

DAQ Programmer's Guide¹

Version 1.1

September 3, 1997 6:01 pm

R. T. Hamilton, University of Iowa²

M. E. Huffer, Stanford Linear Accelerator Center²

J. L. White, Stanford Linear Accelerator Center²

Abstract

All BaBar Data Acquisition Systems have both a hardware and a software component. The software component is called *DataFlow*. *DataFlow*'s principle function is to serve as a mechanism to export the functionality of the underlying hardware component of the system. This component will be called the *Platform*. A user (or application) interacts with the platform through a series of Application Program Interfaces (APIs). These APIs are modeled as a set of C++ classes organized into *packages*. Each package encapsulates a set of classes which cooperate in order to express a common functionality. The collection of all packages constitutes *DataFlow*. A reference guide to these packages can be found in [2]. Packages are arranged to take advantage of the two different perspectives a user has of a platform:

- As a module which is “plugged” into the user's system. Here the user is interested in treating and controlling the platform as a whole, without any regard for its micro-structure. The set of classes which allow the user to maintain this view can be found in the *Management* package. Typically, the user is an application executing in an environment external to the platform (e.g. a Unix work-station). An example is the Online System which views the IR-2 platform as one component of its own system.
- As a system into which one or more application modules can be “plugged”. Once an application module has been plugged into a platform, it effectively becomes a part, or extension, of the platform. Consequently, modules can be (and are) managed by external applications via the classes in the *Management* package. An application plugs into the platform through the classes provide by the *Client* package.

Modules can be plugged into different locations within the platform. The choice of where modules are plugged in is largely driven by the module's requirements in accessing data. This topic is discussed in greater detail within Section 1.1 on page 9.

Application modules usually support access to, reduction of, and transformation of event oriented data. This support is accomplished through a framework which is modeled on a Finite State Machine discipline. (See Section 4.0 on page 26.) The *Client* package also provides this support.

Every *DataFlow* component in a platform maintains two representations of its identity. These are known as the *address* and the *tag*. The address specifies the location of a component within the platform. The tag specifies the location of a component on the detector. Every detector component has a well defined relation to a platform component. The tag and address may be used to label the containers which are used to transport data throughout the platform. The *Container* package introduces these labeled containers as *tagged container* objects which are the fundamental building blocks for data organization within *DataFlow*. Tagged containers allows *DataFlow* to present a regular data access interface throughout a platform and to transport data with a common scheme.

1. The currently released version can be found at www.slac.stanford.edu/BFROOT/doc/www/Computing/Online/OnlineDataFlow.html

2. rhamilt@slac.stanford.edu, mehsys@slac.stanford.edu, jlw@slac.stanford.edu

Table of Contents

List Of Figures	4
List Of Tables.....	5
1.0 The DataFlow Platform.....	6
1.1 The Platform hierarchy.....	9
1.2 Partitioning the Platform	10
1.3 Managing the platform	12
1.3.1 Shutting down the platform	13
1.4 Establishing and monitoring a Partition	13
1.5 Dissolving a partition	14
2.0 Streams.....	15
2.1 The information carried on a stream (Datagrams)	15
3.0 Appliances.....	17
3.1 Inlet.....	19
3.2 Terminators.....	20
3.3 Generators	20
3.4 Finite-State-Machines (FSM).....	21
3.5 Concentrators	21
3.6 Distributors.....	23
3.7 Placing appliances	23
4.0 The Finite State Machine	26
4.1 The FSM Classes.....	28
4.1.1 dfManager	28
4.1.2 dfFsm	30
5.0 Calibration.....	32
6.0 Taking Data.....	36
7.0 An Introduction to Containers in DataFlow.....	38
8.0 Addresses and Tags.....	38
8.1 Address Field Definitions.....	39
8.2 Tag Field Definitions	41
8.3 Assigning Addresses and Tags	43
9.0 Tagged Containers	43
10.0 Event Hierarchy	46
10.1 Maps	47
10.2 Input Event Hierarchy	48
10.3 Applying the Event Hierarchy in Reverse.....	50
11.0 Using The Event Hierarchy	50
12.0 Front End Interface	50
13.0 Role of Time in DataFlow.....	53
13.1 DataFlow Time vs. PEP-II Time	54

13.2	Reinitialization of the Timebase	55
13.2.1	Monotonicity Enforcement.....	55
13.2.2	Timebase is a platform quantity, not a partition quantity	56
13.2.3	Uncertainty in the Timebase.....	56
14.0	Occurrences.....	58
15.0	Trigger monitoring and deadtime	58
16.0	Event damage.....	58
	List of References	R1
	DataFlow Documents.....	R1
	ROM Documents	R1
	FCTS Documents.....	R1
	Wind River Systems (WRS) Documents	R1
	Other Third Party Documents	R2
	DataFlow CDR Supporting Documents.....	R2
	Other BaBar Documents	R2

List Of Figures

FIGURE 1.	The IR-2 Platform.....	8
FIGURE 2.	The Platform Hierarchy	9
FIGURE 3.	Partitioning the IR-2 platform.....	12
FIGURE 4.	The Envelope/Letter idiom as a metaphor for a datagram.....	16
FIGURE 5.	The Appliance as a metaphor for a stream attachment.....	18
FIGURE 6.	Stream after platform reset.....	24
FIGURE 7.	<i>Framework</i> stream ready for data taking.	24
FIGURE 8.	The DataFlow Finite State Machine.	27
FIGURE 9.	An exploded view of the Active state.	31
FIGURE 10.	Timing structure of a calibration cycle executed in the external mode.	32
FIGURE 11.	Timing structure of a calibration cycle executed in the internal mode.	34
FIGURE 12.	Event Hierarchy Naming Scheme.....	39
FIGURE 13.	Notional Picture of a Tagged Container.....	44
FIGURE 14.	A simple tagged container hierarchy	45
FIGURE 15.	DataFlow's Event Hierarchy in Address Space	47
FIGURE 16.	PEP-II and Platform timing structure	54
FIGURE 17.	Flow chart describing timebase reinitialization	57

List Of Tables

TABLE 1.	Address Field Definitions	40
TABLE 2.	Tag Field Definitions.....	42
TABLE 3.	Assignment of Detector Names and Nicknames	43
TABLE 4.	Input Event Hierarchies Mapping at the Segment Level	49
TABLE 5.	Number of components in the Input Event Hierarchies.....	50

1.0 The DataFlow Platform

Naively, one imagines BaBar with only a single platform whose function is to acquire detector data from PEP-II interactions. However, in actuality, the BaBar experiment defines many platforms, only one of which resides in IR-2. These systems serve a myriad of functions ranging from electronics checkout to code development for both individual subsystems and DataFlow. While physically disjoint, every BaBar platform has in common the following features:

- Has one or more VME crates. These crates have a custom J3 backplane and are capable of housing 9U modules.
- Requires an external *Timing System*. Clocking for the platform is derived from an external source to allow synchronization and accurate phasing between the platform and the electronics it is servicing¹.
- Requires an external *Trigger System*. The platform does not include any logic to either calculate or derive a trigger. Instead, it provides thirty-two (32) input lines, each of which, if driven correctly will cause an *LIAccept* to be generated and propagated throughout the platform. These lines may be driven either externally through hardware, or internally through software.
- Contains at least one Fast Control *Partition Master* (FCPM) and Fast Control *Distribution Module* (FCDM). These modules serve as an interface to the external trigger system, provide calibration sequencing, and distribute common clocking and command signals.
- Has one or more *Read-Out-Modules* (ROMs). The ROM is an intelligent VME module whose primary function is to gather event data from its managed Front-End-Electronics (FEEs) and extract from this data the “interesting” features. To support this function, the ROM has the responsibility to both configure and monitor the behavior of its managed FEEs.

In addition, one ROM within a crate is tasked with the responsibility of gathering the feature extracted contributions from each of the other ROMs within its crate and posting the result onto the Bulk Data Fabric (described below). By convention this role is assigned to the ROM residing in the first slot in a crate (the *Slot-1 ROM*).

Both DataFlow and the application code in each Read-Out-Module execute under the supervision of a commercial real-time kernel provided by Wind River Systems (*VxWorks*).

- Has one or more *Generic Unix Boxes* (GUBs) whose function is to serve as both as source and sink for the Bulk Data Fabric. In addition, these boxes serve as the cross-development platform for code development targeted to the platform’s ROMs.

1. Because most platforms outside of IR-2 do not have access to a timing system, the platform includes a clock which can be used in place of the external timing system to drive both the platform and the electronics it services.

- Has a *Bulk Data Fabric* whose function is to transport large volume data (principally event oriented) both into and out of the platform. In IR-2, this function is implemented through a large capacity network based upon a high-speed switch. For platforms outside of IR-2 where cost, not performance is the governing factor, the Bulk Data Fabric and Control Fabric may be shared (typically through Ethernet).
- Has a *Control Fabric* whose function is to provide connectivity between the Slot-1 ROMs and any GUBs. This connectivity is used as a path independent of the Bulk Data Fabric to assist in the transport of small, fixed-sized messages, which represent necessary control, error recovery, and some forms of monitoring traffic.

Greater detail on both the components within a platform and the composition of “interesting” platforms can be found in [3]. However, each platform, regardless of its purpose or location, subsets from the platform found in IR-2. In addition, DataFlow’s architecture assures its portability, independent of either platform composition or platform location. Therefore, this document will use the IR-2 platform as an example when demonstrating platform concepts. An illustration of the composition and topology of this platform is shown in Figure 1.

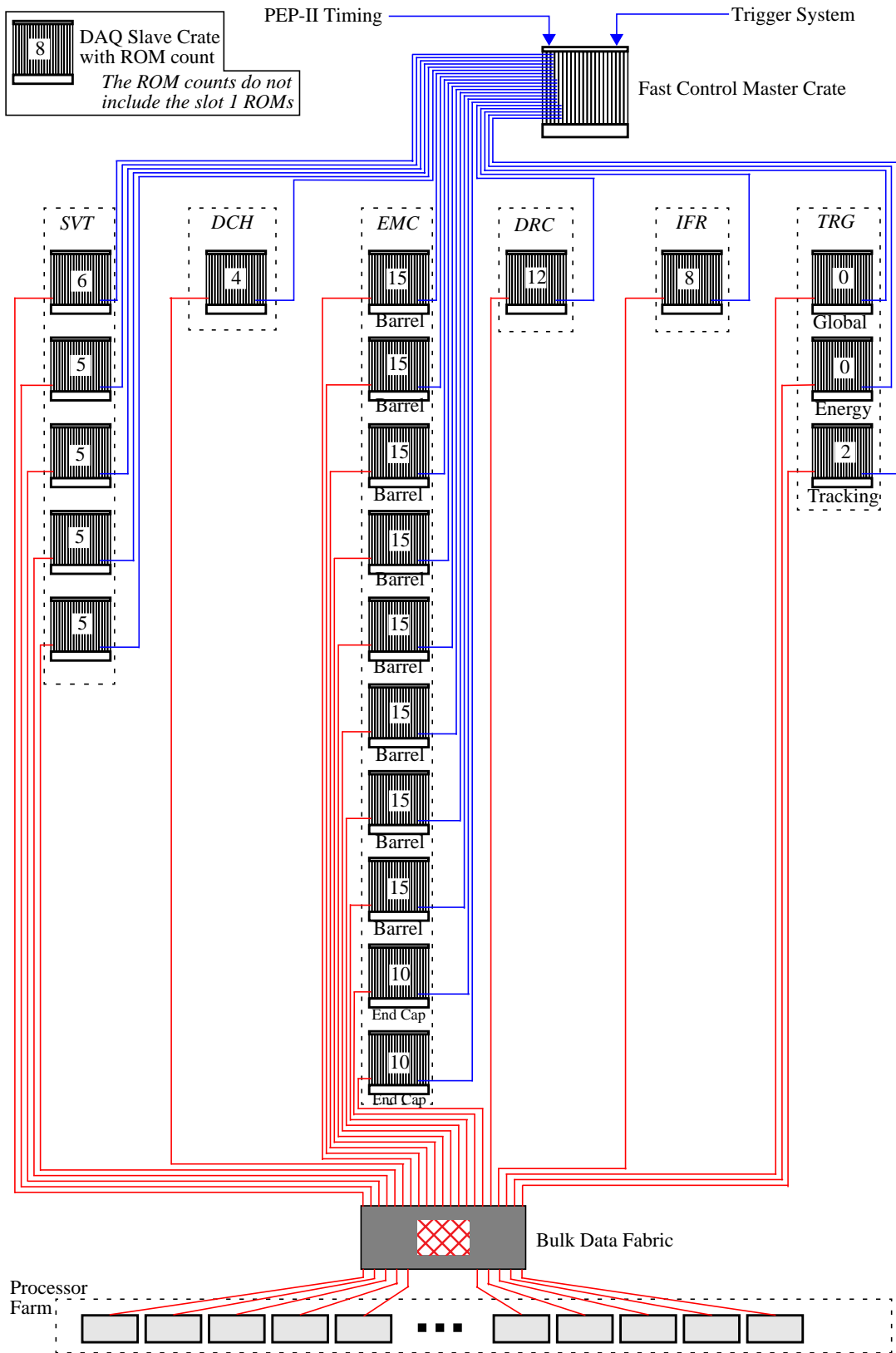


FIGURE 1. The IR-2 Platform

1.1 The Platform hierarchy

The architecture of the DataFlow platform is designed to address the stringent performance requirements imposed by the detector in both event rate and size (see [29]). To this end, all platforms, independent of their composition, have their internal components connected in a hierarchical fashion. This hierarchy allows the platform to exploit its deep buffering capability and achieve a high degree of parallelism. DataFlow reflects the platform hierarchy by defining four levels as represented by the following figure:

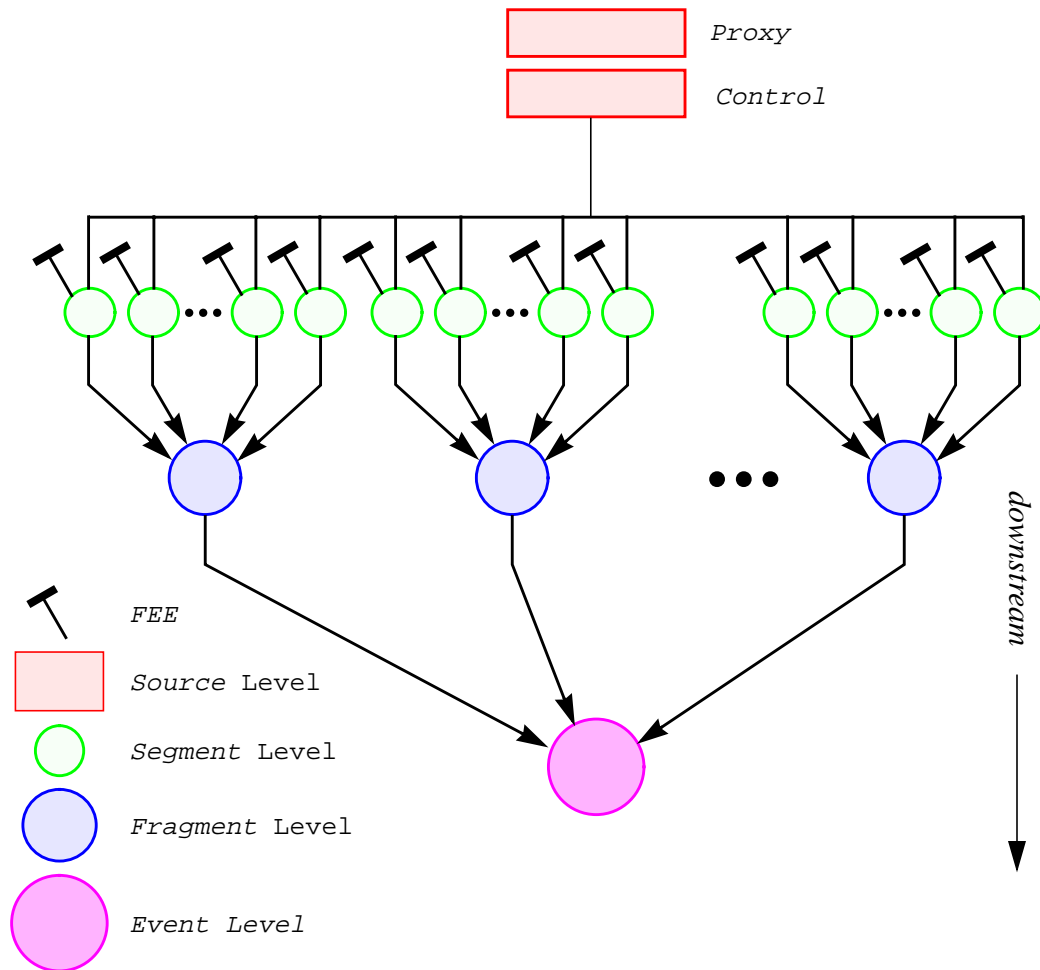


FIGURE 2. The Platform Hierarchy

However, another view of this hierarchy is to consider the platform as a four stage *event*¹ pipeline. All platforms are pipelined in the sense that the entire hierarchy contains many event contributions, in various states of assembly, at any one time. Each event contribution comes into existence at the *Source* stage, propagates down the hierarchy, and, at each stage, is associated with all its corresponding contributions. At the *Event* stage, complete

1. Where here an event corresponds to both control and data information.

(built) events are available for consumption. The function of each stage or level is as follows:

- The *source* level. At this stage, both control and data information come into existence. The underlying mechanism for generating this information is through commands issued by the FCTS's Partition Master. These commands include *LIAccepts* which initiate detector read-out, *CalStrobes* to inject calibration pulses, and *Syncs* used to resynchronize the platform's clocks. The Partition Master is managed by DataFlow software executing in its crate's Slot-1 ROM, therefore this stage is sometimes referred to as the *Control* level. An application can participate directly in this management through the classes contained in the *Management* package. However, normally this control is required from outside the platform environment. To provide this functionality in as transparent a fashion as possible, DataFlow includes a proxy layer, which gives an external application the appearance of directly controlling the Partition Master.
- The *Segment* level. This stage contains both *Triggered* and *Untriggered* ROMs. These ROMs (and their corresponding code) have two primary responsibilities:
 - On the receipt of an *LIAccept*, gather event contributions from the *Front-End-Elements* (FEEs), feature extract the contribution into an *segment*, and post the segment contribution to the *Fragment* stage.
 - As required, monitor, configure, and control the FEEs by issuing subsystem specific "Read" or "Write" commands down the C-link connected between the ROM and its FEEs.
- The *Fragment* level. This stage contains the Slot-1 ROM which has an interface to the Bulk Data Fabric. The responsibility of this ROM (and its corresponding code) is to gather its crate's segments (produced by the *Segment* level) into a *fragment*. The resulting fragment is then posted to the *Event* stage through the Bulk Data fabric.
- The *Event* level. This stage contains one or more GUBs and represents a generic processing farm. The responsibility for gathering the contributions from the *Fragment* stage into whole or complete events rotates among the farm processors. Once the event has been built it is available to event processors, for example, the L3 Trigger processor executing under the control of Online Event Processing (OEP).

