Wheatstone Corporation
Technical Documentation

GP-16P Configuration Tool
Programming Guide

- Programming Button Functions with the Script Wizard
- Creating Custom Scripts with the Script Editor

600 Industrial Drive
New Bern, NC 28562
252.638.7000
www.wheatstone.com

Revision 1.1 – January 2008
Paul Picard

Table of Contents

1 Introduction

1.1 GP-xx Hardware Compatibility 3

2 What You Need to Get Started

2.1 GP-16P Configuration Tool Software 4
2.2 Physical Network Connection 4
2.3 IP Address Settings 4
2.4 XPoint Software 5
2.5 GP-16P Help File 5

3 Using GP-16P Configuration Tool Software

3.1 Programming Procedure Summary 6
3.2 Adding Devices 6
3.3 Toggle On-Line Mode 6
3.4 Create a New Script File 7
3.5 Script Wizard Button Functions 8
3.6 Compile the Script 9
3.7 Starting the Script 9
3.8 Testing 9
3.9 Reviewing the Script Wizard Code 10

4 Configuring Device Properties

4.1 Surface Configuration 11
4.2 Starting the Device Properties Dialog 11
4.3 LIO Configuration 12
4.4 Starting the Device Properties Dialog 12
4.5 Design Philosophy 13

5 LIO Example Using Device Properties

5.1 Configure the Source Signal in XPoint 14
5.2 Configure the GP-16P LIOs 15
5.3 Create the Mic Control Script Using Script Wizard 17
5.4 Reviewing the Script Wizard Code 18
5.5. Beyond the Script Wizard 19

6 What is the Script Editor?

6.1 Script Editor Features 20
6.2 Third Party Editors 21

7 Creating Custom Scripts

7.1 Getting the Example File 22
7.2 Example Script Design 22
7.3 Auto-generated Script Components 23
7.4 Custom Start up Subroutine 23
7.5 Example Script Structure 24
7.6 Example Script –Variables and Constants 25
7.7 Example Script – Subroutines 27
7.8 Example Script – Actions 28
7.9 Custom Scripting Suggestions 29

Table of Contents (continued)

7.10 Scripting Router Control 29
7.11 Scripting Surface Control 29
7.12 Basic Surface functions 29
7.13 Advanced Surface Functions 30
7.14 Example surf_talk Commands 30

8 GP16P Scripting Language Overview
8.1 Case Sensitivity 31
8.2 Comments 31
8.3 Actions 31
8.4 Global Variables 31
8.5 Local & Static Local Variables 32
8.6 Constants 32
8.7 Arrays 32

9 GP16P Scripting Language Structure
9.1 Script Structure 33
9.2 Constant Declarations 33
9.3 Global Variable Declarations 33
9.4 Global Array Declarations 34
9.5 Local & Static Local Variable Declarations 34
9.6 Action Bodies 34
9.7 Action Parameters 35
9.8 Subroutine Bodies 35
9.9 Subroutine Parameters 35

10 Script Debugging
10.1 Finding Compiler Errors 37
10.2 Third Party Editors 37
10.3 Using “Print” and Telnet to Debug 38

Appendix A
A1 - Example Custom Script File – interlock16.ss 40

1 Introduction

This document will guide you through the process of programming a GP-8 or GP-16 panel using the GP-16P Configuration Tool software. This primer document is aimed at familiarizing you with the software's fundamentals and quickly getting your GP-xx panel up and running using the point and click Script Wizard. The Script Wizard will automatically generate computer code based on your Button and Parameter selections. This code can be compiled and downloaded right to your device from within the configuration tool.

Certain sections of this document use material located in the GP-16P Configuration Tool software's extensive Help file.

1.1 - GP-xx Hardware Compatibility

The GP-8 and GP-16 are eight and sixteen button versions of the panel and use an identical hardware platform. Scripts written for an 8 or 16 button version will run on either one with the obvious limitations stemming from the surplus or lack of buttons on the two panels.

2 What You Need to Get Started

Before you get started programming let's review all of the miscellaneous software and connection issues.

2.1 - GP-16P Configuration Tool Software

Make sure you have installed the GP-16P Configuration Tool software that came with your product's install CD-ROM. If you do not have a copy, please contact Wheatstone Technical Support at 252-638-7000 and we will email or FTP it to you.

This document uses screen shots from version 0.5.0 but the general process will apply to earlier versions.

2.2 - Physical Network Connection

Editing of GP-xx devices requires a 100BTX Ethernet connection to the device. There are two ways to connect:

100 BASETX LAN- the GP-xx device and PC are connected to a common 100BTX Ethernet switch or hub typically with straight wired RJ-45 cables. This is the preferred method.

Peer to Peer – a simple cross-over wired RJ-45 cable between the PC and device. Note that when the GP-xx device is power cycled Windows momentarily loses the network connection and takes a moment to recover.

2.3 - IP Address Settings

Make sure your PC is configured to talk to the GPxx panel. The following rules apply:

- The device's IP address is printed on a label affixed to the GP-xx panel .The default factory IP address for GP devices starts at 192.168.1.221 with a subnet mask of 255.255.255.0.
- The PC running the GP-16 Configuration Tool **MUST be on the SAME subnet** as the GP-xx device.
- For example if your GP-xx IP address is 192.168.1.221 then the NIC's IP address must be given a unique IP address on the 192.168.1.xxx subnet.
- ***WsNetServer software*** is used to assign unique static IP addresses to GP-xx panels.

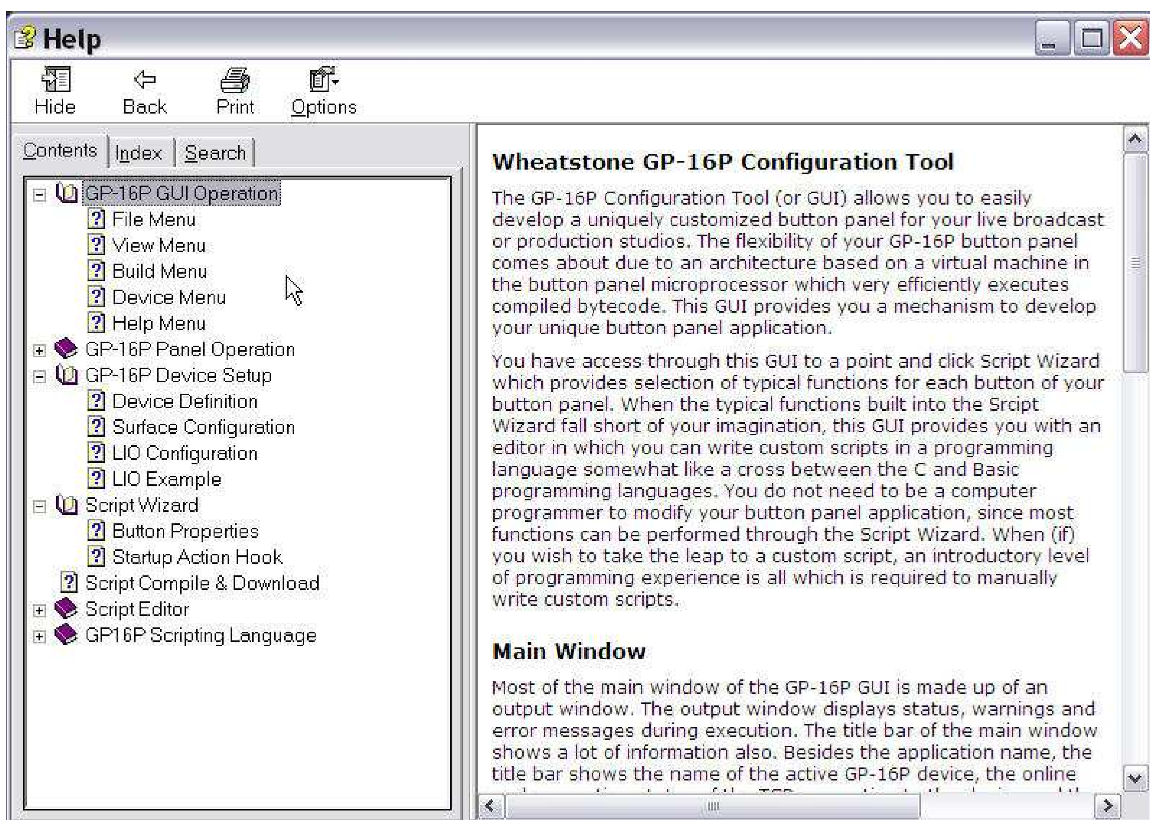
Important:
GP-xx IP addresses are assigned and changed using *WsNetServer* software.
Please refer to the WsNetServer documentation for details on *changing* a GP panel's IP address.

2.4 - XPoint Software

The GP-xx panels may be programmed to control audio and logic signal cross-points, fire Salvos, activate surface presets, and other functions. The GP-16 Configuration Tool may require you to enter Source and Destination signal ID's, Salvo indexes, and other numerical data based on ID numbers generated in XPoint. You will need access to the XPoint software and your system's configuration to get the required information.

2.5 - GP-16P Help File

The GP-16P Configuration Tool software has an extensive Windows Help Menu system. You will definitely want to utilize this asset while programming as it can be an invaluable aid, especially when creating custom scripts.



3 Using GP-16P Configuration Tool Software

OK, now that we have the network connection issues taken care of we can start the GP-16P Configuration Tool software and program the panel to perform some basic functions using the Script Wizard. The general procedure we will follow is listed below.

3.1 - Programming Procedure Summary

The steps required to program your GP-xx device are listed below - let's review them and then perform each in turn.

- Add the Device info to the GP-16P Software Tool
- Connect to the Device in Online Mode
- Create a New Script File
- Use Script Wizard to map functions to buttons
- Compile Script and Download to Device
- Start Script on the Device
- Test Functionality

If you haven't already done so, start the GP-16P Configuration Tool Software.

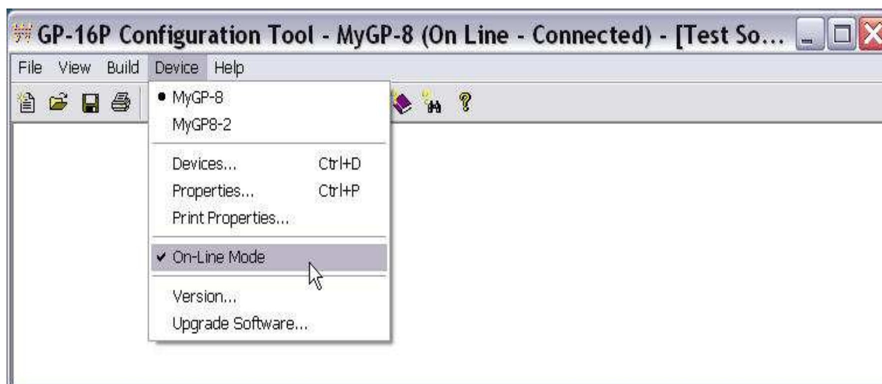
3.2 - Adding Devices

If you previously ran the software, use the Menu item ***Device->Devices…*** to *Add* or *Select* your GP-xx.

Note: If this is the first time you started the software or there are no GP devices saved, the first prompt window you will see asks for a Device Name and IP address. Go ahead and enter this information.

3.3 – Toggle On-Line Mode

Click the menu choice ***Device>On-Line Mode***.
Check that you are (On Line – Connected) in top of Title Bar.



