Guide to using external scaler triggers

Ron fox

Guide to using external scaler triggers

by Ron fox

Table of Contents

List of Examples

Preface

Starting with nscldaq-8.1-pre5, the production and high performanc production readout software are now capable of accepting external scaler triggers. This new feature allows users to override the default timed scaler trigger with either a pre-written or custom built scaler trigger manager.

This document describes this facility. In particular we describe:

- An overview of this new feature and how to select it.
- The pre-built scaler trigger classes, how to create instances (objects) of them so that they can be registered as scaler triggers.
- How to write your own custom scaler trigger classes.

Every effort has been made to ensure the accuracy of this document. If, however you discover errors, please report them to the NSCL Data Acquisition system defect tracking database so that we can fix them as soon as possible. The NSCL Data Acquisition system defect tracking system can be accessed at: http://daqbugs.nscl.msu.edu

Chapter 1. An overview of scaler triggers

By default NSCL readout software periodically reads a set of scalers. These scalers are generally used to track the counting rates in detectors used in an experiment. Prior to nscldaq-8.1-pre5, scaler readout was triggered periodically, and there was no way for users to override this periodic trigger.

With nscldaq-8.1-pre5 and later, a formal scheme for triggering scaler readout was added to the production readout software. This scheme relies on trigger objects registered to the production readout experiment. By default a periodic trigger is registered. It is possible for users to override this trigger by registering either a pre-built scaler trigger type or even by producing and registering their own scaler trigger objects.

Key players in this are:

- The CExperiment object which manages both event and scaler triggers when the run is active.
- A hierarchy of scaler trigger classes derived from the abstract base class CScalerTrigger
- The ability of the user to add code to the `CMyExperiment::SetupScalers` function that creates a trigger object and registers it to replace the default periodic trigger.

Perhaps the simplest way to illustrate how this is done is to show some code where the user has chosen to substitute a pre-built trigger for the periodic scaler trigger. In this example, the user is substituting the `CCAENV977ScalerTrigger` for the periodic trigger. This trigger uses the 15'th bit (numbered from 0) of a CAEN V977 coincidence/output register to trigger scaler readout.

In this example, all code shown are modifications made to the `Skeleton.cpp` of the production readout software:

Example 1-1. Substituting a `CCAENV977ScalerTrigger` scaler trigger

```
...
#include <CCAENV977ScalerTrigger.h>  ❶
...
void
CMyExperiment::SetupScalers(CExperiment& rExperiment)
{
    CReadoutMain::SetupScalers(rExperiment);

   // Insert your code below this comment.

    rExperiment.setScalerTrigger(new CCAENV977ScalerTrigger(0x12340000)); ❷
        ....


}
```

The numbers below refer to the numbers in the example:

❶ This line includes the header file that defines the pre-written trigger clases the user has chosen as a scaler trigger

❷ This line creates a new `CCAENV977ScalerTrigger` object for a module with base address 0x12340000 in VME crate 0. The call to `setScalerTrigger` for the experiment object replaces the currently registered scaler trigger (the default periodic trigger) with the CCAENV977ScalerTrigger that was created

In most cases, you can use a pre-written scaler trigger. Read the next chapter to learn about the pre-written scaler trigger classes, and how to create instances of them.

If none of the pre-written trigger classes will work for your application continue reading as it is possible to write your own trigger class and register it as the scaler trigger.

Chapter 2. Using pre-written scaler trigger classes

Several pre-written scaler trigger classes are supplied with the production readout libraries:

`CTimedScalerTrigger`

This is the default scaler trigger type. When initialized, it records the value of the `frequency` variable. It indicates a trigger approximately every `frequency` seconds.

`CCAENV262ScalerTrigger`

This scaler trigger uses a CAEN V262 I/O register to provide scaler triggers in a manner completely analagous to the *vme trigger*. It is not possible to a single CAEN V262 as both the event and scaler triggers.

`CCAENV977ScalerTrigger`

This scaler trigger uses a CAEN V977 I/O register to provide scaler triggers on the last input. It is possible to share this trigger module with one used to provide event triggers.