As discussed in the introduction, Application Modules can be plugged into any location within the platform hierarchy. The choice of where modules are plugged into the hierarchy is largely driven by the module's data access requirements. For example, an application module which performs feature extraction on the Read-Out-Module (ROM) would plug in at the *Segment* stage, while an application module which requires access to whole (or built) events would plug into the hierarchy at the *Event* stage. In either case, the interface to the data is independent of the stage and is always through the classes contained in the *Client* Package.

1.2 Partitioning the Platform

An examination of Figure 1 on page 8 reveals that a platform has the capability to service the electronics from one or more detectors. These detectors are themselves potentially

capable of independent operation, however, because they share the same platform, they cannot in principle operate independently. This unfortunate situation would result in poor utilization of both the platform and the detectors it services. Therefore, in order to maximize utilization of the platform and its detectors, the platform has the capability of being *partitioned*. One can think of partitioning as a mechanism of sharing the platform in such a way as to maintain the *fiction* of multiple systems operating simultaneously within a single platform¹. In this case, each system corresponds to one platform's detectors. For example, given partitioning, the calibration of different detectors can be accomplished in parallel. A DataFlow partitioned platform has the following characteristics:

- Control of each partition is mediated through an FCTS module called the *Partition Master* (FCPM). A summary of the FCPM's functionality and resources can be found in [3].
- The partition granularity is a single *crate*. However, any combination of DAQ crates can be collected together to form a partition. This would imply, for example, that the EMC could be partitioned as the Barrel, the Endcap, or both.
- The maximum number of simultaneous partitions in any one platform is a function of both the number of crates and Partition Masters. For example, the IR-2 platform has twelve Partition Masters, allowing for a maximum of twelve different partitions simultaneously.
- Within each partition, the hierarchy discussed in section 1.1 on page 9 is preserved. As one consequence, application modules within the platform are unaffected by how the platform is partitioned.
- Triggers can be shared in different partitions. The trigger lines driven by the external Trigger System are bussed to all the FCPMs², with each FCPM having its own programmable trigger mask allowing it to specify the combination of triggers to which it will respond. This would allow, for example, creation a partition which services the SVT detector to use calorimetric triggers *without* having the EMC in its partition.
- Each trigger prescale is programmable and deadtime statistics are accumulated on a partition by partition basis.

Although any combination of DAQ crates may be assigned to a partition, typically the user will want to define partitions along subsystem or detector boundaries. Shown in Figure 3 on page 12 is an example of how the IR-2 platform might be partitioned at a single point in time. In this example, the SVT and DCH are partitioned together, the EMC barrel shares a partition with the DIRC while the EMC endcap has its own partition, all three crates of the Trigger System are partitioned together, and the IFR crate is left unpartitioned. (The code to accomplish this partitioning is described in section 1.4 on page 13.)

The partitioning concept is fundamental to both the platform and DataFlow. Even stand-alone systems consisting of a single DAQ crate must assign that crate to a partition in order to make its services available.

1. A fiction, because partitions must still *share* the potential bandwidth of the platform.
2. In actuality, four of the thirty-two trigger lines are FCPM specific (see [3]).

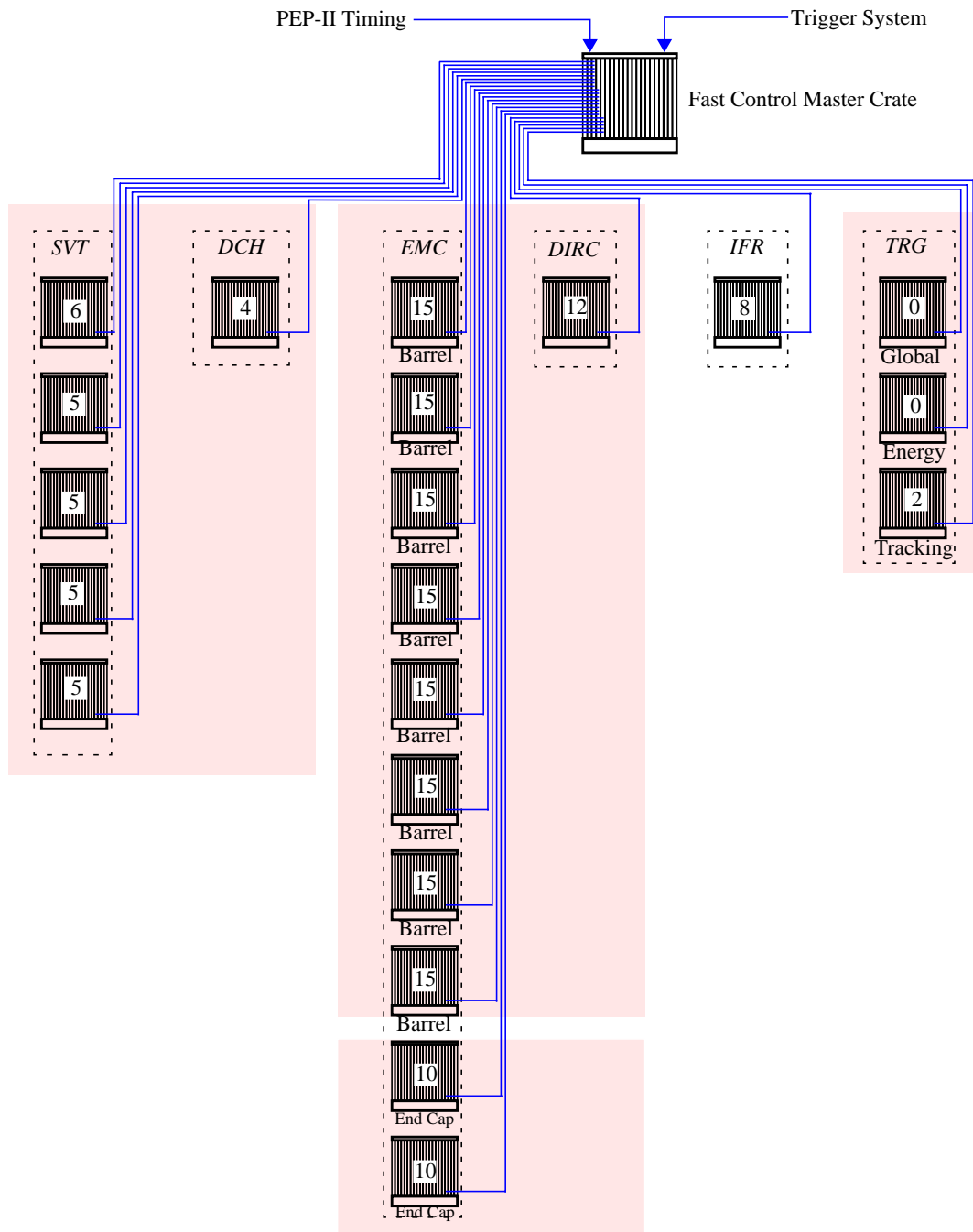


FIGURE 3. Partitioning the IR-2 platform

1.3 Managing the platform

Access to those operations which cross partition boundaries is mediated through the `dfPlatform` class. `dfPlatform` objects may be created either at the *Source* level in the platform hierarchy, or at any location external to the platform.¹ The user is free to have any

number of these objects instantiated at a single time. Since all platform usage is typically done in the context of a partition, the number of services offered by the `dfPlatform` class is necessarily small. These services include:

- Reinitialization of the platform's time (the details of which are described in Section 13.0 on page 53). The platform automatically establishes its own time upon reset. If necessary, the platform's clock can also be reset by means of the `SetTime` member function.
- Retrieving the time at which the platform was reset (via the `Creation` member function).
- Retrieving the current time (via the `Time` member function).

A `dfPlatform` object can also be used to create one or more partition proxies. A proxy can be used to query the platform for information on either a specific partition or on all its partitions. For example, to retrieve a reference to the partition currently containing the Drift Chamber:

```
dfPlatform myPlatform;
dfMap      map(myPlatform); // map the entire system
dfPartition driftChamber(map.Crates(DF_DCH));
```

If the user was interested in iterating over all the platform's partitions, the `Partitions` member function could be used to return a bit-list of the allocated partitions, as follows:

```
unsigned partitions = myPlatform.Partitions();
```

The positions at which a bit is *set* indicate an allocated partition. These bit positions could then be extracted and passed as an argument to `dfPartition`'s constructor.

1.3.1 Shutting down the platform

The *Shutdown* transition (see Section 4.0) must be accomplished through the `dfPlatform` object since this transition occurs outside the context of a partition. However, the user may specify a list of crates on the platform to be shutdown. For example, to shutdown the drift chamber:

```
myPlatform.Shutdown(map.Crates(DF_DCH));
```

For this operation to succeed, the drift chamber crates must be currently in the *Standby* state.

1.4 Establishing and monitoring a Partition

A DataFlow partition is established through an instance of the `dfManager` class [2]. The object's constructor takes as an argument the list of crates to be allocated to the partition. The `dfManager` object may be created either at the *Source* level in the platform hierarchy, or at any location external to the platform, however, for the same crate list, only a single

1. but with connectivity to the platform.

`dfManager` object can be in existence at any given time. The reason for this restriction is not only because a crate can only be in a single partition at a time, but also because the FSMs in a partition can only be sequenced from a single source.

Though each partition can only have one `dfManager`, it is perfectly reasonable to have several references (or proxies) to the partition. For example, several applications spread over different processes might want to map a given pre-existing partition (via `dfMap`) in order to retrieve information about the partition. Instantiating a `dfPartition` object provides the user with this functionality. A discussion of the usage of this class is found in Section 1.3.

The `dfManager` class inherits from the `dfSequencer` class, and uses the `dfCalibCycle` class [2]. The member functions in the `dfManager` class supersede certain member functions of the `dfSequencer` class. These member functions are invoked in order to allow user control over both partition specific application code and the partition's calibration sequencer. Further information on this functionality can be found in Section 4.0.

To create a partition, one simply instantiates an object of the `dfManager` class. The argument to the constructor for the class specifies the composition of the partition. This composition is expressed as the list of crates to be contained in the partition, with each crate being specified by a crate *number*. Crate numbers are physical addresses whose assignment can change as a function of time and platform. To ensure the portability of application code, the crate list is normally conditionalized through a logical address specification and the `dfMap` class. The most natural logical address is the detector number (See “Addresses and Tags” on page 38.). For example, to determine the crates that constitute the SVT detector, independent of platform:

```
dfPlatform myPlatform;  
dfMap      map(myPlatform);           // Map the entire platform...  
unsigned   crates = map.Crates(DF_SVT);
```

Once the platform has been mapped, various combinations of crates can be composed and used to partition a specific platform. For example, to create the partitions illustrated by Figure 3 on page 12:

```
const unsigned TRIGGER = (DF_TRG_GBL | DF_TRG_ENR | DF_TRG_TRK);  
...  
dfManager partition(map.Crates(DF_SVT | DF_DCH));  
dfManager barrel(map.Crates(DF_EM_C_BRL | DF_DRC));  
dfManager endcap(map.Crates(DF_EM_C_ECP));  
dfManager trigger(map.Crates(TRIGGER));
```

1.5 Dissolving a partition

To dissolve a partition, simply delete its instantiated `dfManager` object. Any FSMs (See section 4.0 on page 26.) running within the partition are sequenced back down to the *Standby* state. Once a partition is dissolved, the crates are once again available for the construction of new partitions.

2.0 Streams

The stream constitutes one of DataFlow's fundamental metaphors and provides for the distribution of information (both control and data) throughout the platform. A *stream* carries information quantized in units of *Datagrams*. Since a datagram employs an *Envelope/Letter* idiom, the stream has both the appearance and capability of carrying arbitrary objects. Streams are unidirectional, that is, they transport datagrams in only a single direction. This direction may either be from the control stage to the event stage (*downstream*), or from the event stage to the control stage (*upstream*). Streams always mimic the platform hierarchy, with the consequence that information propagated downstream generally requires fanning-in, while information propagated upstream must be fanned-out. A stream is normally associated with an autonomous service which, while constrained to a single partition, crosses processor boundaries. Since within a partition DataFlow provides more than one service simultaneously, it follows that within a partition many streams coexist simultaneously. These streams are differentiated by their name, with the name suggestive of the supported service. For example, the *FrameWork* stream supports the movement of event oriented data through DataFlow's Finite State Machine discipline. The *Occurrence* stream supports the Occurrence Manager by transporting out-of-band occurrences from their source to potential downstream application handlers. Since all relevant streams are automatically setup and configured by DataFlow the client is not normally concerned with either their construction or installation. Instead, the client's attention is focussed on how to access the datagrams¹ carried on the stream.

