UNIVERSITY OF CALIFORNIA,
IRVINE

ATLAS Particle Detector CSC ROD Software Design and Implementation

AND

Addition of K Physics to Chi-Squared Analysis of FDQM

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Physics

by

Donovan Lee Hawkins

Dissertation Committee:
Professor Andrew Lankford, Chair
Professor Dennis J. Silverman
Professor Myron Bander

2005

The dissertation of Donovan Lee Hawkins
is approved and is acceptable in quality
and form for publication on microfilm

____

____

____

Committee Chair

University of California, Irvine
2005

DEDICATION

To

my mother

who has always been there

and

my grandfather

who I know wanted to be here now.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank the members of my committee for their time and effort. In particular I would like to thank my thesis advisors, Drs. Lankford and Silverman, for their suggestions and revisions. I would also like to thank Drs. Bander, Lankford, and Silverman for their support as thesis advisors both present and previous. Finally, I would like to thank Dr. Bander for his suggestions regarding my selection of these thesis topics.

Curriculum Vitae

Donovan Lee Hawkins

1996	B.S. in Physics with a minor in Mathematics, University of California, Irvine
1998	M.S. in Physics, University of California, Irvine
2005	Ph.D. in Physics, University of California, Irvine

FIELD OF STUDY

Particle Physics

PUBLICATIONS

Hawkins, Donovan *et al.*: *The Interface Vibrational Contribution to the Thermodynamic Functions of the System of Liquid and Solid Separated by a Planar Interface*, Solid State Communications **103**: 381, 27 March 1997

Hawkins, Donovan *et al.*: *Phonon contribution to wetting phenomena: macroscopic theory*, Physics Letters **A234**: 225, 21 July 1997

Dailing, J. *et al.*: *Off-Detector Electronics for a High-Rate CSC Detector,* PROCEEDINGS of the Sixth Workshop on Electronics for LHC Experiments, 11 September 2000

Dailing, J. *et al.*: *Performance and Radiation Tolerance of the ATLAS CSC On-Chamber Electronics,* PROCEEDINGS of the Sixth Workshop on Electronics for LHC Experiments, 11 September 2000

Hawkins, Donovan and Silverman, Dennis: *Isosinglet down quark mixing and CP violation experiments*, Phys. Rev. **D66**: 16008, 31 July 2002

Drego, N. *et al.*: *Off-Detector Electronics for High-Rate CSC Detector*, IEEE Transactions on Nuclear Science **51**: 461, June 2004

Abstract of the Dissertation

ATLAS Particle Detector CSC ROD Software Design and Implementation

And

Addition of K Physics to Chi-Squared Analysis of FDQM

By

Donovan Lee Hawkins

Doctor of Philosophy in Physics

University of California, Irvine, 2005

Professor Andrew Lankford, Chair

In this thesis I present a software framework for use on the ATLAS muon CSC readout driver. This C++ framework uses plug-in Decoders incorporating hand-optimized assembly language routines to perform sparsification and data formatting. The software is designed with both flexibility and performance in mind, and runs on a custom 9U VME board using Texas Instruments TMS360C6203 digital signal processors. I describe the requirements of the software, the methods used in its design, and the results of testing the software with simulated data.

I also present modifications to a chi-squared analysis of the Standard Model and Four Down Quark Model (FDQM) originally done by Dr. Dennis Silverman. The addition of four new experiments to the analysis has little effect on the Standard Model but provides important new restrictions on the FDQM. The method used to incorporate these new experiments is presented, and the consequences of their addition are reviewed.

PART I

ATLAS Muon CSC ROD Software

Motivation

Introduction/Summary of Results

The primary goal of a particle detector is to provide the information necessary to reconstruct what occurred during a collision. Particles must be identified, their momenta determined, and a quick decision made whether to record the data for offline analysis. Specialized hardware and software is needed to digitize and process the signals from detectors and pass them on for analysis and storage. In the ATLAS particle detector, the first tier of this process is the readout driver (ROD). These are custom boards that convert data for different detector subsystems into a common format. The RODs allow the remainder of the data processing chain to be designed independently of the design of the individual detector subsystems.

Only a tiny fraction of a percent of the collisions will be saved in ATLAS. In the muon CSC subsystem, the ROD is responsible for a significant amount of data reduction by suppressing channels containing only noise and eliminating data that comes from times other than the trigger of interest. This requires a significant amount of data processing power to apply this reduction at the 100kHz design trigger rate. By using high-speed digital signal processor (DSP) chips, we can achieve both high performance and a high degree of flexibility when compared to a traditional FPGA design. In fact, the resulting system has turned out to be flexible enough to handle a number of unexpected data formats used during testing without modification of the core software framework. This fact has both leveraged the effort put into this system and provided an early test bed for exercising the software and confirming its proper operation.

Currently the software has reached a level of performance necessary to complete all of the processing required in ATLAS, and has been tested to ensure proper operation without errors. Further testing on the algorithms and implementation using Monte Carlo studies is needed to confirm earlier predictions on the rejection and acceptance rates.

Higgs Physics

Introduction

We begin with a brief examination of the connection between phase and the electromagnetic field. This is **NOT** a calculation (the canonical replacement should be done before solving) but it is still illuminating.

- Classical gauge transformation (4-vector notation):
$$A_\mu \rightarrow A_\mu + \partial_\mu \Lambda(x)$$

- Plane-wave solution in quantum mechanics (4-vector notation):
$$\Psi = e^{ip_\mu x^\mu}$$

- Canonical replacement for introducing E&M to quantum mechanics (4-vector notation):
$$p_\mu^{em} = p_\mu - eA_\mu$$

- Resulting plane-wave solution for quantum + E&M (**NOT** a valid step):
$$\Psi = e^{i(p_\mu - eA_\mu)x^\mu}$$

- Resulting gauge transformation:
$$\Psi \rightarrow e^{i(p_\mu - eA_\mu - e\partial_\mu \Lambda(x))x^\mu} = e^{-ie\partial_\mu \Lambda(x)\, x^\mu}\Psi$$

This hints at the fact that the gauge term Λ causes a phase shift in the wave function. Solving the equations properly with integration would eliminate the derivative in the exponent. In retrospect, this fact is not surprising given the well-known Aharonov-Bohm effect where a difference in phase is related to the line integral of the vector potential.

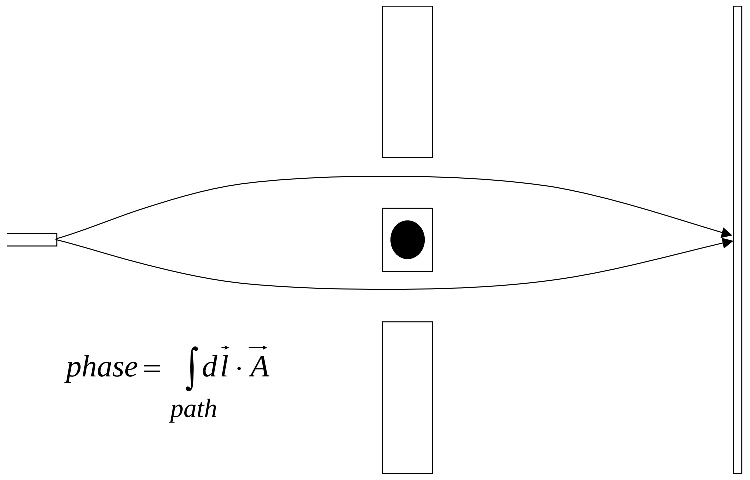


Figure 1: Aharonov-Bohm effect

We are now inspired to believe that a quantum phase that is an arbitrary function of space and time (called a local gauge transformation) is analogous to a gauge transformation in E&M. Such transformations don't change the physics, so we would like to examine a theory that includes this symmetry. Clearly a phase shift is of interest when we have a complex scalar field, which is a straightforward extension of a real scalar field.

- Real scalar field (simplest case with only dynamic and mass terms):

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\,\partial^\mu\phi - \frac{1}{2}m^2\phi^2$$

$$\partial_\mu\left(\frac{\partial\mathcal{L}}{\partial\left(\partial_\mu\phi\right)}\right) - \frac{\partial\mathcal{L}}{\partial\phi} = \partial_\mu\left(\frac{1}{2}\partial^\mu\phi + \frac{1}{2}\partial^\mu\phi\right) + m^2\phi = \left(\partial^\mu\partial_\mu + m^2\right)\phi = 0$$

- Complex scalar field (simplest case with only dynamic and mass terms):

$$\mathcal{L} = \partial_\mu\phi\,\partial^\mu\phi^* - m^2\phi^*\phi$$

- Global gauge transformation is a symmetry:

$$\phi \to e^{-i\Lambda}\phi\,,\ \phi^* \to e^{i\Lambda}\phi^*$$

- Local gauge transformation is not but should be:

$$\phi \to e^{-i\Lambda(x)}\phi\,,\ \phi^* \to e^{i\Lambda(x)}\phi^*$$

$$\partial_\mu\phi \to e^{-i\Lambda(x)}\partial_\mu\phi - i\partial_\mu\Lambda(x)\,e^{-i\Lambda(x)}\phi$$

- Introduce a new field A_μ to solve the problem:

$$\left(\partial_\mu + ieA_\mu\right)\phi \to e^{-i\Lambda(x)}\partial_\mu\phi - i\partial_\mu\Lambda(x)\,e^{-i\Lambda(x)}\phi + ieA_\mu e^{-i\Lambda(x)}\phi$$

$$ieA_\mu\phi \to ieA_\mu e^{-i\Lambda(x)}\phi + i\partial_\mu\Lambda(x)\,e^{-i\Lambda(x)}\phi$$

$$eA_\mu \to eA_\mu + \partial_\mu\Lambda(x)$$

- Complex scalar field invariant under local gauge transformations:

$$\mathcal{L} = \left(\partial_\mu + ieA_\mu\right)\phi\left(\partial^\mu - ieA^\mu\right)\phi^* - m^2\phi^*\phi = \left(D_\mu\phi\right)\left(D^\mu\phi\right)^* - m^2\phi^*\phi$$

The A_μ field, contained within the covariant derivative D^μ, is in exactly the right place to be the electromagnetic field in a canonical replacement. In order to treat this field as a particle (the photon), we need to add a term that will produce the standard equations of electromagnetism in the resulting equations of motion for the A_μ field.

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu$$
$$\mathcal{L} = \left(D_\mu \phi\right)\left(D^\mu \phi\right)^* - m^2 \phi^* \phi - \frac{1}{4} F_{\mu\nu} F^{\mu\nu}$$
$$\partial_\mu F^{\mu\nu} = ie\left[\phi^* \left(D^\nu \phi\right) - \phi\left(D^\nu \phi\right)^*\right] = eJ^\nu$$

For this photon to be massive, we would need a term like $\mathcal{L}_\gamma = \frac{1}{2} m_\gamma^2 A_\mu A^\mu$. However, this is not local gauge invariant so it is not allowed. This is consistent with what we know of photons and does not bother us. However, there are other forces such as the weak interaction that have massive force carriers. It would be nice to find a way to use the same mechanism to introduce massive force carriers as well. The solution is called the Higgs mechanism, and it can be seen most easily using a simple scalar field theory.

- "Phi-fourth" theory:

$$\mathcal{L} = \partial_\mu \phi \, \partial^\mu \phi^* - \mu^2 \phi^* \phi - \lambda \left(\phi^* \phi\right)^2$$

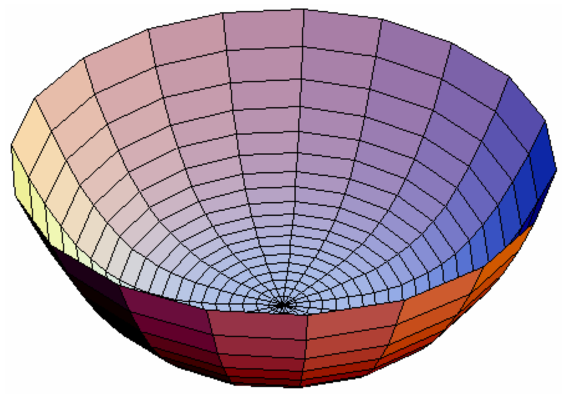
- Local gauge invariant phi-fourth theory:

$$\mathcal{L} = \left(D_\mu \phi\right)\left(D^\mu \phi\right)^* - \mu^2 \phi^* \phi - \lambda \left(\phi^* \phi\right)^2 - \frac{1}{4} F_{\mu\nu} F^{\mu\nu}$$
$$D_\mu = \partial_\mu + ieA_\mu$$

- Ground state is the minimum of the "potential":

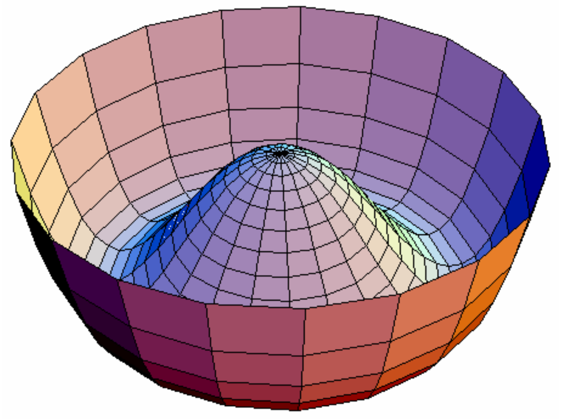
$$V(\phi) = \mu^2 |\phi|^2 + \lambda |\phi|^4$$

$\mu^2 > 0$:



minimum at $|\phi| = 0$

$\mu^2 < 0$:



minimum at $|\phi| = v = \sqrt{\frac{-\mu^2}{2\lambda}}$

Figure 2: Potential plots for a local gauge-invariant phi-fourth theory

- Perform a slight change of variable:
$$\phi = v + \chi$$
$$\mathcal{L} = \left(D_\mu \chi\right)\left(D^\mu \chi\right)^* + 2\lambda v^2 \chi^* \chi - \lambda\left(\chi^* \chi\right)^2 - \frac{1}{4}F_{\mu\nu}F^{\mu\nu}$$
$$+e^2\left(2v\,\mathrm{Re}\,\chi + v^2\right)A_\mu A^\mu - 2iev A_\mu \partial^\mu\,\mathrm{Im}\,\chi$$
$$-2\lambda v\left(2\,\mathrm{Re}\,\chi + v\right)\left(\chi^* \chi\right) - \lambda v^2\left(2\,\mathrm{Re}\,\chi^2 - 2\,\mathrm{Im}\,\chi^2\right)$$

- The χ field is no longer physical, but our photon has acquired a mass term:
$$\mathcal{L} = \frac{1}{2}m_\gamma^2 A_\mu A^\mu$$
$$m_\gamma = \sqrt{2}ev$$

- We can instead select new independent fields η and ξ:
$$\phi = v + \frac{1}{\sqrt{2}}(\eta + i\xi)$$
$$\mathcal{L} = \frac{1}{2}\partial_\mu \eta\, \partial^\mu \eta - \lambda v^2 \eta^2 - \lambda v\sqrt{2}\eta^3 - \lambda\eta^4$$
$$+\frac{1}{2}\partial_\mu \xi\, \partial^\mu \xi + \lambda v^2 \xi^2 - \lambda \xi^4 - \frac{1}{4}F_{\mu\nu}F^{\mu\nu}$$
$$+e^2\left(\frac{\eta^2 + \xi^2}{2} + \sqrt{2}\eta v + v^2\right)A_\mu A^\mu - e\xi A_\mu \partial^\mu \eta$$
$$+e\eta A_\mu \partial^\mu \xi + \sqrt{2}ev A_\mu \partial^\mu \xi - 2\lambda\eta^2\xi^2 - \sqrt{2}\lambda v\eta\xi^2$$

The unphysical ξ field can be eliminated in the unitarity gauge $\phi \to e^{i\tan^{-1}\frac{\mathrm{Im}\phi}{\mathrm{Re}\phi}}\phi$. Thus, we have absorbed the imaginary part of ϕ to become the extra degree of freedom needed for the A_μ field to acquire a mass. The remaining real part of ϕ becomes a massive real scalar field. We can apply the same technique to the electroweak Lagrangian to generate the massless photon A, the massive $W^\pm$ and Z, and a new massive particle called the Higgs. A similar mechanism can generate quark and lepton masses via interactions with the Higgs field by assuming a coupling between the fermion currents and the Higgs with strength proportional to the fermion mass.

Examination of Channels Used for Higgs

Because the coupling of the Higgs to fermions is proportional to the mass of the fermions, the Higgs will generally decay into the most massive particles that are kinematically available to it. Which decay that is will depend on the currently unknown mass of the Higgs.

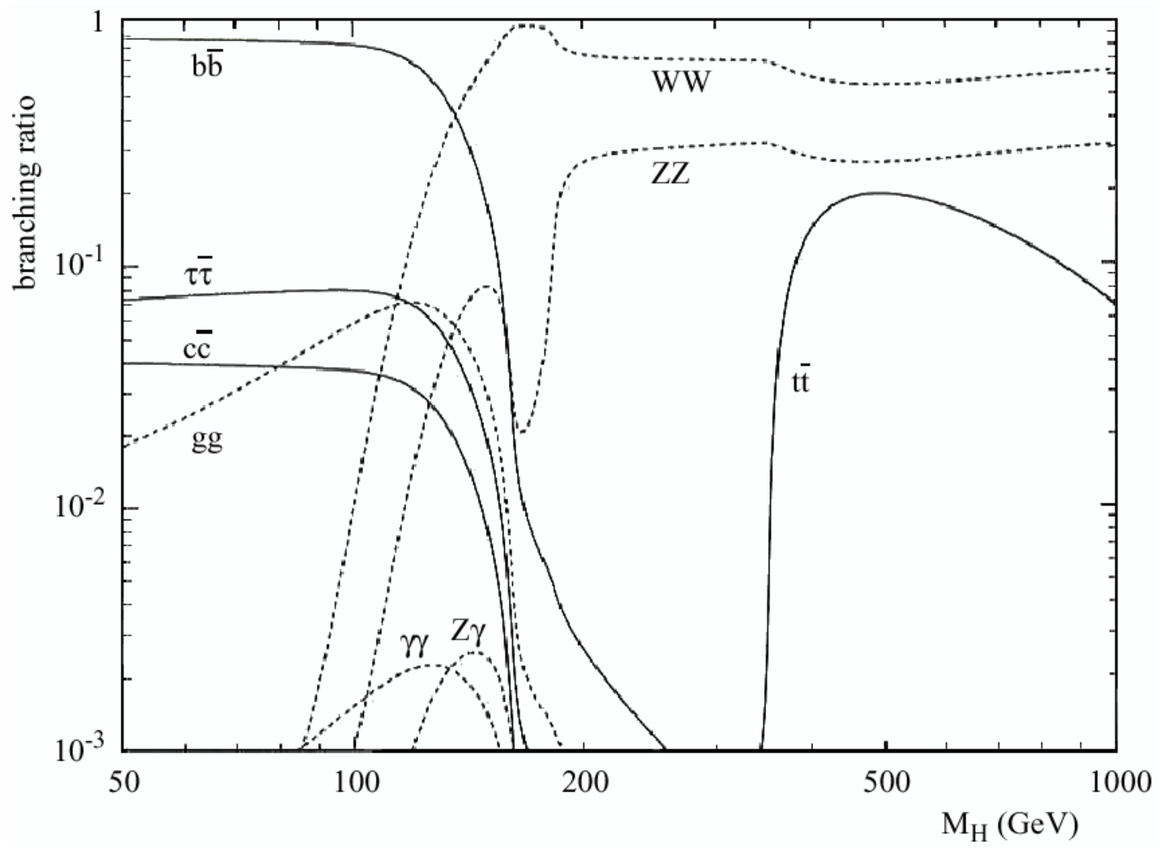


Figure 3: Higgs branching ratios as a function of mass [26]

Branching ratio is not the whole story, however. What matters is how that branching ratio compares to the background that will interfere with its identification. This is given in terms of signal significance, which is the number of signal events expected divided by the square root of the number of background events expected.

Because there is a high background of QCD jets, it will be difficult to detect the Higgs decaying to a quark-antiquark pair. For a Higgs mass of less than 150 GeV where these channels are important, it is necessary to look for a Higgs being produced together with other heavy particles to reduce the background.

For a Higgs mass between 120 GeV and 160 GeV we start to see the production of a single W or Z boson together with a corresponding virtual W or Z boson, which then decay to four leptons. For the Z bosons we get $H \rightarrow ZZ^* \rightarrow l\bar{l}l\bar{l}$ ($ee\bar{e}\bar{e}$, $\mu\bar{\mu}\mu\bar{\mu}$, or $e\bar{e}\mu\bar{\mu}$), while the W bosons have the process $H \rightarrow WW^* \rightarrow l\nu l\nu$. Because the neutrinos can only be inferred kinematically, the Z process provides more information for reconstruction and is preferable in spite of a lower cross section.

These processes reach even greater importance as we cross the mass thresholds that allow for the production of two on-shell bosons. From a Higgs mass around 180 GeV up to 700 GeV, the four lepton decay of two on-shell Z bosons is called the "gold-plated channel" because the background is continuum Z production. A pair of Z bosons produced from the single-body decay of a Higgs concentrate the momentum of the resulting leptons in opposite directions. This, combined with the known mass of the on-shell Z intermediaries, helps reduce the background from continuum Z bosons.

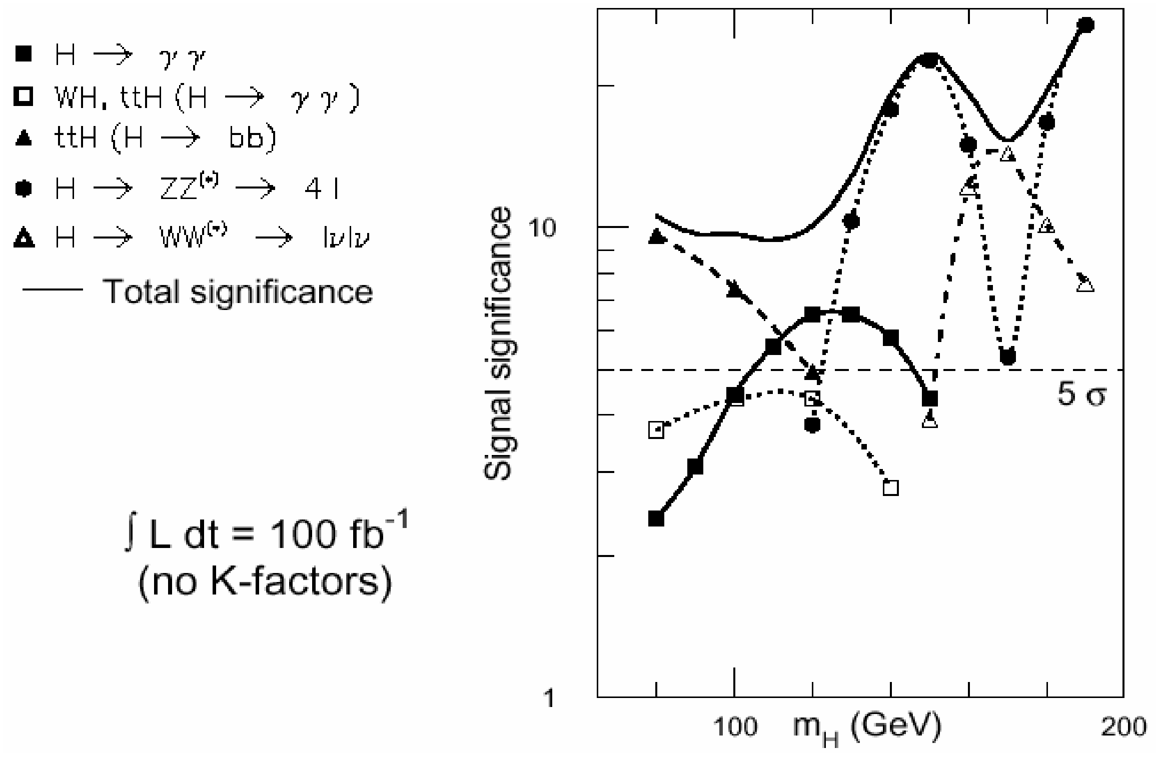


Figure 4: Signal significance as a function of Higgs mass after one year of high luminosity running [1]

Detector

Description of CERN and the LHC

CERN was founded in 1954 and is now the largest particle physics center in the world [16]. Located on the border between France and Switzerland near Geneva, it is the location of LEP (Large Electron-Positron collider). CERN has been an important site for particle physics, and is also the birthplace of the World Wide Web.

The LHC (Large Hadron Collider) is a new collider being built at CERN in the LEP tunnel. It will produce counter-rotating proton beams at 7 TeV inside its 16-mile-long tunnel (the largest of any accelerator).

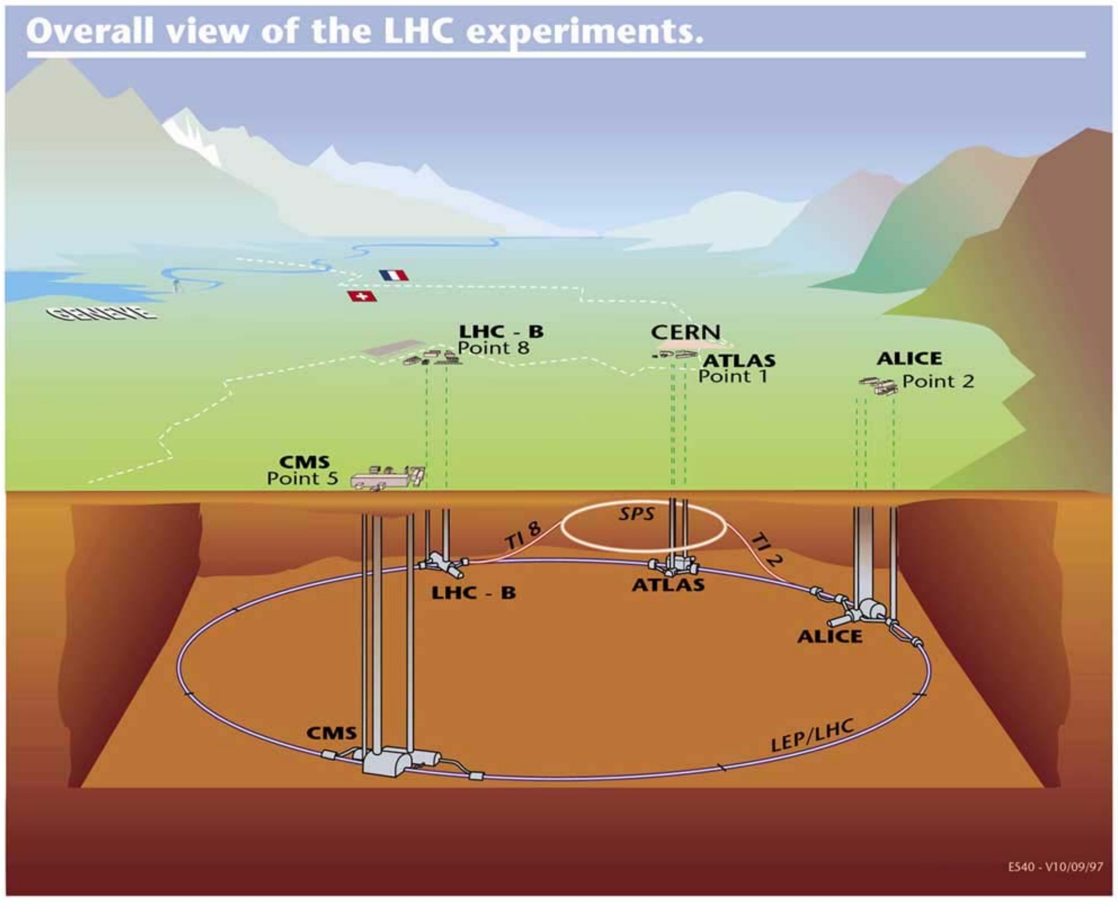


Figure 5: LHC and experiments map [5]

Bunches of about a billion protons are spaced 7.5m (25ns) apart and produce up to around 20 proton-proton collisions when they meet head-on at one of the collision points.

The LHC is currently under construction and is scheduled for final commissioning in 2007. [31]

LHC Specifications: [30]

Accelerates protons and fully-ionized lead ions

26,659 m circumference with an 11.2455 kHz frequency of revolution

Protons injected by SPS (Super Proton Synchrotron) at 450 GeV and accelerated to 7 TeV

Approximately 9300 magnets with up to an 8.33 T magnetic field

Approximately 120 MW of power consumed during operation

Description of ATLAS and the Muon Subsystem

ATLAS

ATLAS = **A** **T**oroidal **L**HC **A**pparatu**S**

ATLAS is a five-story-tall particle detector being built by a collaboration of 1800 physicists in 34 countries. It is the largest collaborative effort ever in physical sciences. [4]

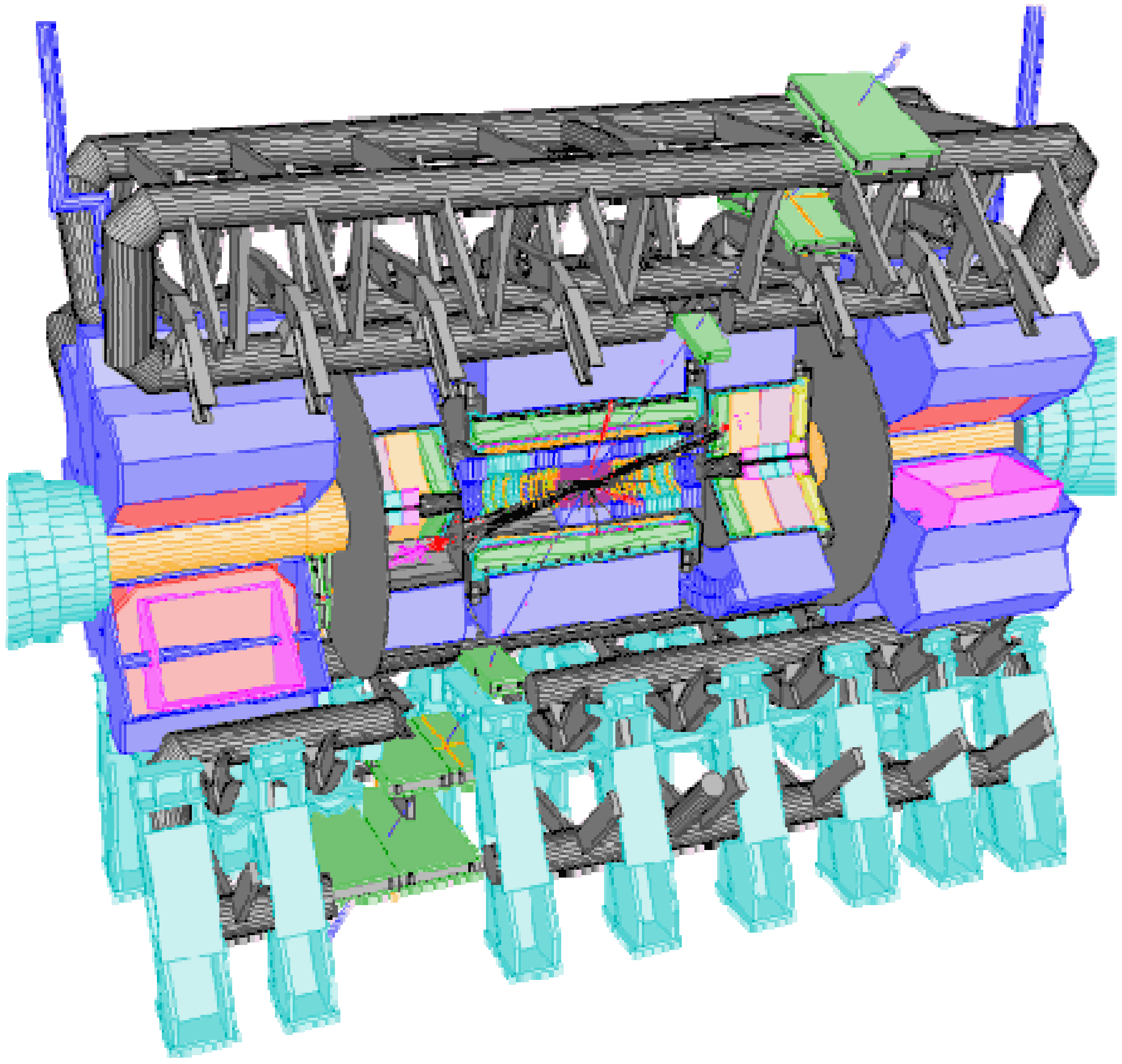


Figure 6: The ATLAS detector [1]

ATLAS has four detector systems, each of which contains various specific types of detectors designed to give different information on the particles produced. Taken together, they provide a complete picture of the collision.

