-trigger blocks for low-rate ADC data (1-64)
	U6	LS Pre Blocks - programmed number of post-trigger blocks for low-rate ADC data (1-64)
	U4	HS Post Blocks - programmed number of post-trigger blocks for low-rate ADC data (1-64)
	U4	HS Pre Blocks - programmed number of pre-trigger blocks for high-rate ADC data (1-16)
	U4	Dead Blocks Shift - programmed number of bit positions to shift Dead Blocks Base value to the left to get the total number of dead blocks (0-15)
	U6	Dead Blocks Base - programmed base number of dead blocks (0-63)
	U1	LS Blocks Error - this bit is set if total number of LS blocks (pre + post) is greater than 64
	U1	HS Blocks Error - this bit is set if total number of HS blocks (pre + post) is greater than 16

HSADC0IHDR1, see FPGA's "HS ADC0 Channel I Trigger Control 2" register	U10	ADC0I trigger level - trigger level for ADC0I channel
	U11	ADC0I trigger Nmax12 - max # of samples between pulse 1 and 2 for ADC0I double pulse triggering
	U11	ADC0I trigger Nmin12 - min # of samples between pulse 1 and 2 for ADC0I double pulse triggering
HSADC0IHDR2, see FPGA's "HS ADC0 Channel I Trigger Control 1" register	U8	ADC0I trigger Nmin1 - min # of samples for pulse 1 for ADC0I single and double pulse triggering
	U8	ADC0I trigger Nmax1 - max # of samples for pulse 1 for ADC0I single and double pulse triggering
	U8	ADC0I trigger Nmin2 - min # of samples for pulse 2 for ADC0I double pulse mode triggering
	U8	ADC0I trigger Nmax2 - max # of samples for pulse 2 for ADC0I double pulse mode triggering
HSADC0QHDR1, see FPGA's "HS ADC0 Channel Q Trigger Control 2" register	U10	ADC0Q trigger level - trigger level for ADC0Q channel
	U11	ADC0Q trigger Nmax12 - max # of samples between pulse 1 and 2 for ADC0Q double pulse triggering
	U11	ADC0Q trigger Nmin12 - min # of samples between pulse 1 and 2 for ADC0Q double pulse triggering
HSADC0QHDR2, see FPGA's "HS ADC0 Channel Q Trigger Control 1" register	U8	ADC0Q trigger Nmin1 - min # of samples for pulse 1 for ADC0Q single and double pulse triggering
	U8	ADC0Q trigger Nmax1 - max # of samples for pulse 1 for ADC0Q single and double pulse triggering
	U8	ADC0Q trigger Nmin2 - min # of samples for pulse 2 for ADC0Q double pulse mode triggering
	U8	ADC0Q trigger Nmax2 - max # of samples for pulse 2 for ADC0Q double pulse mode triggering
HSADC1QHDR1, see FPGA's "HS ADC1 Channel Q Trigger Control 2" register	U10	ADC1Q trigger level - trigger level for ADC1Q channel
	U11	ADC1Q trigger Nmax12 - max # of samples between pulse 1 and 2 for ADC1Q double pulse triggering
	U11	ADC1Q trigger Nmin12 - min # of samples between pulse 1 and 2 for ADC1Q double pulse triggering
HSADC1QHDR2, see FPGA's "HS ADC1 Channel Q Trigger Control 1" register	U8	ADC1Q trigger Nmin1 - min # of samples for pulse 1 for ADC1Q single and double pulse triggering
	U8	ADC1Q trigger Nmax1 - max # of samples for pulse 1 for ADC1Q single and double pulse triggering
	U8	ADC1Q trigger Nmin2 - min # of samples for pulse 2 for ADC1Q double pulse mode triggering
	U8	ADC1Q trigger Nmax2 - max # of samples for pulse 2 for ADC1Q double pulse mode triggering
LSADC, see FPGA's LS ADC1 Trigger Control" register	U8	Padded with 0's
	U3	coincidence mode blocks - number of blocks coincidence window is enabled after LS ADC1 trigger
	U1	LS ADC1 trigger polarity - trigger polarity for LS ADC1 (0 = normal, 1 = inverted)
	U12	LS ADC1 trigger level - trigger level for LS ADC1
	U8	LS ADC1 trigger Nmin - min number of samples above/below trigger level for triggering LS ADC1
TRIGGERMODE, see FPGA's "Trigger Mode" Register	U1	HVPS polarity status, "cation" mode (0) or "anion" mode (1)
	U22	Padded with 0's
	U1	External trigger polarity
	U1	LS ADC coincidence mode enable
	U1	LS ADC trigger enable
	U2	ADC1Q trigger mode – same as above
	U2	ADC0Q trigger mode – same as above
	U2	ADC0I trigger mode 00 – no triggering (default) 01 – threshold mode 10 – single pulse mode 11 – double pulse mode
SPARE0	U32	Padded with 0's
SPARE1	U32	Padded with 0's

SPARE2	U32	Padded with 0's
SPARE3	U32	Padded with 0's
TOFMAX	U2	Padded with 0's
	U10	ADC1Q max value - max sample value on ADC1Q channel since previous event was captured. See FPGA's "ADC1 DQ Channel MinMax" Register.
	U10	ADC0Q max value - max sample value on ADC0Q channel since previous event was captured. See FPGA's "ADC0 DQ Channel MinMax" Register.
	U10	ADC0I max value - max sample value on ADC0I channel since previous event was captured. . See FPGA's "ADC0 DI Channel MinMax" Register.
TOFMIN	U2	Padded with 0's
	U10	ADC1Q min value - min sample value on ADC1Q channel since previous event was captured. See FPGA's "ADC1 DQ Channel MinMax" Register.
	U10	ADC0Q min value - min sample value on ADC0Q channel since previous event was captured. See FPGA's "ADC0 DQ Channel MinMax" Register.
	U10	ADC0I min value - min sample value on ADC0I channel since previous event was captured. See FPGA's "ADC0 DI Channel MinMax" Register.
LSADC0MINMAX, see FPGA's "LS ADC0 Channel MinMax" Register	U8	Padded with 0's
	U12	LS ADC0 max value - max sample value on LS ADC0 channel since previous event was captured
	U12	LS ADC0 min value - min sample value on LS ADC0 channel since previous event was captured
LSADC1MINMAX, see FPGA's "LS ADC1 Channel MinMax" Register	U8	Padded with 0's
	U12	LS ADC1 max value - max sample value on LS ADC1 channel since previous event was captured
	U12	LS ADC1 min value - min sample value on LS ADC1 channel since previous event was captured
LSADC2MINMAX, see FPGA's "LS ADC2 Channel MinMax" Register	U8	Padded with 0's
	U12	LS ADC2 max value - max sample value on LS ADC2 channel since previous event was captured
	U12	LS ADC2 min value - min sample value on LS ADC2 channel since previous event was captured
TOFTRIGGERDELAY	U2	Padded with 0's
	U10	ADC1Q trigger delay - delay in DCLK of trigger condition through comparison FIFO for ADC1Q
	U10	ADC0Q trigger delay - delay in DCLK of trigger condition through comparison FIFO for ADC0Q
	U10	ADC0I trigger delay - delay in DCLK of trigger condition through comparison FIFO for ADC0I
TOFSAMPLEDELAY	U2	Padded with 0's
	U10	ADC1Q sample delay - delay in samples from actual trigger position in event buffer to expected trigger position for ADC1Q
	U10	ADC0Q sample delay - delay in samples from actual trigger position in event buffer to expected trigger position for ADC0Q
	U10	ADC0I sample delay - delay in samples from actual trigger position in event buffer to expected trigger position for ADC0I
TOFTRANSCOUNT, see FPGA's "Transitions Counts" Register	U2	Padded with 0's
	U10	ADC1Q transition count - Number of transitions across the programmed threshold level for ADC1Q
	U10	ADC0Q transition count - Number of transitions across the programmed threshold level for ADC0Q
	U10	ADC0I transition count - Number of transitions across the programmed threshold level for ADC0I
PROCHKADCCHAN01	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "1V POL Current"
	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "1.9V POL Current"
PROCHKADCCHAN23	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "ProcBd Temp1"
	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "ProcBd Temp2"

