BNCS Scripted Panel Exercise

Abstract

An exercise that uses BNCS CPP scripting to provide a record button and inhibit functions

Table of Contents

1. Introduction

This exercise will create a panel which sends control instructions to the VT machine to start and stop recordings. If the output of the VT is routed to the input to the VT, recordings should be inhibited and the panel should indicate to the user that this has happened. Likewise, if the switch on your black box is on, recordings should also be inhibited.

You will be creating the visual panel and its code AND the panel it will sit on – and they will need creating a little bit at a time. You need to break it down into smaller bits that you can test and verify independently as you are unlikely to get it all written correctly the first time round.

The order you need to do the development in is as follows:

1. Create the visual panel of your scripted component
2. Make the buttons start and stop recordings
3. Create the unscripted panel the component will sit on
4. Add the code to make your switch inhibit recordings
5. Add a timer to make a ‘record inhibit’ message appear briefly
6. Add the code to monitor the status of the router
7. Make the router status also control the ‘record inhibit’ behaviour
8. Tidy up your code to remove hard-coded elements
9. Sit back and enjoy your beautiful panel

Note

The last page of this document contains a summary of the most-used bits of code

2. Create the UI of your scripted component.

Start by creating your panel - the scripted panel component - and make careful note of the ID of your labels and buttons. You always require wizard to create the BNCS UI and all the project templates and then modify the BNCS UI component. The wizard does a lot of behind the scenes configuration for the visual studio project. It will also create the "boiler-plate" code that this exercise refers to.

2.1. Make the buttons start and stop recordings

Let's make the buttons do something. Find the button callback section and add code to respond to your 'Start Recording' button – something like this:

Example1: Reacting to the start and stop buttons

```
1  if (b->id() == "startRec")
2   {
3    //send text to your message label
4   }
5  else if (b->id() == "stopRec")
6   {
7    //change the text on the label to say recording stopped
8   }
```

Build the project, run the file from VisEd, confirm that the buttons do what you expected them to. Ask for help if they don't!

Now that we've verified the buttons are doing things reliably, we need to make them do something useful. What we actually want them to create is a notification to the hosting panel, a hostNotify - like this:

```
hostNotify("vtControl=3");   //remember, 3 is the number you need
                             //to write in to slot 1 of the VT to
                             //set it to record

or

hostNotify("vtControl=1");  //…and 1 is what you need to send a
                            //stop to the VT
```

Having created the hostNotify, you can now use that in the connections engine, but right now it isn't populating any dropdown menus for you. In order to get this right, scroll down until you find the code for the connections events helper list. The boiler-plate code will say something like:

```
bncs_stringlist sl;❶
sl << "notify=*";❷
return sl.toString('\n');
```

❶ First this is declaring a stringlist of bncs type and calling it `sl`.
❷ It then takes what is written on the right-hand side of line two and appends it to the stringlist (that's what << means, add this one to the end of the list).

You could have a long list of items here - we've only got the one - replace the notify with your own vtControl so the line will read.

```
sl << "vtControl=*";
```

The last line is taking the list and returning it as an itemised list to the connections engine - '\n' will add in a new line between each item.

2.2. Create the unscripted panel for your component

Build your code - in order to test it we will now need to place the component onto a host panel, so create another panel - an unscripted panel this time. Place your scripted component onto it, as well as a VT control and a VT status component (make these two into vertical lists and buttons respectively). Connect your script to the VT control and see how your vtControl has now turned up as an event in your connections engine. Save and test your panel - you should now be able to start and stop the recording.

2.3. Add code to make your switch inhibit recording

We now want to think about the conditions we want around the recordings and the variables you will need to keep track of them. Create and declare a Boolean variable called something like recAllowed in the header and initialise it to false in the constructor code. First we will set up the condition for the switch on your box to inhibit recordings. Declare a Boolean variable called switchOn.

Just like we did with the vtControl in the events helper list, we will now add the line:

```
sl << "mySwitch=[value]";
```

to the connections command helper list. Like the events, it will work without this, but creating these lists makes sure that people don't mistype things and reminds them that these events/commands are available to them. This gives us a hook to hang the status of our switch on and makes sure the hook gets created with the value we want.