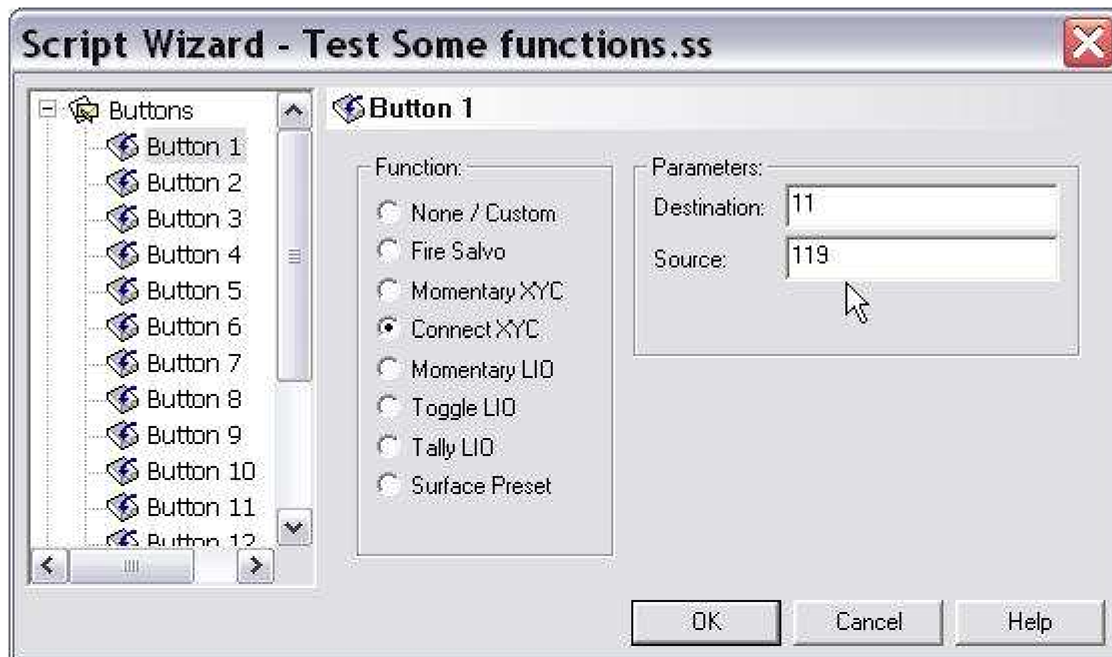
Note: Certain circumstances may cause the software and the GP panel to be out of "sync". "On Line-Connected" reported when in fact you are not connected. When in doubt simply toggle On Line mode OFF and ON.

3.4 - Create a New Script File

Select the Menu item ***File->New*** and the *Script Wizard* opens automatically, once you have specified a file name for the new script.

The *Buttons* list in the scroll pane on left side of the Wizard is where you select which GP panel button you would like to program. Simply click on the button name to select it.

The right side of the Wizard is where you select a function for the selected button. Go ahead and click through the various Functions. You will notice that the Parameters field will display various data entry fields depending on the function selected. Parameters are usually integers that correspond to signal ID numbers or Salvos as configured in the XPoint software.



3.5 - Script Wizard Button Functions

The following functions may be mapped in any combination to the GP-xx buttons. Note that in some cases a button may perform actions on both the Press and Release of the switch. The Help file includes details; go to ***Contents>Script Wizard>Button Properties*** for more information.

Function Summary

None/Custom – select this if you are not using the button or will write a custom script for the button.

Fire Salvo –select this to fire a Salvo created in XPoint. Enter the Salvo's Index number in the Press and Release Parameters fields. Salvos are created in XPoint and are simply a stored set of one or more routes and/or disconnects. The first Salvo in the XPoint Salvo list is index 1, second in the list is 2, etc. You can have a different Salvo fire on both the Press and the Release of the switch. Use this function when you need multiple "patches" to happen simultaneously, like switching speaker and HP feeds to a shared talk studio.

Momentary XYC - XYC stands for X-Y Crosspoint- this option is used to momentarily interrupt a destination with a new source. Useful for talkback or EAS, the interrupted Destination reverts back to previous Source. Enter the Destination and Source signal ID numbers from your XPoint configuration. Just mouse over the signal name in XPoint to get its number

Connect XYC – this function will make a one time X-Y Crosspoint route. Enter the Destination and Source signal ID numbers from your XPoint configuration.

Momentary LIO –this function will trigger a logic connection ON. This function requires mapping of the LIO in *Device>Properties* - see Section 4 or Help File for specific details.

Toggle LIO – this function with toggle the LIO state ON/OFF with each press of the button. This function requires mapping of the LIO in *Device>Properties* – see section 4 or Help for specific details.

Tally LIO – not available at this time – future: use this to turn the button into an indicator lamp. The LED in the button will light when the logic condition is met.

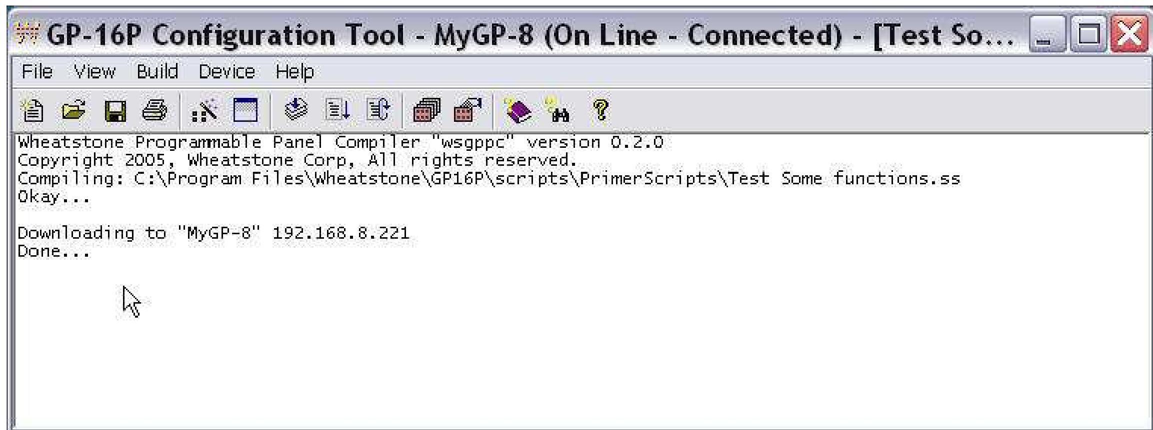
Surface Preset – use this to take a Preset on a Wheatstone control surface. You need to specify two parameters for this function:

> *Surf:* - is the surface ID specified in the *Device-> Properties* form. Surface ID numbers are mapped to the GP-xx panel using the menu choice *Device->Properties….* Enter an IP addresses for each surface the panel needs to talk to. Entering 1 for the ***Surf:*** parameter will cause a button to talk to IP address associated with ***Surface 1:*** in the list.
>
> *Preset*: -this parameter is case sensitive - Name of the Preset located on the surface. Some surfaces like the G4 have only push buttons, so index numbers 1-4 map directly to the buttons.

3.6 - Compile the Script

Once you have mapped functions to the buttons you are ready to compile the auto-generated Script Wizard code and download it to the GP-xx panel. To compile, select the ***Build-> Compile and Download*** menu choice.

If successful you will see the following feedback on the screen.



3.7- Starting the Script

When the *Compile* and *Download* processes are completed, you will be prompted to Start the new script – choose Yes to start it. Note that once the code is transferred into the GP-xx non-volatile flash memory, it will boot your Script every time the unit is powered up.

3.8 - Testing

Now its time to see the results of the code you have downloaded to the GP-xx panel. Obviously, you can go to the button location and listen and watch for changes as you press the buttons. An easy way to check many functions is to have the XPoint software running while you press the buttons. If you align the grid so that the signals of interest are visible, you can watch as temporary, or static connections are made. You can even watch as Salvos are taken to see multiple connections change. This is handy when de-bugging scripts too. Because the button code is portable, you can develop multiple scripts using a single button panel in your office or rack-room, verify the code works as intended, and then download the working scripts to the designated panels in a Studio or Control room.

3.9 - Reviewing the Script Wizard Code

You can use the Script Editor to see the auto-generated (AG) code produced by the Script Wizard. To view the code select menu item ***View->Script Editor…***

Here is a sample Script and its code descriptions:

Wizard code starts here >
// precedes all Comments.

Button types are listed as >
Comments

Define variables >

Startup action calls function to set Button 6 LED on power up.

Action sets LED 6 to ON or OFF depending on the state of LIO 6 on power up.

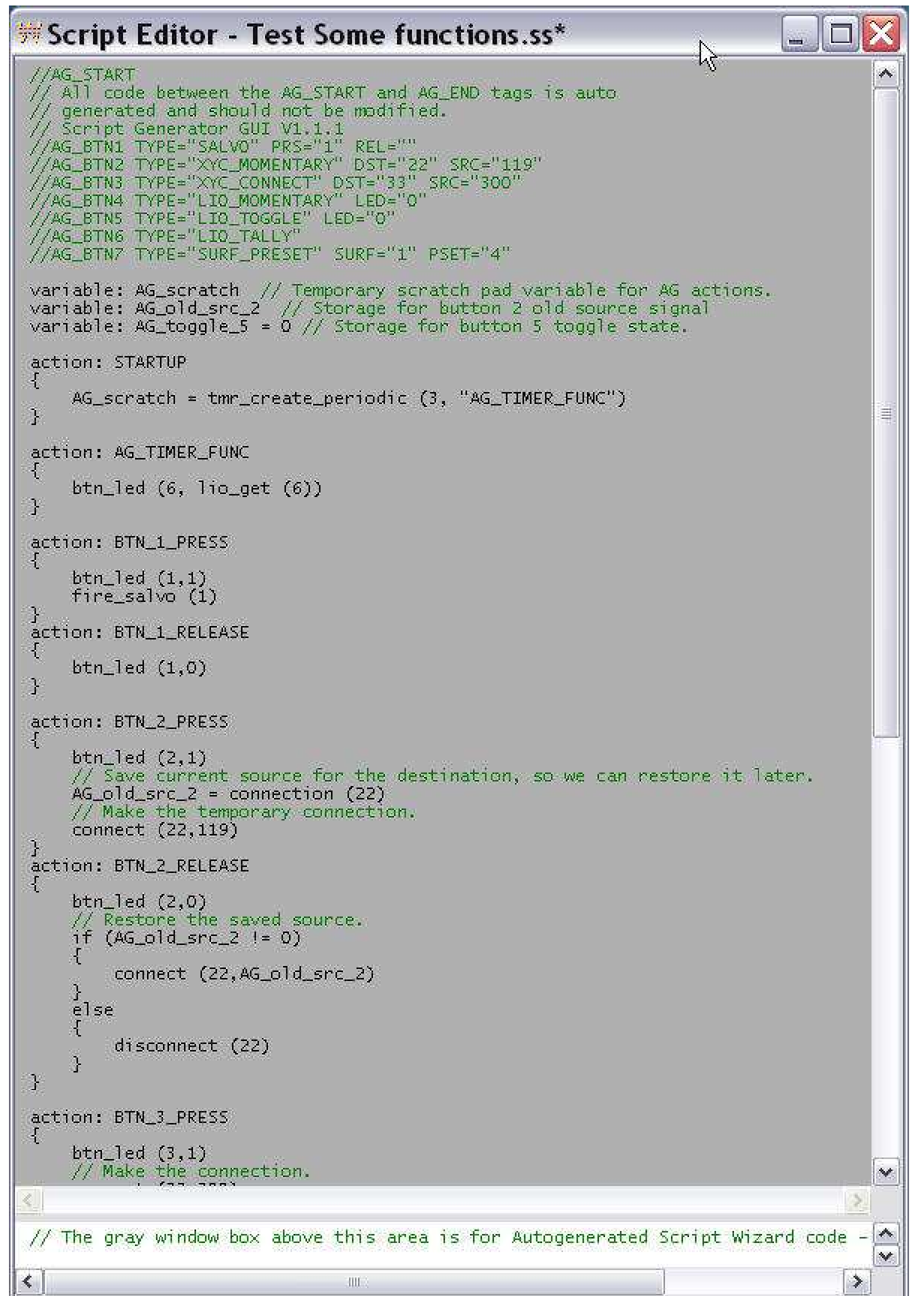
Action lights BTN1 LED and Fires Salvo1.