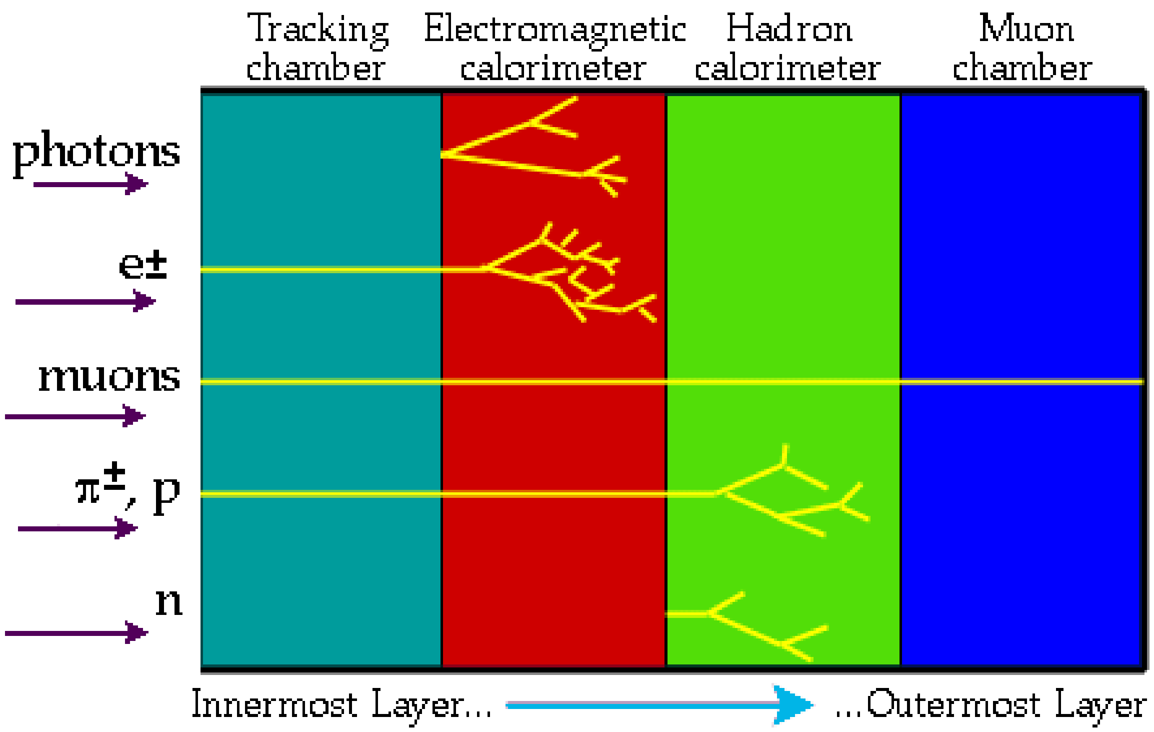


Figure 7: ATLAS detector systems [6]

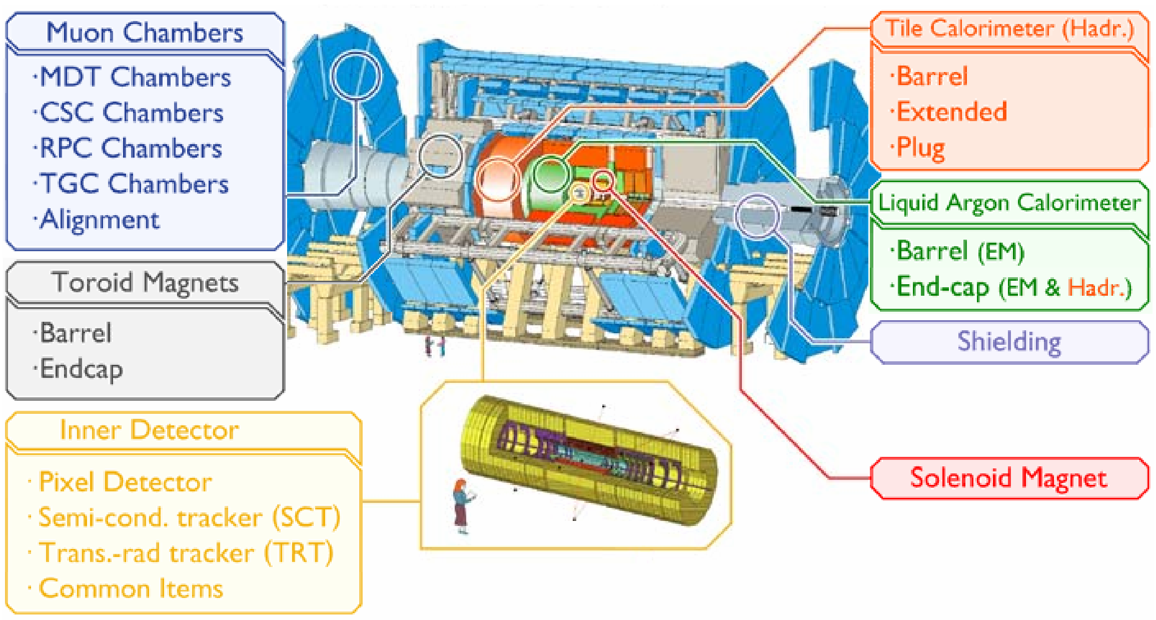


Figure 8: Breakdown of ATLAS systems [7]

Muon System

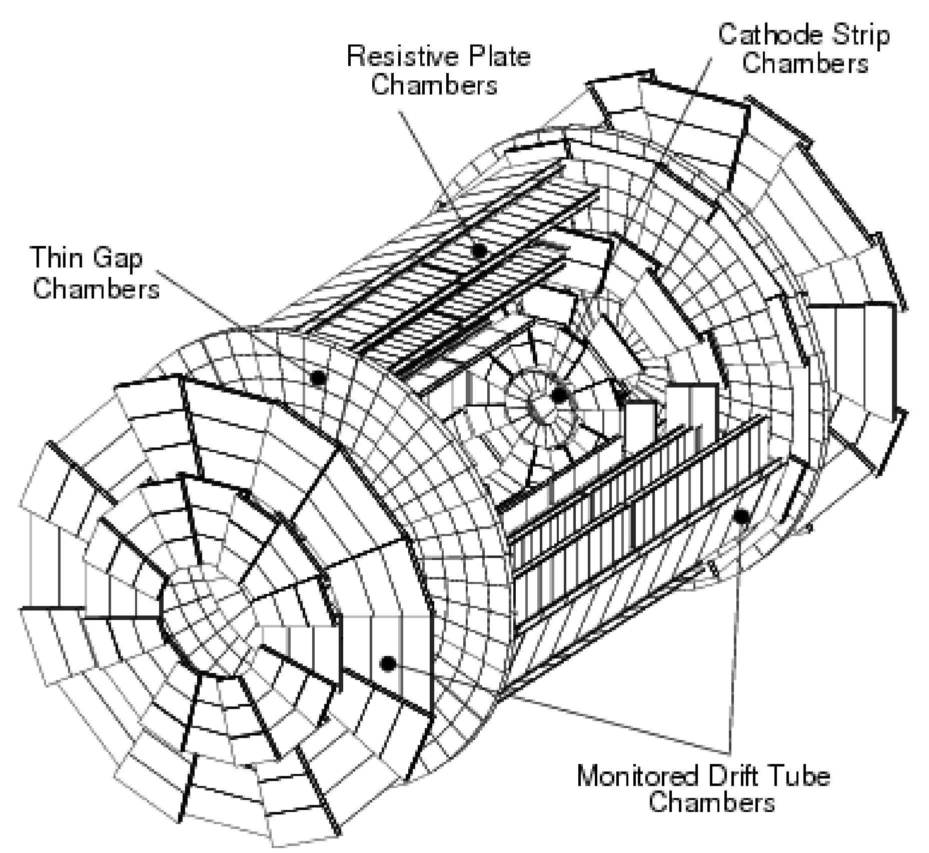


Figure 9: Muon system [3]

The muon system can be grouped into two categories:

- Muon precision chambers (MDT and CSC)
- Muon trigger chambers (RPC and TGC)

The precision chambers produce the critical measurements in the bending direction needed to obtain muon momentum. The trigger chambers provide prompt information for deciding whether to keep an event as well as producing measurements in the non-bending direction for MDT (which has no such capability).

The Muon CSC chambers represent only half a percent of the total area for precision chambers but account for over 15% of the channels. This is necessary because the CSC chambers are located in the highest rate area of the muon system.

	Precision chambers		Trigger chambers	
	CSC	MDT	RPC	TGC
Number of chambers	32	1194	596	192
Number of channels	67,000	370,000	355,000	440,000
Area (square inches)	27	5500	3650	2900

Table 1: Muon chamber numerics [3]

Muon CSC

The muon cathode strip chambers (CSCs) start as two layers of cathode strips with anode wires in between. The precision layer has 192 channels and measures in the direction of magnetic curvature, while the transverse layer has only 48 channels for measurement in the less critical non-bending direction.

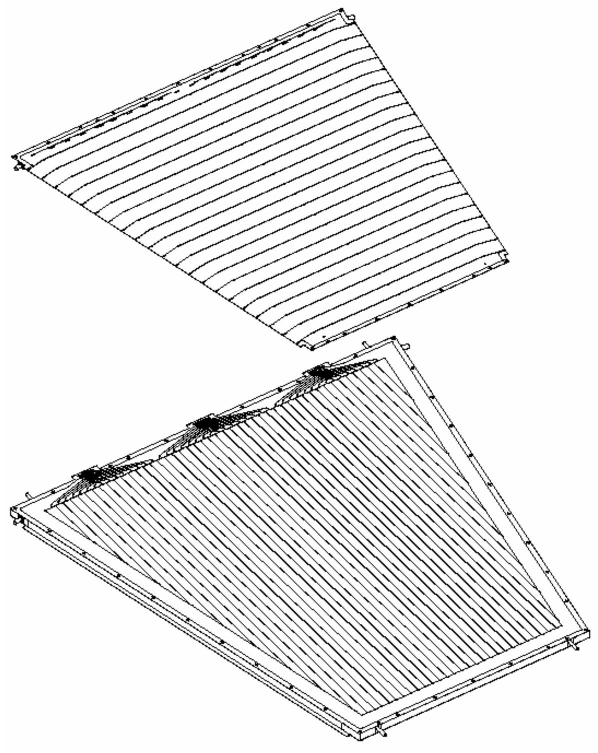


Figure 10: Diagram of CSC planes [3]

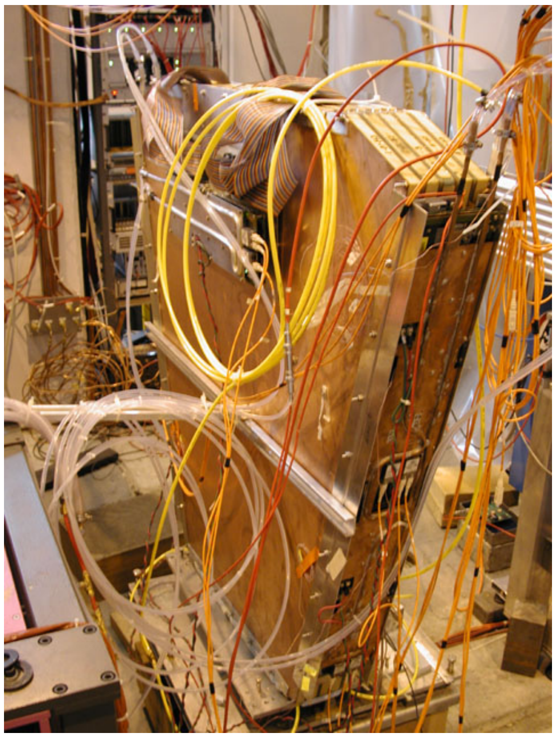


Figure 11: Photo of assembled CSC [42]

Four of these layer pairs are combined to form a single chamber. Each chamber therefore has four precision layers of 192 channels each and four transverse layers with a total of 192 channels (48 each). Five ASM-II boards (four precision, one transverse) are attached to the chamber. Each of these boards buffers, digitizes, and serializes the 192 strip voltages onto a pair of gigabit fiber optic cables. The fiber optic cables carry the data out of the radiation area to be processed by the RODs.

Chamber data read out for a single beam crossing is called a time slice. Each event contains four such time slices taken at 50ns intervals (two beam crossings), which means events can overlap.

Trigger/DAQ Chain and the Role of the ROD

What starts as 40 million events per second from each detector must be reduced the point where it can be permanently stored for offline analysis. The first of these reductions is performed by the Level 1 Trigger, which has access to prompt information on jets and clusters in the calorimeters and muon trigger detectors. This information is compared against a "menu" which lists at what rates different types of events are to be kept. The total of all events kept must be less than the maximum design value of 75 kHz (to be upgraded to 100 kHz at a later date), and there are rules that place additional constraints on triggers (such as 8 triggers max in 80 μs). All data from all events must be buffered until the Leve1 1 Trigger has made its decision.

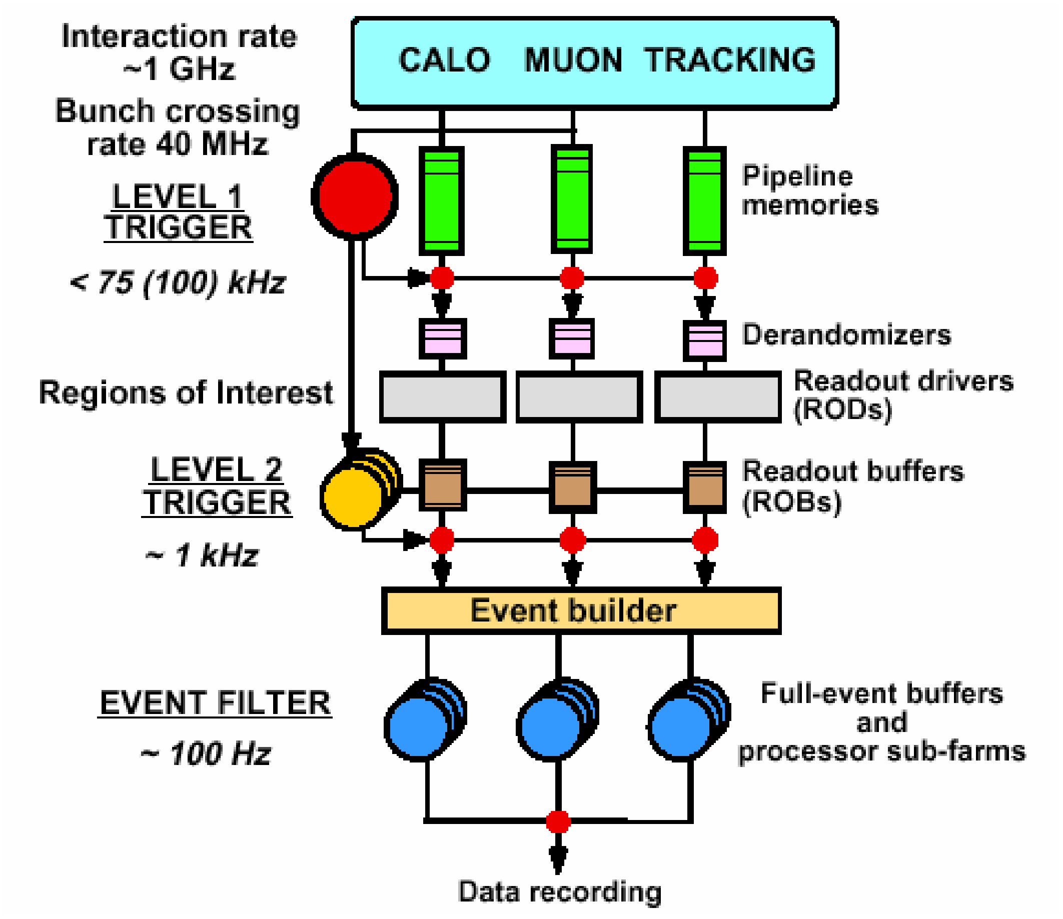


Figure 12: Trigger chain and data flow in ATLAS [2]

After the Level 1 Trigger passes an event, it is read out by the Readout drivers (RODs). These are custom boards designed for each detector that communicate with the detector to generate a digital fragment for each event. This event data is sent to large Readout buffers (ROBs) that hold the data for the Level 2 Trigger. This trigger has access to all the data but is also given Regions of Interest (ROIs) from the Level 1 Trigger to help narrow the search. After passing the Level 2 Trigger, the event data is gathered from all detectors and sent to PCs for a final decision before being written to permanent storage.

The ATLAS CSC ROD and GPUs

The CSC ROD is a fairly generic 9U VME board that can accept high-rate input and perform processing on the data in parallel with up to 12 DSPs. The output then goes through a programmable Data Exchange for final event building and output. A custom back-of-crate card allows changing of the input and output media types without altering the ROD board.

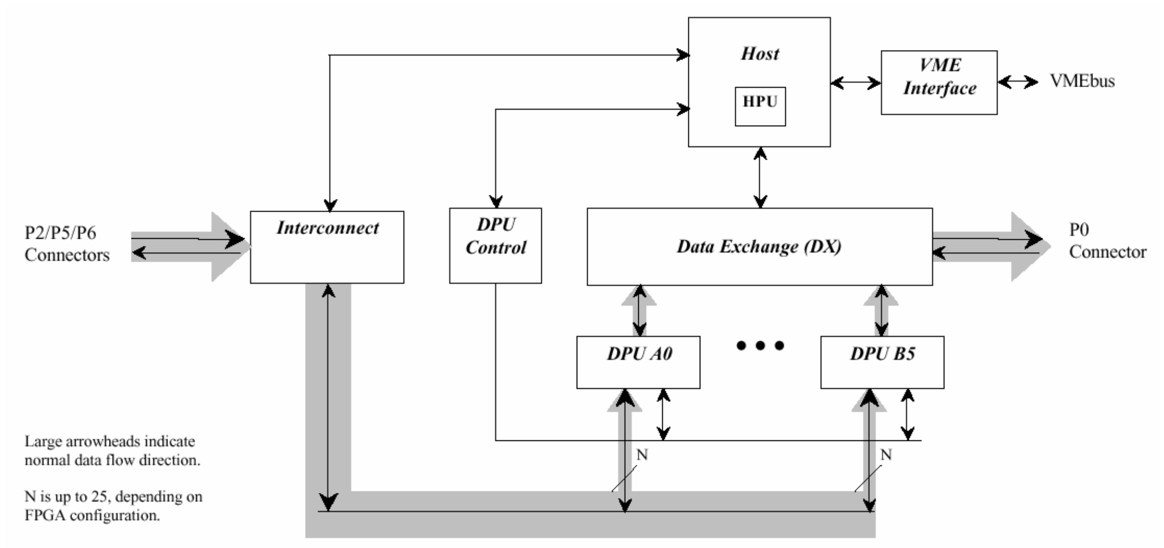


Figure 13: CSC ROD interconnection diagram [33]

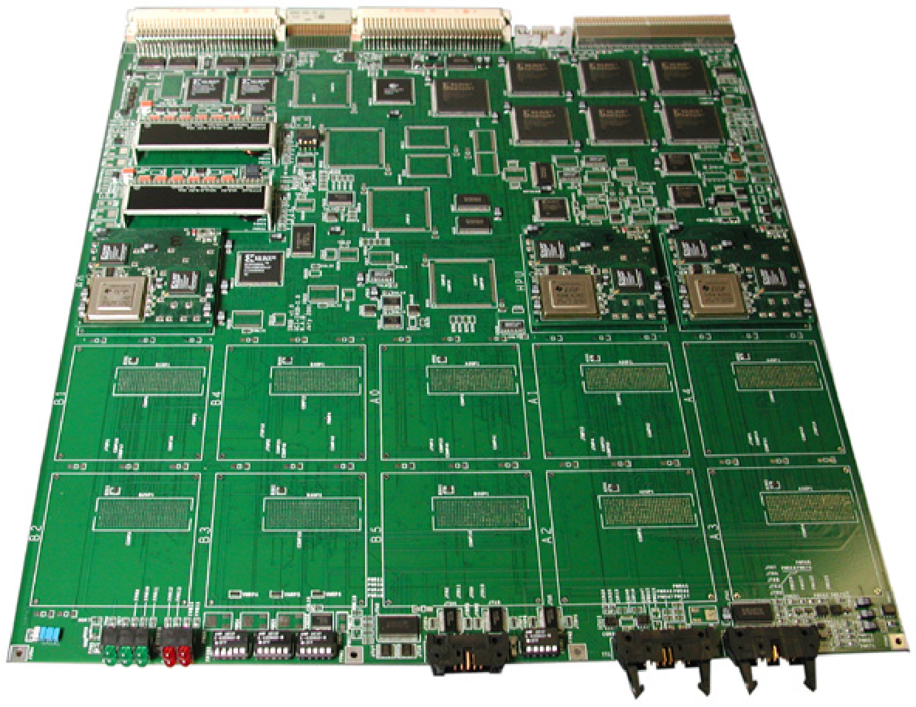


Figure 14: Photo of partially-complete CSC ROD [42]

GPU, HPU, DPU, SPU, RPU

The DSPs on the CSC ROD are on daughterboards called GPU (Generic Processing Unit) modules. The twelve GPUs dedicated to data processing are called DPUs (Data Processing Units), while a thirteenth that oversees the entire ROD board is called the HPU (Host Processing Unit). In the CSC ROD, one DPU will be assigned to each ASM-II board on two chambers for a total of ten SPUs (Sparsifier Processing Units). The remaining two RPUs (ROD Processing Units) will each receive an entire chamber's worth of sparsified data for final neutron rejection and output.

GPU	Generic Processing Unit	x13
HPU	Host Processing Unit	x1
DPU	Data Processing Unit	x12
SPU	Sparsifier Processing Unit	x10
RPU	ROD Processing Unit	x2

Table 2: Table of GPU types [33]

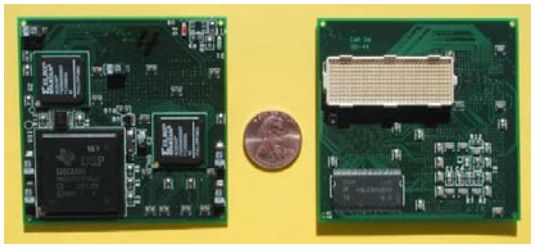


Figure 15: Photo of GPU daughterboard [42]

Each DPU Module can communicate with the Interconnect via the Expansion Bus FPGA (XFPGA) and with the Data Exchange via the EMIF FPGA (EFPGA). The HPU can perform random access on the DPUs' 512kB internal memory via DPU Control and the DPUs have an additional 8MB of external SDRAM.

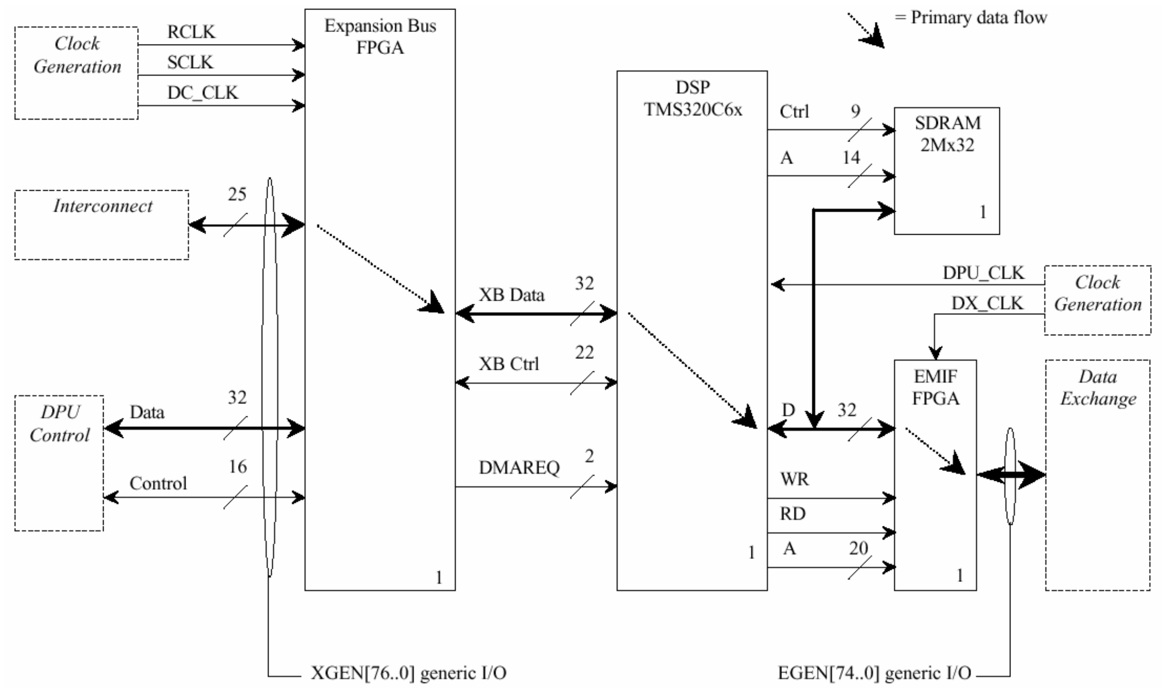


Figure 16: GPU block diagram [33]

Design

SPU vs. RPU

Two gigabit fiber optic cable's worth of data is the limit that one DPU can handle from the XFPGA. Therefore, five DPUs called SPUs are the first to receive data from the five ASMs. Their job is to make a significant reduction in the data by eliminating channels without a hit or with a hit that is from a different beam crossing. This is called sparsification, and it consists of a threshold cut and two time cuts (one coarse, one fine). The SPUs are also responsible for formatting the data, identifying clusters, and generating summary information about them.

The SPUs do not have data from the entire chamber, so there is no way to perform neutron rejection at this level. Therefore, the output from the five SPUs is directed to another DPU called the RPU. The RPU looks at the summarizing information about the clusters and keeps only those clusters that have a corresponding hit in at least one other layer. There is also some final data formatting before output.

The output of two RPUs is combined by the Data Exchange into the ATLAS standard format and sent to the ROB. One ROD therefore handles two chambers.

The SPUs and RPUs have no direct communications with each other. The data is transferred through the Data Exchange via their corresponding EFPGAs.

More information on the SPU and RPU is available in the SPU and RPU sections below.

Events and data flow in the ROD

Events

The input to the SPU is timeslices. Each timeslice contains 96 words of ADC samples and an 8-word trailer. The exact number of timeslices per Event can vary during testing and calibration, but it will be 4 during normal running. However, because some timeslices are shared between Events, the number of timeslices that are removed from input during processing can vary.

The output of the SPU and input to the RPU is called the Sparsified Data Format. This format is variable length but is more easily handled if it is padded to a multiple of some power of two (16 in the current software). Each SPU will output one of these variable-length units per Event, but the RPU will read in five of them per Event (one from each SPU). The final output of the RPU will also be variable-length, but will not be padded because the ATLAS format does not use any special alignment.

To allow for a single DPU framework, we need a generalized Event format that is flexible enough to support all these modes but restricted enough to allow efficient, simple code to be used.

An Event is a fixed number of Packets. This number, *EventLength*, is selectable when setting up for a run. The Packets can be fixed or variable length, with at most *MaxPacketLength* Frames per Packet. If the Event is fixed-length, then each Packet will contain exactly one Frame of *FrameSize* words. If the Event is variable-length, then the first word of the first Frame will be a size word that says how many words are in the Packet (including the size word but excluding padding in the last Frame).

This format is generic enough to handle all the cases needed by the SPU and RPU as well as many unforeseen formats that have come up during testing. The format is completely independent of the data being transferred, which maintains a clear separation between the buffer management code and the Decoder-specific code that processes the data.

	EventLength	**MaxPacketLength**	**FrameSize**
SPU Input	4 (typically)	1 (fixed-length)	96+8
SPU Output	1	38 (typically)	16
RPU Input	5	38 (typically)	16
RPU Output[*]	1	26 (typically)	16

Table 3: Parameters used for various input and output modes (see text for description)

[*] The RPU will not send the size word or padding in its output.

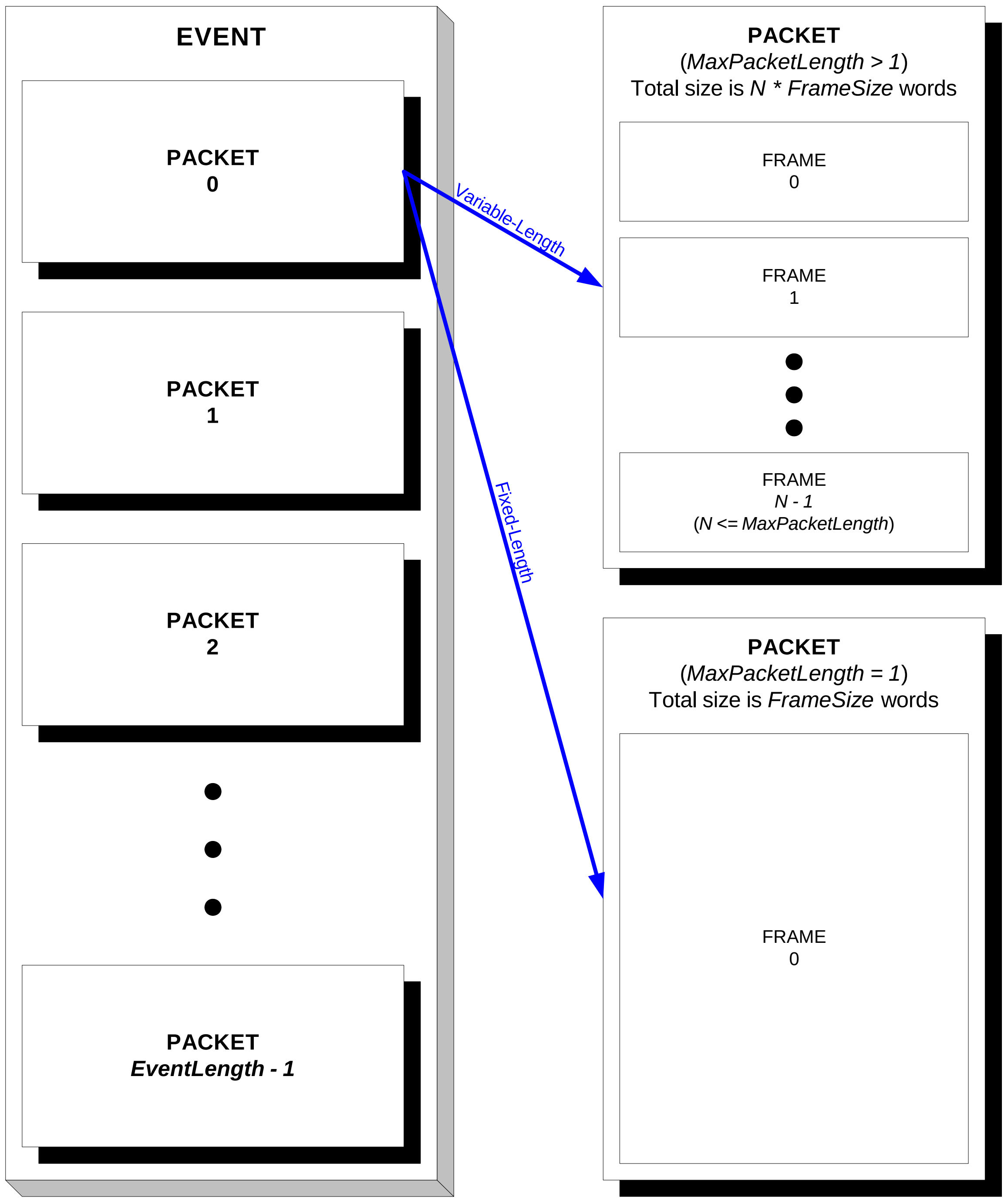

EVENT
PACKET 0
PACKET 1
PACKET 2
PACKET EventLength - 1
Variable-Length
Fixed-Length
PACKET (MaxPacketLength > 1) Total size is N * FrameSize words
FRAME 0
FRAME 1
FRAME N - 1 (N <= MaxPacketLength)
PACKET (MaxPacketLength = 1) Total size is FrameSize words
FRAME 0

Figure 17: Generic Event format

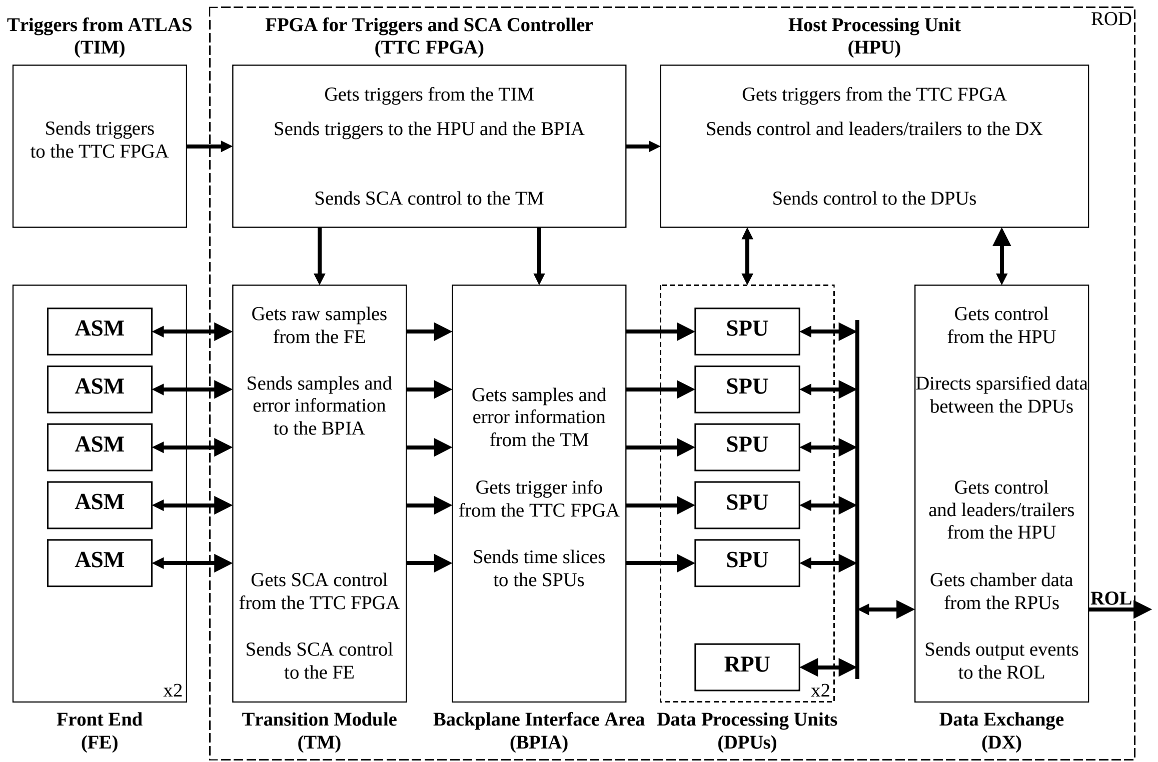
Queues

Events come into the DPU Input Buffer (DIB) and leave through the DPU Output Buffer (DOB). Both buffers use the same Event format, allowing for a variety of configurations of input and output. The classes for the DIB and DOB also support different sources and destinations such as external memory buffers and different FPGAs.

Data flow

The overall flow of data starts at the front end where the chambers are directed to output their ADC samples by the SCA controller on the ROD. These ADC samples are reordered as needed and are sent to the SPUs along with a trailer containing trigger information. The SPUs perform their sparsification and send the result to the RPUs via the Data Exchange. The RPUs then perform their processing and send the result out to the Read Out Link and eventually to the ROB.

The readout is initiated by a trigger that is received by the TTC FPGA. The HPU orchestrates everything and is responsible for keeping data flowing through the ROD.



ROD
Triggers from ATLAS (TIM)
Sends triggers to the TTC FPGA
FPGA for Triggers and SCA Controller (TTC FPGA)
Gets triggers from the TIM
Sends triggers to the HPU and the BPIA
Sends SCA control to the TM
Host Processing Unit (HPU)
Gets triggers from the TTC FPGA
Sends control and leaders/trailers to the DX
Sends control to the DPUs
ASM
ASM
ASM
ASM
ASM
x2
Front End (FE)
Gets raw samples from the FE
Sends samples and error information to the BPIA
Gets SCA control from the TTC FPGA
Sends SCA control to the FE
Transition Module (TM)
Gets samples and error information from the TM
Gets trigger info from the TTC FPGA
Sends time slices to the SPUs
Backplane Interface Area (BPIA)
SPU
SPU
SPU
SPU
SPU
RPU
x2
Data Processing Units (DPUs)
Gets control from the HPU
Directs sparsified data between the DPUs
Gets control and leaders/trailers from the HPU
Gets chamber data from the RPUs
Sends output events to the ROL
Data Exchange (DX)
ROL

Figure 18: Data flow in the ROD

Requirements

There are several requirements of the software.

- **Real-time performance**
  Specifically, there is a hard requirement that the average performance stay under 3000 DSP clock cycles per Event processed (necessary for 100kHz trigger rate) and that the maximum time to perform any task stay under 10,000 DSP clock cycles (necessary to prevent input buffer overflow).

- **Error-free operation**
  It is not acceptable for the DPUs to output erroneous data under any circumstances. Any alternative, including discarding or generating a Fault condition, is preferable.

- **Continuous running**
  While no system can make a 100% guarantee of perfect operation, the DPUs must have minimal failures to maximize data taking. The CSC ROD is one part among many in ATLAS, and the downtime of the entire detector is a combination of the downtimes of the individual parts.

- **Online monitoring**
  Some way must exist to allow the HPU and outside world to monitor the status of the DPUs and make run time changes. In particular, some method for accessing the output of histograms or performing Event capture is needed. There should also be a way to access run performance indicators and halt a run if an error is detected.

- **Handle unusual Events without choking**
  Naturally the system cannot handle any arbitrary input that might exceed the specifications (such as a million Events in a row with every channel hit). However, the system should be able to accept such anomalous Events on an occasional basis. It should also be able to handle any single Event that is theoretically possible without encountering an untested boundary case that causes a crash.

More concisely:

1. *"The DPU software shall operate in such a manner as to allow processing of triggers at the 100kHz rate with no discarding expected under normal operation."*

2. *"The DPU software shall attempt to deal with all situations without data loss except those identified as unavoidable faults."*

3. *"The DPU software shall respond to control initiated by the HPU, and shall relay DPU status information to the HPU on a regular basis."*

The HPU-DPU relationship

In order to meet requirement 3 above, it is necessary to have good communications between the HPU and DPUs. Because the HPU has a total of 12 DPUs to communicate with, it's essential that this communication not be CPU-intensive for the HPU.

DPU Control provides the mechanism for communication. With it, the HPU is able to perform arbitrary accesses into DPU memory via the XFPGA. This lends itself immediately to the use of shared memory for communication. The first level of communication, called the Status struct, does just this. The Status struct is used to initialize the DPU, relay status information to the HPU, and accept high-priority Orders from the HPU (used only to override the normal operation of the DPU during testing or in the event of an error). The Status struct also contains pointers so that the HPU can locate the other buffers used for communication.

A shared memory struct is acceptable for asynchronous communication, but it is not an effective way to serialize commands. This serialization is important to ensure that requests for histogramming or Event capture are received before the corresponding Event has been processed and removed. To accommodate this, the main Command stream is received by the Command Input Buffer (CIB). The CIB is a circular buffer that receives Commands to process an Event or perform any other task in the system. The CIB uses the same class as the DIB, and Commands conform to the generic Event format.