2.1 The information carried on a stream (Datagrams)

Datagrams employ an *Envelope/Letter* idiom, where the datagram represents an envelope which (potentially) contains a letter. A letter represents a single arbitrary object which may be inserted or removed from its datagram. Because of the regular, well-defined features of the datagram it may be inserted (posted) or removed (delivered) at arbitrary sites along a stream. In addition to envelope/letter behavior the datagram comes with a promise of *guaranteed* delivery. That is, the poster of a datagram will receive notification when the datagram is both delivered to and consumed by its downstream recipient. This provides the client with a mechanism to assure deallocation of any resources associated with posted datagrams. If a datagram consumer does not acknowledge delivery of a posted datagram, a time-out mechanism assures notification. A datagram is realized through the `dfDatagram` class (see [2]). The following illustration represents the datagram and its contained object:

1. Actually, the datagram's contents.

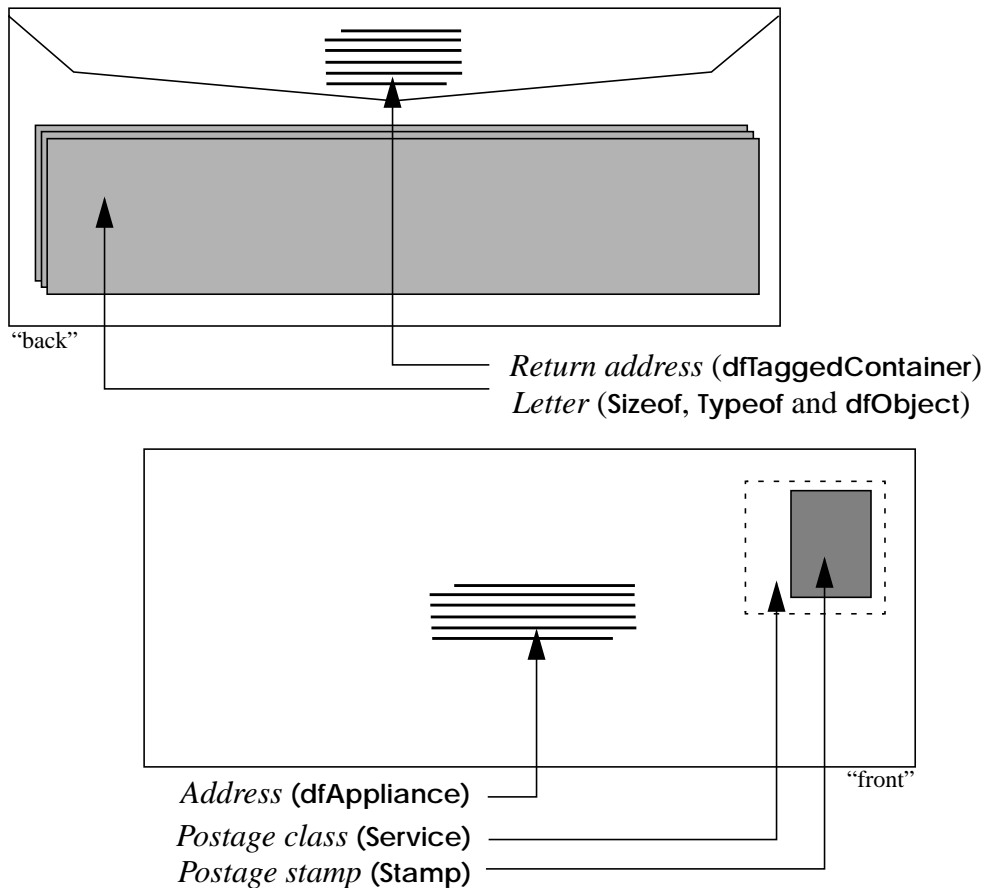


FIGURE 4. The Envelope/Letter idiom as a metaphor for a datagram

Each feature of the envelope is modeled by the class in the following fashion:

- *Postage stamp* - A value typically used as a key to assist in correlation of a sequence of datagrams. This functionality is exploited, for example, by the concentrator appliance described within section 3.5.
- *Postage class* - A small integer which enumerates the function or service the datagram offers as it moves through the stream. This number is accessed through the datagram’s **Service** member functions. Its usage depends on the service associated with the stream. For example, on the *Framework* stream, the service number enumerates a requested FSM transition. Associated with each service number are a pair of context values, accessed through the datagrams’s **System** and **Application** member functions.

- *Letter* - an arbitrary object¹ contained by the datagram. The semantics for inserting and removing the object from its datagram are inherited through the `dfFlaggedContainer` class (See “Tagged Containers” on page 43.) Without “opening” the envelope certain characteristics of the letter are known, these include its type and size. Both characteristics are accessed through the `TypeOf` and `SizeOf` datagram member functions.
- *Return Address* - specifies the source of the datagram. The access member functions for this information are inherited through `dfFlaggedContainer` (see “Tagged Containers” on page 43). These member functions allow the source address to be returned in either `DataFlow` or detector space.
- *Address* - represents the destination address of the datagram. The poster of a datagram does not actually know its final recipient. Instead, the datagram is posted to a stream. Therefore, the address represents the *name* of a stream.

Typically, empty datagrams are created and passed as arguments to application provided action methods, therefore the client is not normally concerned with their creation. For example, the `dfAction` class² is passed a potential output datagram, through the `output` argument to its `Fire` member function.

3.0 Appliances

Streams are in and of themselves relatively useless because they do not provide a mechanism which allows clients to introduce, sample, and extract the datagrams associated with a stream. This missing mechanism is provided by `DataFlow` through an abstraction called an *appliance*. An appliance is attached to either the stream or to other appliances. By attaching to the stream, the client is able to both source and sink datagrams on the stream. By attaching to other appliances, the client may serialize and coordinate stream related activity between itself and other appliances. The attachment mechanism is *regular*, in the sense that the interface remains the same, independent of the client’s location within the platform hierarchy. This interface is represented by the abstract base class `dfAppliance`

1. But must inherit from `dfObject`.
 2. Associated with the `dfFSM` class.

(see [2]). The relationship between the appliance as an abstraction and the member functions of its corresponding class is as follows:

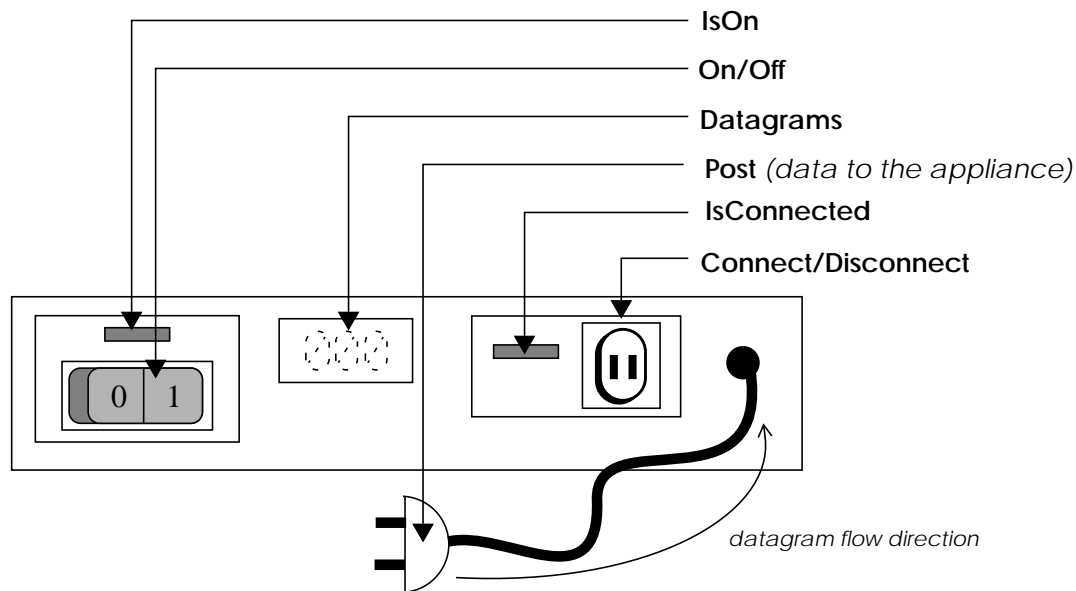


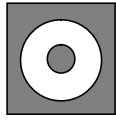
FIGURE 5. The Appliance as a metaphor for a stream attachment

An appliance's major features can be enumerated as follows:

- A *plug*, which allows the appliance to access incoming datagrams. The appliance is “plugged” into another appliance by providing a reference to the appliance as an argument to the **Connect** member function of the appliance to be plugged into.
- An *outlet*, which allows the appliance to retransmit datagrams to another appliance. The **Connect/Disconnect** member functions provide access to the outlet. An indicator is provided so that the appliance can determine whether an appliance is plugged into its outlet (the **IsConnected** member function). The appliance posts datagrams to its plugged-in appliance by calling back its **Post** member function.
- An *on/off switch* which allows the appliance to remain plugged into another appliance, but still able to accept or discard incoming datagrams. A created appliance comes up in the off state. An indicator is provided to determine whether the appliance is either on or off (the **IsOn** member function).
- A *utilization* indicator which indicates how many datagrams have been both sent and received by the appliance (through the **Datagrams** member function). In addition, the creation time of the appliance is stored and can be accessed via the appliance's **Creation** member function.

Although applications are free to invent their own appliances¹, they normally interact with a stream through a set of appliances predefined by DataFlow. These appliances are discussed below.

3.1 Inlet



*Symbol for a
Inlet*

Each processor has, for each stream running through it, two pre-installed and connected appliances, named the *inlet* and the *terminator*². These appliances “bridge” the stream between adjacent stages within the Platform hierarchy. If the stream moves datagrams *downstream*, the inlet is plugged into its upstream stage and the terminator has plugged into its outlet the downstream stage. If datagrams move *upstream*, the inlet has plugged into its outlet the downstream stage and the terminator is plugged into the upstream stage. An inlet has two functions:

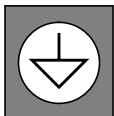
- Provides access to in-coming datagrams on a stream from its upstream stage or level. In the case of either the *fragment* or *event* level, upstream may represent more than just a single processor, therefore the inlet at these stages is actually providing access to datagrams from multiple sources³.
- Provides execution context. The arrival of a datagram normally signifies the start of an asynchronous activity by the inlet. This activity may trigger activity in the appliance plugged into the inlet’s outlet, which in turn will trigger activity in its outlet appliance, and so on. All this activity requires either a process or task under which to execute. This is a *VxWorks* task or process thread with its characteristics specified as an arguments to the inlet’s constructor. Since the inlet box is created by DataFlow automatically (see “Placing appliances” on page 23) the application is not normally concerned with this process or task creation. However, if it becomes necessary to change these characteristics, Dataflow provides a mechanism to allow the user to configure these characteristics⁴.

Since both the inlet and terminator are appliances, client appliances may plug into the inlet to receive datagrams and be plugged into the terminator to retransmit them. The mechanics of attachment require a reference to the necessary appliance. The client obtains a reference to either appliance through member functions of the `dfStream` class (see [2]). For example, to locate both the inlet and terminator for the *FrameWork* stream:

```
dfStream      stream(FrameWork);  
dfAppliance* inlet      = stream.Inlet();  
dfAppliance* terminator = stream.Terminator();
```

-
1. By inheriting from the `dfAppliance` class.
 2. The terminator is discussed in more detail below.
 3. Remember, each datagram has its *own* source address.
 4. Also described on page 23.

3.2 Terminators



Symbol for a Terminator

As is described above, the terminator works in partnership with an inlet to bridge the stream between stages in the platform hierarchy. In this role, it provides a mechanism for client appliances to insert datagrams onto the stream and consequently for those datagrams to be forwarded to their upstream/downstream stage. However, at some point the stream must terminate, therefore the appliance can also serve as a terminus or sink for any of its datagrams. The termination is active, in the sense that the client may specify what *kind* of datagrams to terminate. This termination is keyed off of the datagram's service type. For example, the following code fragment would prevent `L1Accept` transitions from penetrating past the segment stage on the *Framework* stream¹:

```
dfStream      stream(FrameWork);
dfTerminator* terminator = stream.Terminator();
unsigned previous_sinkList = terminator->Sink(L1Accept);
```

The terminator appliance allows a client to selectively turn on monitoring for any terminated datagram. If monitoring is enabled, termination will cause a termination occurrence to be generated in-band (on the stream) See Section 14.0 on page 58 for more information. For example, the following code fragment, executed at the event level would cause any `Map` or `Configure` datagram terminated, to generate an occurrence:

```
dfStream      stream(FrameWork);
dfTerminator* terminator      = stream.Terminator();
unsigned previous_monitorList = terminator->Monitor(Map|Configure);
```

DataFlow initially installs all terminators except at the event stage so that they do *not* terminate and do *not* monitor any terminated datagrams, i.e. its sink list and monitor list are set to zero. The terminator at the event stage is set to terminate and monitor all datagrams².

3.3 Generators



Symbol for a Generator

A Generator is simply an appliance whose pluggable behavior is not exploited. Conceptually it represents the source or origin of Datagrams for a particular stream. Consequently, the generator must understand the semantics of the service associated with the stream. This normally implies that:

1. Only a single generator is “on” the stream.
2. It is located at the Control stage.

1. This code is called on a segment processor.

2. With the exception of `L1Accept` datagrams which for performance reasons are never monitored.

A `dfSequencer` (see [2]) object is an example of a generator. Its function is to sequence the collection of Finite-State-Machine appliances (discussed below) connected to the *Frame-Work* stream.

3.4 Finite-State-Machines (FSM)



*Symbol for a
FSM*

This appliance works in partnership with the `dfSequencer` generator (described above) to provide disciplined access to a DataFlow partition. In this guise the FSM is normally attached to the Framework stream (see 3.7). A comprehensive discussion of the role within DataFlow of state machines in general and the FSM appliance in particular can be found in “The Finite State Machine” on page 26.

3.5 Concentrators



*Symbol for a
Concentrator*

The appliances described up to this point consider arriving datagrams as independent, and consequently may treat each datagram in isolation, i.e. to process any datagram the appliance does not depend on the datagrams that arrived either before or after it. However, this not the case with the *concentrator*. The concentrator is an appliance which assumes a correlation between datagrams on the stream. Its function is to concentrate related datagrams which arrive through its plug and to reconstitute them into a single datagram which is available to other appliances through its outlet. A sequence of correlated datagrams is called an *event*. This process may also be thought of as a fan-in. The proto-typical example of this process is concentration at the event stage (also known as event-building). However, the same process occurs at the fragment stage, where instead of concentrating crate contributions (fragments) to events, the appliance concentrates ROM contributions (segments) to fragments.

By convention, concentrators assume the correlation is described by the datagram’s *stamp*, i.e. the stamp is used by the concentrator as a *key*. The datagram’s stamp is a convenient key, because many interesting correlations are time-related and consequently the stamp represents a time. For example, when building event data, the stamp represents the time at which the FCTS synthesized the *LIAccept* corresponding to the event trigger¹.

A concentrator is represented through the `dfConcentrator` class (see [2]). `dfConcentrator` allows a certain degree of configurability in its operation to allow for the concentration of client specified objects to a client specified output container. To use this class, the client must construct two objects for *each* kind (or type) of object concentrated.

1. An output container, used to represent the concentrated result. This container must inherit from the `dfEventContainer` class.

1. Since the FCTS tags *any* command generated with a time, this example could be generalized to any concentration associated with a FCTS command, which in DataFlow includes almost *all* concentration.

2. An action to be fired when an event comes into existence. This action must inherit from the `dfEvent` class. From the concentrator's perspective this class serves two functions:
 - Instantiates the necessary output container (described above).
 - Provides an estimate of how much time it takes to concentrate any particular event. The concentrator uses this functionality to “time-out” contribution arrivals. The estimate is provided as an argument to the object's constructor and accessed through its `Duration` member function.

Once instantiated, the action is made “known” to the concentrator by passing the action as an argument to the concentrator's `Register` member function.

When the concentrator determines that an event has come into existence, it fires the registered action¹ by calling back its `Create` member functions. This member function instantiates an output container and passes it back to the concentrator. The `Create` member function is passed the number of expected event contributions, allowing an output container of sufficient size to be created. If the `Create` member functions cannot instantiate its container, it returns a NULL result. As contributions arrive for the event, the container's `Insert` member functions will be invoked. The implementation of `Insert` moves the contribution to its output container. The concentrator guarantees that the `Insert` member functions will be called back once and only once for each of its potential contributions. If a contribution does not arrive (a time-out), a datagram is synthesized with a NULL payload. Once all the contributions have been delivered, the concentrator calls the container's `ReSort` member function, allowing the client to implement any functionality required of a container once it has been “filled”. Once filled and resorted, the container is passed to the appliance connected to the concentrator's outlet.

The operation of a concentrator may be illustrated with an example. Assume, first, that the concentrator is operating at the Event level, and second, that its upstream stage is producing a sequence of “vectors of integers”, each vector described by an object of the following type:

```
class intVec: public dfObject {
public:
    int Sizeof() {return sizeof(intVec)};
    int Typeof() {return VectorOfInts;}
private:
    int result[16];};
```

The client must then define both a container to hold the sequence representing an event and an action to be fired when an event is declared. First, the container is defined as follows:

```
class myContainer: public dfEventContainer {
public:
    myContainer() {next = list;};
    void Insert(const dfDatagram& contribution);
    dfEventContainer* ReSort();
```

1. Derived through the `dfEvent` class.

```
private:
    intVec** next;
    intVec sequence[maxContributions];};
```

The container contains the list of vectors, whose current “fill level” is determined by a “next” pointer. It overrides the `Insert` and `ReSort` member functions with the following implementations:

```
void myContainer::Insert(const dfDatagram& contribution) {
    *next++ = (intvec*)contribution->payload();}

void myContainer:Resort() {*next = NULL} // Terminate the list...
```

Next, an action must be defined to create the container:

```
class myEvent public dfEvent {
public:
    dfEventContainer* Create(int number);}

dfEventContainer* myEvent::Create(int number) {
    return((number > maxContributions ? NULL: new myContainer);}
```

Finally, The actual run-time code is simply:

```
#define MICRO_SECOND 60
const int duration = (100 * MICO_SECOND)

main()
{
    dfConcentrator builder;
    myEvent          event(duration);

    builder.Register(VectorOfInts, event);
```

3.6 Distributors

To be discussed.