Action clears BTN1 LED on BTN1 release.

Action lights BTN2 LED, Stores Source ID patched to Dest 22, then connects Source 119 to Dest 22.

Release Action clears BTN2 LED then restores the stored Source IF >0 to Dest 22.

A disconnect is performed if the stored Source is = to zero.



Script Editor - Test Some functions.ss*

```
//AG_START
// All code between the AG_START and AG_END tags is auto
// generated and should not be modified.
// Script Generator GUI V1.1.1
//AG_BTN1 TYPE="SALVO" PRS="1" REL=""
//AG_BTN2 TYPE="XYC_MOMENTARY" DST="22" SRC="119"
//AG_BTN3 TYPE="XYC_CONNECT" DST="33" SRC="300"
//AG_BTN4 TYPE="LIO_MOMENTARY" LED="0"
//AG_BTN5 TYPE="LIO_TOGGLE" LED="0"
//AG_BTN6 TYPE="LIO_TALLY"
//AG_BTN7 TYPE="SURF_PRESET" SURF="1" PSET="4"

variable: AG_scratch  // Temporary scratch pad variable for AG actions.
variable: AG_old_src_2  // Storage for button 2 old source signal
variable: AG_toggle_5 = 0 // Storage for button 5 toggle state.

action: STARTUP
{
    AG_scratch = tmr_create_periodic (3, "AG_TIMER_FUNC")
}

action: AG_TIMER_FUNC
{
    btn_led (6, lio_get (6))
}

action: BTN_1_PRESS
{
    btn_led (1,1)
    fire_salvo (1)
}
action: BTN_1_RELEASE
{
    btn_led (1,0)
}

action: BTN_2_PRESS
{
    btn_led (2,1)
    // Save current source for the destination, so we can restore it later.
    AG_old_src_2 = connection (22)
    // Make the temporary connection.
    connect (22,119)
}
action: BTN_2_RELEASE
{
    btn_led (2,0)
    // Restore the saved source.
    if (AG_old_src_2 != 0)
    {
        connect (22,AG_old_src_2)
    }
    else
    {
        disconnect (22)
    }
}

action: BTN_3_PRESS
{
    btn_led (3,1)
    // Make the connection.
```

```
// The gray window box above this area is for Autogenerated Script Wizard code -
```

4 Configuring Device Properties

Some applications may require the GP-xx panel to talk to control surfaces or interact with certain signals that have logic functions mapped to them. For instance you may wish to take a Preset or turn a channel ON and OFF on a surface. You might also wish to use the GP-xx panel at a talent microphone location in a studio. These applications require you to “tell” the GP-xx panel some information about the surface and logic signals. This is what the Device Properties form is for.

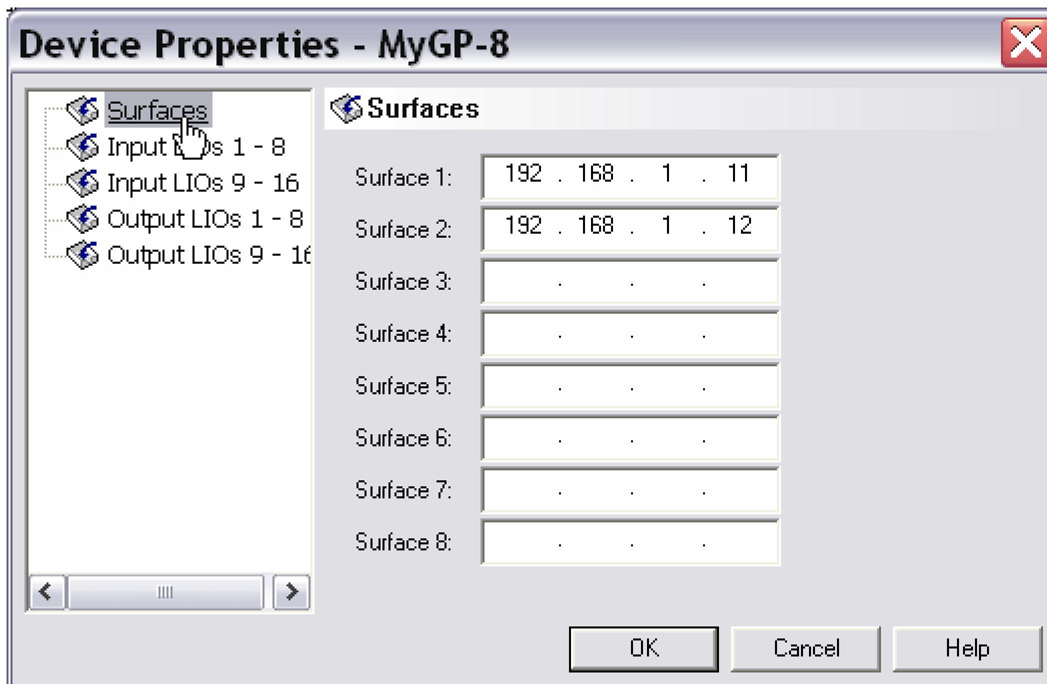
4.1 - Surface Configuration

If you are using your GP-xx button panel to interface with a Wheatstone surface, you will need to setup your GP-xx button panel with a list of each surface to which it will communicate.

The setup steps only need to be performed once since the setup information will be stored in the button panel's Flash memory and on your PC. Select the device which you wish to setup, then use the "Device Properties" dialog box to specify the surface IP addresses.

4.2 - Starting the Device Properties Dialog

Start the Device Properties Dialog by clicking on **"Device"** & **"Properties..."** on the main window menu, or by pressing the **<CTRL>-P** keys. The following dialog box will appear. Select "**Surfaces**" in the tree on the left side of the dialog box.



You may specify up to eight surface IP addresses. The top IP address corresponds to surface "1" in the surface interface functions. This address will be used when you specify a "surfid" of "1" in any of the surf_xxx functions within your scripts or when you select the "Surface Preset" option

in the Script Wizard. The second IP address from the top corresponds to surface "2", the third from the top is surface "3", etc. Unused surfaces should be left blank.

Note: The controls will be disabled if you are **not** connected to the GP-16P device. If you are disconnected, you are actually looking at the device properties which are stored on your PC's hard drive. These properties may not truly reflect the properties of your device, if the device has been more recently configured from another PC.

4.3 - LIO Configuration

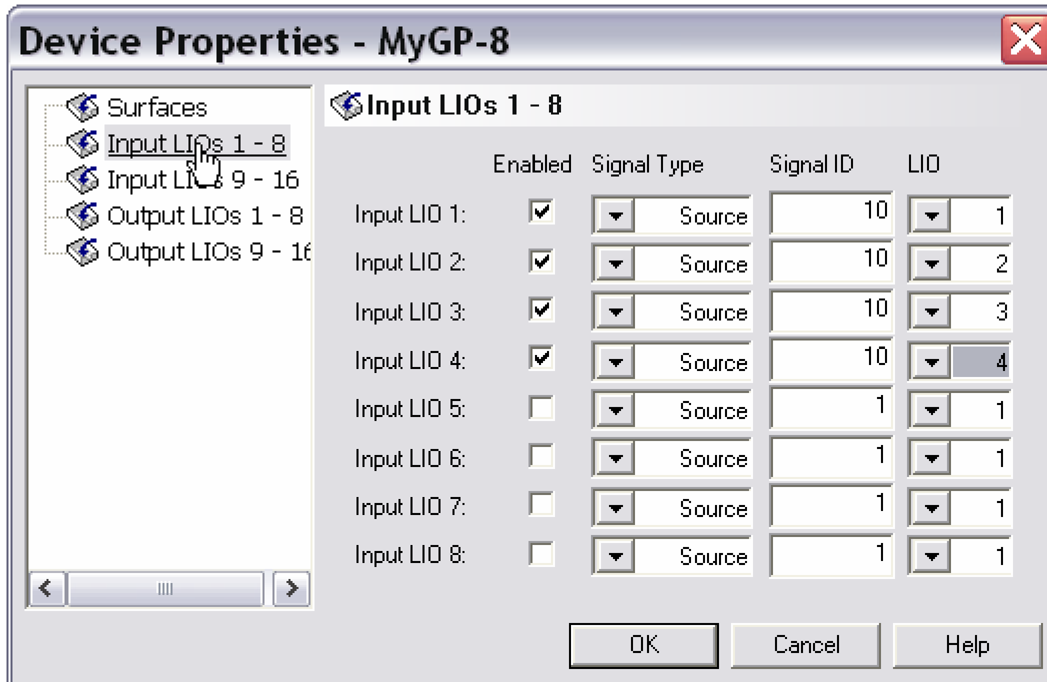
If you are using your GP-16P button panel to interface with Logic I/O on your Wheatstone router, you will need to setup your GP-16P button panel with a list of each LIO which it will access.

- LIO configuration done here maps pre-defined XPoint signal logic to GP buttons for Script Wizard programming. LIO1 maps to button1, LIO2 maps to button2, etc.
- Custom scripts can access any of the 16 input or output LIO's.
- Virtual LIO's may be created so you don't eat up any physical logic i/o. Add a phantom logic card to a rack in XPoint. See section 5 or Help for details.

The setup steps need to be performed each time you create a new script for a panel using mapped logic. For example, Panel 1 is for the Host mic and uses Source ID 10, Panel 2 is Guest 1, Source 11. Update *Device Properties* before you script the Guest panel.

4.4 - Starting the Device Properties Dialog

While CONNECTED-ONLINE - Start the Device Properties Dialog by clicking on **"Device"** & **"Properties..."** on the main window menu, or by pressing the **<CTRL>-P** keys. The following dialog box will appear. Select one of the LIO items in the tree on the left side of the dialog box.



You may map up to 16 Input LIOs and 16 Output LIOs, one for each switch on a GP-16.
Input LIOs correspond to Logic I/O values which are fed IN to the router matrix. Typical types of input LIOs would be from switches like ON,OFF, Cough, Talkback, remote logic signals

associated with a microphone source. In a discrete hardwired system these signals would typically come from a button on the announcer's desk then be fed into an input logic line on an LIO card in your audio router.

Custom scripts for your GP-16P can drive input LIOs using the lio_set() function.

Output LIOs correspond to Logic I/O values which are fed OUT of the router matrix. Typical types of output LIOs would be machine start, machine stop, and ON and OFF tally logic signals to drive remote panel switch LED's associated with a microphone source. In a discrete hardwired system these signals would typically come from an output logic line on an LIO card in your audio router then feed to a logic line on your automation system or to a switch's LED.

In your GP-16P you can read output LIOs using the lio_get() function.

The first input LIO corresponds to LIO id "1" in the lio_set() function, the second to LIO id "2", etc.. The first output LIO corresponds to LIO id "1" in the lio_get() function, the second to LIO id "2", etc..

Note: The controls will be disabled if you are not connected to the GP-16P device. In this situation you are looking at the device properties which are stored on your PC's hard drive. These properties may not truly reflect the properties of your device, if the device has been more recently configured from another PC.

4.5 - Design Philosophy

During the design of the GP-16P we went back and forth on the merits of making the LIO definitions a device property and using property table indexes in the script function calls vs. specifying the LIO definitions directly in the script functions. We felt that the first approach would provide greater value in that if your installation contains several GP-16P panels with similar functionality, you can use one script for all of the GP-16P button panels and just modify the device properties of each GP-16P.

Note: When you specify input LIOs for the GP-16P, you will typically select a logic line which is also configured as an input LIO in the XP GUI program. You can point one of the GP-16P input LIOs at a logic line which is configured as an output LIO in the XP GUI, and the GP-16P will happily drive it. The negative side of doing this is, there might also be another GP-16P or a physical logic card driving the same output LIO. The router has very extensive rules to arbitrate who is driving output logic. These rules are basically bypassed, if you adopt the mixed direction approach. It's much safer to define a new signal which has an input logic line, drive the new signal's input logic line with the GP-16P, then connect the new signal to the signal which has the output logic and let the router apply it's rules to the signal routing.

