Panel Building

1 Introduction

This document describes the whole process of building panels

The tutorials are all available as part of the developer install and many of the tutorials build on the previous one.

Each tutorial explains just one or two features.

2 Prerequisites

- The standard 3 day MR BNCS course. You need to know what in "Infodriver" is, how messages are passed on the network, what drivers are, how to run things etc.
- Visual Studio 6 or Visual Studio 2013 (C++). Note that the tutorial panels and this document were prepared using VS6, so that is shown in all screen-shots. There are certain other differences when using VS 2013. There is some information on this subject in Appendix C.
- Reasonable knowledge of C programming, some understanding of C++

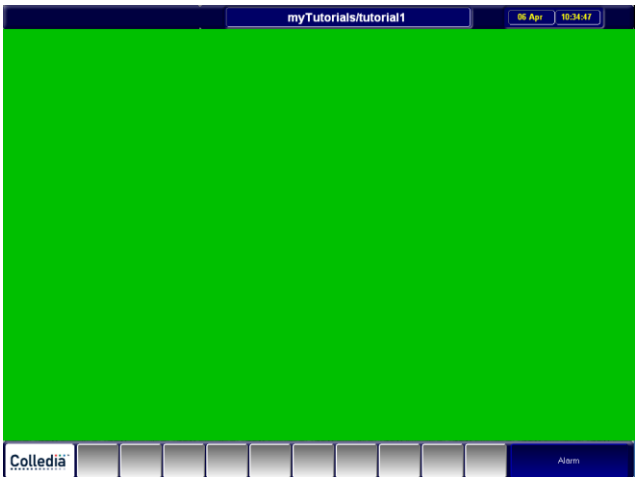
3 Some Fundamentals

3.1 Hosting in Panel manager

Panels are all hosted within a single executable application – Panel Manager (panelman)

Panel manager itself is configurable, so that you can change its layout and appearance. However you are not free to write your own.

Panel manger provides a "home" for all the panel applications and also provides the means of navigating between panels.



The green area above shows the area occupied by the panels – the rest is configurable and maintained by panel manager (for those viewing in black and white it's the plain area below the title and time/date and above the row of buttons at the base).

The elements of panelmanager shown above – the title, the buttons at the bottom are all separate UI resources that you can customise as you see fit.

3.2 Different types of Panel

There are two kinds of panel that panel manager can display:

- A simple dialog description (just a text format representation of the controls you want to show)
- A scripted panel (C++ compiled into a DLL)

The simple dialog description is just that – just the layout of the buttons. This can be used with smart buttons which are most likely used as overview panels used to navigate to other panels, or simple process control panels. There's a simple means of wiring UI components together without code (known as "connections")

A scripted panel means one that is driven from a C++ derived script.

3.3 The general structure of a scripted panel

Each scripted panel may be considered to be its own, self-contained 'ApplCore' panel (if you've come from an earlier BNCS system). They work completely independently of any other panel in the same panel manager.

3.4 Just another button

Every component that has a visual representation is a 'Widget' in Qt speak[1]. This defines a few fundamental things about the component such as its size and position.

Buttons, labels, faders, group boxes and listboxes are all 'Widgets'. So however is the C++ script control, but instead of having a fixed appearance like a button, what it looks like is determined by dialog description files.

3.5 DLL

'Scripts' are actually binary, compiled Dynamic Link Libraries (DLLs). This means that they need to be built. In reality this isn't much different to building EXE's from RC files except that the tool used is Visual Studio rather than Resource Workshop.

These scripts are fast as they are compiled, and have many built in features as they inherit functionality from the code they're derived from.

3.6 Directory per panel

Panels are made up of several components/elements (which in previous versions of BNCS would have been bound into one executable file).

- Dialog descriptions
- 'Scripts'
- Other resources (e.g. bitmaps)

All these files should live within their own separate directory. Take a look at the tutorials – they all work in this way.

All files for a project are kept together in a single folder (it's what directories are for after all) and not scattered across a single directory.

[1] Qt is a cross-platform C++ programming toolkit/framework on which BNCS V4.5 is based

3.7 Parameter passing

There is a very simple parameter passing mechanism used between all components. It's the same mechanism whether you're setting parameters on a component or it's telling you something.

The general format of messages is:

```
parameter=value
```

All messages are the same basic text format.

So to set the colour of the text of a control you'd use something like:

```
colour.text=red
```

The `parameter` bit "colour" has been qualified by adding a sub-parameter "text". So to set the background colour you'd use something like:

```
colour.background=blue
```

The same format is used for a component to tell you something:

```
button=pressed
button=released
```

For a control that hosts multiple buttons you might see something like this:

```
button.1=released
```

where the "`1`" qualifies the parameter.

In rare cases you might see something like:

```
colour.sub1.face1.front.text=green
```

where it takes several sub-parameters to fully qualify the thing you're trying to set.

This means of passing parameters is very forgiving – if you make a mistake and send the wrong thing it won't throw a bizarre run-time message telling you "Can't find entry point colour.toxt" 'cause of a silly typo….it won't however do much interesting either.

It also means that interfaces aren't self-describing so you can't (easily) discover how an interface works or is implemented

4 Installation

Run the installer!

This is called BNCS_V4_5_install.exe.

The standard installer with default options will install all files necessary to develop panels. To get the Demo system with tutorial panels already included, select 'Demo system & tutorials'.

For these tutorials it's assumed that you do your install to:

```
c:\bncs\v4.5
```

All the references in this document are to this location. Of course for real projects the location may be different.

No further installation or configuration is required.

Note: There have been several instances of problems with the install process, not with the installer itself but with multiple versions of files on the same machine. Please remove all versions of BNCS files from common directories such as `c:\winnt` or `c:\Windows`.

5 Sample Tutorials

Each tutorial is included in the 'examples' sub-folder of the 'panels' directory.

6 Getting Help

Throughout the tutorials there are references to the help files for particular classes or components.

All help can be found in the 'docs' directory (e.g. c:\bncs\v4.5\docs) and are all referenced from the home page 'index.htm'.

See 'Appendix B – References' for information on useful documentation

7 Tutorial Progression

The idea is to start the tutorials with familiar concepts – panel building the traditional way.

The way the tutorials work is then to introduce the more modular approach using smart buttons and components.

The end point is to write panels using as many components as possible – hopefully someone else's pre-written components, but otherwise using components of your own.

Getting started – Tutorial 1

7.1 Creating your first panel

We'll create our first panel in a new subdirectory off the 'panels' directory. We'll call this myTutorials – you must create this directory by hand. You should end up with `c:\bncs\v4.5\panels\myTutorials`

There is a code Wizard which produces code from a template. The code it produces is an entirely functional, but dull, panel.

- Run the Code Wizard from the Start Menu->BNCS-v4.5 program group – or run direct from `c:\bncs\v4.5\windows\bin\wiz\wiz.exe`
- Select the 'Create C++ Scripted UI/Panel' option
- In 'Location' navigate to your 'myTutorials' directory. This is the path you've just created and it must already exist.
- Enter 'tutorial1' as the Project Name.
- Hit OK and enter a suitable title (such as 'Getting Started – Tutorial 1').
- Hit OK to the offer of opening this project in explorer.
- Double click the '.dsw' file (or .sln file if you're using VS2013)
- Build the project (Select menu Build->Build tutorial1, or hit F7)

7.2 Making our new panel appear

We can't run our new panel directly – it's a DLL so we must use panel manager.

From the Project->Settings menu in Visual Studio select the Debug tab. Enter the path to panel manager and arguments as shown in **Figure 1 - Project Settings for a Typical Panel**

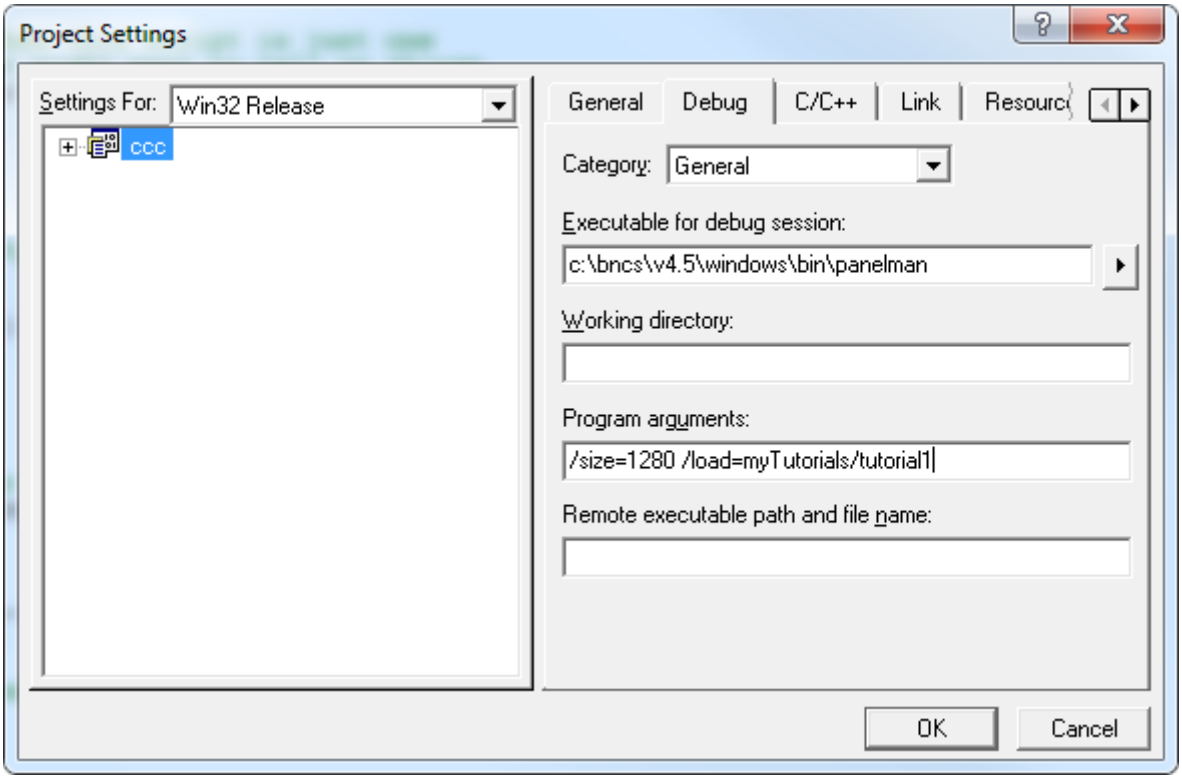
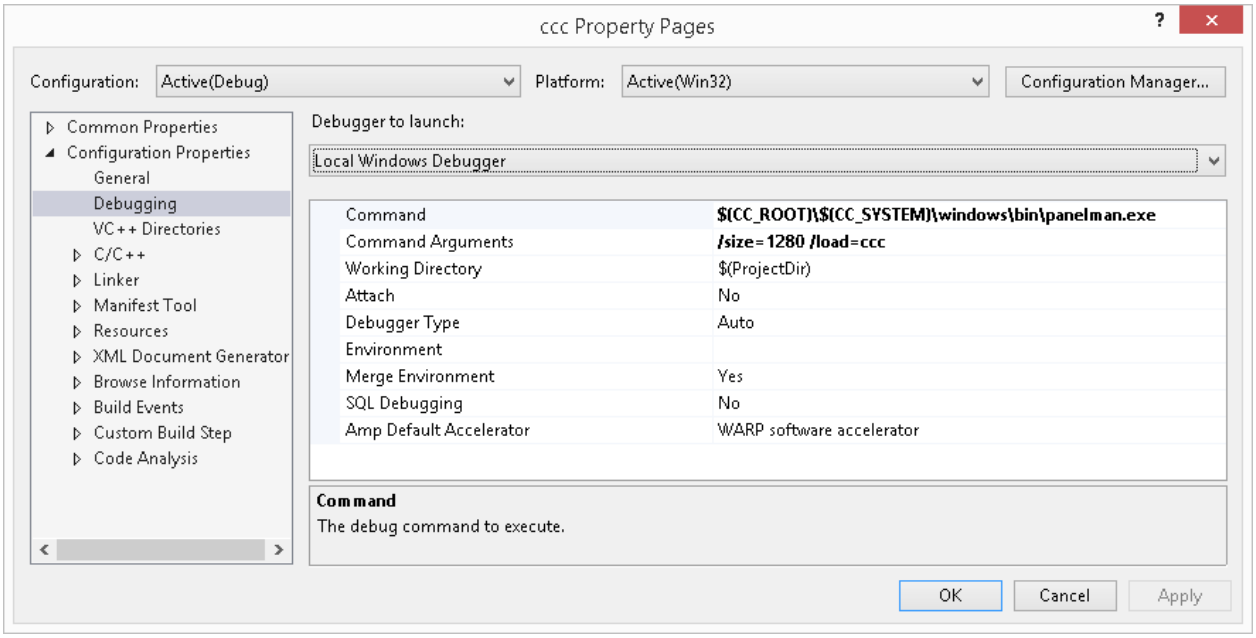