3.7 Placing appliances

When the platform is reset, DataFlow automatically configures and installs the necessary streams. In addition, for each stream on each processor, DataFlow instantiates and installs the appropriate inlet and terminator appliances. For the inlet appliance, a task or process thread (as appropriate) is created whose name corresponds to the service offered by the stream. The characteristics of the created task or process thread are configurable by the user. They include:

- Task or thread priority
- Task or thread stack space

For those processors placed at the fragment or segment level a default¹ concentrator appliance (again for each stream) is created and plugged into the processor's inlet². An arbitrary platform stream would then have the following topology and attachments after reset:

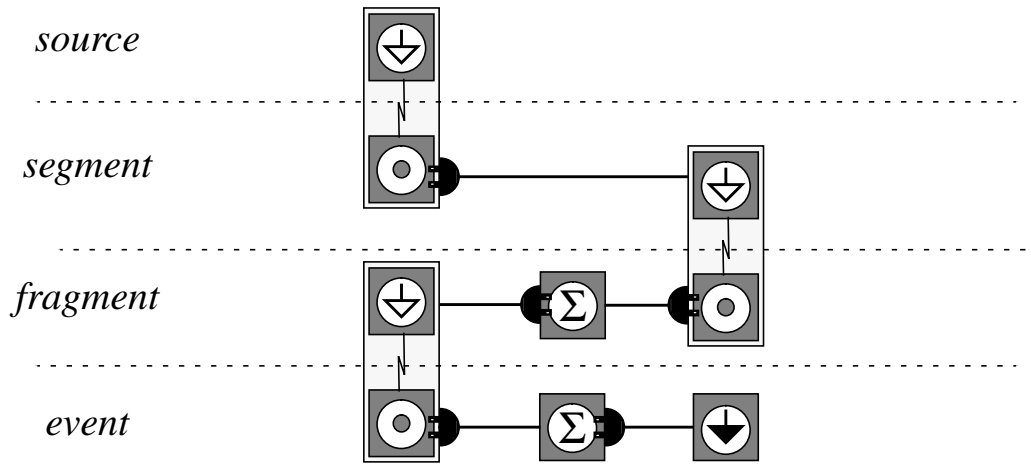


FIGURE 6. Stream after platform reset

Note that the *source* stage does not have an inlet, since it is assumed a client generator will be attached at this location. Once the stream has been configured and installed the Data-Flow client may wish to either add, remove, or replace any attached appliances. This situation occurs, for example, on the *FrameWork* stream, where the set of attached appliances varies, depending on whether the framework is taking data or running a calibration. For example, the *FrameWork* stream would have the following topology and attachments by the time it is prepared to participate in data taking:

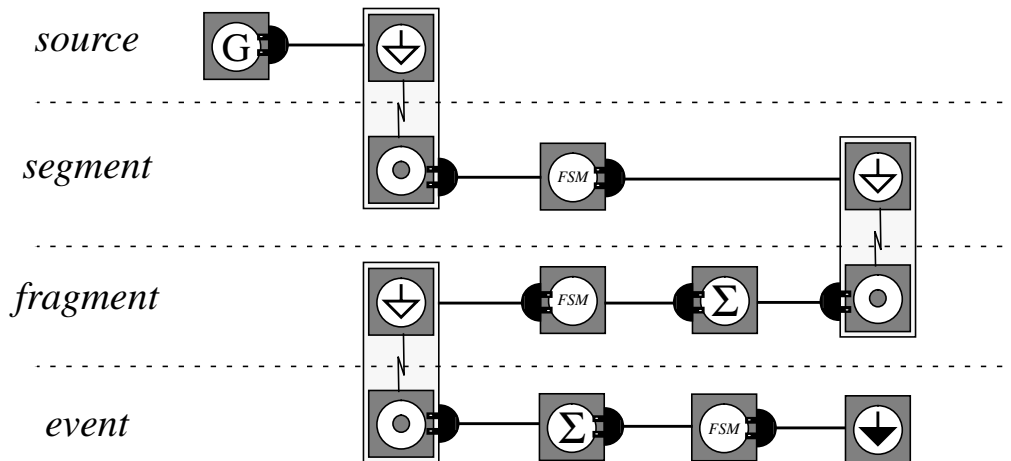


FIGURE 7. Framework stream ready for data taking.

1. Tailorable by the user.
2. This excludes segment stages which service one processor, or event stages which service a single crate.

Although each appliance contains the member functions necessary to reconfigure its attachment to a stream, the amount of plugging and unplugging required to reconfigure a stream (summed over all the attached appliances) is somewhat cumbersome. To overcome this awkwardness, the `dfStream` class provides convenience member functions to facilitate reconfiguration. For example, assume a client residing in an Event level processor. To reconfigure the stream at the Event level from its initial state to the topology and set of attachments illustrated above:

```
dfStream      stream(FrameWork);
dfFsm        dataTakingMachine;
dfConcentrator eventBuilder;

stream.Replace(stream.Concentrator(), &eventBuilder);
stream.Insert(&dataTakingMachine, &eventBuilder);
```

4.0 The Finite State Machine

The control of applications participating in DataFlow is modeled as a Finite State Machine (FSM). The Booch notation for this FSM is illustrated in Figure 8 on page 27. The *Active* state is shown in more detail in Figure 9 on page 31. Each step that participants must go through in order to take data, from power up, to *LIAccept*, and back to power down, corresponds to a transition in the FSM.

The most important of the states displayed in Figure 8 and Figure 9 are as follows:

- *Partitioned* - Partitions are created by a combination of the *Allocate* and *Map* transitions, and thus the *Partitioned* state is the first state which exists in the context of a newly created partition. In this state, both the desired set of crates and an FCPM have been assigned to the partition. States and transitions before *Partitioned* are not relevant to a partition, and are done in the context of the platform.
- *Configured* - In the *Configured* state, the partition has been assigned a set of triggers inputs to which it may respond. There are thirty-two trigger inputs in all, and any partition can be assigned any subset of the thirty-two.
- *Ready* - In the *Ready* state, the partition is available to take data for the first time. All counters have been zeroed, and any system checks that don't require an *LIAccept* may be performed. If the system is not configured correctly, the user can back out quickly using the *Abort* transition without having to go through the *End* transition.
- *Active* - In the *Active* state, the partition takes data. In reality, this state consists of number of substates which must be sequenced in order to run calibrations or take data (see Section 5.0). For the purposes of this discussion, however, it can be treated as a single state.
- *Enabled* - In the *Enabled* state, the partition is generating *LIAccepts* which drive the FSM's *LIAccept* transition.
- *Paused* - The *Paused* state permits the user to pause and resume taking data or calibrations smoothly.

The classes which *drive* the FSM have methods corresponding to all the transitions, while the classes which *enforce* the FSM permit *actions* to be associated with each transition. This FSM model is referred to as *action on transition*. The following sections describe the classes that characterize the FSM, and how they are used by applications to participate with and to manage DataFlow.

Of course, since DataFlow is a highly distributed system, it may also be thought of as a collection of FSMs arranged in a hierarchy. In terms of the *stream* metaphor, this hierarchy consists of a set of appliances attached to the FrameWork Stream. (See Section 2.0 on page 15.) However, to the application that DataFlow plugs into (be it Online Run Control or a GUI controlling a stand alone system), DataFlow appears as a single FSM controlled by the application. State machine transitions are driven either internally by the DataFlow itself or externally by the agent managing DataFlow.

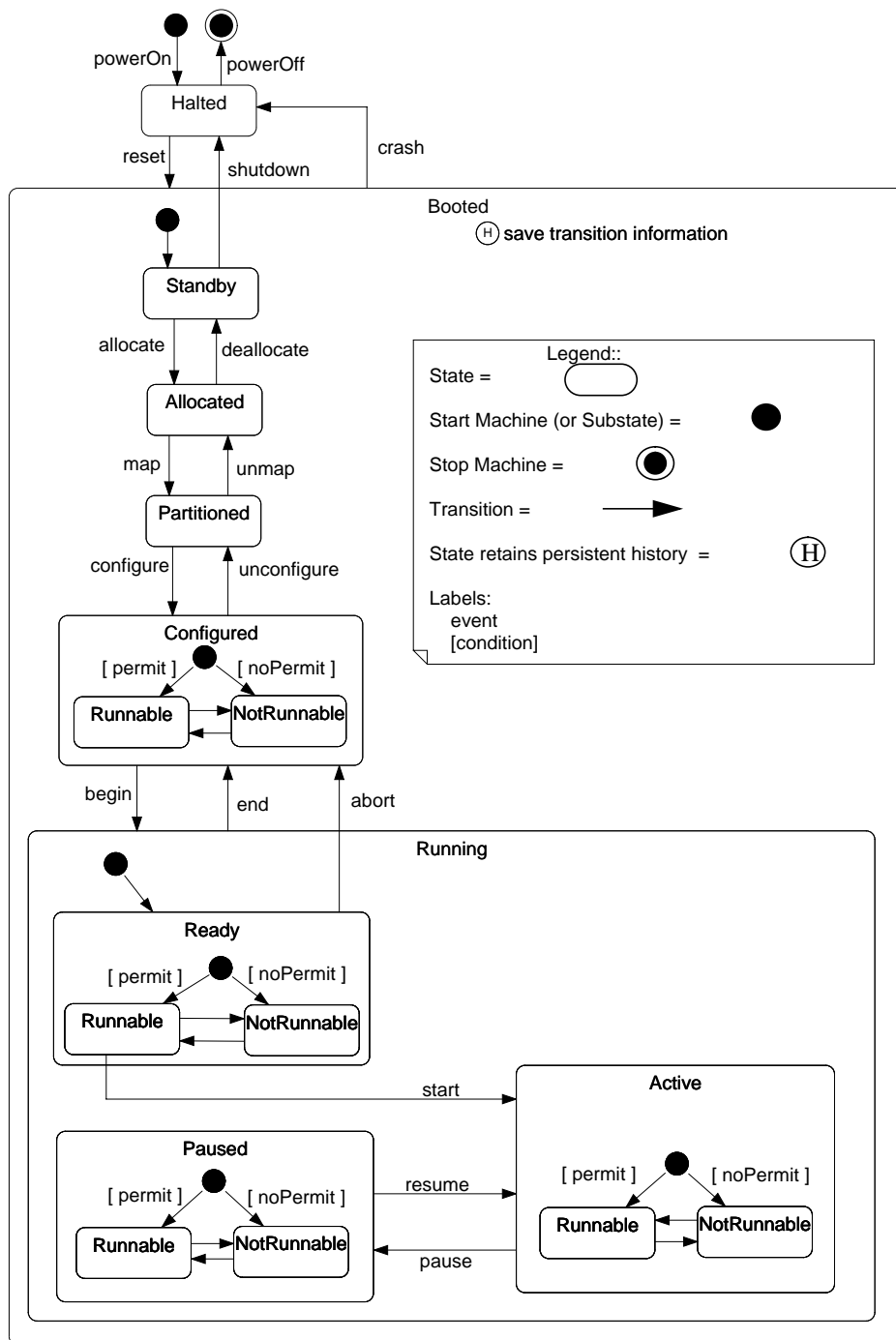


FIGURE 8. The DataFlow Finite State Machine.

4.1 The FSM Classes

The FSM is modeled in software both by classes which *drive* the FSM and by classes which *enforce* the FSM. These classes are contained in the Management Package. A user application drives (or sequences) the DataFlow FSM through the `dfManager` class which inherits from the `dfSequencer` class. The class which enforces the FSM is the `dfFsm` class. Applications which plug into DataFlow use this class to register actions corresponding to each transition. An action is a way for a user to get application code executed on a transition. Any action inherits its behavior from the `dfAction` class.

Whenever a `dfManager` object drives a transition, the actions registered on the corresponding transitions of the each of the `dfFsm` objects in the same partition are executed. If the `dfManager` object attempts to drive an invalid transition for any of its `dfFsm` objects, the manager's attempt fails. To facilitate communication between the user of the manager and the users of the FSMs, DataFlow provides a mechanism for associating a small amount of information with a transition and for delivering it to each of the partition's FSMs. This information is called the transition *context*. Thus, the manager and its FSMs work together to execute the ordered set of operations required for a partition to acquire data.

4.1.1 dfManager

The member functions of `dfManager` are listed in [2]. For each transition, there exist a single member which is overloaded to serve two functions. The first function registers an action whose purpose is to coordinate or serialize the completion of the transition. For example, assume that a `dfManager` object called `partition` has already been created¹. To register the action, `myAction`, for the *Begin* transition:

```
partition.Begin(myAction); // registers completion action
```

and then to drive or trigger the transition, omit the action argument:

```
partition.Begin(); // triggers the Begin transition
```

These two functions are utilized in the following example to sequence the manager's FSMs from the *Configured* state to the *Active* state:

```
dfOccurrence* completion = new dfOccurrence;

// Register appropriate actions
partition.Begin(completion);
partition.Start(completion);

// Trigger appropriate transitions
partition.Begin()->Wait();
if(completion->Reason() == Success)
    partition.Start()->Wait();
else
    ...
```

1. As shown in Section 1.4.

On completion, the sequencer will execute `completions`'s `Fire` method. This method lowers the completion action's synchronization barrier. Once the barrier has lowered, the caller who was blocked (through invoking the `Wait` method) continues execution by asking whether the transition succeeded. If so, the caller attempts to drive the *Start* transition. Note that, by waiting on the action's `Wait` method, the *Start* transition could not be initiated until the *Begin* transition completed.

As discussed above, `dfManager` provides context with each transition. The context has two components: *system* and *application*. Both values are determined by the user of the `dfManager` class. The application context is passed completely uninterpreted by `DataFlow` to the user of the FSMs. However, the system context is interpreted and used by `DataFlow`. The values for both types of context are specified as arguments to the transition method. The application context is an optional argument for all transitions, while the system context is a mandatory argument for certain transitions. For the *Configure* transition, the system context specifies the trigger input enable list. For the *BeginMinor* transition, the system context specifies the calibration sequence which is to be executed during the Minor Cycle¹.

The following example uses the *Configure* transition to demonstrate how the system context is used to establish the input triggers for a partition. As described in [2], trigger input enable list is a bit-list for which each bit's offset corresponds to a physical trigger input to the FCTS. The bit-list is a mask, and thus if a given bit is set, the corresponding trigger input is enabled. If the bit is not set, the trigger is ignored. Normally, triggers are specified through a logical value which is translated to the correct physical trigger input by using a look-up table which inherits from `dfTriggerTable`. (See [2].) The tracking and the energy trigger inputs are both enabled for the partition, as follows:

```
class myTriggers: public dfTriggerTable{
public:
    dfBitList_t Id(dfBitList_t logicalList); // user look-up here

#define TrkTrg 3
const int TrkTrgMask = (1<<TrkTrg);
#define EnrTrg 4
const int EnrTrgMask = (1<<EnrTrg);
...
...
partition.Configure(myTriggers.Id(TrkTrgMask | EnrTrgMask));
```

As described above, for every transition, the `dfManager` accepts as an optional argument a reference to an object the `odfContext` class. The Application context inherits from the `odfContext` class and encapsulates a value which is passed as an argument to a transition method as described above. This object constitutes the application context. The following example implements a trivial example of `odfContext`:

```
class myContext: public odfContext{
public:
    myContext(void *context){storedValue = context;}
```

1. The calibration sequence is described in Section 5.0 on page 32.

```

    void* Value() {return storedValue;}
private:
    void *storedValue;}

```

Now, to transport an application context from the manager to the partitions FSMs for the *Configure* transition:

```

myContext context(0xDEADBEEF);
partition.Configure(context);

```

4.1.2 dfFsm

The `dfFsm` class enforces the rules behind the FSM. An object of this class always maintains a well defined state (returned by its `Value` method), and responds to external requests for transitions. This class rejects invalid transitions. Typically, `dfFsm` is used as a base class from which more sophisticated classes may inherit state machine behavior. Because this class is an appliance¹, it may be instantiated and plugged into any location within a platform.

