C++ For BNCS

Abstract

A quick overview of C++ with emphasis on how it is used when writing BNCS scripted panels. An overview of writing BNCS scripted panels.

Table of Contents

1. C++ Introduction 1
2. Object Orientation (OOP) 2
3. C++ and BNCS 3
4. parentCallback: A bit more detail 7
5. Scripts and the GUI editor 11
A. The Boiler Plate code created by the BNCS wizard 13

1. C++ Introduction

BNCS 4.xx uses C++ as a coding environment for its scripted panels. C++ scripts can be used to provide all the panel functionality by interacting with buttons and sending messages to devices but are best used to provide plug-in reusable components joined together via the connections engine.

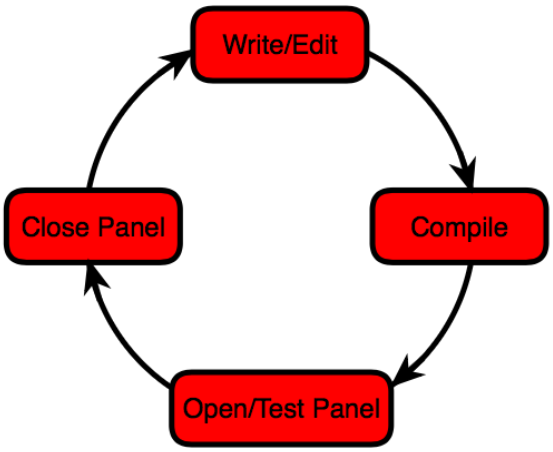
C++ is a very popular programming language used for writing operating systems and applications/libraries on Linux and Windows but also many other platforms. It is a development of the language C which is still widely used especially in Linux and unix systems. You'd be right in thinking that there was also an 'A' and a 'B' language prior to C.

The C language was written to be terse and close to low level assembly language. It is fast and flexible and so well suited to applications programing. C++ is an increment on C, it takes the basics of C and incorporates aspects of Object Orientated Programming (OOP) which allows for more robust code and reusable code. OOP is particularly suited to writing dynamic link libraries that provide common functionality to applications. A modern operating system provides many DLLs to give applications access to functionality such as video decoding. The best way to think of OOP is that objects 'live' and they interact by sending messages. This means they are well suited to an programme environment where you choose objects and interconnect them.

When writing a scripted panel, you will be using OOP features of C++ but you do not need to write OOP code or create classes and objects; you can treat the code as a simple 'script' albeit using CPP syntax.

A significant point in using C and C++ is that these languages are compiled; the code has to be turned into machine instructions and, in the case of BNCS, it is held in a dynamic link library where it can be used by panels. This is unusual for a 'script' which are usually interpreted. Later versions of BNCS may, or are, moving to a scripted environment and that looks likely to Javascript (possibly ECMAscript which is almost identical).

The development cycle is that of all compiled code; write->compile->test->edit etc. Because the DLL is used by the panel when you are testing it, you will need to close the panel before you can (re)compile the code.



> The development cycle is therefore; write->compile->open panel->test->close panel->edit.

Figure1: Scripted Panel Development Cycle

Note

When a scripted panel is 'run' the DLL is held 'open' by either panel manager or the GUI editor. This allows data persistence between callbacks but it does mean that the DLL can not be compiled if it is open in either.

This cycle of write->compile->test is the basic cycle of all compiled programming languages including those that run on a virtual machine like Java. The process of compiling is actually a multistage process consisting of (at least) compiling and linking.

Compiled languages are the ones often used to write desktop applications so it is a very common experience. An Integrated Development Environment (IDE) is by the best way to approach developing compiled languages. An IDE will handle all library paths, build directives, syntax helpers and debugging. Actually a good IDE is useful for all programming whether scripted or compiled.

An important point in programming is that it is rarely 'from scratch' you will be using libraries of functions or objects and an IDE will help by providing support for finding the right functions and even browsing documentation if it is implemented. So be prepared to do a lot of looking up and finding functions that do what you want and how to use them.