Now scroll back up the code a bit and find the parentCallback routines. Here are the shout-outs from the parent (or host), where it is sending info to the component. Find the boiler-plate code for myParam and adapt this for your device and index entries. The parentCallback object that is being returned here is a parameter-value pair, where the value is being returned as a bncs_string. What does that mean? Add in these lines, and we'll look at them in detail:

Example2: Code to set switchOn value from a command.

```
 1  if(p->command() == "mySwitch") // bncs_string can do ==
 2   {
 3    if(p->value() == 0)
 4     {
 5      switchOn = false;
 6     }
 7    else
 8     {
 9      switchOn = true;
10     }
11   }
```

The parameter-value pair is your command in the connections engine - what you've put in the second drop-down menu.

The parameter is the command, the value is - well, the value you've attached to that command. So if in the connections engine you had typed in ‘mySwitch=0’, then mySwitch is the command parameter and 0 is its value. Line 1 in the code above is therefore saying ‘that parameter-value pair you just received – is the command-parameter 'mySwitch' ? If the answer to that is yes, it then extracts the value (i.e. 0) from that sets the value of the Boolean variable switchOn accordingly. Easy-peasy!

It is worth noting that only a value of zero being handed to the code turns the switch off, anything else will indicate the switch is on (and therefor inhibiting recordings) - putting it the other way round (i.e. a value of 1 turns the switch on but anything else turns it off) is less secure. It is worth thinking about what rubbish a user might pass to your code and ensure you've written it in a safe way and that erroneous entries won't cause problems and crash your code.

Now go back up to your button callbacks and add in an if-statement to control whether the record-instruction is to be sent out or not. The control here should be whether the switch is on.

2.4. Use a Timer to remove 'record inhibit' message

Next we want to add in a timer to make the message flash up for a specified period of time (3-4 seconds) to warn the user that record wasn't possible.

Look at the beginning of the code – about line 9 – here there will be a boiler-plate-code-added line defining a timer called TIMER_SETUP as number 1.

Remember, this is an instruction to the preprocessor to go through the code before it is compiled and replace all occurrences of TIMER_SETUP with the number 1. You could yourself just use the number in your code – it won't make any difference at all to the final code – but having it given a name makes it more human-readable and memorable.

Scroll down to the very bottom of the file and you will find the timerCallback functions. If you look at the code for TIMER_SETUP, you will see that all it does is stop the counter. This is where you want to insert your code for what you want to happen on each tick of the timer clock. When you start a timer, you do that by placing a line like this at the right place in your code:

```
timerStart(TIMER_SETUP,500); // What you've called the timer
```

This says a) WHICH timer you want to start – you can have several of them and b) how frequent you want the ticks to occur – this is given in milliseconds. Every single ‘tick’ of the timer will create a callback event which will cause the timerCallback code to be executed – you do not use for-loops or anything like that within the timer code to control the ticking! That way madness lies and will just cause lots of very unpredictable behaviour. You stop the timer by issuing the command.

```
timerStop(TIMER_SETUP); //
```

…but you need to decide how you want to trigger this.

You can for instance leave the timer running until you press a button which issues the timerStop command, or you can create a control variable which you increment on each tick. If you want the clock to 'tick' ten times, you initialise the variable to 0 when you start the timer (be careful about where you initialise it – think about it!), start your timerCallback code with a test on whether the control variable is less than 10 and if it isn't you issue the timerStop, and finally increment the variable.

How would you go about making the message change colour on each tick, say between red and grey – so it looks like it's flashing? Several different ways of achieving this, but all with involve testing for a condition.

2.5. Add the code to monitor the status of the router

Now think about the variables you will need in your code. You will need one for the router device you will be monitoring, one for the destination index and one for the VT source. They will need declaring in the private section of your header file, like this:

```
int myDev;
int myIndex;
int myVT;❶
```

❶ I like using 'my' as a prefix to variables as it shows I've created them and their for my use and they definitely won't be reserved keywords.

To start with, just initialise these variables in the constructor-part of the code (at the very top of the cpp-file, just under the definitions and inclusions) like this:

```
myDev = 15;
myIndex = 8;
myVT = 9;
```

We want to think about making the script monitor the status of the router to check whether the output of the VT is routed to the input to the VT. First you need to register your interest in receiving updates from the router – in v4.x this does not happen automatically. To make things simple we will just put the code for this into the constructor for now, at the top of the code. Find the line that shows your panel and enter these two lines:

```
routerRegister(myDev, 1, 12, false);❶
routerQuery(myDev, 1, 12);❷
```

❶ Registers your interest in revertives (updates) from device 15 (myDev), from destinations in the range 1 through to 12 and replace existing registration.