`CCBDCES8210ScalerTrigger`

This scaler trigger uses the IT4 input of a CES CBD 8210 parallel branch highway driver to trigger scaler readout. This module can be shared with one used to provide event triggers.

Warning

The CES CBD8210 module is obsolete and no longer maintainable the NSCL DAQ support group strongly discourages you from using this module in new setups. We also encourage users of this module to migrate away from it to e.g. the Wiener VC32/CC32 module set, or to a VME based system.

2.1. The `CTimedScalerTrigger` trigger

The `CTimedScalerTrigger` trigger is the default scaler trigger module. If the user does not register a replacement, an object of this class will be used to trigger periodic scaler readouts.

The constructor prototype of this trigger is shown below:

Example 2-1. `CTimedScalerTrigger` constructor

```
#include <CTimedScalerTrigger.h>  ❶
...
CTimedScalerTrigger(CExperiment* pExperiment); ❷
```

❶ If you have code that uses a `CTimedScalerTrigger` object you should include `<CTimedScalerTrigger.h>` This header defines the class and its interfaces.

❷ The constructor of the `CTimedScalerTrigger` takes a pointer to the experiment object (`CExperiment*`). The scaler trigger uses the experiment object to determine the elapsed run time. The trigger will also reference the TCL variable `frequency` to determine the number of seconds between scaler readouts.

Below is an example that shows how to explicitly request that scalers be read via the timed scaler trigger.

Example 2-2. Explicitly requesting the timed scaler trigger

```
...
 #include <CTimedScalerTrigger.h>
 ...
 void
 CMyExperiment::SetupScalers(CExperiment& rExperiment)
 {
     CReadoutMain::SetupScalers(rExperiment);

    // Insert your code below this comment.

     rExperiment.setScalerTrigger(new CTimedScalerTrigger(&rExperiment));
     ....

 }
```

2.2. The `CCAENV262ScalerTrigger` trigger

The `CCAENV262ScalerTrigger` trigger supports accepting scaler triggers from a CAEN V262 I/O register. This register is the same module that is used to accept standard VME triggers for the production readout software.

The CAEN V262 module does not have latching inputs. It is therefore necessary to externall latch the trigger to the module either with a gate and delay generator in latch mode, or via an NSCL latch module.

The trigger for the scaler should be plugged into the IN 0 NIM input. When the trigger has been accepted the SHP2 output will be pulsed. At that time, the trigger latch can be cleared.

If your experiment is using a VME trigger based on the CAEN V262 module it is important that if you choose to use a CAEN V262 module for scaler triggers that it be configured to a base addresss that does not match that of the CAEN v262 module used to supply VME triggers. Choose a base address other than 0x444400. An application note at: http://docs.nscl.msu.edu/daq/appnotes/vmeaddress.html provides information to help you lay out your VME address map.

To use the CAEN V262 as a scaler trigger you must:

1. Configure and install a CAEN V262 module into one of the VME crates in your system. Write down the base address you used to configure this module as you will need it later.
2. Arrange external trigger logic that latches the scaler trigger into the module's IN 0 until the SHP 2 output is pulsed
3. Modify `Skeleton.cpp` to create a `CCAENV262ScalerTrigger` object with the base address you wrote down in step 1 above, and register it with the experiment to be your scaler trigger.

The example below documents the prototype of the `CCAENV262ScalerTrigger` constructor.

Example 2-3. The `CCAENV262ScalerTrigger` constructor.

```
                ...
#include <CCAENV262ScalerTrigger.h>  ❶
...
CCAENV262ScalerTrigger(unsigned long base, unsigned long vmeCrate = 0); ❷
...
```

❶ The `<CCAENV262ScalerTrigger.h>` must be included in any source code files that use the `CCAENV262ScalerTrigger` class. This header file defines the class structure and its interfaces.