Because the output of both the SPU and RPU is facilitated by the Data Exchange, and the Data Exchange is driven by a command stream from the HPU, it is necessary that the HPU know when Events are ready for output and how big they are. This information, as well as the response to any other Command, is returned through a circular buffer called the Response Buffer. This allows the HPU to read the details of several Events at once rather than be forced to handshake each Response one by one.

The DPU software framework

The DPU software framework has 6 main systems:

- **Management System**
- **Data System**
- **Command System**
- **Scheduling System**
- **Hardware Abstraction Layer (HAL)**
- **Drivers**

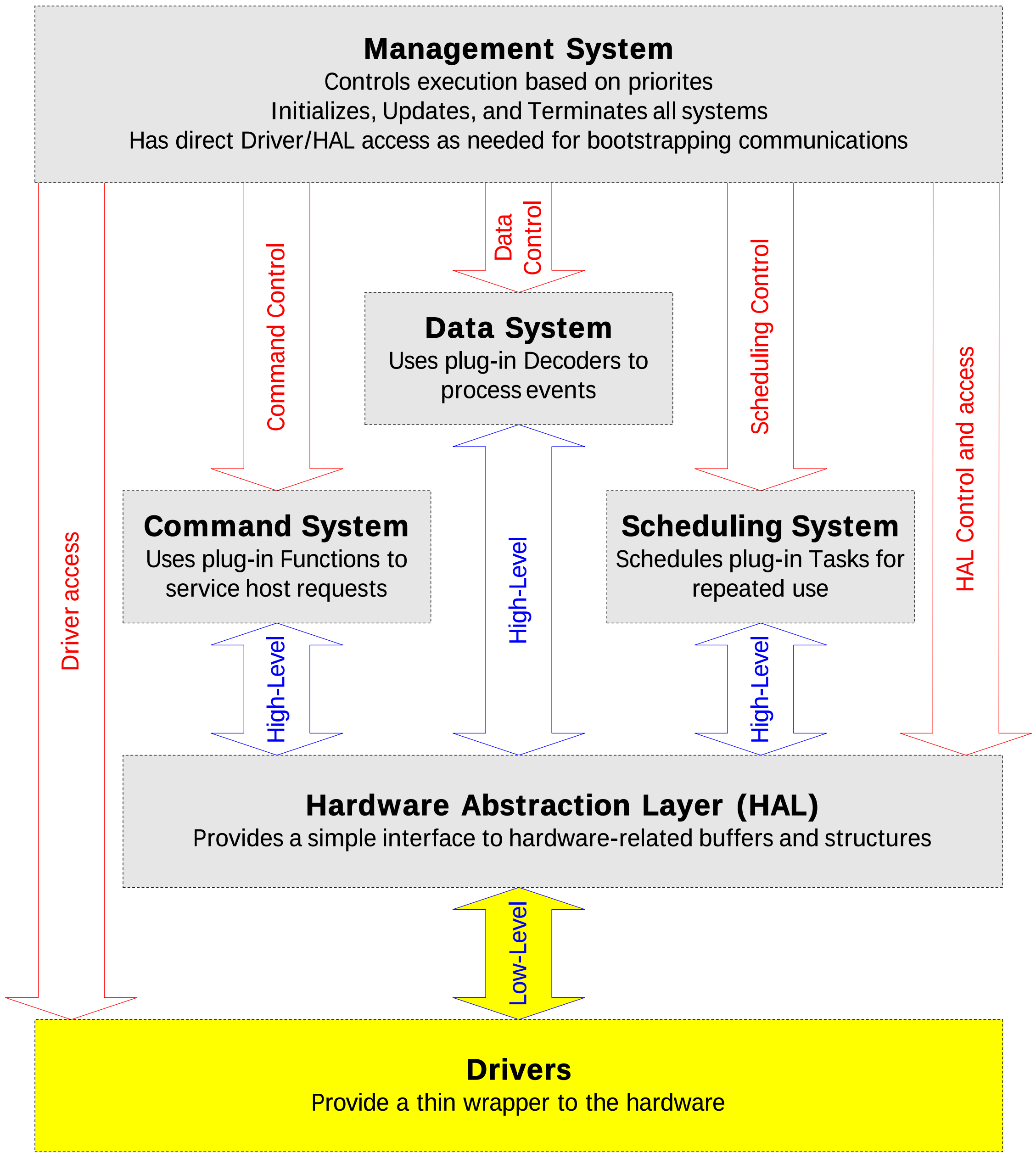

Management System
Controls execution based on priorites
Initializes, Updates, and Terminates all systems
Has direct Driver/HAL access as needed for bootstrapping communications
Command Control
Data Control
Scheduling Control
HAL Control and access
Driver access
Data System
Uses plug-in Decoders to process events
Command System
Uses plug-in Functions to service host requests
Scheduling System
Schedules plug-in Tasks for repeated use
High-Level
High-Level
High-Level
Hardware Abstraction Layer (HAL)
Provides a simple interface to hardware-related buffers and structures
Low-Level
Drivers
Provide a thin wrapper to the hardware

Figure 19: High-level block diagram of DPU software framework

Management System

The Management System looks at the current state of the DPU and decides what should be done next. It is also responsible for *Initialize()*, *Terminate()*, and the regular *Update()* that occurs between all other activities.

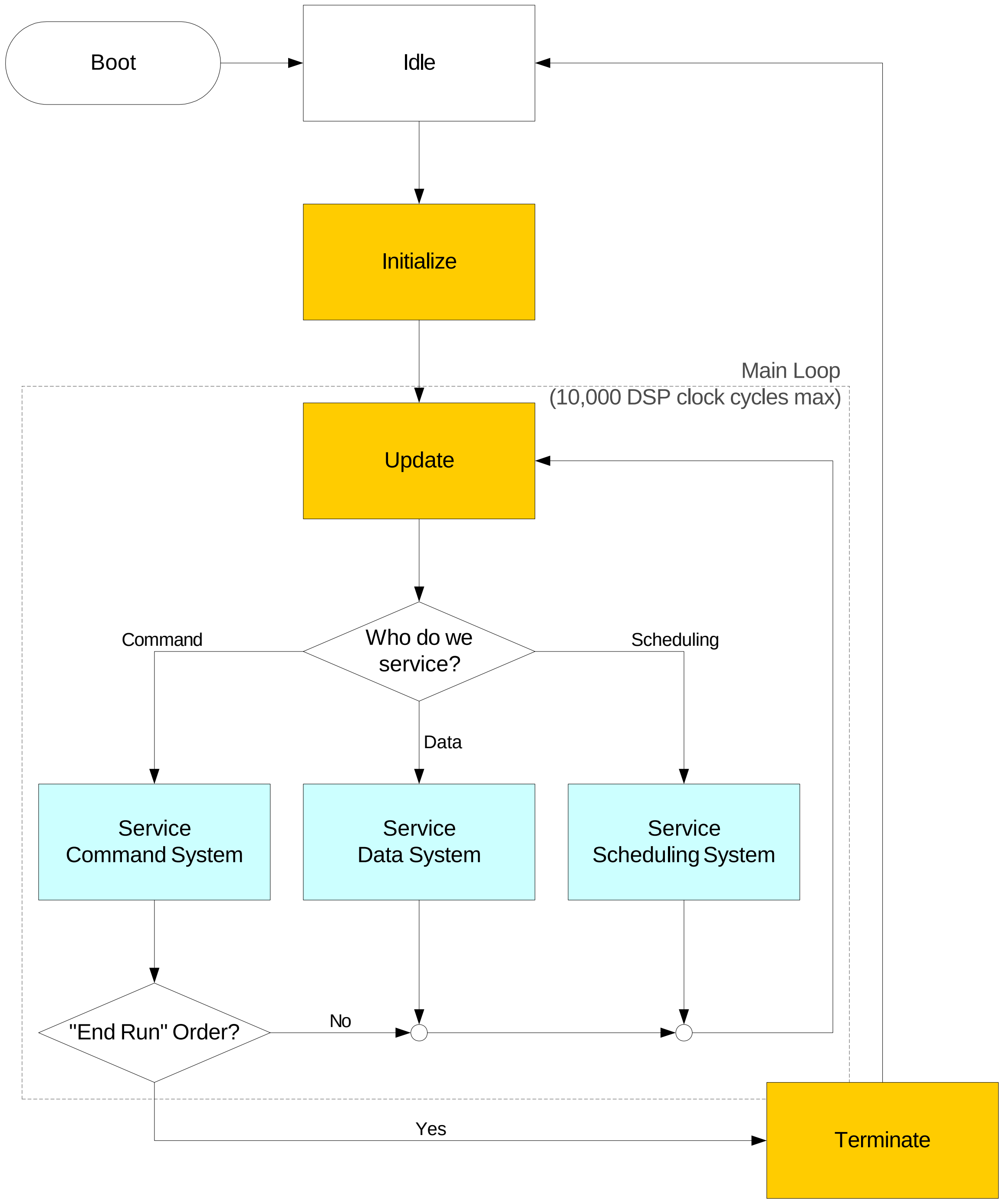


Figure 20: Management System flowchart

Data System

The Data System is responsible for the actual processing of Events. Compile-time plug-ins called Decoders are used to perform this processing.

Each Decoder provides a common interface for *Initialize()* and *Terminate()* as well as a *CanInitialize()* function that tells the Data System whether the Decoder is usable in the current mode of operation. The function *Execute()* handles the processing of one Event.

If for some reason the DIB is filling beyond our ability to drain it, each operational mode also selects a Discard Decoder. This is a special plug-in that is invoked to quickly remove Events without full processing. A special output is generated to let the next stage of processing know that a discard was performed. Discarding is not expected to be needed in normal operation, but it is preferable to a Fault or crash during commissioning. Discarding is automatic and takes priority over the Command stream.

Command System

The Command System handles the incoming Command stream and reports to the Management System what the next Command will be. Generally these are Decode Commands which instruct the DPU to process another Event, but there are also Function Commands that invoke a compile-time plug-in called a Function. These can be used to do anything, including starting, stopping, or requesting output from a Task in the Scheduling System.

Orders are also considered part of the Command System. These are the highest-priority signals the HPU can send and will be executed before all other actions. Sending an Order to a DPU violates normal operation rules and is only used during testing or to end a run.

Scheduling System

The Scheduling System takes care of executing compile-time plug-ins called Tasks. These are started by the HPU and are automatically executed at pre-determined times. They can be set to activate at a given wall clock time or on a given Event number.

Task execution must still compete against Event processing and other Command servicing for time in the Management System, but the Scheduling System decides which Task will be done during the next opportunity. Tasks can request either capture or servicing: capture is scheduled for a specific time or Event number, while service can occur at any time up to a maximum time or Event number. The idea is for Tasks to do only quick operations during their time-critical capture and defer any longer calculations for a service slot that is easier to schedule.

In the event that the Scheduling System falls behind, the entire list of Tasks must be cleared and restarted. The Scheduling System is of lower priority than Event processing and is the first to be sacrificed if things are running behind.

Priorities

The decision of the Management System is based on a color-coded priority assigned to each of the three high-level Systems beneath it. The basic meaning of each priority level is consistent between these Systems.

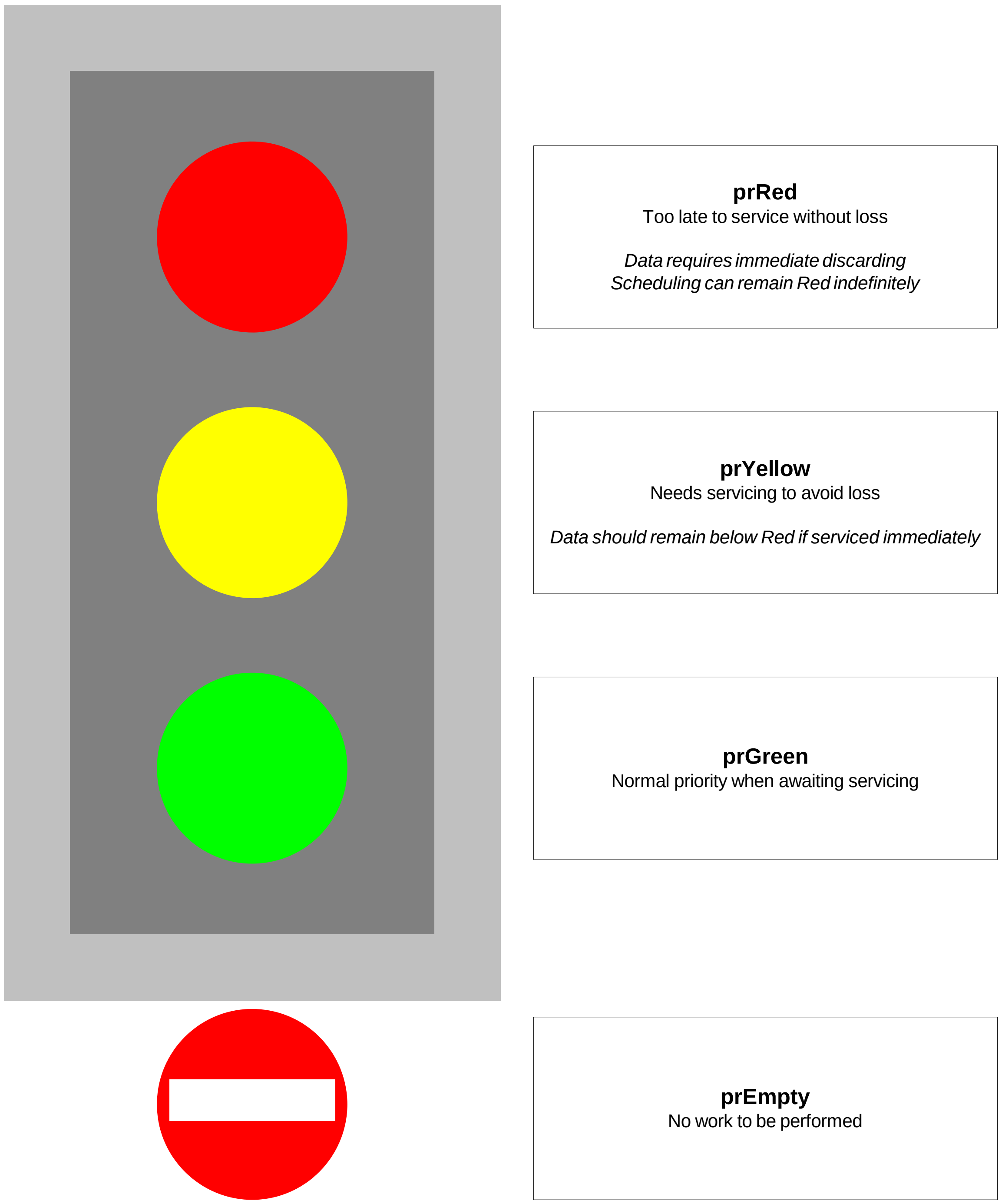

prRed
Too late to service without loss
Data requires immediate discarding
Scheduling can remain Red indefinitely
prYellow
Needs servicing to avoid loss
Data should remain below Red if serviced immediately
prGreen
Normal priority when awaiting servicing
prEmpty
No work to be performed

Figure 21: Color-coded Priority levels

Orders are given the top priority and are serviced by special Functions. The following table summarizes who is serviced or invoked for all other situations:

		Scheduling			
		nothing to do	*service later*	*capture or service now*	*(abort)*
Data	**Next Command**	*prEmpty*	*prGreen*	*prYellow*	*prRed*
prEmpty	*None*	Wait	Scheduling	Scheduling	Scheduling
prEmpty	*Decoder*	Wait	Scheduling	Scheduling	Scheduling
prEmpty	*Function*	Function	Scheduling	Scheduling	Scheduling
prGreen	*None*	Wait	Scheduling	Scheduling	Scheduling
prGreen	*Decoder*	Decoder	Scheduling	Scheduling	Scheduling
prGreen	*Function*	Function	Scheduling	Scheduling	Scheduling
prYellow	*None*	Wait	Wait	Wait	Wait
prYellow	*Decoder*	Decoder	Decoder	Decoder	Decoder
prYellow	*Function*	Function	Function	Function	Function
prRed	*None*	Discard	Discard	Discard	Discard
prRed	*Decoder*	Discard	Discard	Discard	Discard
prRed	*Function*	Discard	Discard	Discard	Discard

Table 4: Management System Priority decision table

Normal rates will not increase Data System Priority if Decoders are constantly serviced. This follows from basic performance requirements. The HPU is responsible for not sending too many Function Commands in between Decode Commands. How this is done depends on the exact performance of the Decoder and the Function Commands being used, but it will generally be specified as a limit on the minimum number of Decode Commands between Function Commands.

The HPU must also ensure that Decode Commands arrive before the Data System goes Yellow. This places a limit on the latency of the HPU in trigger processing.

Hardware Abstraction Layer (HAL)

The HAL provides high-level interfaces to low-level functionality.

- **HAL Control**
  HAL Control is the interface used by the Management System to Initialize, Update, and Terminate the HAL. The HAL is responsible for the underlying drivers.

- **Status Structure**
  The Status struct provides general communications with the HPU using shared memory. The struct is publicly available to the entire framework.

- **Command Input Buffer (CIB)**
  The CIB receives serial Commands from the HPU. All major actions (besides discarding and Orders) taken by the DPU are in response to a Command.

- **Response Buffer**
  The Response Buffer holds outgoing serial Responses to Commands. The Response to a Decode Command (a command to initiate processing an Event) is size information for the resulting Event to be output.

- **DPU Input Buffer (DIB):**
  The DIB provides a simple interface to Decoders (routines responsible for processing an Event) to access an input Event. All buffer management for input is handled by the DIB.

- **DPU Output Buffer (DOB):**
  The DOB provides a simple interface to Decoders (routines responsible for processing an Event) to generate an output Event. All buffer management for output is handled by the DOB.

- **Policy Subsystem**
  The Policy Subsystem maintains a simple database of Event information (*EventLength*, *MaxPacketLength*, *FrameSize*, etc) for different data types. These values can be loaded at run time by the HPU or can be overridden to create a new operational mode.

- **Parameter Subsystem**
  The Parameter Subsystem allows the HPU to pre-load values to be used as the parameters for Commands. The use of these Parameter sets is transparent on the DPU side and reduces the time needed to transfer commonly used sets.

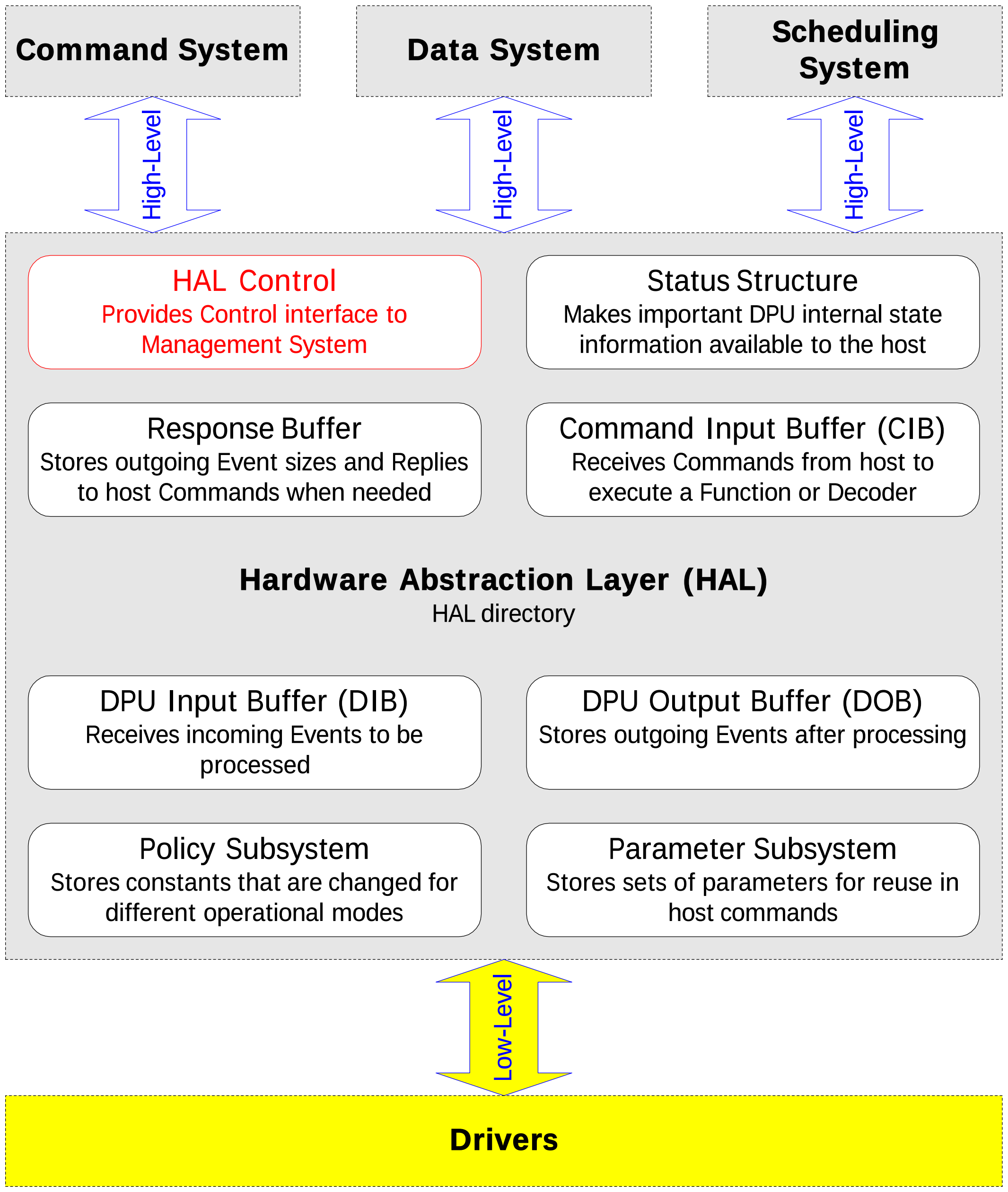

Command System
Data System
Scheduling System
High-Level
High-Level
High-Level
HAL Control
Provides Control interface to Management System
Status Structure
Makes important DPU internal state information available to the host
Response Buffer
Stores outgoing Event sizes and Replies to host Commands when needed
Command Input Buffer (CIB)
Receives Commands from host to execute a Function or Decoder
Hardware Abstraction Layer (HAL)
HAL directory
DPU Input Buffer (DIB)
Receives incoming Events to be processed
DPU Output Buffer (DOB)
Stores outgoing Events after processing
Policy Subsystem
Stores constants that are changed for different operational modes
Parameter Subsystem
Stores sets of parameters for reuse in host commands
Low-Level
Drivers

Figure 22: HAL System diagram

Drivers

The Drivers provide a thin wrapper to four major types of hardware:

- DMA
- FPGAs
- Timers
- Memory

Drivers are primarily used by the HAL to implement its functionality.

Policies and design rules for the software

To meet the requirements of the DPU software, there are a number of rules that must be followed.

1. Expensive operations (such as non-power-of-two divides and dynamic memory allocation) can never be used during a run.

2. Anything that can be done before or after a run should be.

3. The most critical part of the program is the part that is executed for every action (i.e. *Update()* and part of the Management System). This is called the Main Line, and everything should be kept off the Main Line if possible.

4. Maximum time matters as much as, if not more than, average time. You cannot have open-ended loops that execute an unknown number of times during a run, and you should not execute anything more than once per pass on the Main Line.

5. Many memory accesses are *volatile* by nature, but the *volatile* keyword carries double meaning in C++. It implies that the access itself cannot be optimized away, and it prevents the optimizer from reordering across that line. This can prevent proper pipelined loops when copying to/from *volatile* memory regions. Such loops must use the separately-compiled functions such as *FastCopy<>()* and *FastCopyStride<>()* to get full optimization.

6. In spite of the need for optimization, working code is more important than fast code. All code should be as portable and standards-conforming as possible, and should follow the rules of proper encapsulation and data hiding.

SPU

Sparsification algorithm

The SPUs receive 192 channels of data with four 12-bit ADC samples per channel. At a trigger rate of 100 kHz this means 166 MB/s of data. Each RPU would therefore receive over 800 MB/s, which is a phenomenal amount of data. While the CSC are high-rate chambers, the occupancy is quite low; this means that only a few channels contain meaningful information and the rest are noise and leftovers from other beam crossings.

When a particle passes through the chamber and generates a signal, a cluster is formed. This cluster is a group of neighboring channels that show a response to the particle. Each channel in the cluster is a hit, and ideally we only want to keep the hits from the clusters that occurred in a chosen time window for the trigger. A threshold cut is used to eliminate channels that are not hit, and a time cut is used to eliminate clusters that are out-of-time. For efficiency reasons the first time cut is made on channels that are outside a rough 75 ns window, followed by a finer cut on clusters to a programmable window size.

The preamp shapers in the CSC readout electronics produce a 7th order complex bipolar Gaussian [19]. This looks roughly like a parabola on a leading positive lobe followed by a smaller negative lobe with a longer tail. The four ADC samples are spaced 50 ns apart, giving a 200 ns range that covers the positive lobe. Because the triggers come on a 25 ns beam crossing clock, there are two different trigger/sampling phases. To simply things, the sampling is adjusted so that the nominal peaking time for in-time hits falls halfway between the B (second) sample of the later sampling and the C (third) sample of the earlier sampling. The case where the B sample is nearer the nominal peaking time is called phase B, and the case where C is nearer is called phase C.

Thanks to this symmetric sampling, there is a simple algorithm for applying a threshold and rough 75 ns time cut on channels. The requirement is that the nominally largest sample (B for phase B, C for phase C) be larger than a threshold, sample A, and sample D. This test is not sensitive to the exact shape of the positive lobe, and is accurate in the approximation that the positive lobe is symmetric about the peak.

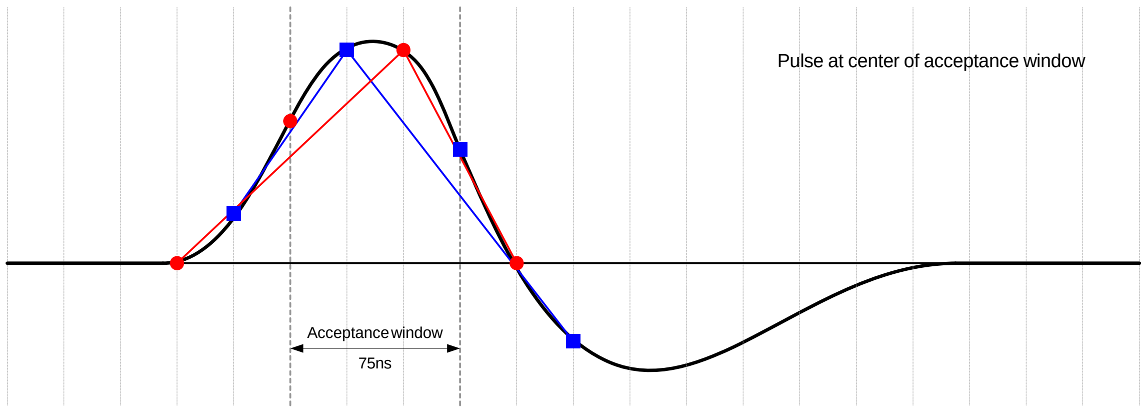


Figure 23: In-time pulse (lines every 25 ns, squares are phase B samples, circles are phase C)

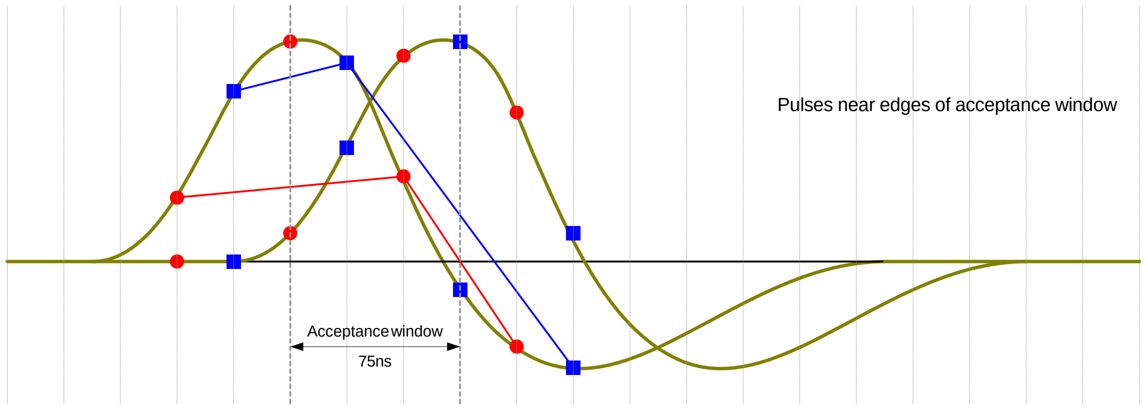


Figure 24: In-time pulses (lines every 25 ns, squares are phase B samples, circles are phase C)

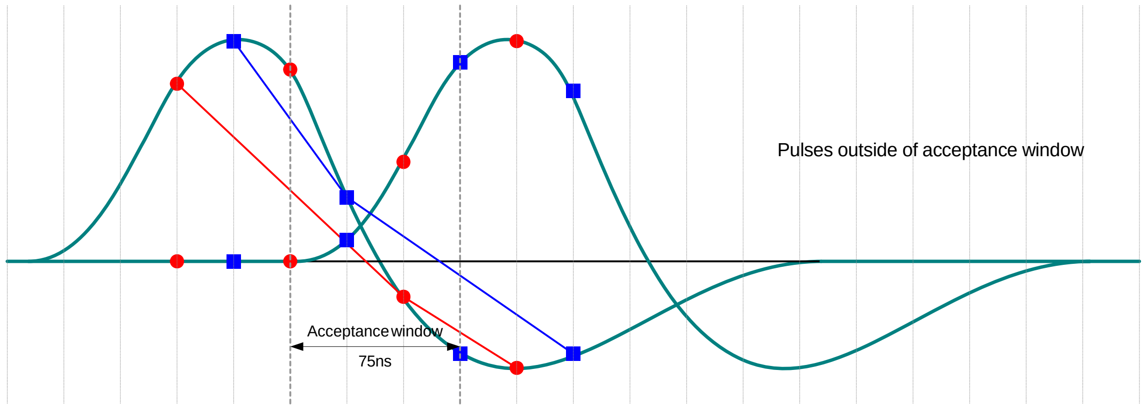


Figure 25: Out-of-time pulses (lines every 25 ns, squares are phase B samples, circles are phase C)

Average flux with standard safety factor of 5 (muon) [3]:	1500 Hz/cm^2
Average flux with standard safety factor of 5 (background) [24]:	1500 Hz/cm^2
Average flux (muon + background):	3000 Hz/cm^2
Precision strip area [3]:	25 cm^2
Transverse strip area [3]:	100 cm^2
Precision cluster probability per beam crossing (25ns):	0.19%
Transverse cluster probability per beam crossing (25 ns):	0.75%
Precision cluster probability in a 75 ns rough time cut:	0.56%
Transverse cluster probability in a 75 ns rough time cut:	2.25%
Typical precision cluster width (number of channels above threshold):	5
Typical transverse cluster width (number of channels above threshold):	2
Precision channel probability in a 75ns rough time cut:	2.8%
Transverse channel probability in a 75ns rough time cut:	4.5%
Average number of precision clusters per SPU after threshold/75ns cut:	1.1
Average number of transverse clusters per SPU after threshold/75ns cut:	4.3
Average number of precision channels per SPU after threshold/75ns cut:	5.4
Average number of transverse channels per SPU after threshold/75ns cut:	8.7

Table 5: CSC rate numerology

After this first cut there are only a few clusters per layer. The next step is to identify the clusters by finding groups of contiguous hit channels. This may cause us to see two overlapping clusters as a single cluster, but this can be dealt with offline. Before we make this cluster identification, it is important to consider the effect of bad channels. A channel that is dead may split a cluster into two clusters, and a channel that is hot or noisy could produce spurious clusters. To address this, we adopt a policy that bad channels cannot be the seed for a cluster, but should be treated as a hit if they are next to a channel that is hit. This will bridge the gap in a broken cluster without producing extra clusters that cannot be relied upon.

This is also the time to mark additional neighbors to be kept. The software offers the option of marking one or more channels next to each cluster to be treated as though they were hit. This option allows for more detailed charge centroid calculations for finding the position by including the below-threshold neighbors. Recent studies [43] suggest using one precision neighbor as the threshold can be set quite low, but more can also be used.

Once bad channels are dealt with and neighbors marked, we can find the clusters and compute a more precise time estimate for the next cut. To get a time estimate, we can treat the positive lobe as a parabola and perform inverse parabolic interpolation using samples A/B/C or samples B/C/D.

- Formulae giving peaking times relative to time of center sample: [34]

$$t(ns) = b(ns) - \frac{25(A-C)}{2B-A-C}$$

$$t(ns) = c(ns) - \frac{25(B-D)}{2C-B-D}$$

While these appear to be very different formulae, shifting to a common time reference gives the same numerator with different denominators. As it turns out, in the ideal case of a perfect parabola we expect the discriminants (denominators) to be equal. We will use the larger denominator of the two: this increases the chance of getting a denominator that is suitably positive to give a useful result and errs on the side of keeping the cluster (smaller time).

In additional to a very small, zero, or negative denominator, there are other reasons why a given channel may not be suitable for time estimation. Channels that were below threshold (neighbors) and channels that are marked as bad should never be used for time estimation. Also, any sample that is high or low saturated should not be used. This can mean actual saturation or merely outside the region of linearity. Normally a linear calibration correction should be applied but it will not affect the time we obtain.

For a given cluster, the best time to use is the time of the largest eligible channel in that cluster. Practically speaking, that means the channel with the largest sample B (for phase B) or sample C (for phase C) that can have a time computed. The choice of which channel has the highest sample is affected by a linear calibration correction, so this must be applied to that sample before comparison. This produces the best time for a cluster, but what if the cluster is really two or more overlapping cluster? If an out-of-time cluster were to overlap with a smaller in-time cluster, the out-of-time cluster would have the largest sample and cause the entire cluster pair to be rejected. To avoid this, we also compute the time for each of the sides of the cluster and compare these to the time obtained from the largest channel. If all three times do not fall into a programmable range, the cluster time is left blank and the cluster is considered to pass. Offline software can work further to compute an accurate time.

Once a time is computed, we apply a final time cut on clusters to around 40-50 ns, which is enough to get all in-time muons [36]. This gives a 25-35 ns leeway between the final window and the original 75 ns window, making the output insensitive to even a substantial error in the 75 ns cut. The times produced by inverse parabolic interpolation are accurate to about 1 ns [20].

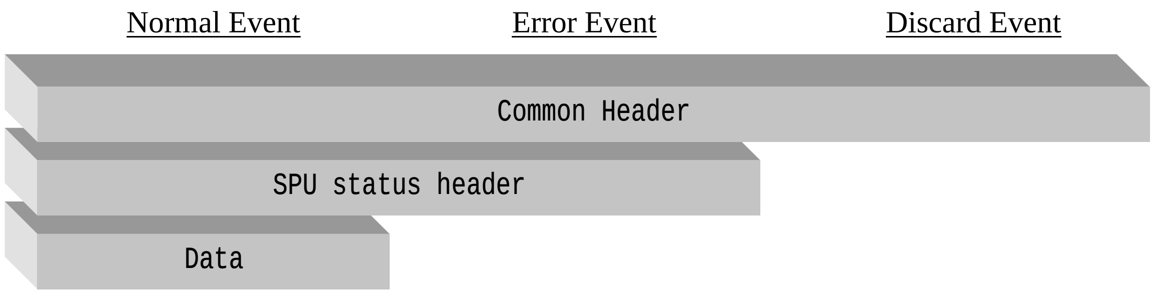
After the last time cut, the SPU must produce a set of cluster bitmaps. These are six words that show which channels are included in the cluster. These, together with a total bitmap for the entire SPU, are used by the RPU for neutron rejection. The bitmaps are compressed by suppressing zero words and sent along with the cluster-based output via the Data Exchange.

