DataFlow Proxies and the Finite State Machine

Electronics Workshop, Imperial College, London

April 23-26, 1997

The DataFlow Group

R. T. Hamilton, University of Iowa

M. E. Huffer, Stanford Linear Accelerator Center

J. L. White, Stanford Linear Accelerator Center

Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Outline

Platform and Partition Proxies

- The Platform and Hierarchy
- The **dfManagement** and **dfClient** Packages
- Partitioning the Platform

The Finite State Machine

- Role of the FSM
- Controlling and Participating in the FSM
- The *Active* Substates
- The **dfManager** and **dfFsm** Classes

Taking Calibrations

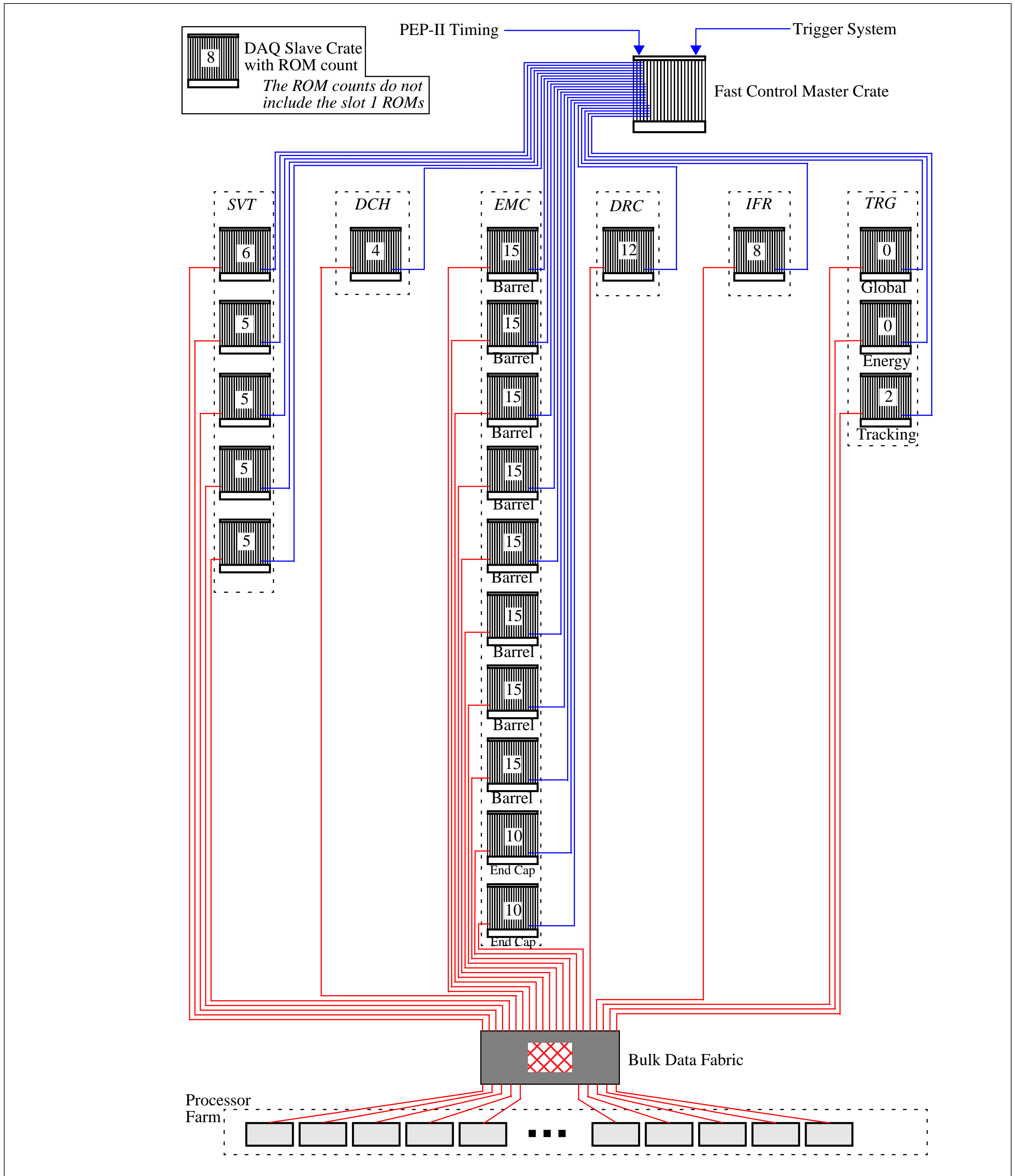
- The Partition Master
- The **dfCalibCycle** and **dfCalibCommand** Classes
- Coding a Calibration

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

The DataFlow Platform

- Has one or more VME crates - with custom J3 backplane and capable of housing 9U modules.
- Requires a *Timing System* (either internal or external). The platform and the electronics it services are synchronized and phased appropriately.
- Requires an external *Trigger System*. The platform accepts 32 trigger input lines, each of which may cause an L1Accept to be generated and propagated throughout the platform. Each trigger line may be driven either externally through hardware, or internally through software.
- Contains at least one FCPM and 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 ROMs. In general, a ROM's function is to gather event data from its managed FEEs. In addition, one ROM in each crate is responsible for gather the event data from all the ROMs in that crate and posting it to the Bulk Data Fabric. This is referred to as the *Slot-1 ROM*. In a stand-alone system it is possible to have a single ROM both manage FEEs and serve as the Slot-1 ROM.
- Has one or more *Generic Unix Boxes* (GUBs) whose function is to serve as both source and sink for the Bulk Data Fabric.
- Has a *Bulk Data Fabric* whose function is to transport large volume data (principally event oriented) both into and out of the platform.
- Has a *Control Fabric* whose function is to provide connectivity between the Slot-1 ROMs and any GUBs.