❷ The parameters of the constructor for the `CCAENV262ScalerTrigger` class define where in the VME space the module used for the trigger is located:

Parameter: *`base`*
Data Type: `unsigned long`
Usage: The base address of the module. This should be the value set in the module rotary switch with two extra hexadecimal zeroes appened. So if the rotary switches are set to be 1234, this should be `0x123400`

Parameter: *`vmeCreate`*
Data Type: `unsigned long`

Usage: The number of the VME crate the module is installed in. The cratelocator can be used to determine which VME crate is which on a multi-crate system. The default value is `0` which is the VME crate number assigned to crates in a single crate system.

The example below illustrates this for a CAEN V262 module configured with a base address of 0x123400 installed in VME crate 1.

Example 2-4. Installing a `CCAENV262ScalerTrigger`

```
...
 #include <CCAENV262ScalerTrigger.h>
 static const int scalerTriggerBase = 0x123400; // Modify this to match your module
 static const int scalerTriggerCrate= 1;        // Modify to match your crate.
 ...
 void
 CMyExperiment::SetupScalers(CExperiment& rExperiment)
 {
     CReadoutMain::SetupScalers(rExperiment);

    // Insert your code below this comment.

     rExperiment.setScalerTrigger(new CCAENV262ScalerTrigger(scalerTriggerBase,
                                                             scalerTriggerCrate));
         ....

 }
```

2.3. The `CCAENV977ScalerTrigger` trigger

The `CCAENV977ScalerTrigger` trigger allows you to use a CAEN V977 coincidence/pattern register bit to trigger scaler readout. You may use this module for both an event and scaler trigger. The event trigger will respond to all but bit 15 (numbered from 0). The scaler trigger will respond to bit 15.

To use the CAEN V977 as scaler trigger you must:

1. Configure and install one of these units in a VME crate in your setup. See the application note at http://docs.nscl.msu.edu/daq/appnotes/vmeaddress.html for some guidelines on choosing an unused base address for the module. Once you have chosen the base address, set the module rotary switches accordingly and write this base address down.

2. Set up external electonics to assert both the module gate and bit 15 when you want the scaler readout to be triggered. The CAEN V977 latches data presented during the gate time so it is not necessary to arrange an external latch.
3. Modify the `Skeleton.cpp` file to create a instance of the `CCAENV977ScalerTrigger` module and to register it as your scaler trigger.

The constructor for `CCAENV977ScalerTrigger` is defined as shown in the example below:

Example 2-5. The `CCAENV977ScalerTrigger` constructor

```
#include <CCAENV977ScalerTrigger.h>  ❶
CCAENV977ScalerTrigger(unsigned long base, unsigned int vmeCrate = 0); ❷
```

❶ The `<CCAENV977ScalerTrigger.h>` header defines the `CCAENV977ScalerTrigger` class and its interfaces. This header file should be included in any C++ source file that uses that class>

❷ The parameters of the constructor for the `CAENV977ScalerTrigger` class are:

Parameter: *`base`*
Data Type: `unsigned long`
Description: The base address you assigned to the module when you set its rotary switches

Parameter: *`vmeCrate`*
Data Type: `unsigned int`
Description: The number of the VME crate in which you installed the module. The cratelocator program can be used to determine which VME crate your module is installed in. The default for this parameter is `0` which is suitable for single VME crate systems.

The example below shows how to create a `CCAENV977ScalerTrigger` object and set it as the scaler trigger.

Example 2-6. Using the CCAENV977ScalerTrigger to trigger scaler readouts

```
...
 #include <CCAENV977ScalerTrigger.h>
 ...
 // Modify these to reflect your choices

 static const unsigned long scalerTriggerBase = 0x12340000;
 static const unsigned long scalerTriggerCrate= 0;
 ...
 void
```

```
CMyExperiment::SetupScalers(CExperiment& rExperiment)
{
    CReadoutMain::SetupScalers(rExperiment);

   // Insert your code below this comment.

    rExperiment.setScalerTrigger(new CCAENV977ScalerTrigger(scalerTriggerBase,
                                                            scalerTriggerCrate));
        ....

}
```

2.4. The `CCBDCES8210ScalerTrigger` trigger

Warning

The CES CBD8210 parallel branch driver module is no longer maunfactured and cannot be repaired if broken. The NSCL Data acquisitions group recommends that no new setups use this module and that existing setups that use this module migrate towards either the Wiener VC/CC32 module set for CAMAC instrumentation or away from CAMAC to VME based systems.