The `dfFsm` allows the application to register an action for each transition in the FSM. The action inherits from the `dfAction` class. (See [2].) For example, to define an action that just prints out the value of the input container:

```

template<class type>
class myAction: public dfAction<type> {
public:
    void Fire(type& input, dfDatagram& output);
    {cout << "A transition occurred" << endl;} };

```

Now, assume the FSM executes on a processor at the Segment level of the platform hierarchy. Then, to create and register the action defined above:

```

#define ONE_MICROSECOND 60
const int DURATION = (100 * ONE_MICROSECOND);

dfFsm myMachine;
myAction<int>* action = new myAction(DURATION);
dfAction<int>& previous;

previous = myMachine.Configure(action);

```

Now, the object `action` is registered for the *Configure* transition. The time-out for this transition is set to 100 μ sec, and `previous` is a reference to the old action. To cancel the action after it's registered,

```

action = myMachine.Configure(NULL);

```

Inside the *Configure* action example above, the following could be used to retrieve the context values which were established in the `dfManager` examples. (See Section 4.1.2)

1. Discussed in see `odfAppliance` on page 35 of [2].

```

dfState* configStateValue = myMachine.Value();
dfBitList_t crates = configStateValue->System();
void* context = configStateValue->Application();

```

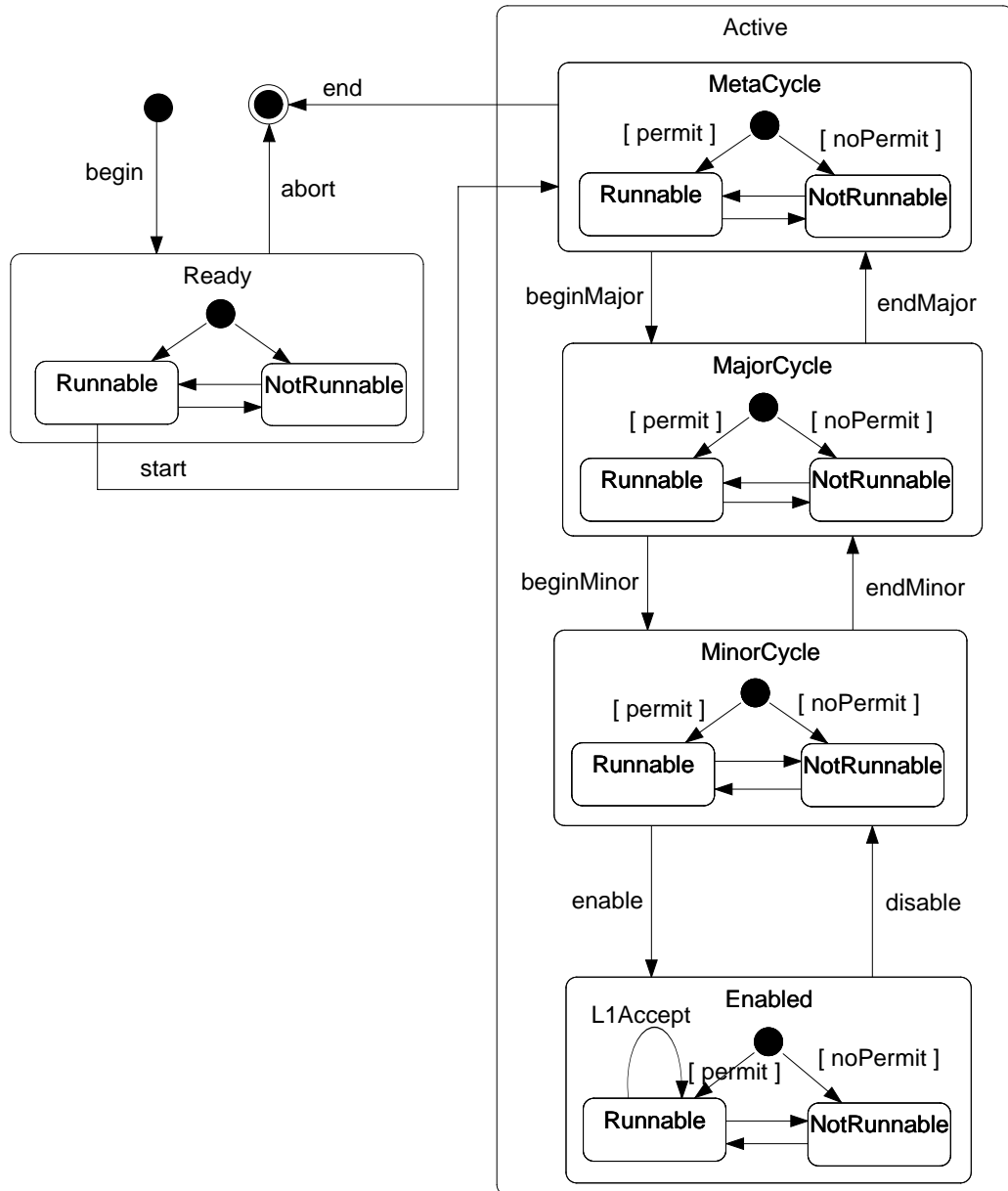


FIGURE 9. An exploded view of the Active state.

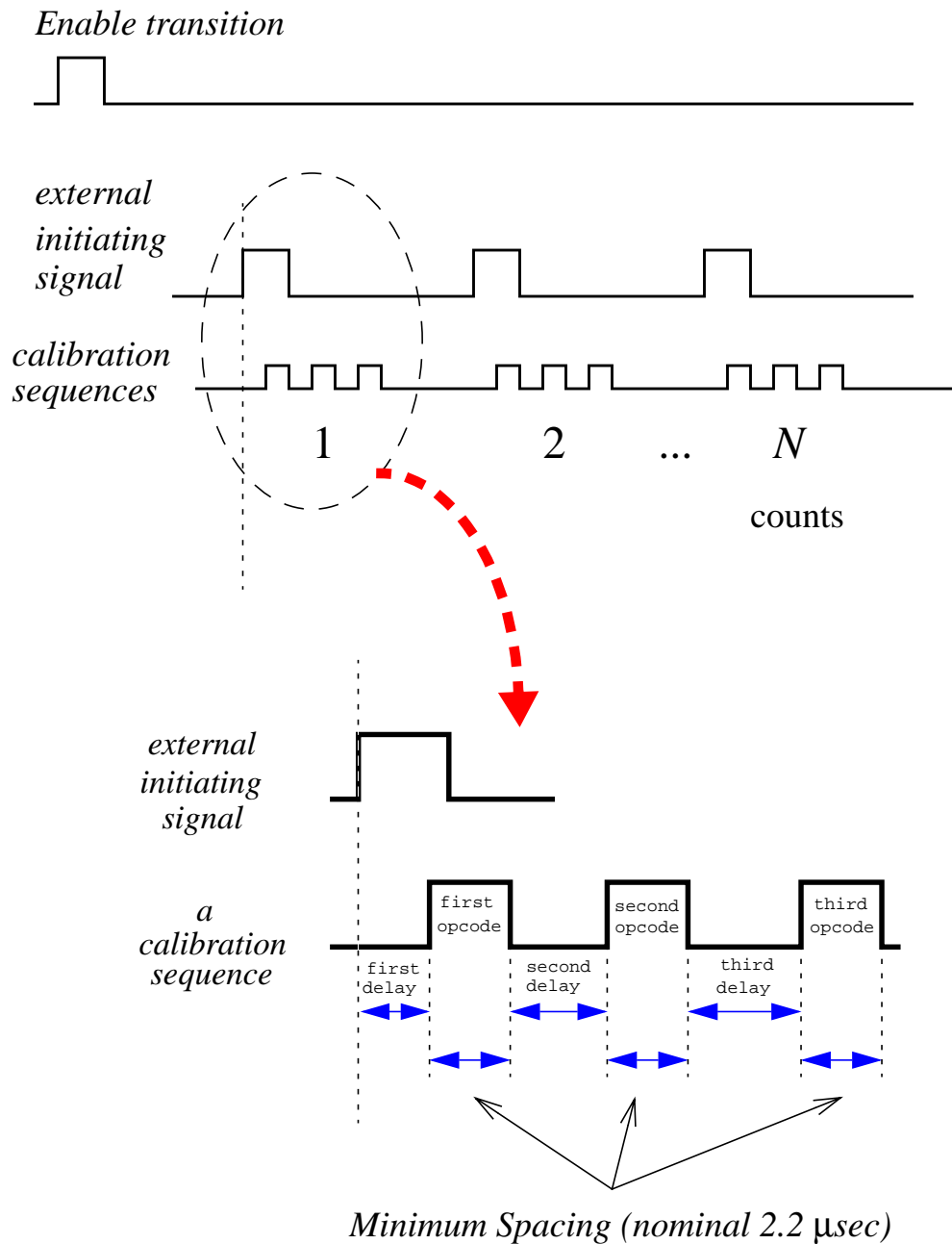


FIGURE 10. Timing structure of a calibration cycle executed in the external mode.

5.0 Calibration

Figure 9 displays an exploded view of the FSM's *Active* state. Knowledge of the details of this state is essential to understanding how to perform calibrations and take data. Both calibration and data taking require the user to sequence DataFlow through the substates of the

Active state. These substates exist so that a set of related calibrations may be performed without sequencing the FSM back to the *Configured* state. These substates and their typical usages are as follows¹:

- *MetaCycle* - A *MetaCycle* consists of a set of *MajorCycles*, and typically constitutes a complete calibration for a single subsystem.
- *MajorCycle* - A *MajorCycle* consists of a series of *MinorCycles*, typically mapping out the response of a subsystem to a range of stimuli, and leading to determination of a set of calibration constants.
- *MinorCycle* - A *MinorCycle* consists of a series of *sequences*, each consisting of a configurable number of pulses (or *commands*). A *MinorCycle* accumulates statistics on the response to a fixed stimulus.
- *Enabled* - The *Enabled* state is the state in which the *LIAccepts* are generated.

Of the above states, the first two, *MetaCycle* and *MajorCycle*, are executed entirely by software, while the *MinorCycle* and *Enabled* states are executed by a combination of DataFlow software and the hardware sequencer in the FCPM. A *MinorCycle* is executed by programming the FCPM to carry out a sequence of operations. For this reason, the *BeginMinor* transition requires the user to specify a set of parameters which program the sequencer. These parameters are encapsulated by the `odfCalibCycle` and `odfCalibCmd` virtual base classes.

`odfCalibCycle` defines a calibration cycle of calibration sequences, each consisting of up to three *opcodes* (each opcode specifying an action to be taken by DataFlow), with an arbitrary time delay before each opcode. Each delay/opcode pair is called a *command*, while a collection of these commands is called a *sequence*. An object of the `odfCalibCmd` class describes a *command*. The number of times the sequence is executed along with how the iteration commences comprises a *calibration cycle*.

The user has the option of gating the calibration cycle either under programmatic control (*internal mode*) or through logic external to the FCTS (*external mode*). The timing pattern of both modes is illustrated in Figure 10 and Figure 11. Note that the minimum spacing requirement is applied to all opcodes, while the delay values are each individually programmable. In Figure 11, once the calibration cycle has started, its timing is entirely determined by the delay arguments.

The Partition Master has connections for up to four independent external sources for the gating signal, one on the front panel and three bussed (see Figure 7 in [3]). Only one of these sources is used at any given time. The `odfCalibCycle` class' `gate` method returns an enumerated value (of type `odfCalibCycle::Source`) which specifies the source of the gate signal to be used.

In the external mode, one sequence is executed for each gate pulse received. Gate pulses are ignored until the completion of the *Enable* transition. In the internal mode, sequences

1. Thanks to GDF for this description.

execute continuously once *Enable* has completed. In both modes, the total number of sequences is determined by the *count* method.

The `odfCalibCmd` class encapsulates the *delay/opcode* pair for a single command in a calibration sequence. *delay* specifies the time delay *before* an opcode is executed (in ticks of `sysClk`), and *opcode* specifies the opcode to be executed.

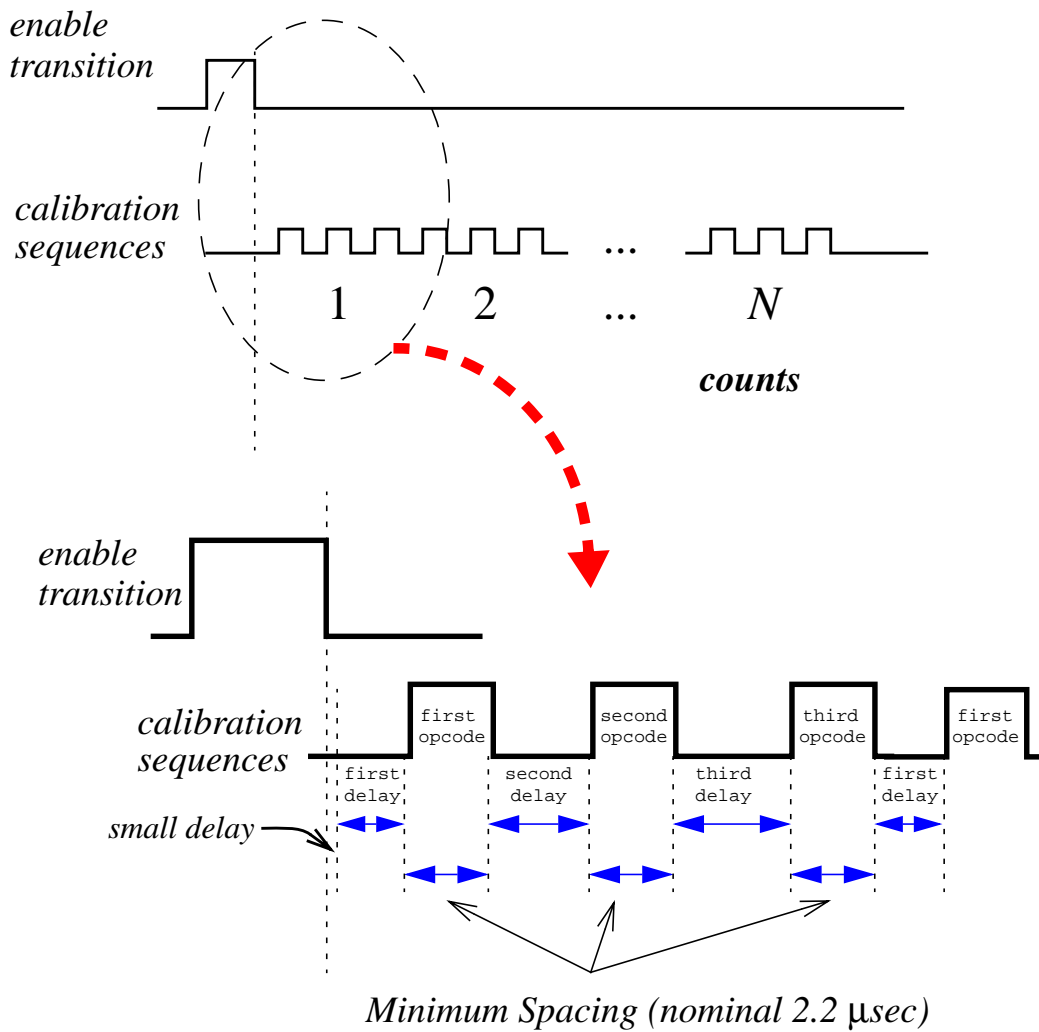


FIGURE 11. Timing structure of a calibration cycle executed in the internal mode.

