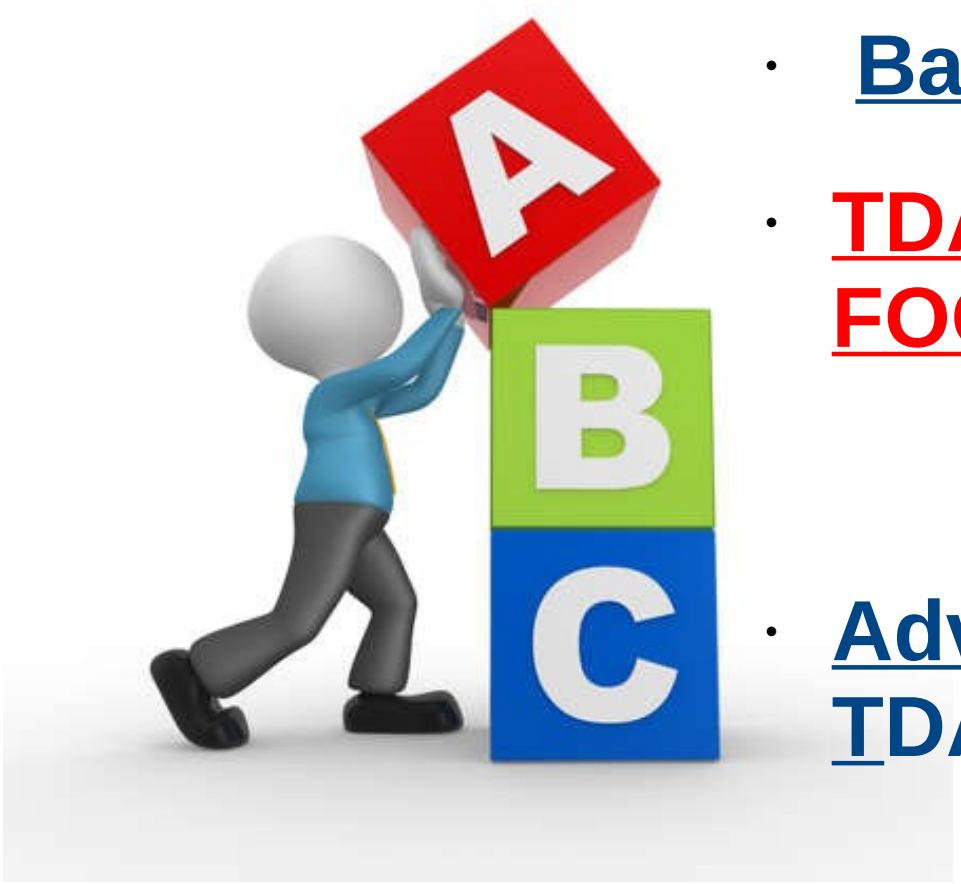
Introduction to Trigger and Data AcQuisition

2/3 – TDAQ infrastructure

Plan



- Basic DAQ concepts
- TDAQ Infrastructure with FOOT Examples
- Advance TDAQ & FOOT TDAQ specific

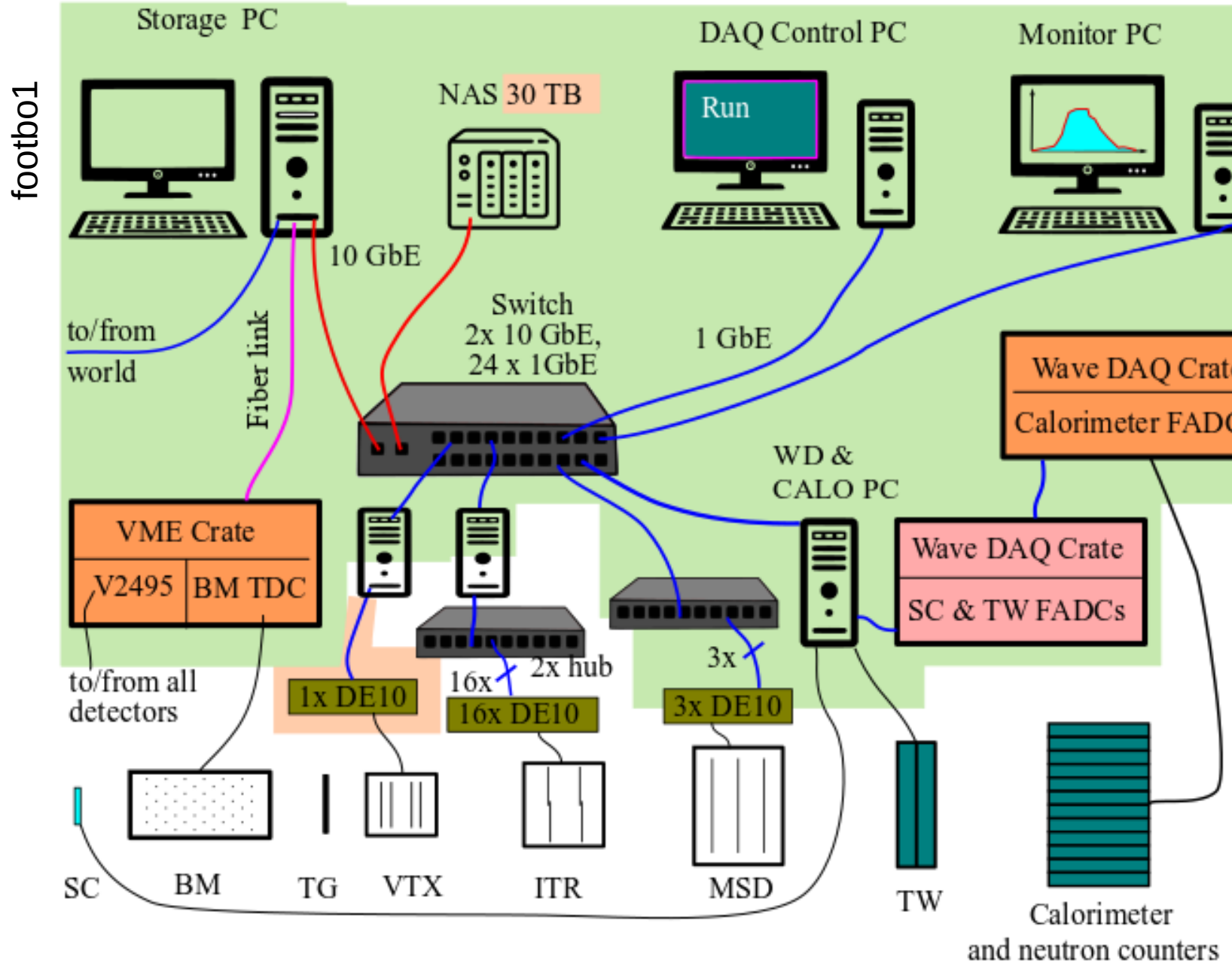
DAQ duties

- Gather data produced by detectors
  - **Readout**
- Form complete events
  - **Data Collection** and **Event Building**
- Possibly feed other trigger levels
  - **H**igh **L**evel **T**rigger
- Store event data
  - **Data Logging**

Data Flow

- Manage the operations
  - **Run Control**, **Configuration**, **Monitoring**

FOOT Architecture



TDAQ

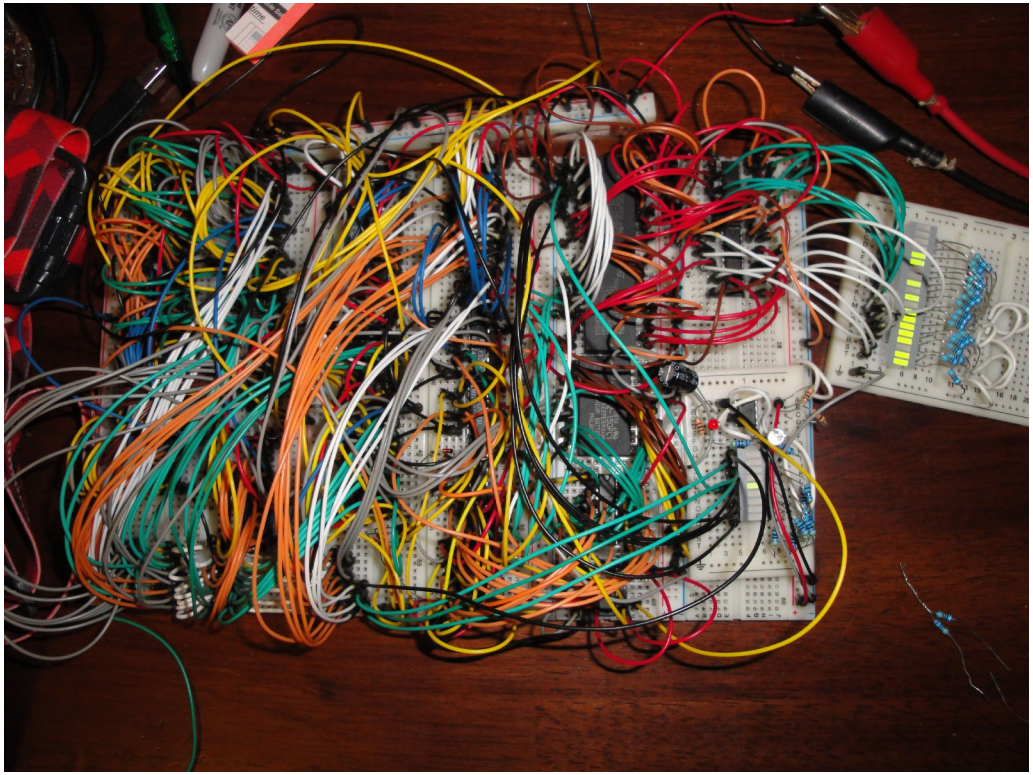
- Even for systems with more than just one detector the TDAQ has to be:
  - Hierarchical
  - Distributed
  - Multiplatform

- FOOT TDAQ: Based on ATLAS TDAQ
  - Complete framework
  - Deals with all aspects of the data taking
    - (and several more!)

What we will see today

- Software packages needed
- Machine configuration
- Inter-process communication (IPC)
- OKS – static system description
- Run-control FSM and
  controllable objects
- Readout modules
- DataChannels
- Triggers
- DataStreams

Spaghetti code



Software packages

- 64 bits linux machine, SLC-6 (next will be cent-OS)
- ATLAS tdaq-01-07-00
  - /afs/cern.ch/atlas/project/tdaq/cmake/projects/tdaq/tdaq-01-07-00
  - ATLAS tdaq-common
- LCG_87 libraries (Boost, ROOT, MYSQL, CMAKE, gcc)
- Atlas drivers → ask Joos

  (knowing a little bit of services in linux is usefull!!)