PROCHKADCCHAN45	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "1V Voltage"
	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "FPGA Temp"
PROCHKADCCHAN67	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "1.9V Voltage"
	U4	Padded with 0's
	U12	Last measurement in raw DN for Processor Board signal "3.3V Voltage"
HVPSHKADCCHAN01	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Detector Voltage"
	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Sensor Voltage"
HVPSHKADCCHAN23	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Target Voltage"
	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Reflectron Voltage"
HVPSHKADCCHAN45	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Rejection Voltage"
	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Detector Current"
HVPSHKADCCHAN67	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Sensor IP"
	U4	Padded with 0's
	U12	Last measurement in raw DN for HVPS Board signal "Sensor IN"
LVPSHKADC0CHAN01	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P3.3VREF_HK"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P3.3VREF_OP"
LVPSHKADC0CHAN23	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "N6V"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P6V"
LVPSHKADC0CHAN45	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P16V"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P3.3V"
LVPSHKADC0CHAN67	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "N5V"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P5V"
LVPSHKADC1CHAN01	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P3.3_IMON"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P16V_IMON"
LVPSHKADC1CHAN23	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P6V_IMON"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "N6V_IMON"
LVPSHKADC1CHAN45	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P5V_IMON"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "N5V_IMON"
LVPSHKADC1CHAN67	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "P2.5V_IMON"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "N2.5V_IMON"
LVPSHKADC2CHAN01	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "THERM_CSA0"
	U4	Padded with 0's
	U12	Last measurement in raw DN for LVPS Board signal "THERM_CSA1"

LVPSHKADC2CHAN23	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “THERM_BUFF”
	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “THERM_LVPS”
LVPSHKADC2CHAN45	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “THERM_HVPS”
	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “P2.5V”
LVPSHKADC2CHAN67	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “N2.5V”
	U4	Padded with 0’s
	U12	Last measurement in raw DN for LVPS Board signal “Spare”
SPARE	U32	Padded with 0’s
SPARE	U32	Padded with 0’s
FSWHDR00	U32	Section reserved for FSW usage
FSWHDR01	U32	Section reserved for FSW usage
FSWHDR02	U32	Section reserved for FSW usage
FSWHDR03	U32	Section reserved for FSW usage
FSWHDR04	U32	Section reserved for FSW usage
FSWHDR05	U32	Section reserved for FSW usage
FSWHDR06	U32	Section reserved for FSW usage
FSWHDR07	U32	Section reserved for FSW usage
FSWHDR08	U32	Section reserved for FSW usage
FSWHDR09	U32	Section reserved for FSW usage
FSWHDR10	U32	Section reserved for FSW usage
FSWHDR11	U32	Section reserved for FSW usage
FSWHDR12	U32	Section reserved for FSW usage
FSWHDR13	U32	Section reserved for FSW usage
FSWHDR14	U32	Section reserved for FSW usage
FSWHDR15	U32	Section reserved for FSW usage
FPGAVER	U32	FPGA version, see FPGA’s “DataTime Register”
SYNCSCI0PKT	U16	Synchronization marker, fixed 0x3333
CRCSCI0PKT	U16	CRC CCITT-16

4.3.2 Housekeeping Data

UPDATE IN FUTURE DRAFT

4.3.3 Engineering Mode Data

UPDATE IN FUTURE DRAFT

4.3.4 In-Flight Calibration Data

UPDATE IN FUTURE DRAFT

Chapter 5

Data Processing

This chapter describes the necessary steps to convert data between each of the levels described in the processing flowchart. Source code snippets are provided in Python.

5.1 Overview

The IDEX data processing pipeline takes the data packets described above, an xml XTCE definition/decommutation table (C), the instrument ground calibration curve, the spacecraft thruster data, and ephemerides data as inputs. Each data product and its corresponding inputs/outputs is described in Figure 5.1.

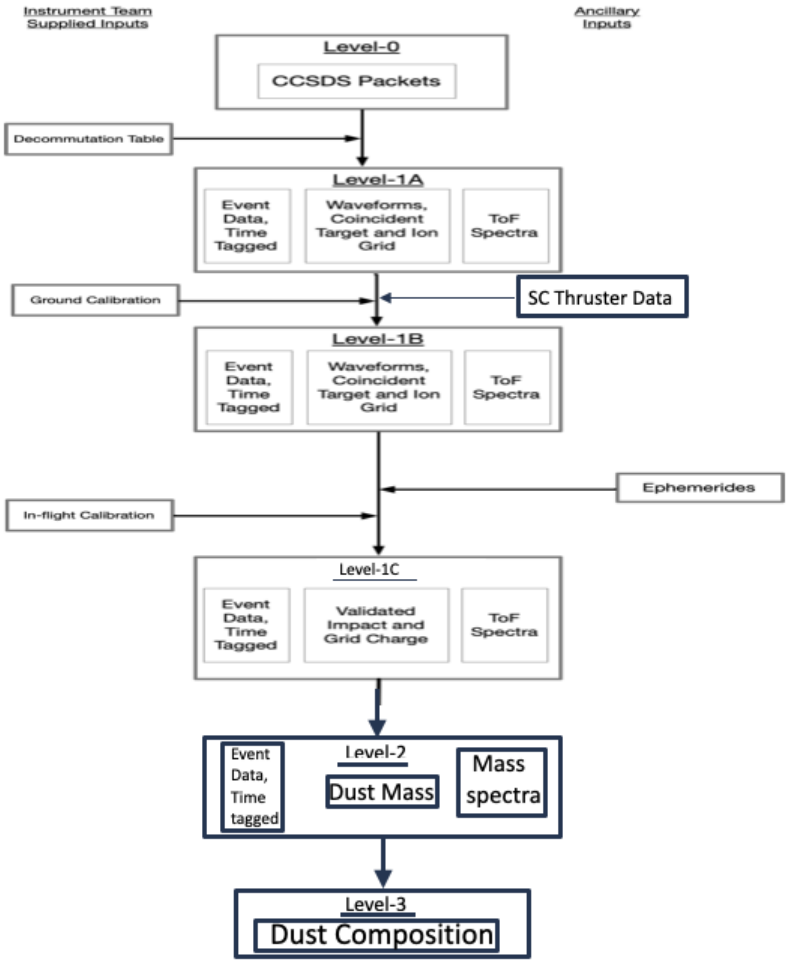

Figure 5.1: The IDEX science data pipeline processing flowchart. The boldface boxes indicate inputs from the March 2022 IMAP data products meeting at APL.