Summary of processing

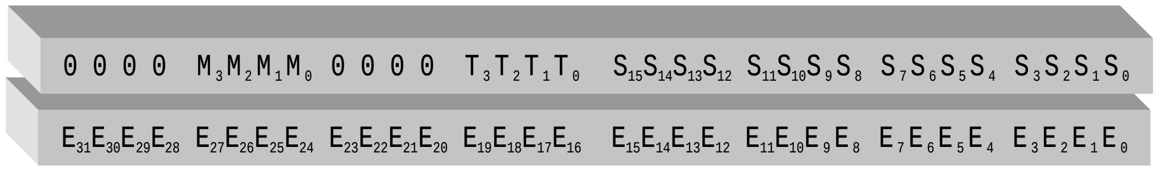
- Sparsification (threshold and 75ns cut)
- Neighbor marking and bad channel masking
- Cluster identification
- Cluster time estimation and time cut
- Bitmap creation
- Output formatting

Sparsified data format

Events

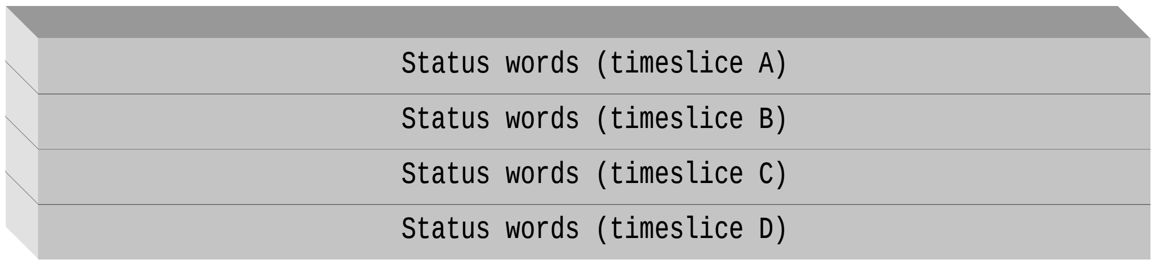


Common header

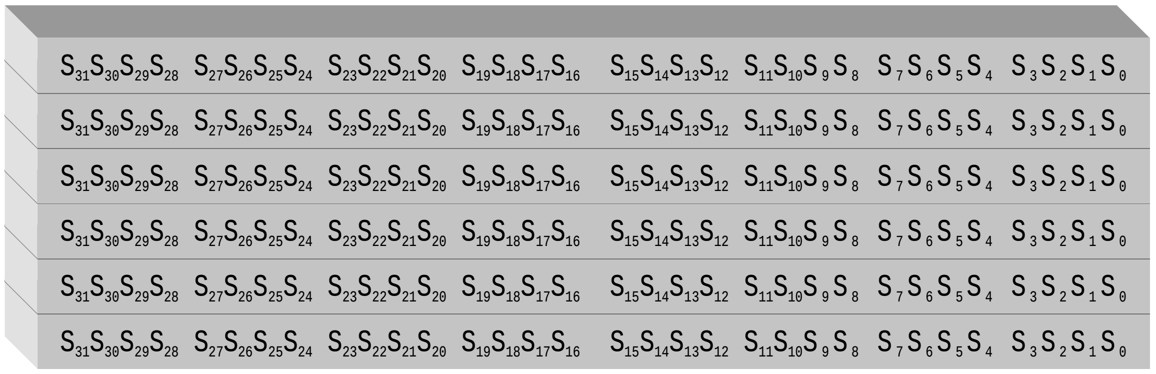

0 0 0 0 $M_3M_2M_1M_0$ 0 0 0 0 $T_3T_2T_1T_0$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$E_{31}E_{30}E_{29}E_{28}$ $E_{27}E_{26}E_{25}E_{24}$ $E_{23}E_{22}E_{21}E_{20}$ $E_{19}E_{18}E_{17}E_{16}$ $E_{15}E_{14}E_{13}E_{12}$ $E_{11}E_{10}E_9E_8$ $E_7E_6E_5E_4$ $E_3E_2E_1E_0$

- M = Module ID
- T = Type of Event (0 = Normal, 1 = Discard, 2 = Error)
- S = Size of Event (in words, including the common header and ghost words)
- E = Event index

SPU status header


Status words (timeslice A)
Status words (timeslice B)
Status words (timeslice C)
Status words (timeslice D)

Status words


$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$
$S_{31}S_{30}S_{29}S_{28}$ $S_{27}S_{26}S_{25}S_{24}$ $S_{23}S_{22}S_{21}S_{20}$ $S_{19}S_{18}S_{17}S_{16}$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$

- S = Status words from BPI

Data

$C_{15}C_{14}C_{13}C_{12}$	$C_{11}C_{10}C_9C_8$	$C_7C_6C_5C_4$	$C_3C_2C_1C_0$	$D_{15}D_{14}D_{13}D_{12}$	$D_{11}D_{10}D_9D_8$	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_9B_8$	$B_7B_6B_5B_4$	$B_3B_2B_1B_0$
Cluster 0							
Cluster 1							
Cluster 2							

•••

- C = Cluster count
- D = Number of data words that follow in clusters
- B = Bitmap of all clusters

Cluster

Cluster words
Bitmap words
Sample words

Cluster words

$0\ A_3A_2A_1$	$A_0L_1L_0D$	$C_7C_6C_5C_4$	$C_3C_2C_1C_0$	$W_{15}W_{14}W_{13}W_{12}$	$W_{11}W_{10}W_9W_8$	$W_7W_6W_5W_4$	$W_3W_2W_1W_0$
0 0 0 0	0 0 0 0	0 0 0 0	$0\ B_2B_1B_0$	0 0 0 F	$T_{11}T_{10}T_9T_8$	$T_7T_6T_5T_4$	$T_3T_2T_1T_0$

- A = Adjustment for channel number
- L/D/C = Layer/Direction/Channel number (D = 0 for precision, 1 for transverse)
- W = Width of cluster (number of sample word pairs)
- B = Bitmap present (each bit activates the corresponding pair of bitmap words)
- F = Failed to compute time (time will be 0)
- T = Time (signed, in nanoseconds)

Bitmap words

$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_{9}B_{8}$	$B_{7}B_{6}B_{5}B_{4}$	$B_{3}B_{2}B_{1}B_{0}$
$B_{31}B_{30}B_{29}B_{28}$	$B_{27}B_{26}B_{25}B_{24}$	$B_{23}B_{22}B_{21}B_{20}$	$B_{19}B_{18}B_{17}B_{16}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_{9}B_{8}$	$B_{7}B_{6}B_{5}B_{4}$	$B_{3}B_{2}B_{1}B_{0}$

- B = Bitmap for this cluster

Sample words

$A_{15}A_{14}A_{13}A_{12}$	$A_{11}A_{10}A_{9}A_{8}$	$A_{7}A_{6}A_{5}A_{4}$	$A_{3}A_{2}A_{1}A_{0}$	$B_{15}B_{14}B_{13}B_{12}$	$B_{11}B_{10}B_{9}B_{8}$	$B_{7}B_{6}B_{5}B_{4}$	$B_{3}B_{2}B_{1}B_{0}$
$C_{15}C_{14}C_{13}C_{12}$	$C_{11}C_{10}C_{9}C_{8}$	$C_{7}C_{6}C_{5}C_{4}$	$C_{3}C_{2}C_{1}C_{0}$	$D_{15}D_{14}D_{13}D_{12}$	$D_{11}D_{10}D_{9}D_{8}$	$D_{7}D_{6}D_{5}D_{4}$	$D_{3}D_{2}D_{1}D_{0}$

- A = sample A
- B = sample B
- C = sample C
- D = sample D

Ghost words

At the end of Events can be added one or more ghost words. The presence of these words can be inferred by comparing the reported total size with the actual size of any data words and known headers. These words are not used in normal running and should be ignored by ATLAS software. The primary use of these words is to aid debugging during commissioning.

There are no current ghost words, since the status words are already sent.

Implementation

The key to efficient performance in the SPU Decoder is the use of parallel assembly for time-critical inner loops. There are 9 assembly language routines used in processing on the SPU:

OOTEliminate assembly function

1. Reads three time slices from the input buffer and the thresholds for each channel.
2. Applies threshold test and 75ns OOT elimination to all channels.
3. Stores a bitmap indicating which channels passed.

Performance: 241 DSP clocks (disables interrupts)

```
   LDW     .D1    *taddr1++,                         thresh1
   LDW     .D1    *baddr1++,                         big1
   LDW     .D1    *laddr1++,                         left1
   LDW     .D1    *raddr1++,                         right1

   NOP
   NOP
   SUB2    .S1    thresh1,         big1,             thresh1
   SUB2    .S1    left1,           big1,             left1

   SUB2    .S1    right1,          big1,             right1
|| AND     .L1    thresh1,         left1,            left1
   AND     .L1    left1,           right1,           right1
   MPY     .M1X   right1,          shift2,           low1
   MPYHL   .M1X   right1,          shift2,           high1

   SMPYH   .M1X   low1,            shift2,           mask1
   MPYH    .M1X   high1,           shift2,           high1
   NOP
   OR      .L1    mask1,           high1,            temp1

   NOP
   SHL     .S1    temp1,           2*(COUNT2%16),    temp1
   OR      .L1    BITMAP1,         temp1,            BITMAP1
```

Figure 26: Side 1 loop kernel for *OOTEliminate()*

This function, like most, relies on a heavily pipelined loop with multiple operations occurring in parallel. This particular loop is a good example of pipelining, as it requires just under five blocks of four DSP clock cycles each in order to be paralleled with itself every four clock cycles. This means that we have a four clock kernel with a 15 clock epilog. Because four channels are processed by the loop, this routine manages to apply both threshold and a rough 75 ns time cut in just one clock cycle per channel.

OOTEliminate() and almost all functions use custom macros *startclock*/*stopclock* to keep track of what runs in parallel with what. Everything with a *clock(N)* macro will occur in clock cycle N between the *startclock* and *stopclock* macros. These and all other custom macros use the m4 macro language [23] and make particular use of diversion streams for storing instructions for the various clock cycles.

This is also one of a few functions that use the custom macros *startloop*/~*stoploop* to generate loops with loop variables. This allows the *N* in the *clock(N)* calls to be specified as a function of the pass through the unrolled loop, as well as to change the shift values *COUNT1*/*COUNT2* and the bitmap words *BITMAP1*/*BITMAP2* during each pass.

MarkNeighbors assembly function

1. Reads the existing bitmap.
2. Marks neighbors for hits as required. Also masks out bad channels.
3. Updates the existing bitmap.

Performance: 20 DSP clocks per neighbor width (interrupt safe)

MarkNeighbors() is a simple function that shifts bitmap words left and right and bitwise ORs them back in. Multiple iterations of this function can mark multiple neighbors, and even one invocation has the effect of bridging gaps caused by removed bad channels. There is also an option to select transverse mode so that neighbors are not marked across layer boundaries (the six word bitmap represents all four transverse layers).

ParseBitmap assembly function

1. Reads four time slices from the input buffer and the existing bitmap.
2. Associates bitmap hits with actual samples from the timeslices.
3. Stores a linked list of samples and a buffer of channel offsets.

Performance: 7 * (# of hits) + 28 DSP clocks (interrupt safe)

ParseBitmap() is a very typical two-pass loop, meaning that the routine parallels with itself halfway through. As with most of the assembly functions, many client-preserved registers (A10-A15, B10-B15, and B3) must be saved to memory before being used.

FindPeak assembly function

1. Reads the existing linked list of samples, the channel offsets, and the calibration constants and saturation values for each channel.
2. Identifies and records clusters along with numerators and denominators for the three channels in each cluster to be used for time computation.
3. Stores a linked list of clusters with numerators and denominators.

Performance: 10 * (# of hits) + 28 DSP clocks (interrupt safe)

This is another two-pass loop that has the added feature of using custom macros *startpass*/~*stoppass* to produce two nearly-identical functions *FindPeakB()* and *FindPeakC()*. The section in the *on_pass()* macro call differs for each pass of the macro and allows the two functions to do slightly different things as needed. In this case, the difference is which sample (B or C) is used to determine the largest channel for time estimation.

CalculateTime assembly function

1. Reads the existing linked list of clusters and the list of time offsets for channels.
2. Calculates a time for each cluster using the largest channel and matches it to the times based on the side channels.
3. Updates the linked list of clusters to include the times with a bit indicating whether the time computation succeeded.

Performance: 14 * (# of clusters) + 34 DSP clocks (interrupt safe)

CalculateTime() is another two-pass loop with multiple versions produced using *startpass*/~*stoppass*. To speed calculation, the central time is computed to a fixed precision of 12 bits by shifting both numerator and denominator before dividing. To avoid doing this expensive operation three times, the side channels are verified to be within a window of the central time by cross-multiplying.

CutTime assembly function

1. Reads the existing linked list of clusters.
2. Applies a cut to reduce the acceptance window to a programmable amount.
3. Updates the linked list of clusters to remove the cut clusters.

Performance: 3 * (# of clusters) + 7 DSP clocks (interrupt safe)

CutTime() is another unrolled function that uses startloop/~stoploop. Unlike *OOTEliminate()*, however, this function does not run for a fixed number of loop iterations. There are enough copies of the loop to handle the largest number of loop iterations that might be needed, and a decreasing counter tells us when to branch back to the return address. This "run out" technique allows a small kernel (less than the 6 clocks it takes to branch and wait for delay slots) to be implemented without dealing with complicated multiple-branching techniques.

AddBitmap assembly function

1. Reads the existing linked list of clusters.
2. Computes cluster bitmap words and determines which pairs of words are non-zero.
3. Updates the linked list of clusters to include the bitmap words and BitmapPresent bits.

Performance: 14 * (# of clusters) + 36 DSP clocks (interrupt safe)

AddBitmap() is another two-pass loop, but this one makes extensive use of lookup tables (LUTs) to compute the bitmaps quickly. LUT techniques are useful for many situations such as bit counting and reversal, but it can be difficult to balance the trade off between speed and LUT memory usage.

LinkList assembly function

1. Reads the existing linked list of clusters and the existing linked list of samples.
2. Splices the linked lists so that cluster words come before the corresponding samples.
3. Updates the linked lists of samples and clusters to form a single list.

Performance: 6 * (# of clusters) + 8 DSP clocks (interrupt safe)

LinkList() is a very straightforward two-pass loop. The power of *LinkList()* is in the hand calculations used to decide which linked list pointers need to be modified and to what value. These formulae are simply churned through and the corresponding values are written. This function also uses a special LUT stored in a single word...shifting by a variable amount allows one of the many small bit patterns to be extracted.

OutputResults assembly function

1. Reads the existing linked list of clusters with samples.
2. Generates the final output by walking the linked list.
3. Stores the final output in the output buffer.

Performance: 3 * (# of hits) + 6 * (# of clusters) + 15 DSP clocks (interrupt safe)

OutputResults() has to perform a task that is normally very inefficient on a deeply pipelined processor: walking a linked list. Because of the four delay slots after each load instruction, it is not possible to walk a standard linked list in less than five clocks per node. To get around that, we use a "pipelined" linked list that effectively stores the pointer to the node after the next. This allows the code to hide the delay slots properly. To further simplify things given the typical use of large contiguous sections of memory, a relative pointer is used that specifies how many bytes to skip in addition to the three words normally advanced by. These skip words are zero when the next node follows immediately and need only be non-zero in the node two before a splice.

This routine uses the more traditional multiple branch technique to get a loop half the size of the usual branch plus delay slots. Because walking such a custom linked list presents the risk of wandering off into random memory on a fatal glitch, the routine aborts after about 500 words are copied.

C++

In addition to the assembly code, it is important that the C++ code connecting these functions be as efficient as possible. Separately-compiled functions like *FastCopy<>()* and *FastCopyStride<>()* are used to avoid problems pipelining loops with *volatile* reads/writes, and every effort is made to get all loops to pipeline when possible.

Calibration array

Status.Basic.Calibration is a pointer to a 1 kW buffer used to store Decoder initialization information. The format of this information for the SPU Decoder is:

- Bitmap of bad channels
  6 words, unsigned
  Any values are allowed

- Precision layer number
  1 word, unsigned
  0-3 if precision, 0 if transverse

- Neighbor count
  Number of times to invoke *MarkNeighbors()* function
  1 word, unsigned
  No more than four times permitted

- Minimum denominator to allow divide for time
  A denominator equal to minimum will fail
  1 word, unsigned
  Must be between 1 and 0x7FFFFFFF, inclusive

- Final time window
  Half-width of the final time cut window in ns, edges fail
  1 word, unsigned

- Cross-check time window
  Determines the half-width of the cross-comparison time window, edges fail
  1 word, unsigned
  See below for format and restrictions on the value

- Large cross-check time window boolean
  Helps determine the half-width of the cross-comparison time window
  1 word, unsigned
  Must be 0 (false) or 1 (true)

- Precision DPU boolean
  Indicates whether the DPU is precision (true) or transverse (false)
  1 word, unsigned
  Must be 0 (false) or 1 (true)

- Threshold array
  Threshold for *OOTEliminate()* test, equal to threshold fails
  192 halfwords, unsigned
  Must be in same order as channels in the timeslice

- Saturation array
  Used to decide if sample is saturated low (low halfword) or high (high halfword)
  Equal to saturation value fails and cannot be used for time computation
  192 words (each word is two packed signed halfwords)
  Must be in same order as channels in the timeslice

- Special zero word
  Needed to follow saturation array with a zero value
  1 word, unsigned
  Must be zero

- Calibration array
  Used for linear calibration of nominally largest sample before comparing
  Pedestal in low halfword, scale factor in high halfword
  192 words (each word is two packed signed halfwords)
  Must be in same order as channels in the timeslice

- Special zero word
  Needed to follow calibration array with a zero value
  1 word, unsigned
  Must be zero

- Time adjustment in phase B array
  Used to adjust computed times by a signed halfword in ns
  192 halfwords, signed
  Must be in same order as channels in the timeslice, used for phase B
  Absolute value must be no more than 1000

- Time adjustment in phase C array
  Used to adjust computed times by a signed halfword in ns
  192 halfwords, signed
  Must be in same order as channels in the timeslice, used for phase C
  Absolute value must be no more than 1000

The cross-check time window (used to compare side channel times with the central cluster time) has a complicated format due to restrictions imposed by the assembly code. The lower halfword of the time window is a shift factor s, while the upper halfword is a multiply factor m. The requirement is that m must be between 0 and 8 inclusive.

When the large cross check time window boolean is true, the window half-width in ns is $64 \cdot (m << s)$. When the large cross check time window boolean is false, the window half-width in ns is $64 \cdot (m >> s)$. The shifting is done this way to keep the values small enough to fit in 32 bit words no matter what shift value is needed.

RPU

Neutron rejection algorithm

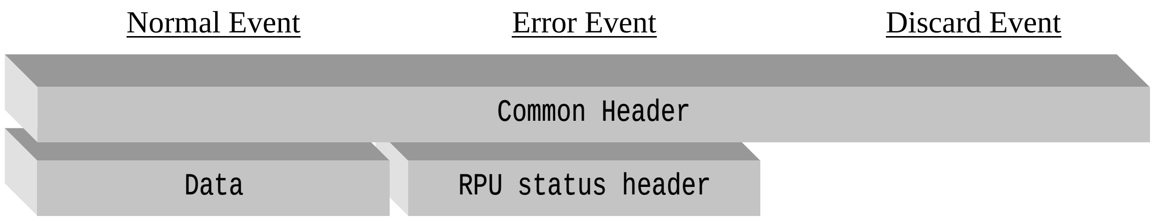
The RPU is the first point at which all four layers of the chamber are available together. This offers the option to reject neutrons and photons by looking for tracks in the chamber. The algorithm for this must be simple and scale well with the number of clusters. The occupancy is still low, but the worst-case performance can blow up severely if the scaling is bad because of the 960 channels of data possible in the RPU.

Neutrons and photons typically only deposit their energy in one layer, while muons pass through all four. Because of this, a cluster that has a corresponding cluster in another layer is more likely to be a muon and less likely to be a neutron or photon. We keep any cluster that has a corresponding cluster in any of the three other layers, and reject only those clusters that have no corresponding cluster in any layer. To allow for track inclination, corresponding clusters in neighboring layers must either overlap or be touching each other at the edges. For example, a cluster in channels 5-12 would match with a cluster in channels 13-19 in the next layer. If the layers are separated by one intervening layer, then there can be a single channel gap between the two clusters. For example, a cluster in channels 5-12 would match with a cluster in channels 14-19. Finally, if there are two intervening layers than a gap of two channels is allowed.

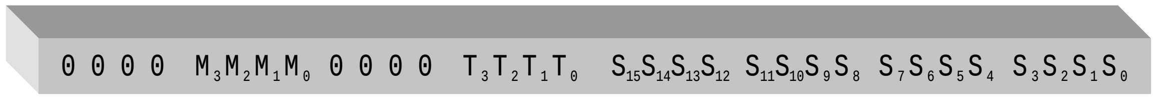
The neutron rejection algorithm easily keeps all muons while rejecting approximately 94% of neutrons and photons [18]. After neutron rejection, the clusters are formatted and output as one half of an ATLAS standard Event fragment.

Neutron-rejected data format

Events

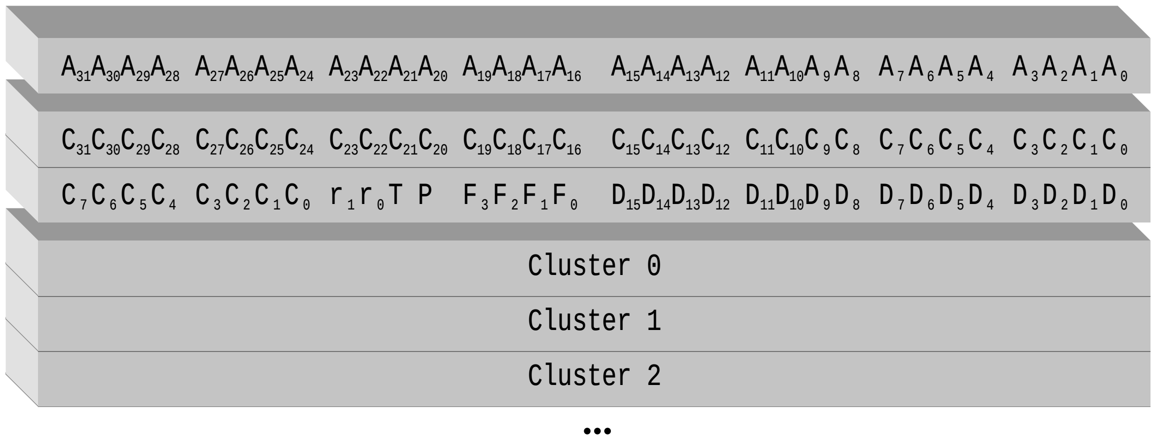


Common header


0 0 0 0 $M_3M_2M_1M_0$ 0 0 0 0 $T_3T_2T_1T_0$ $S_{15}S_{14}S_{13}S_{12}$ $S_{11}S_{10}S_9S_8$ $S_7S_6S_5S_4$ $S_3S_2S_1S_0$

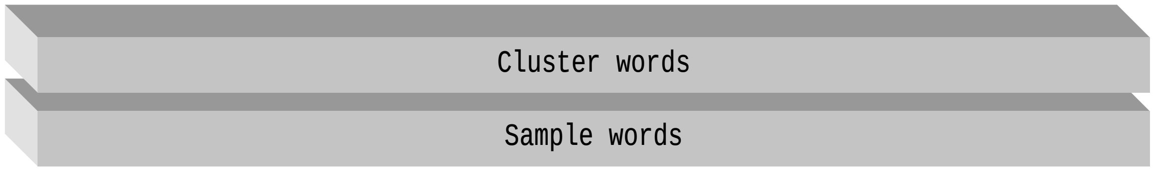
- M = Module ID
- T = Type of Event (0 = Normal, 1 = Discard, 2 = Error)
- S = Size of Event (in words, including the common header and ghost words)

Data


$A_{31}A_{30}A_{29}A_{28}$ $A_{27}A_{26}A_{25}A_{24}$ $A_{23}A_{22}A_{21}A_{20}$ $A_{19}A_{18}A_{17}A_{16}$ $A_{15}A_{14}A_{13}A_{12}$ $A_{11}A_{10}A_9A_8$ $A_7A_6A_5A_4$ $A_3A_2A_1A_0$
$C_{31}C_{30}C_{29}C_{28}$ $C_{27}C_{26}C_{25}C_{24}$ $C_{23}C_{22}C_{21}C_{20}$ $C_{19}C_{18}C_{17}C_{16}$ $C_{15}C_{14}C_{13}C_{12}$ $C_{11}C_{10}C_9C_8$ $C_7C_6C_5C_4$ $C_3C_2C_1C_0$
$C_7C_6C_5C_4$ $C_3C_2C_1C_0$ r_1r_0T P $F_3F_2F_1F_0$ $D_{15}D_{14}D_{13}D_{12}$ $D_{11}D_{10}D_9D_8$ $D_7D_6D_5D_4$ $D_3D_2D_1D_0$
Cluster 0
Cluster 1
Cluster 2
•••

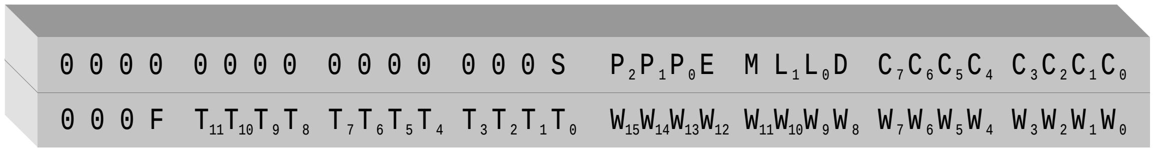
- A = SCA addresses (first timeslice in high byte)
- C = Cluster counts (first precision in high byte of first word)
- r = Reserved (may have any value)
- T = Trigger type (priority)
- P = Trigger/sampling phase (0 = sample B closer to nominal peaking time, 1 = C)
- F = First bit summary (first timeslice in high bit)
- D = Number of data words that follow in clusters

Cluster



Cluster words
Sample words

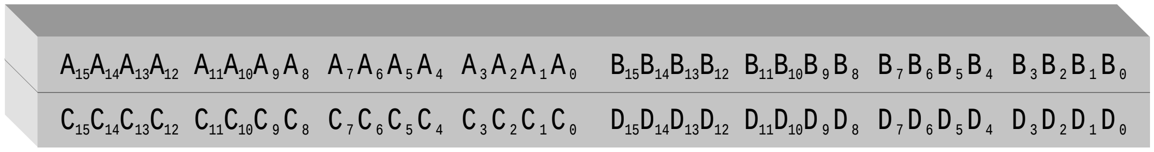
Cluster words



0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 S	$P_2 P_1 P_0$ E	M $L_1 L_0$ D	$C_7 C_6 C_5 C_4$	$C_3 C_2 C_1 C_0$
0 0 0 F	$T_{11} T_{10} T_9 T_8$	$T_7 T_6 T_5 T_4$	$T_3 T_2 T_1 T_0$	$W_{15} W_{14} W_{13} W_{12}$	$W_{11} W_{10} W_9 W_8$	$W_7 W_6 W_5 W_4$	$W_3 W_2 W_1 W_0$

- S = Size bit (big or small CSC chamber)
- P = Phi angle
- E = Eta (endcap)
- M = Multilayer (0)
- L/D/C = Layer/Direction/Channel number (D = 0 for precision, 1 for transverse)
- F = Failed to compute time (time will be 0)
- T = Time (signed, in nanoseconds)
- W = Width of cluster (number of sample word pairs)

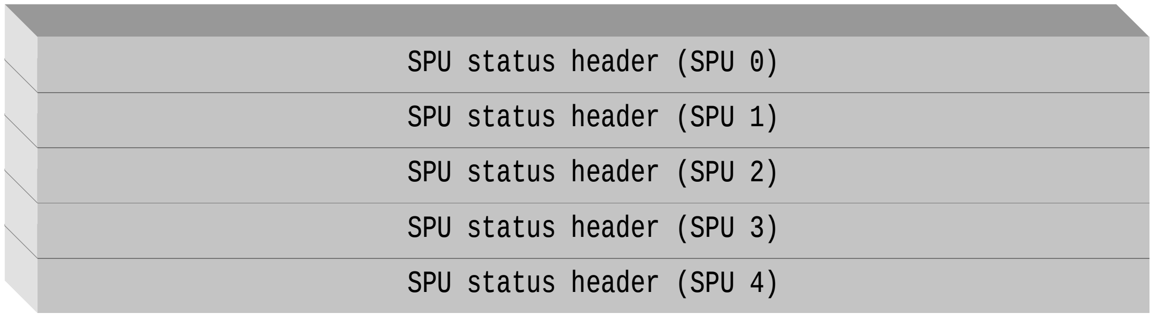
Sample words



$A_{15} A_{14} A_{13} A_{12}$	$A_{11} A_{10} A_9 A_8$	$A_7 A_6 A_5 A_4$	$A_3 A_2 A_1 A_0$	$B_{15} B_{14} B_{13} B_{12}$	$B_{11} B_{10} B_9 B_8$	$B_7 B_6 B_5 B_4$	$B_3 B_2 B_1 B_0$
$C_{15} C_{14} C_{13} C_{12}$	$C_{11} C_{10} C_9 C_8$	$C_7 C_6 C_5 C_4$	$C_3 C_2 C_1 C_0$	$D_{15} D_{14} D_{13} D_{12}$	$D_{11} D_{10} D_9 D_8$	$D_7 D_6 D_5 D_4$	$D_3 D_2 D_1 D_0$

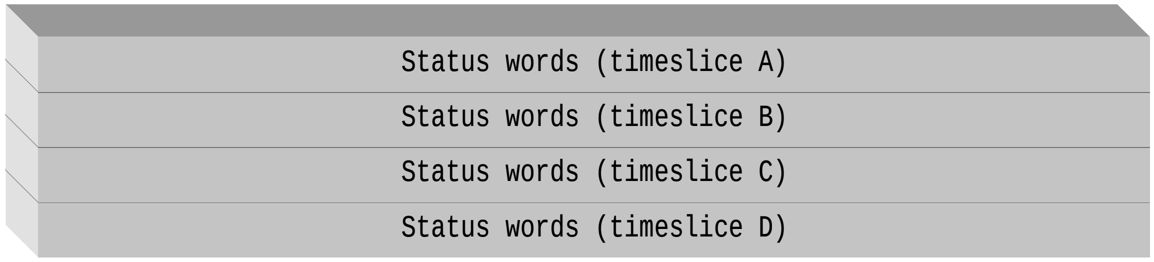
- A = sample A
- B = sample B
- C = sample C
- D = sample D

RPU status header



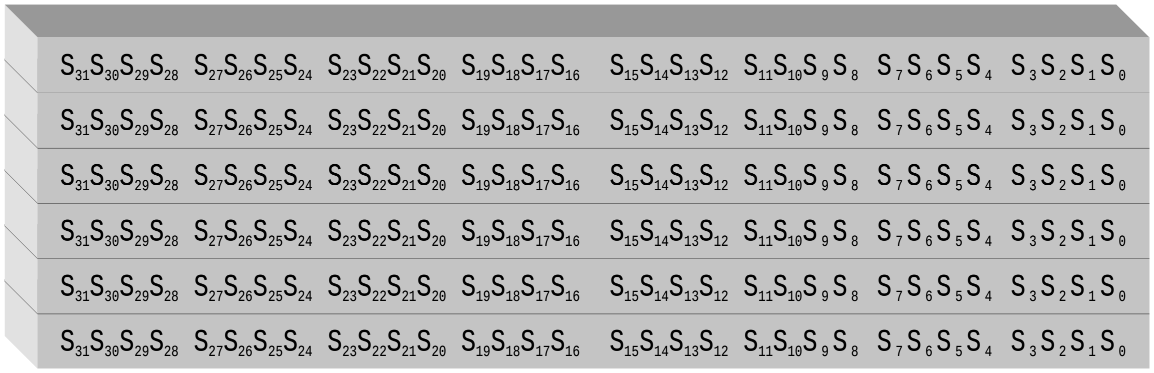
SPU status header (SPU 0)
SPU status header (SPU 1)
SPU status header (SPU 2)
SPU status header (SPU 3)
SPU status header (SPU 4)

SPU status header



Status words (timeslice A)
Status words (timeslice B)
Status words (timeslice C)
Status words (timeslice D)

Status words



$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$
$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$
$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$
$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$
$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$
$S_{31}S_{30}S_{29}S_{28}$	$S_{27}S_{26}S_{25}S_{24}$	$S_{23}S_{22}S_{21}S_{20}$	$S_{19}S_{18}S_{17}S_{16}$	$S_{15}S_{14}S_{13}S_{12}$	$S_{11}S_{10}S_{9}S_{8}$	$S_{7}S_{6}S_{5}S_{4}$	$S_{3}S_{2}S_{1}S_{0}$

- S = Status words from BPI

Ghost words

At the end of Events can be added one or more ghost words. The presence of these words can be inferred by comparing the reported total size with the actual size of any data words and known headers. These words are not used in normal running and should be ignored by ATLAS software. The primary use of these words is to aid debugging during commissioning.

The current ghost words are the Status header.

Implementation

More so than even the SPU, the greatest challenge of the RPU Decoder is controlling the worst-case performance. Because of the low occupancies, there is a huge difference between the typical Event and the worst-case Event. It was important during design to allow the case of all channels hit to be performed efficiently because this can happen if an anode wire is struck and gets an induced charge. Thanks to extensive use of cluster-based algorithms, the worst case on both the SPU and RPU is the minimum number of channels per cluster with a gap of one channel between clusters. There is only one assembly language routine used in processing on the RPU:

NeutronReject **assembly function**

1. Reads the clusters and samples sent from the SPU.
2. Applies neutron rejection to all clusters in all layers.
3. Stores passing clusters to the output buffer.

Performance: 2 * (# of hits) + 11 * (# of clusters) + 39 DSP clocks (disables interrupts)

NeutronReject() is the most complicated single assembly language function in the DPU. It is a three-pass loop that branches out to a run-out routine to copy the cluster's samples when it passes neutron rejection. Thanks to the effort that went into this function, the RPU can handle even the absolute worst case in less than 10,000 DSP clocks as required.

Calibration array

Status.Basic.Calibration is a pointer to a 1 kW buffer used to store Decoder initialization information. The format of this information for the RPU Decoder is:

- Precision overlap count
  Determines how much track inclination is allowed in precision layers
  1 word, unsigned
  See below for format and restrictions on the value

- Transverse overlap count
  Determines how much track inclination is allowed in transverse layers
  1 word, unsigned
  See below for format and restrictions on the value

- Channel mask
  Contains bits to be bitwise ORed into all channel numbers
  1 word, unsigned
  Only bits 1 (0x2) through 16 (0x10000) may be set

The precision and transverse overlap counts tell the RPU how many extra bits to set in the bitmaps before bitwise ANDing with the cluster bitmap and looking for overlap. When the overlap count is zero, clusters must have actual overlap in order to be considered a match. When the overlap count is one, clusters that would touch without overlap if placed in the same layer will match. When the overlap is two or more, then clusters that would have a gap of the overlap count minus one if placed in the same layer will match. The only value allowed for the overlap count is one, but this can be changed with a small modification to the RPU.h file.

The value sent in the overlap count is used for neighboring layers. If there is one intervening layer, twice the overlap count is used. If there are two intervening layers, three times the overlap count is used. Thus the overlap count is a measure of how much the track can be inclined.