Note: When you specify output LIOs for the GP-16P, you will typically select a logic line which is also configured as an output LIO in the XP GUI program. You can point one of the GP-16P output LIOs at a logic line which is configured as an input LIO in the XP GUI, and the GP-16P will happily read it.

5 LIO Example Using Device Properties

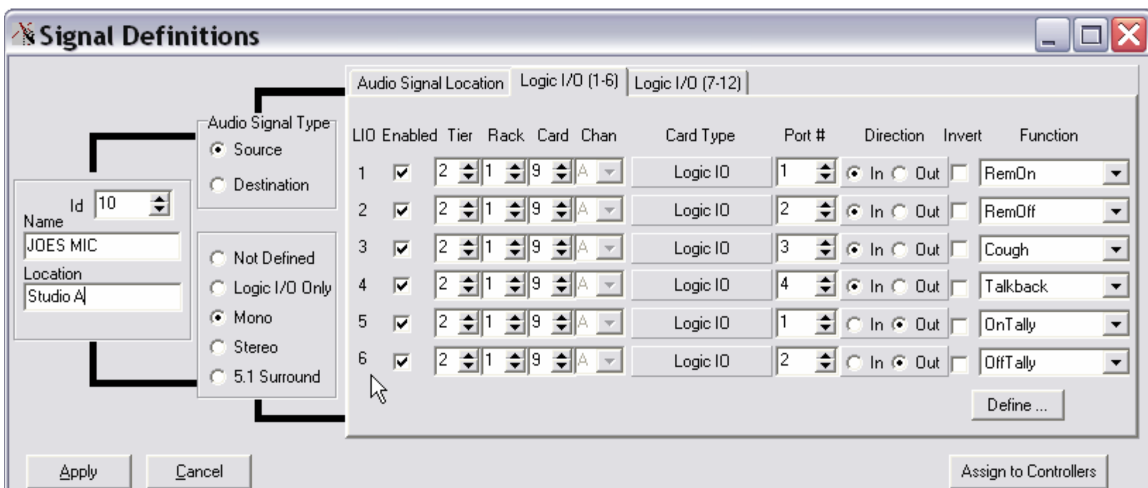
Before we get on with the following example you should understand that there are two primary ways to approach remote control of a surface channel using the GP-xx panel. You can use a custom script to control a *specific fader channel* on a surface or you can use the Device Properties to “point” the GP panel to a *specific source signal*, which has been configured in XPoint with logic associations. The difference may appear to be subtle but it really dictates how your overall script will be written. The former uses surface functions in a custom script while the latter uses the Script Wizard.

This Help File example describes a relatively complex method using the Device Properties LIO mapping feature. The method requires mapping LIO (logic in/out) resources to a virtual Logic card in XPoint, then these pre-defined LIO’s are mapped to physical GP-xx panel buttons using the Device Properties form in the GP-16P Configuration Tool software. Finally, the Script Wizard is used to generate the script. This approach has two benefits- the resulting script is very clean and the GP-xx panel follows the microphone source signal to whichever surface it connected to.

For the sake of this example, let's assume that we have a microphone source named "JOES MIC" in our system. We will be placing a GP-16P button panel next to the announcer, Joe. We would like to use some of the GP-16P buttons to provide Joe with remote ON/OFF, Cough and Talkback capability. We would also like to have the GP panel’s ON/OFF button LED’s follow the console’s fader status.

5.1 - Configure the Source Signal in XPoint

The first thing we need to do is configure the "JOES MIC" source signal with some virtual LIO signals to perform these functions. The following figure shows how the LIOs will be defined for "JOES MIC" in the XPoint GUI.



Defining a virtual LIO signal only differs from defining a real physical LIO signal in that we do not require real physical hardware for the I/O. Since the I/O is virtual and our GP-16P is

emulating the hardware we will point the LIO associated with JOE's mic to an LIO card which is not actually populated in the router rack. You must add an LIO card to your "Rack Defs" dialog box, since this is the only way to reserve the "Tier, Rack & Slot" numbers used for routing the logic. But the slot which you allocated should **not** have a real LIO card inserted into it.

Things to take note of in this diagram are the signal number and the LIOs for each logic function. The signal number and the LIO number will come into play when we configure the logic I/O for the GP-16P.

Item to Note	Type	ID
JOES MIC Signal	Source	10
Remote ON LIO	In	1
Remote OFF LIO	In	2
Cough LIO	In	3
Talkback LIO	In	4
On-Tally LIO	Out	1
Off-Tally LIO	Out	2

5.2 - Configure the GP-16P LIOs

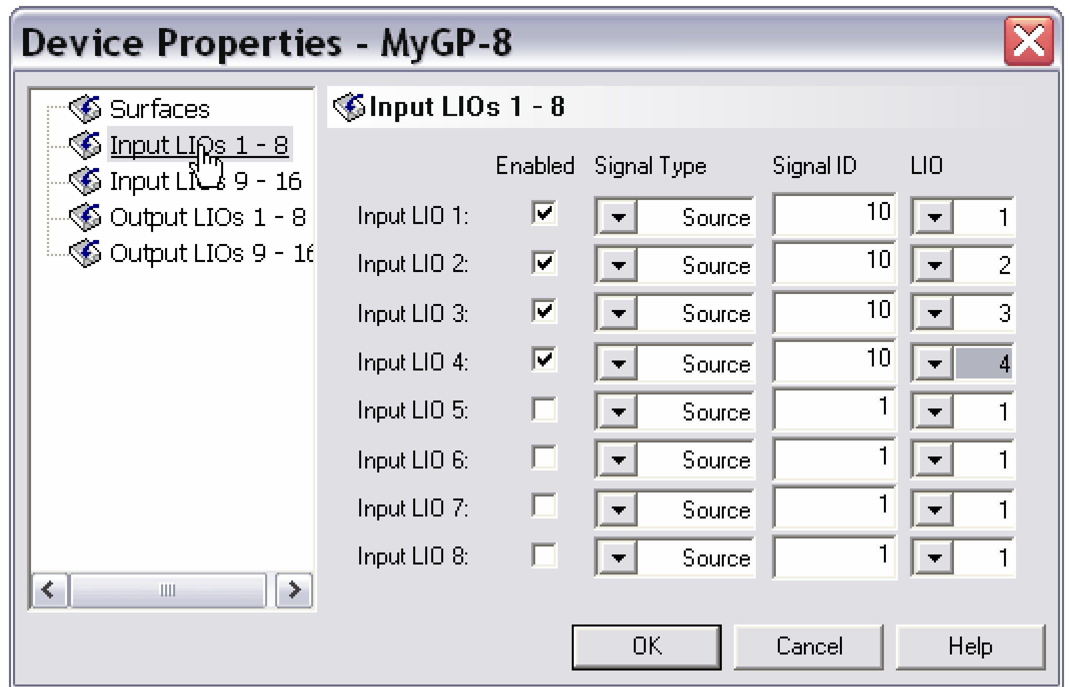
Let's assume that we want to use the first four buttons on our GP-16P to perform these functions.

Button	Function	Details
1	ON	The Remote ON LIO will be triggered when the button is pressed, the button LED will light to indicate that the channel is on air.
2	OFF	The Remote OFF LIO will be triggered when the button is pressed, the button LED will light to indicate that the channel is off air.
3	Cough	The Cough LIO will be triggered when the button is pressed and released when the button is released, the button LED will light to indicate that the button is down.
4	Talkback	The Talkback LIO will be triggered when the button is pressed and released when the button is released, the button LED will light to indicate that the button is down.

The Script Wizard assumes a one-to-one correlation between the LIO number in the GP-16P device properties and the auto generated action which the Script Wizard will generate. Therefore, we need to define the LIOs in the device properties in the proper locations for the button functions. The following figures show how we will define our LIO properties in the GP-16P for this example.

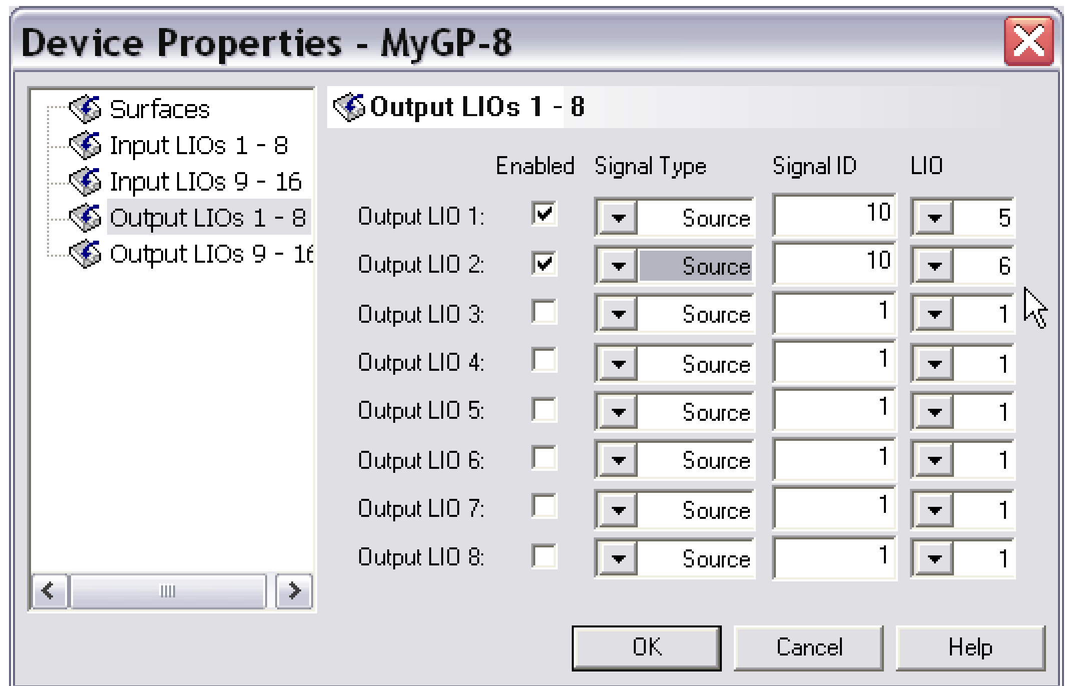
Define the first four input LIOs to match the Remote On, Remote Off, Cough and Talkback LIOs for the "JOES MIC" signal.

Take note that these are configured as "Input" LIOs in the GP-16P since we are sending this logic **into** the router matrix.



Define the first two output LIOs to match the On-Tally and Off-Tally LIOs for the "JOES MIC" signal.

Take note that these are configured as "Output" LIOs in the GP-16P since we are reading this logic **out** of the router matrix.