5.2 Data Volume Expectancy

The estimates for the expected IDEX instrument science data volume are derived from simulations of known IDP and ISD populations. Figure 5.2 depicts counts for ISD (red/blue) and IDP (black) particles expected to be measured by IDEX during the mission.

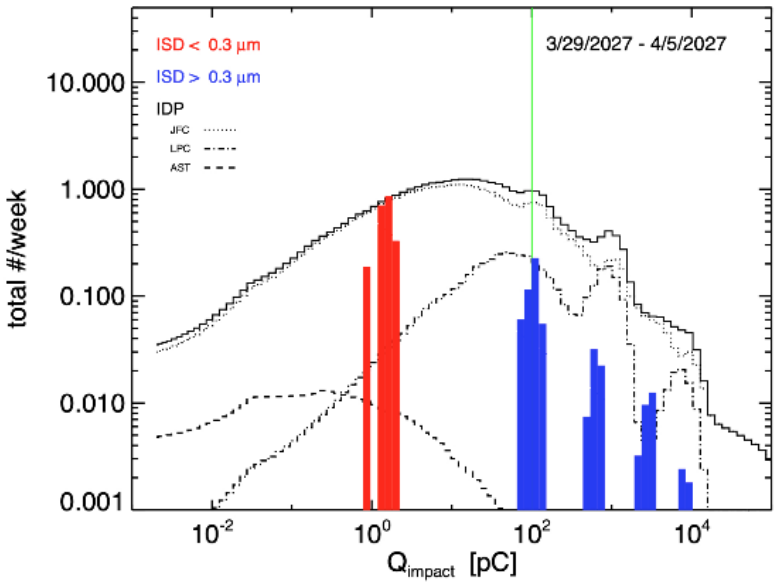

Figure 5.2: Expectations for IDEX's ISD and IDP measurements during the mission. Note that although this is just shown for 6 days in 2027, these simulations have been carried out for all of the prime and extended IMAP mission.

From this simulation, IDEX is expected to measure 16 events per day on average. Using these estimates in conjunction with the bitmap in Figure 4.6, we can estimate the weekly data volume expectancy for IDEX during the mission. This has been carried out for each of the science data product levels, as depicted in table 5.1.

PRODUCT	PRODUCT STANDARD	DESCRIPTION
LEVEL 0	Packet	288 Kilobits per event * 16 Events per day * 7 events per week = **40,852 Kilobits per week**
LEVEL 1A	Raw	40 voltages, currents and temperatures tracked in DN (40*12 = 1,280 bits), 264,192 bits for all 6 signals, so 265,472 per event * 7 * 16 = **29,733 Kilobits per week**
LEVEL 1B	Raw	Conversion to Voltage does not introduce new volume, **29,733 Kilobits per week**
LEVEL 1C	Reduced	2 double words for each charge + 81,920 bits for TOF spectra = 81,984 bits per event * 7 * 16 = **9,183 Kilobits per week**
LEVEL 2	Calibrated	1 Voltage array + 1 double word for mass + up to 12 double words for mass index = 82,336 bits per event * 7 * 16 = **9,222 Kilobits per week**
LEVEL 3	Derived	Up to 12 mass lines and their relative abundances = 24 double words = 768 bits per event * 7 * 16 = **87 Kilobits per week**
CALIBRATION	Ancillary	Library of laboratory mass spectra of candidate minerals, SPICE kernels for spacecraft state vector and attitude

Table 5.1: Expectations for IDEX's weekly data volume throughout the mission, organized by data product level.

5.3 Heritage Data Processing

IDEX has a high-level of heritage from the SUrface Dust Analyzer (SUDA) onboard the upcoming flagship *Europa Clipper* mission. SUDA is nearly identical but has instead been optimized for flybys and detections of organic molecules for a limited impact velocity range of around 4 km/s. The XTCE definition is the same for both instruments (see appendix C), but the interpretation of the waveforms differs, along with voltage/temperature conversions.

Like IDEX, SUDA Event data is composed of seven distinct parts:

1. Time of Flight (TOF) waveform data on the high-gain (HG) channel, sampled at 250 megasamples per second (Msps) with 10 bit resolution
2. TOF waveform data on the mid-gain (MG) channel, sampled at 250 Msps with 10 bit resolution
3. TOF waveform data on the low-gain (LG) channel, sampled at 250 Msps with 10 bit resolution
4. Velocity (Qv) grid charge sensitive amplifier (CSA) waveform data, sampled at 3.75 Msps with 12 bit resolution
5. Ion (Qi) grid CSA waveform data, sampled at 3.75 Msps with 12 bit resolution
6. Target (Qt) CSA waveform data, sampled at 3.75 Msps with 12 bit resolution
7. FPGA Pre-pended metadata header containing IDEX's configuration and status at the time of the trigger, recorded by the FPGA

The velocity grid CSA takes the place of IDEX's second target gain stage. Otherwise, the names, sampling rates, and bit resolutions are all identical. SUDA leverages the use of the SWIFT language-based SPECTRUM software, developed in-house at LASP. To avoid this Mac OS dependency, the IDEX pipeline has been constructed exclusively in Python 3.8.

5.4 Level 0 to Level 1a Processing

5.4.1 Decommutation

Packet decommutation relies on the **LASP packets** library, available on the LASP Bitbucket (LASP VPN access required). The **LASP packets** software hinges on the **generator** method of the **Parsing.PacketParser** module for reading CCSDS packets (please see appendix A for Python source code).

The packet data is read in using Python's **bitstring** library to reduce memory usage. Each packet is associated with a "flattened sequence container" from the XTCE definition. To do this, first, the length of the packet is taken from the CCSDS header:

```
specified_packet_data_length =
↪  self._total_packet_bits_from_pkt_len(header['PKT_LEN'].raw_value)
```

Next, the type of packet as provided by the CCSDS header is cross-referenced with the XTCE definition to find a matching APID, VERSION, and TYPE. If there is a match, the code parses the packet according to the XTCE definition. If there is no match, it will attempt to generally parse it using the length of the packet. The source for this is shown below.

```
    try:
                    if isinstance(self.packet_definition,
                    ↪  xtcedef.XtcePacketDefinition):
                        packet = self.parse_packet(binary_data,

                                                   ↪  self.packet_definition.named_containers,

                                                   ↪  root_container_name=root_container_name,
                                                  word_size=self.word_size)
                    else:
                        packet_def_name, parameter_list =
                        ↪  self._determine_packet_by_restrictions(header)
                        packet = self.legacy_parse_packet(binary_data,
                        ↪  parameter_list, word_size=self.word_size)
```

This process is repeated until we have parsed each of the packets in the dataset.

5.4.2 Science Data

Once the data has been depacketized, the **SCI0TYPE** field designates what data is contained in each packet. Per the table in 4.3.1, the possible values of **SCI0TYPE** include:

1 = FPGA pre-pended metadata header
2 = TOF HG
4 = TOF LG
8 = TOF MG
10 = Target LG (Q_T) CSA
20 = Target HG (Q_T) CSA
40 = Ion grid (Q_I) CSA

By reading in the **SCI0TYPE** field, the appropriate ADC sampling rate (and therefore bit pattern) is determined.