Implementation

File Layout

Base Files

main.cpp	**common.h/.cpp**	**shared.h**	**fault.h**
Starting point, high-level routines	Included by all files, also includes shared.h	Included by all files and by the HPU	List of all Fault codes in C++ macros

Driver Files

DMA.h/.cpp *TChannel* class for controlling DMA channels	**FPGA.h/.cpp** Functions for initializing FPGAs and reading/writing registers	**Timer.h/.cpp** *TTimer* class for measuring time intervals
Platform.h Constants that specify memory extents for the current DSP type	**mini_dpu.h** *TMiniDPU* class for XFPGA init, included by FPGA.h/.cpp	**6202.cmd/6203.cmd** Linker command files for 6202 and 6203 DSPs

Hardware Abstraction Layer (HAL) Files

Status.h Status struct access and Basic initialization	**Policy.h** Policy system access and initialization	**Parameter.h** Parameter system access functions	**Response.h** Response buffer functions for decoders
Queue.h Input/output queue classes	**PriorityQueue.h** *TPriorityQueue* class for Tasks	**Heap.h** *THeap* class for Task memory	**HAL.cpp** Instance definitions for the HAL headers above
struct.h Declaration of *TStatus* struct, included by HPU	**Util.h/.cpp** LED and performance monitor utilities	**simulate.h** Functions used to fill input buffers for testing	**HALControl.h/.cpp** *TControl* interface for the HAL and Drivers

Policy\policies.h Point of declaration for all Policy modes	**Policy\include.h** Point of inclusion for all Policy mode headers	**Policy\dt_input.h** Input data type Policy mode header	**Policy\dt_output.h** Output data type Policy mode header

Queue\input.h *Update()* functions for input	**Queue\input.cpp** *Initialize()*/*Terminate()* functions for input	**Queue\output.h** *Update()* functions for output	**Queue\output.cpp** *Initialize()*/*Terminate()* functions for output

Data System Files

DataControl.h/.cpp *TControl* interface for the Data system	**Data.h/.cpp** Data system main functions	**decoder.h** *TDecoder* base class for Decoders	**enum.h** Enum of all Decoders, included by HPU

decoders\decoders.h Point of declaration for all Decoders	**decoders\include.h** Point of inclusion for all Decoder headers	**decoders\SPU.h** Decoder class for SPU Decoder	**decoders\RPU.h** Decoder class for RPU Decoder
decoders\Timeslice.h Early test version of a timeslice Decoder	**decoders\SIT.h** System Integration Test Decoders	**decoders\Beamtest.h** Decoders used during the beam test	**decoders\DXTest.h** Decoders used for DX testing

Command System Files

CommandControl.h/.cpp *TControl* interface for the Command system	**Command.h/.cpp** Command system main functions	**function.h** *TFunction* base class for Functions	**fenum.h** Enum of all Functions, included by HPU

functions\functions.h Point of declaration for all Functions	**functions\include.h** Point of inclusion for all Function headers	**functions\Basic.h** Basic functions and ones used to control the Scheduling system

Scheduling System Files

SchedulingControl.h/.cpp	**Scheduling.h/.cpp**	**task.h**	**tenum.h**
TControl interface for the Scheduling system	Scheduling system main functions	*TTask* base class for Tasks	Enum of all Tasks, included by HPU

tasks\tasks.h	**tasks\include.h**	**tasks\Hardware.h**
Point of declaration for all Tasks	Point of inclusion for all Task headers	Tasks used to monitor hardware such as temperature

HPU Files

common.h Wrapper for shared.h	Data.h Wrapper for Data\decoders\decoders.h	Command.h Wrapper for Command\functions\function.h	Scheduling.h Wrapper for Scheduling\tasks\tasks.h
Status.h Wrapper for HAL\struct.h	gui.h Wrapper for fault.h	Copies of: shared.h fault.h HAL\struct.h Data\decoders\decoders.h Command\functions\function.h Scheduling\tasks\tasks.h	

Status Structure

The Status structure is a C++ POD struct used to communicate with the HPU. There are three main sections:

Basic Section

Version information
Initialization control
Boot information
Hardware configuration
Mode configuration
Task configuration
Mode overrides
Data-Scheduling-Command priority configuration
Orders
Warnings
Faults
Run configuration

Processing Section

Event processing counters
Time information
Discard counters
Event output counters
Performance counters
Temperature monitoring

Buffer Section

Parameter set pointers
Response buffer pointer and index
DIB/DOB priority information
Command buffer pointer and priority
Scheduling priority

The struct is accessible by both the DPU and the HPU (via DPU Control). A comment is made for each item defining which DSP is allowed to read/write at what stage of operation to prevent conflicts.

In retrospect, some sort of double buffering would have been preferable to allow the HPU to get a consistent snapshot of various counters and flags that are constantly being updated by the DPU. Fortunately this functionality has not been needed, primarily due to the use of the Command and Response buffers for most critical communications.

Policy System

The Policy system maintains a small database of configuration options based on input and output data types. For each possible input or output types, there are several values stored that describe the data and how it fits into the input/output buffers:

Input

InputEventLength	Number of Packets per Event
InputMaxPacketLength	Maximum number of Frames per Packet
InputFrameSize	Size of a Frame, in words
InputYellowBar	Threshold for setting priority Yellow (# of unused Frames)
InputRedBar	Threshold for setting priority Red (# of unused Frames)
InputTransferCount[4]	Number of events to DMA in at various priorities

Output

OutputEventLength	Number of Packets per Event
OutputMaxPacketLength	Maximum number of Frames per Packet
OutputFrameSize	Size of a Frame, in words
OutputYellowBar	Threshold for setting priority Yellow (# of unused Frames)
OutputSkipLength	Used to control padding and sending of size word in output

For every possible data type, there is a set of constants in the *DPU::Policy::Private::<name of data type>* namespace that corresponds to the above values. During initialization, these are copied to a set of read-only variables in the *DPU::Policy::Public* namespace. This way, a Decoder can choose to use the constant values in the private namespace or the run-time values in the public namespace. Using the public variables is more flexible, but using the constants in the private namespaces allows for maximum performance for time-critical Decoders. Decoders that do not use the public values will report whether they are compatible with the current values and not be loaded during the run if they are not.

The read-only variables are actually instances of a template class *TEntry* that takes the type of the variable as its template parameter. The cast operator for that type is overloaded to allow the instance to act like a variable of that type when read, while assignment is restricted to the Policy system initialization functions.

Input/Output Queues

One of the most important aspects of a clean DPU software design is separation of buffer management from data processing. To that end, the input and output queue classes handle all internal aspects of data flow and present a simple interface to Decoders. Furthermore, the layout of the generic event is designed to decouple data flow from data processing by making the input/output queues independent of the data type.

At first glance, the need for a single interface to multiple buffer operational modes (different external sources, variable vs. fixed-length events) suggests the use of virtual functions. More specifically, the Template Method Pattern [22] would allow the sharing of common code with virtual functions to perform the mode-specific operations such as setting up for output or checking whether an entire event is available. However, this has two main problems: the Template Method Pattern would use one virtual function call per function requiring specialization, and it would also require dynamic memory allocation of an instance of one of the derived classes in the queue hierarchy. This contradicts our desire for maximum performance and to avoid dynamic memory allocation. As will be seen below, the solution is to use specialization of template functions in place of virtual functions and use a single member function pointer to duplicate the virtual function call mechanism without dynamic memory allocation.

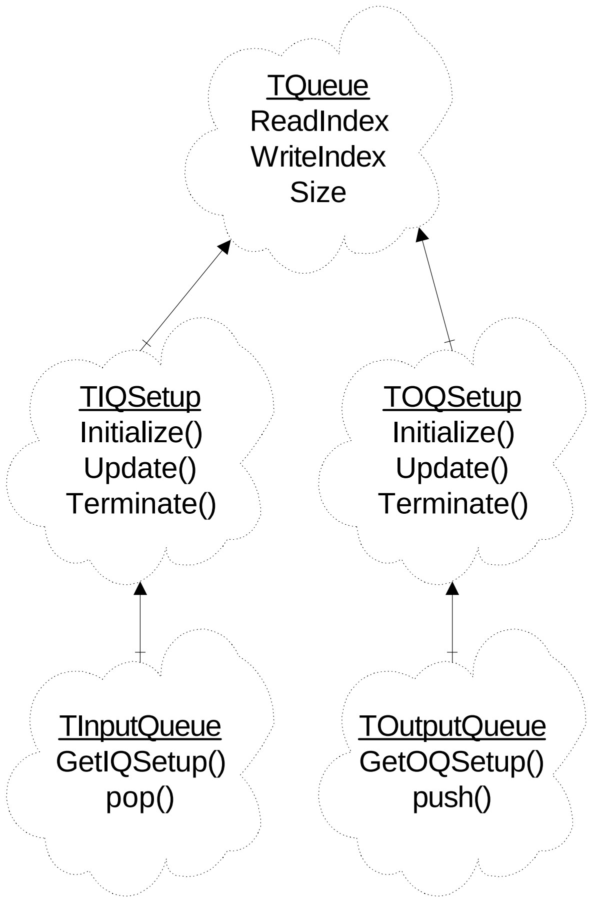


Figure 27: Class hierarchy for the input/output queues

The base class for both input and output queues is a struct called *TQueue*. This does not reflect commonality of interface but rather implementation, so protected inheritance is used. The derived classes, *TIQSetup* and *TOQSetup*, contain the input/output queue-specific data members and most of the functionality of the queues. As with other parts of the DPU software, these classes support *Initialize()*, *Update()*, and *Terminate()*, and most of the work is done in *Update()*. In particular, *Update()* is responsible for keeping data flowing and getting prepared to pass the next event through the Decoder that requests it.

Logically one might expect to see *push()* and *pop()* found in these classes, but this opens up a problem. The triplet of *Initialize()*, *Update()*, and *Terminate()* (and their helper functions) are only supposed to be used by HAL Control during the corresponding phases, but these would also be available to every Decoder with no way to detect a misuse. This same problem was even more serious in the design of the DMA channel class *TChannel*, and the solution there was to use protected inheritance to hide the interface. For the queue classes, this means using protected inheritance to derive the leaf classes *TInputQueue* and *TOutputQueue* from *TIQSetup* and *TOQSetup*, respectively. These leaf classes are where functions like push and pop reside, and a single public function *GetIQSetup()*/*GetOQSetup()* provides a reference to the protected base so HAL Control can access it from the instances of *TInputQueue*/*TOutputQueue* we create. The important thing to note is that this does not prevent intentional misuse, but rather produces a compile-time error if there is any unintentional use. This is less significant for the queue classes, but proved invaluable in the DMA channel classes where catching such misuse revealed a design flaw in how certain initialization was being done.

The question remains of how to implement the above functions. Essentially, the goal is to make a single indirect function call through a function pointer during *Update()* and leave helper functions and *push()*/*pop()* as ordinary function calls. Making *pop()* an ordinary call is done by filling a data structure during *Update()* with everything needed to find the next Event and pop some or all Packets off. The *push()* function is even simpler as the information it needs is provided by the buffer pointers and the function call parameters.

The function *TemplateUpdate()* takes care of the work to be done in *Update()*. This is a template function whose template parameters are an enum for the buffer type and a bool for whether the Events are fixed or variable-length. *TemplateUpdate()* can be specialized completely if necessary, but we can also follow the Template Method Pattern and write a single version of the code that calls helper template functions that get specialized instead. The only thing remaining is storing a pointer to *TemplateUpdate()* during *Initialize()* and calling that in our *Update()* function. Pointers to member functions like *TemplateUpdate()* generally incur size overhead in order to accommodate things like virtual functions, and on the TI DSP these pointers are two words instead of one. To avoid having to load two words during the indirection, we make a friend template function *s_Update()* that is passed a pointer to the queue instance and calls the corresponding *TemplateUpdate()* function on it. A pointer to one of the specializations of *s_Update()* is stored in *p_Update* and called by *Update()* by passing the this pointer. This gives all the advantages of the Template Method Pattern with a single virtual function call emulated with a function pointer to avoid dynamic memory allocation.

Task Priority Queues

Priority Queues

A priority queue is a container that sorts its elements and presents only the highest priority item for removal. This is ideal for use in the Scheduling System for holding Task Capture and Service requests.

Priority queues are generally implemented using a partially sorted binary tree called a binary heap. Each element can have up to two children with the condition that an element is of equal or greater priority than either child. A binary heap with N items occupies the lowest N elements of the array containing it and grows upwards as new items are added.

Standard Template Library

The C++ Standard Template Library (STL) contains an adaptor called *priority_queue* that can be used to convert a container class (such as *vector*) into a priority queue [8]. There are several problems with using this in the DPU software:

- All the container classes, including *vector*, use dynamic memory allocation
- Half of the memory will be wasted if there are separate Capture and Service priority queues because a given Task will have a request in one or the other but never both at the same time
- The TI DSP libraries do not include the STL

Custom Priority Queue

To resolve this, we need a custom version of *priority_queue* that has two features:

- No use of dynamic memory allocation
- Shares memory space between the Capture and Service queues

The basis for this new class, called *TPriorityQueue*, is the implementation of the STL *priority_queue* from a free open-source library called STLPort [40]. Instead of relying on an STL container class, this adaptor is modified to use a fixed-size array of memory and report a Fault if the array overflows.

To address the issue of shared memory, we duplicate the code for managing the binary heap and reverse it so it controls a heap starting at the last element of the array and growing downward. Combining these two versions gives us a two-sided binary heap that will inherently allow memory to be shifted between the Capture and Service priority queues automatically.

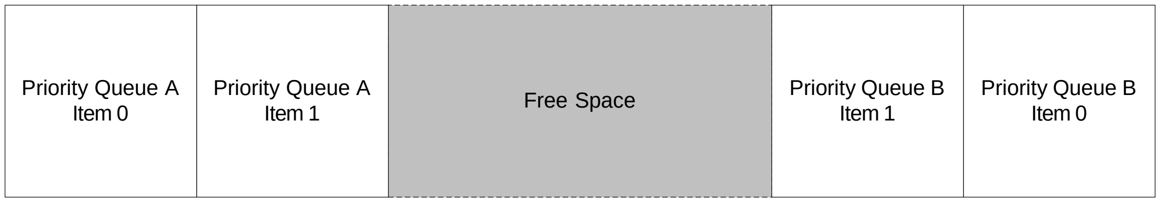


Figure 28: Allocation of memory between the two priority queues with free space between

Task Heap

The one place where dynamic memory allocation is needed is the Scheduling System to allow for runtime creation of Tasks. Unfortunately the built-in C++ *new* operator is optimized for size efficiency and cannot meet our real-time constraints [41]. The simplest memory allocation algorithm we could use is to partition a large buffer into fixed-size chunks large enough to hold the largest possible Task. This would work fine if most Tasks were of comparable size, but we expect to have small Tasks for monitoring a single quantity and large Tasks for Event capture and histogramming. A one-size-fits-all approach would be very inefficient. Having two buffers using two different sized chunks could work, but we might find ourselves with no free memory in one buffer and plenty of free memory in the other. What we would like is to find a solution similar to the double-ended priority queue that would allow us to collocate the small and large chunks while allowing for inevitable fragmentation of memory.

The solution is to choose the large chunk size to be a power-of-two multiple of the small chunk size (for efficiency we choose both chunk sizes to be powers of two). The buffer is divided into large chunks that can be allocated for larger Tasks. If a smaller Task is needed, a large chunk can be subdivided into several small chunks (the ratio of the two sizes) and those small chunks can be handed out.

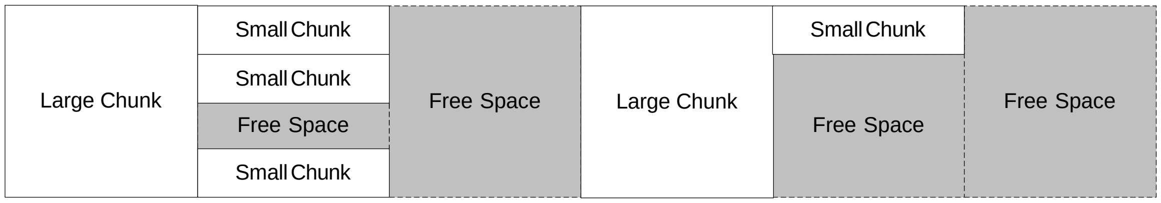


Figure 29: A typical allocation of large and small chunks in the Task heap

The large chunks are managed by a set of *THeapNode* instances that contain a bitmap indicating which contained small chunks are in use when subdivided. These are created in a static array and the index in the array is the index of the corresponding large chunk of memory being managed. Each *THeapNode* instance stores previous/next indices and they are initially connected to form a doubly linked list that contains all free large chunks. When a large chunk is needed, the head of this list is removed and the corresponding chunk is given out for use. When the first small chunk is needed, the first small chunk in the head of the free large chunk list is marked as used in the bitmap and the *THeapNode* instance is moved to a new list for large chunks with one small chunk in use. There are similar lists for large chunks with two, three, or any number of small chunks in use including all of them. When a new small chunk is needed, the lists are checked to find the large chunk with the fewest small chunks free. Using this large chunk helps minimize memory fragmentation.

To free memory the address is used to locate the corresponding *THeapNode* instance (and the small chunk within it if necessary). If it is a large chunk being freed then the chunk is simply placed at the head of the free large chunk list. If it is a small chunk then the bitmap is updated to free the small chunk and the *THeapNode* instance is moved to the list that corresponds to the new number of small chunks in use. This requires splicing the list around the *THeapNode* instance where it is removed (which requires doubly-linked lists).

Allocating a large chunk, freeing a large chunk, and freeing a small chunk are all constant-time operations. Allocating a small chunk is linear in the ratio of the large and small chunk sizes (the head of each list must be checked to find the large chunk with the fewest free small chunks). Since this ratio is fixed, this is also constant time and does not scale with the total number of memory chunks.

Testing

Real data, realistic data, and arbitrary data

When testing to find errors and verify proper operation, there are three possible inputs that can be used:

- Actual data from the front end
- Simulated data that replicates the front end
- Arbitrary random data that fits the input format

At first glance, it would seem that actual data would always be ideal for testing. After all, this is the same kind of data that will be used in the final system. The same argument could be made for accurate simulated data, with the added advantage of being able to generate as much test data as desired. However, consider the case of a bug that only manifests in an obscure case that is unlikely to occur. Using actual or simulated data is far less likely to uncover this bug, but the consequences of not finding the bug are still very serious.

If a bug occurs for only a single input and no others, it wouldn't matter how you chose test input; either it would be found or it would not be found, depending on whether you lucked into using the one input that causes it. Fortunately real bugs tend to occur for a variety of similar cases or when a certain condition occurs, and they are not restricted to the range of typical inputs. In fact, uncaught bugs are more likely to be found in boundary cases where an unusual condition occurs that was not anticipated when designing for the general case. In this case, distributing test cases widely throughout the entire range of possible inputs is more likely to catch all bugs than focusing them in the typical input.

Moreover, there are many places where realistic inputs will be used during testing and commissioning, so these already get extra focus. Therefore, it is best to use random and arbitrary input to test for accuracy and locate bugs.

Testing vs. design for performance

When it comes to high performance software, testing alone cannot guarantee that a non-trivial program is able to meet hard real-time requirements. This is because the program contains many possible decision paths and the worst-case performance may be on a path that was not exercised. To avoid this, it is important to keep the core logic straightforward to avoid seeing large differences in execution time for different input conditions. This links back to the previous discussion of design rules that said we can't have loops that run an indeterminable number of times.

Assembly language functions make performance determination simpler because their clock cycles can be counted directly. Thus the goal of performance testing is to determine the additional cost for the remaining C++ code so that the final performance can be examined. Once complicated logic is simplified, it is possible to run over a sample of realistic data and look at the average and maximum processing time per Event. With these numbers in hand and verified acceptable, the known assembly language function run times can be used to scale these times to higher rates

Because we anticipate running various Tasks, the performance will ideally have headroom for running them. Since Tasks are non-critical and can be shutdown if needed to keep up, this headroom ensures a large safety factor is available for the primary data processing.

Faults and Warnings

Testing does not stop once the final system is online. During normal operation a number of checks are done to catch possible errors. Failure of these tests represents a fatal design flaw and cannot be tolerated, so a Fault is issued. The result of this Fault is a diagnostic code sent to the HPU and a total shutdown of all operation. Faults therefore represent a condition from which recovery is neither possible nor desirable. This addresses the design requirement that bad data should never be output.

When a less serious error occurs that can be recovered from, a Warning is issued and the HPU is so informed. For example, there is a Warning issued when the main loop processing time exceeds the design value, while a Fault is issued if the time exceeds the maximum time that can be tolerated without potential input buffer overflow. Warnings allow errors to be detected without spoiling a run that is likely successful. If necessary, the run data can be rejected after the fact when the cause of the Warning is tracked down.

Beyond Faults and Warnings are a number of assertion tests that are executed only during initial testing. These tests confirm things that are unlikely to go wrong in a fully debugged system but which can be valuable to check early on in development. These tests can often be time-consuming so they cannot be left in the final system. All testing except performance testing is done with asserts enabled.

The decision to add a Fault or assert check is made by the developer based on his or her best judgment. Ideally there will be asserts sprinkled liberally throughout the code, with Faults used in places where an error is deemed potentially likely or particularly serious. Asserts are often sufficient in places where an error could occur during development but which need not be tested every time, such as making sure that the pointer returned by the output buffer is non-zero before dereferencing. Faults are more often needed in situations where the data being tested comes from an outside source.

Accuracy testing

General

To test the system in a situation close to the final mode of operation, we need to take all testing operations (simulation of input, checking of output) off the DPU under test. This is easy to do if we make use of the programmable nature of the Data Exchange. This allows us to route generated input data through two different DPUs: one that runs the software under test and another running software that generates equivalent output. A final DPU runs software to compare the two and determine whether it passes.

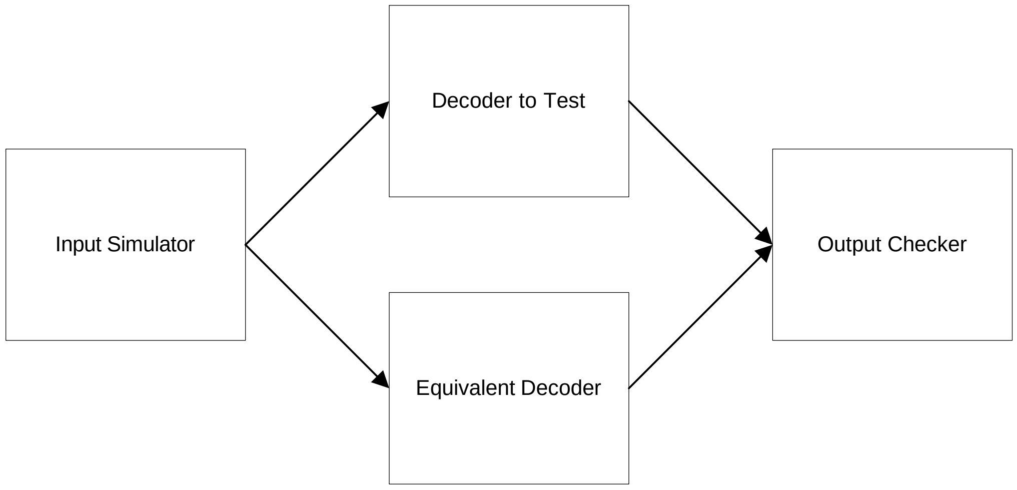


Figure 30: Test setup

Thanks to the flexibility of the DPU software framework, all four DPUs can run the same software with different Decoders used. The input simulator runs *DPU_SimulateDecoder* and the output checker runs *DPU_CompareDecoder*.

Generating random input is easily done, but the question remains of how to check the output. The Decoder algorithm itself is best tested using Monte Carlo simulation to determine the efficiency, which requires a C++ version with no assembly language that can be run on other platforms. What we wish to test here is that the algorithm is properly implemented and produces precisely the output we expect. To meet both goals, the equivalent Decoder is a bit-accurate simulation of the Decoder under test. To minimize the chance of making duplicate errors, the algorithm is reimplemented from scratch in as clear and simple a way as possible. Performance is not a concern here, so completely different methods can be used to implement the equivalent Decoder. Only the most trivial of code can be shared between these two implementations.

Because the two Decoders are bit-identical, the output checker has a very simple job. It compares the two inputs and displays both when an error is found. The data contains the random number seed used so the Event that caused a problem can be repeated easily.

SPU

The SPU Decoder was tested with a few hundred simulated Events to eliminate bugs. Further verification with many millions of Events is now needed to confirm that no other bugs remain.

RPU

The RPU Decoder was tested with a few hundred simulated Events to eliminate bugs but was not subjected to deeper testing. The RPU algorithm is currently being examined in more detail to see if neutron rejection should be used, and will be tested further once that decision is made.

Performance testing

Because we have limited branches and decision points to a minimum, the performance of the main loop is very close to being a constant plus the performance of the individual assembly language functions. To put it another way, the processing time varies significantly only with the number of channels/clusters, and this variation is limited to the assembly language routines. Because of this, we only need one data point to determine this constant. To be safe, we run a number of mock events with the same number of channels/clusters and take the worst time.

The pattern we will use to test is chosen to be higher in occupancy than the real system. We hope to find this close to our 3000 clock cycle limit and thus have maximum accuracy in our fit near the critical point. We place four clusters of one channel each and one cluster of two channels for each SPU, and we place the output of these SPUs into the RPU. This simulates the worst case where everything passes neutron rejection. One neighbor is marked on each SPU; the actual transverse will likely not even need this. All numbers are in DSP clock cycles.

- Precision SPU: 895 in the framework + 2090 in the Decoder = 2985 clocks
- Transverse SPU: 895 in the framework + 2140 in the Decoder = 3035 clocks
- RPU: 1025 in the framework + 2181 in the Decoder = 3206 clocks

The RPU takes longer in the framework because it uses variable-length input.

Now that we have this information, let's look back at the performance numbers for the assembly language routines when we mark one set of neighbors. Let H be the number of initial hits without neighbors and let C be the number of clusters. After neighbors are marked, we will have a total of H + 2C channels being kept.

SPU:

• *OOTEliminate*:		241 clocks
• *MarkNeighbors*:		20 clocks
• *ParseBitmap*:	7 * (H + 2C)	+ 28 clocks
• *FindPeak*:	10 * (H + 2C)	+ 28 clocks
• *CalculateTime*:	14 * C	+ 34 clocks
• *CutTime*:	3 * C	+ 7 clocks
• *AddBitmap*:	14 * C	+ 36 clocks
• *LinkList*:	6 * C	+ 8 clocks
• *OutputResults*:	3 * (H + 4C)	+ 15 clocks

RPU:

• *NeutronReject*:	2 * H + 15 * C	+ 39 clocks

The total for the SPU assembly language functions is 20H + 83C + 418 clocks. The RPU assembly language function must be run five times: four times for precision SPUs and one time for the transverse SPU. Letting the P subscript denote the precision counts and the T subscript denote the transverse, the total for the RPU is $8H_P + 2H_T + 60C_P + 15C_T + 195$ clocks. We can compare these assembly language numbers to the performance test numbers to determine the actual constants for the total time to process one Event:

- Precision SPU: 20H + 83C + 2450 clocks
- Transverse SPU: 20H + 83C + 2500 clocks
- RPU: $8H_P + 2H_T + 60C_P + 15C_T + 2771$ clocks

Looking back at the numerology, the precision SPU has an average of 1.1 clusters and 5.4 channels per Event, while the transverse SPU has an average of 4.3 clusters and 8.7 channels. Plugging these in give us the estimated performance:

- Precision SPU: 2650 clocks
- Transverse SPU: 3031 clocks
- RPU: 2962 clocks

When we compare these numbers to the 3000 clocks we have available at a 100kHz trigger rate, we are concerned. There is no problem currently as the initial design requirement is only a 75kHz trigger rate, but the question remains what will be done during the upgrade to 100kHz. As it turns out, the current Decoders contain a number of runtime checks that ensure proper operation. Most of these checks are unnecessary because they are only verifying that the system has not produced invalid values at some point along the way. After years of error-free operation at the lower 75kHz trigger rate it would be safe to remove these checks during the upgrade, which would save more than enough time to reach the 3000 clock cycle count needed for 100kHz operation. Alternatively, faster DSPs may be available at the time of the upgrade.

We can also look at the worst-case performance. Maximizing the cluster count will maximize the time, so the worst case will be hitting every fourth channel (three channels per cluster after neighbor marking plus a one channel gap between clusters). This means 48 clusters and 48 channels hit before neighbor marking:

- Precision SPU: 7394 clocks
- Transverse SPU: 7444 clocks
- RPU: 6851 clocks

We are well under the 10000 clock cycle limit we have set.

Conclusion

Summary

The DPU software framework is a flexible, high-performance platform for data acquisition. With appropriate Decoders the framework can be adapted to many different uses, and could serve as a starting point for a software framework for a generic ROD. In the case of the CSC Decoders, performance is acceptable for a 75kHz trigger rate and can be tweaked to allow for a 100kHz trigger rate if needed.

The reason things are tight for a 100kHz trigger rate is the effort put into making the worst-case performance acceptable. This is a serious issue in a system with such a low occupancy (and thus a huge difference between the typical and worst cases). Many tricks were used to make things constant-time rather than variable, which explains the large constant terms in all the performance formulae. One advantage to this is it reduces the need for large amounts of headroom, as the time is not strongly varying with the number of hits/clusters.

A DSP-based software framework allows for calculations and flexibility not practical on an FPGA design. The initial FPGA-based sparsifier was expected to perform only the job of the *OOTEliminate* assembly function (along with simple buffer management and output formatting). Thanks to the DSP and software framework, the SPU Decoder is able to do far more and even assists the RPU by pre-calculating the cluster bitmaps. To a great extent, the functionality of the Decoders has expanded to fill the processing time available. This means maximizing work performed for the same price, and can relieve some of the burden on later processing stages. This is something that must be considered when weighing such a flexible solution against a more limited approach using programmable hardware devices.

Future work

The immediate priority is to complete the verification testing for the SPU, which has been delayed due to a problem with a computer needed to operate the ROD. This is the last stage in confirming that the equivalent SPU Decoder is an exact match for the actual SPU Decoder.

The next step is to perform Monte Carlo simulation using the equivalent Decoders to see how the algorithms perform. This testing was done earlier using simplified versions of the algorithms but needs to be reexamined now that the full bit-identical Decoders are available. This is the final step in verifying that the Decoders are acceptable for processing data in ATLAS.

The DPU software framework is general enough to use in other situations. The modular design for the plug-ins is ideal for use in a generic ROD. There are a number of items that could be improved for this purpose. First, the Status structure acts like a giant global variable. Refactoring the design to eliminate this would decouple the individual systems and allow for better unit testing. This is also important if future uses require multiple input or output buffers with different conditions for deciding when to begin processing.

Once the decoupling is done, additional input and output Event formats can be used. The current generic Event has been surprisingly flexible with a number of test formats and is fairly efficient to process, but more flexibility could be added to a new set of queue classes which could be substituted at compile time. This would not affect the current performance but would extend the range of possible uses for the framework.

PART II

K Physics in the FDQM

Motivation

Introduction

The Correspondence Principle tells us that a new theory should be as good as the theory it replaces where the old theory was accurate, and better than the old theory in an area the old theory was incorrect. Put more simply, the best theory is the one that fits experiment best. This seems reasonable, but one could devise a purely empirical "theory" that will always fit experiment perfectly through the use of an arbitrary number of tuned parameters. This is of no predictive value, so some measure of the quality of fit relative to the number of tuned parameters is necessary to evaluate theories. One way to do this is to do a minimization of chi-squared for all possible values of the theory's parameters. This takes everything into account, but is only useful to evaluate what the best values are. In the end, the actual value of chi-squared must be examined to determine a confidence level.

Such confidence levels cannot ever prove a theory, but they can exclude a theory once there is no possible set of parameter values that could fit experiment with any significant confidence. To make it easier to get an intuitive feel, multidimensional contour plots of confidence levels can show what ranges and combinations of parameters are allowed. These plots can guide physicists in selecting lines of experimental research to explore further.

CP violation theory in the K system

The K mesons form an octet with the other light spinless mesons of negative parity. The decay of kaons is of great interest in the study of CP violation because for a time it was the only system where CP violation had been experimentally confirmed.

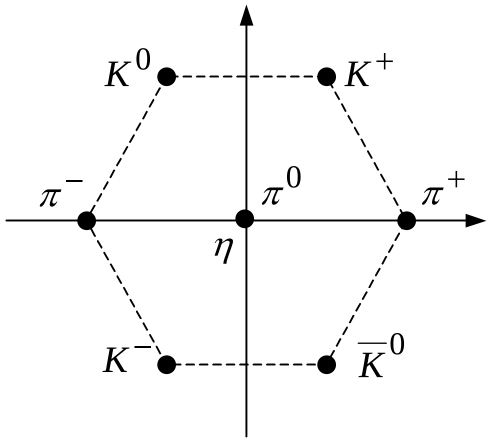


Figure 31: SU(3) octet showing the K meson family in I_z - Y space [10]

There are two potential sources for CP violation in the neutral kaon system, and both have been verified to occur experimentally. The first, indirect CP violation, occurs because the mass eigenstates differ from the CP eigenstates.

The K^0 and $\overline{K}^0$ are flavor eigenstates, but they are not mass eigenstates because they mix. They are also not CP eigenstates because the charge conjugation operator switches particle and antiparticle.

- CP operating on the flavor eigenstates is defined up to a phase: [35]

$$CP\left|K^0\right\rangle = \eta\left|\overline{K}^0\right\rangle \qquad CP\left|\overline{K}^0\right\rangle = \eta^*\left|K^0\right\rangle \qquad |\eta|^2 = 1$$

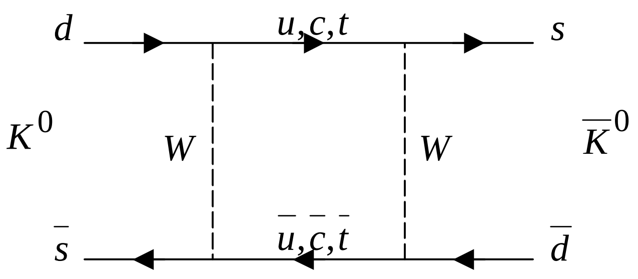


Figure 32: One of two box diagrams for neutral kaon mixing

- We can write down the CP eigenstates trivially if we choose η to be real:

$$|K_1\rangle = \frac{1}{\sqrt{2}}\left(\left|K^0\right\rangle + \left|\overline{K}^0\right\rangle\right)$$

$$|K_2\rangle = \frac{1}{\sqrt{2}}\left(\left|K^0\right\rangle - \left|\overline{K}^0\right\rangle\right)$$

- But if CP is not a conserved quantity, the mass eigenstates will be different: [21]

$$|K_L\rangle = \frac{1}{\sqrt{p^2+q^2}}\left(p\left|K^0\right\rangle + q\left|\overline{K}^0\right\rangle\right)$$

$$|K_S\rangle = \frac{1}{\sqrt{p^2+q^2}}\left(p\left|K^0\right\rangle - q\left|\overline{K}^0\right\rangle\right)$$

K_L and K_S are the long and short lived neutral kaons, respectively. These are the particles with definite mass and lifetime, and are thus the mass eigenstates. Schrödinger's equation is, in general, a matrix equation that happens to be diagonal in the mass basis. This fact can be used to relate the unknown matrix elements in the flavor basis to the measured masses and lifetimes of K_L and K_S.

- Schrödinger's equation has dispersive and absorptive parts: [21]

$$i\frac{d}{dt}|\Psi(t)\rangle = \begin{pmatrix} M_{11} - \frac{i}{2}\Gamma_{11} & M_{12} - \frac{i}{2}\Gamma_{12} \\ M_{12}^* - \frac{i}{2}\Gamma_{12}^* & M_{11} - \frac{i}{2}\Gamma_{11} \end{pmatrix}|\Psi(0)\rangle \qquad \text{flavor basis}$$

$$i\frac{d}{dt}|\Psi(t)\rangle = \begin{pmatrix} m_S - \frac{i}{2}\Gamma_S & 0 \\ 0 & m_L - \frac{i}{2}\Gamma_L \end{pmatrix}|\Psi(0)\rangle \qquad \text{mass basis}$$

Experimentally, we know that CP violation is very small; this is reflected in the matrix elements.