The `CCBDCES8210ScalerTrigger` class supports using the IT4 input of a CES CBD8210 parallel branch highway driver as the scaler readout trigger. The event trigger (the IT2 input) can also run through the same module.

To use the CES CBD8210 as a scaler trigger you must:

1. Choose a branch to use. Note that the branch number implies a VME crate number as well.
2. Arrange external electronics to pulse the IT4 input of the branch driver module you are using as the trigger.

The `CCBDCES8210ScalerTrigger` class has the following construction parameters:

Example 2-7. Constructing a CCBDCES8210ScalerTrigger

```
#include <CCBDCES8210ScalerTrigger.h>    ❶
CCBDCES8210ScalerTrigger(int b = 0);          ❷
```

❶ The `<CCBDCES8210ScalerTrigger>` header should be included in every source code module that will be interacting with `CCBDCES8210ScalerTrigger` objects. This header defines the class and its interfaces.

❷ The parameter of the constructor is the number of the CES CBD8210 branch highway branch. This defaults to 0.

The example below shows how to use the `CCBDCES8210ScalerTrigger` to provide scaler readout triggers.

Example 2-8. Using the CES 8210 to provide scaler triggers

```
...
 #include <CCBDCES8210ScalerTrigger.h>
 ...
 // Modify these to reflect your choices

 static const unsigned long scalerTriggerBranch = 1;
 ...
 void
 CMyExperiment::SetupScalers(CExperiment& rExperiment)
 {
     CReadoutMain::SetupScalers(rExperiment);

    // Insert your code below this comment.

     rExperiment.setScalerTrigger(new CCBDCES8102ScalerTrigger(scalerTriggerBranch));
         ....

 }
```

Chapter 3. Writing custom scaler trigger classes

If none of the pre-written scaler trigger classes described in the previous chapter are suitable for your application, you can write your own scaler trigger. Scaler triggers are classes that inherit from the abstract base class `CScalerTrigger` which is defined in the header `<CScalerTrigger.h>`. The important parts of this header are shown below:

```
class CScalerTrigger
{
public:
  CScalerTrigger();
  virtual ~CScalerTrigger();
private:                                              ❶
  CScalerTrigger(const CScalerTrigger& rhs);
  CScalerTrigger& operator=(const CScalerTrigger& rhs);
  int operator==(const CScalerTrigger& rhs) const;
  int operator!=(const CScalerTrigger& rhs) const;
public:

  //  The members below are usually overridden by concreate subclasses.
  //  only operator() must be overridden, all others have a default no-op
  //  implementation:

  virtual bool operator()()  = 0;                     ❷
  virtual void Initialize();                          ❸
  virtual void Cleanup();                             ❹
};
```

❶ It is not necessary to define copy constructor, assignment, and equality/inequality comparison operators. These are private in the base class and never implemented to ensure that attempts to use them will result in errors at program link time.

❷ Your scaler trigger class must implement a function call operator. This opertor is called to determine if the conditions necessary to read scalers have been met. If scalers should be read, this function should return `true`, if not, `false`.

❸ If your scaler trigger must do some initialization prior to the run being made active, it should override and implement this function to perform those actions. If your scaler trigger does not provide an `Initialize` member, the default is a function that does nothing

❹ If your scaler trigger must do some cleanup as the run is becoming inactive, it should override and implement this function to perform those actions. If your scaler trigger does not provide a `Cleanup` member, the default is a function that does nothing.

The example below shows the definition and implementation of the simplest form of Scaler trigger. `CNoScalerTrigger` This scaler trigger can be used when your data taking application does not need to read any scalers.

Example 3-1. The `CNoScalerTrigger`

```
#include <CScalerTrigger.h>                    ❶
class CNoScalerTrigger : public CScalerTrigger ❷
{
public:
   virtual bool operator()() {                 ❸
        return false;
    }
};
```

❶ In order to write a scaler trigger you must include the <CScalerTrigger.h> header. This header defines the base class for all scaler trigger classes.