Figure 1 - Project Settings for a Typical Panel

And here's the same thing for VS2013 (for a panel project called "ccc"):



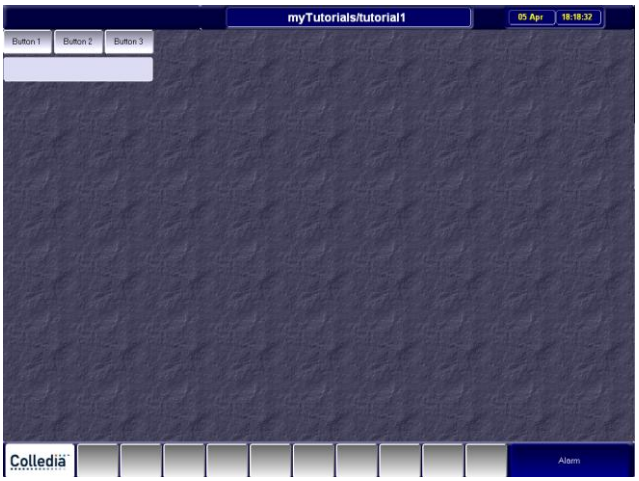
The '/load=' option means that panel manager does not require any further configuration to make this panel appear. '/size=1024' isn't really required as panel manger defaults to this resolution.

Note: Don't forget to select Settings For: 'All Configurations'.

Sadly these settings can't be automatically included as part of the Wizard.

To make the panel appear hit 'Execute this application' (or hit Ctrl F5).

You should get something that looks like this:



This is the very simple project that the code wizard has generated for us. Hitting any of the 3 buttons will generate an appropriate message on the label.

This next bit requires "developer mode" enabled on your machine - search your V4.5 system for "bncs_inst_env.exe" and run that – checking the box that says "Developer Mode".

Developer mode means you can right click on the UI of a running panelmanager to be able to edit that UI (but only the next time you run the app).

Right clicking a running application to edit the UI's works for both the tutorial and panelman's own resource files. Try right-clicking any of the buttons top left, then try over the clock and the "Alarm" button.

7.2.1 Important note about environment variables

Developer mode, and other settings such as the "pointers" you your current V4.5 system are stored in environment variables (all the ones that start "CC_" for example "CC_DEVELOPER". These are set and usually fixed when you start an application running. So if you change any of them (such as developer mode) they won't take effect until you next start that application. The real gotcha is that this includes all running Windows explorer windows which pass on their (old) environment when starting new applications.

7.3 Looking at the User Interface description

One of the files created by the wizard is 'p1.bncs_ui'. This is an XML description of the layout of the buttons.

This file is `c:\bncs\v4.5\panels\myTutorials\p1.bncs_ui`

Double click this from a file explorer window or open the Visual Editor from the Start Menu->BNCS-v4.5 program group and use the File->Open on that application to navigate to that dialog description.

You should end up with a window that looks like **Figure 2 – Visual Editor showing Simple Panel.**

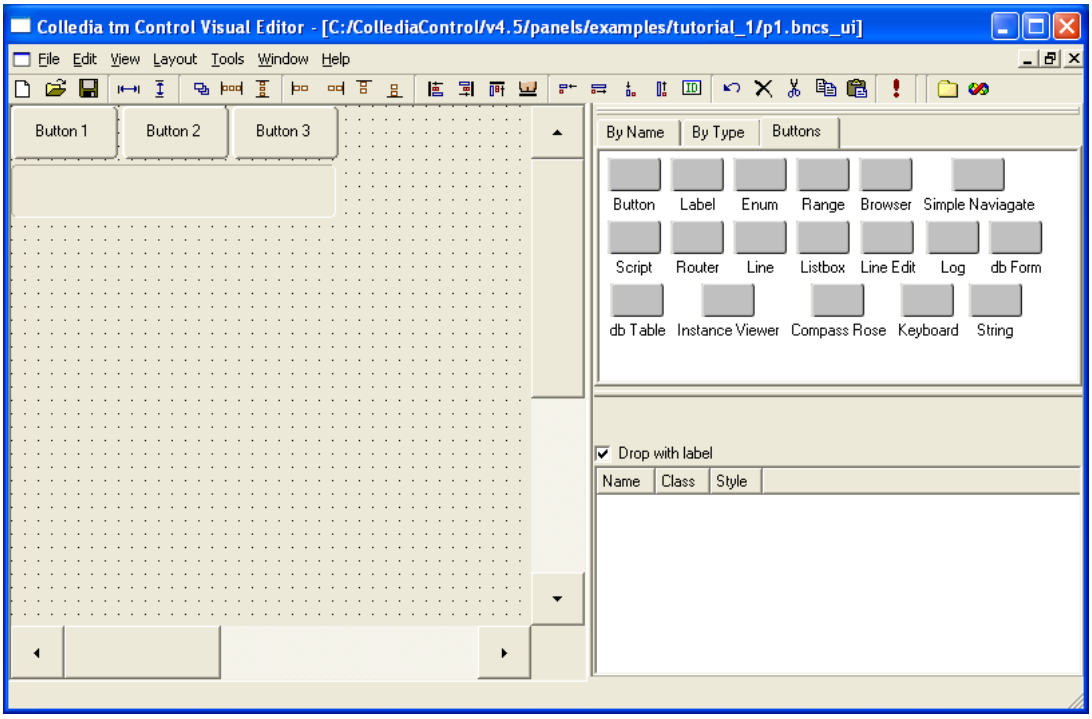


Figure 2 – Visual Editor showing Simple Panel

Try double clicking on any of the buttons at the top left of the display – you'll get the properties dialog for that control. Most of the settings on the control should be pretty obvious. Look at the label control below the buttons – check what it's ID is. We'll need this in just a moment.

We don't need to make any changes at this point. You can close this editor without saving.

7.4 A line by line step through of what this panel does

Let's take a look at how this panel works.

We'll start with a very quick look at the header file (a few irrelevant lines left out for clarity):

```
#include <bncs_script_helper.h>
class tutorial1 : public bncs_script_helper
{
public:
     tutorial1( bncs_client_callback * parent, const char* path );
     virtual ~tutorial1();

     void buttonCallback( buttonNotify *b );
     int revertiveCallback( revertiveNotify * r );
     void databaseCallback( revertiveNotify * r );
     bncs_string parentCallback( parentNotify *p );
     void timerCallback( int );
};
```

The only thing to note really about this header file really is the fact that the class is derived from a class called bncs_script_helper. That means our new class automatically has all the functions that bncs_script_helper has.

The functions defined in this class are where we're going to be told interesting things that have happened.

Look in the help pages for 'User Guides/Panel Writing/Classes' for full documentation of bncs_script_helper and the other the classes used here.

Let's now take a look at the cpp file (again with a few lines left out for clarity):

```
// this nasty little macro to make our class visible to the outside world
EXPORT_BNCS_SCRIPT( tutorial1 )
```

This line is a macro which is used just to make our source easy to read. It's defined in <bncs_script_helper.h> if you really want to have a look-see what it does. It's just used to create an instance of your class[2].

```
tutorial1::tutorial1( bncs_client_callback * parent, const char * path ):
    bncs_script_helper( parent, path )
{
    // show a panel from file p1.bncs_ui and we'll know it as our panel 1
    panelShow( 1, "p1.bncs_ui" );
}
```

This is the constructor for this class – this is where we do all the initialisation. All we're doing here is creating our "panel". The panelShow command has two parameters – the first is what we'll know this panel as, the second is what file to load.

This creates a panel based on our external panel file "p1.bncs_ui". This file was also created by the code wizard when we created this project.

Let's look at the code called when a button is pressed:

```
void tutorial1::buttonCallback( buttonNotify *b )
{
    if( b->panel() == 1 )
    {
            switch( b->id() )
            {
                    case 1: textPut( "text=you pressed|control 1", 1, 4 ); break;
                    case 2: textPut( "text=you pressed|control 2", 1, 4 ); break;
                    case 3: textPut( "text=you pressed|control 3", 1, 4 ); break;
            }
    }
}
```

[2] To be precise it is used to make all panel dlls provide an identically-named function call that panelman can use to create the panel when the dll file is loaded.

The buttonCallback function is called whenever one of our buttons wants to tell us something. We're passed a pointer to an object that already contains everything that we need to know about that button push in a form that we can use.

The reference for the buttonNotify class is linked from the bncs_script_helper class documentation so you should be able to fairly casually find the help information for it.

The parameter '`b`' has information about which button was pressed (its ID) and the panel that button sits on. It also has information on which parameters the button passed back to us – for the moment we just need to know that the button told us something.

First we check that the button push emanated from the right panel:

```
if( b->panel() == 1 )
```

Strictly speaking we don't need to do this here as we've only got one panel – but it's good practice.

Then we'll work out which button's been pressed:

```
switch( b->id() )
```

and then write some text back to the display:

```
case 1: textPut( "text=you pressed|control 1", 1, 4 ); break;
```

You'll note a slightly unusual format to the textPut command. You can't just send text to a control and have that text appear. There are many commands that the control will accept (colour, text, alignment etc.) so we need to tell it that we're passing it new "text". This command is in the format

```
parameter=value
```

mentioned earlier.

The text itself has a BNCS "newline" character in the form of a pipe.

This command writes new text to control 4 on panel 1.

None of the other sections are relevant at the moment and are just the default implementations that the wizard produces.

How do we know what commands we can send to a control and what responses it'll give us? Look in the reference for that control – these are listed in the Appendix.

The control we're dealing with here is a Generic Control (bncs_control)

7.5 Handy tip!

The auto-complete and helper features within Visual Studio are available when you have the right header files incorporated within the environment in the right place.

Having compiled your application once you get an 'External Dependencies' folder. Move the files shown in **Figure 3 - Moving the Dependencies into the Header section** into your 'Header Files' folder.