The waveform data must be read in a pattern specific to high or low sampling rate ADCs in the schema outline below:

For **high sampling rate (TOF) data (10-bit samples)**, handle the raw data as 32-bit units like this (most significant bit first):

2 bits = Padding set to zeros, ignore this
10 bits = oldest sample
10 bits = next sample
10 bits = next (newest) sample

For the remaining **low sampling rate data (12-bit samples)**, handle the raw data as 32-bit units like this (most significant bit first):

8 bits = Padding set to zeros, ignore this
12 bits = oldest sample
12 bits = next (newest) sample

Once this schema has been used for the waveform data, all the science event data has been read in and stored. The example Python script below will read in an IDEX raw binary data file, plot each of the event waveforms, and write all of the data to an easily accessible HDF5 file. Plots are saved to a */Plots/* directory in the same directory as the script, and the HDF5 files are written to the */HDF5/* directory.

```
#!/opt/anaconda3/bin/python3
# -*- coding: utf-8 -*-

"""
A Python object to store IDEX packets.
__author__      = Ethan Ayari & Gavin Medley,
Institute for Modeling Plasmas, Atmospheres and Cosmic Dust

Works with Python 3.8.10
"""

# || Python libraries
import argparse
import os
import socket
import bitstring
import h5py
import shutil
import matplotlib.pyplot as plt
plt.style.use("seaborn-pastel")
import numpy as np

# || LASP software
from lasp_packets import xtcedef  # Gavin Medley's xtce UML implementation
```

```
from lasp_packets import parser  # Gavin Medley's constant bitstream
↪  implementation
import cdflib.cdfwrite as cdfwrite
import cdflib.cdfread as cdfread

# ||
# ||
# || Generator object from LASP packets
# || to read in the data
class IDEXEvent:
    def __init__(self, filename: str):
        """Test parsing a real XTCE document"""
        # TODO: Change location of xml definition
        idex_xtce = 'idex_combined_science_definition.xml'
        idex_definition = xtcedef.XtcePacketDefinition(xtce_document=idex_xtce)
        # assert isinstance(idex_definition, xtcedef.XtcePacketDefinition)


        idex_packet_file = filename
        print(f"Reading in data file {idex_packet_file}")
        idex_binary_data = bitstring.ConstBitStream(filename=idex_packet_file)
        print("Data import completed, writing packet structures.")

        idex_parser = parser.PacketParser(idex_definition)
        idex_packet_generator = idex_parser.generator(idex_binary_data,
                                                      # skip_header_bits=64,
                                                      skip_header_bits=32,  # For
                                                      ↪  sciData
                                                      show_progress=True,

                                                      ↪  yield_unrecognized_packet_errors=True)


        print("Packet structures written.")
        idex_binary_data.pos = 0
        idex_packet_generator = idex_parser.generator(idex_binary_data)
        self.data = {}
        self.header={}
        evtnum = 0
        for pkt in idex_packet_generator:
            print(evtnum)
            if 'IDX__SCIOTYPE' in pkt.data:
                # print(evtnum)
                if pkt.data['IDX__SCIOTYPE'].raw_value == 1:
                    evtnum += 1
                    # print(pkt.data)
                    print(f"^*****Event header {evtnum}******^")
```

```
                print(f"AID = {pkt.data['IDX__SCIOAID'].derived_value}")  #
                ↪  Instrument event number
                print(f"Event number =
                ↪  {pkt.data['IDX__SCIOEVTNUM'].raw_value}")  # Event number
                ↪  out of how many events constitute the file
                # print(f"Time = {pkt.data['IDX__SCIOTIME32'].derived_value}")
                ↪  # Time in 20 ns intervals
                print(f"Rice compression enabled =
                ↪  {bool(pkt.data['IDX__SCIOCOMP'].raw_value)}")   # TODO:
                ↪  Use this to determine if we should rice_decompress the
                ↪  data {0,1}
                # self.header[evtnum][f"TimeIntervals"] =
                ↪  pkt.data['IDX__SCIOTIME32'].derived_value  # Store the
                ↪  number of 20 ns intervals in the respective CDF "Time"
                ↪  variables
                self.header[(evtnum, 'Timestamp')] =
                ↪  pkt.data['SHCOARSE'].derived_value +
                ↪  20*(10**(-6))*pkt.data['SHFINE'].derived_value  # Use this
                ↪  as the CDF epoch
                print(f"Timestamp = {self.header[(evtnum, 'Timestamp')]}
                ↪  seconds since epoch (Midnight January 1st, 2012)")

            if pkt.data['IDX__SCIOTYPE'].raw_value in [2, 4, 8, 16, 32, 64]:
                # print(pkt.data['IDX__SCIOTYPE'].raw_value)
                # print(evtnum)
                if pkt.data['IDX__SCIOTYPE'].raw_value not in self.data:
                    # self.data[(evtnum, pkt.data['IDX__SCIOTYPE'].raw_value)]
                    ↪  = pkt.data['IDX__SCIORAW'].raw_value
                    self.data.update({(evtnum,
                    ↪  pkt.data['IDX__SCIOTYPE'].raw_value):
                    ↪  pkt.data['IDX__SCIORAW'].raw_value})
                else:
                    # self.data[(evtnum, pkt.data['IDX__SCIOTYPE'].raw_value)]
                    ↪  += pkt.data['IDX__SCIORAW'].raw_value
                    self.data[(evtnum, pkt.data['IDX__SCIOTYPE'].raw_value)]
                    ↪  += pkt.data['IDX__SCIORAW'].raw_value

        # Parse the waveforms according to the scitype present (high gain and low
        ↪  gain channels encode waveform data differently).
        i = 1
        for scitype, waveform in self.data.items():
            self.data[scitype] = parse_waveform_data(waveform, scitype[1])

        names = {2: "TOF H", 4: "TOF L", 8: "TOF M", 16: "Target L", 32: "Target
        ↪  H", 64: "Ion Grid"}
        datastore = {}
        for scitype, waveform in self.data.items():
```

```
            datastore[(scitype[0], names[scitype[1]])] = self.data[(scitype[0],
            ↪  scitype[1])]
        self.data = datastore
        self.numevents = evtnum
        # print(self.data.keys())

        # ||
# ||
# || Parse the high sampling rate data, this
# || should be 10-bit blocks
def parse_hs_waveform(waveform_raw: str):
    """Parse a binary string representing a high gain waveform"""
    w = bitstring.ConstBitStream(bin=waveform_raw)
    ints = []
    while w.pos < len(w):
        w.read('pad:2')  # skip 2
        ints += w.readlist(['uint:10']*3)
    print(len(ints))
    return ints[:-4]

# ||
# ||
# || Parse the low sampling rate data, this
# || should be 12-bit blocks
def parse_ls_waveform(waveform_raw: str):
    """Parse a binary string representing a low gain waveform"""
    w = bitstring.ConstBitStream(bin=waveform_raw)
    ints = []
    while w.pos < len(w):
        w.read('pad:8')  # skip 2
        ints += w.readlist(['uint:12']*2)
    return ints

# ||
# ||
# || Use the SciType flag to determine the sampling rate of
# || the data we are trying to parse
def parse_waveform_data(waveform: str, scitype: int):
    """Parse the binary string that represents a waveform"""
    print(f'Parsing waveform for scitype={scitype}')
    if scitype in (2, 4, 8):
        return parse_hs_waveform(waveform)
    else:
        return parse_ls_waveform(waveform)

# ||
# ||
# || Gather all of the events
```

```
# || and plot them
def plot_all_data(packets, fname: str):
    fname = os.path.split(fname)[-1]
    # Create a folder to store the plots
    PlotFolder = os.path.join(os.getcwd(), f"Plots/{fname}")
    if os.path.exists(PlotFolder):  # If it exists, remove it
        shutil.rmtree(PlotFolder)