The following code fragment shows how to set up and configure the FCPM's hardware sequencer. In particular, this code establishes a generic calibration sequence consisting of a *CalStrobe* followed by an *LIAccept*. These two opcodes are separated by a delay of 1 μ sec. Defining a calibration sequence is a two step process. First, one defines commands by inheriting from the `odfCalibCmd` class and providing an implementation, for example, as follows:

```
const short one_microsecond = 60; // 16.7 nsec * 60 = 1 usec
```

```

class myCalStrobe: public odfCalibCmnd {
public:
    odfCalibCmnd::OpCode Opcode() {return CalStrobe;}
    unsigned short Delay() {return one_microsecond;}

class myL1Accept: public odfCalibCmnd {
public:
    odfCalibCmnd::OpCode Opcode() {return L1Accept;}
    unsigned short Delay() {return one_microsecond;}

```

Next one assembles these commands into a cycle by inheriting from the `odfCalibCycle` class and providing an implementation. The following code fragment defines a calibration cycle consisting of one thousand iterations of a two command sequence, where each sequence is gated by an external signal applied to the FCPM's front panel:

```

const int one_thousand = 1000;
const int twoCommands = 2;

class myCycle: public odfCalibCycle {
public:
    unsigned int Count() {return one_thousand;};
    odfCalibCycle::Source Source() {return external2;};}

```

The command and cycle classes defined above are now used to describe how one might calibrate the EMC. First, one establishes the partition for the EMC, instantiates the `cycle` object, and registers actions for the appropriate transitions.

```

dfManager partition(map.Crates(DF_EMC_BAR | DF_EMC_ECP));
myCycle cycle;
// Now register action for all transitions
partition.Configure(action);
...
partition.Unconfigure(action);

```

Next one sequences the partition up to the *MetaCycle* state. Note that since the `Configure` method is called with no argument, the current trigger mask is left unchanged.

```

partition.Configure()->Wait();
partition.Begin()->Wait();
partition.Start()->Wait();

```

Then one performs the calibrations by sequencing through the *Active* state's substates. On each *BeginMinor* transition the `cycle` object is used as an argument. When all *Minor Cycles* have completed, a single calibration is done.

```

// ***This needs updating - put cycle on Enable transition
// Perform Calibration
partition.BeginMajor()->Wait();
do{
    partition.BeginMinor(cycle)->Wait();
    partition.Enable()->Wait();
    partition.EndMinor()->Wait();
} while(); // Major Cycle not done

```

```
partition.EndMajor()->Wait();
```

Finally, the partition is sequenced back to the *Partitioned* state in order to allow the user to configure the partition for a different mode. An example of such a different mode might be taking data, as described in Section 6.0.

```
partition.Stop()->Wait();
partition.End()->Wait();
partition.Unconfigure()->Wait();
```

6.0 Taking Data

Now assume that one is in the *Partitioned* state after performing the EMC calibrations described in Section 5.0. In order to take data with the partition, one must first define a new cycle consisting of a single command. Defined below is an appropriate command (consisting of a *LIAccept* opcode with zero delay) and an appropriate cycle (consisting of infinite iterations with no external gate source):

```
const int noDelay    = 0;
const int oneCommand = 1;

class dataTakingCommand: public odfCalibCmnd {
public:
    odfCalibCmnd::OpCode Opcode() {return dfL1Accept;}
    unsigned int Delay() {return noDelay;}}

// ***This is obsolete***
class dataTakingCycle: public odfCalibCycle {
public:
    unsigned int Count() {return dfForever;}
    odfCalibCycle::Source Source() {return internal;}}
```

Then one configures the partition's trigger mask on the *Configure* transition, and sequences the partition up through the *Enable* transition to start generating *LIAccepts*. When it is time to stop, one simply sequences from *Disable* through *End*.

```
dataTakingCycle dataCycle;

// ***This is obsolete***
partition.Configure(myTriggers.id(...))->Wait();
partition.Begin()->Wait();
partition.Start()->Wait();
partition.BeginMajor()->Wait();
partition.BeginMinor(dataCycle)->Wait();
...
partition.Enable()->Wait();
...
// And when it's time to stop
...
partition.Disable()->Wait();
partition.EndMinor()->Wait();
partition.EndMajor()->Wait();
```

```
partition.Stop()->Wait();  
partition.End()->Wait();
```

7.0 An Introduction to Containers in DataFlow

Tagged container objects are the fundamental building blocks for data organization within DataFlow. Any data to be transported by DataFlow must be inserted into a tagged container and any data delivered by DataFlow comes stored in a tagged container¹. This minor restriction allows DataFlow to present a regular interface throughout its levels and to transport data with a common scheme. The `dfContainer` Package specifies the API interface for the tagged container and its related classes.

A tagged container (`dfTaggedContainer`) can hold an arbitrary object² and a collection of tagged containers. The fact that containers can hold other containers allows the construction of a hierarchy of containers reflecting the fundamental organization of the data as it is built and processed by the various stages of DataFlow. A hierarchy of containers is navigated with an iterator object (`dfTcBrowser`) which can be thought of as a “smart pointer”.

To assist the iterator and to provide context for its contents, a tagged container maintains its location within the platform hierarchy (its *address*) and an additional *tag* which typically encodes information about a detector component. The combination of the information from all the levels above any tagged container in the hierarchy is called the *path* to that container. The path is analogous to a directory path within a file system and uniquely identifies the container’s location within the hierarchy.

Tagged containers themselves will be discussed in Section 9.0, “Tagged Containers,” on page 43. The identification and naming scheme will be discussed in Section 8.0, “Addresses and Tags,” on page 38. The organization imposed by using tagged containers with this naming scheme to build the event hierarchy structure will be discussed in Section 10.0, “Event Hierarchy,” on page 46.

8.0 Addresses and Tags

Every DataFlow component in a DataFlow Platform maintains two versions of its identity. These are known as the *address* and the *tag*. The address specifies the location of the component within the platform. For example, the crate number and slot number are both addresses. The tag generally indicates the location on the detector to which an electronics component is attached. For example, the EMC barrel detector identification and a DCH quadrant number are tags. An address from each level in the DataFlow can be strung together to form a complete and unique identification of the address of any component. This complete string of addresses is called an *address path*. This convention also applies to tags where a complete specification is called a *tag path*. Any particular address path (or tag path) can also be thought of as a value in *address space* (or *tag space*). For a given level in the address path, the tags are not guaranteed to be unique. For example, multiple crates can each have the same detector number. However, for both spaces the components under

1. Through a datagram, see Section 2.1 on page 15.

2. Here, arbitrary means that the object’s class inherits from `dfObject`.

a common parent are guaranteed to be unique. For example, in address space the slot numbering is unique for each crate, and in tag space module numbering is unique for each detector.

The notion of tags and addresses generally applies only in the context of a DataFlow platform which has a naming scheme for the levels as listed in Figure 12 on page 39. Each of the fields in both of these spaces will be discussed in more detail in the following sections.

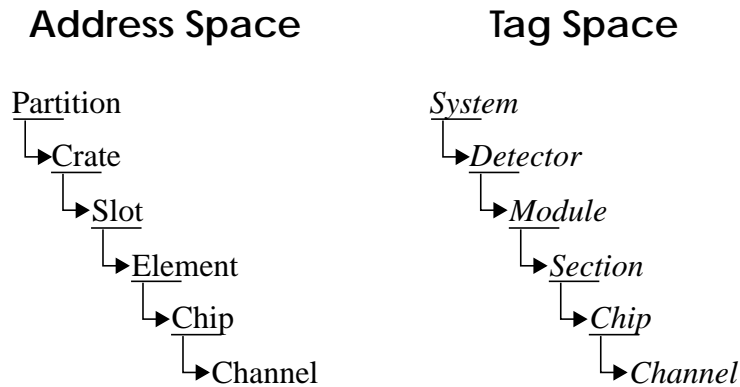


FIGURE 12. Event Hierarchy Naming Scheme

An address or tag also has two arbitrary strings associated with it. These are referred to as the *name* and the *nickname*. The name should be suggestive of the component represented but there is no absolute rule on the formation of a name string. A nickname simply provides an alternate version of the name string that is usually an abbreviation of the name. The name and nickname strings are not stored or transported with the tagged containers. Instead, they are configured and stored separately into lookup tables which use the `dfLabel` abstract base class as an interface. This class can convert string to value and value to string for any particular address or tag object.

8.1 Address Field Definitions

Each field in an address path is specified by a small integer. An address can be manipulated through a class derived from the `dfAdr` class that is appropriate to the field the address represents. These six level specific classes allow access to the `dfAdr` and `dfLabel` interfaces.

TABLE 1. Address Field Definitions

Field	Origin	Function	Range	Reserved
Partition	Set by the FCPM's id number. The partition master is selected when the partition is created.	Locates a partition within a DAQ system.	0-15	0=unknown 15=stored object
Crate ^{a b}	Set by the crate number that is encoded into each crate's FCDM. This value also corresponds to the FCPR output number from the Master Crate.	Locates a crate within a platform.	0-31	0=unknown or single crate system 31=stored object
Slot	Set by the slot number in a Slave Crate. The slot value is set by jumpers on the J3 backplane and read back through the ROM's Controller Card.	Locates the slot number within a crate.	0-31	0=unknown 31=stored object
Element	Set by the Front-End-Element input electronics on the ROM's Personality Card and Controller Card.	Locates the input and control buffers within the Personality Card and the Controller Card of the ROM and the electronics to which it is attached.	0-31	None
Chip ^{c d}	Built into the detector specific digitization and read-out electronics. Information about the electronics is stored into non-volatile RAM on the ROM.	Locates a detector specific piece of electronics within an element	0-31	0=unknown
Channel ^c	Built into the detector specific digitization and read-out electronics. Information about this electronics is stored into non-volatile RAM on the ROM.	Locates a detector specific digitizer.	0-255	0=unknown.

- a. Crate 31 is reserved for addressing the CPU's which exist at the Event stage.
- b. If the system does not contain an FCPR (i.e, a single crate system), the crate address is set to 0.
- c. The Chip and Channel assignments are not under the control of DataFlow, consequently DataFlow can not directly read these from the detector specific electronics. Their assignment for both the address and the tag paths is stored in non-volatile memory on the ROM and reloaded as required.
- d. The Chip level of organization may not be present in all detectors. When it is present, the values are reflected either by values in the header portions of the raw data stream from the front end section or by the numbering scheme of the channels.

8.2 Tag Field Definitions

Each field in a tag path (except for the system number) is specified by a small integer. A tag can be manipulated through a class derived from the `dfTag` class that is appropriate to the field the tag represents. These six, level specific, classes allow access to the `dfTag` and `dfLabel` interfaces.

The system tag is a special exception to the general rules for tags. The system value represents the notion of a collection of detectors. The system also has a version number with an initial value of one (1) which must be incremented whenever any changes are made to the tag space. The most obvious example of a system is the central detector housed in IR-2 (named *Babar*). However, the system component is used to represent any geographically qualified installation. For example, a stand-alone test stand at LBL for SVT checkout, or the Calorimeter Electronics testing at Imperial College would each constitute individual systems.

The system number is not carried as a field in the tag path value since it does not directly identify or locate any component within the hierarchy or platform. This value is, however, extremely important in interpreting the data contained within the hierarchy, in configuring the components of the platform, and in storing or retrieving the results of calibration. Consequently, the system tag value is stored within each tagged container hierarchy.

TABLE 2. Tag Field Definitions

Field	Origin	Function	Range	Reserved
System ^{a b}	A value in permanent storage and not directly part of a tag path. Changed only when the system configuration changes.	Identifies a collection of detector electronics which are serviced by a platform.	0-255	0=unknown
Detector ^{b c}	Set for an entire crate by dip switches in the crate's FCDM.	Identifies which detector the crate belongs to.	0-31	0=unknown
Module ^d	Set by dip-switches on the ROM. The value set must be unique across all of the crates belonging to a particular detector.	Identifies the portion of the detectors's electronics to which this ROM is attached.	0-255	0=unknown
Section	Built into the detector's front-end read-out electronics	Locates the particular section of the detector to which a subsystem specific electronics component is attached.	0-31	None
Chip ^{e f}	Built into the detector specific digitization and read-out electronics. Information about the electronics is stored into non-volatile RAM on the ROM.	Locates a detector specific ASIC sized piece of electronics within a section.	0-31	0=unknown
Channel ^e	Built into the detector specific digitization and read-out electronics. Information about this electronics is stored into non-volatile RAM on the ROM.	Locates a detector specific digitizer.	0-255	0=unknown

a. A system number is also assigned to stand alone and test systems.

b. The assignment of this number is managed by DataFlow.

c. The detector tags have a common set of names and nicknames as shown in Table 3 on page 43. The detector name and the subsystem name are the same if there is no meaningful division within the subsystem.

d. The values assigned to this field are completely arbitrary and are left to the discretion of the subsystem user. The only restriction is that the value used for any particular component must make the path value to that component unique.

e. The Chip and Channel assignments are not under the control of DataFlow and DataFlow can not directly read these from the detector specific electronics. Their assignment for both the address and the tag paths is stored in non-volatile memory on the ROM and reloaded as required.

f. The Chip level of organization may not be present in all detectors. When it is present, the values are reflected either by values in the header portions of the raw data stream from the front end section or by the numbering scheme of the channels.

TABLE 3. Assignment of Detector Names and Nicknames

Detector (Name)	Subsystem	Nickname
<i>Silicon Vertex Detector</i>	SVT	SVT
<i>Drift Chamber</i>	DCH	DCH
<i>Detector for Internally Reflected Cherenkov light</i>	DRC	DRC
<i>Electromagnetic Calorimeter Barrel</i>	EMC	EMC_BRL
<i>Electromagnetic Calorimeter End-Cap</i>	EMC	EMC_ECP
<i>Instrumented Flux Return</i>	IFR	IFR
<i>Trigger-Global</i>	TRG	TRG_GBL
<i>Trigger-Tracking</i>	TRG	TRG_TRK
<i>Trigger-Energy</i>	TRG	TRG_ENR
<i>Online Event Processing</i>	ONL	OEP

8.3 Assigning Addresses and Tags

In general, subsystem users maintain control over tag values and their initial assignment for their own detector, while DataFlow maintains control over all address values and their assignment. However, once a tag has been defined and assigned it is DataFlow's responsibility to apply the tag to the appropriate component to ensure that the set of all tag paths in any given system are unique and consistent with the correct operation of the detectors in the system. To this end, DataFlow serves as a clearing-house or registration service for the user's tags. It should be emphasized that the definition and assignment of both tags and addresses are considered an immutable operation within any system version (i.e. until the system number changes).

9.0 Tagged Containers

A tagged container object serves as the basic structural unit of data organization within DataFlow. Tagged containers are distinguished from one another by having an address label¹ and a tag label. A tagged container can hold a set of other tagged containers. This imposes a multiple branch hierarchy or tree structure on tagged containers. Further identi-

1. See Section 8.0, "Addresses and Tags," on page 38 for the description of these labels and their relationship to each other.

fication information comes from the relative containment arrangement of the members of the set by providing context in which to interpret the address and tag. The particular organization used within DataFlow will be described in Section 10.0, “Event Hierarchy,” on page 46. A container keeps track of the hierarchy level of its children. The level must be homogeneous for the children of a single parent. This feature allows levels within the event hierarchy to be skipped for particular purposes or subsystems.

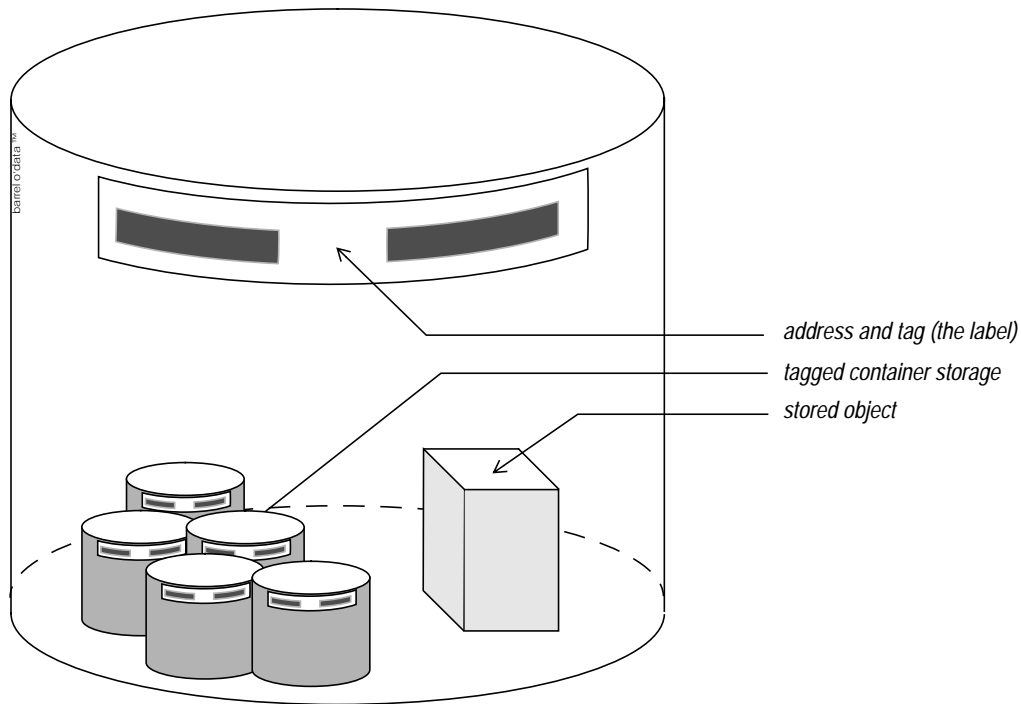


FIGURE 13. Notional Picture of a Tagged Container

Additionally, a tagged container can hold any object which inherits from `dfObject`. This aspect of the tagged container allows it to carry a payload which is distinct from its stored tagged containers. The distinction provides a clean separation between the tagged container’s contents and its structure, allows traversal of a hierarchy built from tagged containers with a straightforward iterator, and makes context dependent interpretation of the contents possible.

Figure 13 on page 44 illustrates these concepts with a notional picture of an isolated tagged container object. The figure shows the object itself as a container labeled with both an address and a tag. The container contains a box representing an arbitrary object and some little labeled containers representing stored tagged containers. The big container itself could in turn be placed into another container, just as the little containers can in turn contain other containers.

The shape and layout of a simple example hierarchy with three levels is shown in Figure 14 on page 45. This hierarchy does not correspond directly to the organization of

the platform but is intended only to illustrate several key points. The container at the top of the hierarchy does not have a parent container. The missing context for the address and tag resolution is provided by the location within DataFlow where this hierarchy is examined. None of the containers at the bottom of the hierarchy in the example have child containers, but all of them have stored objects. An empty container at the bottom of the hierarchy is also allowed. The stored objects are usually channel oriented data objects.