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

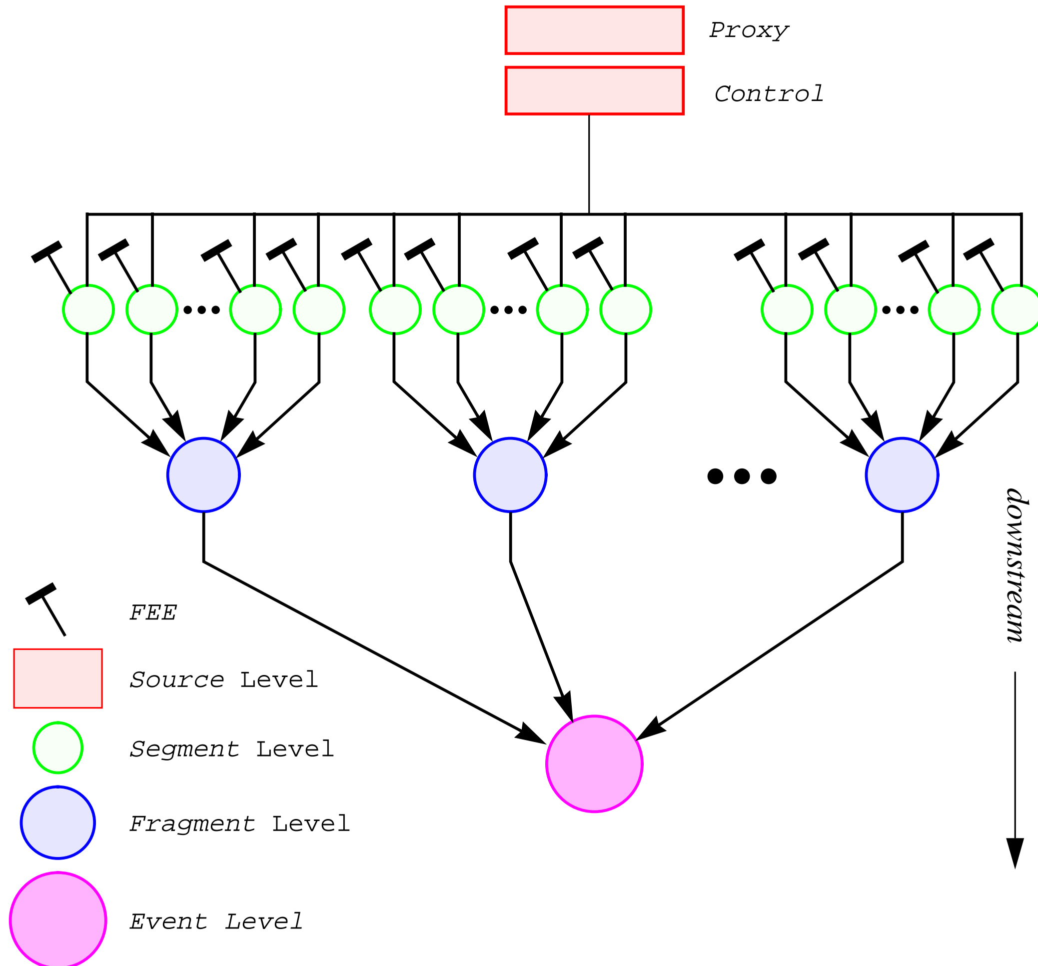
The Platform Hierarchy

All platforms, independent of their composition, have their internal components connected in a hierarchical fashion. DataFlow reflects the platform hierarchy by defining four levels:

- The *Source* level. This is the level at which both control and data information come into existence. This information is generated through commands issued by the FCTS's Partition Master. These commands include *L1Accepts* which initiate detector read-out, *CalStrobes* to inject calibration pulses, and *Syncs* used to resynchronize the platform's clocks.
- The *Segment* level. This level corresponds to both Triggered and Untriggered ROMs. On receipt of an L1Accept, this level is responsible for gathering the event contributions from the FEEs, feature extracting the contribution into a segment, and posting the segment to the Fragment level. In addition, this level is responsible for monitoring, configuring, and controlling the FEEs.
- The *Fragment* level. This level 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 level 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)

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

The Platform Hierarchy (2)



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

The Management and Client Packages

Applications interacting with DataFlow may take one of two roles - *control* or *participation*.

Control of DataFlow is accomplished by means of the *Management Package*:

- dfPlatform - serves as proxy for the platform.
- dfPartition - serves as read-only proxy for a partition.
- dfManager - configures and controls a partition. Also drives the transitions of the Finite State Machine.
- dfCalibCycle
- dfCalibCommand

} Used by dfManager

The dfPlatform, dfPartition, and dfManager are all *mappable*. (See Joe's talk.)

