

# GPC-IP SYSTEM

---

## STUDIO TURRET

### TECHNICAL MANUAL

Wheatstone Corporation  
September 2009



**GPC-IP Studio Turret Technical Manual**

©2009 Wheatstone Corporation

 *Wheatstone Corporation*

600 Industrial Drive  
New Bern, North Carolina 28562  
tel 252-638-7000 / fax 252-637-1285

# GPC-IP System

## Table of Contents

### Chapter 1 – GPC-IP Hardware

<b>General Information .....</b>	<b>1-2</b>
<b>GP-3 Headphone Panel .....</b>	<b>1-3</b>
Replacement Parts .....	1-3
GP-3 Pinouts .....	1-4
GP-3 Schematic .....	1-5
GP-3 Load Sheet .....	1-6
<b>GP-3-SK Headphone Panel .....</b>	<b>1-7a</b>
Replacement Parts .....	1-7a
GP-3-SK Pinouts .....	1-7b
<b>GP-4S 4 Switch Mic Control Panel .....</b>	<b>1-7</b>
Replacement Parts .....	1-7
GP-4S Pinouts .....	1-8
GP-4S Schematic .....	1-9
GP-4S Load Sheet .....	1-10
<b>GP-4W 4 Switch Control Panel .....</b>	<b>1-11</b>
Replacement Parts .....	1-11
GP-4W Pinouts .....	1-12
GP-4W Schematic .....	1-13
GP-4W Load Sheet .....	1-14
<b>GPIP-8 8 Switch Programmable Switch Panel .....</b>	<b>1-15</b>
Replacement Parts .....	1-15
GPIP-8 Pinouts .....	1-16
GPIP-8 Schematic .....	1-17
GPIP-8 Load Sheet .....	1-18
GPC-1 Schematic .....	1-19
GPC-1 Load Sheet .....	1-22
<b>GPIP-16 16 Switch Programmable Switch Panel .....</b>	<b>1-23</b>
Replacement Parts .....	1-23
GPIP-16 Pinouts .....	1-24
GPIP-16 Schematic .....	1-25
GPIP-16 Load Sheet .....	1-26
GPC-1 Schematic & Load Sheet see pages 1-19 - 1-22	
<b>GPC Chassis Full Size Template .....</b>	<b>1-27</b>
<b>GP-3 Headphone Panel Full Size Template .....</b>	<b>1-27a</b>
<b>GPC-IP System Parts List .....</b>	<b>1-28</b>
<b>GPC-IP Installation Kit Parts List .....</b>	<b>1-28</b>

## Appendix

### WheatNet-IP GPIP-16P Configuration Tool ..... Appendix-1

Title Page	
Table of Contents	i
continued	ii
1 Introduction	3
1.1 GPIP-xx Hardware Compatibility	3
1.2 Panel Types	3
1.3 Power Supply	3
1.4 LED's	3
2 What You Need to Get Started	4
2.1 WhwatNet IP GP-16P Configuration Tool Software	4
2.2 Physical Network Connection	4
2.3 IP Address Settings	4
2.3.1 Changing the GP Panel's IP Address	5
2.4 WheatNet-IP Navigator Software	5
2.5 WheatNet GP-16P Help File	6
3 Using GPIP-16P Configuration Tool Software	7
3.1 Programming Procedure Summary	7
3.2 Adding Devices	7
3.3 Selecting Devices	7
3.4 Create a New Script File	8
3.5 Script Wizard Button Functions	9
3.6 Script Wizard Output Functions	10
3.7 Script Wizard Custom Action Hook	10
3.8 Compile the Script	10
3.9 Starting the Script	11
3.10 Testing	11
3.11 Reviewing the Script Wizard Code	12
4 Configuring Device Properties	13
4.1 Device Properties Tab	13
4.2 Host BLADE Setting	13
4.3 Surface Configuration	13
4.4 Audio Processor	14
4.5 Soft LIO Configuration	14
5 LIO Example Using Soft LIO's	17
5.1 Configure the Source Signal in Navigator	17
5.2 Assign GPIP Soft LIO's	17
5.3 Create the Mic Control Script Using Script Wizard	18
5.4 Reviewing the Script Wizard Code	19
6 What is the Script Editor?	20
6.1 Script Editor Features	20
6.2 Third Party Editors	21
7 Creating Custom Scripts	22
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	23
7.6 Example Script - Variables and Constants	24
7.7 Example Script - Subroutines	25
7.8 Example Script - Actions	26