2. Object Orientation (OOP)

There are a number of paradigms for programming, declarative (SQL), functional (lisp), procedural (C and Pascal) and Object Orientation. There is a whole science that covers all of these and a great deal of, shall we just say, debate over which is best. Suffice it to say that all programming is about doing things to data there really isn't much difference

if you look at it that way. The only one that sticks out as being significantly different and may not look like a programming language at all is the declarative paradigm.

C++ is Object Orientated or, to be more accurate, C++ has object orientated features. There is, shall we just say, a debate over whether it really counts as a true object orientated language but that is largely irrelevant. It does mean though that you will be using a syntax that is Object Orientated.

The key to OOP is message passing. Objects live and send and receive messages between each other. These messages cause things to happen. So you can send a message to an object that, perhaps, makes it send a message to another object that sends a message via a serial port but they key is messages and this makes it particularly suitable for plugging modules together.

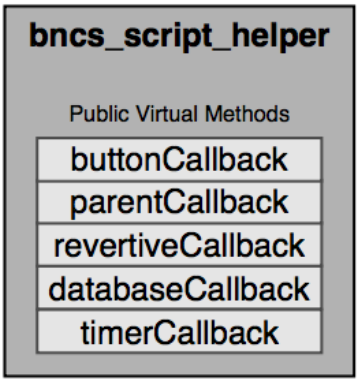
2.1. Concepts of OOP

OOP is about message passing between objects so we need to define an object and the messages. The popularity of OOP is the we do not need to code everything, the code already exists in an object and we can just use it as required adding our own functionality to the object.

An Object is a complete self contained bit of programme, it contains the data and the code that operates on that data. This code may be visible and so callable from outside: these are the messages. By outside we mean the rest of the programme code. The data is not (generally agreed should be never be) accessible directly from outside. The object is self contained so once proved and tested it should be a reliable component to use.

3. C++ and BNCS

The script for a panel is based upon extending the class *bncs_script_helper*. bncs_script_helper defines five virtual (i.e have no code by default) public methods; the panel script has to provide the implementation of these and these are provide the interface between BNCS and the panel script; the BNCS API if you like.



> The BNCS environment provides a class called bncs_script_helper. It outlines all the interfaces to BNCS from and to the script. The virtual methods need to be implemented in the panel script (there are none provided) and provide asynchronous calls *to* the script *from* BNCS.

Figure2: bncs_script_helper virtual methods

bncs_script_helper also provides methods for communicating from the panel script to BNCS.

When the panel manager opens the script, it instantiates (OOP term) the panel script class and keeps a reference to it, from then on it can interact with the panel script. The panel script interacts with BNCS using other methods of bncs_script_helper . The same code can be used in multiple panels because a new object is instantiated for each. (Instantiation is a process where code for a class is loaded into memory ready to use with a pointers to where the data is stored for that object.)

Note

In all modern programming languages you are interacting with library objects. It is not possible to deduce these from first principles; you need a good reference to the object definitions. In the case of BNCS library this is provided in the help file 'Documentation.chm'.

Example1: virtual methods in bncs_script_helper

```
// callbacks

virtual void buttonCallback( buttonNotify *b );

virtual int revertiveCallback( revertiveNotify *r );

virtual bncs_string parentCallback( parentNotify *p );

virtual void timerCallback(int id );

virtual void databaseCallback( revertiveNotify *r );

// Useful but not core

virtual void httpCallback( httpNotify *p );
```

The first 3 of these functions are the most important and are the most used. These are the methods that need to be fleshed out to make the script work. The 4th, timerCallback, is useful but its use is not essential. The 5th, databaseCallback has a specific use for databasses likewise for httpCallback but this is likely to be used even less often.

buttonCallback. As you would expect, this gets called whenever any control on your panel generates a 'notify'. What gets notified from each control is configurable from its properties. It passes a buttonnotify object to your script that you interrogate to determine which control generated the callback.

parentCallback. This gets called when the panel manager sends you a command. These commands will be initiated from the connections engine and during panel start up to set parameters. It passes a parentnotify object that contains information about what has triggered the callback.

revertiveCallback. This is called for all revertives that your script has registered. To determine what has caused the revertive, the 'revertivenotify' object given has to be inspected for device ID.

databaseCallback. This is called when a database change event is generated. It passes a revertivenotify object that can be used to determine what triggered the callback.