Participation in DataFlow is accomplished by means of the *Client Package*:

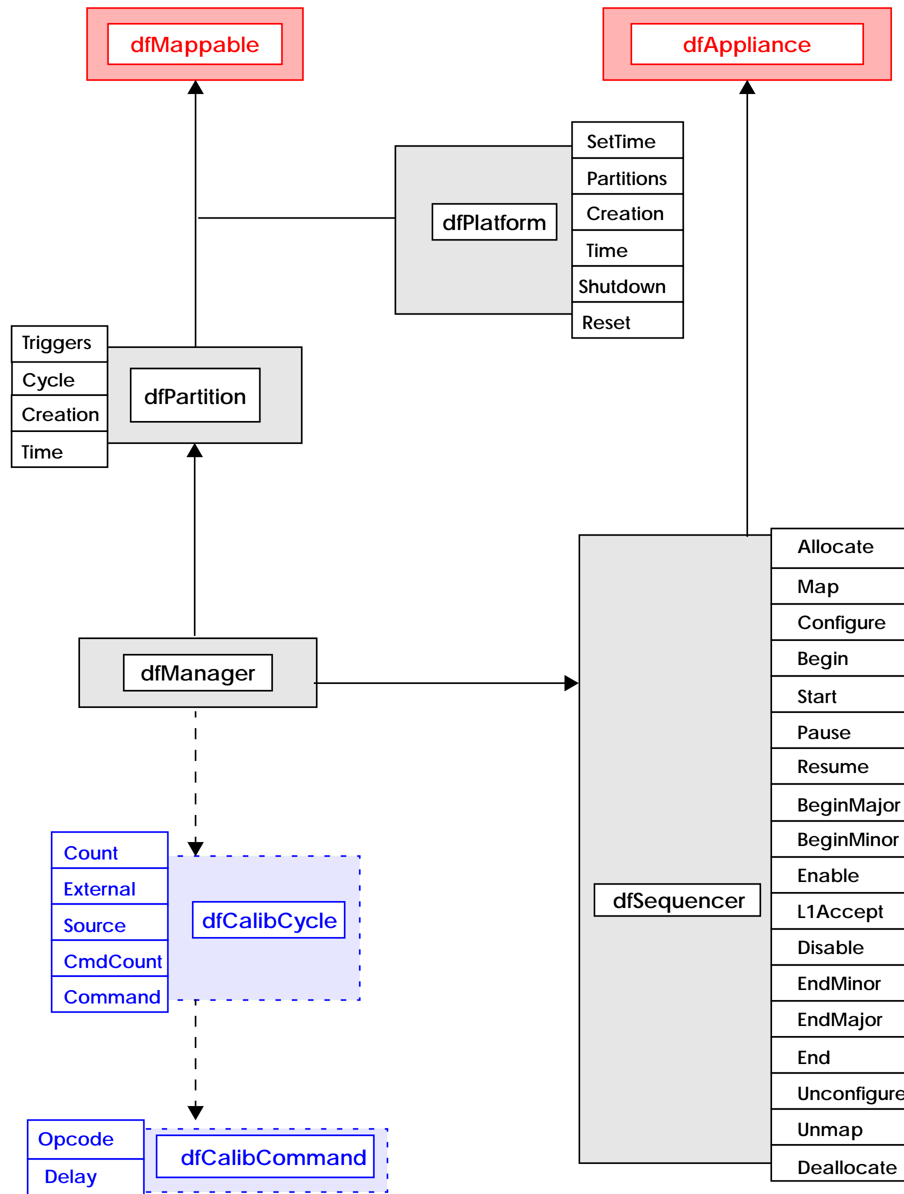
- dfFsm - enforces the Finite State Machine by accepting or rejecting transitions. Also permits an application to associate an *action* with each transition.
- dfAction
- dfState
- dfContext

} Used by dfFsm

The dfManager and dfFsm are *appliances*. (See Mike's talk.)

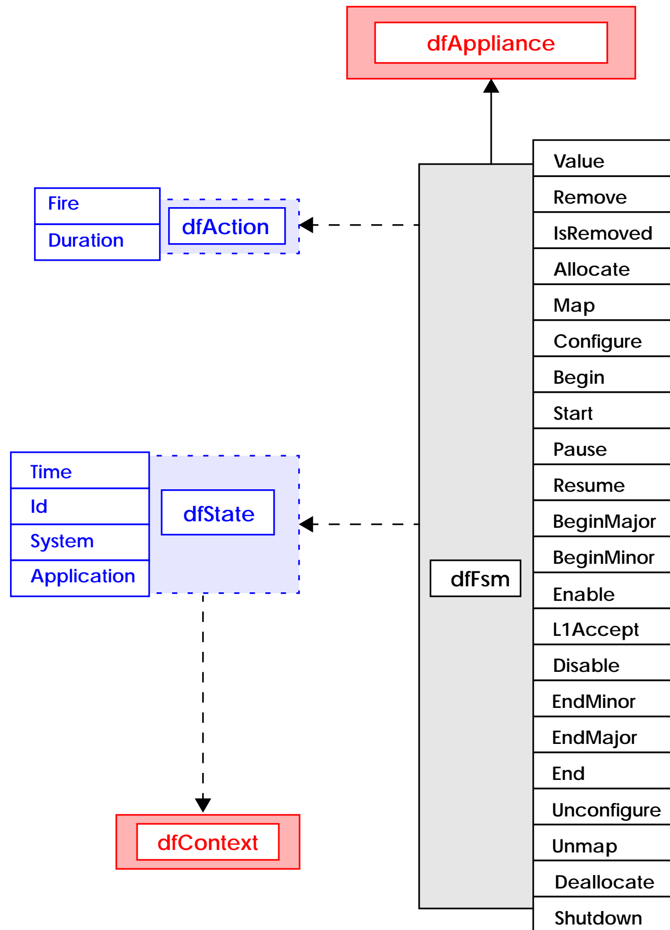
■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