Note: gcc compiler: 6.2.0, compatible C++11, C++14

Non-trivial configurations 1/2

- Setting of */etc*/hosts
  - All used machines should be present in hosts:

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
#
192.168.1.1 footbo1   footbo1.tdaq.it
192.168.1.2 footbo2   footbo2.tdaq.it
192.168.1.6 pcvmelucid4  pcvmelucid4.tdaq.it
192.168.1.7 footbo3   footbo3.tdaq.it
192.168.1.3 de0nano1  de0nano1.tdaq.it
192.168.1.4 de10nano2  de10nano2.tdaq.it
```

- Setting of ulimits (TDAQ opens many files)
  - ulimit -n → 65536

```
$ cat /etc/security/limits.d/99-max-files.conf
# set the max number of open files - issue
*               soft    nofile          65536
*               hard    nofile          65536
```

Non-trivial configurations 2/2

Login with tdaq accounts without passwords, without question and print-outs

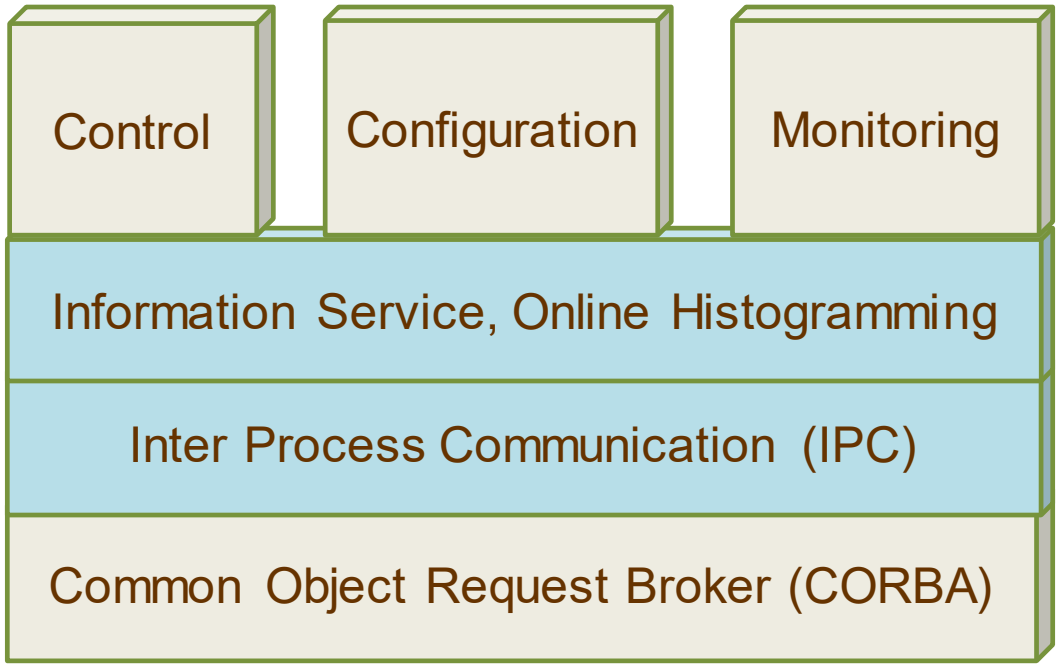
For each tdaq user and for each tdaq machine, do:
cd $HOME
ssh-keygen
(press always the enter key)
id_rsa and id_rsa.pub files will be produce in $HOME/.ssh

#all files id_rsa.pub have to be collected and appended to .ssh/authorized_keys
cp .ssh/id_rsa.pub .ssh/id_rsa_$(basename $HOSTNAME .bo.infn.it).pub
cat .ssh/id_rsa_$(basename $HOSTNAME .bo.infn.it).pub >> .ssh/authorized_keys
chmod 644 .ssh/authorized_keys

To be sure there are no questions please do:

```
ssh machine ,  ssh machine.domain
```

TDAQ Online Software Architecture



High level Services

Basic info sharing services

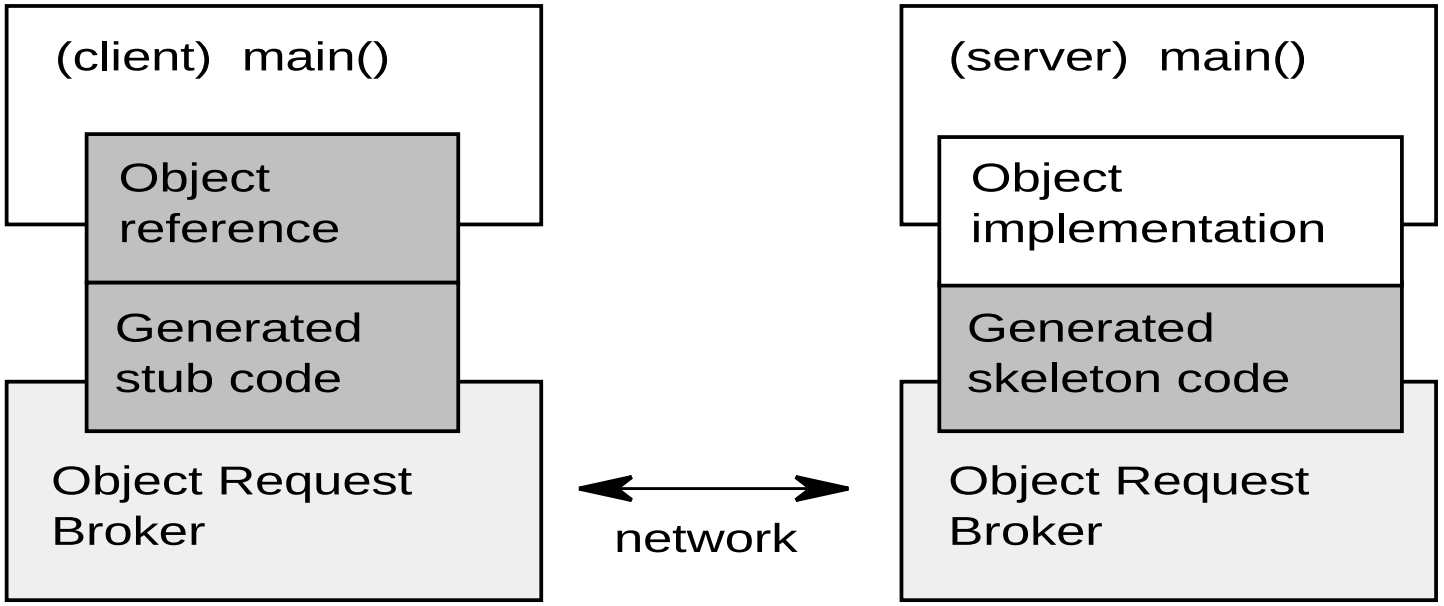
Communication abstraction API

Communication middleware:
OmniORB & JacORB

- TDAQ is highly distributed system running 30K processes on about 3000 computers (ATLAS numbers)
- Service based approach has been used for implementing the TDAQ management:
  - Each service has well-defined responsibility within TDAQ
  - High level services use basic ones
  - Some basic services are exposed to the end-users (IS, OH)

Common Object Request Broker Architecture (CORBA)

It provides an environment to describe and to ship OBJECTS among different PCs
Objects are described with an Interface Description Language (IDL)



Data objects can be exchanged and **procedure calls** can be performed remotely. Flexible but difficult to use.

In TDAQ: Inter Process Communication (IPC)

IPC settings

Environment variable: $TDAQ_IPC_INIT_REF

- Contains the top level file for synchronization
- `TDAQ_IPC_INIT_REF=file:/labdaq/com/ipc_root.ref`

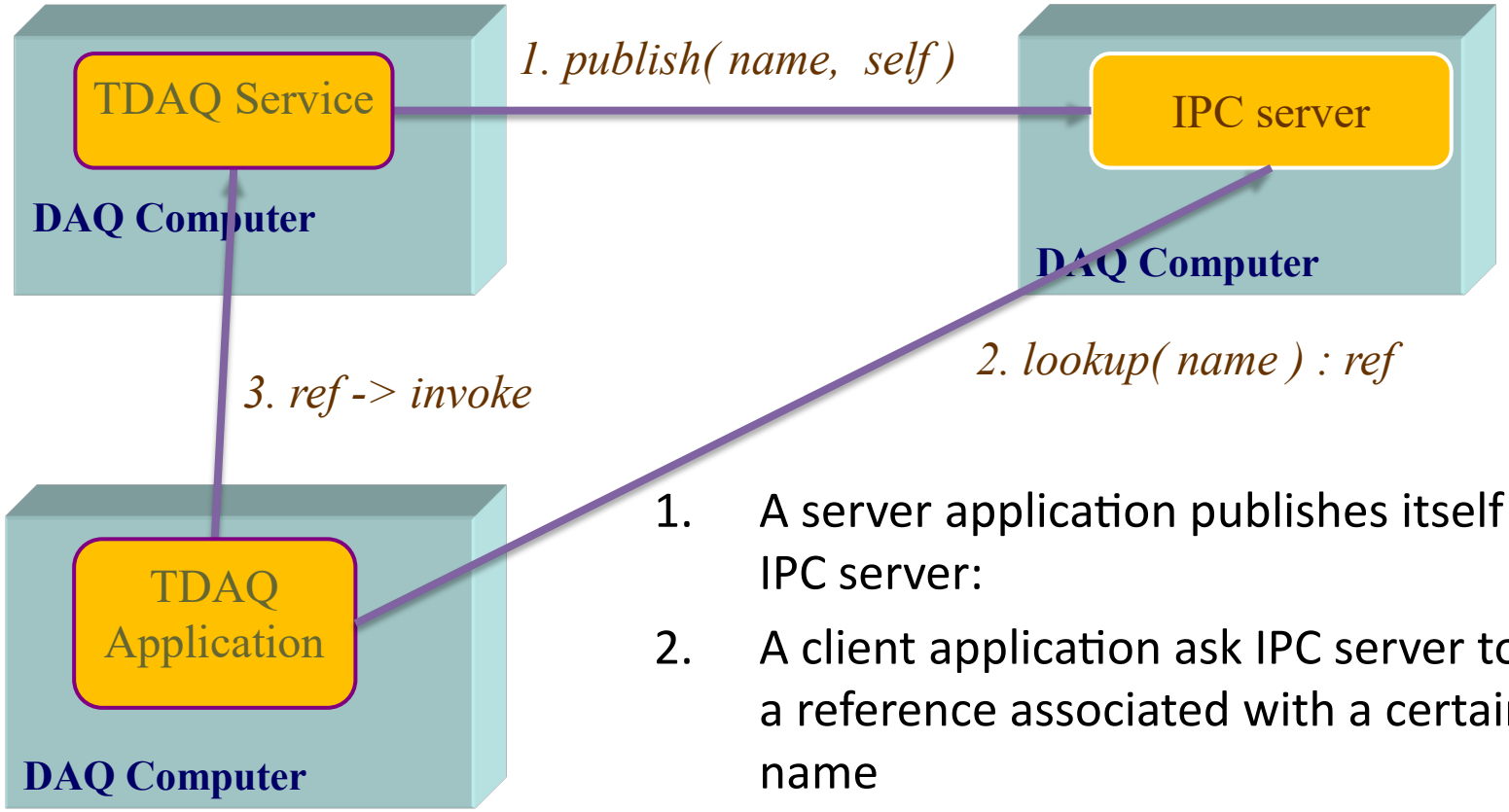
A reference partition “initial” is defined with the command (auto):

```
$ ipc_server -b -d /tmp/tdaq-07-01-00/initial
```

- To see what is inside:
- `$ ipc_ls`

initial
- ipc/servant:
- is/repository:
  - PMG
  - RunCtrl
  - RunParams
- pmgpriv/SERVER:
  - AGENT_footbo1
- rc/commander:
  - DefaultRootController
- rdb/cursor:
  - ISRepository

Inter Process Communication



1. A server application publishes itself to the IPC server:
2. A client application ask IPC server to get back a reference associated with a certain service name
3. The client application uses this reference to request an action on the service

Examples

An application wants to know the **current run number**:

- The Main TDAQ process publishes this info in IS/RunParams
- The app asks the IPC server “Information Service” for the data stored in the block “RunParams” and in particular asks for the value of “RunNumber”.

A *Readout Module* **V2495** wants to publish **monitoring data**:

- The app asks to the IPC server for the “IS/monitoring” server and sends the object with the data to publish on “IS/monitoring/V2495”. Data structure (class, schema) has to be known in advance by the IS and V2495.
- V2495 monitoring data can be then accessed by everyone.

An Application needs to receive **configuration information**:

- The main TDAQ process sends this information at specific FSM transaction (transition setup/configure)

OKS (Object Kernel Support)

- The OKS is a library to support a simple, active persistent in-memory object manager. It is suitable for applications which need to create persistent structured information with fast access but do not require full database functionality.
-
- As C++ it uses two type of files:

  SCHEMA : Classes, superClasses (derivation)

  DATA: objects, vectors, pointers, links

Examples of classes and data