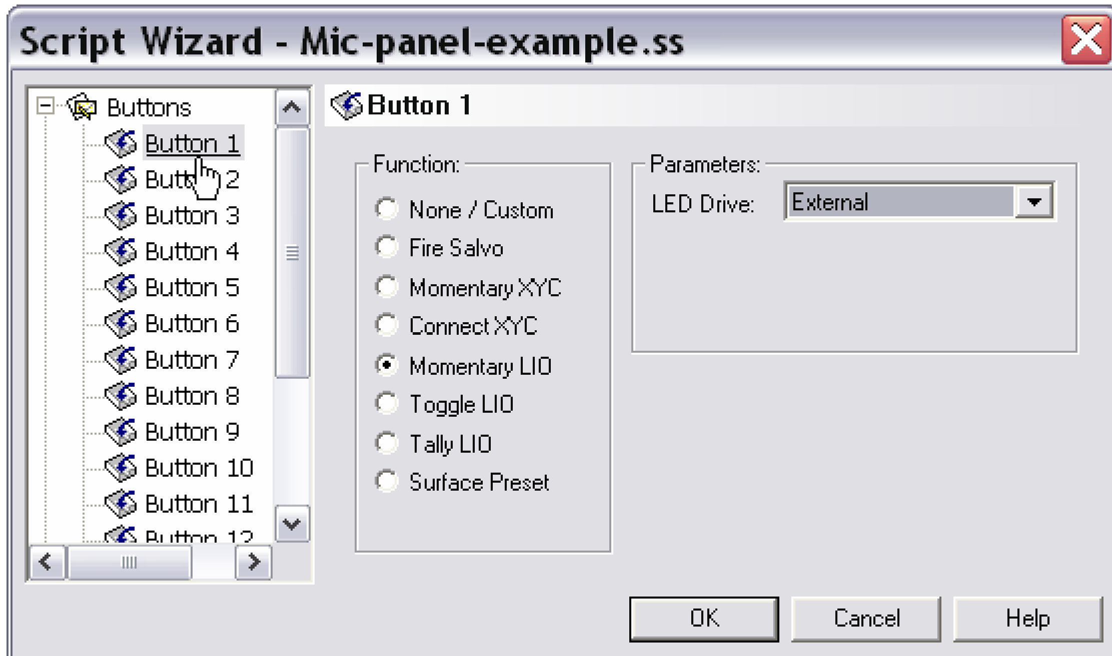
Important Distinctions

- The "Signal Type", "Signal ID" and "LIO" fields are configured to match the values from the XP GUI signal definition dialog box.
- The LIO field value, 1 through 12, is NOT the Logic Card's port number, but the *LIO Enabled* # in the Signal Definitions form.

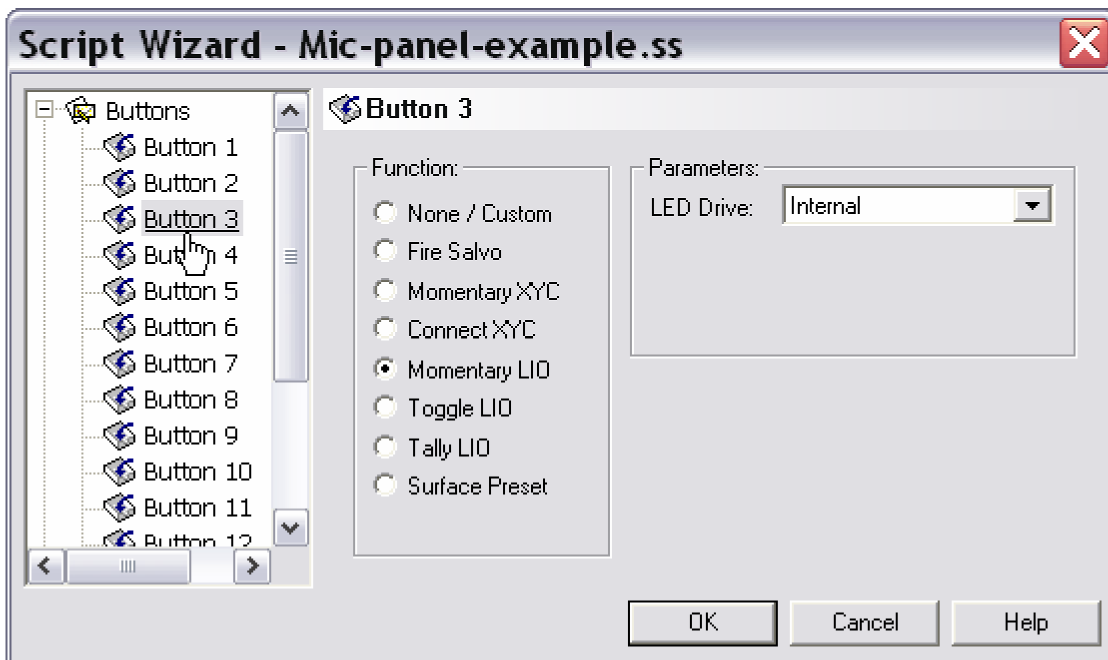
5.3 - Create the Mic Control Script Using Script Wizard

Now we want to use the Script Wizard to generate a script for the GP-xx.

Configure the first and second buttons to be **Momentary LIO** functions with **External** LED drive.



Then configure the third and fourth buttons to be **Momentary LIO** functions with **Internal** LED drive.



5.4 - Reviewing the Script Wizard Code

The following script will be generated. The button 1 & 2 actions simply drive the LIOs and LEDs corresponding to the buttons. A periodic timer drives the button 1 & 2 LEDs with the value read from the LIO corresponding to those buttons. The button 3 & 4 actions simply drive the LIOs and LEDs corresponding to the buttons.

```
//AG_START//AG_START
// All code between the AG_START and AG_END tags is auto
// generated and should not be modified.
// Script Generator GUI V1.1.1
//AG_BTN1 TYPE="LIO_MOMENTARY" LED="1"
//AG_BTN2 TYPE="LIO_MOMENTARY" LED="1"
//AG_BTN3 TYPE="LIO_MOMENTARY" LED="0"
//AG_BTN4 TYPE="LIO_MOMENTARY" LED="0"

variable: AG_scratch  // Temporary scratch pad variable for AG actions.

action: STARTUP
{
   AG_scratch = tmr_create_periodic (3, "AG_TIMER_FUNC")
}

action: AG_TIMER_FUNC //LIO 1 and 2
{
   btn_led (1, lio_get (1))  // get LIO 1 value and light LED1 (ON) if true
   btn_led (2, lio_get (2)) //get LIO 2 value and light LED2 (OFF) if true
}
```

The auto-generated script code for the first two buttons will assert the input LIO while the button is pressed and de-assert the input LIO when the button is released. The button LED will light from the results of the periodic timer query in section above..

```
action: BTN_1_PRESS //mapped as REMOTE ON in Device Properties
{
   lio_set (1,1)
}
action: BTN_1_RELEASE
{
   lio_set (1,0)
}

action: BTN_2_PRESS //mapped as REMOTE OFF in Device Properties
{
   lio_set (2,1)
}
action: BTN_2_RELEASE
{
   lio_set (2,0)
}
```

The auto-generated script code for the third and fourth buttons will assert the input LIO while the button is pressed and de-assert the input LIO when the button is released. The button LED will light to indicate that the button is down.

```
action: BTN_3_PRESS //mapped as COUGH in Device Properties
{
   btn_led (3,1)
   lio_set (3,1)
}
action: BTN_3_RELEASE
{
   btn_led (3,0)
   lio_set (3,0)
}

action: BTN_4_PRESS //mapped as TALKBACK in Device Properties –puts surface fader in
                              //CUE speaker
{
   btn_led (4,1)
   lio_set (4,1)
}
action: BTN_4_RELEASE
{
   btn_led (4,0)
   lio_set (4,0)
}

//AG_END
```

Note:

In this example we have seen how the Script Wizard associates a button with the corresponding LIO from the LIO definitions in the Device Properties dialog box. This one-to-one correspondence is only a limitation of the Script Wizard. If you are writing a custom script you may access any LIO defined in *Device Properties* from any action or subroutine.

5.5 - Beyond the Script Wizard

The Script Wizard is a nice way to get some fundamental features up and running quickly and will suffice for many broadcast applications. Certain applications with multiple panels in which actions are triggered under Boolean conditions are a bit more complex and will probably require some head scratching and, you guessed it –a custom script.

6 What is the Script Editor?

The Script Editor is a specialized text editor built into the GP-16P Programming tool. This editor provides a convenient way to write custom scripts and also view Script Wizard code.

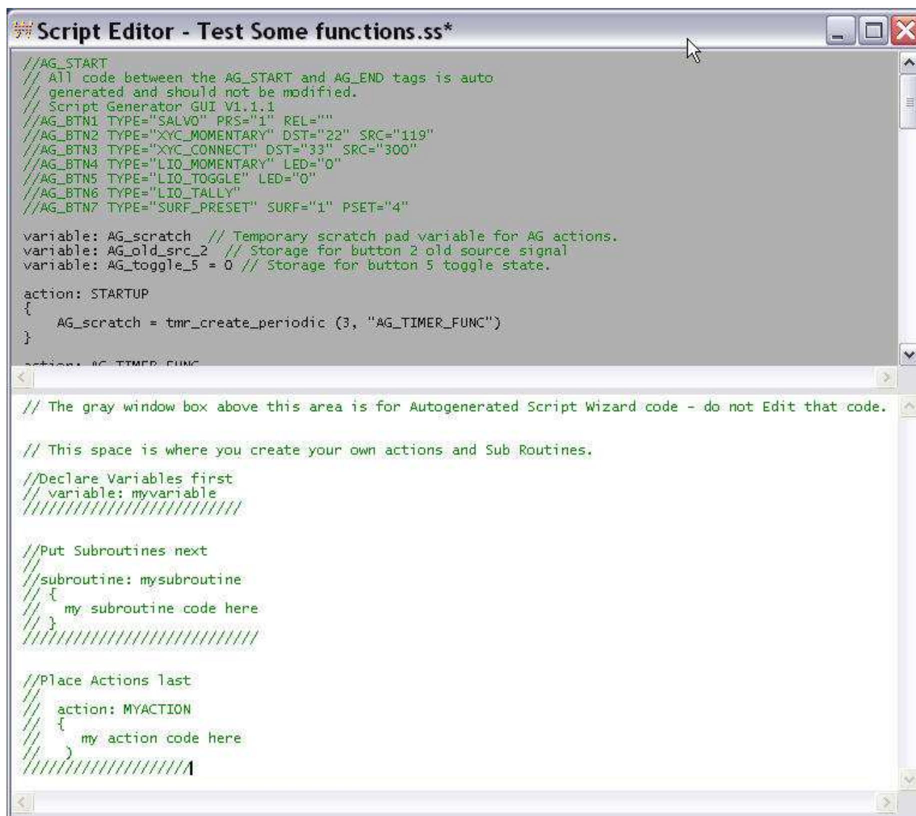
GP-xx scripts are actually specially formatted text files saved with a ".ss " file extension.

.

The Script Editor automatically separates the Script Wizard code from your custom code by dividing the file into two panes – the top "read only" pane has a gray background and houses the AG or auto generated Script Wizard code. The bottom pane is the editable text editor pane used for writing your own scripts.

6.1 - Script Editor Features

- Script Wizard code is separated and displayed in a "read only" pane.
- Script text is displayed in a "context sensitive" color scheme with comments in green, and keywords in blue.
- Standard text select, cut, copy, paste, undo, and redo functions.
- Compiler error finder jumps the cursor to problem line when the reported error is clicked.



6.2 Third Party Editors

Scripts may also be opened, written, and edited in a programming oriented editor but care must be taken to be sure that the file structure, formatting, and script syntax is maintained. Avoid using generic text editors like Notepad or Wordpad for script creation. You will know right away at Compile time if there is a problem

If you plan on doing a lot of scripting you might consider using a third party programming editor. Notepad++ is a nice freeware editor. When you open a GP script in Notepad++, you can choose a "Language" skin, like "Flash actionscript", that will give you line numbers and a context sensitive text color scheme. You will still have to open the file in the GP16P tool before you compile – be sure to save the file in the editor first.

You can do an Internet search for "Notepad++" to download this editor.

7 Creating Custom Scripts

A good way to learn how to write custom scripts is through experimentation - so we will open a custom script and examine the format and syntax of the file. Then feel free to edit button behavior and add features. You can also use the Script Wizard to generate code to see specific function examples, then copy and paste into a new file for further experimentation.