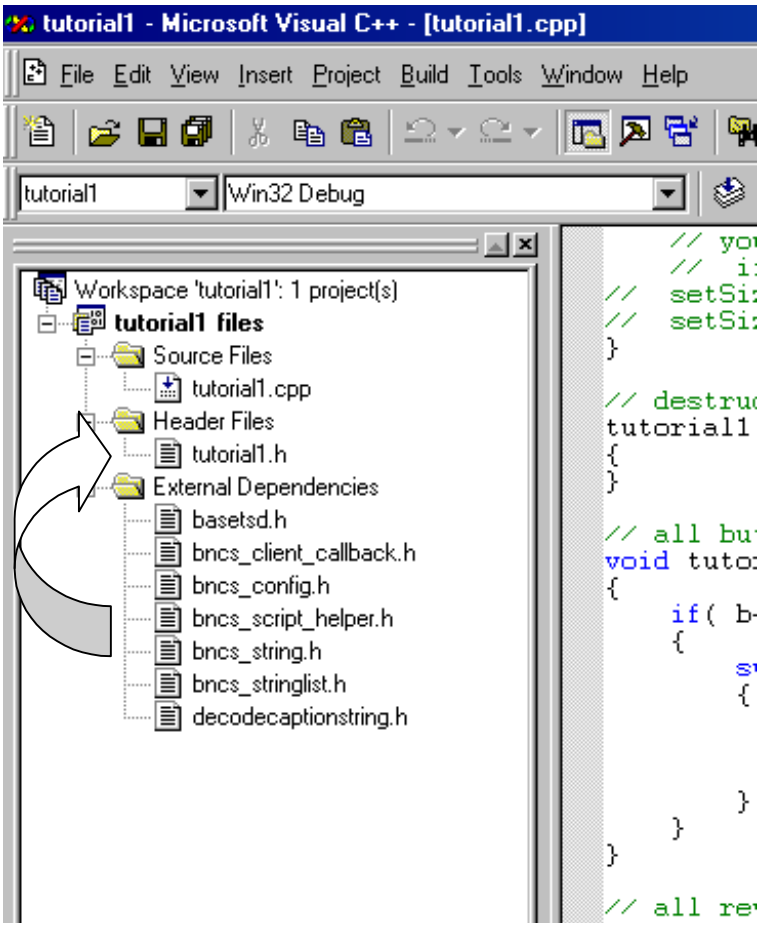


Figure 3 - Moving the Dependencies into the Header section

Note however that this is not a good thing to do if you habitually use the Class View within Visual Studio. If you do so then all the classes in these headers will also appear there, considerably cluttering up the display.

7.6 Debugging

Even though you're using an external program to run your DLL, there's nothing to stop you using the integrated Visual Studio Debugger to debug your DLL. Set a breakpoint somewhere appropriate in the buttonCallback function and 'run' your DLL using 'Debug Application' (F5). When you now hit a button on your panel the application stops on your breakpoint in Visual Studio.

Further information on how to use the Visual Studio Debugger is out of the scope of this document.

7.7 What to try next

Familiarise yourself with the Visual Editor. Try adding extra buttons; look at the properties for the buttons. Don't worry if you don't understand everything at this stage. Try adding a few buttons and use the tools to align them and adjust their size.

7.8 Getting Started – Tutorial 1 Summary

What you should have learnt from this tutorial:

- How to create a new user interface project

- How to compile the project
- How to make your new user interface appear within panel manager
- Where the user interface description file is to be found and how to open it for editing

8 Responding to button presses - Tutorial 2

What we'll do in this tutorial is extend slightly the functionality of Tutorial 1 to do a bit more with the button push information.

In addition to setting the text on the label, we'll also change its colour to red when a button is pressed, and to green when a button is released.

We need to edit the dialog description. Open the 'p1.bncs_ui' file and check the 'Notify' checkboxes for 'Press' and 'Release' on **all three buttons**.

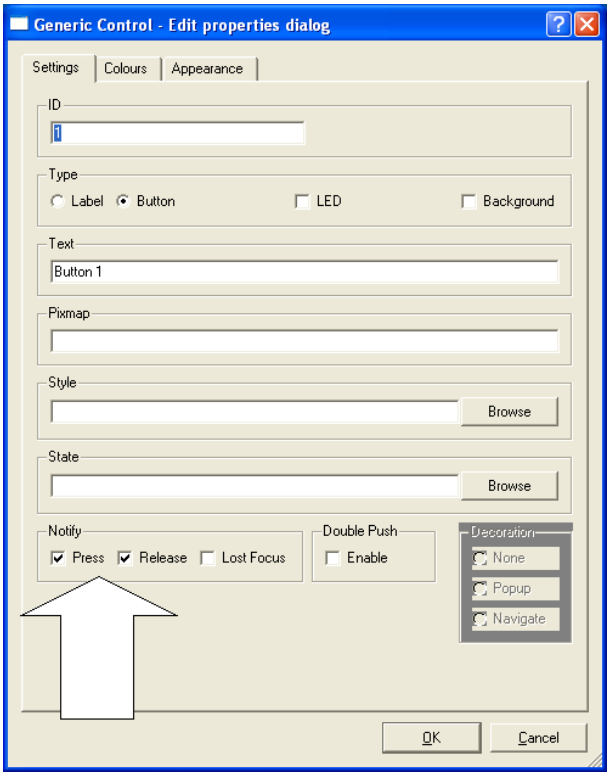


Figure 4 - Generic Control Dialog

Here's the new buttonNotify code – everything else in our project is the same.

The significant modified lines are highlighted.

```
void tutorial_2::buttonCallback( buttonNotify *b )
{
	if( b->panel() == 1 )
	{
		// work out the type of notification - pressed or released?
		if( b->value() == "pressed" )
		{
			textPut( "colour.background", "red", 1, 4 );
		}
		else if( b->value() == "released" )
		{
```

```
                                        // this is much the same as Tutorial 1 - just set the text
                                        // depending upon which button we pressed
                                        switch( b->id() )
                                        {
                                                    case 1: textPut( "text=control 1", 1, 4 ); break;
                                                    case 2: textPut( "text=control 2", 1, 4 ); break;
                                                    case 3: textPut( "text=control 3", 1, 4 ); break;
                                        }
                                        textPut( "colour.background", "green", 1, 4 );
                              }
                    }
          }
```

Remember the

```
parameter=value
```

parameter passing system?

We now get 2 notifications when a button is pressed and then released and these are:

```
button=pressed
button=released
```

In tutorial 1 we didn't need to determine what the notification was – the default for this type of button is a single 'released' notification. Now we've enabled the 'pressed' notification we need to distinguish between the two.

How do we know what notifications are sent? They are all listed out in the documentation for the controls, but can also be determined using the 'What's This?' help in the design time dialog.

Double click one of the controls at design time and then hit the question mark next to the close button at the top right of the dialog. The cursor will change to a question mark. Then go and click on the 'Notify->Release' checkbox. You should get something that looks like **Figure 5 - Tool Tip for Notify Release.**

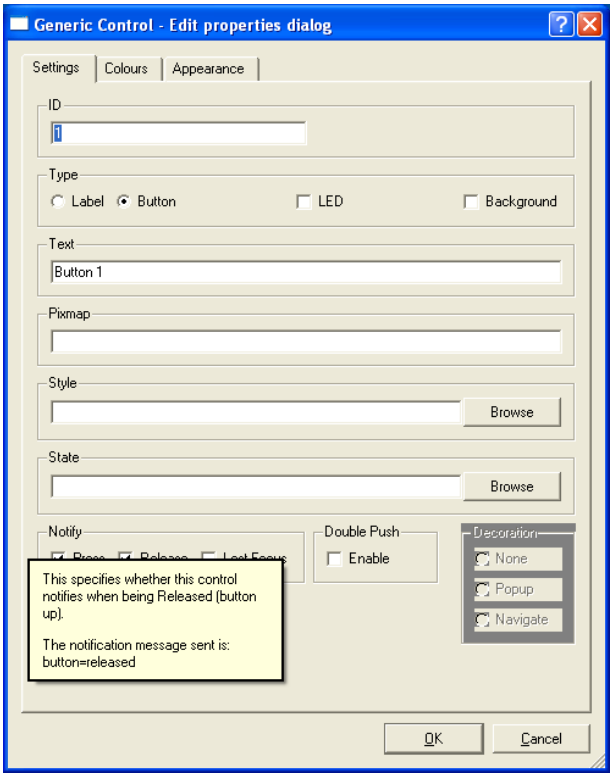


Figure 5 - Tool Tip for Notify Release

Note the last line of the popup help box tells you what gets sent back by this control for this notification.

Our buttonCallback parameter 'b' has all the information that we require in a form that we can readily use it.

For a notification of

```
button=pressed
```

b->command() will return 'button'

b->value() will return 'pressed'

You'll notice in our example above that we're still not checking to see that the notification is a 'button' notification – perhaps we ought but this type of control doesn't put out any other kind of notification.

8.1 Responding to button presses - Tutorial 2

What you should have learnt from this tutorial:

- How to change the colours of a control
- How to respond to different notifications from a control

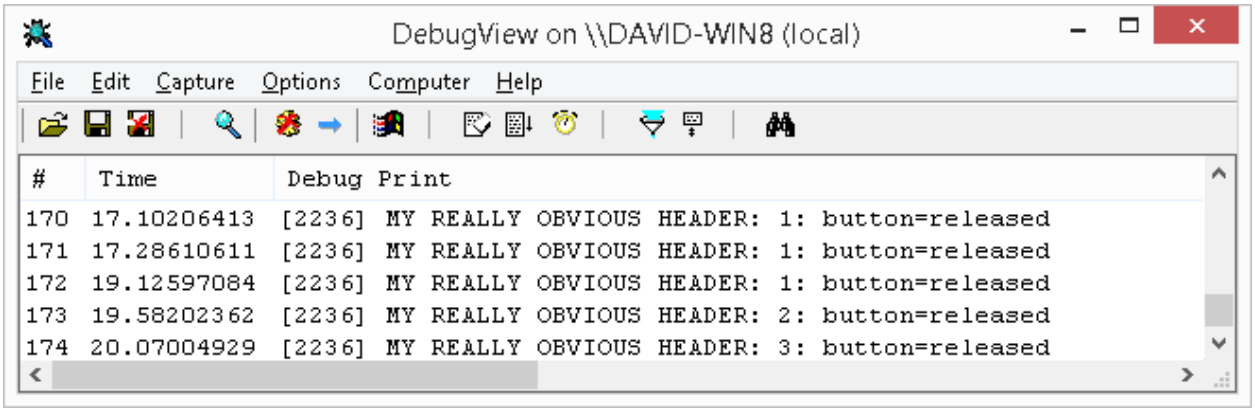
8.2 Handy debugging tip

Sometimes you just want to trace the button events – you can write your own debug routine but there's a built in function that you can use to do this for you:

```
// all button pushes and notifications come here
void ccc::buttonCallback( buttonNotify *b )
{
    b->dump( "MY REALLY OBVIOUS HEADER" );

    if( b->panel() == 1 )
    {
        switch( b->id() )
        {
```

Just use the dump method of the buttonNotify object – this sticks a line with your header text to the external debug viewer (such as dbgview.exe). The example above looks like this in the debugger – you get the numeric button id (1, 2 or 3) then the param=value notification that you're sent:



9 Button naming - Tutorial 3

In this tutorial we'll not extend the functionality of tutorial 2 at all. But we will change the way in which we reference the controls.