7.9 Custom Scripting Suggestions .....	28
7.10 Scripting Router Control .....	28
7.11 Scripting Surface Control .....	28
7.12 Basic Surface Functions .....	28
7.13 Advanced Surface Functions .....	29
7.14 Example surf_talk Commands .....	29
8 GPIP-16P Scripting Language Overview .....	30
8.1 Case Sensitivity .....	30
8.2 Comments .....	30
8.3 Actions .....	30
8.4 Global Variables .....	30
8.5 Local & Static Local Variables .....	31
8.6 Constants .....	31
8.7 Arrays .....	31
9 GPIP-16P Scripting Language Structure .....	32
9.1 Script Structure .....	32
9.2 Constant Declarations .....	32
9.3 Global Variable Declarations .....	32
9.4 Global Array Declarations .....	33
9.5 Local & Static Local Variable Declarations .....	33
9.6 Action Bodies .....	33
9.7 Action Parameters .....	34
9.8 Subroutine Bodies .....	34
9.9 Subroutine Parameters .....	34
10 Script Debugging .....	36
10.1 Finding Compiler Errors .....	36
10.2 Third Party Editors .....	37
10.3 Using "Print" and Telnet to Debug .....	37
Appendix A .....	39
Appendix A1 - Example Custom Script File .....	39

# GPC-IP Hardware

## Chapter Contents

<b>General Information .....</b>	<b>1-2</b>
<b>GP-3 Headphone Panel .....</b>	<b>1-3</b>
Replacement Parts .....	1-3
GP-3 Pinouts .....	1-4
GP-3 Schematic .....	1-5
GP-3 Load Sheet .....	1-6
<b>GP-3-SK Headphone Panel .....</b>	<b>1-7a</b>
Replacement Parts .....	1-7a
GP-3-SK Pinouts .....	1-7b
<b>GP-4S 4 Switch Mic Control Panel .....</b>	<b>1-7</b>
Replacement Parts .....	1-7
GP-4S Pinouts .....	1-8
GP-4S Schematic .....	1-9
GP-4S Load Sheet .....	1-10
<b>GP-4W 4 Switch Control Panel .....</b>	<b>1-11</b>
Replacement Parts .....	1-11
GP-4W Pinouts .....	1-12
GP-4W Schematic .....	1-13
GP-4W Load Sheet .....	1-14
<b>GPIP-8 8 Switch Programmable Switch Panel .....</b>	<b>1-15</b>
Replacement Parts .....	1-15
GPIP-8 Pinouts .....	1-16
GPIP-8 Schematic .....	1-17
GPIP-8 Load Sheet .....	1-18
GPC-1 Schematic .....	1-19
GPC-1 Load Sheet .....	1-22
<b>GPIP-16 16 Switch Programmable Switch Panel .....</b>	<b>1-23</b>
Replacement Parts .....	1-23
GPIP-16 Pinouts .....	1-24
GPIP-16 Schematic .....	1-25
GPIP-16 Load Sheet .....	1-26
GPC-1 Schematic & Load Sheet see pages 1-19 - 1-22	
<b>GPC Chassis Full Size Template .....</b>	<b>1-27</b>
<b>GP-3 Headphone Panel Full Size Template .....</b>	<b>1-27a</b>
<b>GPC-IP System Parts List .....</b>	<b>1-28</b>
<b>GPC-IP Installation Kit Parts List .....</b>	<b>1-28</b>