Containers above the bottom of the hierarchy can either contain stored objects or not, completely at the user's discretion. As an interesting example, they could be used to store summary information for a level in the parent or they could be used to store a different kind of information than the container's children store. The containers in the bottom level of the example are divided between two parents. This division matters for both the navigation of the hierarchy and the interpretation of the containers and their contents.

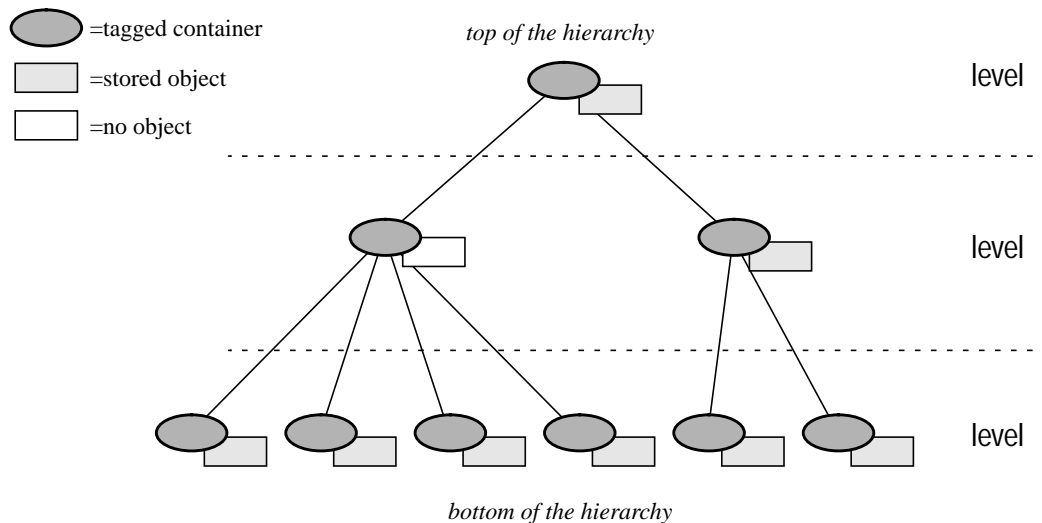


FIGURE 14. A simple tagged container hierarchy

Access to the members of a hierarchy and their contents is through an associated iterator object called a hierarchy *browser*¹. A browser is created from the `dfTcBrowser` for a particular hierarchy. Multiple browsers can be created for the same hierarchy if required. A browser has a set of overloaded operators which allow the traversal of a hierarchy in a variety of ways. The browser allows navigation and access by always internally “pointing”² or “locating” itself at some container in the event hierarchy.

From this container, the browser can be “pointed” at a different container by using a set of operators and functions whose rules depend upon the particular subclass of `dfTcBrowser` from which it was created. The different subclasses provide different kinds of navigation

1. The term *browser* is used in place of iterator to reflect the fact that navigation can up and down the levels as well as across the levels.
 2. The term pointer is not meant to suggest an internal implementation of the browser class.

by changing the rule by which the next container to which the browser points is determined. The `dfAdrBrowser` class implements the `dfTcBrowser` class by providing an address based meaning for “next”. This is the most commonly used and fastest browser implementation. The `dfTagBrowser` class implements the `dfTcBrowser` class by providing a *tag* based meaning for “next”. Independent of the meaning of next, these operators act on the browser in a manner that is consistent with the way they act on simple pointers and, in effect, “move” the browser’s internal container pointer so that it points to another container in the same hierarchy.

A browser’s operators are defined as follows:

- The `++` and `--` operators will traverse the containers at the same level and in the same parent container as the current container.
- The `<<` operator will traverse up a hierarchy by one or more levels from the current container’s level.
- The `>>` operator will traverse down a hierarchy by one or more levels from the current container’s level.
- The `*` operator will de reference the browser’s internal pointer and return the object held by the container.
- The `->` operator allows access to the container’s member functions (as opposed to the browser’s member functions).

10.0 Event Hierarchy

During the building of a complete system event, tagged containers are added to a hierarchy which has a shape that matches a subset of the platform’s shape. The shape for a complete event is shown schematically in Figure 15 on page 47. This hierarchy is always formed in address space since that is the way in which the platform components are hooked together. As shown in the figure, each level in the hierarchy can be sparsely populated, possibly on an event by event basis. An early level in the platform sees only a portion of the complete hierarchy corresponding to its own level and below.

To allow the event hierarchy to be built, six specific kinds of tagged containers are defined and derived from the `dfTaggedContainer` base class. The containers correspond to the levels in the address space of the hierarchy and allow customization of the containers implementation specific to both its position in the hierarchy and the processor on which it is used. These derived classes do not add additional behavior.

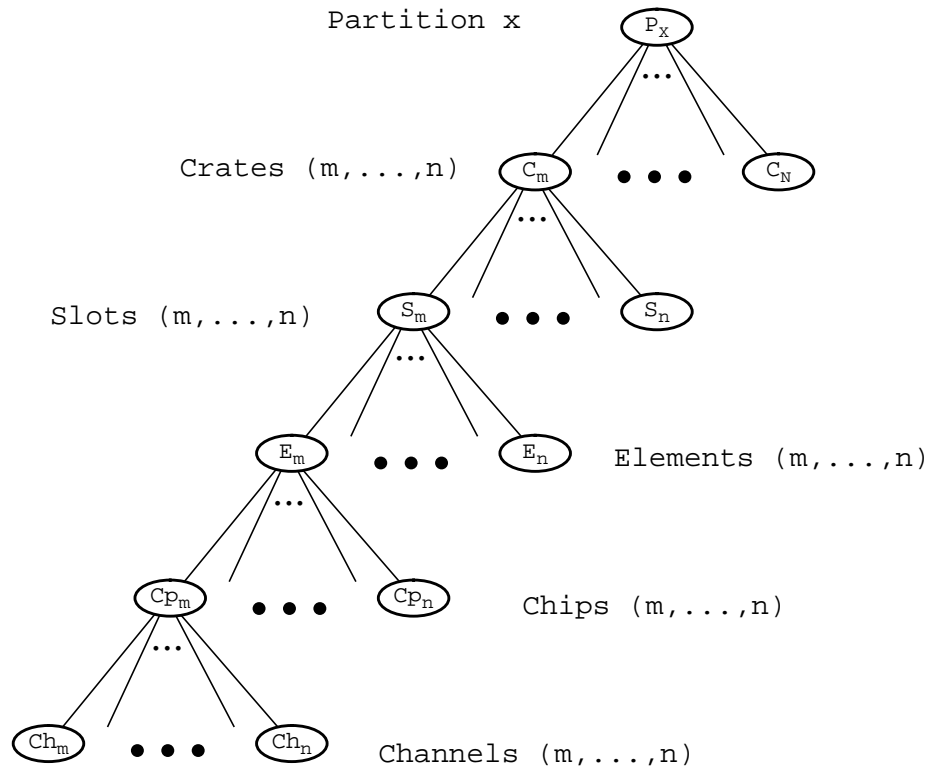


FIGURE 15. DataFlow's Event Hierarchy in Address Space

10.1 Maps

For purposes of understanding and determining the scope of a hierarchy or portion of a platform, a *map* of any portion of the complete hierarchy may be built using the `dfMap` class. A map is essentially a hierarchy of empty (containing no data object) tagged containers which is fully populated (containing the maximum possible number of children for that platform). A map can only be formed for the platform level on which the map object is created or below. The normal tagged container and browser functions are available to the map class. A map effectively gives the user three kinds of information. First, it gives the complete set of possible children under any particular tagged container in the hierarchy. Second, it allows translation between address and tag spaces for any relevant values. Last, it provides a storable picture of a hierarchy suitable for revisiting at a later time through the `dfStoredMap` class.

10.2 Input Event Hierarchy

Each of the detectors also has a unique input event hierarchy structure which is handed to an *LIAccept* action on each of the Segment level processors (the ROMs) as a slot (module) sized tagged container hierarchy. The structure and mapping of these hierarchies is summarized in Table 4 on page 49. The actual data from each of the detectors are in stored objects at the channel level. Each detectors has one kind of stored object that encapsulates a channels worth of data and inherits from `dfObject`. The objects are shown below^{1 2}. Additional stored summary objects will encapsulate the chip and element data if they are part of the detector's data stream.

On each ROM, DataFlow will insert all of the objects for a particular *LIAccept* into a slot hierarchy appropriate for the detector. These hierarchies constitute the smallest granularity of hierarchy handed to a user of DataFlow on an *LIAccept* transition's action routine. All of the hierarchies consist of a set of Sections (*Elements*). As can be seen in Table 4, the meanings of the various fields in these hierarchies do not correspond exactly to the general meaning in the address and tag paths although the structure of the hierarchy is the same so that the same tagged container and browser machinery can be used.

```
class dfEmcSample : public dfObject {
}

class dfSvtHit : public dfObject {
}

class dfSvtSummary : public dfObject {
}

class dfDchHit : public dfObject {
}

class dfDchSummary : public dfObject {
}

class dfDrcHit : public dfObject {
}

class dfIfrHit : public dfObject {
}

class dfTrgGbl : public dfObject {
}

class dfTrgTrk : public dfObject {
}
```

-
1. Without the implementation of the `dfObject` methods.
 2. The layout of these classes depends upon the way in which the detector specific electronics function, the way in which the Personality Card transports and reformats the data into the intermediate store, and the way in which the DMA supervisor reformats the data into the ROM's main store. The uncertainty in these areas precludes a realistic listing of the class structures.

```
class dfTrgEnr: public dfObject {
}
```

TABLE 4. Input Event Hierarchies Mapping at the Segment Level

Detector	Slot or Module	Section or Element	Chip	Channel	Stored Object
EMC_BRL	I/O Board ^a	Fiber	—	Sample	24 encoded channel ADC values, one from each of 24 crystals, plus encoded trigger primitive information.
EMC_ECP	I/O Board ^b	Fiber	—	Sample	24 encoded channel ADC values, one from each of 24 crystals, plus encoded trigger primitive information.
SVT	2 Detector Modules	Read-Out Section	The Chip	Strip	An FADC waveform plus embedded TDC hit values
DCH	Quadrant	Read-Out Unit ^c	Elephant	Wire	An FADC waveform plus embedded TDC values
DRC	Sector	Section	—	PMT	An ADC value.
IFR	Barrel or Endcap	Read-Out Section	Front-End Card	Strip	A TDC value.
TRG_GBL	?	?	?	?	?
TRG_TRK	?	?	?	?	?
TRG_ENR	?	?	?	?	?

a. (EMC) I/O Board – A row along the barrel.

b. (EMC) I/O Board – An Endcap “chunk”.

c. (DCH) Read-Out Unit – one of four hardware layers within a 1/16 th phi wedge.

TABLE 5. Number of components in the Input Event Hierarchies

Detector	Slot or Module	Section or Element	Chip ^a	Channel
EMC_BRL	120	3	—	64
EMC_ECP	20	2	—	64
SVT	26	8	16	128
DCH	4	16	3-23	8
DRC	12	?	—	64
IFR	4 barrel, 4 end-cap	4 barrel, 5 end-cap	6	16
TRG_GBL	1	?	?	?
TRG_TRK	1	?	?	?
TRG_ENR	3	?	?	?

a. Maximum number of chips per section. The maximum depends on the particular section for some detectors.

10.3 Applying the Event Hierarchy in Reverse

(fan data out to a platform)

(Place holder for a future discussion.)

11.0 Using The Event Hierarchy

Code example of processing an input container, coming soon.

12.0 Front End Interface

Each of the subsystems will require access to their respective front end electronics by injecting onto the C-link through the Readout module. A set of classes has been designed to provide this functionality through an interface analogous to a device driver.

The driver class `odfFeeDriver` provides a set of methods which have a familiar *device driver-like* syntax. In addition to the driver class, a set of pure virtual *request oriented* classes `odfRequest`, `odfWrite`, `odfReadFixed`, `odfReadVariable` provides a natural interface for those classes which will inherit from them to provide the user's functionality. There are two modes in which the driver can operate - synchronous and asynchronous. The `issue()` method provides a synchronous interface by returning only upon completion of the command, and the `queue()` method provides an asynchronous interface by utilizing a queueing

mechanism which DataFlow provides. Each of these methods exists in several variations to allow full flexibility in calling any of the several types of commands: request, write, readFixed and readVariable (see the class definitions).

Let's illustrate the use of these classes with an example pertaining to the drift chamber where we reset the TDCs on one Front End Assembly.

Consider the following (incomplete) class declaration:

```
class tdcRst : public odfRequest
{
public:
tdcRst() : odfRequest(odfOpcode::SS4, odfOpcode::ADR1){}
~tdcRst(){}
void fire(unsigned status, int size){//The request has completed}
};
```

and the instantiation of the class:

```
tdcRst mytdcRst;
```

Given a driver instantiation (with proper binding) of:

```
odfFeeDriver myDriver(anOdfTaskObject);
```

then the command to perform a TDC reset on all ELEFANT ICs simply becomes either:

```
myDriver.issue(mytdcRst); //returns only on completion

myDriver.queue(mytdcRst); //returns immediately,
myDriver.issue(myFifoRst); //could do other things!
```

Now let's do another example where we write the ELEFANT IC setup registers in one Front End Assembly. The registers are loaded with a serial stream of 55 bits but the stream is downsampled at the Front End Assembly to ~1.86MHz from the C-link speed of 59.5MHz, so the total size of the transfer to the Front End Assembly is $55 \text{ bits} \times 32 / 4 = 220 \text{ bytes}$. So, let's now consider the following (again incomplete) class declarations:

```
class eleSetup : public odfWrite
{
eleSetup(unsigned char* buffer) : odfWrite(buffer, 55*32/4),
odfRequest(odfOpcode::SS0, odfOpcode::ADR3){}
~eleSetup(){}

void fire(unsigned status, int size)
{ //check the completion status of the request; }
};
```

and the instantiation of the class:

```
eleSetup myeleSetup(mysetupdata.buffer());
```

mysetupdata is a completely user defined and implemented object which would encapsulate the downsampling and deliver a properly populated buffer of size 220 bytes to the `odfWrite` class via the `eleSetup` class.

Once again, given a driver instantiation (with proper binding) of:

```
odfFeeDriver myDriver(anOdfTaskObject);
```

then the command to setup the ELEFANT ICs simply becomes one of:

```
myDriver.issue(myeleSetup); //returns only on completion  
  
myDriver.queue(myeleSetup); //returns immediately,  
myDriver.issue(myFifoRst); //could do other things!
```

There are perhaps two more cases to consider, that of the `readFixed` and `readVariable`. Further examples to follow, stay tuned to this Bat Channel.

13.0 Role of Time in DataFlow

This section is a discussion of PEP-II and BaBar timing issues as they relate to data acquisition. In particular, this section will discuss how time is

- defined within the DataFlow Platform
- derived from external sources (for example, the PEP-II timing system)
- used both by DataFlow and its clients.

Typically a DataFlow platform counts time independently in each of its components, *e.g.* processor clocks, FCPMs' clock, or Front-End-Electronic's clocks. DataFlow ensures that all these clocks are synchronized, so that its clients are able to time correlate information across components. The resolutions of these clocks vary, but any particular clock can never have a resolution better than one *sysclk*¹. However, the client may consider time as though it were derived from a single logical clock which will be called the *timebase*. The timebase is a 64-bit quantity which has a well defined relationship to wall-clock time, with one count defined to be one *sysclk*.

Every time an *LIAccept* (or other command) is generated by the FCTS, the timebase is sampled (to a resolution of a *sysclk*) and its value associated with the command. This value is called a *timestamp*. In the case of an *LIAccept*, the timestamp is used to check the synchronization of the data from the Front-End-Elements. In addition, DataFlow uses the timestamp to gather data segments associated with a particular *LIAccept* into data fragments, and then gather those fragments into events. This process (event building) is described in further detail in Section 3.5 on page 21.

The *sysclk* signal used to clock the IR-2 Platform is derived from the PEP-II timing system by dividing down its RF signal (476 MHz) by eight. PEP-II's RF signal corresponds to *buckets* in the accelerator, every other one of which is filled with a *bunch*. At any point in time, there are 3492 buckets in the accelerator, and thus up to 1746 bunches. 5% of the bunches are missing due to the ion clearing gap, leaving 1658 bunches in the ring at any one time. PEP-II provides a *fiducial* signal whose period corresponds to one complete revolution of a bunch, or 3492 RF periods (7.3 μ sec). An illustration of these relationships is shown in Figure 16 on page 54.

DataFlow provides its clients a consistent and regular interface to access and manipulate platform time, for example, to sample the current value of the timebase. This interface is through the `dfTime` class.

```
dfTime time; // Get current DataFlow time
```

The reference material for this class is contained in [2] starting on page 111.