    os.makedirs(PlotFolder)

    # print("Number of packet items = ", len(packets.items()))
    fig, ax = plt.subplots(nrows=6)  # Make this general
    fig.set_size_inches(18.5, 10.5)
    for i, (k, v) in enumerate(packets.items()):  # k[0] = Event number, k[1] =
    ↪  channel name, v=waveform data

        # fig = plt.figure(figsize=(17,12))
        # print(i%6)
        i=i%6  # We take modulo 6 so it is the same for each event
        x = np.linspace(0, len(v), len(v))  # Number of samples
        # Scale number of samples by 4 ns (high rate) or 250 ns (low rate) to get
        ↪  to time.
        if(i<=2):
            x *= .004  # LS
        else:
            x *= .25  # HS
        # Conver to microseconds
        x /= 1e3

        ax[i].plot(x, v)
        # ax[i].fill_between(x, v, color='r')
        ax[i].set_ylabel(k[1], font="Times New Roman", fontsize=15,
        ↪  fontweight='bold')
        ax[i].set_xlabel(r"Time [$\mu$ s]", font="Times New Roman", fontsize=30,
        ↪  fontweight='bold')
        plt.subplots_adjust(bottom=0.2)

        text = f'Min = {min(v)} [dN]'+ '\n'+ f'Avg = {int(np.mean(v))} [dN]'+ '\n'
        ↪  + f'Std = {int(np.std(v))} [dN]'+ '\n' + f'Max = {max(v)} [dN]'
        # ax[i].text(1.13, 0.8, text, fontsize=15, va="top", ha="right",
        ↪  transform=ax[i].transAxes, bbox=dict(facecolor='none',
        ↪  edgecolor='black', boxstyle='round,pad=.5'))
        ax[i].text(1.125, 0.85, text, fontsize=15, va="top", ha="right",
        ↪  transform=ax[i].transAxes)
        plt.suptitle(f"{fname} Event {k[0]}", font="Times New Roman", fontsize=30,
        ↪  fontweight='bold')
        # plt.tight_layout()
        plt.savefig(os.path.join(PlotFolder, f"{fname}_Event_{k[0]}.png"),
        ↪  dpi=100)
```

```
        if i==5:  #  End of the event, lets free up some memory
            plt.show()
            plt.close()
            del fig, ax
            fig, ax = plt.subplots(nrows=6)  # Make this general
            fig.set_size_inches(18.5, 10.5)


# ||
# ||
# || Write the waveform data
# || to an HDF5 file
def write_to_hdf5(waveforms: dict, filename: str):
    os.chdir('./HDF5/')
    filename = os.path.split(filename)[-1]  # Just get the name of the file
    # Prepend HDF5 folder to filename

    # print(waveforms.keys())
    # print(waveforms.values())

    # TODO: Change this to a suitable file location for the .h5 dump
    if os.path.exists(filename):
        os.remove(filename)
    h = h5py.File(filename,'w')
    for k, v in waveforms.items():
        # print(np.array(v))
        # h.create_dataset(k, data=np.array(v, dtype=np.int8))
        h.create_dataset(f"/{k[0]}/{k[1]}", data=np.array(v))
        if(k[1]=='TOF L'):
            time = np.linspace(0, len(v), len(v))
            h.create_dataset(f"/{k[0]}/Time (high sampling)", data=time)
        if(k[1]=='Ion Grid'):
            time = np.linspace(0, len(v), len(v))
            h.create_dataset(f"/{k[0]}/Time (low sampling)", data=time)
    os.chdir('../')
    # h.create_dataset("Time since ")

# ||
# ||
# || Write the waveform data
# || to CDF files
def write_to_cdf(packets):

    cdf_master = cdfread.CDF('imap_idex_l0-raw_0000000_v01.cdf')
    if (cdf_master.file != None):
    # Get the cdf's specification
        info=cdf_master.cdf_info()
        cdf_file=cdfwrite.CDF('./IDEX_SSIM.cdf',cdf_spec=info,delete=True)
```

```
    # if (cdf_file.file == None):
    #     cdf_master.close()
    #     raise OSError('Problem writing file.... Stop')

    # Get the global attributes
    globalaAttrs=cdf_master.globalattsget(expand=True)
    # Write the global attributes
    cdf_file.write_globalattrs(globalaAttrs)
    zvars=info['zVariables']
    print('no of zvars=',len(zvars))
    # Loop thru all the zVariables --> What are zvars vs rvars?
    for x in range (0, len(zvars)):
        # Get the variable's specification
        varinfo=cdf_master.varinq(zvars[x])
        print('Z ============>',x,': ', varinfo['Variable'])


# Z ============> 0 :  Epoch
# Z ============> 1 :  IDEX_Trigger
# Z ============> 2 :  TOF_Low
# Z ============> 3 :  TOF_Mid
# Z ============> 4 :  TOF_High
# Z ============> 5 :  Time_Low_SR
# Z ============> 6 :  Time_High_SR
# Z ============> 7 :  Target_Low
# Z ============> 8 :  Target_High
# Z ============> 9 :  Ion_Grid

        if(varinfo['Variable']=="Epoch"):
            vardata = None
        if(varinfo['Variable']=="IDEX_Trigger"):
            vardata = packets.header[(1,"Timestamp")]
        if(varinfo['Variable']=="TOF_Low"):
            print(len(np.array(packets.data[(1,"TOF L")])))
            vardata = np.array(packets.data[(1,"TOF L")], float)
        if(varinfo['Variable']=="TOF_Mid"):
            vardata = np.array(packets.data[(1,"TOF M")])
        if(varinfo['Variable']=="TOF_High"):
            vardata = np.array(packets.data[(1,"TOF H")])
        if(varinfo['Variable']=="Time_Low_SR"):
            vardata = np.linspace(0, len(packets.data[(1,"Target L")]),
            ↪  len(len(packets.data[(1,"Target L")])))
        if(varinfo['Variable']=="Time_High_SR"):
            vardata = np.linspace(0, len(packets.data[(1,"TOF L")]),
            ↪  len(len(packets.data[(1,"Target L")])))
        if(varinfo['Variable']=="Target_Low"):
            vardata = np.array(packets.data[(1,"Target L")])
        if(varinfo['Variable']=="Target_High"):
```

```
            vardata = np.array(packets.data[(1,"Target H")])
        if(varinfo['Variable']=="Ion_Grid"):
            vardata = np.array(packets.data[(1,"Ion Grid")])
        # Get the variable's attributes
        varattrs=cdf_master.varattsget(zvars[x], expand=True)
        if (varinfo['Sparse'].lower() == 'no_sparse'):
            # A variable with no sparse records... get the variable data
            # vardata= None
            # Create the zVariable, write out the attributes and data
            cdf_file.write_var(varinfo, var_attrs=varattrs, var_data=vardata)
        else:
            # A variable with sparse records...
            # data is in this form [physical_record_numbers, data_values]
            # physical_record_numbers (0-based) contains the real record
            # numbers. For example, a variable has only 3 physical records
            # at [0, 5, 10]:
            varrecs=[0,5,10]

            # vardata=None  # np.asarray([.,.,.,..])
            # Create the zVariable, and optionally write out the attributes
            # and data
            cdf_file.write_var(varinfo, var_attrs=varattrs,
                        var_data=[varrecs,vardata])
    rvars=info['rVariables']
    print('no of rvars=',len(rvars))
    # Loop thru all the rVariables
    for x in range (0, len(rvars)):
        varinfo=cdf_master.varinq(rvars[x])
        print('R ============>',x,': ', varinfo['Variable'])
        varattrs=cdf_master.varattsget(rvars[x], expand=True)
        if (varinfo['Sparse'].lower() == 'no_sparse'):
            vardata=None
            # Create the rVariable, write out the attributes and data
            cdf_file.write_var(varinfo, var_attrs=varattrs, var_data=vardata)
        else:
            varrecs= None  # [.,.,.,..]
            vardata= None  # np.asarray([.,.,.,..])
            cdf_file.write_var(varinfo, var_attrs=varattrs,
                        var_data=[varrecs,vardata])
    cdf_master.close()
    cdf_file.close()

# || Test code: Import file and write the relevant data to an hdf5 file
if __name__ == "__main__":
    # Initalize parsing object to pass filename
    aparser = argparse.ArgumentParser()
    aparser.add_argument("--file", "-f", type=str, required=True)
    args = aparser.parse_args()
```

```
packets = IDEXEvent(args.file)
# print(packets.data.keys())
plot_all_data(packets.data, args.file)
write_to_hdf5(packets.data, args.file+'.h5')
# write_to_cdf(packets)
```