```
<class name="Computer" description="Describes a computer.">
  <superclass name="ComputerBase"/>
  <superclass name="Platform"/>
  <superclass name="HW_Object"/>
  <attribute name="Memory" description="Memory in MB" type="u32" init-value="16384" is-not-null="yes"/>
  <attribute name="RLogin" description="Defines command used for computer remote login. "
             type="string" init-value="ssh" is-not-null="yes"/>
  <relationship name="Interfaces" description="Eth interfaces." class-type="Interface"
             low-cc="zero" high-cc="many" is-composite="yes" is-exclusive="no" is-dependent="yes"/>
 </class>
```

```
<obj class="Computer" id="footbo1.bo.infn.it">
 <attr name="HW_Tag" type="enum">"x86_64-slc6"</attr>
 <attr name="Memory" type="u32">256</attr>
 <attr name="CPU" type="u16">1000</attr>
 <attr name="NumberOfCores" type="u16">4</attr>
 <attr name="RLogin" type="string">"ssh"</attr>
 <rel name="Interfaces" num="1">
  "NetworkInterface" "footbo1.eth0"
 </rel>
</obj>

<obj class="NetworkInterface" id="footbo1.eth0">
 <attr name="IPAddress" type="string">"131.154.10.41"</attr>
 <attr name="InterfaceName" type="string">"eth0"</attr>
</obj>
```

SCHEMA files

- Are used to describe the logical structure of elements in the system
- Defines classes which can contain superclasses
- Define attribute properties of a class; can provide default values
- A schema file can include other schema files
- → very like the include files in C++
- Examples:
  - Schema for Computer class, schema for an environment variable, schema for a Readout module, schema for information to publish on the information service

DATA files

Define the objects content

Can include other DATA files

Must include at least one SCHEMA file (for the object it wants to create)

Examples:

- All computers in the system are defined in a Computer.data.xml file
- Many instances of a DE10ReadoutModule are defined in DE10Remote.data.xml

Counter example: data for the IS monitoring are *not static* and *are not described* in a .data.xml file

TDAQ System description

Schema and data files are well suited for a description of the system

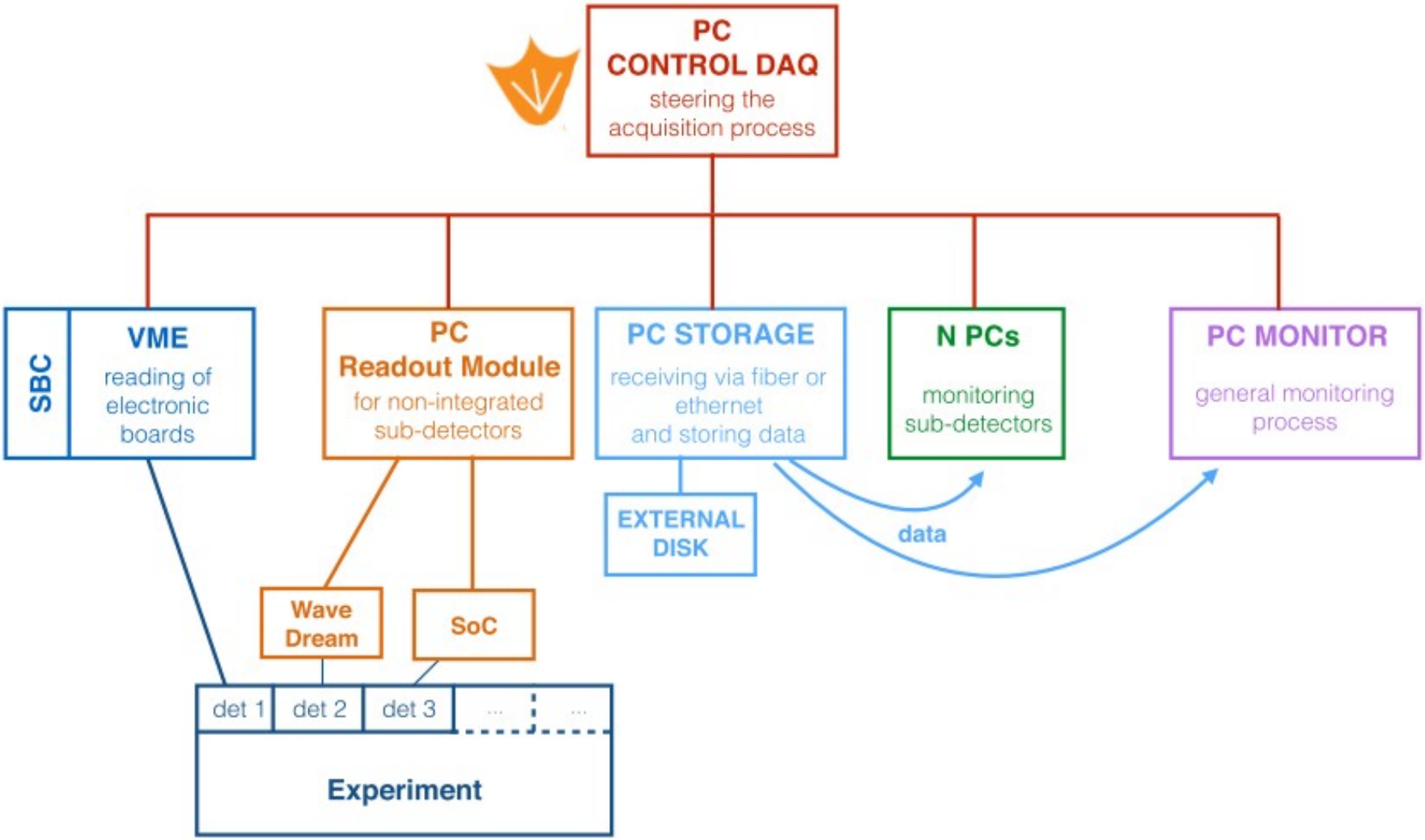
Object partition contains “segments”

Each “segment” can contain other “segments”

Or “Readout Systems” or “Readout Devices”

A “Readout Device” can contain many “Readout Modules”

Example 1 – GSI 2019 data taking



Example 1 : GSI 2019

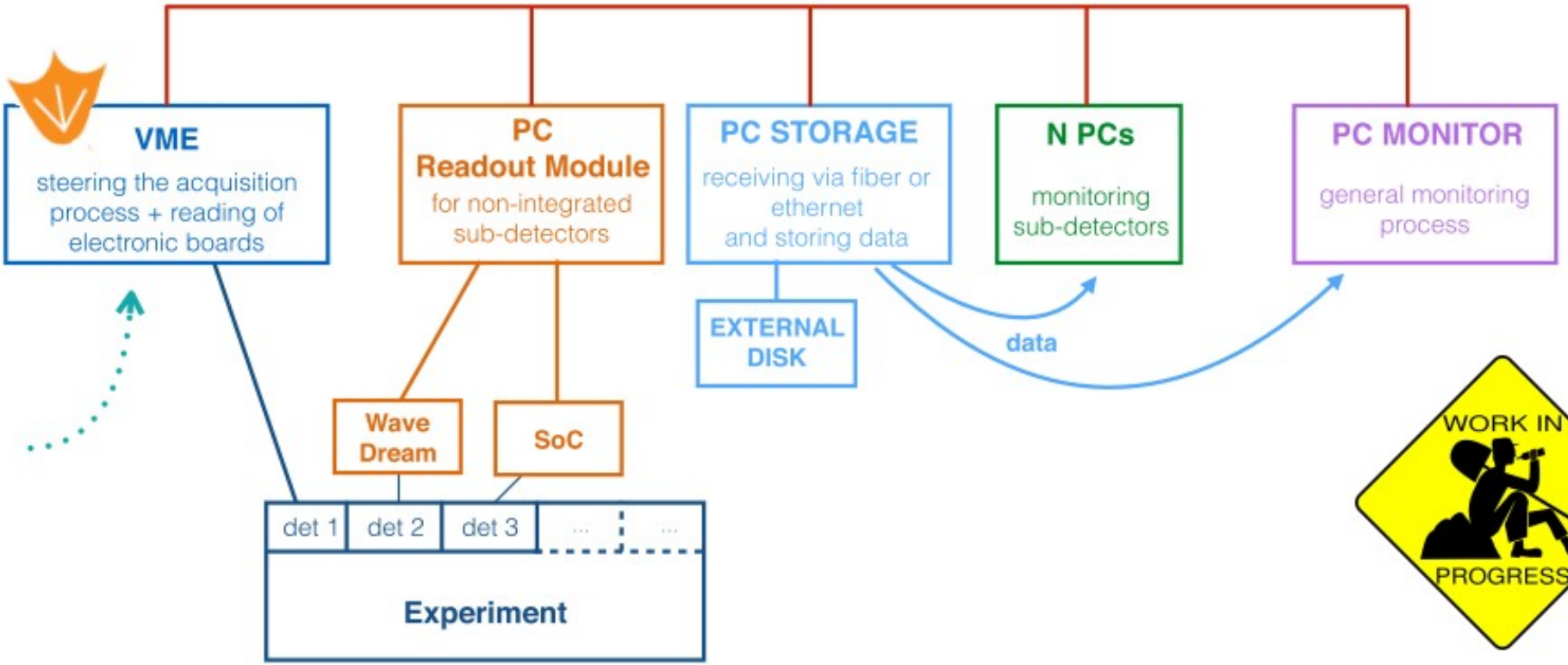
We used an SBC to read the VME modules, footbo2 to read WD and VTX, footbo1 to perform the Event Building

Partition: FootPartition (on footbo1)
Contains Segment: FOOTtest (on footbo1)
Contains segment FOOT_EB, RCD VMETRIG-RCD, REMOTE-RCD

RCD VMETRIG-RCD runs on the SBC and read V2495 and TDC data (partial event building)
RCD REMOTE-RCD runs on footbo2 and collects data from WD and VTX (partial event building)
Segment FOOT_EB runs on footbo1, collects data from SBC and footbo2 and provides the final event building

Example 2

- SBC replaced with **Bridge CAEN V2718**:
  - **PCI express card** with two optical links for PC interface;
  - channel bandwidth 1.25 Gb/s;
  - no boot required, ready at power ON!
  - Up to 80 MB/s sustained data transfer rate.



Example 2

When we are using a VME bridge all the daq runs on footbo1

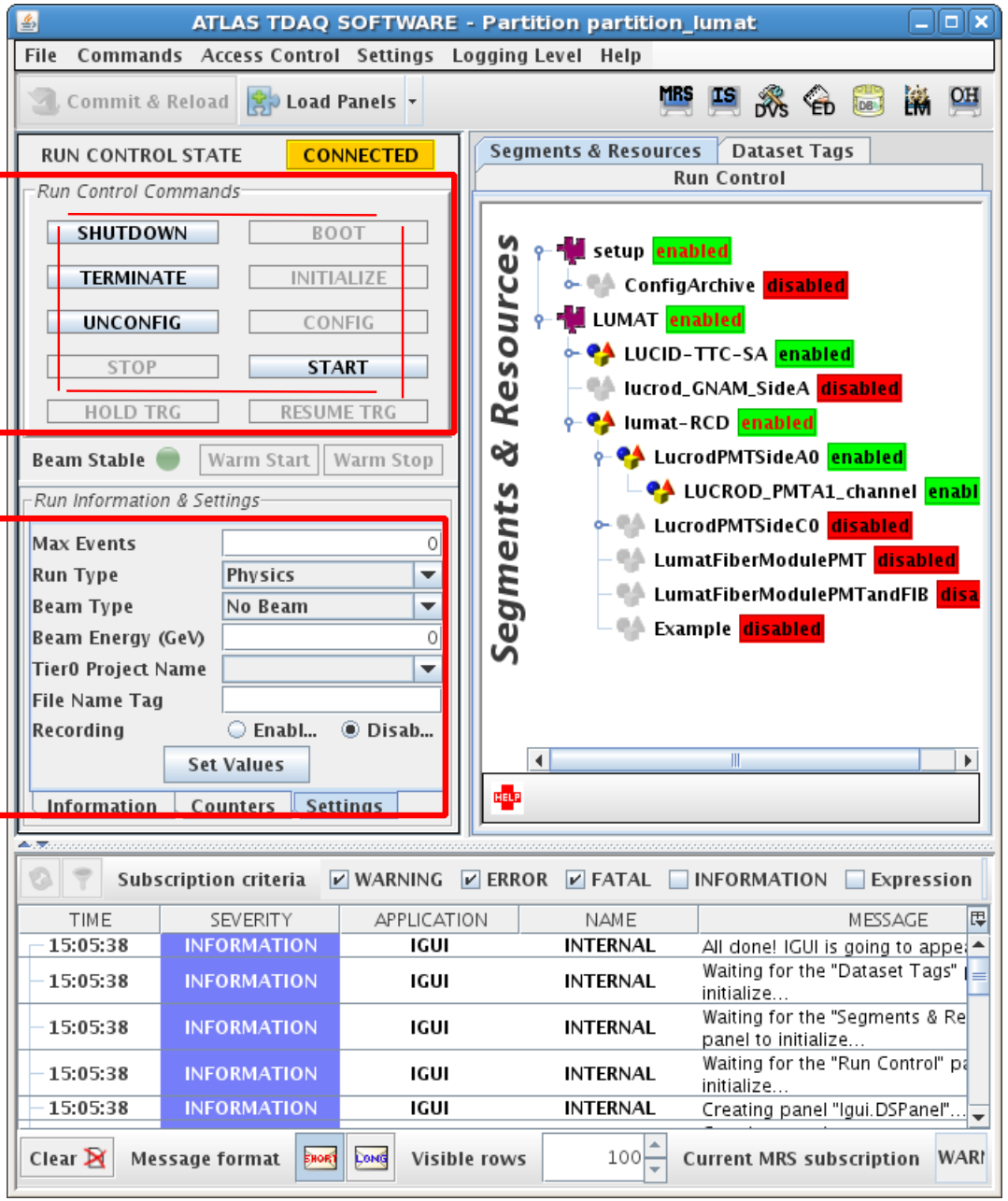
- Partition: FootVMEBrigdePartition
- Contains Segment: FOOTBridge
- Contains RCD: FOOT-RCD
- Contains all Readout Modules and performs the event building
-
-
- Easier system….. (apart from VME...)

Run Control and Controllable Objs

Main DAQ state

State Machine

Run settings

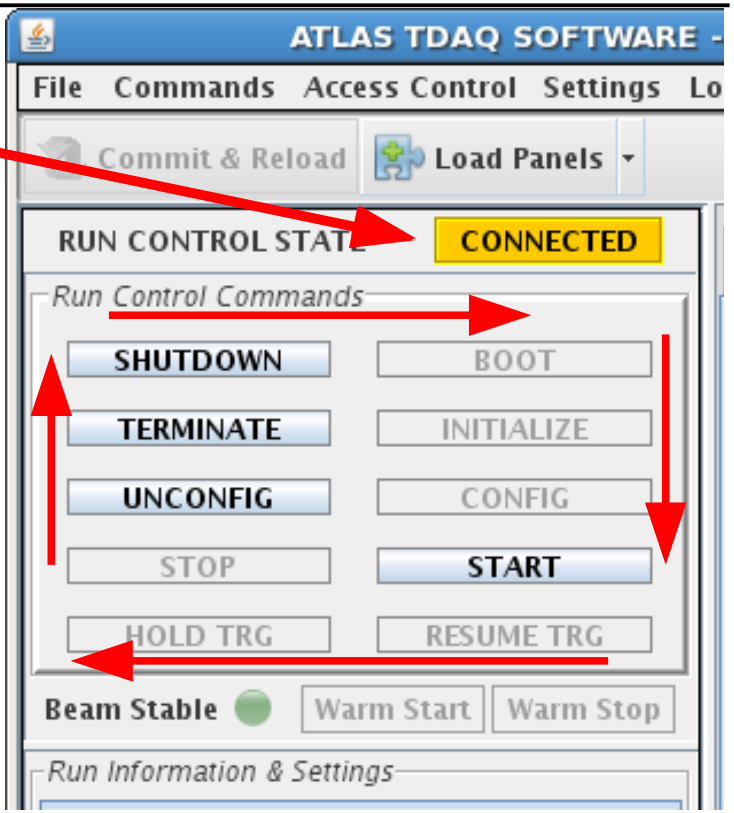


Logical view of the system

Information
Warnings
Errors

Finite state machine

The system is said to be in a given state (e.g.)
INITIAL
READY
CONNECTED
RUNNING



BOOT → loads IPC comunication programs on remote machines
INITIALIZE → load “controllable applications” on remote machines
CONFIG → calls “configure” on controllable apps
START → calls “prepareForRun”
STOP → calls “StopDC”
UNCONFIG → calls “unconfigure”
TERMINATE →stops controllable apps

Controllable application

A controllable application is a **main executable** holding an **instance** of a class derived from the Controllable class:

```
class Controllable : public ….  {
  public:
    virtual ~Controllable() noexcept {}

    virtual void configure(const TransitionCmd& cmd);
    virtual void connect(const TransitionCmd& cmd);
    virtual void prepareForRun(const TransitionCmd& cmd);
    virtual void stopDC(const TransitionCmd& cmd);
    virtual void disconnect(const TransitionCmd& cmd);
    virtual void unconfigure(const TransitionCmd& cmd);
    virtual void publish();

    virtual void user(const UserCmd& usrCmd);
    virtual void enable(const std::vector<std::string>& comps);
    virtual void disable(const std::vector<std::string>& comps);
  };
```

The structure of the main is almost fixed.

FOOT Example 1/3

At the end of a run we would like to copy the acquired file to a new storage area. We can do it with a scripts that starts at the end-of-run.

```
enum RunAt_t {AtConnect, AtPrepareForRun, AtStopArchiving,
              AtDisconnect};

class ExecScriptsActions: public daq::rc::Controllable {
public:
  ExecScriptsActions();
  ~ExecScriptsActions() noexcept;

  ExecScriptsActions(const ExecScriptsActions&) = delete;
  ExecScriptsActions& operator=(const ExecScriptsActions&) = delete;

  // Override only methods corresponding to "interesting" transitions
  void configure(const daq::rc::TransitionCmd&) override;
  void stopArchiving(const daq::rc::TransitionCmd&) override;

protected:
  void runScript();

private:
  unsigned long     m_runNumber;
  std::string       m_shell;     // 1 arg of execlp
  std::string       m_directory; // 2 arg of execlp
  std::string       m_script;    // 3 arg of execlp
  RunAt_t           m_runmoment;
};
```

FOOT Example 2/3

Schema file:

```
<class name="ExecScriptsApplication" description="Extension of a Run
                          Control application for EOR/SOR scripts running">
  <superclass name="RunControlApplication"/>
  <attribute name="shell" description="The shell to run" type="string"
                                                    init-value="sh"/>
  <attribute name="dir" description="where to find the script"
                                                 type="string" init-value=""/>
  <attribute name="script" description="The script itself" type="string"
                                               init-value="" is-not-null="yes"/>
  <attribute name="runat" description="when to run: 0-config,
       1-prepareforrun, 2-endofrun, 3-unconfig" type="u8" init-value="2"/>
 </class>
```

Data file:

```
<obj class="ExecScriptsApplication" id="script_app">
 … attributes for "RunControlApplication"...
 <attr name="shell" type="string">"/bin/sh"</attr>
 <attr name="dir" type="string">"/labdaq/FootTDAQ/scripts"</attr>
 <attr name="script" type="string">"EndOfRunScript.sh"</attr>
 <attr name="runat" type="u8">2</attr>  <!-- 2= stop archiving -->
</obj>
```

Now exists in the system an object "ExecScriptsApplication" with these attributes! It is an application of the FOOT_EB segment!

FOOT Example 3/3

- Code snippets...

```
void ExecScriptsActions::configure(const daq::rc::TransitionCmd& cmd) {
  daq::rc::OnlineServices& rcSvc = daq::rc::OnlineServices::instance();
  const ExecScriptsApplication*
trApp=rcBase.cast<ExecScriptsApplication>();
  if(trApp != nullptr) {
    m_shell = trApp->get_shell();
    m_directory = trApp->get_dir();
    m_script = trApp->get_script();
    m_runmoment = static_cast<RunAt_t> (trApp->get_runat());
  }
}

void ExecScriptsActions::stopArchiving(const
daq::rc::TransitionCmd& cmd) {
  if( m_runmoment==AtStopArchiving ) runScript();
}

void ExecScriptsActions::runScript() {
  std::ostringstream s;
  s<<m_shell<<" "<<m_directory<<"/"<<m_script<<" "<<m_runNumber;
  std::string strcmd(s.str());
  system(strcmd.c_str());
}
```

CORBA & IDL !

Readout Application

- The most important Controllable application is the ReadoutApplication. It is THE Readout program for each single RCD (PC in the DAQ).
- It is fully configurable via xml and plug-ins.
- It holds:
  - A set of **Readout Modules** (for actual read-out)
  - A **Trigger** plugin (for trigger generation)
  - A **Data Out** plugin (for final storage)

FOOT example 1 : GSI 2019

- VME boards where read out via a SBC.
- On that SBC there was a ReadoutApplication
- With:
  - A readout module for V2495 (trigger board)
  - A readout module for the V1190 (TDC)
  - A triggerIn plugin (V2495HWTriggerIn) for the generation of software triggers
  - A TCPDataOut plugin for shipping data to footbo1

We code for readout modules and the infrastructure takes care of the rest!

FOOT example 2 : GSI 2019

- The **event building** was made on footbo1
- On footbo1 there was a ReadoutApplication with
  - A readout module EthSequentialReadoutModule
    - With two input channels: from VME and from footbo2
  - A DataDrivenTriggerIn
  - A FileDataOut plugin for writing data to disk

All elements were provided by the TDAQ infrastructure!
Not a single line of code was written…..
…. but lot of time was spent on the xml files!

No need to mess with the Data Out plugins: TCP or FILE are enough!

Readout Module (s)

Plug-in to handle a specific part of the system (such as a TDC, a FADC, a DE10…)

- It will be an object of a ReadoutApplication
- Readout modules responds to DAQ transitions and have a set of **DataChannels (InputChannels in the xml files)**

```
class IOMPlugin : public daq::rc::Controllable {...};


class ReadoutModule : public IOMPlugin {
  public:
    virtual void setup(DFCountedPointer<Config> config);
    virtual const std::vector<DataChannel *> * channels() =0;
    virtual void clearInfo() = 0;
    Virtual void publish() = 0;
    ...
};
```

Readout modules respond to TDAQ transitions and have a specific way to receive the configuration → method “setup”

FOOT example : ModuleV1190

- Configuration data are stored in xml files

```
<obj class="ModuleV1190" id="TDC1190A">
 <attr name="VMEbusAddress" type="u32">0xe00000</attr>
 <attr name="BoardName" type="string">"V1190A"</attr>
 <attr name="TriggerMatchingMode"  type="bool">1</attr>
….
</obj>
```

- Data are read in during the initialize to config transition

```
void ModuleV1190::setup(DFCountedPointer<Config> configuration) {
  m_vmeAddress          = configuration→getUInt("VMEbusAddress");
  m_boardName           = configuration->getString("BoardName");
  m_triggerMatchingMode= configuration->getBool("TriggerMatchingMode");
  ...
}
.
```

FOOT example : ModuleV1190

- Configuration is set at the config to start transition

```
void ModuleV1190::configure(const daq::rc::TransitionCmd&){
  bool opened =  m_board.open(m_vmeAddress, 0x10000, VME_A32);
  // trigger window setting
  if(m_triggerMatchingMode){
       int ret = configureTriggerWindow();
       if( ret != 0)  ERS_INFO("unsuccessful trigger window configuration ");
  }
  ….
}
```

- Then prepare for a data taking