Note

As usual for C++ (and C) e.g *p means a "pointer to" an object of that type. This means our code will have to use the arrow syntax e.g "p->". You will also need to have the documentation for e.g. revertiveNotify.

The partner to parentCallback is hostNotify. Using this we send a message as a bncs_string to the panel manager this is called an "event" and the message can be forwarded to another control using the connections engine.

Example2: bncs_script_helper hostNotify

```
// hostNotify generates an event that can be passed to
// another control via the connections engine
// it's called a notification in the GUI builder

void hostNotfy(const bncs_string &message);
```

The Class bncs_script_helper also provides all (a lot) of real methods used to interact with the panel and to make requests on the bncs system.

Example3: A selection of other methods in bncs_script_helper

```
// put text into a control

void textPut(const bncs_string &what,
             const bncs_string &value,
             const bncs_string &pnl,
             const bncs_string &id);

// get text from a control

void textGet(const bncs_string &what,
             const bncs_string &pnl,
             const bncs_string &id,
             const bncs_string &ret);


// disable a control

void controlDisable(const bncs_string &pnl,
                    const bncs_string &id);
```

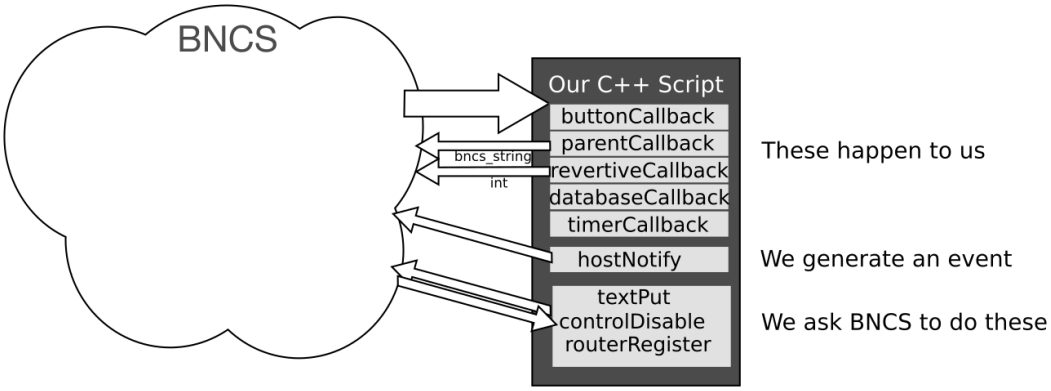
```
// register for revertives from a router
// this will call our revertiveCallback when a revertive is generated

void routerRegister(int dev,
                    int start,
                    int end,
                    bool add=false);

// poll a router (won't see anyting unless registered)

void router Poll(int dev, int start, int end);


// register for revertives from an info driver
// this will call our revertiveCallback when a revertive happens

void infoRegister(int dev);


// poll an info driver

void infoPoll(int dev, int start, int end);
```

The class bncs_string is the main way of carrying information to and from the world of bncs to your script. In C++ strings are always arrays of some form so have to passed as pointers so all the bncs_string values above use the syntax "&value" meaning the "address of value".

Methods that return anything other than a basic data type e.g. int, will return data as a pointer.

Note

There are dozens of this methods so you really do need access to the bncs_script_helper and bncs_string documentation to do anything.



All communication to and from our script is via the bncs_script_helper class. Messages are sent to our script using the callbacks, the script can generate an event for the connections engine using hostNotify and the script can interact with the panel and BNCS as a whole using methods of bncs_script_helper.