It has been updated to be used as a command line tool, so we may use:

```
python idex_packet.py --file filename
```

Where *filename* must be replaced with the name of the packet file, and *python* must be an alias for a python installation. The $--file$ can be substituted for $-f$.

5.4.3 Housekeeping data

UPDATE IN FUTURE DRAFT

5.5 Level 1a to Level 1b Processing

UPDATE IN FUTURE DRAFT

5.6 Level 1b to Level 1c Processing

The level 1b to 1c conversion uses fits to the ion grid and two target signals to estimate the total impact charge.

The fit function is given by (Horányi, 2014),

$$y(t; t_0, C_0, C_1, C_2, \tau_0, \tau_1, \tau_2) = C_0 + H(t - t_0)[C_2\left(1 - e^{-\frac{t-t_0}{\tau_1}}\right)e^{-\frac{t-t_0}{\tau_2}} - C_1] \tag{5.1}$$

where $H(t - t_0)$ is the Heaviside function. Here, C_0 s the initial baseline.

The impact charge is calculated by taking the difference between the initial baseline and the peak of the signal. An example fit for the ion grid signal is depicted in figure 5.3.

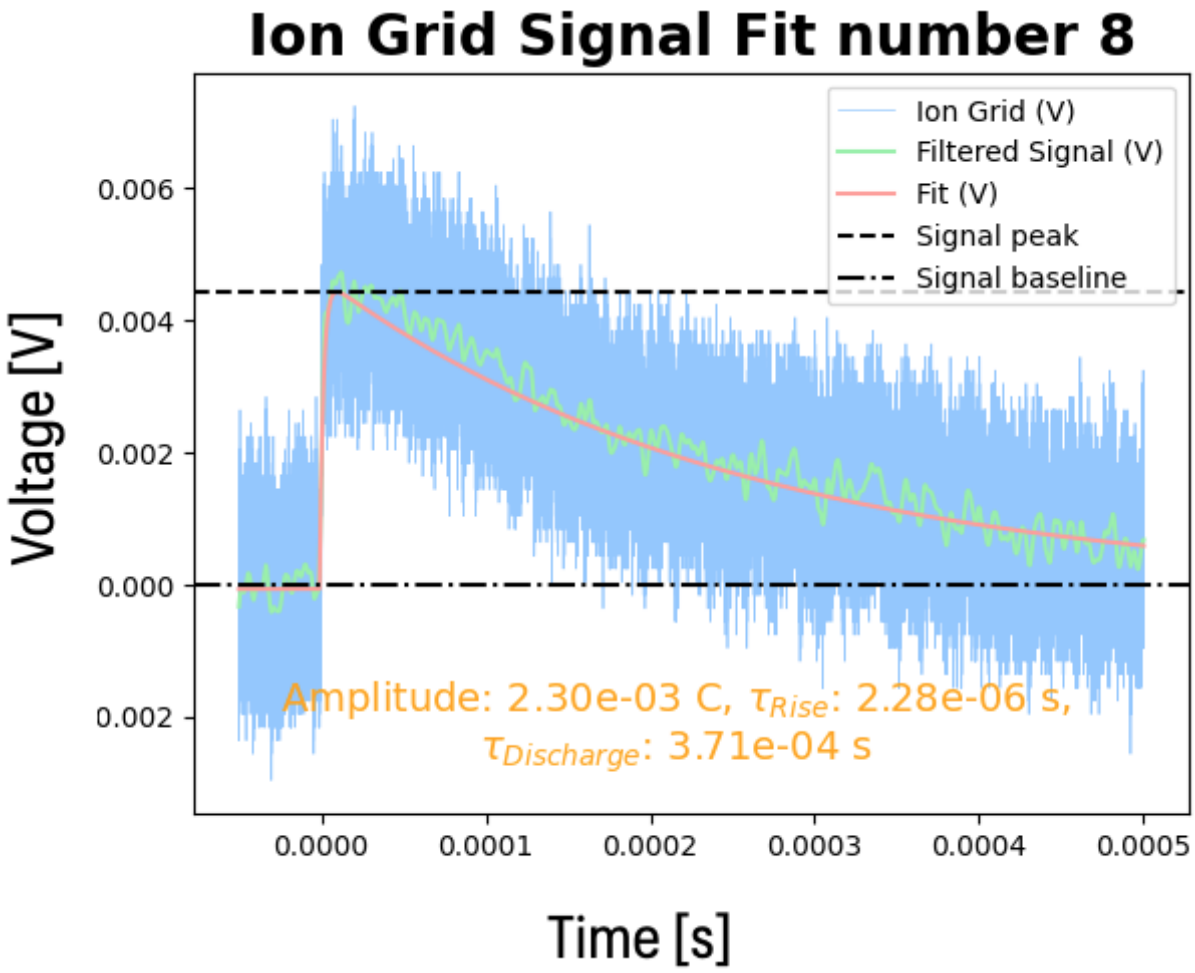

Figure 5.3: An example fit to the target signal. The original signal is blue, the filtered signal is given in green and the fit is the red curve. The fit parameters are displayed in orange text.

Once the signal has been fitted, a linear equation is used to convert the amplitude of the fit from pC to V. This linear equation is derived from fits made to the CSAs during EM calibration. An

example EM calibration fit is shown in figure 5.4.