```
void ModuleV1190::prepareForRun(const daq::rc::TransitionCmd&){
  m_board.softwareClear();
  if(m_eventFIFOEnabled)
    m_board.prepareForBLT(m_board.getConnectionType());
}
```

- Where the actual data taking is performed ??

The DataChannel

- Data channels are the way to read data:

```
class DataChannel {
  public:
    DataChannel(unsigned int id, unsigned int configId,
                unsigned long physicalAddress) ;
    virtual ~DataChannel() ;
  private:
    static std::vector < DataChannel * > s_dataChannels;
    unsigned int m_id ;        //Logical ID
}
```

```
class SingleFragmentDataChannel : public DataChannel {
  public:
    SingleFragmentDataChannel(unsigned int id, unsigned int configId,
                              unsigned long rolPhysicalAddress,
                            DFCountedPointer<Config> configuration,
                            SingleFragmentDataChannelInfo* info);

    virtual ~SingleFragmentDataChannel();

    virtual int getNextFragment(unsigned int* buffer, int max_size,
                                unsigned int* status,
                                unsigned long pciAddress=0) = 0;
};
```

Data channels have memory where to store data

Data Channels are Module specific

- They are part of the Module class

```
class ModuleV1190 : public ReadoutModule{
  public:
    virtual const std::vector<DataChannel *> *channels();
  private:
    std::vector<DataChannel *> m_dataChannels;
};
```

- They are tuned for each device to read

```
class DataChannelV1190 : public SingleFragmentDataChannel {
  public:
    DataChannelV1190(u_int id, u_int configId, u_int rolPhysAddr,
                     DFCountedPointer<Config> config, V1190Board *m_aBoard);
    virtual ~DataChannelV1190() ;
    virtual int getNextFragment(u_int* buffer, int max_size,
                                u_int* status, unsigned long pciAdd = 0);
    ….
};
```

Creation and destruction in a Module

- Creation during a configure transition

```
void ModuleV1190::configure(const daq::rc::TransitionCmd&){
   ...
   //Create the DataChannel
   DataChannelV1190 *channel = new DataChannelV1190(m_id, 0,
                           m_rolPhysAdd, m_configuration, &m_board);
   m_dataChannels.push_back(channel);
   ...
}
```

- Destruction during an unconfigure transition

```
void ModuleV1190::unconfigure(const daq::rc::TransitionCmd&){
   // delete the datachannel created at configure!!
   while (m_dataChannels.size() > 0) {
     DataChannel *channel = m_dataChannels.back();     // reverse order!!
     m_dataChannels.pop_back();
     delete channel;                          // we created them
   }
 }
```

Description in data.xml files

```
<!-- Memory management -->
<obj class="MemoryPool" id="SBC-MemoryPool-1">
 <attr name="Type" type="enum">"CMEM_BPA"</attr>
 <attr name="NumberOfPages" type="u32">1</attr>
 <attr name="PageSize" type="u32">8192</attr>
</obj>

<!-- V1190 Input channel  -->
<obj class="HW_InputChannel" id="V1190InputChannel_0">
 <attr name="Id" type="u32">0x3130</attr>
 <attr name="PhysAddress" type="u32">1190</attr>
 <rel name="MemoryConfiguration">"MemoryPool" "SBC-MemoryPool-1"</rel>
</obj>

<!-- V1190 Readout module parameters -->
<obj class="ModuleV1190" id="TDC1190B">
 <attr name="VMEbusAddress" type="u32">0x200000</attr>
 <attr name="BoardName" type="string">"V1190B"</attr>
 <attr name="TriggerMatchingMode"  type="bool">1</attr>
 <!-- attributes of the Data Channel base class -->
 <rel name="Contains" num="1">
   "HW_InputChannel" "V1190InputChannel_0"
 </rel>
</obj>
```

Memory type and size to use

Data channel information

Unique idenfication in the system

The readout module with its channel(s)

OK… but how you read the data??

In the readout application a thread calls the “getNextFragment” function for **all channels** after every software trigger

```
int DataChannelV1190::getNextFragment(u_int* buffer, int max_size,
                                      u_int* status, unsigned long ){
  u_int bufPtr = buffer;
  *bufPtr++ = m_channelId;        // Write the channel number
  //Add time infos: seconds and microseconds from the epoch
  timeval tv;
  gettimeofday( &tv, NULL);
  *bufPtr++ = tv.tv_sec;
  *bufPtr++ = tv.tv_usec;

  *bufPtr++ = m_eventNumber++;     // Add current eventNumber number


   … then all the TDC data  !!
  }
```

WHO?

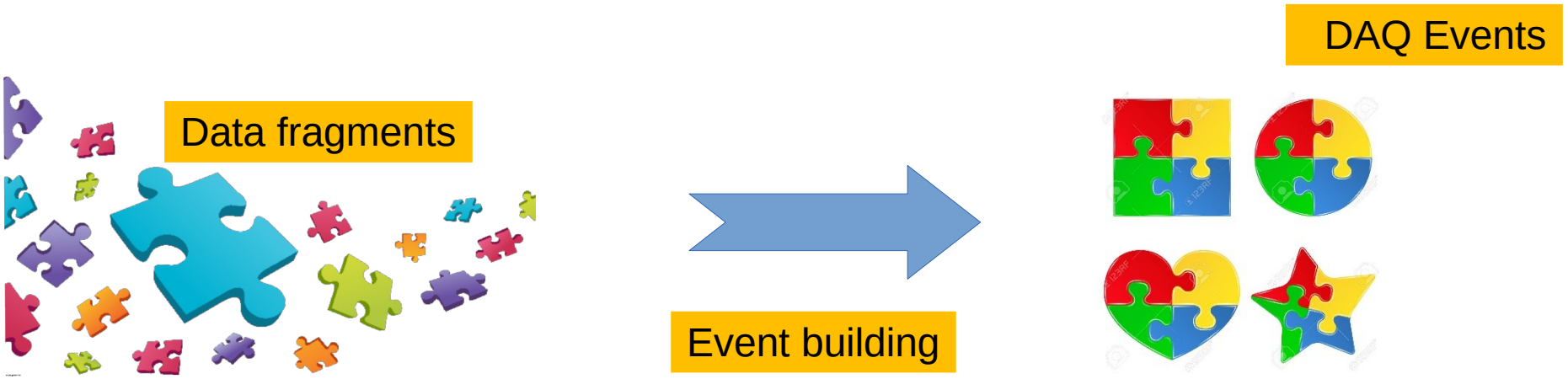
WHEN?

WHAT?

Readout modules do not call the getNextFragment!

Local event building

- The Readout application then builds a “local” event with all the data provided by each DataChannel
- Its not yet a FULL EVENT unless there is just ONE Readout application



What about the Trigger plugin ?

```
class TriggerIn : public DFThread, public IOMPlugin {};


class HardwareTriggerIn : public TriggerIn {
  public:
    virtual void setup(IOManager* iom, DFCountedPointer<Config> conf);
    virtual void prepareForRun(const daq::rc::TransitionCmd&);
    virtual void stopGathering(const daq::rc::TransitionCmd&);
    virtual void clear()=0;

  protected:
    virtual void run();
    virtual void cleanup();

    //! Method that actually waits for the hardware trigger
    virtual bool waitForTrigger()=0;
  …
}
```

Run() is called in a separated thread

Returns true when a trigger can be issued
False when not.

FOOT: V2495TriggerIn

```
class V2495HWTriggerIn : public HardwareTriggerIn {};


bool V2495HWTriggerIn::waitForTrigger(){
  if (m_busy) {
    return false;
  } else {
    m_fifoevts = m_board.getEventsWaiting();
    if( m_fifoevts>0 ){
      m_busy = true;
      return true;
    }
  }
  return false;
}