1. nominally 16.7 nsec

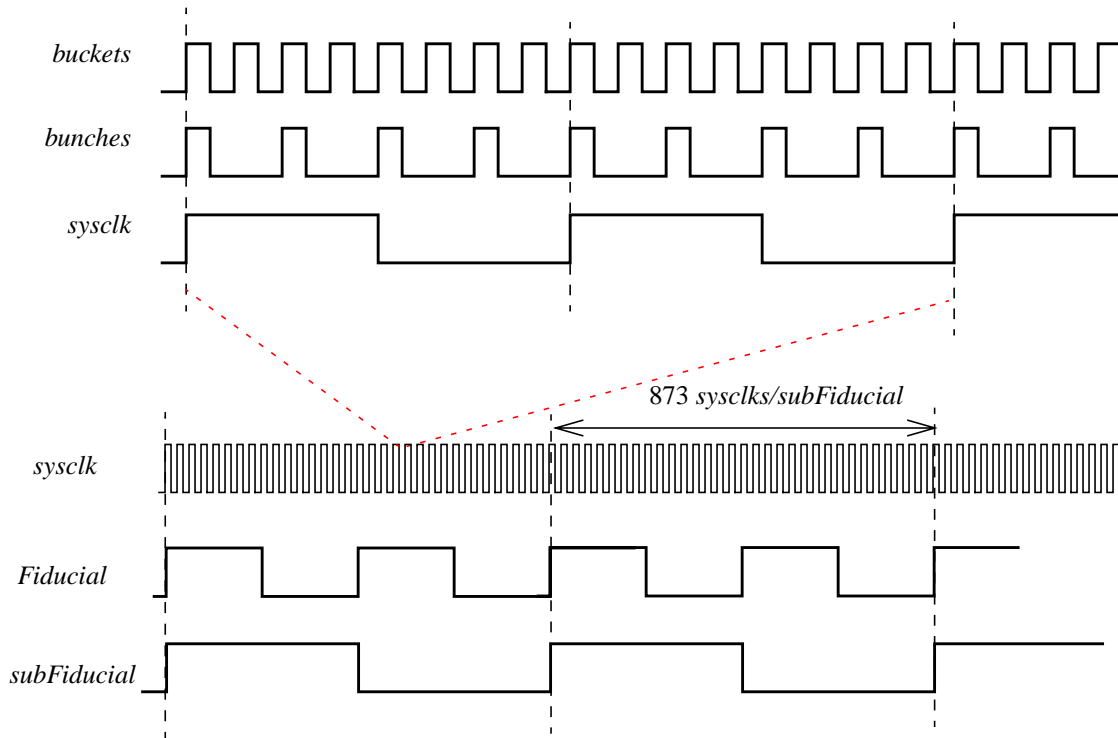


FIGURE 16. PEP-II and Platform timing structure

13.1 DataFlow Time vs. PEP-II Time

So that detector performance and event timing near the ion clearing gap can be studied, the timebase is related to the bunch structure of PEP-II. This is accomplished by synchronizing *sysclk* to PEP-II's fiducial signal with a precision of a single bucket. Assuming sufficient precision in both the latency of the trigger system and the times measured in the FEEs, DataFlow clients could conceivably correlate *LIAccepts* with the bunch structure of PEP-II to very high precision. However, given that 3492 is not evenly divisible by the number of buckets/*sysclk* (8), a fiducial transition only corresponds to a DataFlow *sysclk* transition every other period. Thus, the incoming fiducial signal is divided by 2 in the Fast Control Timing Module (FCTM) [3][12][13]. The result is referred to as the *subFiducial* signal. The FCTM aligns *sysclk* with the rising edge of the *subFiducial*. There are 873 *sysclks* in one *subFiducial*. When a new timebase value is loaded into a Fast Control Partition Master (FCPM)[3][12][13], the FCPM waits until the next *subFiducial* to begin incrementing its clock. DataFlow ensures that the initial value of the timebase is an even multiple of 873, such that the timebase (modulus 873) uniquely yields the number of *sysclks* since the last *subFiducial*.

13.2 Reinitialization of the Timebase

Within a Platform, many events exist simultaneously. Proper operation of DataFlow requires it to distinguish these events. Events are distinguished through their timestamp. This requirement is met by having the timebase monotonically increase over the lifetime¹ of a Platform.

Within BaBar, events are stored in a persistent database. Proper operation of the database requires it to distinguish its events. Since events are distinguished through their timestamp, the IR-2 DAQ System has a stronger requirement which is that the timebase must be monotonically increasing over the lifetime of the experiment. Therefore, this System must have a mechanism to persist time across reinitializations of the IR-2 Platform's timebase <OnlineRef >.

To ensure this monotonicity, DataFlow executes a particular algorithm when the platform's timebase is reinitialized. Figure 17 on page 57 illustrates this algorithm. When DataFlow receives a request to reinitialize the timebase of the platform, it immediately shuts the platform to external requests. Statistics continue to accumulate in both the Fast Control Gate Module (FCGM) [3][13] and the FCPM. Once the system has been shut, DataFlow reinitializes and resynchronizes the clocks in the partition serviced by each of the FCPMs, and when finished, reopens the system.

When DataFlow reinitializes the timebase, it checks each FCPM to see if it has been allocated. Allocated FCPMs are servicing the set of crates which constitute a partition. If the FCPM has been allocated, a marker command is sent to the partition to flush any pending data. The marker command's timestamp is used to bound the latest possible time in the Platform. If the FCPM is not allocated, a marker command is not sent and the flush time is set to the current time². After all partitions have been flushed, the largest flush time is saved.

DataFlow now requests the initial value for the timebase and rounds it up. The new time value is calculated using an epoch defined relative to midnight, Jan. 1, 1997, GMT. (On most Unix systems this time is obtained simply by taking the system time and subtracting 27 years of seconds (corrected for leapyears...)) A round up value is measured to account for the finite time it will take to load all FCPMs with the new time value, and to ensure that the new initial time value correspond to a *subFiducial* transition.

```
// code example using dfTime to implement rounding...
```

13.2.1 Monotonicity Enforcement

In order to ensure that the timebase is monotonically increasing, the algorithm compares the requested timebase value to the saved flush time. If the requested value is greater than the flush time, the requested value is accepted as an initial value for the timebase. If, how-

1. Here lifetime denotes time between reinitializations of the timebase.

2. Since there is no partition, there is no data to flush.

ever, the new value is less than or equal to the flush time, the algorithm requests another value. To allow the algorithm to converge successfully, the value provided should represent the reinitialization requestors best estimate of the wall-clock time. For example, if a timebase value one hour ahead of wall-clock had been successfully loaded into an FCPM, and at some later time an attempt is made to reinitialize the timebase to a value corresponding to the current wall-clock time, the requestor could be forced to wait one hour before the algorithm would converge successfully.

13.2.2 Timebase is a platform quantity, not a partition quantity

Within DataFlow, time is a quantity associated with the platform. *i.e.* it transcends partition boundaries. This fact is not relevant for standalone systems, which by definition can consist of only a single partition. For multi-partition systems, however, it is necessary that DataFlow ensure that all FCPMs contain the same timebase value. This is accomplished by means of a common *reset* signal generated by the FCTM, which starts the time counters on all FCPMs synchronously. A multi-partition system is thus *required* to have an FCTM.

13.2.3 Uncertainty in the Timebase

The relative uncertainty ΔT in the absolute time as determined by DataFlow after two successive time initializations is (assuming a perfect external time source) determined by the relation

$$\Delta T = \Delta t_1 + \Delta t_2 + \Delta t_3 + \Delta t_4$$

where Δt_1 is the uncertainty in the initialization time relative to the next *subFiducial* transition (on average half the *subFiducial* period), Δt_2 is the uncertainty in the time for the new value to propagate from the external source to DataFlow (hopefully small relative to the *subFiducial* period), Δt_3 is the uncertainty in the propagation delay before the initial time value is loaded into the Partition Master (small relative to the *subFiducial* period), and Δt_4 is the uncertainty in the rounding calculation (on average half the *subFiducial* period) (more here...14.7 μ sec is the *subFiducial* period).

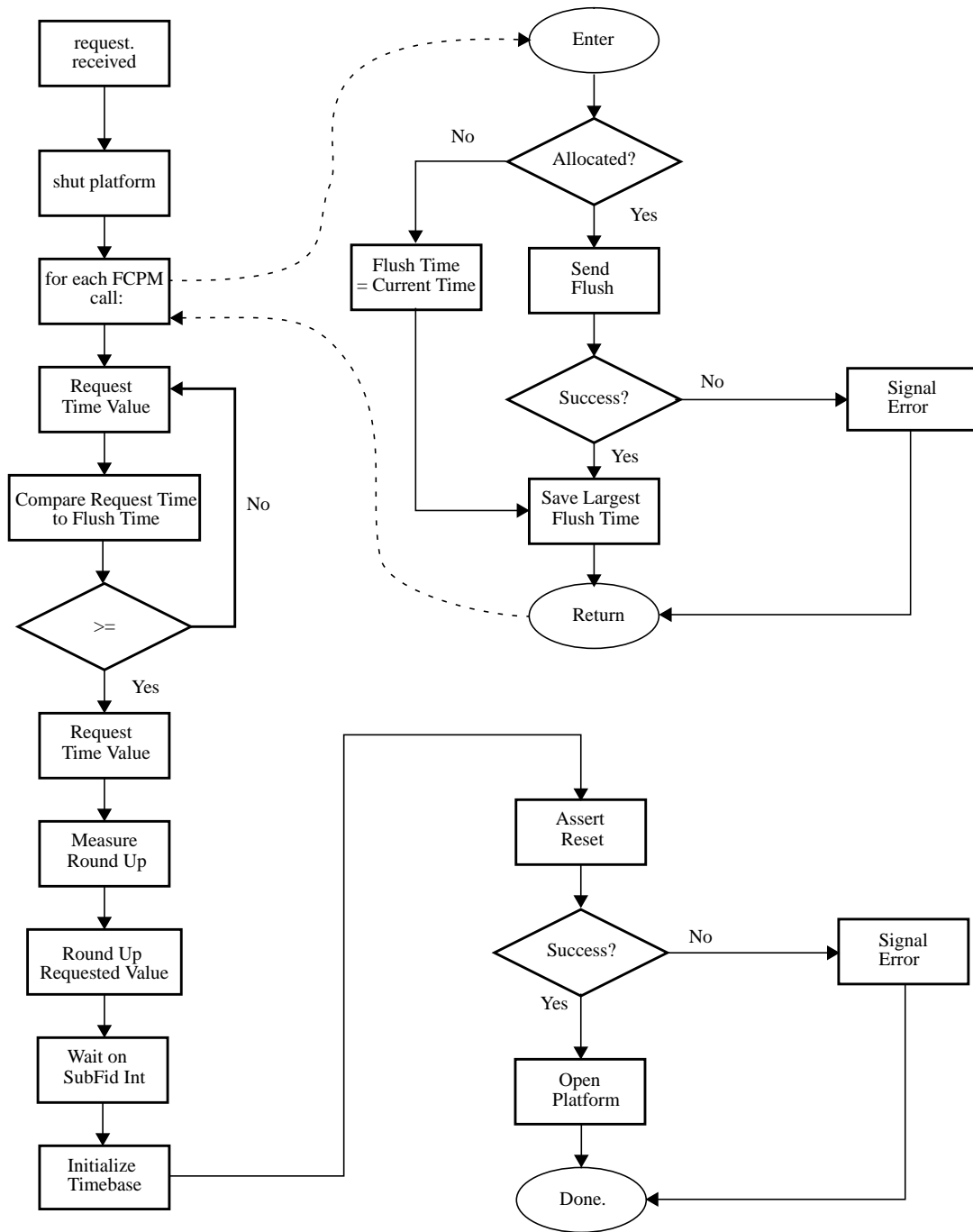


FIGURE 17. Flow chart describing timebase reinitialization

14.0 Occurrences

this section remains to be written...

15.0 Trigger monitoring and deadline

It is both natural and expected in the normal course of a platform's operation that whole events may be discarded due to DataFlow's inability to garner the resources necessary to either describe an event or to transport its associated data. This loss is called *deadline*, and while it cannot be made identically zero, DataFlow, of course, strives to minimize it. DataFlow monitors and logs both deadline volume and rate.

Deadline is the natural result of flow-control applied to event generation. DataFlow's model for flow-control is back-pressure coupled with trigger regulation. Typically, the propagation of an event downstream requires the allocation of one or more resources at each stage. If the stage cannot garner its necessary resources then it voluntarily enters a *resource wait* state, where it simply pends waiting for the resource to become available. When the resource does become available¹ the stage is removed from its resource wait state, allowing it to re-arbitrate for the resource. In this way, pressure is naturally asserted back through the hierarchy until it reaches the *Source* stage. This stage cannot enter resource wait because no-one exists to remove it from this state. Instead, it looks at least a single event ahead and conditionally asserts *available/full* to the Partition Master which, on the basis of this assertion, either enables or inhibits the transmission of *LIAccepts*.

this section is not yet complete...

16.0 Event damage

The acceptance of an *LIAccept* by DataFlow implies the expectation that an event can be delivered intact to the Event level, however, DataFlow cannot always honor this expectation. As event data contributions propagate downstream, they may encounter errors which prevent their further transport. These errors are acausal with the generation of the *LIAccept*, originating in the intrinsic unreliability of DataFlow and its platform. This implies that any hypothetical event may contain all, part, or none of its data. However, because DataFlow assumes an event has an existence independent of its associated data, it is able to transport all events, however incomplete. Events which do not contain all their expected data are deemed *damaged*. Damaged events are tagged with both the reason and extent of the damage. Note, that *damage* does not imply a corruption of either the consistency of the event's data structure or contents.

this section is not yet complete...

1. By another entity "freeing" the resource.

List of References

DataFlow Documents

- [1] Hamilton, R. T., M. E. Huffer, and J. L. White, *DAQ Programmer's Guide*.
- [2] Hamilton, R. T., M. E. Huffer, and J. L. White, *DataFlow Reference Manual*.
- [3] Hamilton, R. T., M. E. Huffer, and J. L. White, *The DataFlow Platform User's Guide*.

ROM Documents

- [4] Haller, G., *DAQ Readout Module - Overview*
- [5] Huffer, M. E., *Conceptual Model of the ROM from the perspective of DataFlow*
- [6] Dowdell, J., *Architecture for the BaBar DAQ Read-Out Module*
- [7] Oxoby, G., *PCI Mezzanine Card*
- [8] Dowdell, J., *BaBar DAQ Controller Card Description*
- [9] Haller, G., *Architecture for the BaBar Triggered Personality Card*
- [10] Intel, *i960RP Microprocessor User's Manual*, Part #272736-001, 1996.

FCTS Documents

- [11] Lankford, A., *Functional Requirements of the BaBar Fast Control and Timing System*.
- [12] Haller, G., *Conceptual Design of the BaBar Fast Control and Timing System*.
- [13] Sapozhnikov, L., *Architecture for the BaBar Fast Control and Timing System*.
- [14] Haller, G. and G. Oxoby, *Electronics Control and Dataflow between the Read-Out Module and the Front-End Electronics Systems*, BaBar Note 281 Version 1.1.

Wind River Systems (WRS) Documents

- [15] WRS, *User's Guide: Tornado 1.0 (UNIX Version)*.
- [16] WRS, *Tornado Release Notes (UNIX Version), 1.0, Rev. 2*.
- [17] WRS, *Wind River Products Installation Guide, Torando 1.0 (UNIX Version), Rev. 3*.
- [18] WRS, *Tornado for PowerPC Architecture Supplement, 1.0*.

- [19] WRS, *VxWorks Programmers Guide*, 5.3.
- [20] WRS, *VxWorks Reference Manual*, 5.3.
- [21] WRS, *Tornado API Guide*, 1.0 Beta.
- [22] WRS, *GNU ToolKit User's Guide*, 2.6.
- [23] WRS, *Debugging with GDB*, 4.12.

Other Third Party Documents

- [24] Radstone, *Radstone PPC603/603e/604 User Guide*.
- [25] Radstone, *Radstone PPC603/4 Rev. 004 Installation Guide*.
- [26] Tundra, Universe Chapter, *VMEbus Interface Components Manual*, spring 1996.
- [27] VITA, *The VMEbus Specification*, ANSI/IEEE STD 1014-1987, ICE 821 and 297.
- [28] EPICS Documentation.

DataFlow CDR Supporting Documents

- [29] Day, C. T., R. T. Hamilton, M. E. Huffer, and J. L. White, *The Data Flow System*.
- [30] Hamilton, R. T., *Fast Control Software Manual*.
- [31] Huffer, M. E., *The Segment Builder*.
- [32] White, J. L., *The Fragment Builder*.
- [33] Day, C. T., *The Event Builder*.

Other BaBar Documents

- [34] BaBar Collaboration, *BaBar Technical Design Report*.