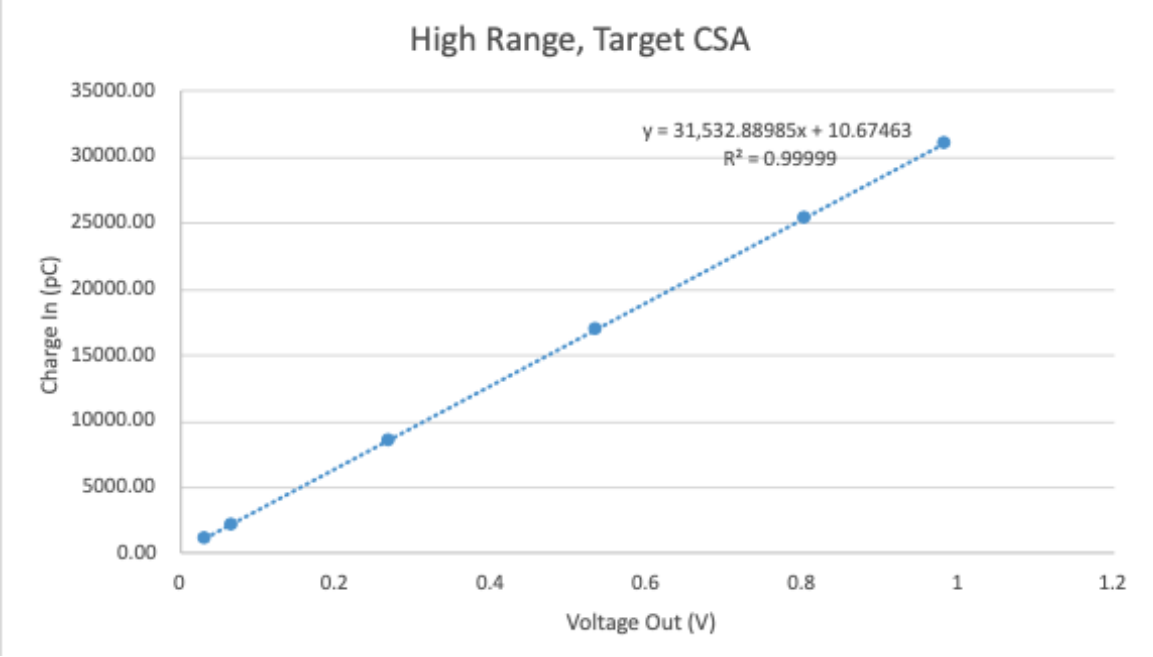

Figure 5.4: An example linear fit for the low gain (high range) target CSA.

Using this equation, fit parameters from a signal in V can be used to calculated the impact charge in pC. These fits have been established for the IDEX EM, and are tabulated in 5.2.

Table 5.2: Engineering Model Calibration Values.

	Variable (pC)	Conv. factor (pC/DN)	Sampling Rate (MHz)	bits
High TOF	$Q_{TOF,tot}$	2.89×10^{-16}	130	10
Mid TOF	$Q_{TOF,tot}$	1.13×10^{-14}	130	10
Low TOF	$Q_{TOF,tot}$	5.14×10^{-13}	130	10
Ion Grid	Q_{IG}	7.46×10^{-4}	4	12
Target High	Q_{tgt}	1.63×10^{-1}	4	12
Target Low	Q_{tgt}	1.58×10^{1}	4	12

5.7 Level 1c to Level 2 Processing

UPDATE IN FUTURE DRAFT

5.8 Level 2 to Level 3 Processing

UPDATE IN FUTURE DRAFT

Appendix A

LASP Packets

The **generator** method from the **parser.py** script in LASP Packets.

```
def generator(self,
                  binary_data: bitstring.ConstBitStream,
                  parse_bad_pkts: bool = True,
                  skip_header_bits: int = 0,
                  root_container_name="CCSDSPacket",
                  ccsds_headers_only: bool = False,
                  yield_unrecognized_packet_errors: bool = False,
                  show_progress: bool = False):
        """Create and return a Packet generator. Creating a generator object to
        ↪  return allows the user to create
        many generators from a single Parser and reduces memory usage.

        Parameters
        ----------
        binary_data : bitstring.ConstBitStream
            Binary data to parse into Packets.
        parse_bad_pkts : bool, Optional
            Default True.
            If True, when the generator encounters a packet with an incorrect
↪   length it will still yield the packet
            (the data will likely be invalid). If False, the generator will still
↪   write a debug log message but will
            otherwise silently skip the bad packet.
        skip_header_bits : int, Optional
            If provided, the parser skips this many bits at the beginning of every
↪   packet. This allows dynamic stripping
            of additional header data that may be prepended to packets.
        root_container_name : str, Optional
            The name of the root level (lowest level of container inheritance)
↪   SequenceContainer. This SequenceContainer
            is assumed to be inherited by every possible packet structure in the
↪   XTCE document and is the starting
```

```
            point for parsing. Default is 'CCSDSPacket'.
        ccsds_headers_only : bool, Optional
            If True, only parses the packet headers (does not use the provided
↪   packet definition).
        yield_unrecognized_packet_errors : bool, Optional
            If False, UnrecognizedPacketTypeErrors are caught silently and parsing
↪   continues to the next packet.
            If True, the generator will yield an UnrecognizedPacketTypeError in
↪   the event of an unrecognized
            packet. Note: These exceptions are not raised by default but are
↪   instead returned so that the generator
            can continue. You can raise the exceptions if desired. Leave this as
↪   False unless you need to examine the
            partial data from unrecognized packets.
        show_progress : bool, Optional
            If True, prints a status bar.

        Yields
        -------
        : Packet or UnrecognizedPacketTypeError
            Generator yields Packet objects containing the parsed packet data for
↪   each subsequent packet.
        """
        def progressbar(current_value, total_value, progress_char, end='\r'):
            bar_length=20
            percentage = int((current_value / total_value) * 100)  # Percent
            ↪  Completed Calculation
            progress = int((bar_length * current_value) / total_value)  # Progress
            ↪  Done Calculation
            loadbar = "Progress: [{:{len}}]{}%".format(progress * progress_char,
            ↪  percentage,
                                                       len=bar_length)  # Progress
                                                       ↪  Bar String
            print(loadbar, end=end)

        start_time = time.time_ns()
        n_packets = 0
        total_length = len(binary_data)
        while binary_data.pos < total_length:
            if show_progress is True:
                progressbar(binary_data.pos, total_length, progress_char="=")


            if skip_header_bits:
                binary_data.pos += skip_header_bits

            # Parse the header but leave the cursor in the original position
            ↪  because the header is defined
```

```
#   in the overall packet definition as well. We only parse the header
↪  here, so we can determine the
#   packet definition type (usually by APID, VERSION, and TYPE but
↪  potentially based on
#   any header data).
packet_starting_position = binary_data.pos
logger.debug(f"Parsing packet starting at {packet_starting_position}.
↪  Total length is {total_length}.")
header = self._parse_header(binary_data, reset_cursor=True)
n_packets += 1  # Consider it a counted packet once we've parsed the
↪  header
specified_packet_data_length =
↪  self._total_packet_bits_from_pkt_len(header['PKT_LEN'].raw_value)
if ccsds_headers_only is True:
    binary_data.pos += specified_packet_data_length
    yield Packet(header=header, data=None)
    continue

logger.debug(
    f"Packet header: "
    f"{' '.join([str(parsed_param) for param_name, parsed_param in
    ↪  header.items()])}")

try:
    if isinstance(self.packet_definition,
    ↪  xtcedef.XtcePacketDefinition):
        packet = self.parse_packet(binary_data,

                                   ↪  self.packet_definition.named_containers,

                                   ↪  root_container_name=root_container_name,
                                   word_size=self.word_size)
    else:
        packet_def_name, parameter_list =
        ↪  self._determine_packet_by_restrictions(header)
        packet = self.legacy_parse_packet(binary_data, parameter_list,
        ↪  word_size=self.word_size)

except UnrecognizedPacketTypeError as e:
    # Regardless of whether we handle it, we do want to move the
    ↪  cursor to the next packet
    binary_data.pos = packet_starting_position +
    ↪  specified_packet_data_length
    logger.debug(f"Unrecognized error on packet with APID
    ↪  {header['PKT_APID'].raw_value}'")
    if yield_unrecognized_packet_errors is True:
        # Yield the caught exception without raising it (raising ends
        ↪  generator)
```

```
                yield e
                continue
            else:
                # Silently continue to next packet
                continue

        if packet.header['PKT_LEN'].raw_value != header['PKT_LEN'].raw_value:
            raise ValueError(f"Hardcoded header parsing found a different
            ↪  packet length "
                             f"{header['PKT_LEN'].raw_value} than the
                             ↪  definition-based parsing found "
                             f"{packet.header['PKT_LEN'].raw_value}. This
                             ↪  might be because the CCSDS header is "
                             f"incorrectly represented in your packet
                             ↪  definition document.")

        actual_length_parsed = binary_data.pos - packet_starting_position

        if actual_length_parsed != specified_packet_data_length:
            logger.warning(f"Parsed packet length starting at bit position:
            ↪  {binary_data.pos} "
                           f"({actual_length_parsed}b) did not match "
                           f"length specified in header
                           ↪  ({specified_packet_data_length}b). "
                           f"Updating bit string position to correct position
                           ↪  "
                           "indicated by CCSDS header.")
            binary_data.pos += specified_packet_data_length -
            ↪  actual_length_parsed
            if not parse_bad_pkts:
                logger.warning("Skipping bad packet because parse_bad_pkts is
                ↪  falsy.")
                continue

        yield packet
    end_time = time.time_ns()
    elapsed_ns = end_time - start_time
    delta = dt.timedelta(microseconds=elapsed_ns / 1000)
    kbps = int(total_length * 1E6 / elapsed_ns)
    pps = int(n_packets * 1E9 / elapsed_ns)
    if show_progress is True:
        progressbar(binary_data.pos, total_length, progress_char="=", end=f"
        ↪  [Elapsed: {delta}, Parsed {total_length} bits ({n_packets}
        ↪  packets) at {kbps} kb/s ({pps} pkts/s)]\n")
```