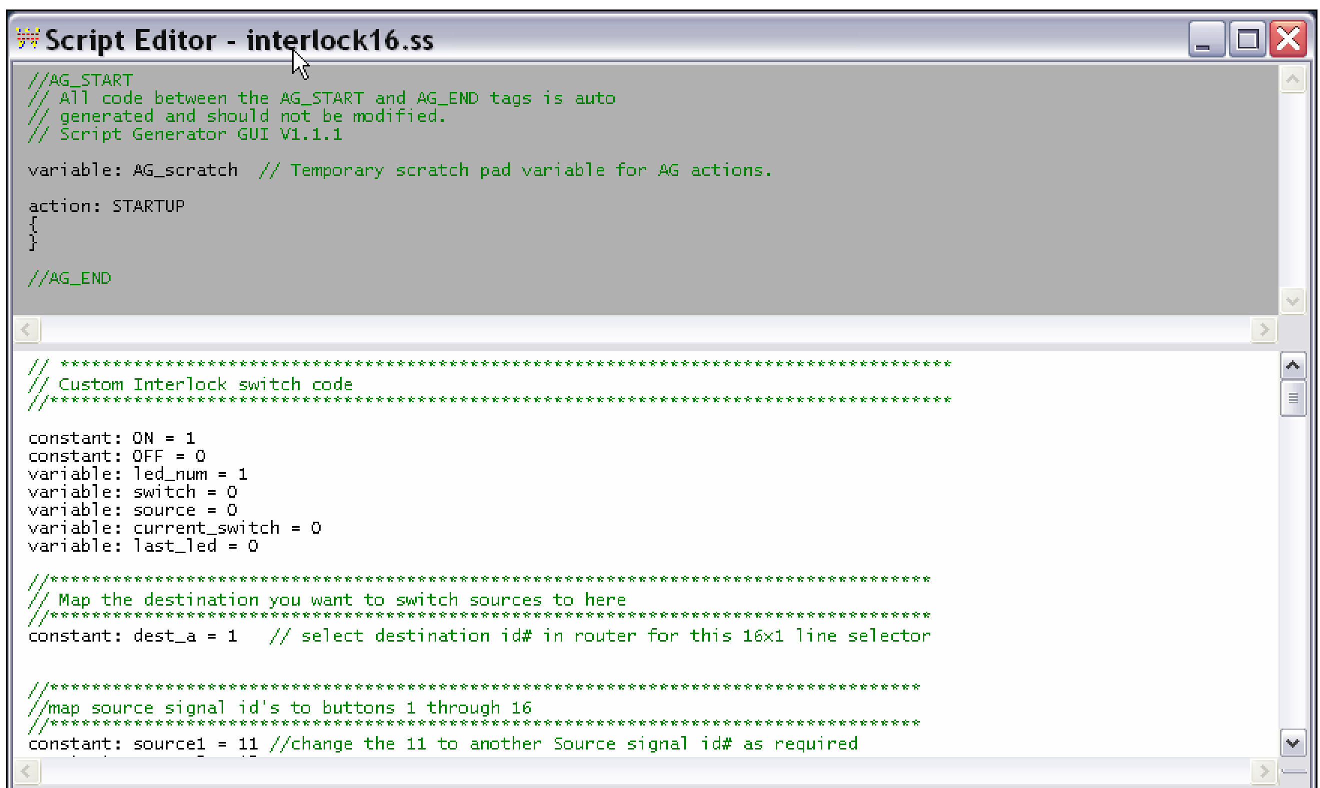
7.1 - Getting the Example File

The example script file, interlock16.ss, is located in Appendix A of this document and may be copy and pasted into the Script Editor user's window. Copy and paste details are located in Appendix A.

7.2 - Example Script Design

The custom script used in this example is designed to act as an "interlocked" source selector with latching LED indicators. Each button will "patch" an audio Source to a common Destination and light the button's LED on the panel. The button's LED must be "latched" ON so the operator knows which button is currently selected. "Interlocked" simply means that with each button press the previous source and LED are disconnected and are replaced by the current button press. In logical terms the 16 switches and LED's are "exclusive OR'd".

Open the Script Editor by choosing ***View->Script Editor…***



7.3 - Auto-generated Script Components

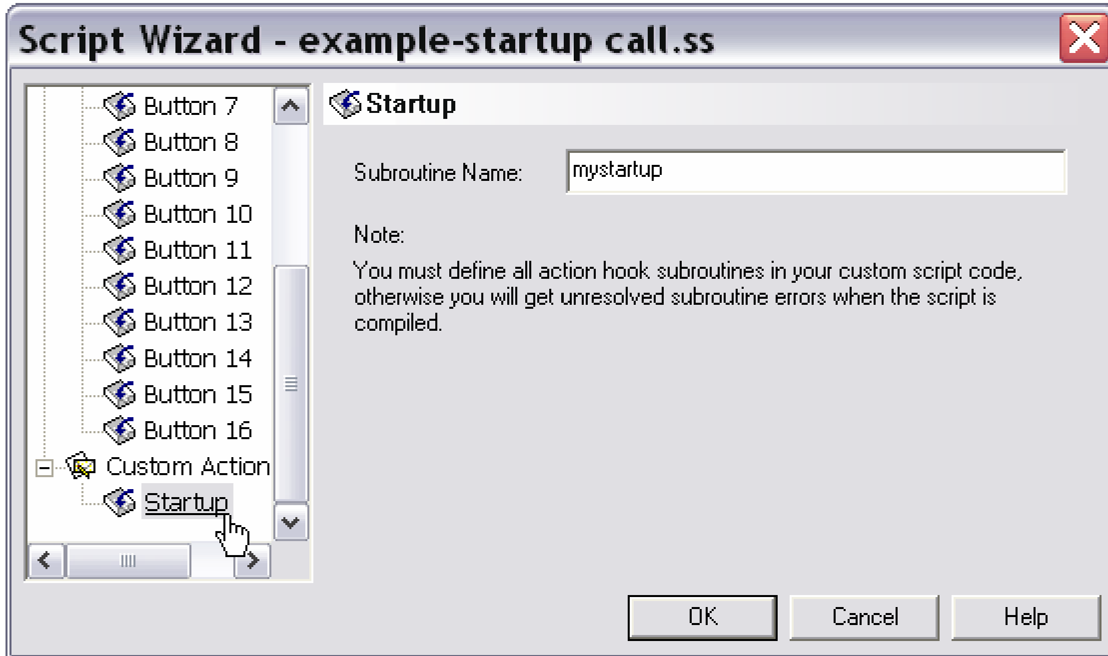
Notice that the first section of the custom script has a few lines of auto-generated code. These are minimum startup lines and must not be altered or deleted.

```
//AG_START
// All code between the AG_START and AG_END tags is auto
// generated and should not be modified.
// Script Generator GUI V1.1.1

variable: AG_scratch  // Temporary scratch pad variable for AG actions.

action: STARTUP // The startup action is empty because blank a new file has no Start requirements.
{                    // You can use the Script Wizard to point this startup to your own startup subroutine.
}                    //See the next section for details.

//AG_END
```

7.4 - Custom Start up Subroutine

Let's digress for a moment- sometimes you might want your panel to startup in a special state prior to any button actions. Or perhaps the LED's in your design are being driven from remote logic states and you'd like to synchronize them on power-up of the GP –xx panel.

Use the Script Wizard's Custom Action Startup dialog to point to your startup subroutine. In the case below we will call "mystartup" subroutine when the GP-panel powers up.



Resulting code with new subroutine and some structure comments added.

```
//AG_START
// All code between the AG_START and AG_END tags is auto
// generated and should not be modified.
// Script Generator GUI V1.1.1
//AG_HOOK TYPE="STARTUP" ACTION="mystartup"

variable: AG_scratch  // Temporary scratch pad variable for AG actions.

action: STARTUP
{
   call mystartup ()
}

//AG_END
//****************************************************************************
// Custom Script starts here
//***************************************************************************
//Define global variables first
//Define Constants next
//Define actions and subroutines last

// The subroutine “mystartup” is called by the AG code's STARTUP action
//when first powered up or the Script is re-started.

subroutine:mystartup
{
//put your startup code here

}


// Custom Script ends
```

7.5 - Example Script Structure

Now back to the Example interlock16.ss script file. The first thing you will notice in the example script is a comment. Comments are extremely useful as they help you and anyone else working with the script understand and decipher what is going on. Comments must always start with a double forward slash

```
//this is a comment line
```

Comments are ignored by the compiler and can contain any characters. You can have as many comments as you’d like in your script.

Scripts must follow a certain format in order for the compiler to evaluate it correctly. The example script follows this format:

- AG Start code – auto-generated code from the wizard and a basic startup action.
- This code must be present even if you plan on scripting all of the button functions and generally should not be modified. This code is only displayed in the Script Editors top window. The top window does not allow editing.

- Constants and variables - define all your constants and global variables first. Example constants are Source or Destination signal ID numbers, words that make your script easier to read and write like ON- OFF, LED5, etc. Constants are fixed and never change during run time. Variables may be local or global in scope and may be modified during runtime.

- Global variables are listed at the top along with constants and are “visible” anywhere in the script.

- Actions and Subroutines- next comes the main components of your script. It does not matter which order you put these in but it makes sense to keep all button actions together for readability.
- Local variables are defined within the curly braces of an action or subroutine and are only “visible” within that action or subroutine

Let’s look at the example code in sections.

7.6 - Example Script –Variables and Constants

The example script needs to know which switch is pressed and when to light its LED. We also have to map the destination we want to route to and define the sources to be switched.

You seldom know all the variables your script will require when you begin, so just add them here at the top as you go. It makes sense to group certain variables according to how they are used in the script. This can make reading and deciphering the script easier now and when you have to edit it a year from now!

```
// Custom Interlock switch code
//***************************************************************************************
*
variable  //intentional error - no colon after the word variable -no variable name
constant: ON = 1
constant: OFF = 0            // Constants can be mixed in with variables as you see fit.
variable: led_num = 1
variable: switch = 0
variable: source = 0
variable: current_switch = 0
variable: last_led = 0
```

Comments added to the Constants section help readability. Notice how the Destination and Sources are defined as constants. These signal ID numbers could have been "hard coded" as numbers in the Action section but are easier to modify in the future by listing here. Additional comments could include the Source signal names in XPoint or the constant names could even be the Source signal names – whatever makes the most sense to you the programmer.

```
//************************************************************************************
// Map the destination you want to switch sources to here
//************************************************************************************
constant: dest_a = 1   // select destination id# in router for this 16x1 line selector

//***********************************************************************************
//map source signal id's to buttons 1 through 16
//***********************************************************************************
constant: source1 = 11 //change the 11 to another Source signal id# as required
constant: source2 = 12
constant: source3 = 13
constant: source4 = 14
constant: source5 = 15
constant: source6 = 16
constant: source7 = 17
constant: source8 = 18
constant: source9 = 109
constant: source10 = 110
constant: source11 = 111
constant: source12 = 112
constant: source13 = 113
constant: source14 = 114
constant: source15 = 115
constant: source16 = 116 //change the 116 to another source signal id# as required
```

7.7 - Example Script – Subroutines

The example script uses two subroutines – one to handle the switch presses and one to store the last switch pressed so it's LED can be turned OFF on a subsequent switch press. Note that a custom startup routine was not included. Try writing a startup subroutine that figures out which source is currently feeding "dest_a" and then light the appropriate button's LED.

The first subroutine – handle_sw_press - is called by the Button Actions defined at the end of the Script. Button Actions "pass" two variables, $1 and $2 to this subroutine.

This subroutine:

- Modifies the value of "switch" to equal $1 and "source" to equal $2.
- Turns OFF the previously selected switch's LED.
- Calls subroutine to store the currently selected switch number.
- Connects the currently selected source.
- Lights the LED in the currently selected switch.