Figure3: Summary of messages for bncs script

4. parentCallback: A bit more detail

Parent callback is the main interaction from the BNCS 'world' to the script. Here we will handle the "Commands" sent to us and allow us to list the commands we expect. We can also list the "Events" we expect to send using hostNotify. Both of these come through the connections engine and, importantly, neither of these are binding. You can type whatever you like in the connections commands or events.

We can also get and set parameters. These parameters are stored with the GUI file and given to our script when it is activated on a panel.

Using commands, events and parameters should give our script all it needs to work with. If the script needs more detailed information, such as a list of names, numbers, there are mechanisms for storing XML files provided by BNCS (and C++ provides everything you need to access, files, websites, databases etc.) .

Note

None of the following is mandatory; there are no checks carried out by BNCS. We can respond to Commands and send Events we don't list. We can list Commands and Events we don't respond to or send. Good practice is important.

4.1. Commands - receive from BNCS land

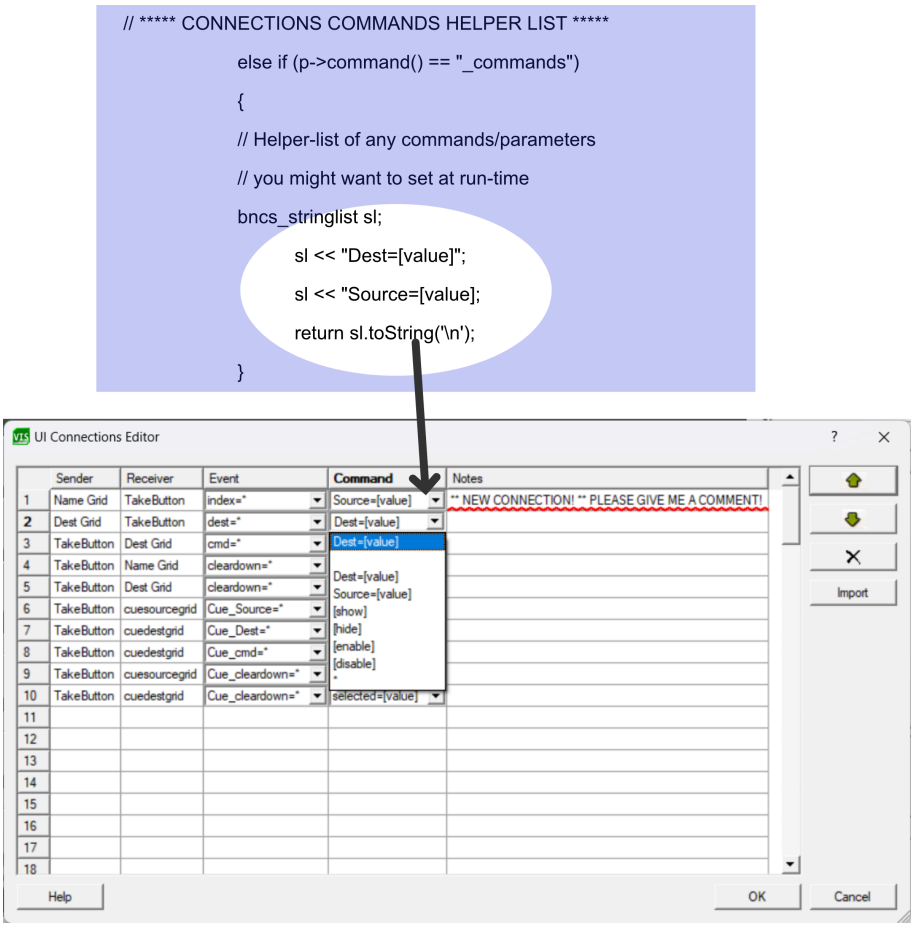


Figure4: Listing our Commands

The commands we receive are handled in individual statements. It is the same code pattern as the setting parameters.