The last parameter is whether this registration is an additional router registration or a replacement registration – with this set to false it will replace any previous registrations with this new one. You don't want to end up having registered for revertives from lots of devices or you might end up flooding your system.

❷ Query router device 15 for destinations 1 through 12.

This forces a query of the router and makes the router tell you about the status of the specified destinations. Again, this is a range and if the device is very large you may wish to think about whether you need to know the status of all destinations – receiving 8000 revertives may slow your code down somewhat.

You can use either routerQuery or routerPoll for this – the difference is that routerQuery sets up a private session for these updates and therefore only the machine running this panel will receive the updates whereas a routerPoll will cause the revertives to be sent to every machine who has registered for revertives. Again, this can flood a system if you're not careful.

2.6. Use router status to control the 'record inhibit'

Now that your code is monitoring the router you'll need to add the code to make it do something useful when a revertive arrives. Scroll down to just below the buttonCallback routine, here's your revertive code.

You will need to put in some if-tests (or a switch statement) to check which device the revertive came from, which destination – and whether the source routed is the one you care about – and set a control variable accordingly.

Finally you will need to make sure that your button code is updated such that the record instruction only gets sent if the switch is in the correct position AND the VT is not routed to itself.

2.7. Tidy up your code to remove hard-coded elements

The code now basically does everything we asked for, but you can refine it and make it better.

We have hard-coded a number of things here which limits how reusable the code is – let's look at making this more flexible. The most obvious thing is setting the router destination from the connections engine.

First find the section in the `parentCallback` which is headed 'CONNECTIONS COMMANDS HELPER LIST' and add a new entry to the `stringlist`. This is the command you want people to send stuff to in the connections engine, so for instance for your index number I'd add something like:

```
sl << "myIndex = [value]";
```

This will populate the dropdown box in the GUI – but to be able to use it you now need to create the command callback. This needs to be another 'else if' statement, like this:

```
else if (p->command() == "myIndex")
  {
   //do stuff
  }
```

The value that has come in with this command is available to you as p->value(), but it does come as a bncs_string. This is one of the quirks of BNCS, all the data that is passed to the code is sent as type bncs_string.

The variable we need to set is `myIndex`, but this is an int and the code might whinge at you if you try just assigning the value to the variable – it is at least bad form to do this. There is however a method available that will change the string into an integer in a safe way and it is called `toInt()` – call it by typing

```
p->value().toInt()
```

and NOW you can assign the result to myIndex.

You can now alter your unscripted panel to pass the index to the component, for instance using a line edit box and '`myIndex=[value]`' should now be available in the dropdown menu in the connections engine.

The device number we probably want to have a bit 'harder' coded, but again not actually inside the code.

Just like with the index, create a

```
p->command()=="myDev"
```

else if statement to assign your device number, but don't add the entry to the connections engine helper list.

Instead find the very first line of the parent callback,

```
if (p->command()=="return")
```

and go down to the section for `p->value()=="all"`.

As the comment below indicates, the `stringlist` here gets returned to the visual editor itself, rather than the connections engine. Add your line to the list – syntax like so:

```
sl << bncs_string("myDev=" + m_dev);

or

sl << bncs_string("myDev=%1").arg(m_dev);
```

These two lines achieve the same thing – they take the string "`myDev=`" and appends the value of the variable m_dev to it, creates one bncs_string and returns this to the `stringlist`.

You will find the second line in use fairly frequently though so you may as well get used to the syntax. %1 means 'replace this with the first argument I'm about to supply' and you could have a long list of these, so if you followed up with %2 you'd replace it with

the second argument and etcetera. Remove the hard-coded initialisation from your constructor-code – but think about what you want to do with your router registration-and-query code – this is now in the wrong place.

Where do you need to move it to? When you've saved and built your solution, open up the unscripted panel in VisEd and double-click on the scripted component. You should now find that myDev has popped up in the box at the bottom, complete with a parameter-value pair where you can set the value here – just type into the box. Verify that the panel still works as it did earlier.

Finally, let's move the source for the VT out of the hard-coded section too. Here we're going to use the third option available for setting object information – the objectsettings.xml file. Here's a snippet from an `objectsettings.xml` file on a different system for inspiration and to show the syntax – use this to create a suitable entry in your (at present empty) file:

objectsettings.xml

```
<object id="AVON_ROUTER">
  <setting id="cue_router_number" value="812" />
  <setting id="sources" value="1,2,3,4,5,6,7,8" />
  <setting id="cue_OFF_source" value="16" />
</object>
```

…and in the code this is called thusly:

```
m_iCueRouter=getObjectSetting("AVON_ROUTER", "cue_router_number");
```

Remember that the `getObjectSetting` method returns a `bncs_string` object, so you will need to make it into an integer for your code.