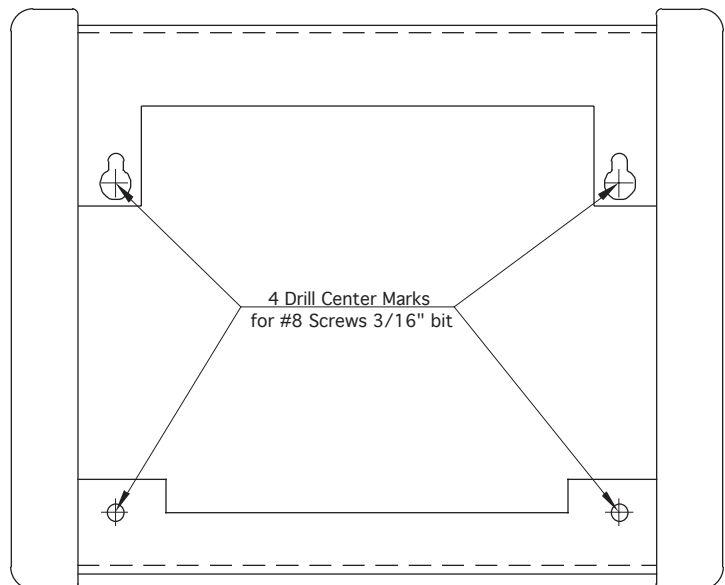
# GPC-IP Hardware



## General Information

The GPC system (W# 008710) is comprised of a desk turret (W# 008700) having some combination of the available panels installed. The turret can hold three single-wide panels, or one double-wide panel and one single-wide panel. Several single-wide panels are offered: the GP-3 (W#008705) headphone panel, the GP-4S (W#008706) 4 switch mic control panel, the GP-4W (W#008707) 4 switch control panel, the GPIIP-8 (W# 008703) 8 switch programmable switch panel, and the GP-BK (W# 008720) blank panel. The double-wide GPIIP-16 (W#008704) 16 switch programmable switch panel is also available. The panels are described in details on the following pages.

On the bottom part of the turret are four predrilled holes (3/16"D) that are used for mounting the turret to the countertop. Drill holes in the countertop by using the supplied full size turret template (W# 008712; see page 27). Then place the turret on the counter and secure it with the supplied #8 screws.



GPC-3 Chassis Template

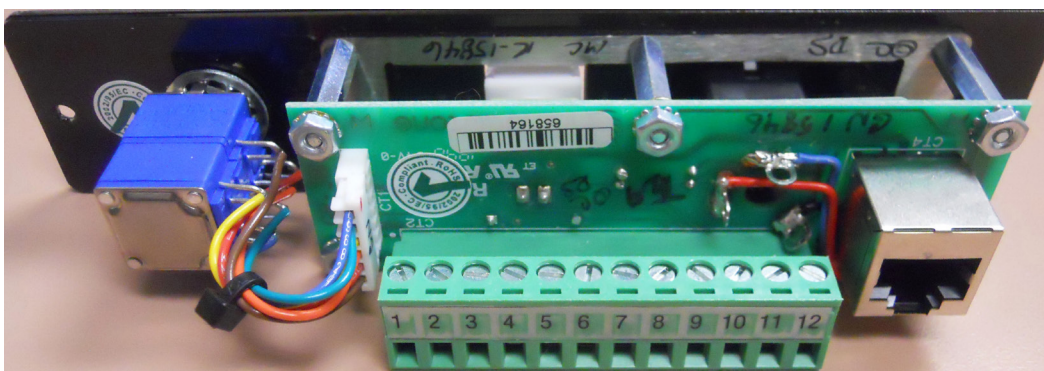
## GP-3 Headphone Panel (W# 008705)

The GP-3 panel is comprised of a switch, a Low Z level pot, and the 1/4" RTS and 3mm Stereo headphone jacks.

All user wiring to the GP-3 panel takes place at the 12-position plug terminal and the RJ-45 connector mounted on the GP-3PCB.