Example4: Handling commands

```
else if (p->command() == "Dest")
   { // Persisted value or 'Command' being set here
     handleOurDestCommand(p->value());
   }
else if (p->command() == "Source")
   { // Persisted value or 'Command' being set here
     handleOurSourceCommand(p->value());
   }
```

4.2. Events - send to BNCS land

Events are things that we send to BNCS land using hostNotify. We can list them so they can be used in the connection engine drop down or just send them and expect the event is manually typed into the connection engine.

```
// ***** CONNECTIONS EVENTS HELPER LIST *****
        else if (p->command() == "_events")
        {
        // Helper-list of everything in this component
        // generated by hostNotify's
                bncs_stringlist sl;
                sl << "notify=*";
                sl<<"Cue_Dest=*";
                sl<<"Cue_Source=*";
                sl<<"Cue_Cleardown=*";
                sl<<"Cue_cmd=*";
                return sl.toString('\n');
        }
```

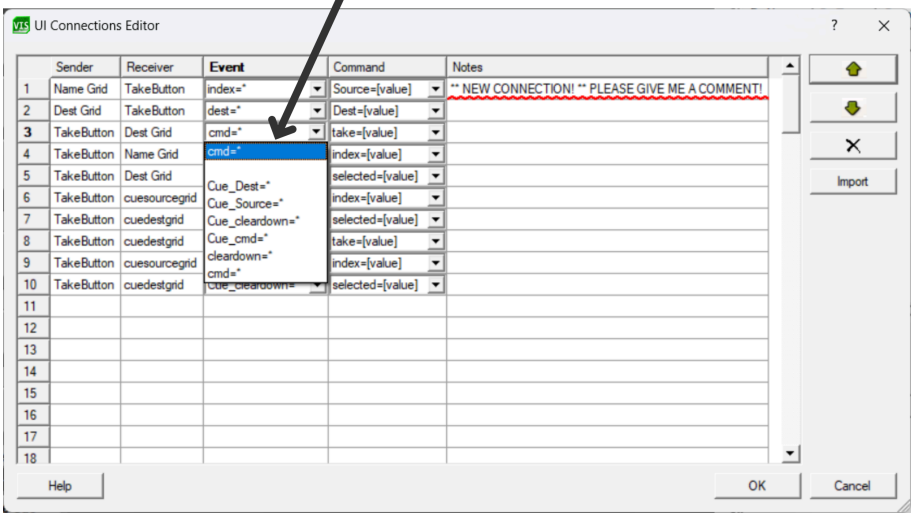
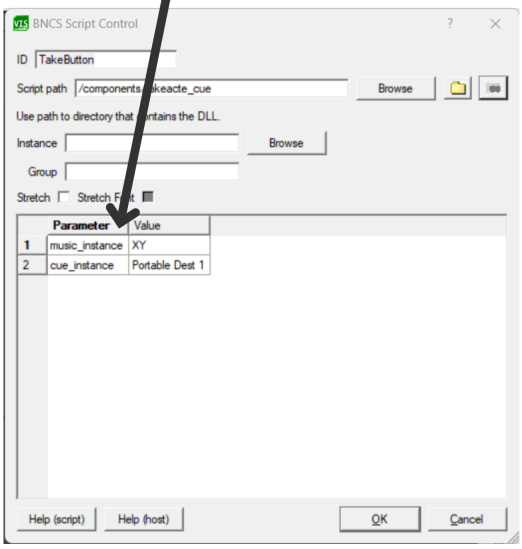


Figure5: Listing the Events we expect to send with hostNotify

4.3. Parameters - values that are handed to us at start up by BNCS land

Parameters are values that are stored in the BNCS GUI file and are handed back to our script when it is activated on a panel. These are often termed persisted values as each time our script starts we get them back.

```
if (p->command() == "return")
        {
        if (p->value() == "all")
                { // Persisting values for bncs_vis_ed
                        bncs_stringlist sl;
                        sl << bncs_string("cue_instance=%1").arg(cue_instance);
                        sl << bncs_string("music_instance=%1").arg(music_instance);
                        return sl.toString('\n');
                }
```



The table is filled in by requesting the values from the script when the table is drawn. At script start-up, BNCS sends stored values to the script so the table should always reflect the stored values in the GUI file.