For this tutorial we've changed the names of the buttons from

```
1, 2, and 3
```

to

```
button_1, button_2, and button_3.
```

And the name of the label from

```
4
```

to

```
status
```

9.1 Looking at the program

Starting with the constructor. In tutorial 2 to show a panel we used

```
panelShow(  1, "p1.bncs_ui" );
```

but now we want to give it a more meaningful name:

```
panelShow( "mainPanel", "p1.bncs_ui" );
```

This means show panel of template 'p1.bncs_ui' which I'll know as 'mainPanel'. Wherever we need to refer to this panel, we must use the identifier 'mainPanel'.

Look at the difference between the types of the first parameter. One is an integer, one is a string. You wouldn't be able to do this in normal C programming but C++ allows you have several functions of the same name which take different parameter types. The class that allows us to do this is the bncs_string class which is a very versatile string handling class – it has automatic casts to allow you to use a string in place of an integer and vice-versa, and has various other useful features we'll come to throughout these tutorials.

Taking a look at the code that writes the text to the status label. In our original example it looked like this:

```
textPut( "text=control 1", 1, 4 );
```

and now the same code looks like this:

```
textPut( "text=control 1", "mainPanel", "status" );
```

Note that we use our new panel identifier 'mainPanel' and that we call the label by its new name of 'status'.

You'll also have noticed if you compare Tutorial 2 and Tutorial 3 that we've had to change the `switch-case` in the buttonCallback as `switch` can only ever work with constants that can be evaluated at compile-time. This has been replaced with an `if-else` chain.

Otherwise Tutorial 2 and Tutorial 3 are functionally entirely the same.

9.2 Button naming - Tutorial 3 Summary

What you should have learnt from this tutorial:

- How control names can be numeric or alpha numeric
- How to use control names programmatically

9.3 Other notes on button naming

This example (for which there is no separate tutorial) requires a little more C++ knowledge than previous examples – you may wish to skip this section at this stage.

If you have an array of buttons you can always give them names you can split up and make use of in your program.

Lets say you have two button groups 'source' and 'dest'. Call the first array of buttons

'source_1' to 'source_100', and the second array 'dest_1' to 'dest_100'.

If we can split the button id on the underscore we can use the numerical part of the identifier directly.

When you get a buttonCallback notification split the value string on '_'. Consider the following code fragment:

```
void tutorial_3::buttonCallback( buttonNotify *b )
{
     // we'll expect button id's such as:
      //    source_1
      //    source_44
      //    dest_88
      //    take

     bncs_string  part1, part2;
     // split the id() into 2 on an underscore – look at the
      //  bncs_string documentation
     b->id().split( '_', part1, part2 );

     if( part1 == "source" )
     {
               doSourceStuff( part2 );          // part2 has just the src button
no.
     }
     else if( part1 == "dest" )
     {
               doDestStuff( part2 ); // part2 has just the dest button no.
     }
     // neither source, nor dest – might be something else
      else if( b->id() == "take" )
      {
            doTake();
      }
```

10 Using pixmaps and getting back data – Tutorial 4

In this tutorial we'll see how to use pixmaps on buttons and get settings back from a control. For simplicity this tutorial is based on Tutorial 2 (where we set different colours for the pressed and released events)

In the dialog description we've given each of the buttons a pixmap – and chosen the text and alignment options so the pixmap appears on top of the text.

Here's what it looks like **Figure 6 - Panel with Pixmaps**

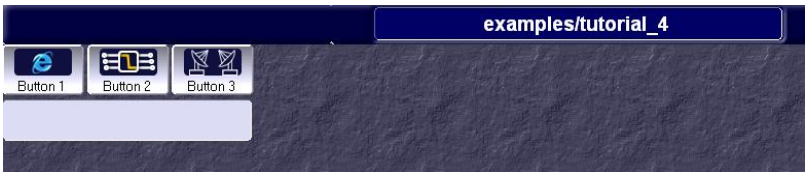


Figure 6 - Panel with Pixmaps

Double click one of the buttons at design time – find out where the pixmap names are set and where the text and pixmap alignment options are.

Very important note:

Pixmap paths are relative to the '.bncs_ui' file that loaded them. So in this case the image names have no path so appear in the same directory as the '.bncs_ui' file.

'Absolute' paths that begin with a drive letter (such as 'C:\myimages\stuff.png') are not allowed.

For paths that start with a slash (usually an absolute path) such as '/images/shared.png' the root is taken to be from your systems 'panels' directory (in this case this path would equate to C:\bncs\v4.5\panels\images\shared.png).

Here are some path examples:

Path	Valid?	
/images/myimage.png	**Y**	Path where the root is the panels directory of this BNCS system
\images\myimage.png	**Y**	As above – forward slashes and back slashes can be used interchangably.
../../images/myimage.png	**Y**	up two directories and back down via an images subdirectory
images/myimage.png	**Y**	subdirectory off this path
C:\stuff.png	**N**	Absolute paths are not allowed – images should exist underneath the panels sub-directory of this system.

Now when you hit any of the buttons, the pixmap on the label changes. Then when released it changes to the pixmap from the button you pressed. Try it!

Here's the code with the relevant parts highlighted:

```
void tutorial_4::buttonCallback( buttonNotify *b )
{
```

```
        if( b->panel() == 1 )
        {
                // work out the type of notification - pressed or released?
                if( b->value() == "pressed" )
                {
                        textPut( "pixmap=surprise/image5.png", 1, 4 );
                        textPut( "pixmapstretch=true", 1, 4 );
                        textPut( "text=", 1, 4 );
                }
                else if( b->value() == "released" )
                {
                        bncs_string pix;

                        switch( b->id() )
                        {
                                case 1:
                                        textPut( "text=control 1", 1, 4 );
                                        textGet( "pixmap",1, b->id(), pix );
                                        textPut( "pixmap", pix, 1, 4 );
                                        break;
                                case 2:
                                        textPut( "text=control 2", 1, 4 );

                                        textGet( "pixmap",1, b->id(), pix );
                                        textPut( "pixmap", pix, 1, 4 );
                                        break;
                                case 3:
                                        textPut( "text=control 3", 1, 4 );

                                        textGet( "pixmap",1, b->id(), pix );
                                        textPut( "pixmap", pix, 1, 4 );
                                        break;
                        }
                        textPut( "pixmapstretch=false", 1, 4 );
                }
        }
}
```

Let's consider what happens when a button is pressed:

```
                        textPut( "pixmap=surprise/image5.png", 1, 4 );
                        textPut( "pixmapstretch=true", 1, 4 );
                        textPut( "text=", 1, 4 );
```

- We set the image on the label (control 4) to 'image5.png' which is in a subdirectory 'surprise' relative to the 'p1.bncs_ui' file.
- We make it so that the pixmap fills the entire button by stretching it.
- We reset the text to nothing so that the pixmap displays cleanly.

Let's look at what happens when a button is released:

```
bncs_string pix;

    switch( b->id() )
    {
            case 1:
                    textPut( "text=control 1", 1, 4 );
                    textGet( "pixmap", 1, b->id(), pix );
                    textPut( "pixmap", pix, 1, 4 );
                    break;
}
textPut( "pixmapstretch=false", 1, 4 );
```

- We set the text on the control as in previous tutorials.
- We get the pixmap from the button that was released with a textGet command and put the name of the pixmap in local variable 'pix'. We've specified what parameter we want the control to return us. In this case we're asking for the pixmap name but it could be any setting this control supports.
- We put the pixmap value we've just obtained to the label control (control 4). Functionally this "copies" the pixmap from the button to the label.
- We set the button to display the pixmap at its proper size (unstretched)

10.1 Using pixmaps and getting back data – Tutorial 4 Summary

In this tutorial we've learnt:

- How to set pixmaps at design time
- How to set pixmaps dynamically at run time
- How paths to pixmaps work
- How to get the value of a setting from a control

11 Using a Timer and Introducing Private variables – Tutorial 5

This tutorial shows how to use timers and introduces 'global' variables for this application.

For simplicity, it's based on a basic wizard generated project.

The result is the 'label' (control 4) flashing between red and green from the moment the panel starts up.

11.1 A look at the code:

The important code is highlighted below:

```
tutorial_5::tutorial_5( bncs_client_callback * parent, const char * path ) :
bncs_script_helper( parent, path )
{
    // show a panel from file p1.bncs_ui and we'll know it as our panel 1
    panelShow( 1, "p1.bncs_ui" );

    toggle = false;
    timerStart( 1, 500 );
}
```

We're starting a 500 millisecond timer we'll know as timer '1'.

When a timer event occurs we get told about it in the timerCallback function:

```
// timer events come here
void tutorial_5::timerCallback( int id )
{
    if( id == 1 )
    {
            if( toggle )
                textPut( "colour.background", "red", 1, 4 );
            else
                textPut( "colour.background", "green", 1, 4 );

            toggle=!toggle;      // flip the flag to the opposite state
    }
}
```

We get told the identity of the timer that we passed when we started the timer – in this case '1'.

Let's look at the `toggle` definition in the header file for this application:

```
class tutorial_5 : public bncs_script_helper
{
public:
    tutorial_5( bncs_client_callback * parent, const char* path );
```

```
        virtual ~tutorial_5();

        void buttonCallback( buttonNotify *b );
        int revertiveCallback( revertiveNotify * r );
        void databaseCallback( revertiveNotify * r );
        bncs_string parentCallback( parentNotify *p );
        void timerCallback( int );
    private:
        bool toggle;
    };
```

Variables that need to be available for this application should be declared like this – as a private variable within your panel application class.

You *could* declare it here in the cpp file:

```
// this nasty little macro to make our class visible to the outside world
EXPORT_BNCS_SCRIPT( tutorial_5 )
// and this nasty little macro to describe it
DESCRIBE_BNCS_SCRIPT( "Using a timer and Private variables - Tutorial 5" )

bool toggle;                    // don't declare this here

// constructor - equivalent to ApplCore STARTUP
tutorial_5::tutorial_5( bncs_client_callback * parent, const char * path ) :
bncs_script_helper( parent, path )
{
    // show a panel from file p1.bncs_ui and we'll know it as our panel 1
    panelShow( 1, "p1.bncs_ui" );

    toggle = false;
```

It would work - but then there would be one instance ever. If you had two instances of your panel (or component) they'd share a single instance of this variable. If you set it from one instance of the panel you'd set the same variable for both panels – this could lead to some very odd functionality and it would probably be quite hard to determine what's actually going on.

11.2 What to try next

Try making all the controls flash at different rates

11.3 Using a Timer and Introducing Private variables – Tutorial 5 Summary

What you should have learnt from this tutorial:

- How to start a timer, and where to get the timer events
- How and where to add 'global' variables to this class

12 Connecting to the network – Tutorial 6

In this tutorial we'll connect to a network device – an infodriver. This tutorial is based on a simple Wizard generated project.

Functionally this application is virtually identical to a standard Wizard generated project but instead of writing text directly to the label, it's written to a network device. The revertive from this network device sets the text on the label.

12.1 A look at the code

The registration and polling of the network device is highlighted below.

In this example we're registering for the first slot on device 1.

```
tutorial_6::tutorial_6( bncs_client_callback * parent, const char * path ) :
bncs_script_helper( parent, path )
{
    // show a panel from file p1.bncs_ui and we'll know it as our panel 1
    panelShow( 1, "p1.bncs_ui" );

    infoRegister( 1, 1, 1 );
    infoPoll( 1, 1, 1 );
}
```

When any revertive we've registered for arrives we get told about it in our revertiveCallback function. The information regarding this event arrives ready split in a form that we can already make use of.