Enjoy!

A. Useful methods and commands:

Some useful bits of C++ for the panel.

butonCallback.

```
/////////////// buttonCallback ///////////////

b->panel()
        /*
        Returns the ID of the panel - your first panel will be
        1 second 2 etc.
        */

b->id()
       /*
       The ID of the button that the notification is coming from
       – remember it might be a name rather than a number
       */

b->value()
       /*
       The ID of the button that the notification is coming from
       – remember it might be a name rather than a number
       */
```

revertiveCallback.

```
/////////////// revertiveCallback ///////////////

r->device()
       /*
       Holds the device driver number the revertive is coming from
       */

r->index()
       /*
       Holds the index number the revertive relates to
       - for a router this will be the destination number,
       - for a infodriver or gpidriver it will be the slot number
       */

r->info()
       /*
       Returns the related source information
       – for a router driver this will be the source index
       - for a gpi it will be the gpi state (0 or 1).
       Doesn't make sense for infodrivers so will always contain 0
       */

r->sInfo()
       /*
       Returns the source name for a router driver,
       the slot content for an infodriver but for gpidrivers this will
       always just be 0
       */
```

parentCallback.

```
/////////////// parentCallback //////////////

p->command()
       /*
       The parent command that issued this notification
       – this comes from the connections engine and is the left-hand side
 of
       the equal-sign in the command drop-down box
       */

p->value()
       /*
       The value the parent command attached to the notification
       – in the connections engine this is the right-hand side of the
       equal-sign in the command drop-down box
       */
```

timerCallback. Doesn't have notifications in quite the same form as the others. The only significant parameter of a timer callback is the id of the timer which is just an integer. The two methods are:

```
timerStart(int timer,int duration); //parameters are timer id and
                                    //milliseconds btween 'ticks'

timerStop(int timer);               //'timer' is the id of the
                                    //timer to stop
```

Other misc bits of useful code.

```
hostNotify("someParameter=someValue");❶

textPut("text","the actual words",[name of panel],
       [name of component]); ❷

textPut("colour.led","blue",[name of panel],
       [name of component]);❸

debug("Code has executed line 322 \n"); ❹//useful for checking
                                        //your logic note.
                                        // /n is new line

debug(myIndex + "\n"); ❺ //note that variables do not go inside
                         //quotes, strings do! \n is new line
```

❶ This is how your scripted component calls out to the panel it is hosted on. These are the events you can use in the connections engine and 'someValue' is what you can pass back to the panel.

❷ The textPut command doesn't just get used to change text, you can use it to get to pretty much anything which you can alter about the component. In the line above, "text" means that it is the text component we're altering. We could instead have used the following:

❸ Sets parameters of the control, in this case set the colour of the led on the named component to blue.

❹ Using debug to print out information about where your code has reached. Proving things you think are happening are actually happening.

❺ String concatenation in the debug print allows debug information to contain values from your code. This is useful to make sure things you think should have changed have changed and to the right values.

B. The Boiler Plate code created by the BNCS wizard

ExampleB.1: Boiler plate [yourScriptName].h. In this case testScript.h

```
 1  #ifndef testScripted_INCLUDED
 2  #define testScripted_INCLUDED
 3
 4  #include <bncs_script_helper.h>
 5
 6  #ifdef WIN32
 7  #ifdef DOEXPORT_SCRIPT
 8  #define EXPORT_SCRIPT __declspec(dllexport)
 9  #else
10  #define EXPORT_SCRIPT
11  #endif
12  #else
13  #define EXPORT_SCRIPT
14  #endif
15
16  class testScripted : public bncs_script_helper
17  {
18   public:testScripted( bncs_client_callback *parent,
       const char* path );
19
20   virtual ~testScripted();
21
22   void buttonCallback( buttonNotify *b );
23   int revertiveCallback( revertiveNotify * r );
24   void databaseCallback( revertiveNotify * r );
25   bncs_string parentCallback( parentNotify *p );
26   void timerCallback( int );
27
28   private:
29   bncs_string m_myParam;
30   bncs_string m_instance;
31
32  };
33  #endif // testScripted_INCLUDED
```