Figure6: Startup parameters we expect to receive.

As for commands, there is provision for getting and setting individual parameters, this is used when we update the parameter from the GUI.

Example5: Getting (returning) individual parameters

```
else if (p->value() == "cue_instance")
   { // Specific value being asked for by a textGet
      return (bncs_string("%1=%2").arg(p->value()).arg(cue_instance));
   }
else if (p->value() == "music_instance")
   { // Specific value being asked for by a textGet
      return (bncs_string("%1=%2").arg(p->value()).arg(music_instance));
   }
```

Example6: Setting individual parameters

```
// We may need to run some code if any of these are set
else if (p->command() == "cue_instance")
   { // Persisted value or 'Command' being set here
     cue_instance = p->value();
     // runSomeCode();
   }
else if (p->command() == "music_instance")
   { // Persisted value or 'Command' being set here
     music_instance = p->value();
     // runSomeCode();
   }
```

5. Scripts and the GUI editor

Scripted panels can be run in the same way as any other panel using the GUI editor or via the panel manager. In addition, if you include a scripted component on another panel, other aspects of your script are exposed in the GUI editor that support the connections engine and setting basic parameters.

There is no validation between the GUI and the script. You can change values in the GUI (e.g component labels) and the script will not 'know' and vice versa. This means that changing your mind about names can be troublesome so a more structured plan and maybe a labelled sketch for the GUI can be useful.

The class *bncs_script_helper* also provides a number of methods that communicate to BNCS as a whole allowing the script to access the panel components to write text, etc.

Panels are probably best written making use of instances and the connection engine so a good approach for writing scripted panels is to write reusable components rather than whole panels. This approach allows best use of the OOP nature by building and thoroughly testing the components before they are used it should make the final overall panel more robust and modular.

As with almost all software it is good practise to avoid hard coding any values into the script. In BNCS, the gui editor can proved the script with basic configuration that is saved in the panel file. If more configuration is needed then use a file that is identified by configuration from the GUI editor.

Note

There is no automatic code linking from the GUI to the script so you have to manually ensure that they join up. If you decide to change a button name to make what it does clearer, then you will have to reflect this change your the code. Write your code so you only have to do this in one place if at all possible. You can use #defines or constants to help. If the panel is complicated, maybe an additional [myGui].h file to store them all.

General programming tips. Here are a few bits picked up over the years programming in various languages. The first thing is to realise that there is a lot of

hidden configuration in a project and the IDE takes care of that. You need a really good reason to choose to manually configure a different IDE other than the default.

1. Programming is totally artificial you can't be expected to know what the person or team who invented it meant. Look for documentation and don't worry if you have to do a lot of searching. If you can't see how to do something it's probably because it's not documented well enough.

2. Use the tools provided in the IDE (integrated development environment): the syntax highlighting, pop up suggestions, red underlines, bracket matching. The IDE often has access to documentation about the code and will be able to analyse headers so it's like having the book open.

3. Use the IDE 'refactor' tools if you want/need to change any variable, function or class names in your code. This tool is a code-aware search and replace that 'knows' about variables and functions so will only change those that make sense.

4. Read any error messages they are always correct but be aware that, especially for syntax errors, the actual error may be a result of an problem earlier in the code.

5. Sometimes what you are provided with doesn't do what you need if it doesn't look for alternatives ways. If BNCS doesn't provide the data you need by default, is there a way to make it give you that data.

6. Use or make up a convention and stick to it. If you abbreviate 'source' to 'src' make it an instinct. Likewise for capitalisation, the general rules are: constants are all caps, variables are camel case, classes start with a capital and are camel case.

A. The Boiler Plate code created by the BNCS wizard

ExampleA.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
```

ExampleA.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
```