void V2495HWTriggerIn::clear(){
  // Note: this is called after every trigger!!
  m_busy = false;
}
```

Called continuously during running

No new trigger when busy is ON

New trigger!! turn the busy ON

At the end of single trigger processing

turn the busy OFF

Recap Readout Application

Read a subsystem running on a single PC

- It contains a triggerIn plugin to know when to start the DAQ
- It contains a vector of ReadoutModules that can have many DataChannels. The ReadMod configure the data taking, and the DataChannels perform the actual reading (in a separate thread)
- The collected data are then given in input to a DataOut object for file writing or data sending to an Event Builder PC.
- All elements are plugins that can be configured via xml
- **In TDAQ sessions we talk rarely of the readout application, but this does not mean it is not there!!**

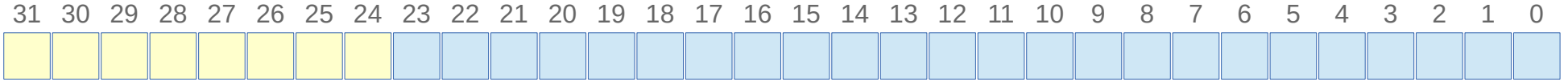
Data stream

A data file is waiting for you to process it !

1234cccc 00000004 0000000e 00000204 aa1234aa 00000081 00000018 05000000
00460039 00000001 00000000 00000000 5caa2a6c 2f0c0970 0000000d 00000000
0000000f 000008a4 00000000 0000000d 0000000d 00000000 00000000 00000069
00000000 00000000 00000000 00000000 dd1234dd 0000003d 0000000a 05000000
00460030 00000003 00000000 00000000 00000000 00000000 ee1234ee 00000009
03010000 00465230 000008a4 0000000d 0000000d 00000000 00000000 00463130
5caa2a2e 000c3968 0000000d 400001bf 0800d9a3 00080493 0068047d 00800648
00c80477 00200641 00500928 00b00919 00e0066c 00680917 00680b54 1800d00c
0900d9a3 0110062a 01781417 011008cc 01100f06 1900d006 8000029f 44000044
00463030 5caa2a2e 000c399d 0000000d 00061779 09059da7 000974fe 0000000d
0017189f 00020016 000000e0 33333333 00000000 00000000 00000002 00000025
00000001 dd1234dd 0000002c 0000000a 05000000 00460033 00000003 00000000
00000000 00000000 00000000 ee1234ee 00000009 03010000 00465231 000008a4
0000000d 0000000d 00000000 00000000 00463230 5caa2a3c 00058a2e 0000000d
12341234 00000000 00463330 5caa2a3c 00058a2e 0000000d 12341234 00000000
00463833 5caa2a3c 00058a2e 00000001 0000000d 55445544 22222222 00000003
00000003 00000000 00000003 00000013 00000001 1234cccc 00000004 0000000f
00000218 aa1234aa 00000086 00000018 05000000 00460039 00000001 00000000
00000000 5caa2a6c 2f0c0970 0000000e 00000000 0000000f 000008a4 00000000
0000000e 0000000e 00000000 00000000 0000006e 00000000 00000000 00000000
00000000 dd1234dd 00000042 0000000a 05000000 00460030 00000003 00000000
00000000 00000000 00000000 ee1234ee 00000009 03010000 00465230 000008a4
0000000e 0000000e 00000000 00000000 00463130 5caa2a2e 000c570a 0000000e
400001df 0800ef43 000803e4 006803d6 00b004bb 00c803e4 00200500 007005ae
0080057a 00d00648 00280523 007806d4 00b0089e 00e004fd 005009b9 00a00c30
00e00951 00c80d1e 1800e012 0900ef43 01080a63 0178139b 011009cf 1900e005

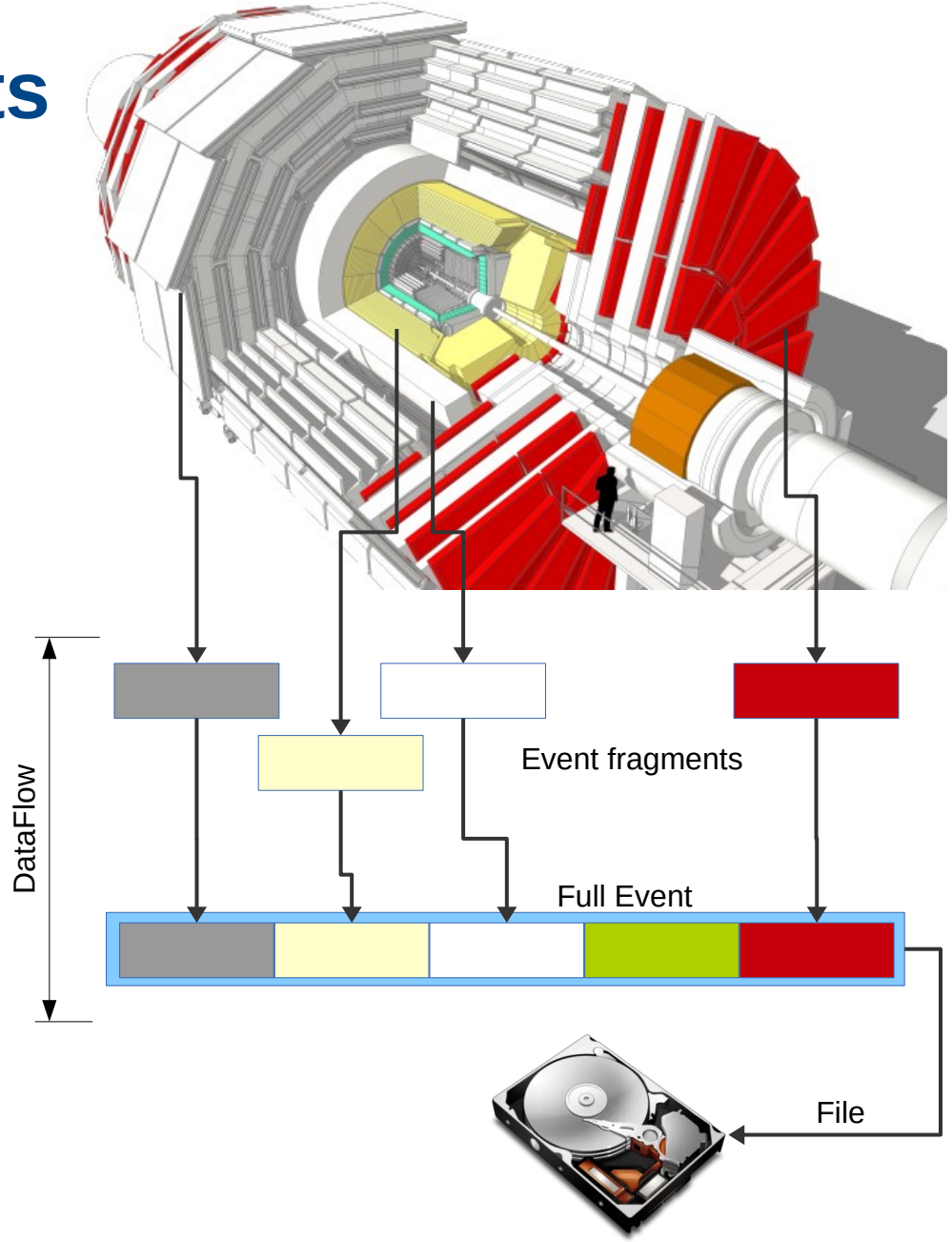
Data Encoding

- Data encoded in **digital** format
  - Arrays of words of fixed size: 2, **4**, 8 bytes
- The quantum of information must contain
  - A digital value + an unique channel identifier
- Example
  - Beam Monitor: channel ID and TDC counts
  - Tof Wall: channel ID and ADC counts
- For example, one can split a word in two
  - e.g. n bits for module id, 32-n bits for TDC/ADC counts
  - Number of used bits depends on ADC/TDC range



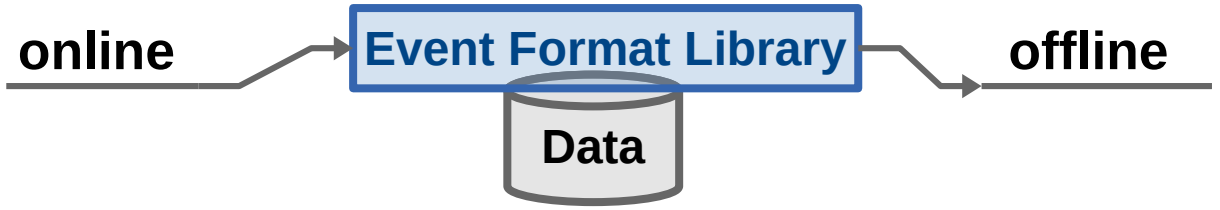
In case of multiple subdetectors

- Several data **fragments**
  - from different parts of the detector (**sources**)