ExampleB.2: Boiler plate [yourScriptName].cpp. In this case testScript.cpp

```
 1 #include <stdio.h>
 2 #include <bncs_string.h>
 3 #include <bncs_config.h>
 4 #include "testScripted.h"
 5
 6 #define PNL_MAIN 1
 7
 8 #define TIMER_SETUP 1
 9
10 // this nasty little macro to make our class visible to the
      outside world
11 EXPORT_BNCS_SCRIPT(testScripted)
12
13
14
```

```
15 // constructor - equivalent to ApplCore STARTUP
16 testScripted::testScripted(bncs_client_callback *parent,
                   const char *path):bncs_script_helper(parent, path)
17 {
18
19  // show a panel from file p1.bncs_ui and we'll know it as
       our panel PNL_MAIN
20  panelShow(PNL_MAIN, "p1.bncs_ui");
21
22  // you may need this call to set the size of this component
23  //  if it's used in a popup window
24  // setSize( 1024,668 );  // set the size explicitly
25  // setSize( PNL_MAIN );  // set the size to the same as the
       specified panel
26 }
27
28 // destructor - equivalent to ApplCore CLOSEDOWN
29 testScripted::~testScripted()
30  {
31  }
32
33 // all button pushes and notifications come here
34 void testScripted::buttonCallback(buttonNotify *b)
35  {
36   if (b->panel() == PNL_MAIN)
37   {
38    switch (b->id())
39    {
40     case 1:
41      textPut("text", "you pressed|control 1", PNL_MAIN, 4);
42      break;
43     case 2:
44      textPut("text", "you pressed|control 2", PNL_MAIN, 4);
45      break;
46     case 3:
47      textPut("text", "you pressed|control 3", PNL_MAIN, 4);
48      break;
49    }
50  }
51 }
52
53 // all revertives come here
54 int testScripted::revertiveCallback(revertiveNotify *r)
55  {
56   /* switch( r->device() )
57   {
58    case 123:
59     textPut( "text", r->sInfo(), PNL_MAIN, 3 );
60     break;
61   }
62   */
63  return 0;
64 }
65
66
67
```

```
 68 // all database name changes come back here
 69 void testScripted::databaseCallback(revertiveNotify *r)
 70  {
 71  }
 72
 73 // all parent notifications come here i.e.
      when this script is just one
 74 // component of another dialog then our host might want to tell
      us things
 75 bncs_string testScripted::parentCallback(parentNotify *p)
 76   {
 77    if (p->command() == "return")
 78     {
 79      if (p->value() == "all")
 80       { // Persisting values for bncs_vis_ed
 81         bncs_stringlist sl;
 82
 83         sl << bncs_string("myParam=%1").arg(m_myParam);
 84
 85         return sl.toString('\n');
 86       }
 87
 88      else if (p->value() == "myParam")
 89        { // Specific value being asked for by a textGet
 90          return (bncs_string("%1=%2").
                 arg(p->value()).arg(m_myParam));
 91        }
 92     }
 93     else if (p->command() == "instance" && p->value() != m_instance)
 94      { // Our instance is being set/changed
 95         m_instance = p->value();
 96         // Do something instance-change related here
 97      }
 98
 99      else if (p->command() == "myParam")
100       { // Persisted value or 'Command' being set here
101         m_myParam = p->value();
102       }
103
104      // ***** CONNECTIONS EVENTS HELPER LIST *****
105      else if (p->command() == "_events")
106        { // Helper-list of everything in this component generated by
               hostNotify's
107          bncs_stringlist sl;
108          sl << "notify=*";
109          return sl.toString('\n');
110        }
111
112      // ***** CONNECTIONS COMMANDS HELPER LIST *****
113      else if (p->command() == "_commands")
114       { // Helper-list of any commands/parameters you might want to
               set at run-time
115         bncs_stringlist sl
116         sl << "myParam=[value]";
117         return sl.toString('\n');
118       }
```

```
119      return "";
120 }
121
122 // timer events come here
123 void testScripted::timerCallback(int id)
124  {
125   switch (id)
126    {
127     case TIMER_SETUP:
128       timerStop(id);
129       break;
130    default: // Unhandled timer event
131       timerStop(id);
132       break;
133    }
134 }
135
136  ////////////////////////////////////////////////////////////////
137  ///////////////// Callbacks above - Methods below ///////////////
138  ////////////////////////////////////////////////////////////////
139
```