- We can place limits on some of the matrix elements: [21]

$$|\mathrm{Im}\,M_{12}| << |\mathrm{Re}\,M_{12}|$$

$$|\mathrm{Im}\,\Gamma_{12}| << |\mathrm{Re}\,\Gamma_{12}|$$

$$|\mathrm{Im}\,\Gamma_{12}| << |\mathrm{Im}\,M_{12}|$$

These approximations allow us to obtain a number of useful relationships when we diagonalize the matrix for the flavor basis and compare it to the mass basis.

$$p = \sqrt{M_{12} - \frac{i}{2}\Gamma_{12}} \qquad q = \sqrt{M_{12}^* - \frac{i}{2}\Gamma_{12}^*}$$

$$\frac{q}{p} = \sqrt{\frac{M_{12}^* - \frac{i}{2}\Gamma_{12}^*}{M_{12} - \frac{i}{2}\Gamma_{12}}} = \sqrt{\frac{1 - \dfrac{i\,\mathrm{Im}\,M_{12} + \frac{1}{2}\mathrm{Im}\,\Gamma_{12}}{\mathrm{Re}\,M_{12} - \frac{i}{2}\mathrm{Re}\,\Gamma_{12}}}{1 + \dfrac{i\,\mathrm{Im}\,M_{12} + \frac{1}{2}\mathrm{Im}\,\Gamma_{12}}{\mathrm{Re}\,M_{12} - \frac{i}{2}\mathrm{Re}\,\Gamma_{12}}}} \approx 1 - \frac{i\,\mathrm{Im}\,M_{12} + \frac{1}{2}\mathrm{Im}\,\Gamma_{12}}{\mathrm{Re}\,M_{12} - \frac{i}{2}\mathrm{Re}\,\Gamma_{12}}$$

$$pq = \sqrt{\left(M_{12} - \frac{i}{2}\Gamma_{12}\right)\left(M_{12}^* - \frac{i}{2}\Gamma_{12}^*\right)} = \sqrt{\left|M_{12}\right|^2 - \left|\frac{\Gamma_{12}}{2}\right|^2 - i\,\mathrm{Re}\left(M_{12}\Gamma_{12}^*\right)}$$

$$\approx \mathrm{Re}\,M_{12} + \frac{i}{2}\mathrm{Re}\,\Gamma_{12}$$

$$\Delta m \equiv m_L - m_S = 2\,\mathrm{Re}\,pq \approx 2\,\mathrm{Re}\,M_{12} \qquad \Delta\Gamma \equiv \Gamma_L - \Gamma_S = 4\,\mathrm{Im}\,pq \approx 2\,\mathrm{Re}\,\Gamma_{12}$$

Figure 33: Relationships and approximations for the flavor basis matrix elements [21]

- Define a parameter ε that is a measure of indirect CP violation: [21]

$$\varepsilon = \frac{p-q}{2p} = \frac{1}{2}\left(1 - \frac{q}{p}\right) \approx \frac{i\,\mathrm{Im}\,M_{12} + \frac{1}{2}\mathrm{Im}\,\Gamma_{12}}{2\,\mathrm{Re}\,M_{12} - i\,\mathrm{Re}\,\Gamma_{12}} \approx \frac{i\,\mathrm{Im}\,M_{12} + \frac{1}{2}\mathrm{Im}\,\Gamma_{12}}{\Delta m - \frac{i}{2}\Delta\Gamma}$$

$$\approx \frac{i\,\mathrm{Im}\,M_{12}}{\Delta m - \frac{i}{2}\Delta\Gamma}$$

Using experimental values for Δm and $\Delta\Gamma$, we find that the phase of ε is very close to $\frac{\pi}{4}$ [14]. The parameter ε represents the deviation of the CP eigenstates from the mass eigenstates. If CP were conserved, the eigenstates would be the same because the Hamiltonian would be simultaneously diagonalized in both bases. Thus ε is a measure of indirect CP violation.

It is also possible to have direct CP violation if the decay amplitudes are CP violating. Of particular interest are the two pion and three pion final states. If there were no CP violation, the two pion states $\pi^+\pi^-$ and $\pi^0\pi^0$ would both have the same CP as K_S. Similarly, the three pion states $\pi^+\pi^-\pi^0$ and $\pi^0\pi^0\pi^0$ would have the same CP as K_L. This is why K_L has the longer lifetime: kinematically there is far more phase space available in the two pion states than in the three pion. Because of CP violation, however, a small number of K_L are able to decay to two pions. By comparing the amplitude for these to the CP-allowed decays, we get dimensionless measures of direct CP violation.

- Dimensionless measures of CP violation for pion final states: [10]

$$\eta_{+-} = \frac{\left\langle \pi^+\pi^- \middle| T \middle| K_L \right\rangle}{\left\langle \pi^+\pi^- \middle| T \middle| K_S \right\rangle}$$

$$\eta_{00} = \frac{\left\langle \pi^0\pi^0 \middle| T \middle| K_L \right\rangle}{\left\langle \pi^0\pi^0 \middle| T \middle| K_S \right\rangle}$$

$$\varepsilon' = \frac{\eta_{+-} - \eta_{00}}{3}$$

The quantity $\mathrm{Re}\,\frac{\varepsilon'}{\varepsilon}$ is what is generally measured by experiment.

FCNC in the Standard Model and beyond

In the Standard Model, the quark weak eigenstates are not the same as the mass eigenstates. We can choose to take the up-type quarks as being the same in both the weak and mass bases, while the down-type quarks are related by the Cabibbo-Kobayashi-Maskawa (CKM) mixing matrix.

- The CKM matrix converts the mass eigenstates to the weak eigenstates:
$$\begin{pmatrix} d' \\ s' \\ b' \end{pmatrix} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} \begin{pmatrix} d \\ s \\ b \end{pmatrix}$$

- The signed weak interaction depends on weak eigenstate currents:
$$\mathcal{L}_W = \frac{g}{\sqrt{2}}\left(W_\mu^- J^{\mu+} + W_\mu^+ J^{\mu-}\right)$$
$$J^{\mu-} = \bar{u}'_{iL}\gamma^\mu d'_{iL} = V_{ij}\bar{u}_{iL}\gamma^\mu d_{jL}$$

- The down-type neutral weak interaction is unaffected by the CKM matrix:
$$\mathcal{L}_Z = \frac{e}{\sin\theta_W \cos\theta_W}\bar{d}'_{iL} I_3 \gamma^\mu d'_{iL} = \frac{e}{\sin\theta_W \cos\theta_W}\left(-\frac{1}{2}\bar{d}_{iL}\gamma^\mu d_{iL}\right)$$

Both the up-type and down-type neutral weak interactions remain diagonal in both the weak and mass bases, while the signed weak interaction picks up a CKM matrix element in the mass basis. As a result, the Standard Model does not allow tree-level Flavor Changing Neutral Currents (FCNC). It is possible, however, to have an effective FCNC at higher order through penguin and box diagrams.

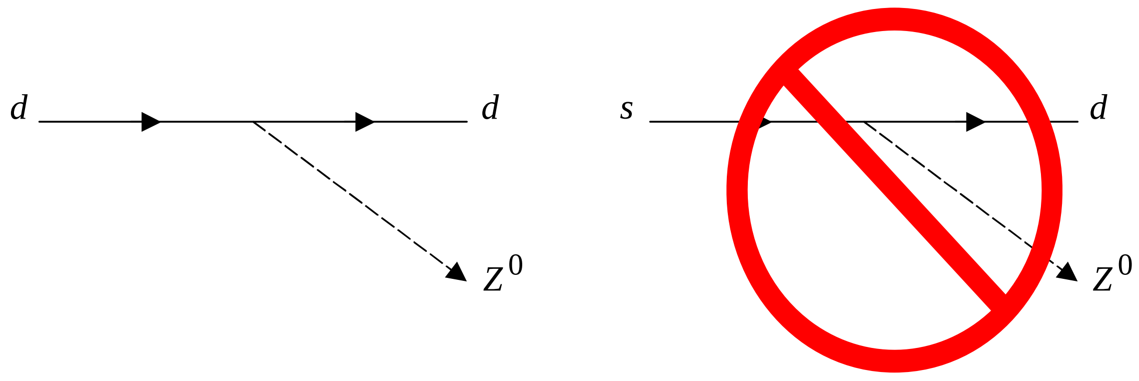


Figure 34: Quark flavor cannot change in a Standard Model neutral interaction

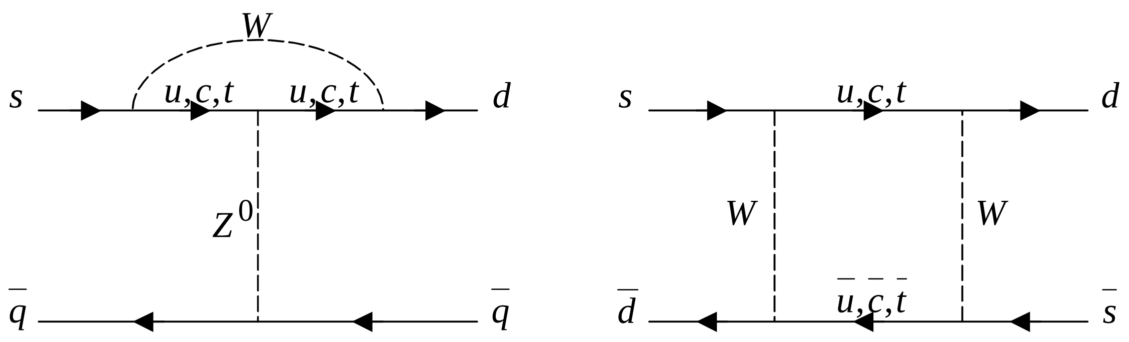


Figure 35: The Standard Model does allow FCNC through penguin (left) and box (right) diagrams

Inspired by some beyond the Standard Model (BSM) theories such as E_6 [27], we can add a single down-type quark that is a singlet under the weak interaction. Such an isosinglet down quark would mix with other down-type quarks in a 4x4 extension to the CKM matrix [9]. This is called the Four Down Quark Model (FDQM).

- Approximate version of the 4x4 extension to the CKM matrix: [17]

$$V = \begin{pmatrix} c_{12}c_{34} & s_{12}c_{34} & s_{13}e^{-i\delta_{13}} & s_{14}e^{-i\delta_{14}} \\ -s_{12} & 1 & s_{23} & s_{24}e^{-i\delta_{24}} \\ \left(s_{12}s_{23} - s_{13}e^{i\delta_{13}}\right) & -s_{23} & 1 & s_{34} \\ V_{4d} & V_{4s} & V_{4b} & c_{34} \end{pmatrix}$$

Unlike the regular down-type quarks, the isosinglet down quark has $I_3 = 0$. This changes the down-type neutral weak interaction.

- The down-type neutral weak interaction in the FDQM:

$$\mathcal{L}_Z = \frac{e}{\sin\theta_W \cos\theta_W}\left(-\frac{1}{2}\bar{d}'_{iL}\gamma^{\mu}d'_{iL} + \frac{1}{2}\bar{d}'_{4L}\gamma^{\mu}d'_{4L}\right)$$
$$= \frac{e}{\sin\theta_W \cos\theta_W}\left(-\frac{1}{2}\bar{d}_{iL}\gamma^{\mu}d_{iL} + \frac{1}{2}\bar{d}_{iL}V_{4i}^{*}V_{4j}\gamma^{\mu}d_{jL}\right)$$

We now have a non-diagonal term that produces tree-level FCNC. The quantity $V_{4i}^{*}V_{4j}$ determines the strength of these FCNC.

- Define coefficients of FCNC in the FDQM: [37]

$$U_{ij} = -V_{4i}^{*}V_{4j}$$

It is most convenient to define effective vertices when working with FCNC in the Standard Model. These can be compared to the FDQM to find an interesting relationship.

$$Box(\Delta S = 2) = \lambda_i^2 \frac{G_F^2}{16\pi^2} M_W^2 S_0(x_i)(\bar{s}d)_{V-A}(\bar{s}d)_{V-A}$$

$$Box\left(T_3 = -\frac{1}{2}\right) = \lambda_i \frac{G_F}{\sqrt{2}} \frac{\alpha}{2\pi \sin^2\theta_W} B_0(x_i)(\bar{s}d)_{V-A}(\bar{\mu}\mu)_{V-A}$$

$$Box\left(T_3 = \frac{1}{2}\right) = \lambda_i \frac{G_F}{\sqrt{2}} \frac{\alpha}{2\pi \sin^2\theta_W} [-4B_0(x_i)](\bar{s}d)_{V-A}(\bar{\nu}\nu)_{V-A}$$

$$\bar{s}Zd = i\lambda_i \frac{G_F}{\sqrt{2}} \frac{e}{2\pi^2} M_Z^2 \frac{\cos\theta_W}{\sin\theta_W} C_0(x_i)\bar{s}\gamma_\mu(1-\gamma_5)d$$

$$\bar{s}\gamma d = -i\lambda_i \frac{G_F}{\sqrt{2}} \frac{e}{8\pi^2} D_0(x_i)\bar{s}\left(q^2\gamma_\mu - q_\mu \not{q}\right)(1-\gamma_5)d$$

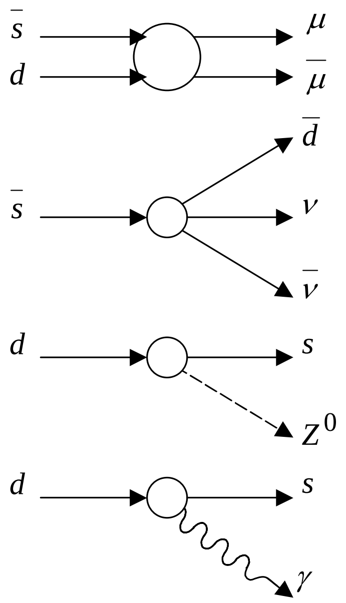


Figure 36: Effective vertices for FCNC, $\lambda_i = V_{is}^* V_{id}$ [12]

- FCNC term for s and d quarks in the FDQM Lagrangian: [17]

$$\mathcal{L}_{FCNC_{sd}} = \frac{-e}{\sin\theta_W \cos\theta_W}\left(\frac{1}{2}\bar{s}_L U_{sd}\gamma^\mu d_L\right) \propto U_{sd}\,\bar{s}_L\gamma^\mu d_L$$

- Weak penguin effective vertex from above (top quark):

$$\bar{s}Zd_{top} = i\lambda_t \frac{G_F}{\sqrt{2}} \frac{e}{2\pi^2} M_Z^2 \frac{\cos\theta_W}{\sin\theta_W} C_0(x_t)\bar{s}\gamma_\mu(1-\gamma_5)d$$
$$\propto \lambda_t C_0(x_t)\bar{s}_L\gamma_\mu d_L$$

Because these two terms differ only in the constants out front, it is possible to modify any result that includes $\lambda_t C_0(x_t)$ so that it also includes the FDQM contribution to FCNC.

- Modification to include FDQM in FCNC results:

$$\lambda_t C_0(x_t) \to \lambda_t C_0(x_t) + \frac{\pi^2}{\sqrt{2}G_F M_W^2} U_{sd}$$

This makes things much easier as many results have already been calculated by Buras and given in term of the basic functions ($B_0(x_t), C_0(x_t)$, etc) that were calculated by Inami and Lim. [28]

One challenge in obtaining the above formula is the varying sign conventions and constant definitions used in phenomenology literature. In particular, the above result for $\mathcal{L}_{FCNC_{sd}}$ has a minus sign that is due to a different sign convention, while the vertex is really $i\mathcal{L}$. This is critical to understand in order to obtain the correct replacement. Even considering this issue, papers by Buras and other authors quote a different formula.

- Commonly seen version of the standard FDQM replacement formula:

$$\lambda_t C_0(x_t) \to \lambda_t C_0(x_t) + \frac{\pi^2}{\sqrt{2} G_F M_W^2} U_{ds} = \lambda_t C_0(x_t) + \frac{\pi^2}{\sqrt{2} G_F M_W^2} U_{sd}^*$$

This differs by a sign change in $\mathrm{Im} U_{ds}$. Because U_{ds} is essentially an independent and unknown quantity this difference has little effect except when comparing results between authors, so we will use the more common version that has U_{ds}.

Minus convention: $D_\mu = \partial_\mu - ig\vec{\tau} \cdot \vec{W}_\mu - ig' \frac{Y}{2} B_\mu$

Plus convention: $D_\mu = \partial_\mu + ig\vec{\tau} \cdot \vec{W}_\mu + ig' \frac{Y}{2} B_\mu$

Mixed convention: $D_\mu = \partial_\mu - ig_2\vec{\tau} \cdot \vec{W}_\mu + ig' \frac{Y}{2} B_\mu$

Uses $\frac{Y}{2}$ as shown above: $Q = I_3 + \frac{Y}{2}$

Uses Y in place of $\frac{Y}{2}$: $Q = I_3 + Y$

$e > 0$: charge of particle $= Qe$

$e < 0$: charge of particle $= Q|e| = -Qe$

Figure 37: Common places where sign conventions and constant definitions can differ

The CKM program

The basis for this work is a FORTRAN program originally developed by Dr. Dennis Silverman for doing chi-squared analysis of the CKM matrix in the FDQM. Fourteen experiments are included in the computation of chi-squared for over 100 billion combinations of angles and phases in the extended CKM matrix [25]. To combine the chi-squared for Gaussian distributions with a Poisson distribution for a few events, we must use a different formula and add $2N_{events}$ degrees of freedom. [38]

- Chi-squared formula:
$$\chi^2 = \frac{\left(x_{exp} - x_{theory}\right)^2}{\sigma_{exp}^2 + \sigma_{theory}^2}$$
- Chi-squared formula for $Br\left(K^+ \to \pi^+ \nu\bar{\nu}\right)$:
$$\chi^2 = 2\frac{Br_{theory}}{Br_{exp}}$$

The output of the program is a series of chi-squared contour plots showing what regions of various variables are allowed and not allowed. [25]

Four new K physics experiments were added to this program. Each one produces an equation for the result and its sigma as a function of the angles and phases of the extended CKM matrix. These equations are computed for each set of angles/phases and added into the total chi-squared to contribute to the contour plots. Any dependencies not included in the angles/phases are plugged in manually and their uncertainties become part of the sigma function.

```
C Experiment #8: epsilon
C Some premultiplies
        ilamtrlamt = ilamt*rlamt
        ilamcrlamt = ilamc*rlamt
        rlamcilamt = rlamc*ilamt
        ilamcrlamc = ilamc*rlamc
        Q(8)  = -43592.4*(ilamtrlamt)
     $         -  32.832*(ilamcrlamt+rlamcilamt)
     $         - 10.8379*(ilamcrlamc)
        S2_Q8 = 4.91393e7*((ilamtrlamt)**2)
     $         +  67462.2*ilamtrlamt*(rlamcilamt+ilamcrlamt)
     $         +  22266.1*ilamcrlamc*ilamtrlamt
     $         +  81.9506*(((rlamcilamt)**2)+((ilamcrlamt)**2))
     $         +  53.0295*ilamcrlamc*(rlamcilamt+ilamcrlamt)
     $         +  11.9803*((ilamcrlamc)**2)
        sum = sum + ((Q(8)-P(8))**2)/(abs(S2_Q8)+S(8)**2)
        if(sum.gt.default_value) goto 101
        count(8)=count(8)+1
```

Figure 38: Snippet of the FORTRAN code for the added K physics experiments

Theory

Calculation of ε

We start from [14], equation 3.22, which gives ε in terms of M_{12}. We drop the $\frac{\pi}{4}$ phase and ignore the second term in ε as negligible. Combining with equation 3.37 gives an equation for ε in terms of basic functions. These basic functions can be found in [12], equations 3.17 through 3.19.

- [14], equation 3.22:

$$\varepsilon = \frac{e^{\frac{i\pi}{4}}}{\sqrt{2}\Delta M_K}\left(\operatorname{Im} M_{12} + 2\frac{\operatorname{Im} A_0}{\operatorname{Re} A_0}\operatorname{Re} M_{12} \right)$$

- [14], equation 3.37:

$$M_{12} = \frac{G_F^2}{12\pi^2} F_K^2 B_K m_{K^0} m_W^2 \times \left[\lambda_c^{*2}\eta_1 S_0(x_c) + \lambda_t^{*2}\eta_2 S_0(x_t) + 2\lambda_c^* \lambda_t^* \eta_3 S_0(x_c, x_t)\right]$$

- [12], equations 3.17 through 3.19:

$$S_0(x_c) = x_c$$

$$S_0(x_t) = \frac{4x_t - 11x_t^2 + x_t^3}{4(1-x_t)^2} - \frac{3x_t^3 \ln x_t}{2(1-x_t)^3}$$

$$S_0(x_c, x_t) = x_c\left[\ln\frac{x_t}{x_c} - \frac{3x_t}{4(1-x_t)} - \frac{3x_t^2 \ln x_t}{2(1-x_t)^2}\right]$$

- Our theoretical value for ε, SM (without the $\frac{\pi}{4}$ phase):

$$\varepsilon_{SM} = \frac{1}{\sqrt{2}\Delta M_K}\operatorname{Im} M_{12}$$

This is the contribution due to box and double penguin $\Delta S = 2$ diagrams. Wherever you have a double penguin, you can also get a double FCNC diagram in the FDQM. This contribution is obtained from the second term of [17], equation 45, by replacing their f_K^2 with $2F_K^2$.

- [17], equation 45:

$$|\varepsilon| = \frac{G_F f_K^2 B_K m_K}{12\Delta m_K}\left[\frac{\alpha}{4\pi \sin^3 \theta_W}\mathrm{Im}\left(-\tilde{E}^*\right) + \mathrm{Im}U_{ds}^2\right]$$

- Our theoretical value for ε, BSM (without the $\frac{\pi}{4}$ phase)

$$\varepsilon_{BSM} = \frac{1}{\sqrt{2}\Delta M_K}\mathrm{Im}M_{12} + \frac{G_F F_K^2 B_K m_K}{6\Delta m_K}\mathrm{Im}U_{ds}^2$$

We use the following values with the errors added in quadrature.

- $m_c = 1.25 \pm 0.15\,\mathrm{GeV}$ [15]
- $m_t = 165 \pm 5\,\mathrm{GeV}$ [14]
- $B_K = 0.85 \pm 0.13$ [39]
- $\eta_1 = 1.38 \pm 0.20$ [14]
- $\eta_2 = 0.57 \pm 0.01$ [14]
- $\eta_3 = 0.47 \pm 0.04$ [14]

We use the following values as exact.

- $G_F = 1.166 \times 10^{-5}\,\mathrm{GeV}^{-2}$ [15]
- $F_K = 0.1598\,\mathrm{GeV}$ [15]
- $m_{K^0} = 0.497672\,\mathrm{GeV}$ [15]
- $m_W = 80.41\,\mathrm{GeV}$ [15]
- $\Delta M_K = 3.489 \times 10^{-15}\,\mathrm{GeV}$ [15]

The resulting formula for ε is a function of λ_c, λ_t, U_{ds}, and their corresponding uncertainties. This result is compared to an experimental value from [14], equation 3.39, of $(2.280 \pm 0.013) \times 10^{-3}$ (dropping the common $\frac{\pi}{4}$ phase).

Calculation of $\mathrm{Re}\frac{\varepsilon'}{\varepsilon}$

We use [14], equation 5.5, to get $\frac{\varepsilon'}{\varepsilon}$ in terms of $F_{\varepsilon'}$, and equations 5.16 through 5.18 to express $F_{\varepsilon'}$ in terms of basic functions. [14], table 5 gives the coefficients needed; we interpolate to determine the dependence of these on $\Lambda^{(4)}_{\overline{MS}}$. When the NDR and HV schemes give different results we take the average and use the difference as a 2σ error. The basic functions used are from [12], equations 3.11 through 3.25, as needed.

- [14], equation 5.5:

$$\frac{\varepsilon'}{\varepsilon} = \mathrm{Im}\lambda_t \; F_{\varepsilon'}$$

- [14], equations 5.16 through 5.18:

$$F_{\varepsilon'} = P_0 + P_X X_0(x_t) + P_Y Y_0(x_t) + P_Z Z_0(x_t) + P_E E_0(x_t)$$

$$P_i = r_i^{(0)} + r_i^{(8)} R_6 + r_i^{(8)} R_8$$

$$R_6 = B_6^{(1/2)}\left[\frac{137 MeV}{m_s(m_c) + m_d(m_c)}\right]^2$$

$$R_8 = B_8^{(3/2)}\left[\frac{137 MeV}{m_s(m_c) + m_d(m_c)}\right]^2$$

Here $r_i^{(j)}$ are the coefficients that depend on $\Lambda^{(4)}_{\overline{MS}}$ and the renormalization scheme used. $B_6^{(1/2)}$ and $B_8^{(3/2)}$ are "bag parameters" determined from lattice QCD simulations.

The form of equation 5.16 above includes an implicit approximation that $\frac{\varepsilon'}{\varepsilon}$ is real. We will therefore write this explicitly.

- Our theoretical value for $\mathrm{Re}\frac{\varepsilon'}{\varepsilon}$, SM:

$$\mathrm{Re}\frac{\varepsilon'}{\varepsilon}_{SM} = \mathrm{Im}\lambda_t \; F_{\varepsilon'}$$

	$\Lambda_{\overline{MS}}^{(4)} = 290\,\mathrm{MeV}$			$\Lambda_{\overline{MS}}^{(4)} = 340\,\mathrm{MeV}$			$\Lambda_{\overline{MS}}^{(4)} = 390\,\mathrm{MeV}$		
i	$r_i^{(0)}$	$r_i^{(6)}$	$r_i^{(8)}$	$r_i^{(0)}$	$r_i^{(6)}$	$r_i^{(8)}$	$r_i^{(0)}$	$r_i^{(6)}$	$r_i^{(8)}$
0	−2.771	9.779	1.429	−2.811	11.127	1.267	−2.849	12.691	1.081
X_0	0.532	0.017	0	0.518	0.021	0	0.506	0.024	0
Y_0	0.396	0.072	0	0.381	0.079	0	0.367	0.087	0
Z_0	0.354	−0.013	−9.404	0.409	−0.015	−10.230	0.470	−0.017	−11.164
E_0	0.182	−1.144	0.411	0.167	−1.254	0.461	0.153	−1.375	0.517
0	−2.749	8.596	1.050	−2.788	9.638	0.871	−2.825	10.813	0.669

Table 6: $r_i^{(j)}$ coefficient dependence on $\Lambda_{MS}^{(4)}$ [14]

To add the BSM contribution we use the standard replacement on the C_0 inside the gauge-independent basic functions X_0, Y_0, and Z_0.

- Buras [12], equations 3.23 through 3.28:

$$X_0(x_t) = C_0(x_t) - 4B_0(x_t) = \frac{x_t}{8}\left[\frac{x_t+2}{x_t-1} + \frac{3x_t-6}{(x_t-1)^2}\ln x_t\right]$$

$$Y_0(x_t) = C_0(x_t) - B_0(x_t) = \frac{x_t}{8}\left[\frac{x_t-4}{x_t-1} + \frac{3x_t}{(x_t-1)^2}\ln x_t\right]$$

$$Z_0(x_t) = C_0(x_t) + \frac{1}{4}D_0(x_t) = -\frac{1}{9}\ln x_t + \frac{18x_t^4 - 163x_t^3 + 259x_t^2 - 108x_t}{144(x_t-1)^3}$$

$$+\frac{32x_t^4 - 38x_t^3 - 15x_t^2 + 18x_t}{72(x_t-1)^4}\ln x_t$$

- Our theoretical value for $\mathrm{Re}\frac{\varepsilon'}{\varepsilon}$, BSM:

$$\mathrm{Re}\frac{\varepsilon'}{\varepsilon}_{BSM} = \mathrm{Im}\lambda_t\, F_{\varepsilon'} + \frac{\pi^2}{\sqrt{2}G_F m_W^2}\mathrm{Im}U_{ds}\left(P_x + P_y + P_z\right)$$

We use the following values with the errors added in quadrature.

- $m_d = 7 \pm 2\,\text{MeV}$ [14]
- $m_s = 130 \pm 25\,\text{MeV}$ [14]
- $m_t = 165 \pm 5\,\text{GeV}$ [14]
- $B_6^{(1/2)} = 1.0 \pm 0.3$ [14]
- $B_8^{(3/2)} = 0.8 \pm 0.2$ [14]
- $\Lambda_{\overline{MS}}^{(4)} = 340 \pm 50\,\text{GeV}$ [14]
- $r_0^{(i)}$ (from table 3, also depends on $\Lambda_{\overline{MS}}^{(4)}$)

We use the following values as exact.

- $r_X^{(i)}$, $r_Y^{(i)}$, $r_Z^{(i)}$, $r_E^{(i)}$ (from table 3, still depend on $\Lambda_{\overline{MS}}^{(4)}$)
- $G_F = 1.166 \times 10^{-5}\,\text{GeV}^{-2}$ [15]
- $m_W = 80.41\,\text{GeV}$ [15]

The resulting formula for $\text{Re}\frac{\varepsilon'}{\varepsilon}$ is a function of $\text{Im}\lambda_t$, $\text{Im}U_{ds}$, and their corresponding uncertainties. This result is compared to an experimental value from [29], figure 11, of 0.00172 ± 0.00018.

Calculation of $Br\left(K^+ \to \pi^+ \nu\bar{\nu}\right)$

We use [14], equation 6.7, for the branching ratio and add the isospin breaking correction in equations 6.8 and 6.9. The X_{NL} are interpolated from [13], table 1, for their m_c and $\Lambda^{(4)}_{\overline{MS}}$ dependencies. The basic functions used are from [12], equations 3.23 and 3.26, as needed. The NLO corrections to the basic functions are from [14], equation 6.2.

- [14], equation 6.7 with isospin correction r_K :

$$\frac{Br\left(K^+ \to \pi^+ \nu\bar{\nu}\right)}{Br\left(K^+ \to \pi^0 e^+ \nu\right)} = r_K \frac{\alpha_K^2}{2\pi^2 |V_{us}|^2 \sin^4 \theta_W} \sum_{l=e,\mu,\tau} \left|\lambda_c X_{NL}^l + \lambda_t X(x_t)\right|^2$$

- [12], equations 3.23 and 3.26:

$$X_0(x_t) = C_0(x_t) - 4B_0(x_t) = \frac{x_t}{8}\left[\frac{x_t+2}{x_t-1} + \frac{3x_t-6}{(x_t-1)^2}\ln x_t\right]$$

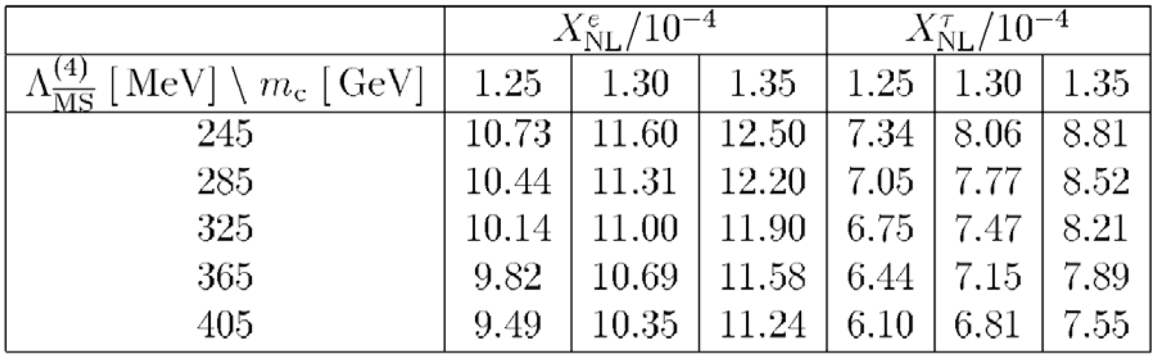
- [14], equation 6.2:

$$X(x_t) = \eta_X X_0(x_t) = 0.994 X_0(x_t)$$

Our theoretical value for $K^+ \to \pi^+ \nu\bar{\nu}$ branching ratio, SM:

$$Br\left(K^+ \to \pi^+ \nu\bar{\nu}\right)_{SM} = Br\left(K^+ \to \pi^0 e^+ \nu\right) \times$$

$$r_K \frac{\alpha_K^2}{2\pi^2 |V_{us}|^2 \sin^4 \theta_W} \sum_{l=e,\mu,\tau} \left|\lambda_c X_{NL}^l + \lambda_t X(x_t)\right|^2$$



	$X_{NL}^e/10^{-4}$			$X_{NL}^\tau/10^{-4}$		
$\Lambda^{(4)}_{\overline{MS}}$ [MeV] \ m_c [GeV]	1.25	1.30	1.35	1.25	1.30	1.35
245	10.73	11.60	12.50	7.34	8.06	8.81
285	10.44	11.31	12.20	7.05	7.77	8.52
325	10.14	11.00	11.90	6.75	7.47	8.21
365	9.82	10.69	11.58	6.44	7.15	7.89
405	9.49	10.35	11.24	6.10	6.81	7.55

Table 7: X_{NL} dependence on m_c and $\Lambda^{(4)}_{\overline{MS}}$ [13]

To add the BSM contribution we use the standard replacement on the C_0 inside the gauge-independent basic function X_0.

- Our theoretical value for $K^+ \to \pi^+ \nu\bar{\nu}$ branching ratio, BSM:

$$Br\left(K^+ \to \pi^+ \nu\bar{\nu}\right)_{BSM} = Br\left(K^+ \to \pi^0 e^+ \nu\right) \times$$

$$r_K \frac{\alpha_K^2}{2\pi^2 \left|V_{us}\right|^2 \sin^4 \theta_W} \sum_{l=e,\mu,\tau} \left| \lambda_c X_{NL}^l + \lambda_t X(x_t) + \frac{\pi^2}{\sqrt{2} G_F m_W^2} U_{ds} \right|^2$$

We use the following values with the errors added in quadrature.

- $m_c = 1.25 \pm 0.15\,\text{GeV}$ [15]
- $m_t = 165 \pm 5\,\text{GeV}$ [14]
- $\Lambda_{\overline{MS}}^{(4)} = 340 \pm 50\,\text{GeV}$ [14]
- $Br\left(K^+ \to \pi^0 e^+ \nu\right)_{\exp} = 0.0482 \pm 0.0006$ [15]

We use the following values as exact.

- $\alpha_K = \frac{1}{129}$ [14]
- $\sin\theta_W = \sqrt{0.23124}$ [15]
- $G_F = 1.166 \times 10^{-5}\,\text{GeV}^{-2}$ [15]
- $m_W = 80.41\,\text{GeV}$ [15]

The resulting formula for the branching ratio is a function of λ_c, λ_t, V_{us}, U_{ds}, and their corresponding uncertainties. This result is compared to an experimental value from [32], below figure 6, of 1.5×10^{-10} with a Poisson error.

Calculation of $Br\left(K_L \to \mu^+\mu^-\right)_{SD}$

We use [11], equations 7.71 through 7.74, for the branching ratio. The Y_{NL} are interpolated from [13], table 3, for their m_c and $\Lambda^{(4)}_{\overline{MS}}$ dependencies. The basic functions used are from [12], equations 3.11 through 3.25, as needed. The NLO corrections to the basic functions are from [13], equation 19.

- [11], equations 7.71 through 7.74:

$$Br\left(K_L \to \mu^+\mu^-\right)_{SD} = \frac{\kappa_\mu}{\left|V_{us}\right|^{10}}\left[\mathrm{Re}\lambda_c\, Y_{NL} + \mathrm{Re}\lambda_t\, Y(x_t)\right]^2$$

$$\kappa_\mu = Br\left(K^+ \to \mu^+\nu\right)\frac{\tau_{K_L}}{\tau_{K^+}}\frac{\alpha_K^2\left|V_{us}\right|^8}{\pi^2\sin^4\theta_W}$$

- [12], equations 3.24 and 3.27:

$$Y_0(x_t) = C_0(x_t) - B_0(x_t) = \frac{x_t}{8}\left[\frac{x_t-4}{x_t-1} + \frac{3x_t}{(x_t-1)^2}\ln x_t\right]$$

- [13], equation 19:

$$Y(x_t) = \eta_Y Y_0(x_t) = 1.012\, Y_0(x_t)$$

- Our theoretical value for $K_L \to \mu^+\mu^-$ branching ratio, SM:

$$Br\left(K_L \to \mu^+\mu^-\right)_{SM} = Br\left(K^+ \to \mu^+\nu\right)\times$$
$$\frac{\tau_{K_L}}{\tau_{K^+}}\frac{\alpha_K^2}{\pi^2\left|V_{us}\right|^2\sin^4\theta_W}\left[\mathrm{Re}\lambda_c\, Y_{NL} + \mathrm{Re}\lambda_t\, Y(x_t)\right]^2$$

To add the BSM contribution we use the standard replacement on the C_0 inside the gauge-independent basic function Y_0.