This subroutine includes a "*Print*" statement to print a message to a Telnet window –please see the Script de-bugging section for details on using *Print* and Telnet.

```
//**********************
// Subroutines
//**********************

subroutine: handle_sw_press //This subroutine does most of the work.
                          //It receives switch# and source info from the button
                          //press actions.
{
  switch = $1        // $1(reads "string one") is the switch number passed here when subroutine called by
                          //  action.
  source = $2       // $2
  btn_led (last_led, OFF)
  call store_switch (switch)
  connect (dest_a, source)   //dest_a is a fixed destination defined above as a constant
  btn_led (switch, ON)

  print ("connecting Source ID " # source # " to  Dest " # dest_a # "." )
}
```

The second subroutine simply receives a variable value, "switch", and stores it. Note that this could have been done in the "handle_sw_press" subroutine, but as an exercise this illustrates variable passing and subroutine nesting. Notice that the variable "current_switch" was never used in the script.

```
subroutine: store_switch //
{
  current_switch = $1 // string 1 passed here = value of the "switch" variable in the calling subroutine.
  last_led = $1           // the "last led" variable is set to = the "switch" variable.
}
```

7.8 - Example Script – Actions

For this example each button is given its own *Press* action. *Release* and *Over-press* actions were not required. By putting the "guts" of the script behavior in Subroutines, the Actions are kept simple and straight forward. Each button press uniquely sets the value of "switch" and "source" and then passes those variables to the "handle_sw_press" subroutine.

```
// Button press section
// ****************************
action: BTN_1_PRESS
{
switch = 1
source = source1
call handle_sw_press(switch, source)
}

action: BTN_2_PRESS
{
switch = 2
source = source2
call handle_sw_press(switch, source)
}

action: BTN_3_PRESS
{
switch = 3
source = source3
call handle_sw_press(switch, source)
}

action: BTN_4_PRESS
{
switch = 4
source = source4
call handle_sw_press(switch, source)
}

action: BTN_5_PRESS
{
switch = 5
source = source5
call handle_sw_press(switch, source)
}

action: BTN_6_PRESS
{
switch = 6
source = source6
call handle_sw_press(switch, source)
}

action: BTN_7_PRESS
{
switch = 7
source = source7
call handle_sw_press(switch, source)
}

action: BTN_8_PRESS
{
switch = 8
source = source8
call handle_sw_press(switch, source)
}
```

```
action: BTN_9_PRESS
{
switch = 9
source = source9
call handle_sw_press(switch, source)
}

action: BTN_10_PRESS
{
switch = 10
source = source10
call handle_sw_press(switch, source)
}

action: BTN_11_PRESS
{
switch = 11
source = source11
call handle_sw_press(switch, source)
}

action: BTN_12_PRESS
{
switch = 12
source = source12
call handle_sw_press(switch, source)
}

action: BTN_13_PRESS
{
switch = 13
source = source13
call handle_sw_press(switch, source)
}

action: BTN_14_PRESS
{
switch = 14
source = source14
call handle_sw_press(switch, source)
}

action: BTN_15_PRESS
{
switch = 15
source = source15
call handle_sw_press(switch, source)
}

action: BTN_16_PRESS
{
switch = 16
source = source16
call handle_sw_press(switch, source)
}
```

7.9 - Custom Scripting Suggestions

Before you embark on your scripting expedition take the time to map out the requirements in a spread sheet or note pad. Spending a bit of time in the planning phase can save you some headaches later on and will at least make it easier to stay focused on coding once you start getting deep into it. Also writing out the requirements, (i.e. Turn the xx ON when yy AND zz are true OR nn is NOT true) can be helpful for scripting complex logic statements.

The GP Scripting language is a cross between the C and Basic programming languages. Correct syntax is essential, and is a common source of compiler errors so be sure to carefully check case sensitive spelling, braces and parentheses placement, etc. whenever you get a compiler error.

7.10 - Scripting Router Control

By now you have been exposed to many of the router control functions available. You can review in detail the complete set of Router Functions available by opening the Router Functions section of the Help file. There you will find information on the using the following:

Router Function	Description
Connect	Makes a cross-point connection in the router.
Disconnect	Breaks a cross-point connection in the router.
Lock	Locks a cross-point connection in the router.
Unlock	Unlocks a cross-point connection in the router.
Connection	Queries a destination to find out what source is connected to it.
Locked	Queries a destination to find out if it is locked.
Fire_salvo	Fires a pre-defined Salvo - requires the Salvo ID number.
Find_src	Returns the source signal ID number when you know the source name and location.
Find_dst	Returns the destination signal ID when you know the dest name and location.
Find_salvo	Returns a Salvo ID number when you know the Salvo name.
Lio_get	Returns the current value – 1 or 0 – of a logic signal in the router.
Lio_set	Sets the value – 1 or 0 – of a logic signal in the router.

7.11 - Scripting Surface Control

Control Surfaces may be directly controlled using a built in surface script functions. You can find detailed information on these functions in the Help file's "Surface Functions" section.

Surface Functions can be divided into two groups. The first set of basic functions control the rudimentary tasks of taking a surface preset, getting a fader's ON status, and turning a fader channel ON. The second "advanced" set allows you to utilize the Automation Controller protocol built into each surface.

7.12 - Basic Surface functions

These functions may be used directly in your script and require a minimum amount of scripting knowledge.
surf_take_preset – takes an "Event' stored on a surface. The surface ID parameter is an index into the surface list entered in the Device Properties form.

surf_get_ input_on – returns the channel ON status; 1= ON, 0 = OFF.
surf_set_input_on – turns a channel ON or OFF.

7.13 - Advanced Surface Functions

These functions require just a bit more programming knowledge to implement correctly. The function “surf_talk” is very powerful because it allows you to use all of the surface’s Automation Control Interface (ACI) command set. The automation protocol is ASCII based which makes it easy to incorporate ACI commands using the built in surface functions. Virtually every switch, fader level, knob settings, etc. is accessible. The ACI commands are available on an “as needed” basis for Wheatstone customers. Please contact a Customer Support representative for details on acquiring this information.

surf_talk – use this to send ACI commands to a surface.
surf_reply – use this to retrieve the last reply received from a surface.
surf_string- use this to parse a reply string.

7.14 - Example surf_talk Commands

If you are reading this then your curiosity must be piqued so here are a couple of examples of the syntax required for use with surf_talk.

surf_talk (1, “INPUT:7|FADER:192”) // sets fader 7 to 0dB on surface 1.

surf_talk (2, “INPUT:4|ON:0”) // turns channel 4 OFF on surface 2.

surf_talk (3, (“INPUT:5|CUE:1”) //puts fader 5 in CUE on surface 3.

The Surf ID used in the examples above comes from the list of Surfaces defined in Device Properties.All of these ACI commands generate replies from the surface that may be stored, parsed, and acted upon in your script. Fader values fall into the range of 0-256. Note that nominal dB level conversions to integers suitable for use with “surf_talk” vary by surface type and may be calculated using special set of equations, which are available on request along with the ACI commands.

8 GP16P Scripting Language Overview

The following Script Language overview may be found in the GP-16PConfiguration Tool's Help file. The Overview and Structure sections are included for reference and will give you an idea how a script is built.

Please refer to the Help File for specific details on writing Statements, Boolean Expressions, etc.

The scripting language used to define virtual machine instructions for the programmable button panel is a very simple language to learn. If you are familiar with C or Basic or any number of any other languages you should feel at ease writing scripts for the GP16P in no time.

8.1 - Case Sensitivity

Everything in a script file is case sensitive. The identifiers "xYz" and "xyz" are not equivalent.

8.2 - Comments

A comment starts with two forward slash characters. Once a comment starts all characters are ignored until the end of the current line. A comment can also start with /* and end with */. The following example shows some comments.

```
// This is a comment
// More comments can make your script easier to read

    x = x + 1    // Comments can end a line of script code

    /*
    This is a
    multiline comment
    */
```

8.3 - Actions

Actions are the basic execution unit of a script. A typical script will contain several action definitions. Events that occur within the GP16P will trigger an action.

Action names can be any unique non-reserved identifier. An identifier can be up to 32 characters long. The first character must be a letter; the following characters may be letters, numbers or the underscore character ("_").

8.4 - Global Variables

Scripts may have an unlimited number of global variables. Global variables have visibility throughout the script file. Every action and subroutine has visibility to a global variable. Global variables retain their values between execution of each action.

Variable names can be any unique non-reserved identifier. An identifier can be up to 32 characters long. The first character must be a letter; the following characters may be letters, numbers or the underscore character ("_").

All variables in the scripts are treated as character strings. You can define a variable (ie x), assign a text string to x, perform some string operations on x, then assign a number to x, and perform mathematical operations on x.

8.5 - Local & Static Local Variables

Script actions and subroutines may have an unlimited number of local variables. Local variables have visibility throughout the action or subroutine, but do not have visibility from within other actions or subroutines. Static local variables retain their values between execution of each action or subroutine.

8.6 - Constants

Scripts may have an unlimited number of constants. Constants have visibility throughout the script file. Constants have all the same properties as global variables, except that you can not assign a value to a constant at runtime.

Constant names can be any unique non-reserved identifier. An identifier can be up to 32 characters long. The first character must be a letter; the following characters may be letters, numbers or the underscore character ("_").

8.7 - Arrays

Scripts may have an unlimited number of global arrays. Global arrays have visibility throughout the script file. Each element of an array has all the same properties as global variables.

When an array is declared an array dimension is also declared. When indexing elements of an array, the first element has an index value of zero. This is the same as arrays in the C language.

Out of bounds write access to an array will be ignored. Out of bounds read access to an array will return an empty string.

Array names can be any unique non-reserved identifier. An identifier can be up to 32 characters long. The first character must be a letter; the following characters may be letters, numbers or the underscore character ("_").

9 GP16P Scripting Language Structure

9.1 - Script Structure

The structure of a script file is shown below. Global variable declarations must be done at the start of the file before any actions are defined. There can be any number of actions defined in the script file. Comments may appear at any point in the script file.

```
constant declarations
variable declarations
array declarations
action bodies
subroutine bodies
```

9.2 - Constant Declarations

A constant declaration begins with the keyword "constant:" followed by the constant name and a value assignment. The following example shows the structure of constant declarations.

```
constant: name = number
constant: name = "string"
```

The following example shows the declaration of two constants. The first global constant "c1" is initialized with the numeric value of 1000. The second constant "c2" is initialized with the string "Have a nice day.".

```
constant: c1 = 1000
constant: c2 = "Have a nice day."
```

9.3 - Global Variable Declarations

A global variable declaration begins with the keyword "variable:" and the variable name. After the variable name an optional assignment may be specified. The following example shows the structure of global variable declarations.

```
variable: name
variable: name = number
variable: name = "string"
```

The following example shows the declaration of three global variables. The first global variable "v1" is not initialized. The virtual machine will initialize this variable to an empty string. The second global variable "v2" is initialized with the numeric value of 10 . The third global variable "v3" is initialized with the string "Hello World".

```
variable: v1
variable: v2 = 10
variable: v3 = "Hello World"
```

9.4 - Global Array Declarations

A global array declaration begins with the keyword "array:" and the array name. After the array name an array dimension must be specified. Arrays may be one or two dimensional. The following example shows the structure of global array declarations.

```
array: name [size]
array: name [size][size]
```

The following example shows the declaration of two global arrays. The first global array "a1" has ten elements and the second global array "a2" has 100 elements.

```
array: a1[10]
array: a2[100]
```

The following example shows the declaration of a two dimensional global array.

```
array: a1[10][4]
```

Note: The virtual machine treats all arrays as one dimensional. The compiler will flatten all two dimensional array accesses into a single dimension linear array.

9.5 - Local & Static Local Variable Declarations

A local variable declaration begins with the keyword "variable:" and the variable name. After the variable name an optional assignment may be specified. The following example shows the structure of local variable declarations.

```
variable: name
variable: name = number
variable: name = "string"
```

The following example shows the structure of static local variable declarations.

```
static variable: name
static variable: name = number
static variable: name = "string"
```

The example in the Action Bodies section shows the use of a temporary and a static local variable.