❷ All scaler triggers must be public derivations of the base class `CScalerTrigger` this line declares a new class definition for `CNoScalerTrigger` that is derived from `CScalerTrigger`.

❸ All scaler triggers must implement `operator()` The implementation for the `CNoScalerTrigger` is a function that always returns `false`. This ensures that a scaler readout will never occur if a `CNoScalerTrigger` scaler trigger object is registered on the experiment object.

Appendix A. Annotated source code for the `CTimedScalerTrigger` class

For instructive purposes,we supply the entire header and source code for the `CTimedScalerTrigger` class. If you are contemplating writing a custom scaler trigger study of this example may be helpful.

Example A-1. `<CTimedScalerTrigger.h>`

```
#ifndef __CTIMEDSCALERTRIGGER_H           ❶
#define __CTiMEDSCALERTRIGGER_H
#include <CScalerTrigger.h>          ❷

//  Forward classes::

class CExperiment;                        ❸

/*!
    This class is a scaler trigger that can be used to do a timed periodic
    trigger of scaler readouts..  This trigger will in general be registered
    by default and must be overidden by the user if they want something different.

    We depend on the following:
    - The application has a ge5tScalerPeriod() member
    - The experiment has a GetElapsedTime function.

*/
class CTimedScalerTrigger : public CScalerTrigger     ❹
{
private:
  CExperiment*    m_pExperiment;                    ❺
  unsigned int    m_nLastTrigger;                   ❻
  unsigned int    m_nNextTrigger;                   ❼
  unsigned int    m_nTriggerInterval;               ❽

public:
  // Constructors and other canonicals.

  CTimedScalerTrigger(CExperiment* pExperiment);    ❾
  virtual ~CTimedScalerTrigger();                   (10)
private:                                            (11)
  CTimedScalerTrigger(const CTimedScalerTrigger& rhs);
  CTimedScalerTrigger& operator=(const CTimedScalerTrigger& rhs);
  int operator==(const CTimedScalerTrigger& rhs) const;
  int operator!=(const CTimedScalerTrigger& rhs) const;
public:

  // Overrides:
```

```
  virtual void Initialize();                          (12)
  virtual bool operator()();                          (13)
};

#endif
```

❶ This #ifdef is used to ensure that if the header is included twice in the same compilation the class will not be doubly defined. All headers should have an *ifdef gaurd* like this.

❷ Since the `CTimedScalerTrigger` class has `CScalerTrigger` as its base class, the `<CScalerTrigger.h>` header must be included to provide the compiler with the details about the structure of that class.

❸ This declaration allows the use of the type `CExperiment` as long as the details of the type are not needed. The `CTimedScalerTrigger` class contains a data member that is a pointer to a `CExperiment`. While it would be possible to just include the `<CExperiment>` header, this can lead to cases where header files will have circular dependencies that cannot be resolved by the compiler. A good rule of thumb is to do a *forward class definition* like this wherever possible in header files, reserving the include of the header until the implementation file.

❹ This line informs the compiler that we are defining a new class named `CTimedScalerTrigger`, and that this class should inherit all of the data and methods of the `CScalerTrigger` base class. This has several implications. The two that are important for this case are that `CScalerTrigger` defines a set of interfaces scaler triggers provide and override, and that a `CScalerTrigger*` or `CScalerTrigger&` that points to an object that is actually a `CTimedScalerTrigger` will exhibit the run-time behavior of a `CTimedScalerTrigger`. This second property is called *polymorphism* and is an essential component of object-oriented programming systems.

❺ The `m_pExperiment` pointer will point to the `CExperiment` object that controls the active readout. The implementation of this class will make use of some services exported by that object.

❻ The `m_nLastTrigger` member will be used to store the time at which the previous trigger fired.

❼ The `m_nNextTrigger` member will store the earliest time at which it is ok to declare a scaler trigger.

❽ The `m_nTriggerInterval` will store the time in seconds between each trigger.

❾ A constructor is declared and will be implemented. The constructor is a function that takes action when an object of this type is created. In this case, our constructor will initialize the data members to appropriate initial values. This includes setting the `m_pExperiment` variable.