dfManagement



Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

dfClient



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Creating a Partition

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 platform.

A DataFlow partitioned platform has the following characteristics:

- Control of each partition is mediated through an FCTS module called the *Partition Master* (FCPM).
- 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 platform hierarchy 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 FCPMs, 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.

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Creating a Partition (2)

A partition is created by instantiating an object of the dfManager class. In the following example, first a dfPlatform object is created to serve as a proxy for the platform, then this object is mapped:

```
dfPlatform myPlatform;
dfMap      map(myPlatform);
```

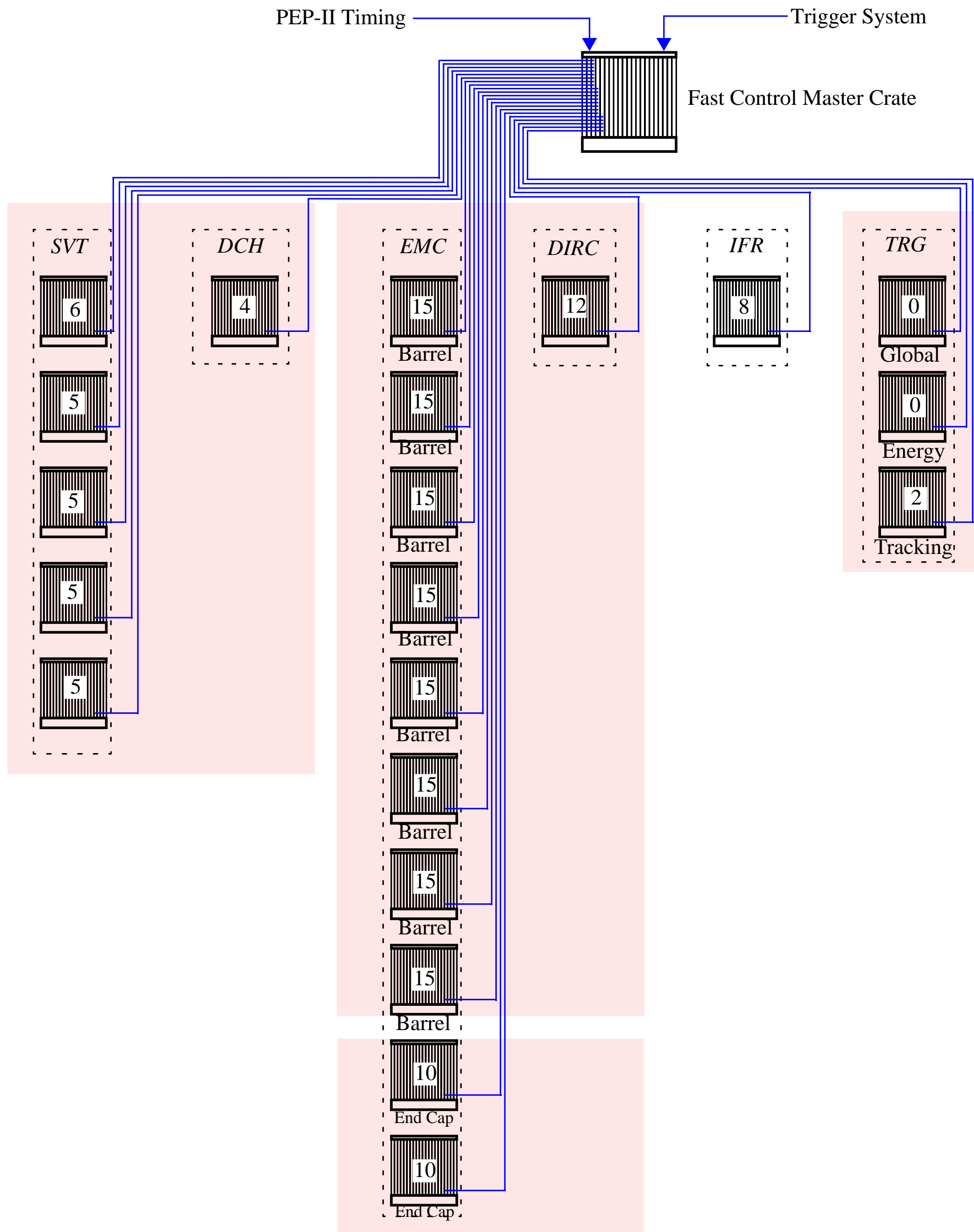
Next, dfManager objects are created for each partition:

```
dfManager tracking(map.Crates(DF_SVT | DF_DCH));
dfManager barrel(map.Crates(DF_EMC_BRL | DF_DRC));
dfManager endcap(map.Crates(DF_EMC_ECP));

const unsigned TRIGGER =
         (DF_TRG_GBL | DF_TRG_ENR | DF_TRG_TRK);

dfManager trigger(map.Crates(TRIGGER));
```


Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

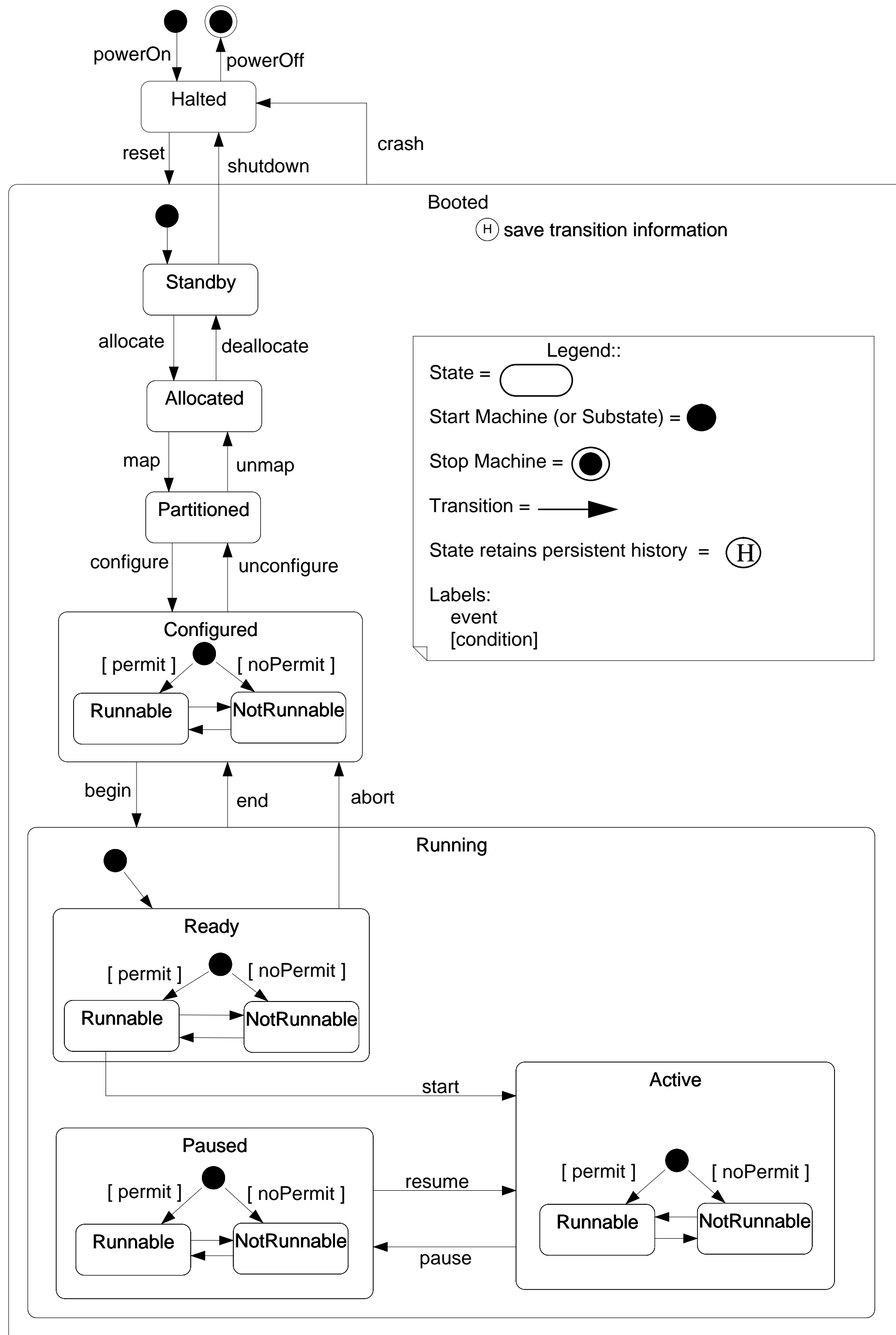
Role of the Finite State Machine

The control of applications participating in DataFlow is modeled as a Finite State Machine (FSM). Each step that participants must go through in order to take data, from power up, to *L1Accept*, and back to power down, corresponds to a transition in the FSM.

- *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 *L1Accept* 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. For the purposes of this discussion, however, it can be treated as a single state.
- *Enabled* - In the *Enabled* state, the partition is generating *L1Accepts* which drive the FSM's *L1Accept* transition.