Front View



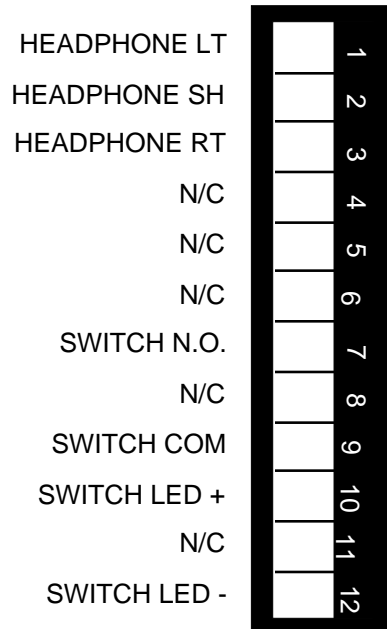
Rear View

### REPLACEMENT PARTS

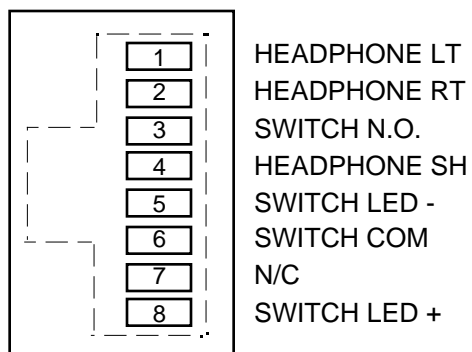
PART NAME	W#
FACEPLATE	008725
GP-3 SWITCH BARRIER LEFT	008714
GP-3 SWITCH BARRIER RIGHT	008719
SWITCH	510109
CLEAR FLAT TOP CAP WITH WHITE BASE AND WHITE INSERT	530109
POT DUAL LINEAR LOW Z	500121
21MM GRAY COLLET KNOB	520023
21MM BLACK CAP WITH WHITE LINE	530319
6 PIN PLUG	230031
6 PIN HEADER	250065
RTS JACK	260005
3.5MM STEREO JACK	260074
12-POSITION PLUG ON BARRIER STRIP	260045
12-POSITION BOXED HEADER	260046
RJ-45 CONNECTOR	260048
SWITCH LED RED	600027