9.6 - Action Bodies

An action declaration begins with the keyword "action:" followed by the action name, then an opening curly brace. Any number of statements may reside within the action body. The end of an action is indicated by a closing curly brace. The following example shows the structure of an action body.

```
action: name
{
    local variable declarations
    statements
}
```

The following example shows a typical action body. This action is named "BTN_1_PRESS". It has two local variables. The variable "count" is a static variable that will be incremented each time the action is executed. After the count is incremented a message string is built up with the count included and the message is printed to the console (a Telnet window).

```
// ------------------------------------------------------------
// This action will print the messages:
// SVM: This action has been executed 1 times.
// SVM: This action has been executed 2 times.
// SVM: This action has been executed 3 times.
// SVM: This action has been executed 4 times.
//    etc ...
// ------------------------------------------------------------
action: BTN_1_PRESS
{
    static variable: count = 0
    variable: message

    count = count + 1
    message = "This action has been executed " # count # " times."
    print (message)
}
```

9.7 - Action Parameters

When an action is executed a set of four parameters will be passed to the action. All four parameters are not always used. If a particular action type does not use all four parameters, the unused parameters will contain empty strings.
The meaning of the parameters is specified by the source of the action, see the section **action types**. Action parameters are accessed by the built-in variable names "$1", "$2", "$3" and "$4".

9.8 - Subroutine Bodies

A subroutine declaration begins with the keyword "subroutine:" followed by the subroutine name, then an open curly brace. Within the subroutine body are any number of statements. The end of a subroutine is indicated by a closing curly brace. The following example shows the structure of a subroutine body.

```
subroutine: name
{
    local variable declarations
    statements
    optional return
}
```

9.9 - Subroutine Parameters

When a subroutine is executed a set of four parameters will be passed to the subroutine. All four parameters are not always used. If a particular action type does not use all four parameters, the unused parameters will contain empty strings.
Subroutine input parameters are accessed by the built-in variable names "$1", "$2", "$3" and "$4". The following example shows the use of parameters within subroutines.

A Subroutine may return one parameter to the caller. The caller will access the returned parameter through the built-in variable name "$0". This parameter will remain valid until the next subroutine call is made.

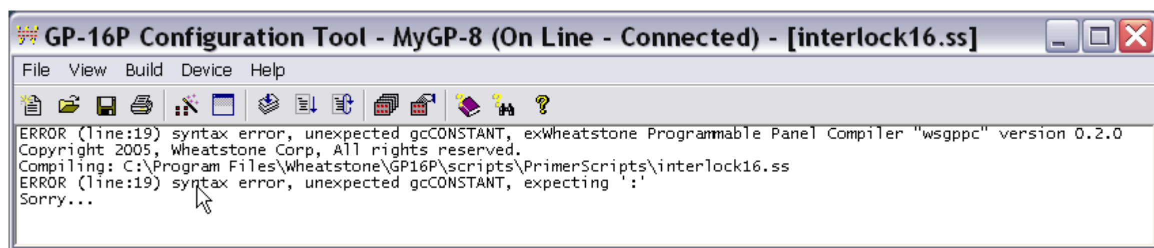
```
subroutine: sum_up_1
{
    var sum
    sum  = $1 # $2 # $3
    return sum
}

subroutine: sum_up_2
{
    return ($1 + $2 + $3 + $4)
}

subroutine: print_sum
{
    print_sum ("Sum = " # $1)
}

//--------------------------------------------------------------------
--
// This action will result in the followinfg message on the console:
// SVM: Hello World
// SVM: Sum = 100
//--------------------------------------------------------------------
--
action: test_action
{
    call sum_up_1 ("Hello", " ", "World")
    print ($0)
    call sum_up_2 (10, 20, 30, 40)
    call print_sum ($0)
}
```

10 Script Debugging

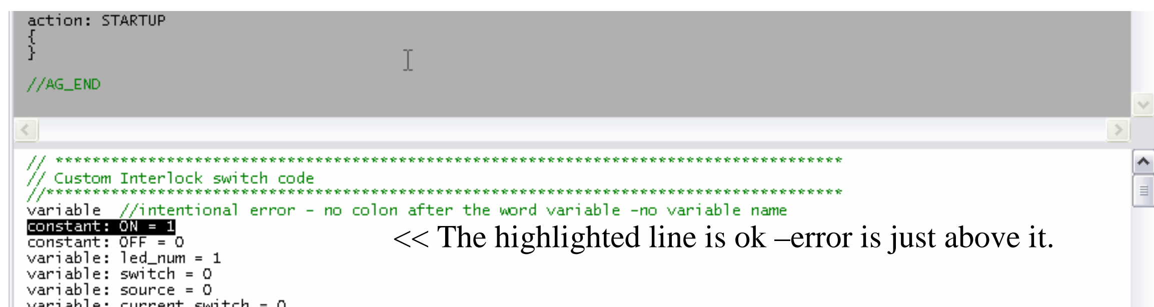
If you have delved into writing your own scripts you will inevitably have to debug them - if only to root out spelling or other minor syntax errors. Programming and debugging go hand in hand. Fortunately there are a couple of very useful tools to aid you in your time of need.

10.1 - Finding Compiler Errors

The “jump to error” feature in the Script Editor allows you to click on a reported compiler error in the Main GP16P window to jump to the line in the Script near or where the error is present. This feature is handy for tracking down bugs in scripts that will not compile. A word of caution, there are literally endless ways to write bad code, so this feature will usually get you close to the line with an error and not on the exact error. Also the Script Editor lacks a line number feature so it can be difficult to count lines out, especially in large scripts.



Clicking right on the compiler ERROR line shown above will cause the Script Editor to highlight the approximate error location – shown below.



10.2 - Third Party Editors

If you plan on doing a lot of scripting you might consider using a third party programming editor. Notepad++ is a nice freeware editor. When you open a GP script in Notepad++, you can choose a “Language” skin, like “Flash actionscript”, that will give you line numbers and a context sensitive text color scheme. You will still have to open the file in the GP16P tool before you compile – be sure to Save in the editor first.
Do an internet search for “Notepad++” to download this editor.

10.3 - Using “Print” and Telnet to Debug

The Print statement may be inserted anywhere into the script code to print messages, variable values, etc. to a Telnet window. This feature is extremely useful for tracking down bugs or displaying script behavior in compiled code running on the GP-xx panel.
Here’s how it works.

Add a Print statement anywhere in a subroutine or action. Add it to a button press action to print every time the button is pressed or released.
.
Example Print Statements:

Print (your_variable_name)
Print (“Put text in quotes”)
Print (“Put text in quotes and ” #variable# “ use the # sign to concatenate variables and text”)

To Telnet to the GP panel you need to know three things:

- IP address of the GP-xx panel
- User Name: knockknock
- Password: whosthere

Use any Telnet client or open a Command Prompt Window and type:
telnet 192.168.1.221 (or whatever the IP address of your GP-xx panel is).

Toggle the ECHO OFF and enter the username and password; you should see a screen similar to this one:

```
Telnet 192.168.8.221
Ctrl-D - Exit
Ctrl-E - Toggle Echo

Please log in...
Username: knockknock

Password: whosthere


Welcome knockknock


Software Version: 1.0.3 Built:Oct 18 2006 at 10:33:04
Panel Type: GP-16P
FPGA Version: 0A02


Available Commands:
---------- ---------- ---------- ---------- ---------- ----------
help       ver        width      read       write      uptime
errs       debug
mac        telnet     exit       nds
dbghdw     ip         props      sdbg       sstart     svar
srun       sshow      dstlist    srclist    slvlist    cdbg
---------- ---------- ---------- ---------- ---------- ----------
Type "help <command>" for help on a specific command.
Type "!!" to repeat the last command.


->
```

Once you are logged on you need to toggle Script Debugging ON.

To toggle Script Debug ON type:

sdbg 1 <Enter>

To turn it OFF type:

sdbg 0 <Enter>

Now when you press a button on the GP-xx panel running the Example interlock16.ss script, you will see the Print statements as they are executed.

Appendix A

A1 - Example Custom Script File – interlock16.ss

To open this file in the GP16PConfiguration Tool do the following:

1-Start the GP-16P Configuration Tool software
2- Click ***File-New***
3- Select interlock16 as the filename and click SAVE.
4- The Script Wizard opens automatically – click ***CANCEL*** to close it.
5- Select ***View >Script Editor***
6-Copy and paste everything between the //START HERE and //END HERE lines directly into the bottom window of the Script Editor.
7- Save as ***interlock16.ss***

```
//****START HERE*****************************************************
//************************************************************************************
// Custom Interlock switch code –file interlock16.ss – email paulpicard@wheatstone.com with any questions.
//************************************************************************************
constant: ON = 1
constant: OFF = 0
variable: led_num = 1
variable: switch = 0
variable: source = 0
variable: current_switch = 0
variable: last_led = 0

//************************************************************************************
// Map the destination you want to switch sources to here
//************************************************************************************
constant: dest_a = 1   // select destination id# in router for this 16x1 line selector

//************************************************************************************
//map source signal id's to buttons 1 through 16
//************************************************************************************
constant: source1 = 11 //change the 11 to another Source signal id# as required- repeat for the rest
constant: source2 = 12
constant: source3 = 13
constant: source4 = 14
constant: source5 = 15
constant: source6 = 16
constant: source7 = 17
constant: source8 = 18
constant: source9 = 109
constant: source10 = 110
constant: source11 = 111
constant: source12 = 112
constant: source13 = 113
constant: source14 = 114
constant: source15 = 115
constant: source16 = 116 //change the 116 to another source signal id# as required
```

```
//**********************
// Subroutines
//**********************

subroutine: handle_sw_press //This subroutine does most of the work.
                         //It receives switch# and source info from the button
                         //press actions.
{
  print ("Subroutine-handle_sw_press")

  switch = $1
  source = $2
  btn_led (last_led, OFF)
  call store_switch (switch)
  connect (dest_a, source)   //dest_a is a fixed destination defined above as a constant
  btn_led (switch, ON)

  print ("connecting Source ID " # source # " to  Dest " # dest_a # "." )
}


subroutine: store_switch
{
  current_switch = $1
  last_led = $1

}


//****************************
// Button press section
// ***************************

action: BTN_1_PRESS
{
switch = 1
source = source1
call handle_sw_press(switch, source)
}

action: BTN_2_PRESS
{
switch = 2
source = source2
call handle_sw_press(switch, source)
}

action: BTN_3_PRESS
{
switch = 3
source = source3
call handle_sw_press(switch, source)
}
```

```
action: BTN_4_PRESS
{
switch = 4
source = source4
call handle_sw_press(switch, source)
}

action: BTN_5_PRESS
{
switch = 5
source = source5
call handle_sw_press(switch, source)
}

action: BTN_6_PRESS
{
switch = 6
source = source6
call handle_sw_press(switch, source)
}

action: BTN_7_PRESS
{
switch = 7
source = source7
call handle_sw_press(switch, source)
}

action: BTN_8_PRESS
{
switch = 8
source = source8
call handle_sw_press(switch, source)
}

action: BTN_9_PRESS
{
switch = 9
source = source9
call handle_sw_press(switch, source)
}

action: BTN_10_PRESS
{
switch = 10
source = source10
call handle_sw_press(switch, source)
}

action: BTN_11_PRESS
{
switch = 11
source = source11
call handle_sw_press(switch, source)
}
```

```
action: BTN_12_PRESS
{
switch = 12
source = source12
call handle_sw_press(switch, source)
}

action: BTN_13_PRESS
{
switch = 13
source = source13
call handle_sw_press(switch, source)
}

action: BTN_14_PRESS
{
switch = 14
source = source14
call handle_sw_press(switch, source)
}

action: BTN_15_PRESS
{
switch = 15
source = source15
call handle_sw_press(switch, source)
}

action: BTN_16_PRESS
{
switch = 16
source = source16
call handle_sw_press(switch, source)
}


//****END HERE *******************************************************
```