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

The Finite State Machine



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Controlling and Participating in the FSM

An application drives the FSM by calling the member functions of the dfManager class:

```
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
   ...
```

An application participates in the FSM be registering actions to be associated with transitions. This is done by calling the member functions of the dfFsm class:

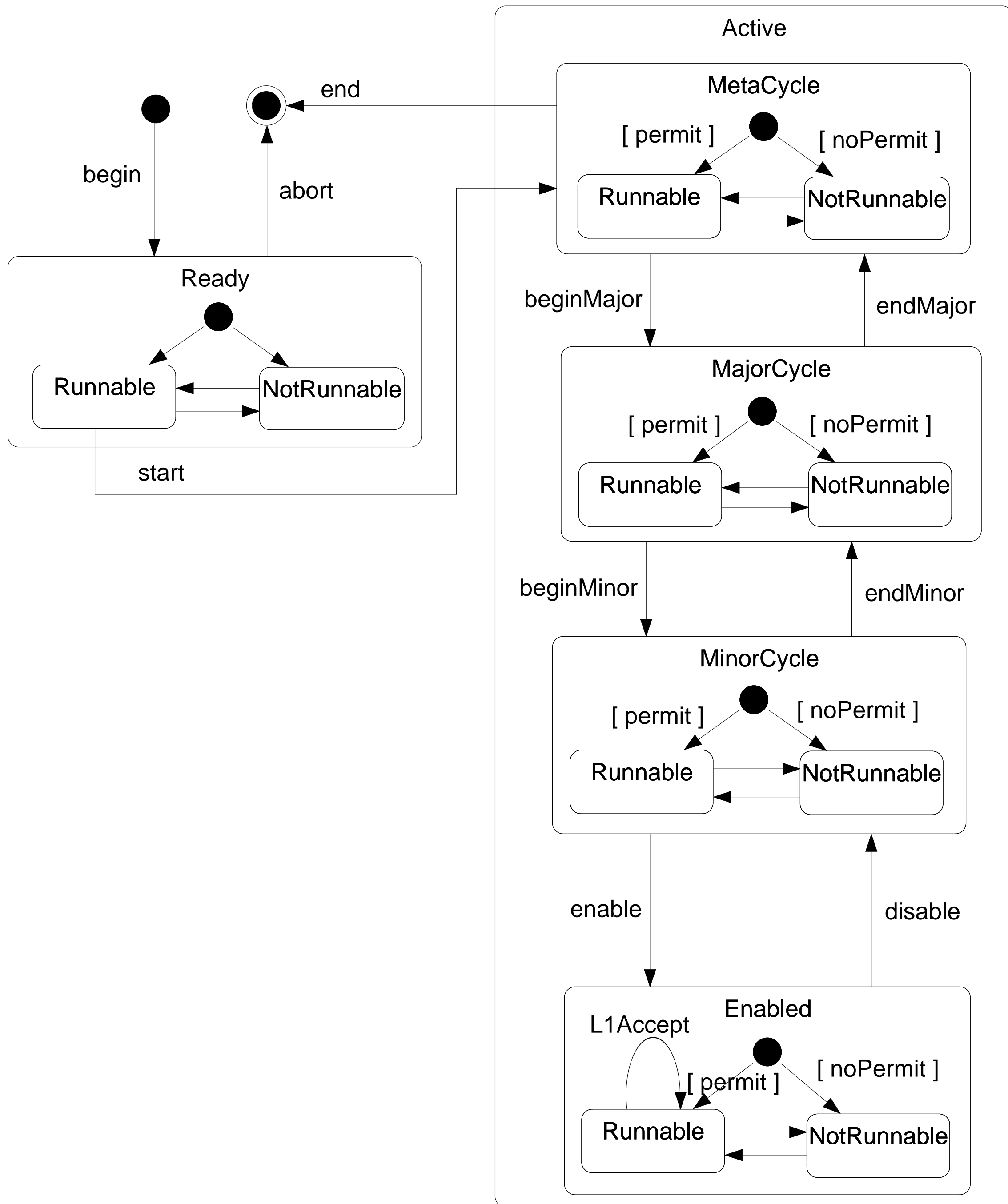
```
#define ONE_MICROSECOND 60
const int DURATION = (100 * ONE_MICROSECOND);

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

previous = myMachine.Configure(action);
```


Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Active state (exploded)



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

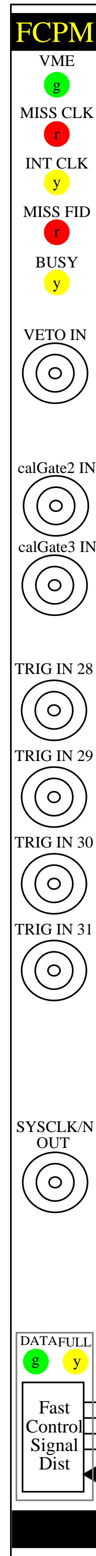
Role of the *Active* Substates

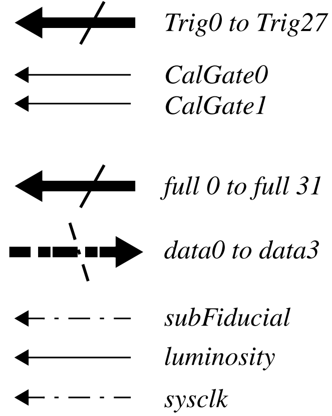
Both calibration and data taking occur in the context of the *Active* state. Since data taking is in fact a special case of calibration, the rest of this talk focuses on calibration.