Appendix B

Telemetry Packet Conversions

Appendix C

IDEX/SUDA Combined XTCE Definition

The first 100 lines of the combined IDEX/SUDA XTCE definition are shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xtce:SpaceSystem xmlns:xtce="http://www.omg.org/space/xtce" name="IDEX">
  <xtce:Header classification="BuildType=FLIGHT,CreatedBy=XTCEEditor" date="Created=2023/01/12 15:27:45" version="DocumentVersion=FSW Build ,DocumentVersion=20190109v1,LASPXTCESchemaVersion=1.2,XTCEEditorVersion=0.47"/>
  <xtce:TelemetryMetaData>
    <xtce:ParameterTypeSet>
      <xtce:IntegerParameterType signed="false" name="VERSION_Type">
        <xtce:UnitSet/>
        <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="3" />
      </xtce:IntegerParameterType>
      <xtce:IntegerParameterType signed="false" name="TYPE_Type">
        <xtce:UnitSet/>
        <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="1" />
      </xtce:IntegerParameterType>
      <xtce:IntegerParameterType signed="false" name="SEC_HDR_FLG_Type">
        <xtce:UnitSet/>
        <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="1" />
      </xtce:IntegerParameterType>
      <xtce:IntegerParameterType signed="false" name="PKT_APID_Type" >
        <xtce:UnitSet/>
```

```
20          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="11
    "/>
21        </xtce:IntegerParameterType>
22        <xtce:IntegerParameterType signed="false" name="SEQ_FLGS_Type"
    >
23          <xtce:UnitSet/>
24          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="2"
    />
25        </xtce:IntegerParameterType>
26        <xtce:IntegerParameterType signed="false" name="
    SRC_SEQ_CTR_Type">
27          <xtce:UnitSet/>
28          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="14
    "/>
29        </xtce:IntegerParameterType>
30        <xtce:IntegerParameterType signed="false" name="PKT_LEN_Type">
31          <xtce:UnitSet/>
32          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="16
    "/>
33        </xtce:IntegerParameterType>
34        <xtce:FloatParameterType name="SHCOARSE_Type">
35          <xtce:UnitSet>
36            <xtce:Unit>dn</xtce:Unit>
37          </xtce:UnitSet>
38          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="32
    "/>
39        </xtce:FloatParameterType>
40        <xtce:FloatParameterType name="SHFINE_Type">
41          <xtce:UnitSet>
42            <xtce:Unit>dn</xtce:Unit>
43          </xtce:UnitSet>
44          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="16
    "/>
45        </xtce:FloatParameterType>
46        <xtce:FloatParameterType name="CHECKSUM_Type">
47          <xtce:UnitSet>
48            <xtce:Unit>dn</xtce:Unit>
49          </xtce:UnitSet>
50          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="32
    "/>
51        </xtce:FloatParameterType>
52        <xtce:IntegerParameterType name="IDX__SCI0AID_Type">
53          <xtce:UnitSet/>
54          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="32
    "/>
55        </xtce:IntegerParameterType>
56        <xtce:IntegerParameterType name="IDX__SCI0TYPE_Type">
57          <xtce:UnitSet/>
```

```
58          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="8"
    />
59        </xtce:IntegerParameterType>
60        <xtce:IntegerParameterType name="IDX__SCI0CONT_Type">
61          <xtce:UnitSet/>
62          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="8"
    />
63        </xtce:IntegerParameterType>
64        <xtce:IntegerParameterType name="IDX__SCI0SPARE1_Type">
65          <xtce:UnitSet/>
66          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="13
    "/>
67        </xtce:IntegerParameterType>
68        <xtce:EnumeratedParameterType name="IDX__SCI0PACK_Type">
69          <xtce:UnitSet/>
70          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="1"
    />
71          <xtce:EnumerationList>
72            <xtce:Enumeration value="0" label="DS"/>
73            <xtce:Enumeration value="1" label="EN"/>
74          </xtce:EnumerationList>
75        </xtce:EnumeratedParameterType>
76        <xtce:EnumeratedParameterType name="IDX__SCI0FRAG_Type">
77          <xtce:UnitSet/>
78          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="1"
    />
79          <xtce:EnumerationList>
80            <xtce:Enumeration value="0" label="DS"/>
81            <xtce:Enumeration value="1" label="EN"/>
82          </xtce:EnumerationList>
83        </xtce:EnumeratedParameterType>
84        <xtce:EnumeratedParameterType name="IDX__SCI0COMP_Type">
85          <xtce:UnitSet/>
86          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="1"
    />
87          <xtce:EnumerationList>
88            <xtce:Enumeration value="0" label="DS"/>
89            <xtce:Enumeration value="1" label="EN"/>
90          </xtce:EnumerationList>
91        </xtce:EnumeratedParameterType>
92        <xtce:IntegerParameterType name="IDX__SCI0EVTNUM_Type">
93          <xtce:UnitSet/>
94          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="16
    "/>
95        </xtce:IntegerParameterType>
96        <xtce:IntegerParameterType name="IDX__SCI0CAT_Type">
97          <xtce:UnitSet/>
```

```
98          <xtce:IntegerDataEncoding encoding="unsigned" sizeInBits="8"
    />
```