(10) Declares a destructor. A destructor is called when an object of the type is destroyed either because it was `delete`ed, or because it went out of scope.

(11) These functions are forbidden. Forbiddenness is enforced by declaring the functions private, and never implementing them. This means that if an external object attempts to call them, the compiler will catch this as a violation of the privacy of these functions. If a function of `CTimedScalerTrigger` calls themm, the linker will notice that the functions are not defined.

(12) The `CTimedScalerTrigger` class will need to implement an override to the functionality of the `Initialize` function.

(13) As is the case for all concrete scaler triggers a `operator()` must be implemented.

Example A-2. `CTimedScalerTrigger.cpp`

```
#include <config.h>
#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif


#include <CTimedScalerTrigger.h>        ❶
#include <CExperiment.h>
#include <CReadoutMain.h>



/*!
   Construct a timed scaler trigger.  At construction time
   we just need to save the experiment object pointer.

   \param pExperiment CExperiment*
       Pointer to the experiment
*/
CTimedScalerTrigger::CTimedScalerTrigger(CExperiment* pExperiment) :  ❷
  m_pExperiment(pExperiment),
  m_nLastTrigger(0),
  m_nNextTrigger(0),
  m_nTriggerInterval(0)
{

}
/*!
   Destructor is basically a no op.
*/
CTimedScalerTrigger::~CTimedScalerTrigger()
{
}

/*!
    Initialization requires that we figure out when 'now' is, the
    scaler periodicity and when we read the next scaler.
*/
void
CTimedScalerTrigger::Initialize()               ❸
{
  m_nLastTrigger     = m_pExperiment->GetElapsedTime(); // start timing from 'now'.
  m_nTriggerInterval = CReadoutMain::getInstance()->getScalerPeriod()*10; // 10'ths
  m_nNextTrigger     = m_nLastTrigger + m_nTriggerInterval;

}
```

```
/*!
   Check for the trigger.  We have a trigger if the current elaspsed time
   is >= the next trigger time.  In that case we'll need to compute the
   next trigger time as well as return true.
*/
bool
CTimedScalerTrigger::operator()()            ❹
{
  unsigned int now = m_pExperiment->GetElapsedTime();
  bool     triggered= false;
  if (now >= m_nNextTrigger) {
    triggered = true;
    m_nLastTrigger = now;
    m_nNextTrigger = now + m_nTriggerInterval;
  }
  return triggered;
}
```

❶ We need to include the following headers:

`<CTimedScalerTrigger.h>`

The header for the class this file implements. This defines the class and its interfaces so that the compiler knows that what we implement is what is defined.

`<CExperiment.h>`

The header for the `CExperiment` class. This class provides services for the experiment as a whole. the `CTimedScalerTrigger` class will make use of the interfaces that access the time the run has been active.

`<CReadoutMain.h>`

Each Readout program has a singleton `CReadoutMain` object. This object includes interfaces that allow access to experiment wide settings, including various run state variables that we will need.

❷ Construction of the `CTimedScalerTrigger` object requires that the *`pExperiment`* pointer be saved so that the experiment object's services can be accessed by the other member functions. *`pExperiment`* is stored in the member data `m_pExperiment`

❸ The `Initialize` member is called when the run is about to become active. This can happen both because the run has begun or because a paused run is being resumed.

- We use the `GetElapsedTime` function in the experiment object to store the start time of the current time interval in `m_nLastTrigger`
- We use the `getScalerPeriod` function in the `CReadoutMain` singleton to obtain the value of the `frequency` Tcl variable. Note the use of the class function `CReadoutMain::getInstance` to return a pointer to the one and only `CReadoutMain` object.

- The time for the next trigger is then computed and stored in `m_nNextTrigger`.

❹ The `operator()` is called to poll for a scaler trigger. If the current elapsed run time is $\geq$ the next trigger time, it's time to declare a trigger. The subsequent trigger time is computed and the function returns `true`. if the current elapsed run time is < the next trigger time, the operator returns `false`