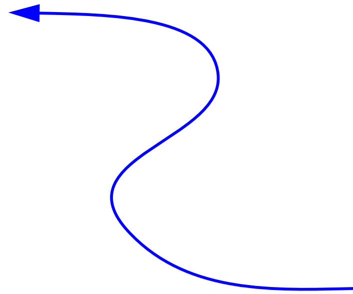
The FSM requires that calibrations be organized into a set of nested cycles. These cycles are described by the substates of the *Active* state, as shown below:

- *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 *L1Accepts* are generated.

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199







Inputs for external gate signals.

■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

dfCalibCycle and dfCalibCommand

A calibration MinorCycle is executed by means of a hardware *calibration sequencer* in the FCPM. DataFlow models the functionality of this hardware with the dfCalibCycle and dfCalibCommand classes.

dfCalibCycle 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. It is an abstract base class with the following member functions:

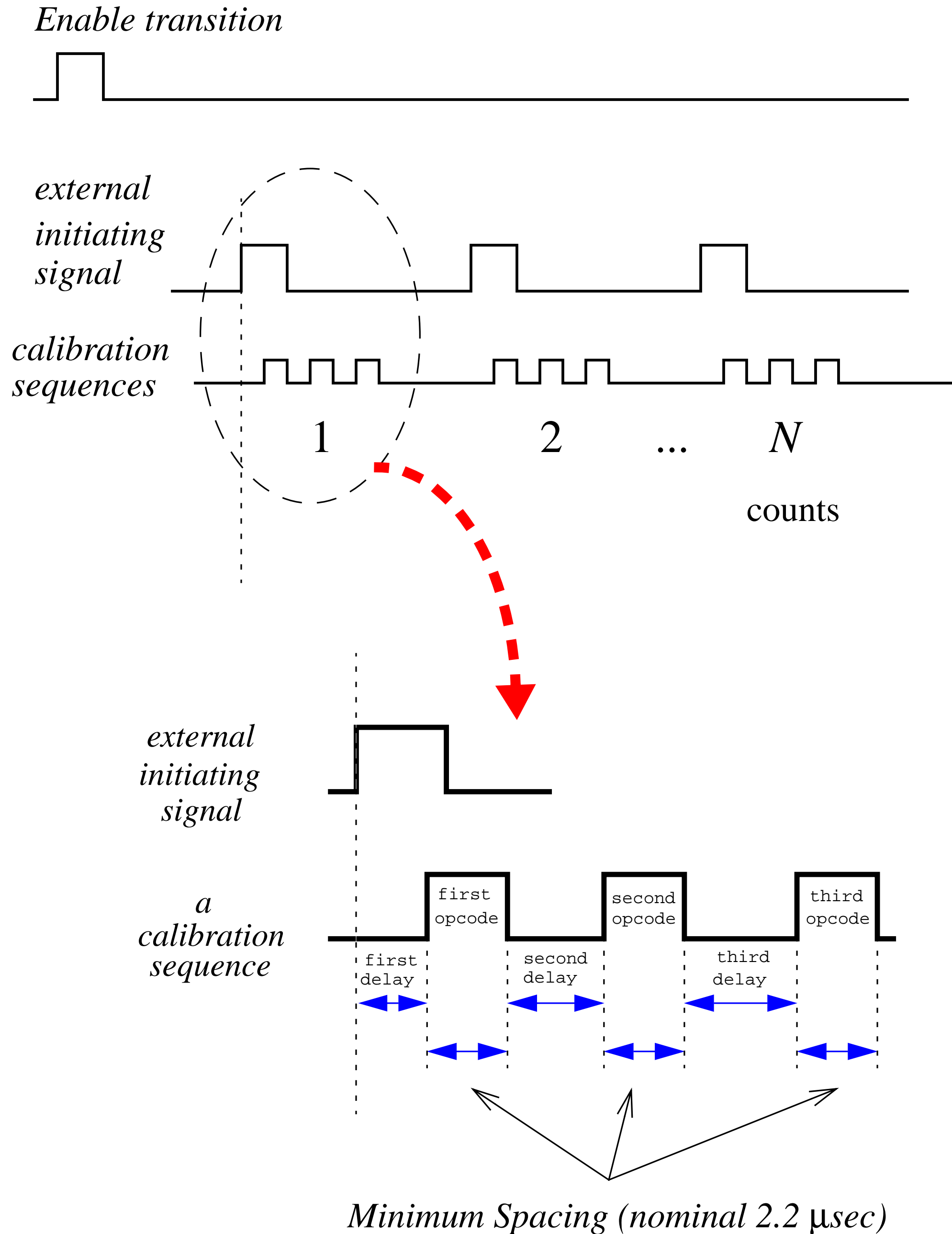
- **Count() -** returns the number of iterations to repeat the sequence.
- **External() -** returns the *gating mode* for the cycle.
- **Source() -** returns the source for the external gating signal.
- **CmdCount() -** returns the number of commands in the sequence.
- **Command() -** returns a specified command in the sequence.

dfCalibCommand encapsulates a single *command*, consisting of an opcode and a delay. It is an abstract base class with the following member functions:

- **Opcode() -** returns the opcode for the command.
- **Delay() -** returns the time delay *before* the opcode is executed.