The relevant bits of code are highlighted below:

```
int tutorial_6::revertiveCallback( revertiveNotify * r )
{
    switch( r->device() )
    {
            case 1:
                    switch( r->index() )
                    {
                            case 1:
                                    textPut(   "text",
                                "revertive=" + r->sInfo(),
                                1,
                                4 );
                                    break;
                    }
                    break;
    }
    return 0;
```

```
}
```

To find out about the device that generated this revertive we can look at the parameter 'r' that was passed with this function. This has various member functions that tell us everything that we need to know about this revertive. Look at the documentation for revertiveNotify to find out how revertives of different types are split up – there are examples for each type (GPI, Router, Infodriver).

We'll use the following statement to determine the current device number:

```
switch( r->device() )
```

Then we need to check the slot – or rather more generically known as *index*:

```
switch( r->index() )
```

We're feeding both of these into switch statements to make using them easy – you could equallly have used

```
if( r->device() == 1 )
```

The information – in this case as it's an infodriver revertive, is returned in a 'string info' function `sInfo()`:

```
"revertive=" + r->sInfo(),
```

What we're doing here is appending the revertive to the end of a static string. We can do this because `sInfo()` returns the versatile `bncs_string` class.

12.2 Writing to the network

The relevant parts of the code are highlighted below:

```
void tutorial_6::buttonCallback( buttonNotify *b )
{
    if( b->panel() == 1 )
    {
            switch( b->id() )
            {
                    case 1: infoWrite( 1, "you pressed|control 1", 1 );
    break;
                    case 2: infoWrite( 1, "you pressed|control 2", 1 );
    break;
                    case 3: infoWrite( 1, "you pressed|control 3", 1 );
    break;
            }
    }
}
```

We've changed this code from that generated by the Wizard so that we write messages to the infodriver slot when buttons are pressed.

12.3 What to try next

Modify this example to accept router and GPI revertives.

12.4 Connecting to the network – Tutorial 6 Summary

What you should have learnt from this tutorial:

- How to register for revertives from a device
- Where revertives arrive in your application
- How to write to a device

13 Database Name Changes – Tutorial 7

In this tutorial we'll make simple BNCS database name changes. This tutorial is based on a simple Wizard generated project.

The tutorial simply changes the name of source 12 to name 'Name-1', 'Name-2' or 'Name-3' depending upon which button was pressed.

Notification of the name change is displayed in the label of the wizard generated panel.

13.1 A look at the code

The section for generating name changes from button pushes is shown below. The relevant section is highlighted.

```
void tutorial_7::buttonCallback( buttonNotify *b )
{
	if( b->panel() == 1 )
	{
		switch( b->id() )
		{
			case 1: routerModify( 1, 0, 12, "Name-1", false ); break;
			case 2: routerModify( 1, 0, 12, "Name-2", false ); break;
			case 3: routerModify( 1, 0, 12, "Name-3", false ); break;
		}
	}
}
```

when a button is pressed a database name change request is sent to the network using the routerModify() call. In this is example device 1, source (database 0), index 33 with new name 'Name-1'. The last parameter determines whether a tally dump is sent to the network.

When a name changes on the network our databaseCallback() function is called. . The information regarding this event arrives ready split in a form that we can already make use of it. This is the same structure as the revertiveCallback() function of a previous tutorial.

In our tutorial all we do is formulate a string and write to the display – this uses all the relevant members of the revertiveNotify class.

```
void tutorial_7::databaseCallback( revertiveNotify * r )
{
	bncs_string s = bncs_string( "device=%1, index=%2|database=%3, name=%4" )
.arg( r->device() )
.arg( r->index() )
.arg( r->database() )
.arg( r->sInfo());

	textPut( "text", s, 1, 4 );
}
```

The only other thing to note is that we must be registered for this device in order to receive notifications.

This must be added to the contstructor to enable notifications for this device:

```
infoRegister( 1,1, 1 );
```

Note that you don't have to be registered for the actual slot being changed – just anything on this device. This is a standard BNCS feature.

13.2 What to try next

Write an on-screen keyboard to enter text and then use this text to set database name changes.

13.3 Database name changes – Tutorial 7 Summary

What you should have learnt from this tutorial:

- how to change BNCS database names
- how to get notification of database name changes

14 Using our panel as a component – Tutorial 8

In this tutorial we'll see how to wrap up a "panel" and use this as a component on another panel. The idea is that you (or hopefully someone else) write some complicated thing for controlling a piece of equipment and you just want to reuse this component on your panel.

For this we'll have to create two projects – these are just slightly modified Wizard applications.

We need to create:

- a simple reusable component
- the host application

14.1 Creating the host application

The host is a normal Wizard application – but with all the buttons removed from the dialog.

That's it for now – we'll come back to this in a bit.

14.2 Creating the component.

We'll just use a virtually standard wizard generated project here. This is called 'component' for obvious reasons!

The only couple of things to note are:

- We'll create this in a subdirectory of the tutorial_8 directory. This is so that if you copy the tutorial_8 directory you also take with it its components.
- We must set the size of the dialog in the Visual Editor – there is a resize handle in the bottom right of the dialog but may be more easily done by right clicking on the background of the dialog and editing the size in 'Dialog Properties'
- We must set the size of this component to the size of the dialog

Set the size of the component using one of the functions in the constructor:

```
component::component( bncs_client_callback * parent, const char * path ) :
bncs_script_helper( parent, path )
{
	// show a panel from file p1.bncs_ui and we'll know it as our panel 1
	panelShow( 1, "p1.bncs_ui" );

	// you may need this call to set the size of this component
	//  if it's used in a popup window
//	setSize( 1024,668 );	// set the size explicitly
	setSize( 1 );					// set the size to the same as the specified panel
}
```

There are two ways of setting the size of the component – either in pixels or by matching it to the size of a dialog. The dialog method is easiest so that's what we're doing here. Note that the number, '1' in this case, is the panel identifier as used in panelShow.

14.3 Using this component

To make our component appear we need to go back and look at our host application dialog.

From the buttons tab on the Visual Editor select a 'Script' control and drop this onto the dialog:

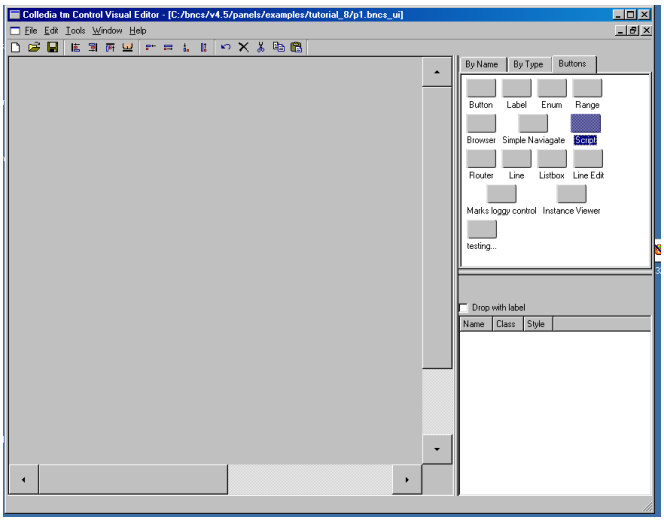


Figure 7 - Select a Script Control

Nothing of any great interest will appear.

Double click the script control to get the properties dialog up:

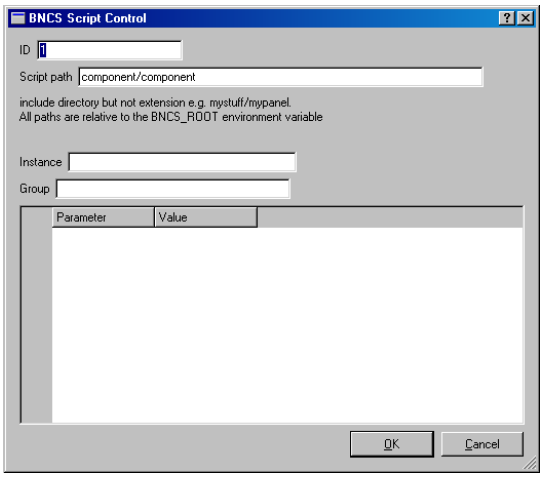


Figure 8 - Script Control Edit Dialog

Add in the path to your script component. Note the same rules regarding paths apply to scripts as to pixmaps as we saw in the pixmap example (Tutorial 4). In this case the path does not start with a slash as it's relative to the host script (i.e. the 'component' subdirectory off the 'tutorial_8' directory). We then must give the DLL name which incidentally matches the directory name. Using relative paths ensures that this host/component pair will continue to work together wherever they are placed in a directory structure.

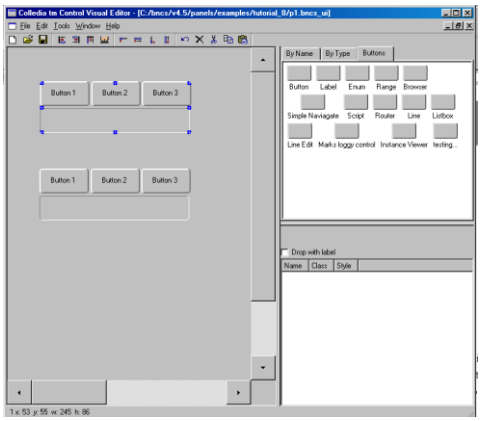


Figure 9 - Panel using Two Components

You should then have our familiar 3 buttons with label display. Only instead of this being individual buttons they appear as a single control when selected in the Visual Editor. Copy this control using Copy and Paste or Ctrl-C Ctrl-V and arrange both controls so that they don't overlap.

Save the host dialog and then run the tutorial_8 in panel manager.

Our controls do the usual 'Your pressed control 1' to the label but note that they are entirely independent.

14.4 What to try next

Change the component to use pixmaps (referenced relatively) – note that the paths are always relative to the component that loads them.

14.5 Using our panel as a component – Tutorial 8 Summary

What you should have learnt from this tutorial:

- The minor differences between a 'top level' view and a component

15 Having our component tell us things – Tutorial 9

In this tutorial we'll extend the functionality of Tutorial 8 to have our components tell the host application which button has been pressed.

The previous tutorial wraps up a simple function that is entirely stand-alone. More useful is if the control can interact with its host application.

15.1 Component.

This as you'll remember is more or less a standard wizard application. All we did was set the size of the dialog.

The only further modification for this tutorial is shown below. This has changed in layout from the wizard generated code but only the highlighted parts are different.

```
void component::buttonCallback( buttonNotify *b )
{
	if( b->panel() == 1 )
	{
		switch( b->id() )
		{
			case 1:
				textPut( "text", "you pressed|control 1", 1, 4 );
				hostNotify( "button=control 1" );
				break;
			case 2:
				textPut( "text", "you pressed|control 2", 1, 4 );
				hostNotify( "button=control 2" );
				break;
			case 3:
				textPut( "text", "you pressed|control 3", 1, 4 );
				hostNotify( "button=control 3" );
				break;
		}
	}
}
```

When we press one of our buttons we send a message to our host – this message is in the standard <parameter>=<value> format.

15.2 Host

The two script components on the dialog of the host panel are just buttons – they have ids just like any other button and can send messages just like any other button.

Our 2 script components have ids of '1' and '2'. Open up the host dialog description (.bncs_ui) file and check this out.