- Our theoretical value for $K_L \to \mu^+\mu^-$ branching ratio, BSM:

$$Br\left(K_L \to \mu^+\mu^-\right)_{BSM} = Br\left(K^+ \to \mu^+\nu\right)\times$$
$$\frac{\tau_{K_L}}{\tau_{K^+}}\frac{\alpha_K^2}{\pi^2\left|V_{us}\right|^2\sin^4\theta_W}\left[\mathrm{Re}\lambda_c\, Y_{NL} + \mathrm{Re}\lambda_t\, Y(x_t) + \frac{\pi^2}{\sqrt{2}G_F m_W^2}\mathrm{Re}U_{ds}\right]^2$$

	$Y_{\mathrm{NL}}/10^{-4}$			$P_0(Y)$		
$\Lambda_{\overline{\mathrm{MS}}}^{(4)}$ [MeV] \ m_c [GeV]	1.25	1.30	1.35	1.25	1.30	1.35
245	2.75	2.94	3.13	0.117	0.125	0.134
285	2.80	2.99	3.19	0.119	0.128	0.136
325	2.84	3.04	3.24	0.121	0.130	0.138
365	2.89	3.09	3.29	0.123	0.132	0.140
405	2.92	3.13	3.33	0.125	0.133	0.142

Table 8: Y_{NL} dependence on m_c and $\Lambda_{\overline{MS}}^{(4)}$ [13]

We use the following values with the errors added in quadrature.

- $m_c = 1.25 \pm 0.15\,\mathrm{GeV}$ [15]
- $m_t = 165 \pm 5\,\mathrm{GeV}$ [14]
- $\Lambda_{\overline{MS}}^{(4)} = 340 \pm 50\,\mathrm{GeV}$ [14]
- $Br\left(K^+ \to \mu^+ \nu\right)_{\exp} = 0.6351 \pm 0.0018$ [15]

We use the following values as exact.

- $\tau_{K_L} = 5.17 \times 10^{-8}\,\mathrm{s}$ [15]
- $\tau_{K^+} = 1.2386 \times 10^{-8}\,\mathrm{s}$ [15]
- $\alpha_K = \frac{1}{129}$ [14]
- $\sin\theta_W = \sqrt{0.23124}$ [15]
- $G_F = 1.166 \times 10^{-5}\,\mathrm{GeV}^{-2}$ [15]
- $m_W = 80.41\,\mathrm{GeV}$ [15]

The resulting formula for the branching ratio is a function of λ_c, λ_t, V_{us}, U_{ds}, and their corresponding uncertainties. This result is compared to an experimental value of $(0 \pm 4.01) \times 10^{-5}$. [39]

Error analysis

Finding a nominal value isn't very difficult, although there are some complexities. However, that value is meaningless without a corresponding uncertainty. In general, errors can be computed in the traditional way for a function $f(x_i)$ of variables x_i with independent errors σ_{x_i}.

- Uncertainty in a general function f of multiple independent variables:

$$\sigma_{f(x_i)} = \sqrt{\sum_i \left(\frac{\partial f}{\partial x_i} \sigma_{x_i} \right)^2}$$

To make it easier to perform the computation and avoid mistakes, the algebra for computing this σ can be automated using *Mathematica*. To do this, a temporary function of a single variable is created for each x_i. The square of the derivative of this function times the corresponding sigma is added into a running total and returned as the result.

```
sigma[ func_, arg_, sig_ ] :=
  Block[ { s, i, l, f, a },
    s = 0;
    For[ i = 1, i <= Length[ arg ], i++,
      l = arg;
      f[ x_ ] :=
        Block[ {},
          l[[ i ]] = a;
          Apply[ func, l ] /. a->x ];
      s += ( f'[ arg[[ i ]] ] sig[[ i ]] )^2 ];
    Sqrt[ s ] ]
```

Figure 39: Mathematica code for the computation of errors

Some values must be computed by means other than a formula, in which case it is not immediately possible to compute a σ function as above. For example, the Penguin Box Expansion (PBE) coefficients are given in table 6. A linear interpolation is sufficient to describe the dependence on $\Lambda^{(4)}_{\overline{MS}}$. The renormalization scheme dependence would largely be cancelled by similar dependence in $B_6^{(1/2)}$ and $B_8^{(3/2)}$; however, those values are not well enough understood to separate that dependence from the overall errors [14]. Instead, an average value is used for $r_0^{(j)}$ with the difference between the two schemes treated as $2\sigma_{r_0^{(j)}}$.

Results

Confidence plots

The final output from the FORTRAN program is a series of confidence plots for various variables. Of particular interest for the K physics experiments are the $\rho - \eta$ plots. Of the four experiments added here, only the ε experiment helps constrain the $\rho - \eta$ plot in the Standard Model so the remaining experiments are best left out.

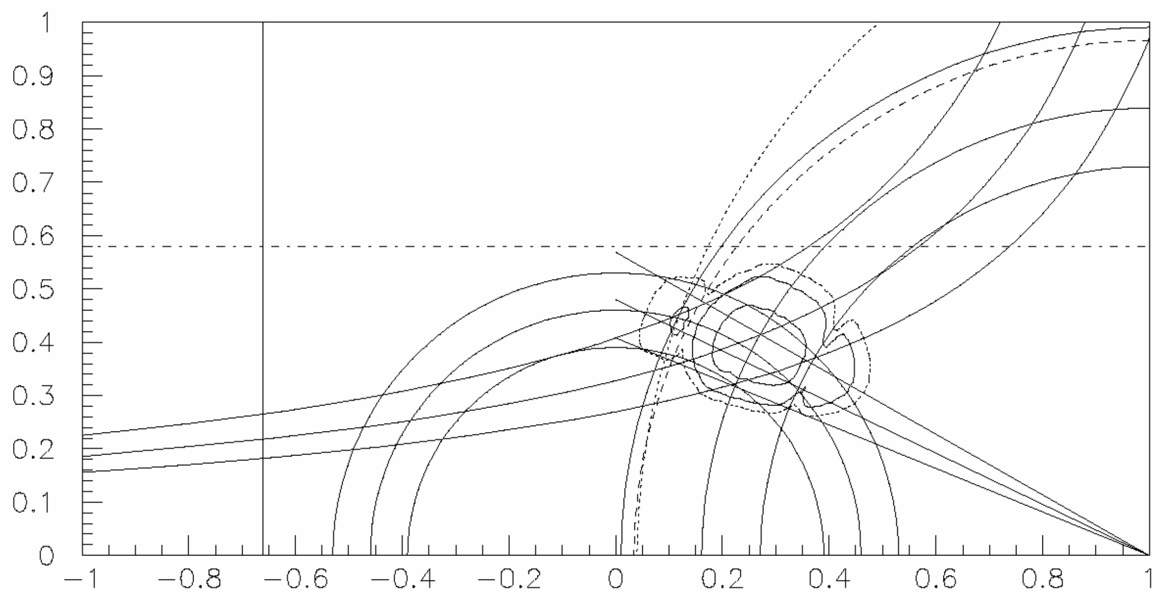


Figure 40: Plot of ρ vs. η in the Standard Model, including K physics experiments [25]

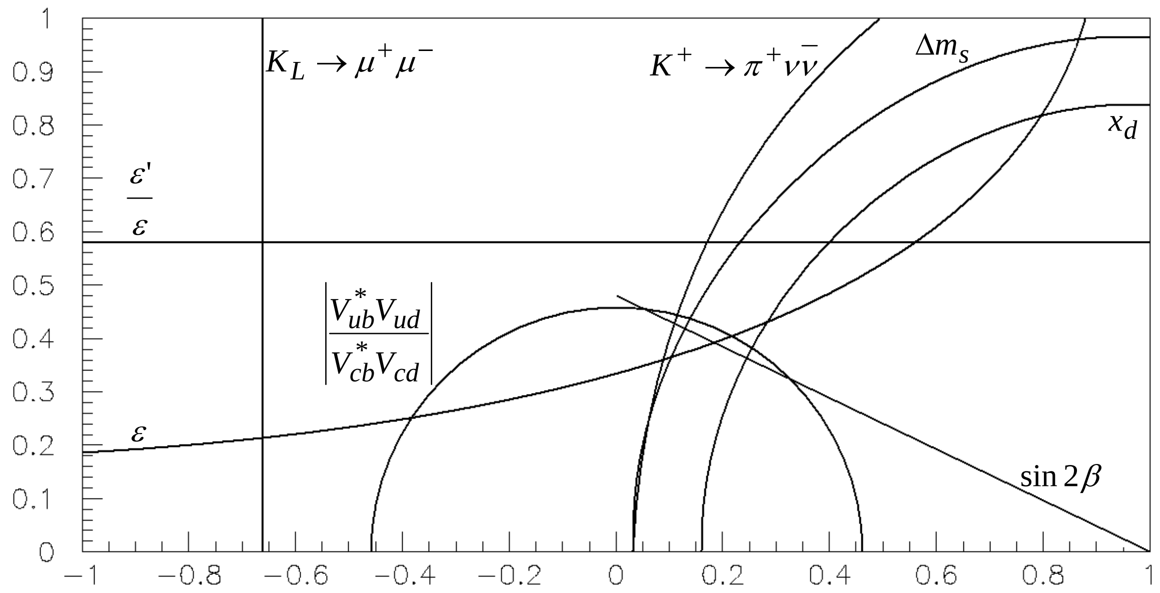


Figure 41: Part of same plot showing which experiments correspond to which lines

The situation is different in the FDQM because many of the standard experiments no longer provide the same constraints. For the $\rho - \eta$ plot, the annulus defined by the $\left|\frac{V_{ub}^{*}V_{ud}}{V_{cb}^{*}V_{cd}}\right|$ constraint is restricted to the upper half ($\eta > 0$) by $K^{+} \to \pi^{+}\nu\bar{\nu}$ and ε.

The phase angle in the $\rho - \eta$ plane (δ_{13}) is not otherwise constrained, although a more precise value of $\sin 2\alpha$ could lock down the value of η.

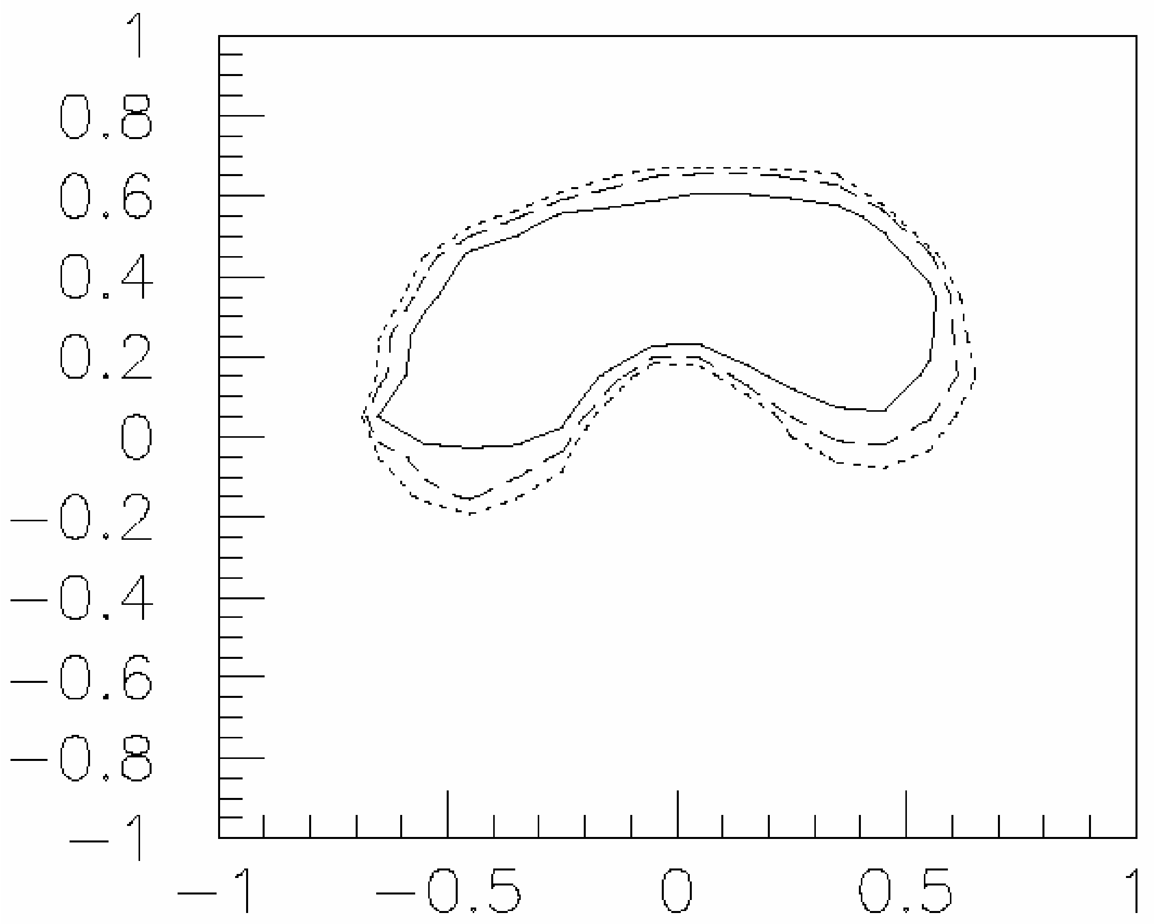


Figure 42: Plot of ρ vs. η in the FDQM [25]

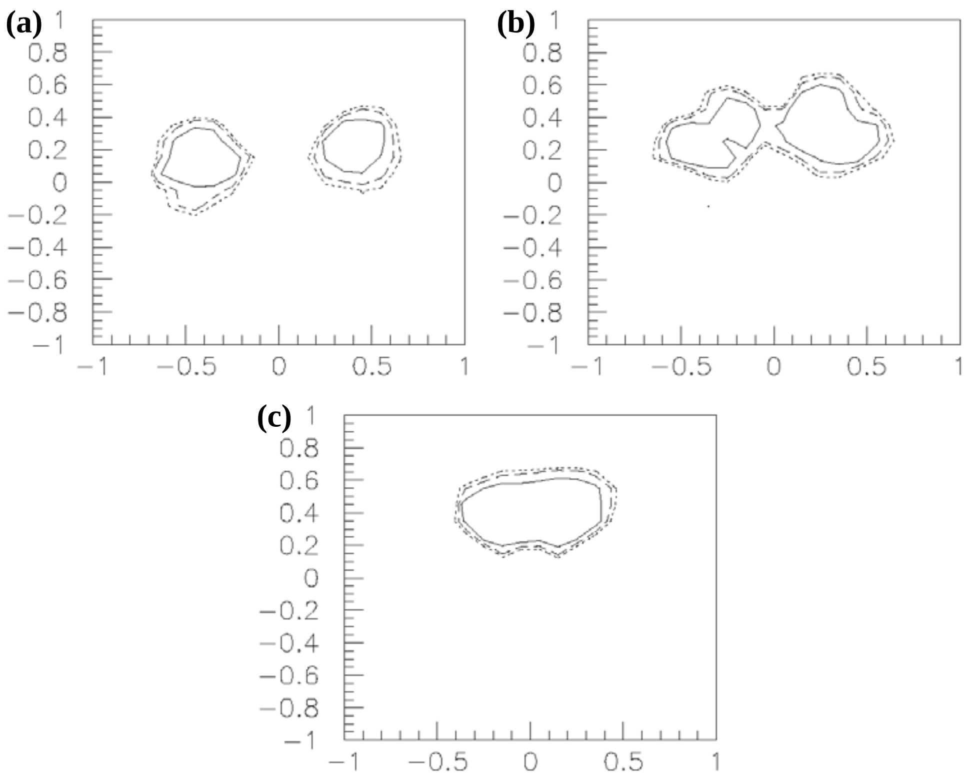


Figure 43: Plot of ρ **vs.** η **in the FDQM for (a)** $\sin 2\alpha = -1$**, (b)** $\sin 2\alpha = 0$**, (c)** $\sin 2\alpha = 1$ **[25]**

Along with the wider range of δ_{13} in the FDQM, the value of ε can now have a BSM contribution. While the contribution due to the FDQM can be as low as 0%, it can also be as high as 60% at 1σ.

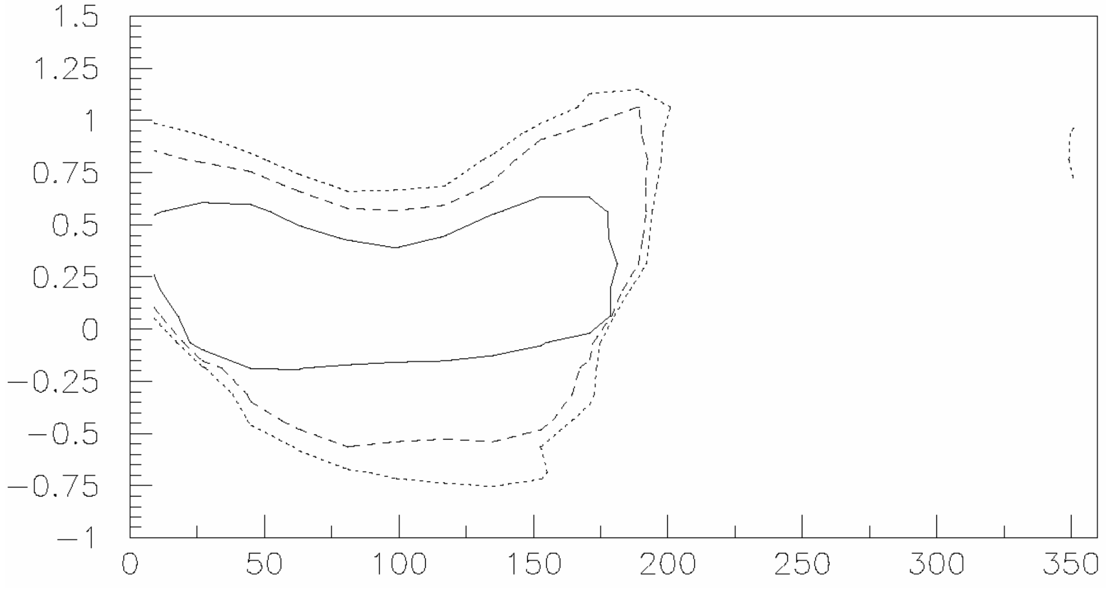


Figure 44: Plot of $\frac{\varepsilon_{BSM}}{\varepsilon_{\exp}}$ **vs.** δ_{13} **in the FDQM [25]**

Finally, we can look at the range of values allowed for $U_{sd} = U_{ds}^{*}$. The value $U_{sd} = 0$ is allowed, which means the Standard Model cannot be excluded by current experiments. The range of nonzero values for U_{sd} doesn't tell us anything about the likelihood of the FDQM being accurate though; as long as there is a nonzero difference between the Standard Model and experiment we will see a potential for a nonzero value of BSM quantities. Currently the error bars are close to 100% for many of the K physics experiments, so the contribution to CP violation due to BSM physics can be equal to the Standard Model contribution.

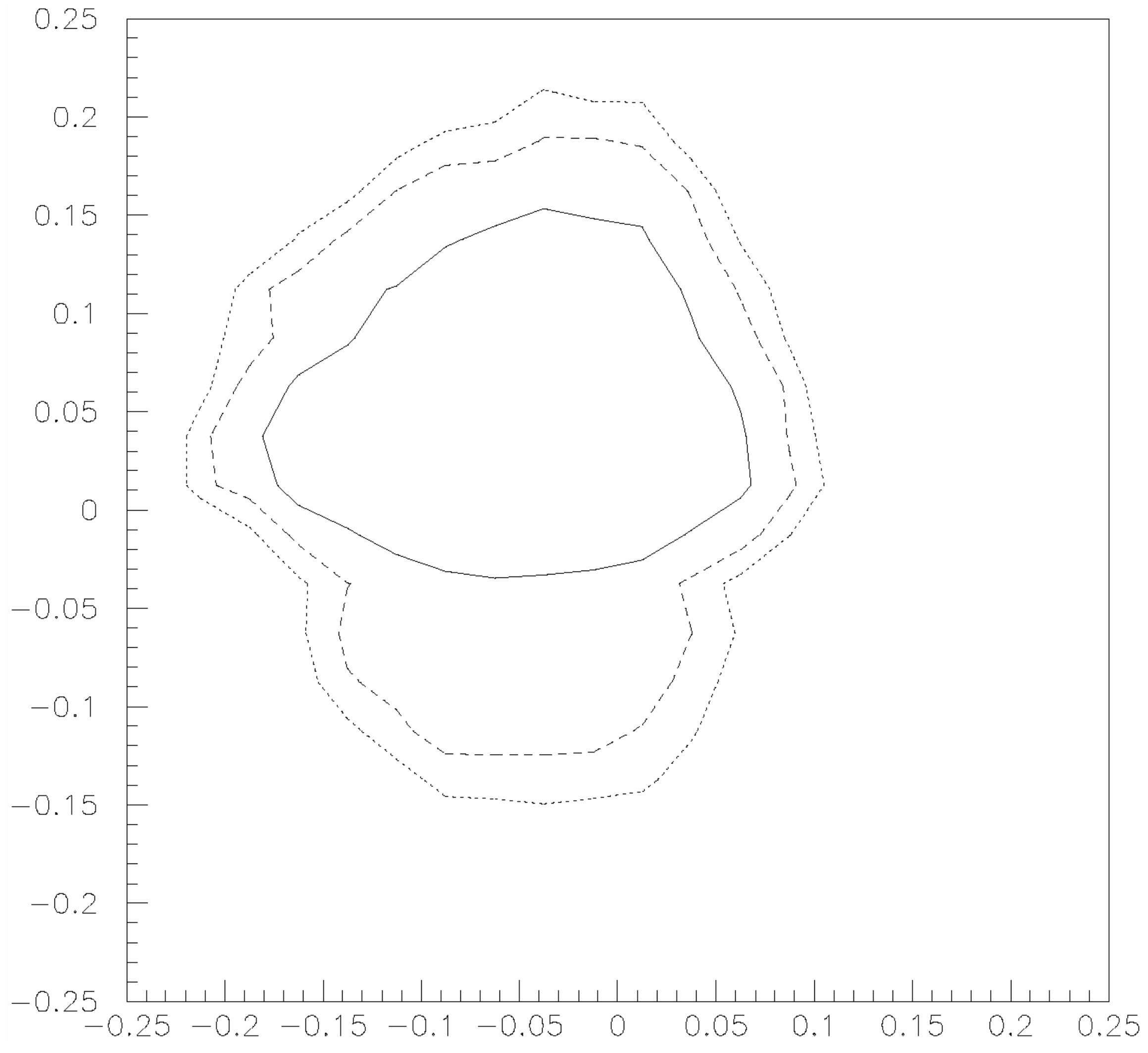


Figure 45: Plot of $\mathrm{Im}U_{sd}$ vs. $\mathrm{Re}U_{sd}$ (in units of 10^{-4}) [25]

Future work

Chi-squared analysis can never prove a theory, but it can be used to exclude theories and to suggest areas where better experimental results are need. For example, we can see that improved measurement of $\sin 2\alpha$ could be useful to restrict η. Such restrictions will gradually edge out invalid theories and home in on the consistent ones.

We have also seen that the addition of new experiments, even ones with large error bars, can be significant. In particular, the inclusion of $K^+ \to \pi^+ \nu \bar{\nu}$ is very important in the FDQM, although it provides too weak a constraint to be useful in the Standard Model. It is important to consider this and look carefully at early results from new detectors to see if they can be valuable even when the statistics are low.

The future of this analysis is largely dependent on future experimental advances. As new results come in, the analysis can be extended and revised to look at the effects of each experiment and suggest areas where improvement would be useful. In this way, phenomenology acts as a feedback bridge connecting theoretical results back to experimental effort and provides a measure for judging how well the two match.

PART III

Appendices

References

[1] ATLAS Collaboration: *ATLAS Detector and Physics Performance Technical Design Report*, CERN/LHCC/99-14, 25 May 1999

[2] ATLAS Level-1 Trigger Group: *ATLAS Level-1 Trigger Technical Design Report*, CERN/LHCC/98-14, 24 June 1998

[3] ATLAS Muon Collaboration: *ATLAS Muon Spectrometer Technical Design Report*, CERN/LHCC/97-22, 5 June 1997

[4] ATLAS Website: http://atlas.ch/

[5] ATLAS Website: http://atlas.ch/atlas_photos/lhc/lhc.html

[6] ATLAS Website: http://atlasexperiment.org/etours_exper/etours_exper07.html

[7] ATLAS Website: http://atlasexperiment.org/atlas_photos.html

[8] Austern, Matthew: *Generic Programming and the STL*, Addison-Wesley, 1999

[9] Botella, F. J. and Chau, L.-L.: *Anticipating the higher generations of quarks from rephasing invariance of the mixing matrix*, Phys. Lett. **B168**: 97, 27 February 1986

[10] Branco *et al.*: *CP Violation*, Oxford University Press, 1999

[11] Buras, Andrzej and Fleischer, Robert: *Quark Mixing, CP Violation and Rare Decays After the Top Quark Discovery*, hep-ph/9704376, 22 April 1997

[12] Buras, Andrzej: *Weak Hamiltonian, CP Violation and Rare Decays*, hep-ph/9806471, 24 June 1998

[13] Buras, Andrzej: *The Rare Decays* $K \to \pi\nu\bar{\nu}$, $B \to X\nu\bar{\nu}$, *and* $B \to l^+l^-$ *: An Update*, hep-ph/9901288, 13 January 1999

[14] Buras, Andrzej: *CP Violation and Rare Decays of K and B Mesons*, hep-ph/9905437, 21 May 1999

[15] Caso, C. *et al.*: *The Review of Particle Physics*, The European Physical Journal **C3**: 1, 1998

[16] CERN Website: http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhatIsCERN/WhatIsCERN-en.html

[17] Choong, Woon-Seng and Silverman, Dennis: *New phases in CP-violating B decay asymmetries from mixing to singlet down quarks*, Phys. Rev. **D49:** 2322, 1 March 1994

[18] Dailing, J. *et al.*: *Off-Detector Electronics for a High-Rate CSC Detector*, PROCEEDINGS of the Sixth Workshop on Electronics for LHC Experiments, 11 September 2000

[19] Dailing, J. *et al.*: *Performance and Radiation Tolerance of the ATLAS CSC On-Chamber Electronics*, PROCEEDINGS of the Sixth Workshop on Electronics for LHC Experiments, 11 September 2000

[20] Drego, N. *et al.*: *Off-Detector Electronics for High-Rate CSC Detector*, IEEE Transactions on Nuclear Science **51**: 461, June 2004

[21] Fayyazuddin and Riazuddin, *A Modern Introduction to Particle Physics*, World Scientific, 1992

[22] Gamma, Erich *et al.*: *Design Patterns*, Addison-Wesley, 1995

[23] GNU m4 website: http://www.gnu.org/software/m4/

[24] Gordeev, A. *et al.*: *CSC Performance at High Background Rates*, ATL-MUON-2000-005, 19 October 1999

[25] Hawkins, Donovan and Silverman, Dennis: *Isosinglet down quark mixing and CP violation experiments*, Phys. Rev. **D66**: 16008, 31 July 2002

[26] Hendriks, Patrick: *ATLAS Muon Reconstruction from a C++ Perspective*, Thesis: Amsterdam Univ., 26 April 2000

[27] Hewett, J. L. and Rizzo, T. G.: *Low-energy phenomenology of superstring-inspired E_6 models*, Phys. Rept. **183**: 193, November 1989

[28] Inami, Takeo and Lim, C. S.: *Effects of Superheavy Quarks and Leptons in Low-Energy Weak Processes* $K_L \to \mu\bar{\mu}$, $K^+ \to \pi^+\nu\bar{\nu}$, *and* $K^0 \leftrightarrow \overline{K^0}$, Progress of Theoretical Physics **65**: 297, January 1981

[29] Kessler, R.: Recent KTeV Results, hep-ex/0110020, 19 October 2001

[30] LHC Website: http://lhc-machine-outreach.web.cern.ch/lhc-machine-outreach/lhc-vital-statistics.htm

[31] LHC Website: http://sylvainw.home.cern.ch/sylvainw/planning-follow-up/Schedule.pdf

[32] Molzon, William: *Quark Mixing Matrix Studies and Lepton Flavor Violation Searches Using Rare Decays of Kaons*, hep-ex/0001024, 11 January 2000

[33] Pier, Steve: *IROD Architecture*, http://positron.ps.uci.edu/~pier/csc/IRODBlockDiagram9.pdf, 25 May 2001

[34] Press, William *et al.*: *Numerical Recipes in C*, Cambridge University Press, 1988

[35] Roe, Byron: *Particle Physics at the New Millennium*, Springer-Verlag, 1996

[36] Schernau, Michael: *CSC Drift Time Summary*, http://positron.ps.uci.edu/~schernau/ROD/drift.ps, 1 March 2001

[37] Shin, M. *et al.*: *Flavor changing neutral processes and* $B_d^0 - \overline{B_d^0}$ *mixing*, Phys. Lett. **B219**: 381, 16 March 1989

[38] Silverman, Dennis: *Joint Bayesian Treatment of Poisson and Gaussian Experiments in a Chi-squared Statistic*, http://xxx.lanl.gov/pdf/physics/9808004, 26 June 2002

[39] Silverman, Dennis: Private correspondences, 2000

[40] STLPort Website: http://www.stlport.org/

[41] Texas Instruments, *TMS320C6000 Code Generation Tools v4.10*, run-time library source code

[42] U.C. Irvine CSC ROD Group Website: http://positron.ps.uci.edu/~schernau/ROD/pix/

[43] U.C. Irvine CSC ROD Group Website: http://positron.ps.uci.edu/~schernau/ROD/SIT/results/rate/effi.html

Appendix A: ATLAS Muon CSC ROD Software

DPU documentation for the HPU

Boot

Before the DPU is released from reset, make sure that address 0x80000000 is set to 0 using DPU Control. This signals the presence of the HPU to the DPU when it boots. After boot, poll the same address looking for a non-zero value. This value will be the address of the Status struct. All other memory addresses of interest will be located in the Status struct. Verify that this value is a legal pointer in the range of DPU memory available to the HPU.

At this point, only a few items are available to the HPU, all of which should be verified:

- All bools
  Verify that the corresponding *DummyInteger* is set to 0xB00? (Status.h).
- *Status.Basic.RealHPU*
  Verify that it is set to *true*.
- *Status.Basic.RealDSP*
  Verify that it is set to *true*.
- *Status.Basic.Module*
  Verify that the module ID is in the range 0 to 11 inclusive.
- *Status.Basic.MajorVersion*
  Verify that it is set to *DPU_VERSION_MAJOR* (common.h).
- *Status.Basic.MinorVersion*
  Verify that it is set to *DPU_VERSION_MINOR* (common.h).
- *Status.Basic.BootStatus*
  Verify that it is set to *TStatus::bsBasic* (Status.h).
- *Status.Basic.Initialize*
  Verify that it is set to *false*.
- *Status.Basic.Calibration*
  Verify that it is a legal pointer in the range of DPU memory available to the HPU.
- *Status.Buffer.ParameterArray*
  Verify that it is a legal pointer in the range of DPU memory available to the HPU.
- *Status.Buffer.ParameterSize*
  Verify that it is a legal pointer in the range of DPU memory available to the HPU.
- *Status.Buffer.Response*
  Verify that it is a legal pointer in the range of DPU memory available to the HPU.
- *Status.Buffer.CIB*
  Verify that it is a legal pointer in the range of DPU memory available to the HPU.

Initialization

Generally only the items in *Status.Basic* and the buffers whose pointers lie in *Status.Basic* will be initialized. Those items are:

- *Status.Basic.Initialize*
  This item should be switched from *false* to *true* when all initialization is complete.

- *Status.Basic.BigInput*
  Setting this *true* causes the DIB to use 1/2 of memory and the DOB to use 1/4. Setting this *false* reverses the situation. In ATLAS, we will use *true*.
- *Status.Basic.FastInput*
  Setting this *true* causes the DIB to use a higher-priority DMA channel than the DOB. Setting this *false* reverses the situation. In ATLAS, we will use *true*.
- *Status.Basic.InputBuffer*
  This is an enum to specify the DIB input source. In ATLAS, we will use *TStatus::btXFPGA* for the SPU and *TStatus::btEFPGA* for the RPU.
- *Status.Basic.OutputBuffer*
  This uses the same enum as above to specify the DOB output destination. In ATLAS, both the SPU and RPU will use *TStatus::btEFPGA*.
- *Status.Basic.OutputInterrupt*
  This option is not currently available and should be set to *false*.
- *Status.Basic.EFPGA_FrameSize*
  The EFPGA must be told what Frame size to use for DPU DMA input (DPU acting as destination). If *Status.Basic.InputBuffer* is set to *TStatus::btEFPGA* then the EFPGA is configured using the Policy subsystem's *FrameSize*. Otherwise the value in this variable is used. A reasonable default is 16.
- *Status.Basic.DXMode*
  This is an enum to specify whether we use loopback mode in the EFPGA. In ATLAS, we will set this to *TStatus::dxNormal*.

- *Status.Basic.Verbose*
  Setting this *true* enables *printf* of diagnostic information by the DPU. This setting has no meaning if the DPU was compiled with *PRINT =OFF* (make.bat). In ATLAS, we will set this to *false*.
- *Status.Basic.CommandMode*
  This is an enum to select whether the CIB will be fed Default Decode Commands (used when the HPU is unable to send them). In ATLAS, we will set this to *TStatus::cmHPU*.
- *Status.Basic.InputData*
  This is an enum that selects the Policy set to be used for the DIB. In ATLAS, we will set this to *TStatus::itTimeslice* for the SPU and *TStatus::itSparsifiedData* for the RPU.

- *Status.Basic.DefaultDecoder*
  This is an enum that selects which Decoder is the default. In ATLAS, we will set this to *Data::DPU_SPUDecoder* for the SPU and *Data::DPU_RPUDecoder* for the RPU (decoders.h).
- *Status.Basic.DiscardDecoder*
  This is an enum that selects which Discard Decoder is used. In ATLAS, we will set this to *Data::DPU_SPUDiscardDecoder* for the SPU and *Data::DPU_RPUDiscardDecoder* for the RPU (decoders.h).
- *Status.Basic.OutputData*
  This is an enum that selects the Policy set to be used for the DOB. In ATLAS, we will set this to *TStatus::otSparsifiedData* for the SPU and *TStatus::otATLASData* for the RPU.
- *Status.Basic.PrimeInput*
  This allows us to pre-fill the input buffers with data. In ATLAS, we will set this to *false*.
- *Status.Basic.MaxInterruptCount*
  This option is not currently available and should be set to 0.
- *Status.Basic.MaxClockWarning*
  This is the number of clock cycles that a single pass through the main loop is allowed to take before a Warning is issued. In ATLAS, we will set this to 10000.
- *Status.Basic.MaxClockFault*
  This is the number of clock cycles that a single pass through the main loop is allowed to take before a Fault occurs.

- *Status.Basic.MaxTaskSize*
  This value determines the largest Task that may be created (in words). In ATLAS, we will set this to 0xFFFFFFFF.
- *Status.Basic.OptimalTaskSize*
  This value sets the size of small memory chunks in the Scheduling System's Task Heap (in words). This value will be rounded up to a power of two..
- *Status.Basic.TimeTaskResolution*
  This value sets the resolution of time-based Task scheduling (in microseconds). The Scheduling System will activate a Task once the Task is within this amount of time of the scheduled time, so this should be set larger than the largest expected time for the main loop.
- *Status.Basic.PromptTimeTasks*
  Setting this *true* will give time-based Tasks priority over Event-based ones. In ATLAS, we will set this to *true*.