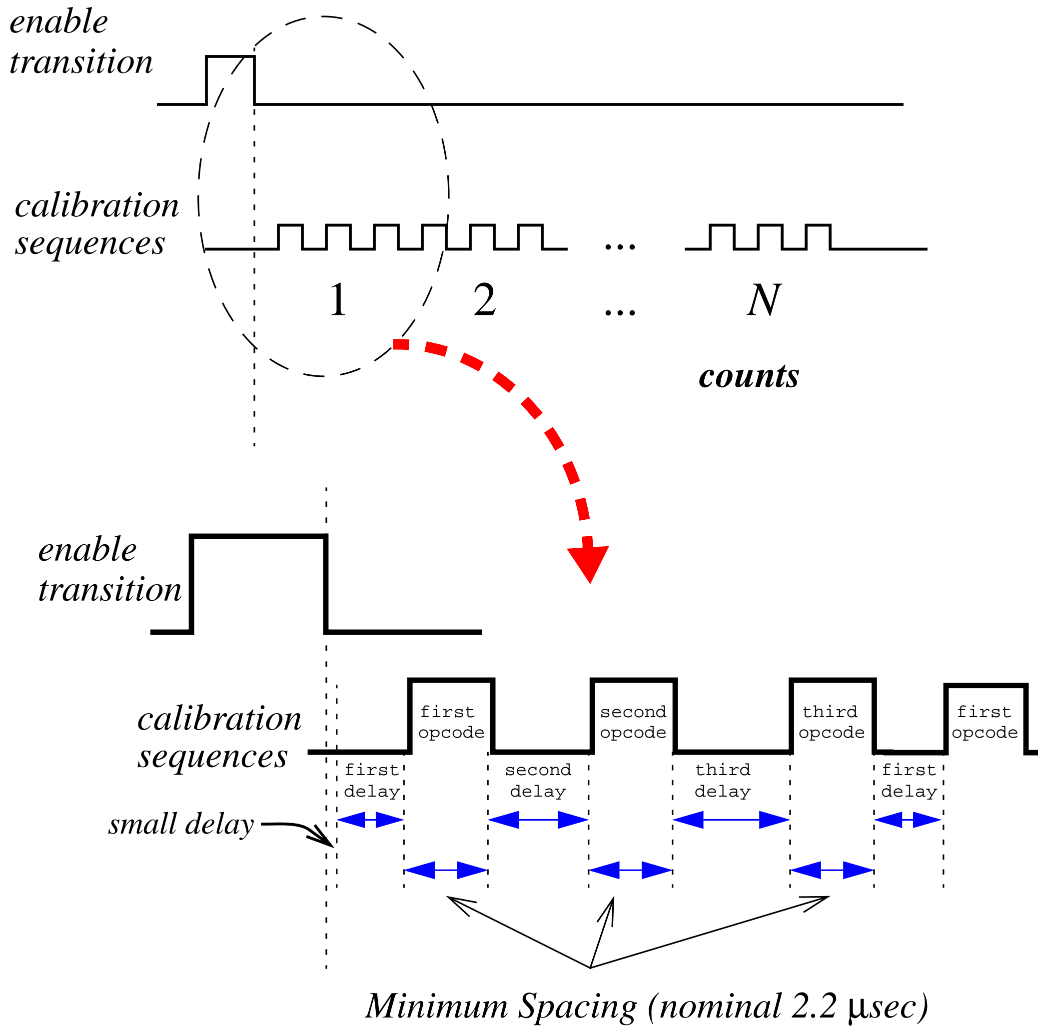
■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Timing in *External* Mode



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Timing in *Internal* Mode



■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Example Calibration Code

The following sample code establishes classes derived from dfCalibCommand which encapsulate the *CalStrobe* and *L1Accept* opcodes, both with a 1 μsec delay:

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

class myCalStrobe: public dfCalibCommand {
 public:
  dfOpCode_t Opcode()  {return CalStrobe;}
  unsigned int Delay() {return one_microsecond;}}

class myL1Accept: public dfCalibCommand {
 public:
  dfOpCode_t Opcode()  {return L1Accept;}
  unsigned int Delay() {return one_microsecond;}}
```


■
Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199

Example Calibration Code (2)

Here we assemble the previous two command classes into a cycle of 1000 iterations and gated from the FCPM front panel, by defining a class derived from the dfCalibCycle class:

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

class myCycle: public dfCalibCycle {
public:
    unsigned int Count()    {return one_thousand;}
    bool External()         {return true;}
    dfSource_t Source()     {return CalGate2;}
    unsigned int CmdCount() {return twoCommands;}
    dfCalibCommand& Command(dfCmd_t index);
private:
    myCalStrobe cmdFirst;
    myL1Accept cmdSecond;}

myCycle::Command(index){
    if (index == 0)      return &cmdFirst;
    else if (index == 1) return &cmdSecond;
    else                 return NULL;}
```

Finally, we load the cycle into the FCPM by calling the dfManager's BeginMinor member function:

```
do{
 partition.BeginMinor(cycle)->Wait();
 partition.Enable()->Wait();
 partition.EndMinor()->Wait();
} while(); // Major Cycle not done
```


Rowan Hamilton
rhamilt@slac.stanford.edu
(415) 926-4199