Our host receives messages from any of its buttons in the buttonCallback function exactly as in previous tutorials.

```
void tutorial_9::buttonCallback( buttonNotify *b )
{
      if( b->panel() == 1 )
      {
                  switch( b->id() )
                  {
                              case 1: // command is "button"
                              // value is "control n" where n is button number
                              textPut( "text", "component 1 - " + b->value(), 1, "status" );
                              break;

                              case 2:
                              textPut( "text", "component 2 - " + b->value(), 1, "status" );
                              break;
                  }
      }
}
```

This code is almost exactly the same as if this host code were dealing with normal buttons. The two parts of the returned string of "button=control 1" are already split up in the buttonNotify class. We simply construct a string to indicate which component generated the event and then add the value that was returned from the component.

Here's the sample output – the last button pressed was button 2 on the lower component:

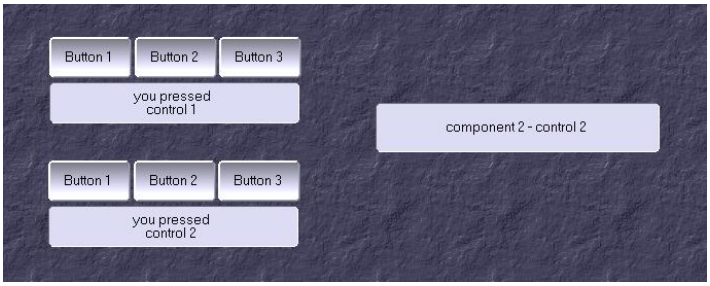


Figure 10 - Panel with two Components

15.3 What to try next

Extend this example so that the component notifies on button press and button release. Clue: notifications should change to the format:

```
button.<button id>=pressed
```

So button 2 released would have the notification:

```
button.2=released
```

15.4 Having our component tell us things – Tutorial 9 Summary

What you should have learnt from this tutorial:

- how a component may pass information back to its host

16 Telling our component things – Tutorial 10

In this tutorial we'll extend Tutorial 9 to pass information to a control to give it context.

Control id "1" will have a red label and control id "2" a green label.

Here's what it should look like:

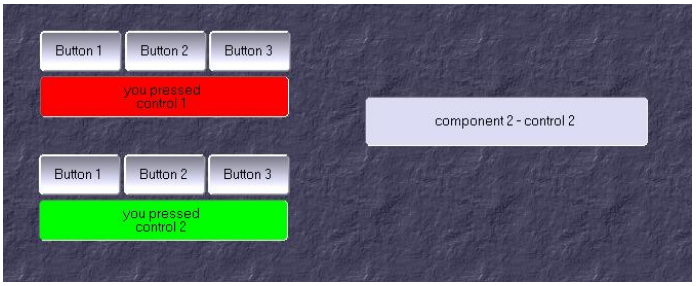


Figure 11 - Panel with two components with different properties

16.1 Component

The component is told of messages passed to it in a parentCallback function. This is passed a parameter with all the parameters required by the parentCallback function in a form that we can already use.

```
bncs_string component::parentCallback( parentNotify *p )
{
	if( p->command() == "colour" )
		textPut( "colour.background", p->value(), 1, 4 );
	return "";
}
```

The component expects a command in the form

```
colour=red
```

so the command() function of parentNotify will contain 'colour' and the value() the actual colour (in this case 'red').

In our example above we just pass this value directly onto our status label.

16.2 Host

The host must tell its component the colour to go to – this is done using the usual parameter passing mechanism. Our two components are ids '1' and '2' so in the constructor of the host application we see something like this:

```
tutorial_10::tutorial_10( bncs_client_callback * parent, const char * path ) :
bncs_script_helper( parent, path )
{
	// show a panel from file p1.bncs_ui and we'll know it as our panel 1
	panelShow( 1, "p1.bncs_ui" );

	// colour the components
	textPut( "colour", "red", 1, 1 );
```

```
textPut( "colour", "green", 1, 2 );
```

This is the generic means of sending commands to a control – setting the colour is a trivial example but you can set slots, parameters, device numbers using the same mechanism.

16.3 Important Note

This tutorial demonstrates how you can pass parameters to a control; however, these can only be passed at runtime (i.e. in a script). If you want to configure a control at design time then see later tutorials.

16.4 What to try next

Extend the parentCallback function so that you may set the text of each of the 3 buttons individually.

Clue: the command should be in the format:

```
text.<id>=<text>
```

e.g.

```
text.2=Hello
```

sets the text of button 2 to "Hello"

16.5 Having our component tell us things – Tutorial 9 Summary

What you should have learnt from this tutorial:

- how you may pass parameters to a control so that 2 identical components are "purposed"

17 Having a component save it's settings – Tutorial 11

In this tutorial we'll extend Tutorial 10 so that the colour of the component can be set at design time from within the Visual Editor.

Here's the design-time properties dialog for one of the components we're trying to create:

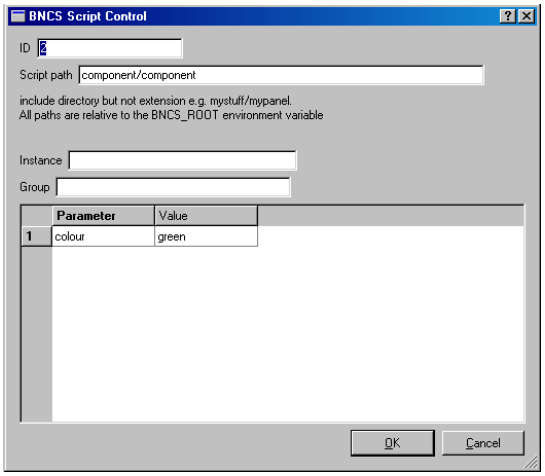


Figure 12 - Properties of a Script

In the bottom half of this display we're able to set the colour we want this component to be.

17.1 Host

There are no changes to the C++ code. We'll edit the dialog description (.bncs_ui) file later.

17.2 Component

The editor needs to ask this component what settings it needs to store. This is done by asking the component to "return" settings.

The component is passed a command of 'return' and a value of 'all'. All components should respond to the 'all' value and may optionally respond to individual parameter requests. In our case we'll only ever return 'all' – for clarity we won't even check the value being requested.

In our previous tutorial we just pass the colour directly onto the label control. Here we must store the colour in a private variable so that we've got it when asked for it later (see Tutorial 5 for example of private variables).

```
bncs_string component::parentCallback( parentNotify *p )
{
    if( p->command() == "colour" )
    {
        colour = p->value();

        textPut( "colour.background", colour, 1, 4 );
    }
    else if( p->command() == "return" )
```

```
        {
                bncs_string ret = "colour=" + colour;

                return ret;
        }
    return "";
}
```

The colour command still works much the same as it did in the last tutorial except that we're storing the colour in a variable.

We've added an 'if' statement for a 'return' command. All we have to do is formulate a string with as many <parameter>=<value> pairs as this control wants to save. For more than one these are newline delimited.

Because we've returned a value in the format

```
colour=red
```

this is saved when the dialog description is saved and appears as one of the parameters in the lower half of the script design time properties editor. What's saved when the dialog description is saved, and what appears in this list of settings are directly related.

17.3 What to try next

Extend the component so that you can set the names of the 3 buttons (as in the previous tutorial's 'What to try next' and then save this text. Remember the return value is newline delimited so you should end up with something that looks a bit like this:

```
. . .

 else if( p->command() == "return" )
{
        bncs_string ret = "colour=" + colour + "\n" +
                "text.1=" + text1 + "\n" +
                "text.2=" + text2 + "\n" +
                "text.3=" + text3;

        return ret;
}
```

17.4 Having a component save it's settings – Tutorial 11 Summary

What you should have learnt from this tutorial:

- how a component saves its settings
- how to give a component design time editable properties

18 Navigation – Tutorial 12

18.1 Navigating to another view

18.2 Navigating to another app

19 Settings – Tutorial 13

In this tutorial we'll look at how to obtain information about this workstation such as:

- workstation
- user / level
- workstation settings
- object settings

The tutorial just has a few buttons on the dialog that are populated from a few simple calls to obtain information listed above.

19.1 Getting the workstation

There is a simple call to access this information:

```
textPut( "text", workstation(), 1, "workstation" );
```

19.2 Getting user information.

There are 3 simple calls to obtain the name of the user and their "level".

```
textPut( "text", getUser(), 1, "user" );
textPut( "text", getUserLevel(), 1, "userlevel" );
textPut( "text", getUserLevelInt(), 1, "userlevelint" );
```

The user level defines their basic privilege (such as Administrator, User etc.). There is a numeric user level value. This value increases with authority so a User is 1, but an Administrator is 3. It's assumed that each user level has access to everything at that level and below so you might see something like this:

```
if (getUserLevelInt() >=3 )
      doStuff();
```

doStuff() when we're Administrator or higher privilege.

19.3 Getting workstation specific settings

Workstation specific settings are things like:

- Video or audio router monitoring destinations associated with this workstation
- Suite number – where this workstation sits in a building
- Desk number – where this workstation sits on a dest

These are editable from within the CC V4.5 configuration tool.

There is a simple call to obtain workstation settings:

```
textPut( "text", getWorkstationSetting( "mon" ), 1, "workstationsetting" );
```

This example gets hold of a setting called 'mon' for this workstation – the workstation number is implied and does not need to be specified.

19.4 Getting device specific settings

Object specific settings provide information for a named object.

These are editable from within the CC V4.5 configuration tool. There is a simple call to access this information.

```
textPut( "text", getObjectSetting( "testObject", "mon" ), 1, "objectsetting" );
```

This call returns the value of the ‘mon’ parameter for the object ‘testObject’.

“Object settings” tends to be the dumping ground for a few random settings for this-and-that. The command is easily implemented in the UI and the file format pretty easily understood and consistent.

If your requirements get more complicated try coming up with a specific XML file that you read with a bncs_config object. The file can then be whatever (XML) format you like and the changes easier to version, share and deploy.

20 Connections – Tutorial 14

You've seen in the earlier tutorials how to create monolithic applications. Hopefully however you can create or even better *reuse* some sub-components and stick them together to create composite panels.

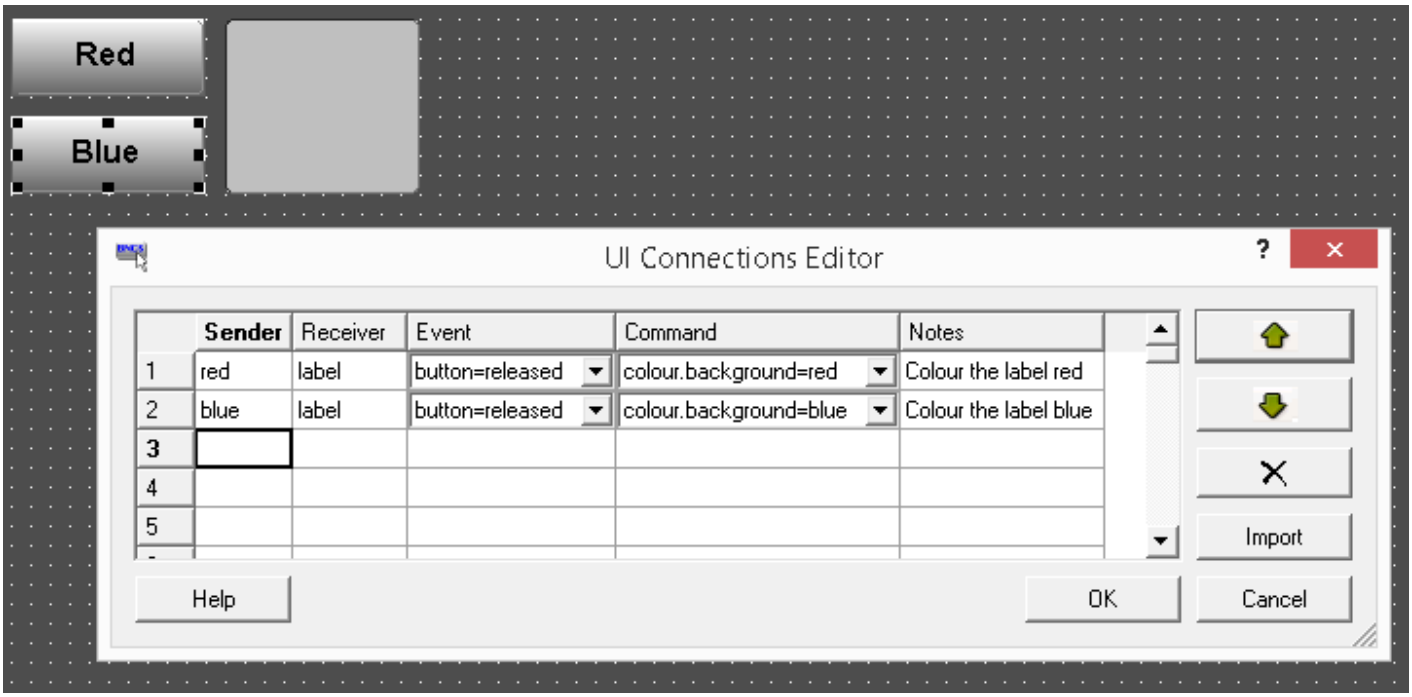
This has shown to work really well – there are literally hundreds and hundreds of panels that are just collections of other components glued together in different ways.