  - **flowing** via buses and networks from readout system to event filter to data storage
  - to be **assembled** together in the event builder
  - to be **stored** on self consistent files



Event Format

- Necessary to define an **event format**
  - How event data is encoded, stored and decoded
- It is the core of your experiment
  - The bridge between **online** and **offline** worlds
  - Online for shipping data among data-flow components and for storage
  - Offline to access and decode the data for analysis
- The library implementing the format must be unique and shared between online and offline

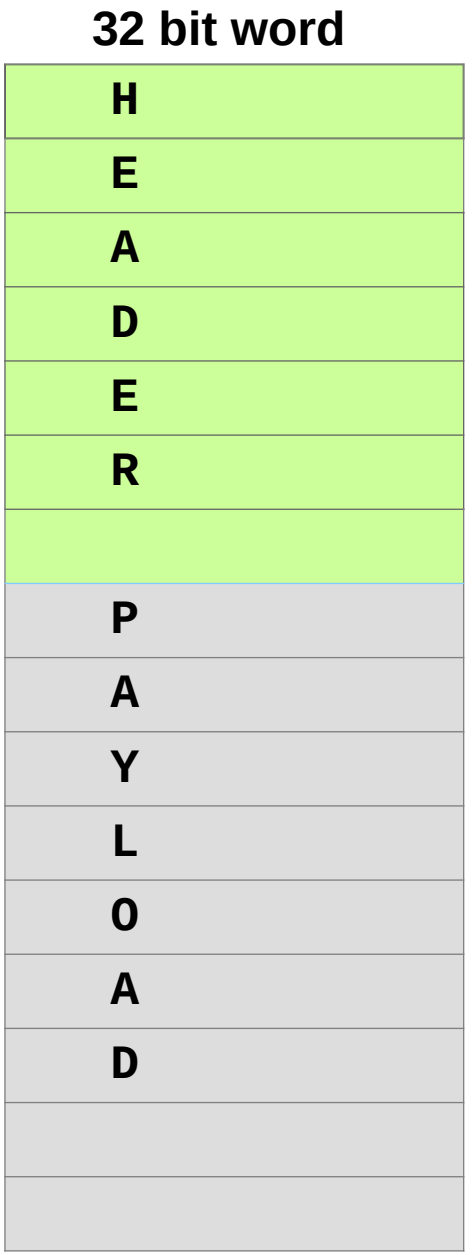


Event Format

- Identify every chunk of data, w/ a **source id**
  - Both during data taking and offline
- Associate data to the proper **time stamp**
  - to collect all fragments belonging to the same event
- Keep track of the event format **version number**
  - That may evolve during experiment lifetime
- Possibility to easily extend the format
    - e.g.: adding sub-detectors
- w/ some redundancies
  - For debugging purpose

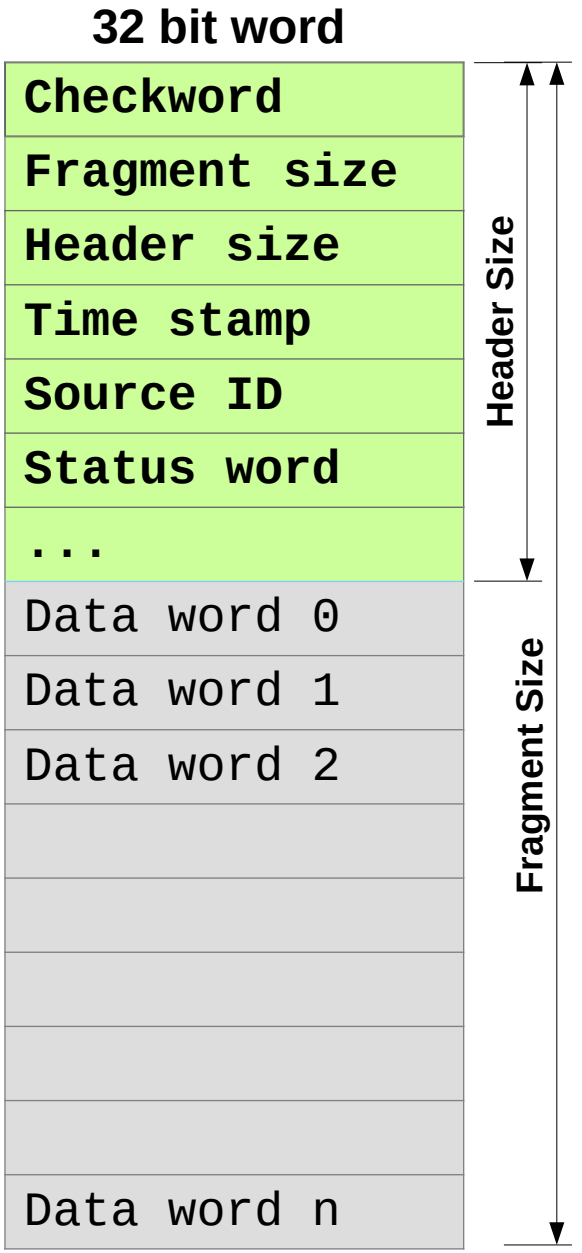
Header and payload

- Each data fragment composed by
  - A **payload**: the actual detector data
  - An **header**: that describes the payload
  - In some cases a **trailer**
- Header structure
  - **Checkword**: begin of frag. (0xEE1234EE)
  - **Fragment size**: where actual data ends
  - **Header size**: where actual data starts
  - **Time/bunchID**: timestamp
  - **Source ID**: where data is coming from
  - **Event ID:** event counter
  - **Error/status word(s)**: truncations, bad detector status, missing elements, ...



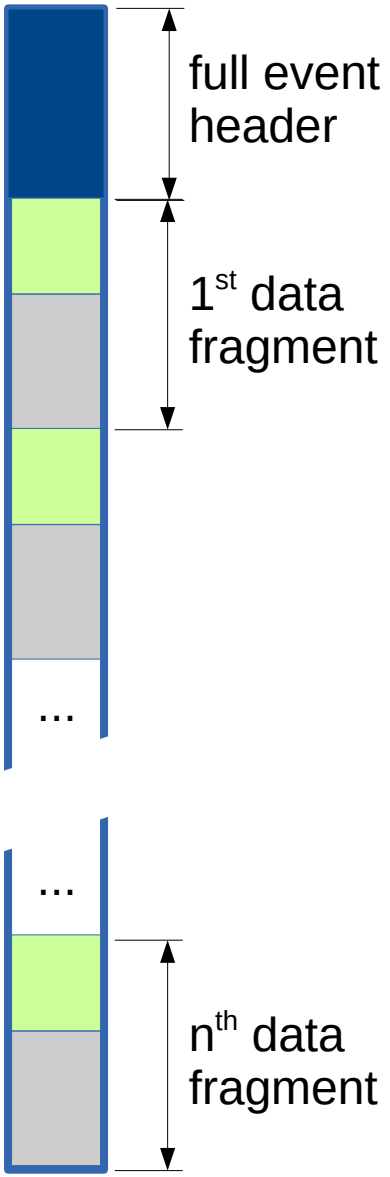
Header and payload

- Each data fragment composed by
  - A **payload**: the actual detector data
  - An **header**: that describes the payload
  - In some cases a **trailer**
- Header structure
  - **Checkword**: begin of frag. (0xEE1234EE)
  - **Fragment size**: where actual data ends
  - **Header size**: where actual data starts
  - **Time/bunchID**: timestamp
  - **Source ID**: where data is coming from
  - **Event ID:** event counter
  - **Error/status word(s)**: truncations, bad detector status, missing elements, ...



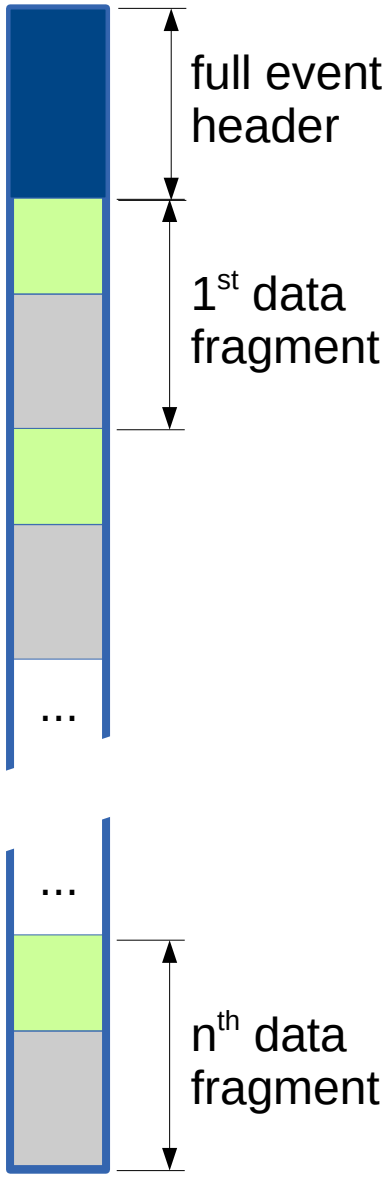
Full event

- A full **event** is a collection of **fragments**
  - There could be intermediate containers
- A full event is composed by
  - A **payload**: the “array” of data fragments
  - An **header**: that describes the event and is the portal to the collection of fragments
- Application reading a file must be able to
  - Find the 1st full event header
  - Navigate among the fragments
    - NB: fragment size word in each header
  - Up to the next event or the end of file



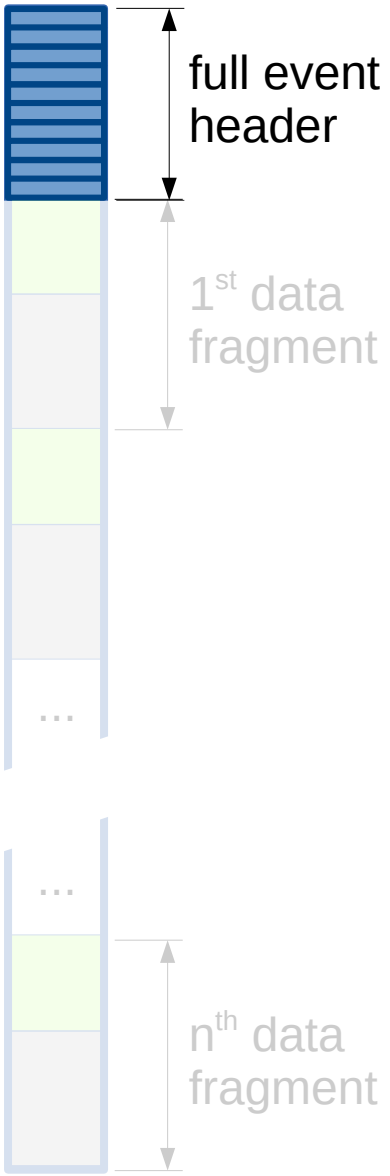
Full event

- A full **event** is a collection of **fragments**
  - There could be intermediate containers
- A full event is composed by
  - A **payload**: the “array” of data fragments
  - An **header**: that describes the event and is the portal to the collection of fragments
- Application reading a file must be able to
  - Find the 1st full event header
  - Navigate among the fragments
    - NB: fragment size word in each header
  - Up to the next event or the end of file

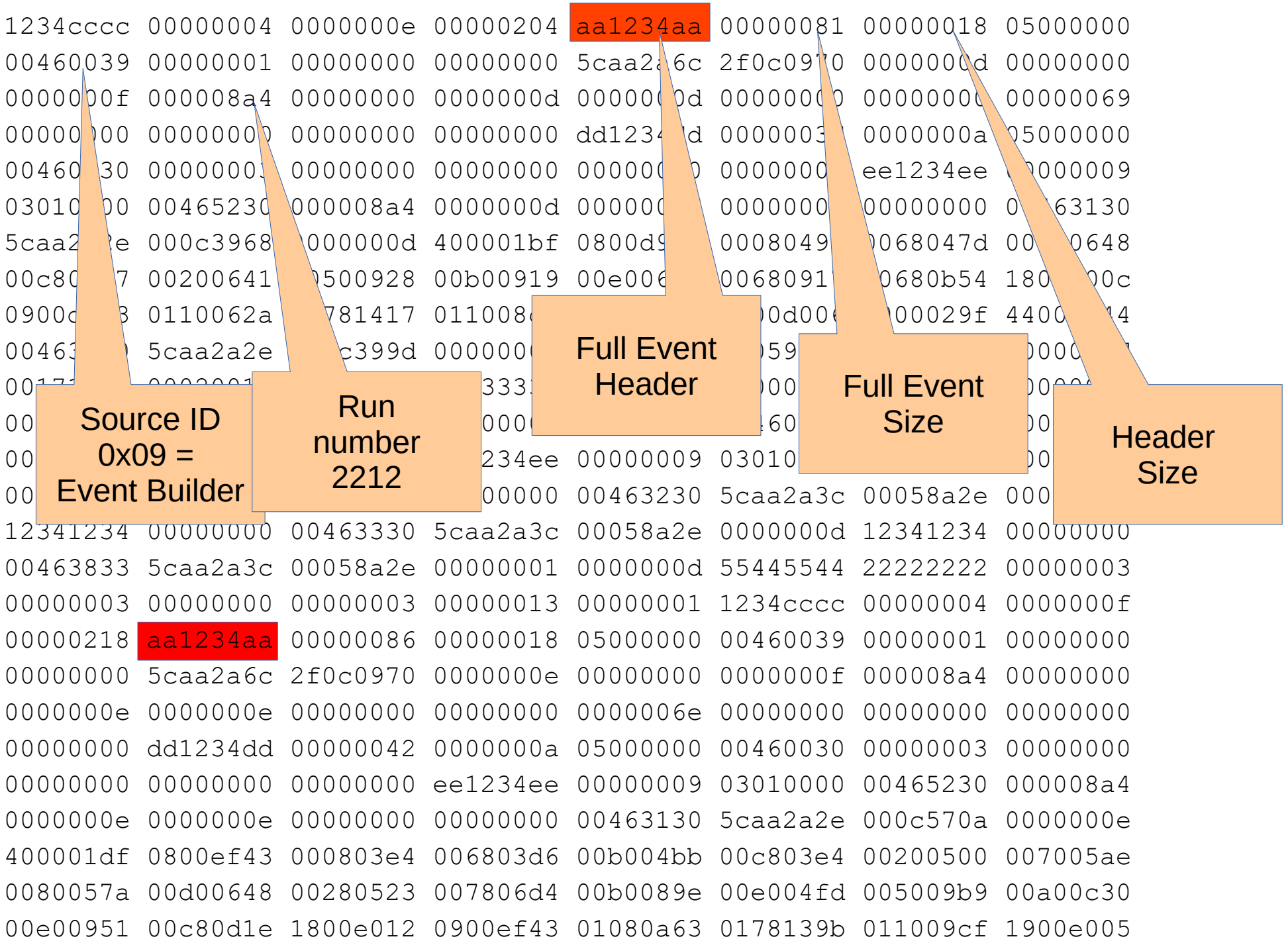


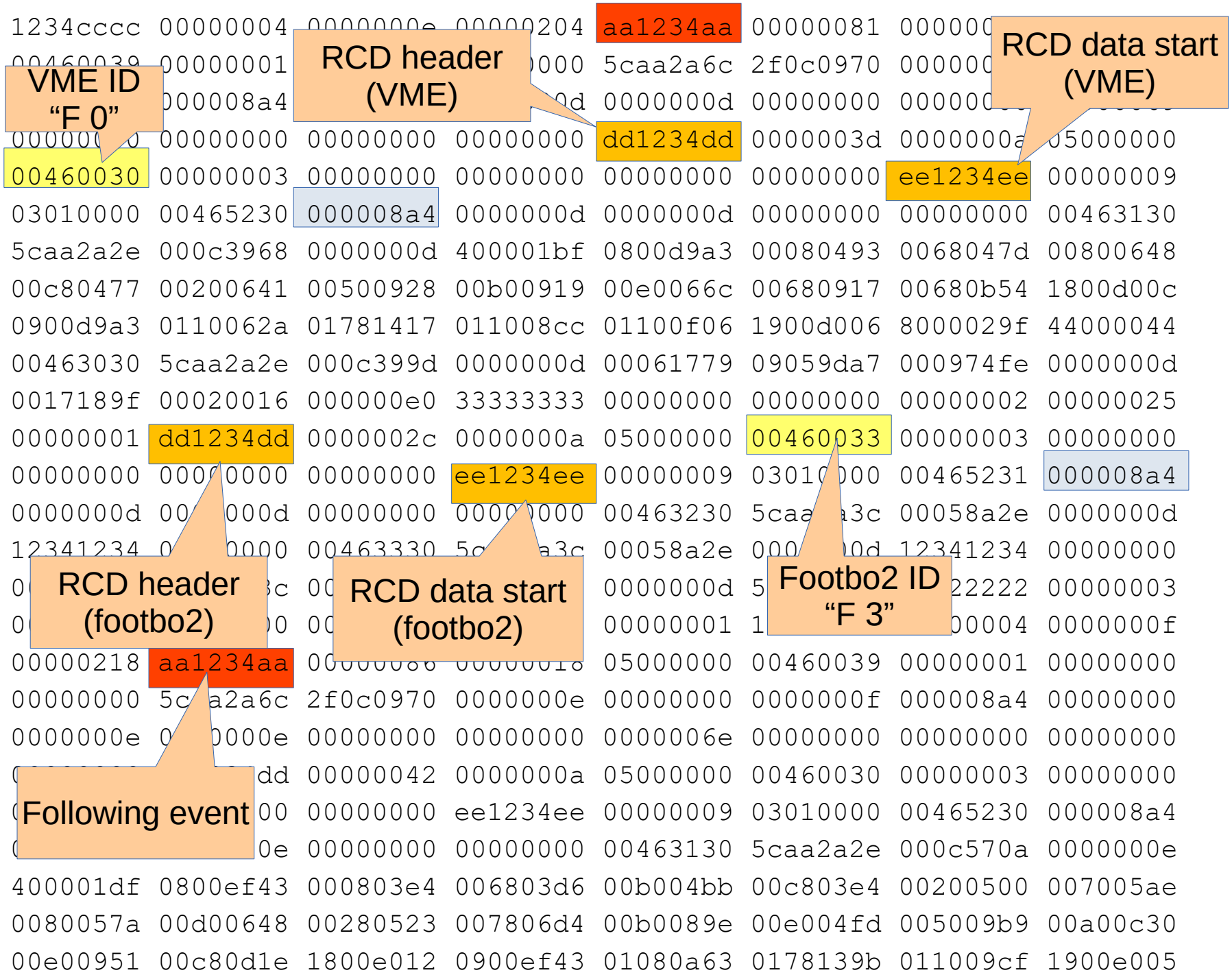
Full event header

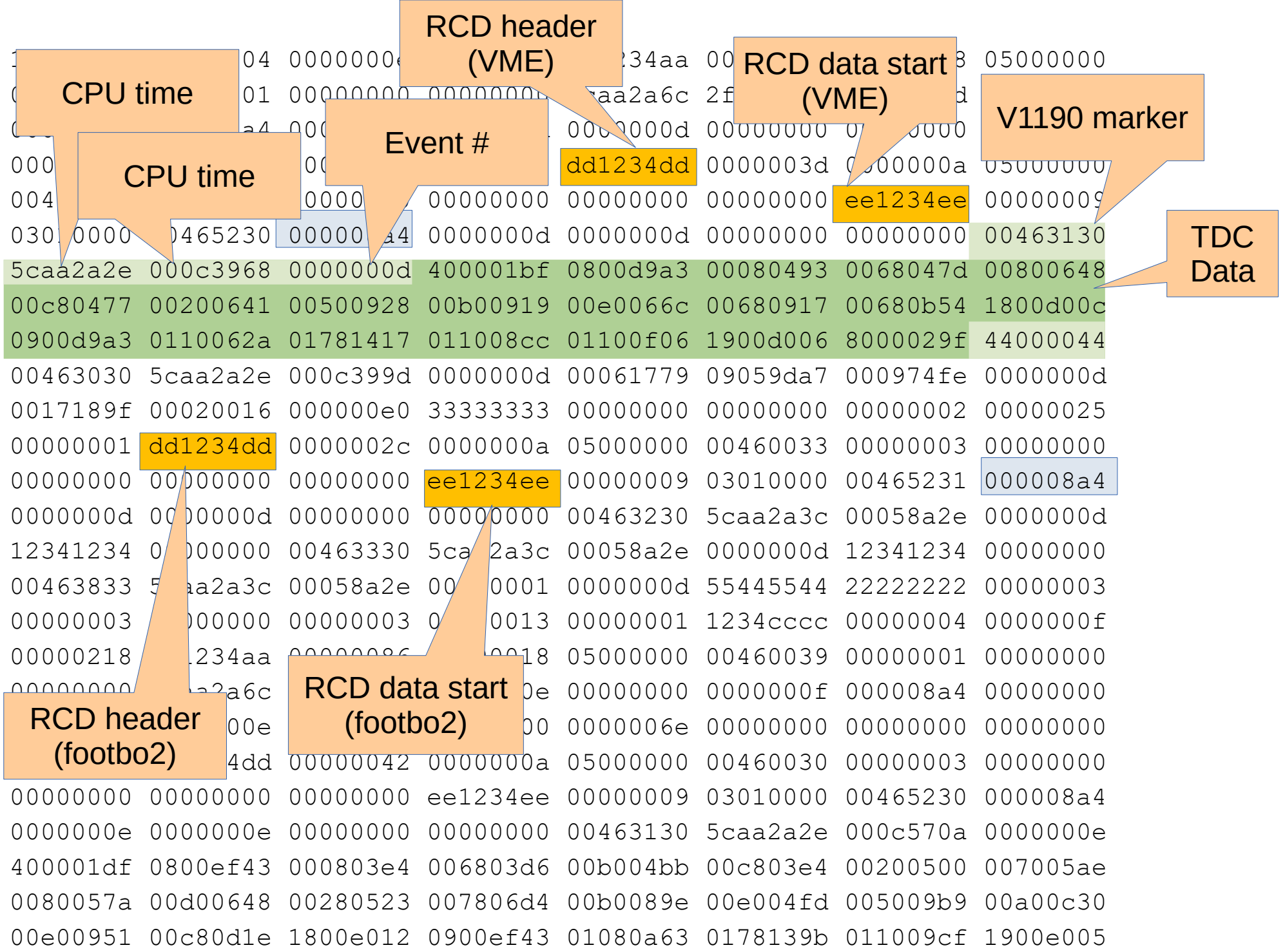
- **Checkword**: begin of frag (e.g.: 0xAA1234AA)
- **Fragment size**: where actual data ends
- **Header size**: where actual data starts