- *Status.Basic.Override*
  Setting this *true* enables override for mode values. Decoders must support this feature or they will not be available. In ATLAS, we will set this to *false*.
- All other mode overrides
  These values are used to override the corresponding values in the Policy subsystem. Even if *Status.Basic.Override* is set *true*, the individual values are not overridden if they are set to 0 (or *trNone* for *OutputSkipLength*). In ATLAS, we will set all these values to 0 (*trNone* for *OutputSkipLength*).

- *Status.Basic.E_x_Y_Priority*
  This value allows fine control of the priorities of the Management System. In ATLAS, we will set this to *prEmpty*.
- *Status.Basic.G_x_E_Priority*
  This value allows fine control of the priorities of the Management System. In ATLAS, we will set this to *prEmpty*.
- *Status.Basic.G_x_G_Priority*
  This value allows fine control of the priorities of the Management System. In ATLAS, we will set this to *prEmpty*.
- *Status.Basic.G_x_Y_Priority*
  This value allows fine control of the priorities of the Management System. In ATLAS, we will set this to *prEmpty*.

- *Status.Basic.Order*
  Order from the HPU. In ATLAS, we will set this to *TStatus::orNone*.

- *Status.Basic.Calibration*
  This variable holds a pointer to the array used for Calibration and other Decoder initialization (NOT the array itself). The exact details of what goes into this array depend on the Decoder.

There are a couple items outside *Status.Basic* that may be optionally set:

- *Status.Buffer.ParameterArray*
  This variable holds a pointer to the array of Parameter sets (NOT the array itself). The Parameter sets can be written directly before a run to save time.
- *Status.Buffer.ParameterSize*
  This variable holds a pointer to the array of Parameter set sizes (NOT the array itself). The Parameter set sizes can be written directly before a run to save time.

- *Status.Buffer.CIB*
  This variable holds a pointer to the CIB (NOT the buffer itself). Commands CANNOT be written before the run is started.

After everything is set, *Status.Basic.Initialize* should be set to *true* to begin a run. Note that this may be done long before the actual run begins (especially useful if Tasks need to be created and started). Wait until *Status.Basic.BootStatus* is equal to *TStatus::bsNormal*.

Creating and using Parameter sets

Parameter sets are stored in the array pointed to by *Status.Buffer.ParameterArray*, which contains *TStatus::MaxParameterSize* times *TStatus::MaxParameterCount* words. Each Parameter set can hold up to *TStatus::MaxParameterSize* words, and there are *TStatus::MaxParameterCount* sets available. The first *TStatus::MaxParameterSize* words are Parameter set 0, followed by the remaining Parameter sets in order.

There are two ways to fill a Parameter set from the HPU. The first is to write directly to the appropriate words in the Parameter array. Once done, the corresponding word in *Status.Buffer.ParameterSize* must be set to the number of words in the new Parameter set. The second way is to send a Function Command with the Parameter set words attached. The specified Parameter set will be filled with those words before the Function is invoked.

On the DPU side, plug-ins will use *GetParameter()* and *GetParameterSize()* (Parameter.h) to access the Parameter sets.

Sending Commands during a run

The CIB is a circular buffer that receives Commands via DPU Control. One curiosity is that the CIB runs backwards in memory; this is necessary because DPU Control is only able to write forwards. The beginning of the buffer is at the highest address, and the end is at the lowest. A pointer to the lowest address is stored in *Status.Buffer.CIB*, and the size (in words) is available as *TStatus::CIBSize*.

During *Initialize()* (triggered by setting *Status.Basic.Initialize* equal to *true*), the DPU sets the entire CIB to zero and starts polling the first word (highest address) for a non-zero value. This word is the standard size word for a variable-length Packet. The size in the size word specifies the total number of words in this Command (including the size word itself); there is no padding because the *FrameSize* is 1. After (below) the Payload will be the size word of the next Command; this must be set to zero BEFORE the first size word is set non-zero to keep the DPU from reading past.

For space efficiency, the size word contains more than just the size. The lower halfword contains the size, while the upper halfword is determined by the Command being issued.

The actual Payload of the Command is stored in the normal (forward) direction in memory. This means that the lowest word just above the next size word is the first word of the Command, and the last word of the Command will be at the highest address just below the current size word.

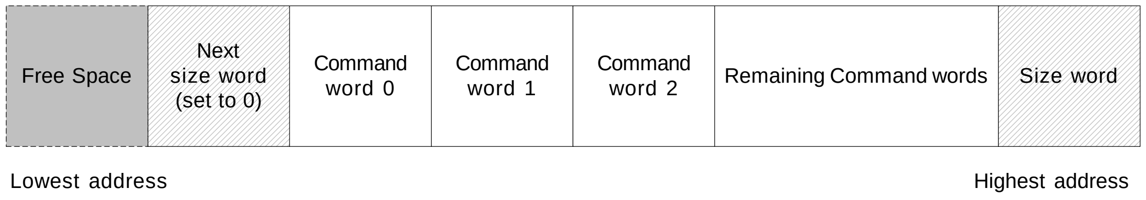


Figure 46: Layout of Commands in CIB memory (begin at the right)

To send a command, the HPU should send the following in the forward direction:

- Zero (next size word)
- Command word 0
- Command word 1
- Command word 2
- (any other Command words in order)
- Size word (should overwrite the previous zero)

When we get to the end (lowest address) of the buffer, something must be done to restart. It is not permitted for a Command to wrap the buffer. If a Command exactly fills the buffer, then the next Command can start at the beginning (highest address) with no special action taken. If the new Command cannot fit, the value *QueueRestart* (common.h) must be written in the size word following the last Command and the new Command goes at the beginning (highest address). Note that *QueueRestart* must be written AFTER the new Command is written, and that this will require two DPU Control writes. Management of CIB memory to prevent overflow is the HPU's responsibility.

There are 5 types of Commands that can be written:

- Default Decode Command
  This is a simple command to process N events using the default Decoder.

- Non-Default Decode Command
  This is a slightly more complex Decode Command that can use any Decoder.

- Function Command
  This Command invokes a Function and has several arguments

- Multi-Decode Command
  This allows a different Decode Command to be sent to every DPU

- Multi-Function Command
  This allows a different Function Command to be sent to every DPU

Multi Commands are the only way to send different Commands to different DPUs. DPUs can only look at one Command per pass through the main loop and have no mechanism to skip over Multi Commands that have only NOPs for them. A DPU could idle for many passes popping off Commands that are not meant for it, so this should be avoided. It is the HPUs responsibility to ensure DPUs are not delayed in receiving Decode Commands too long or drown in Function Commands; the exact policy for this depends on the Decoder used.

Three words are common to many Commands:

- **Size word (Size word above)**
  Upper halfword: CIndex
  Lower halfword: Size

- **Command word (Command word 0 above)**
  Upper halfword: Type
  Lower halfword: Command

- **Param word (Command word 1 above)**
  Entire word: Param

Default Decode Command:

- **Size word**
  CIndex = number of Events to Default Decode
  Size = 1

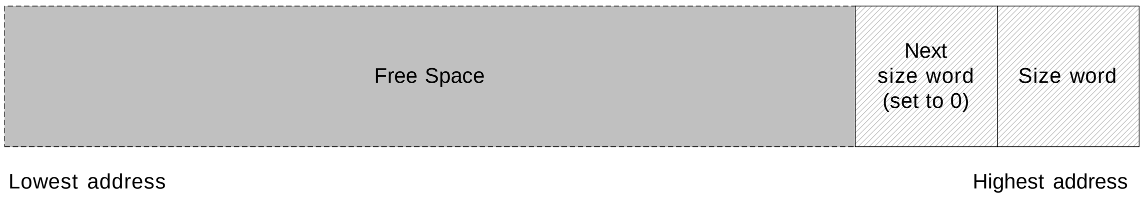


Figure 47: Memory layout for Default Decode Command

Non-Default Decode Command:

- **Size word**
  CIndex = number of Events to Decode
  Size = 2

1. **Command Word**
   Type = Decoder to use
   Command = *Command::DPU_DecodeEvent* (Command.h)

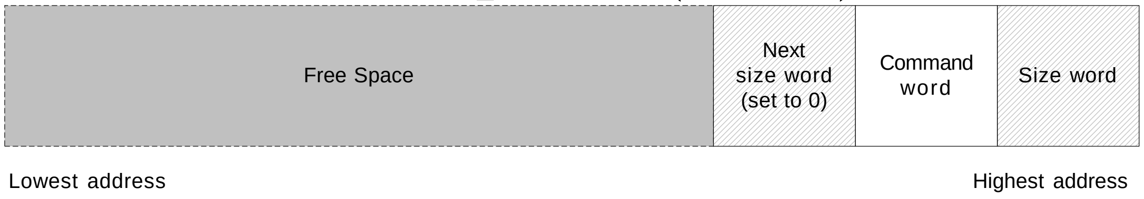


Figure 48: Memory layout for Non-Default Decode Command

Function Command:

- **Size word**
  CIndex = incrementing Function Command Index
  Size = 3 + number of optional Parameter words

1. **Command word**
   Type = Decoder to use
   Command = Function to invoke
2. **Param word**
   Param = Parameter set number to use
3. **Beginning of optional Parameter words**
   Any additional words tells the DPU to copy these words to the Parameter set specified in Param before invoking the Function

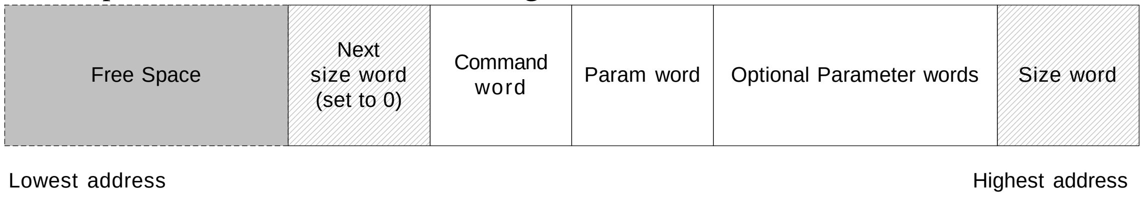


Figure 49: Memory layout for Function Command

<u>Multi-Decode Command:</u>

- **Size word**
  CIndex = unused
  Size = 2 + *MaximumDPUCount* (common.h)

1. **Command word**
   Type = unused
   Command = *Command::DPU_MultiDecode* (Command.h)
2. **Decode words (one for each Module ID)**
   Upper halfword = Number of Events to Decode, or 0 for a NOP
   Lower halfword = Decoder to use (0 for Default), or 0 for a NOP

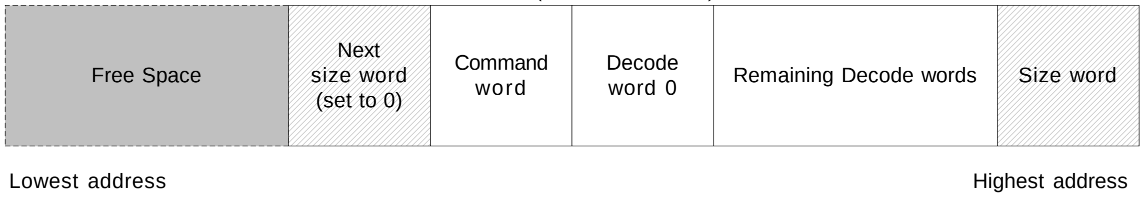


Figure 50: Memory layout for Multi-Decode Command

<u>Multi-Function Command:</u>

- **Size word**
  CIndex = incrementing Function Command Index
  Size = 2 + *MaximumDPUCount* (common.h) + size of Commands

3. **Command word**
   Type = unused
   Command = *Command::DPU_MultiCommand* (Command.h)
4. **Offset words (one for each Module ID)**
   Upper halfword = offset to the actual Command word relative to the Command word above, or 0 for a NOP
   Lower halfword = size of the actual Command below (does not include a size word), or 0 for a NOP
5. **Specific Commands**
   Function Commands for individual DPUs (without size words)

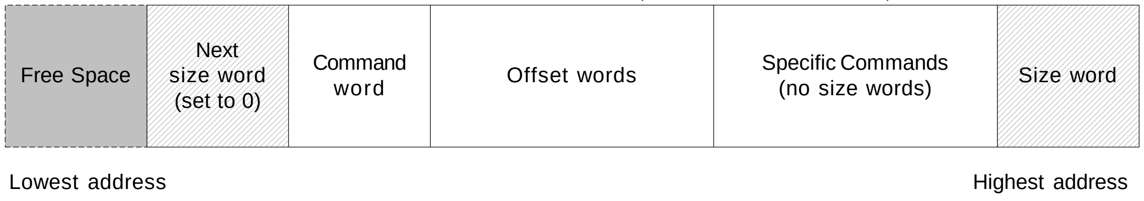


Figure 51: Memory layout for Multi-Function Command

Sending Orders during a run

Orders are asynchronous Commands that are executed as soon as the DPU sees them. Because of this, Orders violate all guarantees of stability and performance and should never be used except during testing or to end a run.

To send an Order, first fill *Status.Basic.Parameter* and *Status.Basic.ParameterSize* as appropriate for the Order being sent. Orders can be found in main.cpp in their own section. Once the Parameter set is ready, set *Status.Basic.Order* to the desired value from *TStatus::EOrder*. Poll *Status.Basic.OrderResponse* and read once the Reply word is non-zero. Be sure to zero this Reply word before sending another Order.

To safely end a run using an Order, first wait until all Commands have received a Response in the Response Buffer. Then send the Order *TStatus::orHalt*.

Using Tasks during a run

There are Functions that can interact with Tasks:

- *DPU_TaskCreateFunction*
  Set Type to the Task to be created. Param is unused
  Returns a Task ID of the created Task in the Response

- *DPU_TaskInitializeFunction*
  Set Type to the Task ID to be initialized. Param depends on the Task
  Response depends on the Task

- *DPU_TaskSignalFunction*
  Set Type to the Task ID to be signaled. Param depends on the Task
  Response depends on the Task

- *DPU_TaskOutputFunction*
  Set Type to the Task ID to be output. Param depends on the Task
  Response depends on the Task

- *DPU_TaskTerminateFunction*
  Set Type to the Task ID to be terminated. Param depends on the Task
  Response depends on the Task

- *DPU_TaskDestroyFunction*
  Set Type to the Task to be destroyed. Param is unused
  Response contains no value

A Task must be created but need not be destroyed until just before the end of the run. Before each use the Task should be initialized and then terminated when completed. Output is used to request output (current Temperature average or histogram, for example), while Signal can be used to send other messages to a Task.

Reading Responses during a run

The *TResponse* struct (common.h) is used to give a Response to a Decode or Function Command when complete. It has two items:

- **Reply word**
  Upper halfword: CIndex
  Lower halfword: Reply

- **Data[2] array**

For a Decode Response, Reply will be *reDecode*. CIndex will contain the least-significant 16 bits of the Event Index, while *Data[0]* contains the size of the Event being output (in words).

For a Function Response, Reply will be either *reSuccess* or one of the *reFailure* variants. CIndex will contain the incrementing Function Command Index, while Data will vary depending on the Command. For example, *DPU_TaskCreateFunction* will return the created Task ID in *Data[0]*.

The Response Buffer runs forward, and the entire Reply word will be zero in the next Response if it is not ready yet. Do not use *Status.Buffer.ResponseIndex*.

Reading Warnings during a run

The mask *Status.Basic.Warnings* contains flags described in *EWarning* (common.h). To clear these flags during a run, set the same flags in *Status.Basic.WarningsSeen* and wait until *Status.Basic.WarningsSeen* is cleared. Do not set additional Warnings as seen until this is cleared by the DPU.

Reading Faults during a run

If a Fault occurs, *Status.Basic.FaultCount* will be 1 (the first Fault will stop all operation when running normally in ATLAS). Read *Status.Basic.Fault[0]* to get the file (upper halfword) and code (lower halfword).

Reading Status during a run

Other values in the Status struct such as priorities or index/counters should not be used for anything except debugging. The Response Buffer gives all the information that the HPU needs to keep state with the DPUs. Using any other items introduces complex timing and synchronization issues that may lead to incorrect values. These items exist in the Status struct only to make them available to the entire DPU and for diagnostics.

Setting up for a new run

When the run ends (caused by sending the *orHalt* Order), *Status.Basic.Initialize* will be set back to *false* and *Status.Basic.BootStatus* will return to *TStatus::bsBasic*. To start a new run, there are a few items to check before setting this back to true:

- *Status.Basic.Order*
  Set this to *TStatus::orNone*.
- *Status.Buffer.ParameterArray*
  If any Parameter sets were changed during the run, they can be set back to their original values now.
- *Status.Buffer.ParameterSize*
  The corresponding sizes should also be adjusted.

DPU documentation for the plug-in writer

Getting started

A plug-in generally consists of a single header file, though a source file may also be needed. Both files must be assigned file numbers and added to the list of files in Fault.h as follows:

```
DPU_FAULT_FILE( 0x4309, "file.cpp" )
DPU_FAULT_FILE( 0x430A, "file.h" )
```

Decoders are numbered 0x43??, Functions are numbered 0x53??, and Tasks are numbered 0x63??. These numbers will be reported in the upper halfword of any Faults in these files and must be unique.

The header file must also be added to the corresponding list of plug-in headers:

- Data\decoders\include.h
- Command\functions\include.h
- Scheduling\tasks\include.h

The line in the list of headers is a simple include with no path (the file should be located in the same folder as include.h):

```
#include "file.h"
```

To register a plug-in, it must be added to the corresponding list of plug-ins:

- Data\decoders\decoders.h
- Command\functions\functions.h
- Scheduling\tasks\tasks.h

Details on the specifics of these registration macros are contained in the comments for these files. More than one plug-in (of the same type only) can be in a single header file, but each must be registered separately.

The source file must be added to the batch file MAKE.bat in order to be compiled. It should be added to end of the *OPTIONS_F3* line:

```
set OPTIONS_F3=Data\decoders\file.cpp
```

Any time the number 10,000 DSP clocks is mentioned as a limit, this refers to the total Main Line processing time including *Update()* and Management System overhead.

Boilerplate

Header, top

```
// ---------------------------------------------------------------
//  Data\decoders\file.h  My Decoders for a DPU, by John Smith
// ---------------------------------------------------------------


// This file will be included specially, so no protection is needed


// Check file version number

#ifndef DPU_VERSION_1_40
  #error "Assertion Failed: Incorrect Data\\decoders\\file.h version"
#endif


// Include the C6x intrinsics

#include <c6x.h>


// Include the basic definitions used everywhere

#include "..\\..\\common.h"


// Include the HAL headers

#include "..\\..\\HAL\\Status.h"
#include "..\\..\\HAL\\Policy.h"
#include "..\\..\\HAL\\Queue.h"


// Include the Decoder header file

#include "..\\decoder.h"


// Set this file's number (must be after all includes)

#define DPU_FILE_NUMBER  0x430A



// This file adds to the DPU::Data::MyDecoder namespace

namespace  DPU
{
  namespace  Data
  {
    namespace  MyDecoder
    {
```

Header, bottom

```
// -----------
//  Namespace
// -----------


// This file adds to the DPU::Data::MyDecoder namespace

    }
  }
}


// Clear this file's number

#undef DPU_FILE_NUMBER
```

Source, top

```
// --------------------------------------------------------------
//  Data\decoders\file.cpp  My Decoders for a DPU, by John Smith
// --------------------------------------------------------------


// Check file version number

#ifndef DPU_VERSION_1_40
  #error "Assertion Failed: Incorrect Data\\decoders\\file.cpp version"
#endif


// Include the basic definitions used everywhere

#include "..\\common.h"


// Include the corresponding header file

#include "file.h"


// Set this file's number (must be after all includes)

#define DPU_FILE_NUMBER  0x4309



// -----------
//  Namespace
// -----------


// This file adds to the DPU::Data::MyDecoder namespace

namespace  DPU
{
  namespace  Data
  {
    namespace  MyDecoder
    {
```

Source, bottom

```
// -----------
//  Namespace
// -----------


// This file adds to the DPU::Data namespace

    }
  }
}
```

This is boilerplate used in most DPU files. The exact headers to include vary, but most all plug-ins will likely need Status.h and Policy.h to access the Status struct and Policy namespaces. Queue.h is needed by all Decoders and some Tasks to access the DIB and DOB. The file Decoder.h should be replaced with Command.h or Task.h as appropriate.

There is no special restriction on the namespace, but a unique namespace under the Data, Command, or Scheduling namespace is appropriate for Decoders, Functions, and Tasks, respectively.

The file version line must be updated whenever the DPU software version is changed. There is a macro in the batch folder that does this automatically, so be sure to add any new files to the list in batch\list.txt so that it is updated.

Policy namespaces

As was discussed in the description of the Policy subsystem, plug-ins have the choice of whether they use the constants for a specific mode or whether they use the variables for the current mode. Using the constants is more limiting but it allows for maximum performance.

To isolate this decision from the code, plug-ins (especially Decoders) should create their own local namespace and bring the appropriate constants or variables in with using statements. For example:

```
// -------------------
//  Policy namespaces
// -------------------


// Namespace used for my special timeslices

namespace PolicyTimeslice
{
  using Policy::Public::InputEventLength
  using Policy::Private::InputMyTimeslice::InputFrameSize;
}
```

In this case, we are using the public (run-time) version of *EventLength* and the private (mode-specific constant) of *FrameSize* for input. To support this, we need to check this in *CanInitialize()*:

```
bool  CanInitialize()
{
  if ( Policy::Public::InputEventLength !=
       PolicyTimeslice::InputEventLength )  return  false;
  if ( Policy::Public::InputEventLength !=
       Policy::Public::OutputEventLength )  return  false;
  if ( Policy::Public::InputMaxPacketLength  != 1 )  return  false;
  if ( Policy::Public::OutputMaxPacketLength != 1 )  return  false;
  if ( Policy::Public::InputFrameSize !=
       PolicyTimeslice::InputFrameSize )  return  false;
  if ( Policy::Public::InputFrameSize  !=
       PolicyTimeslice::OutputFrameSize )  return  false;
  return  ( ( Status.Basic.InputData == TStatus::itMyTimeslice ) &&
            ( Status.Basic.OutputData == TStatus::otMyTimeslice ) );
}
```

The first check makes sure that the current mode uses the same *InputEventLength* that we are using. The second check makes sure that both output and input use the same *EventLength* (useful for Decoders that copy input to output...otherwise output will generally be in the local Policy namespace as well). The third and fourth checks look for fixed-length input and output, which is simpler to test against 1 rather than bringing *MaxPacketLength* into the local Policy namespace. The fifth and sixth checks do the same thing for *InputFrameSize* that the first and second do for *InputEventLength*. Finally, the last check makes sure the basic data types match what we expect.

With these tests, we can handle overrides as long as *InputFrameSize* and *OutputFrameSize* don't change and both input and output remain fixed-length. If we wanted to change the Decoder (or cut/paste a new version) to use any *FrameSize* or to bind to a constant *EventLength*, we only have to touch the local Policy namespace; all the tests in *CanInitialize()* are fine.

There is one very important warning:

DO NOT ATTEMPT TO *USING* THE LOCAL POLICY NAMESPACE!

The following code will NOT work:

```
namespace PolicyTimeslice
{
  using Policy::Public::InputEventLength
  using Policy::Private::InputMyTimeslice::InputFrameSize;
}

using namespace PolicyTimeslice;
```

This seems like a clever shortcut, but the namespace lookup rules for C++ cause this to fail miserably. Always be explicit and write it out, such as writing *PolicyTimeslice::InputFrameSize* in the case above.

Faults and asserts

All files should use the macro functions *Fault()* and *assert()* to report errors (common.h). *Fault()* checks remain in the final system, while *assert()* checks exist only in debug mode. It is important that any time-consuming checks use *assert()* to preserve performance.

A call to *Fault()* with a unique Fault code for the file produces a Fault, so the check should be done in a conditional. *assert()* works the same way as a standard C++ *assert()* macro. When a *Fault()* or failed *assert()* occurs, the system stops processing and enters an infinite loop doing nothing. This allows for debugging using the HPU or emulator.

A Fault is reported in Status.Basic.Fault[0] with the file code in the upper halfword and the Fault code in the lower halfword. An assert is reported the same way, but the Fault code will have the high bit set and the remaining bits will be the line number for the assert. All Faults should be entered into fault.h but asserts are not. The entries in fault.h should appear directly below the file's entry and include the function/method that invoked a Fault and a brief message:

```
DPU_FAULT_CODE( 0x0001, "Execute, Multi-command too small" )
DPU_FAULT_CODE( 0x0002, "Execute, Command Index was out of sequence" )
DPU_FAULT_CODE( 0x0003, "Execute, Function out of range" )
```

Because of the nature of the separation between the DPU software framework and the individual plug-ins, it should not be necessary to perform any special testing of the framework when new plug-ins are added. Normal testing of the plug-ins, as designed by the plug-in creator, should be sufficient. These would typically involve using the plug-in for a large period of time in both typical and atypical conditions. All asserts should be enabled during this or any other testing (except performance testing).

Decoders

Decoders are derived from the base class *TDecoder* and overload the virtual functions *CanInitialize()*, *Initialize()*, *Execute()*, and *Terminate()*. *CanInitialize()* shall return true if and only if the Decoder works in the current mode. *Initialize()* and *Terminate()* are invoked only if *CanInitialize()* returned true, and *Execute()* is invoked to process one Event. If *Execute()* can't complete the necessary processing in 10,000 DSP clock cycles, it should return without popping and continue during the next invocation.

Below is our sample Decoder that copies input to output:

```
// ---------------
//  My Timeslices
// ---------------


// Decoder that takes fixed-length timeslices and copies to output
//  Override: Supported for EventLength
//            Needs InputFrameSize == OutputFrameSize
//            and InputEventLength == OutputEventLength

class  TMyTimesliceDecoder : public  TDecoder
{
 public:


// CanInitialize virtual function

...same as the example in the Policy namespaces section...


// Initialize virtual function

void  Initialize()
{
 // Return
  return;
}


// Terminate virtual function

void  Terminate()
{
 // Return
  return;
}
// Execute virtual function

void  Execute()
{
 // Get the event
  const Queue::TPacket*  event = Queue::DIB.front();

 // Copy the event to output and push it on
  for ( uint  i = 0; i < PolicyTimeslice::InputEventLength; ++i )
  {
    uint*  dest = Queue::DOB.back();
    assert( dest );
    FastCopy< uint >( event[ i ].Payload, dest,
                      PolicyTimeslice::InputFrameSize );
    Queue::DOB.push_fixed( Status.Processing.EventIndex );
  }

 // Pop off the event
  Queue::DIB.pop( PolicyRaw::InputEventLength );

 // Increment the EventIndex/EventCounter
  ++volatile_value( Status.Processing.EventIndex );
  if ( !Status.Processing.EventIndex )
    ++volatile_value( Status.Processing.EventCounter );

 // Return
  return;
}


};
```

The call to *DIB.front()* returns an array of *TPacket*, each of which contains the members *Size* and *Payload* for one Packet. *Size* contains the number of words in Payload, while *Payload* contains the data for the Packet. Since we are only using fixed-length Packets here, we don't bother to check the redundant size information. If we were using a variable-length format that placed information in the upper halfword of the size word, we would need to mask before checking the size.

A call to *DOB.back()* returns a pointer where the next output Packet should be written. This output is finished when *DOB.push_fixed()* or *DOB.push_variable()* (Queue.h) is called as appropriate to the data type, at which point *DOB.back()* will return the pointer for the next Packet to be output.

After all the copying done, *DIB.pop()* is called with the number of Packets to be removed. If some Packets are shared between Events, this number may be less than the *EventLength*. *EventIndex* and *EventCounter* should also be incremented, but be sure to use the old value of *EventIndex* in any calls to *DOB.push_fixed()* or *DOB.push_variable()*.

FastCopy() is a template function in common.h that should be used for any large blocks of copies for optimal performance. If the source or destination needs to skip over intervening values, the alternative *FastCopyStride()* is available.

Functions

Functions are derived from the base class *TFunction* and overload the virtual functions *CanInitialize()*, *Initialize()*, *Execute()*, and *Terminate()*. *CanInitialize()* shall return true if and only if the Function works in the current mode. *Initialize()* and *Terminate()* are invoked only if *CanInitialize()* returned true, and *Execute()* is invoked to perform the Command. If *Execute()* can't complete the necessary processing in 10,000 DSP clock cycles, it should return without popping and continue during the next invocation.

Below is a sample Function that returns the last Decoder used by the DPU. Since this is only going to be used in diagnostics, we will simply return the value sitting in *NextDecoder* (Command.h). This isn't entirely valid (such as when the run has just started), but will normally give what was the next Decoder before this Command was executed. To be a little safer, we will return failure if *EventIndex* is 0 since *NextDecoder* hasn't been written to yet:

```
// ---------------------------
//  Previous Decoder Function
// ---------------------------


// Function that returns the value of NextDecoder

class  TDummyFunction : public  TFunction
{
 public:

// CanInitialize virtual function

bool  CanInitialize()
{
  return  true;
}


// Initialize virtual function

void  Initialize()
{
 // Return
  return;
}


// Terminate virtual function

void  Terminate()
{
 // Return
  return;
}
// Execute virtual function

void  Execute( uint  CIndex, uint  Type, uint  Parameter )
// Type and Parameter are unused
{
 // Respond with success and give the NextDecoder value
 //  unless we haven't had an Event yet
  if ( Status.Processing.EventIndex || Status.Processing.EventCount )
  {
    Queue::Respond( reSuccess, CIndex, NextDecoder );
  }
  else
  {
    Queue::Respond( reFailure, CIndex );
  }

 // Pop off the command
  Queue::CIB.pop( 1 );

 // Return
  return;
}

};
```

CanInitialize() is trivial in this case since this function works in any mode. The primary responsibility of *Execute()* is to give a Response and pop the Command out of the CIB. The extra two values to *Respond()* are optional and their meanings depend on the context. In our case, *Data[0]* (the first of the two values) is the value of *NextDecoder* and *Data[1]* (the second value) is unused. When the Function fails, the values have no meaning.

In this simple case, the value being returned was available simply by including Command.h, but this is not generally the case. Extra effort may be needed to access protected data in classes where no provision was made to read from the outside.

Tasks

Tasks are derived from the base class *TTask* and overload the virtual functions *CanInitialize()*, *Initialize()*, *Signal()*, *Output()*, *Terminate()*, *Capture()*, and *Service()*. *CanInitialize()* shall return true if and only if the Task works in the current mode. *Initialize()* and *Terminate()* are invoked only if *CanInitialize()* returned true.

Tasks are able to schedule their *Capture()* or *Service()* functions (one at a time, not both) to activate in a certain amount of time or on a certain Event number. *Capture()* will not be activated early, but *Service()* can take place any time up to the specified limit. The intention is for *Capture()* to quickly grab a copy of relevant, time-sensitive data and *Service()* to perform any complicated processing. This is critically important because multiple Tasks could coincidentally schedule their *Capture()* at the same time. Moving work from *Capture()* to *Service()* increases the likelihood that all *Capture()* requests can be completed before the system is forced to move back to processing Events. If *Service()* can't complete the necessary processing in 10,000 DSP clock cycles, it should schedule another *Service()* and continue during the next invocation. Scheduling an additional *Capture()* is possible for Event-based Tasks (not time-based) but is not recommended.

Below is a sample Task that monitors the temperature reported by the EFPGA:

```
// ------------------
//  Temperature Task
// ------------------


// Task records at a non-zero time-based interval, runs until stopped
//  Initialize with Parameter = interval (in microseconds)
//  Output needs no parameter, single TAverage returned by pointer
//  Terminate needs no parameter, single TAverage returned by pointer

class  TTemperatureTask : public  TTask
{
 protected:


// Schedule interval (in microseconds) (set to zero to indicate a
missed Capture scheduling)

uint  Interval;


// Next scheduled time (in microseconds)

uint  NextTime;


// Accumulating average

TAverage  Average;


// Average used for HPU communication

TAverage  OutputAverage;
```

```
 public:

// CanInitialize static function

static bool  CanInitialize()
{
  return  true;
}


// Initialize virtual function

void  Initialize( uint  CIndex, uint  Parameter )
{
 // Running
  Running = true;

 // Store interval and next time
  Interval = Parameter;
  NextTime = Status.Processing.TimeIndex;
  if ( !Interval )
  {
    Queue::Respond( reFailureParameter, CIndex );
    return;
  }

 // Zero average
  Average.CountLSW = 0;
  Average.CountMSW = 0;
  Average.SumLSW   = 0;
  Average.SumMSW   = 0;
  Average.Minimum  = 0xFFFFFFFF;
  Average.Maximum  = 0;

 // Schedule and Respond
  NextTime += Interval;
  if ( RequestTimeCapture( this, NextTime ) )
    Queue::Respond( reSuccess,         CIndex );
  else
    Queue::Respond( reFailureResource, CIndex );

 // Return
  return;
}


// Terminate virtual function

void  Terminate( uint  CIndex, uint  Parameter )
{
  assert( Running );
  TTemperatureTask::Output( CIndex, Parameter );
  Running = false;
}


// Signal virtual function

void  Signal( uint  CIndex, uint  Parameter )
{
 // Not supported
  Queue::Respond( reFailureSupport, CIndex );
  return;
}
```

```
// Output virtual function

void  Output( uint  CIndex, uint  Parameter )
{
 // Check for previous inability to schedule
  assert( Running );
  if ( !Interval )
  {
    Queue::Respond( reFailureResource, CIndex );
    return;
  }

 // Copy for HPU
  volatile_value( OutputAverage.CountLSW ) = Average.CountLSW ;
  volatile_value( OutputAverage.CountMSW ) = Average.CountMSW;
  volatile_value( OutputAverage.SumLSW   ) = Average.SumLSW;
  volatile_value( OutputAverage.SumMSW   ) = Average.SumMSW;
  volatile_value( OutputAverage.Minimum  ) = Average.Minimum;
  volatile_value( OutputAverage.Maximum  ) = Average.Maximum;

 // Respond and return
  Queue::Respond( reSuccess, CIndex, 1, &OutputAverage );
  return;
}


// Capture virtual function

void  Capture()
{
 // Must not have missed a Capture scheduling
  assert( Interval );

 // Stop now without reschedule if not running
  if ( !Running )  return;

 // Get the new temperature
  int  temp = FPGA::EFPGAGetTemp();

 // Update Count
  ++Average.CountLSW;
  if ( !Average.CountLSW )  ++Average.CountMSW;  // LSW roll-over

 // Update Sum
  Average.SumLSW += temp;
  if ( temp > Average.SumLSW )  ++Average.SumMSW;  // LSW roll-over

 // Update min/max
  if ( temp < Average.Minimum )  Average.Minimum = temp;
  if ( temp > Average.Maximum )  Average.Maximum = temp;

 // Schedule
  NextTime += Interval;
  if ( !RequestTimeCapture( this, NextTime ) )  Interval = 0;

 // Return
  return;
}
```

```
// Service virtual function

void  Service()
{
 // Not used
  Fault( 0x0002 );  // Service requested
}

};
```

All Tasks should use the *Running* bool in the *TTask* base class to keep track of their state. Scheduling for *Capture()* is done by calling the static function *TTask::RequestTimeCapture()*; similar functions exist for Event-based operations and *Service()* requests. Because the processing is so trivial in this case, we do it in *Capture()* rather than *Service()*.

Because there is no mechanism to get back into the *Capture()*/*Service()* queue (besides custom support in *Signal()*), most Tasks will remain constantly in one of the two queues. When a Task is terminated, setting *Running* to *false* can be used to inform the next *Capture()* or *Service()* not to proceed.

If a problem occurs during *Capture()* or *Service()*, a flag should be set so the problem can be reported to the HPU during the next *Output()*, *Signal()*, or *Terminate()*. The problem cannot be reported immediately because the HPU is not expecting a Response from *Capture()* or *Service()*.

Software source code

A complete copy of the source code can be found at http://www.cephira.com/thesis/.

Appendix B: K Physics in the FDQM

Software source code

A complete copy of the source code can be found at http://www.cephira.com/thesis/.