To work though you need to be able pass messages from one component to another. You can create a DLL project just to link the output of one component to the input of another – lots of projects do just this.

Wouldn't it be handy though if you could do this direct in the UI? Have no code at all? You can!

Run Tutorial 14. It shows two buttons, "Red" and "Blue" and a label. Hitting the button changes the colour of the label. There's no code though – just a ".bncs_ui" file.

In the editor open the UI file and right click the background – select the "Connections" option – you'll get this:



This shows the connections within this UI.

There are four columns:

"Sender"

This is where the messages are coming from. We've given fixed names here ("red" and "blue") so a single button will match the rule – though you can use wildcards.

"Receiver"

This is where to direct messages to that match this rule – again we've got a fixed name here but it doesn't have to be

"Event"

This is what to look out for. If a message is going to be passed it must match both the "sender" field and the "event" field.

"Command"

Having matched the sender, *and* the event *and* got somewhere to send a message, this is the thing we're going to send.

20.1 Simple walk-through

So….let's say we hit and release the button whose id is "blue" – the rule on line 1 won't be matched – (that only matches when button with id red notifies us), but it does match line 2.

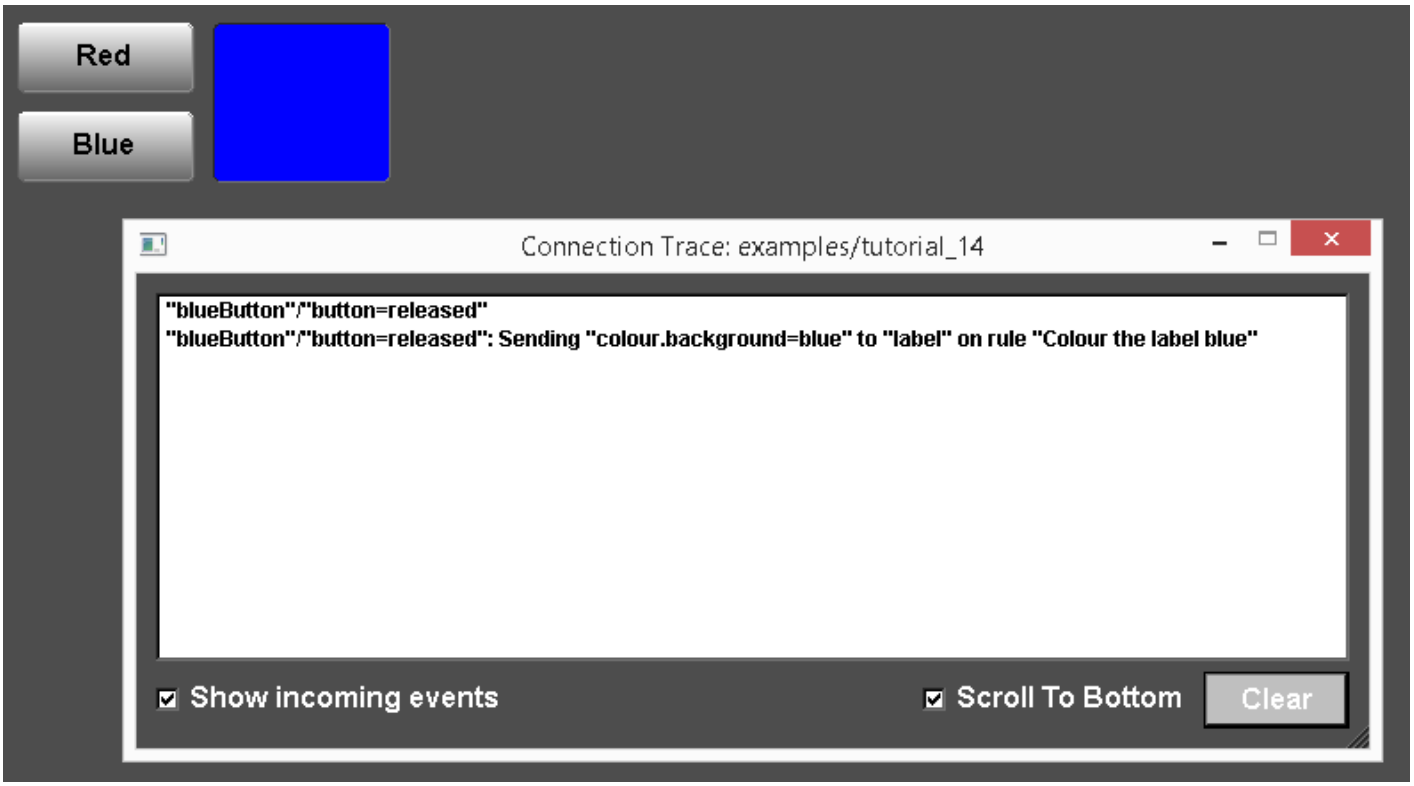
Default for a normal button is to send "button=released" events – which is what we're configured to look for in the event field. If that button been enabled for "button=pressed" notifications as well then this notification won't match this rule.

So, the rule on line 2 has been activated – both a source and an event we're interested in.

Now we send the "command" string to the receiver – so we send the command "colour.background=blue" to the "label" button.

20.2 Debugging

If you right-click a panel running in panelmanger and it has connections then you can trace these connection messages. This shows incoming messages (if enabled) as well as the messages that get sent out. Try it!



You can have a mix of connections and still have a DLL script associated with this .bncs_ui file. Best to try either having a connections-based panel or a script panel not both (but you can if you must).

If you have multiple components then you can also have multiple debug windows open to show the connection traffic for that component.

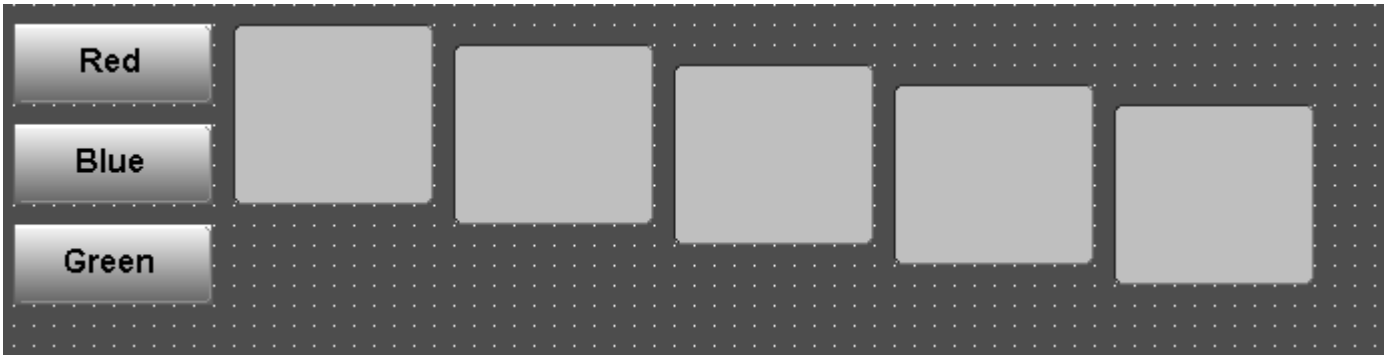
20.3 Extending connections

Our example is really simple – take a control with a specific id and connect it to another explicit control sending hard-coded messages.

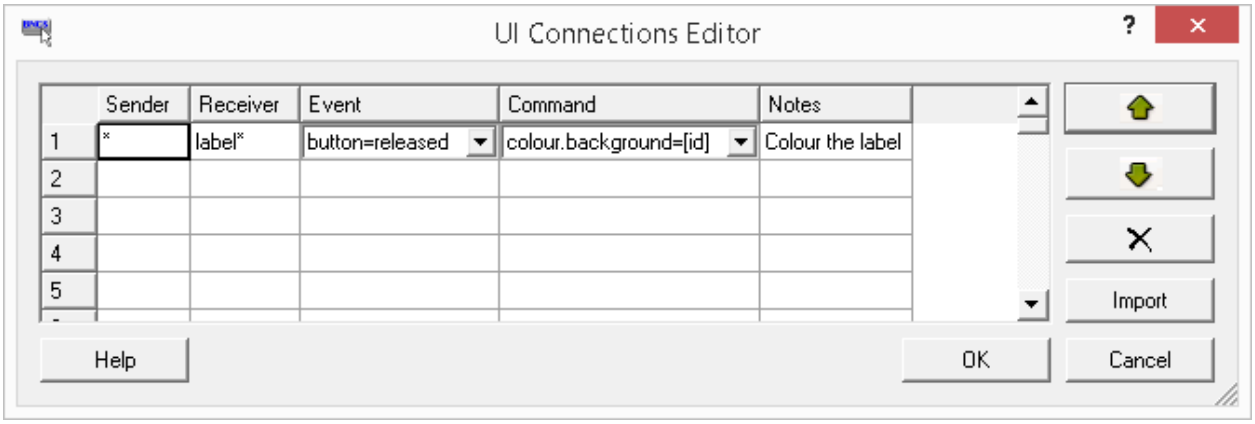
More interesting is using wildcards to match senders and receivers and don't send fixed strings but wrangle them on the fly.