- **Time/bunchID**: timestamp
- **Run number**
- **Event classification**
- **Error words**
- Array of offset (one for each fragment)
  - Implemented only if random access is required
  - Otherwise, just navigate from fragment to fragment

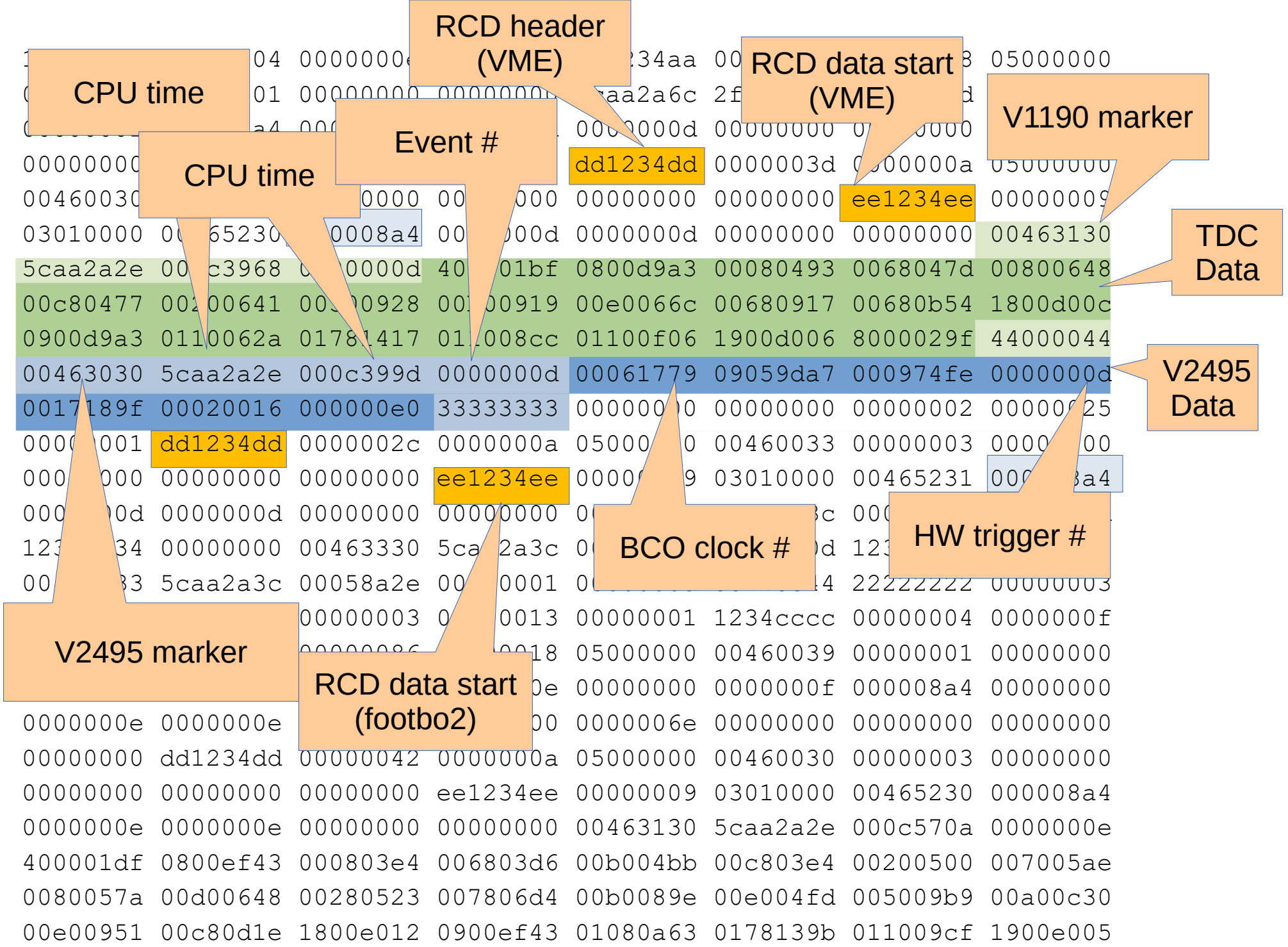


1234cccc 00000004 0000000e 00000204 aa1234aa 00000081 00000018 05000000
00460039 00000001 00000000 00000000 5caa2a6c 2f0c0970 0000000d 00000000
0000000f 000008a4 00000000 0000000d 0000000d 00000000 00000000 00000069
00000000 00000000 00000000 00000000 dd1234dd 0000003d 0000000a 05000000
00460030 00000003 00000000 00000000 00000000 00000000 ee1234ee 00000009
03010000 00465230 000008a4 0000000d 0000000d 00000000 00000000 00463130
5caa2a2e 000c3968 0000000d 400001bf 0800d9a3 00080493 0068047d 00800648
00c80477 00200641 00500928 00b00919 00e0066c 00680917 00680b54 1800d00c
0900d9a3 0110062a 01781417 011008cc 01100f06 1900d006 8000029f 44000044
00463030 5caa2a2e 000c399d 0000000d 00061779 09059da7 000974fe 0000000d
0017189f 00020016 000000e0 33333333 00000000 00000000 00000002 00000025
00000001 dd1234dd 0000002c 0000000a 05000000 00460033 00000003 00000000
00000000 00000000 00000000 ee1234ee 00000009 03010000 00465231 000008a4
0000000d 0000000d 00000000 00000000 00463230 5caa2a3c 00058a2e 0000000d
12341234 00000000 00463330 5caa2a3c 00058a2e 0000000d 12341234 00000000
00463833 5caa2a3c 00058a2e 00000001 0000000d 55445544 22222222 00000003
00000003 00000000 00000003 00000013 00000001 1234cccc 00000004 0000000f
00000218 aa1234aa 00000086 00000018 05000000 00460039 00000001 00000000
00000000 5caa2a6c 2f0c0970 0000000e 00000000 0000000f 000008a4 00000000
0000000e 0000000e 00000000 00000000 0000006e 00000000 00000000 00000000
00000000 dd1234dd 00000042 0000000a 05000000 00460030 00000003 00000000
00000000 00000000 00000000 ee1234ee 00000009 03010000 00465230 000008a4
0000000e 0000000e 00000000 00000000 00463130 5caa2a2e 000c570a 0000000e
400001df 0800ef43 000803e4 006803d6 00b004bb 00c803e4 00200500 007005ae
0080057a 00d00648 00280523 007806d4 00b0089e 00e004fd 005009b9 00a00c30
00e00951 00c80d1e 1800e012 0900ef43 01080a63 0178139b 011009cf 1900e005


Source ID
0x09 =
Event Builder
Run
number
2212
Full Event
Header
Full Event
Size
Header
Size


VME ID
“F 0”
RCD header
(VME)
RCD data start
(VME)
RCD header
(footbo2)
RCD data start
(footbo2)
Footbo2 ID
“F 3”
Following event


RCD header (VME)
RCD data start (VME)
V1190 marker
CPU time
CPU time
Event #
TDC Data
RCD header (footbo2)
RCD data start (footbo2)


RCD header (VME)
RCD data start (VME)
V1190 marker
CPU time
CPU time
Event #
TDC Data
V2495 Data
V2495 marker
RCD data start (footbo2)
BCO clock #
HW trigger #

Identification of Data Channels

Fragment type	Fragment Source	ID	Description	Readout Module
0x30='0'	0x30='0'	'F00'	Trigger data	ModuleV2495
0x31='1'	0x30='0'	'F10'	DC - TDC board 0	ReadoutModuleV1190
0x31='1'	0x31='1'	'F11'	DC - TDC board 1	ReadoutModuleV1190
0x32='2'	0x30='0'	'F20'	Start counter and DE/DX	ModuleWD
0x33='3'	0x30='0'	'F30'	Vertex detector	ModuleVTX
0x33='3'	0x31='1'	'F31'	Inner tracker	ModuleDE0 ?
0x33='3'	0x32='2'	'F32'	Inner tracker	ModuleDE0 ?
0x34='4'	0x30='0'	'F40'	Outer tracker	ModuleDE0 ?
0x35='5'	0x30='5'	'F50'	Calorimeter	*ModuleRemote*
0x38='8'	0x30='0'	'F80'	Dummy empty from PC 0	ModuleEmpty
0x38='8'	0x31='1'	'F81'	Dummy empty from PC 1	ModuleEmpty
0x38='8'	0x32='2'	'F82'	Dummy empty from PC 2	ModuleEmpty

'F' stands for FOOT; defined as DETECTOR ID

DAQ concepts

DAQ Mentoring

-
- **enjoy** the DAQ