Try this:

- Create a third colour button – give it an ID of "green" and text to match
- Clone (by copy/paste or use the duplicate option) the "label" button so that you have "label2" and "label3" etc. It should look something like this:



Then change the rules to look like this:

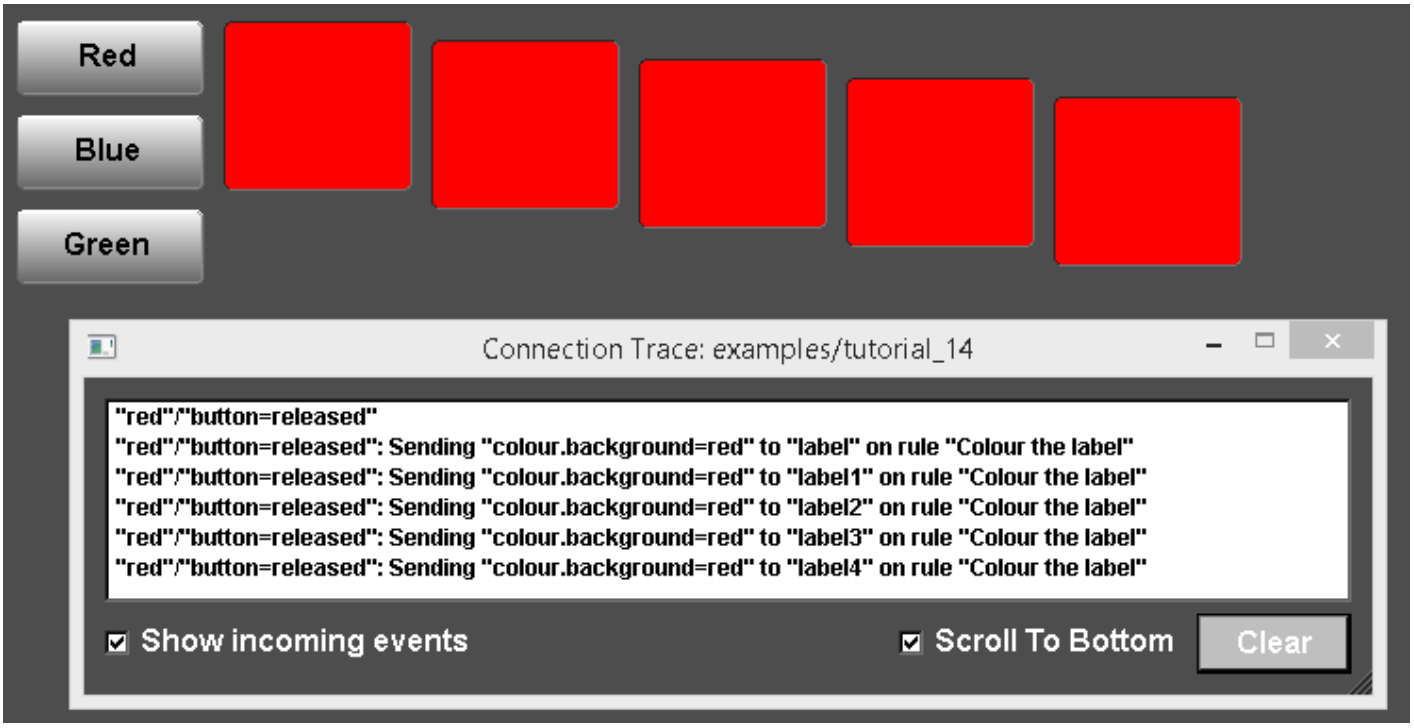


i.e.

- we've deleted one of the rules
- sender is now * (i.e. anything – this is typical wildcard matching)
- add a * to the "receiver" so it's "label*"
- set the "command" to set the colour to [id] rather than the colour itself

The rule now reads – for any control that sends me "button=released" notification, send a colour command that includes the ID of the button that sent it to me, and send this to any control that starts with an id of "label" – this includes "label", "label1", "label2" etc.

The resulting debug for this single line connection having hit the "Red" button now looks like this:



Our one line connection will now set the colour of all our labels to the value provided in the id of the button.

This is perhaps not a real-world example but a good indication of how powerful the connections mechanism can be.

21 Strings – Tutorial 15

In this tutorial we'll have a look at string handling. A number of commonly required features are available to construct strings from other strings and to split them up again.

This tutorial has not been written yet

Please refer to the bncs_string documentation

21.1 Assembling Strings

21.2 Splitting Strings

21.3 Making Lists of strings

21.4 Splitting in to a list of strings

22 Smart buttons / introducing the 'Instance' – Tutorial 16

In this tutorial we'll write a complete panel without using any C++ script at all.

You need to create a suitable subdirectory off the 'panels' directory. For the tutorials this is 'examples/tutorial_16'.

Run the Visual Editor and create a new dialog and then save it in our examples/tutorial_16 directory as tutorial_16.bncs_ui. **It's important that the name of the file matches the name of the directory.**

Drag onto the dialog a Range button from the buttons list:

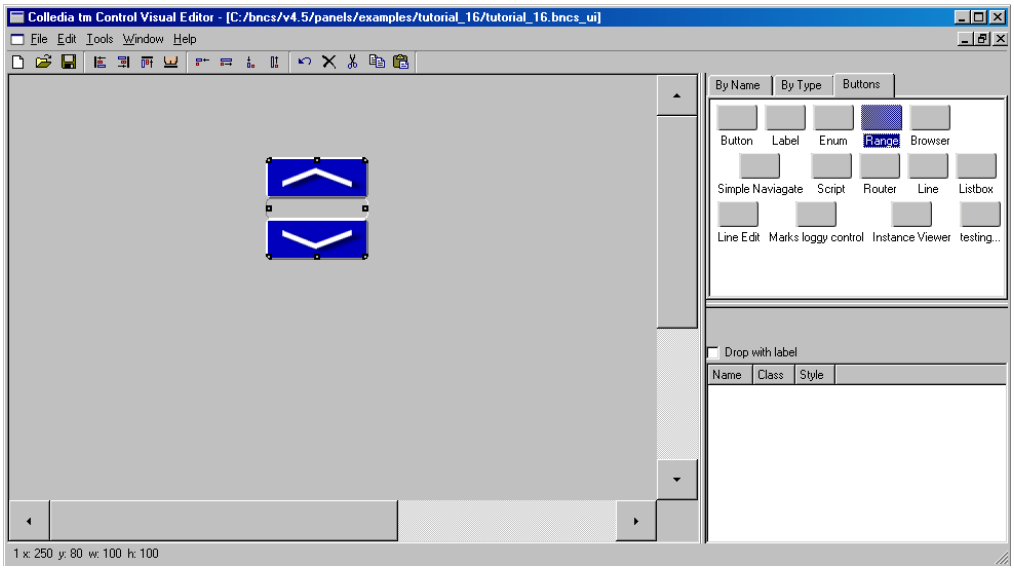


Figure 13 - A Range Control

A 'range' button is one that controls an 'analogue' parameter such as gain or frequency by incrementing or decrementing a value in a slot.

Activate the properties dialog for this control.

The first tab 'Style' sets the basic style of this control which we'll leave as 'Up/Down'.

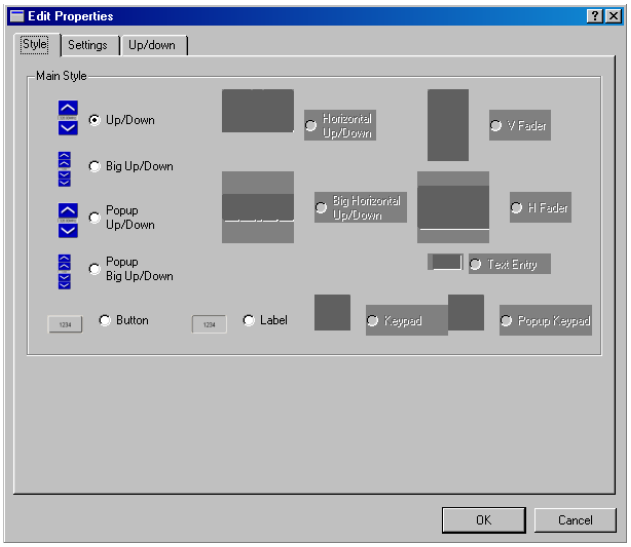


Figure 14 – 'Style' Tab for a Range Control

Now look at the Settings tab:

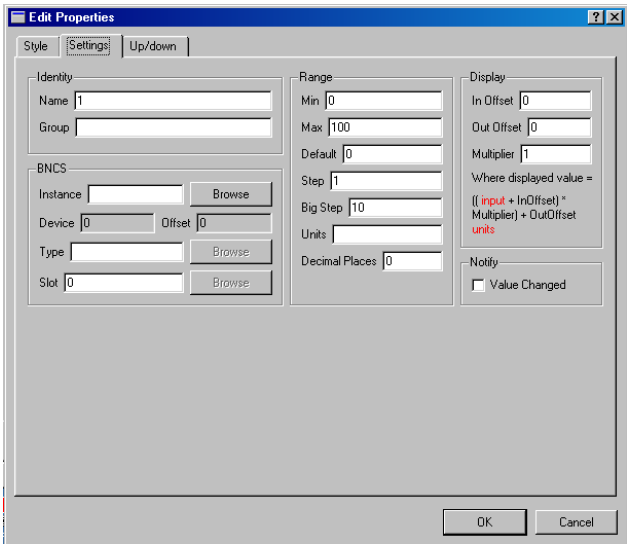


Figure 15 – 'Settings' Tab for a Range Control

If you were using this control on a "real" device you'd set up the all the range parameters to suit that device. For a bit of testing however we'll leave them at their defaults (which will at least do something).

We need to tell this control which slot on the network we want to use so we must give it an appropriate slot number in the BNCS section. However we now use a parameter name for this rather than a numeric value.

Device numbers have also been replaced with the idea of 'instances'. Put simply this is just a lookup of name to device number/offset.

22.1 Using standard smart buttons – without a script

23 Customising Smart buttons – Tutorial 17

In this tutorial we'll look at how to use a script component to customise the editor for a smart button.

This tutorial has not been written yet.

23.1 When smart buttons don't do exactly what you want

24 Help

Appendix A – Terms

Term	Meaning
BNCS version 4.5	Configuration driven system management and design. Can be thought of as a number of related lists that go do define a system.
Panel /Dialog	Used interchangeably to mean a single group of controls on a "page".
View	The same as an ApplCore application in that it may consist of several panels/dialogs and possibly custom logic
Smart control	This is a "button" (User Interface component) that connects directly to the network and provides high level control of a particular function. Some are available in ApplCore applications (up/down buttons). Many more are available in BNCS to the extent that some views can be implemented solely using smart buttons.
Skin	How buttons etc. are drawn. i.e. what the edges of the button looks like and whether the control is flat filled or shaded
Style	The "static" appearance of a control. i.e. a master alarm button might have bigger than normal font size
State	The "dynamic" appearance of a control. i.e. a master alarm button might have 2 states – red for "fail" and green for "OK"

25 Appendix B – References

The following documents are relevant:

- Installer documentation – User Guides/Installation
- Generic Control Reference – User Guides/Panel Writing/Controls/Button/Label
- Enumerated Control Reference – User Guides/Panel Writing/Controls/Smart Enum
- Range Control Reference – User Guides/Panel Writing/Controls/Smart Range
- Listbox Control Reference – User Guides/Panel Writing/Controls/ListBox

26 Appendix C – Using visual Studio 2013

All the examples in this document were developed using Visual Studio 6 and all screen-shots are also VS6. This is no longer available from Microsoft so future work will done using later Visual Studio

The BNCS project wizard generates .dsp (project) and .dsw (workspace) files that are immediately suitable only for VS 6.

To use later VS either load and convert the old .dsw file or new projects are created with compatible .sln/.vcxproj files

27

28 Version Control

Version	Date	Author	Comments

0.1		David Yates	Original version. Pre Core_System
0.2		Richard Kerry	Added to Core_system and revised,
0.3	30 October 2006	Richard Kerry	New Siemens template.
0.4	29 September 2008	Simon Armstrong	New Siemens template.
0.5	16th January 2015	David Yates	First revision of content in nearly 10 years….removes Siemens branding, removes colledia name, clarifies examples, adds new examples
0.6	19th January 2015	David Yates	More changes, enhancements and corrections