## GP-3 Pinouts

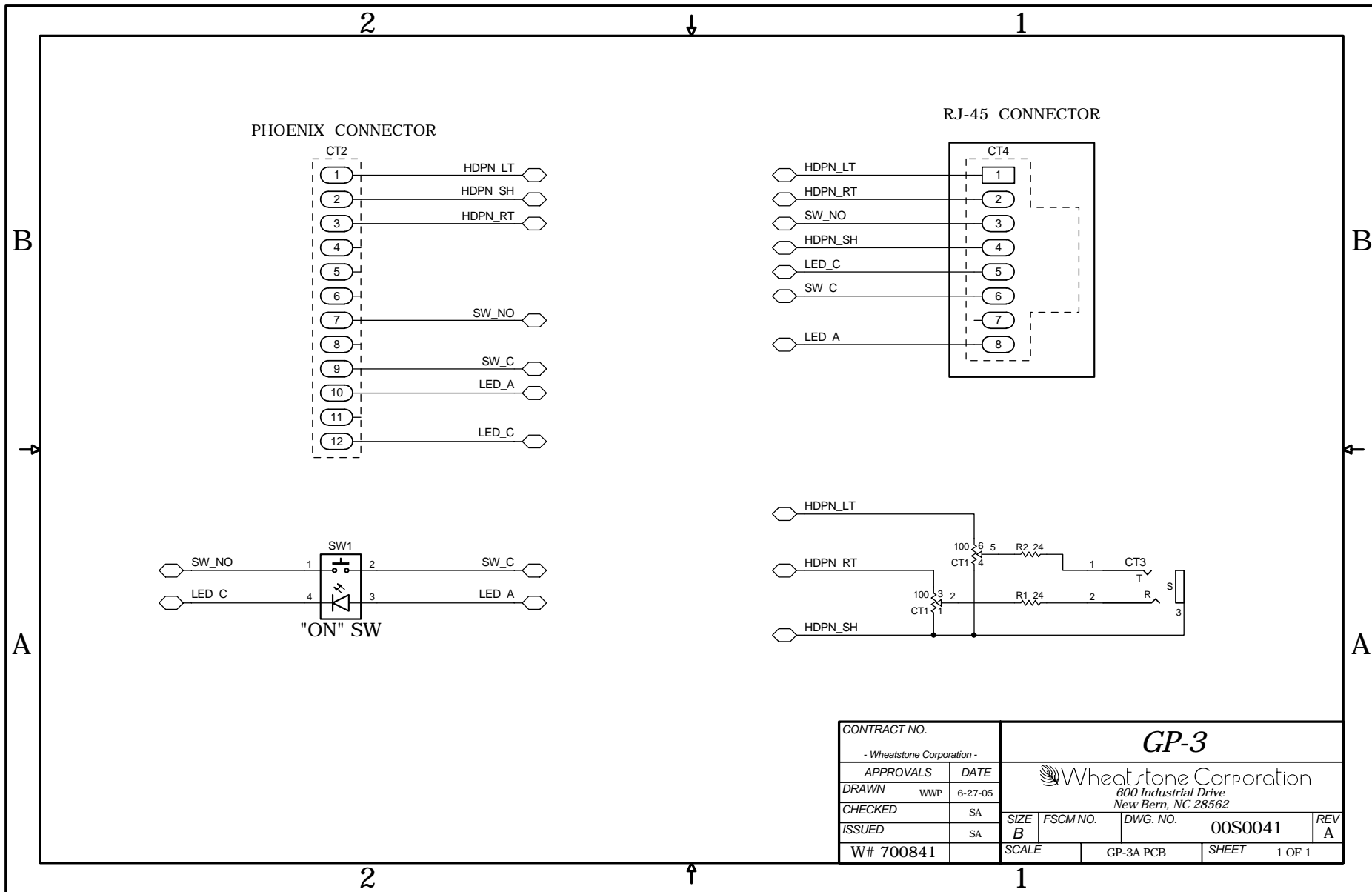
### *Plug Terminal*



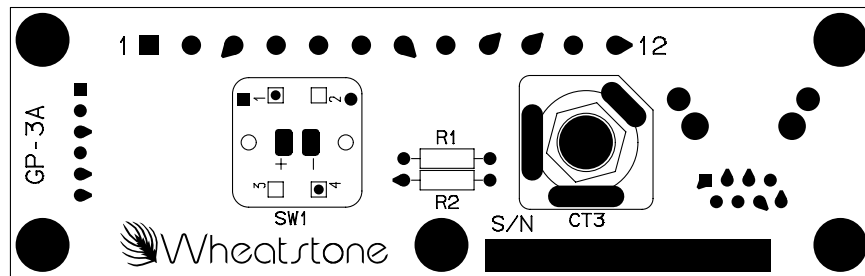
### *RJ-45 Connector*



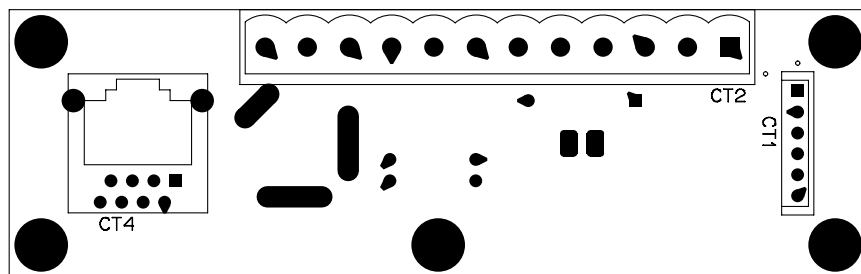
Note: Level pot is Low Z (100 ).



CONTRACT NO.		<div>GP-3</div>				
- Wheatstone Corporation -		<div>Wheatstone Corporation</div> <div>600 Industrial Drive</div> <div>New Bern, NC 28562</div>				
APPROVALS	DATE					
DRAWN WWP	6-27-05					
CHECKED	SA					
ISSUED	SA	SIZE B	FSCM NO.	DWG. NO.	00S0041	REV A
W# 700841		SCALE	GP-3A PCB		SHEET	1 OF 1



Top



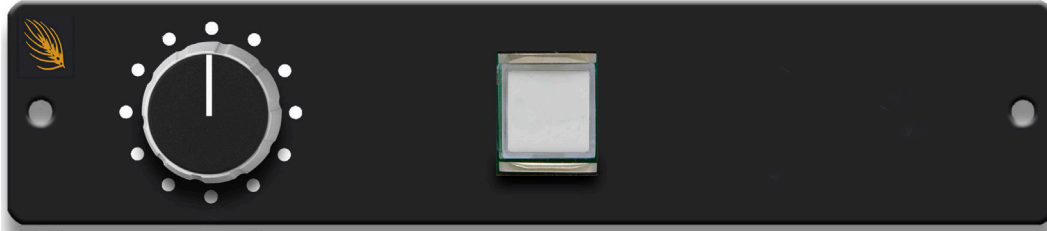
Bottom

## GP-3 Headphone Panel Load Sheet

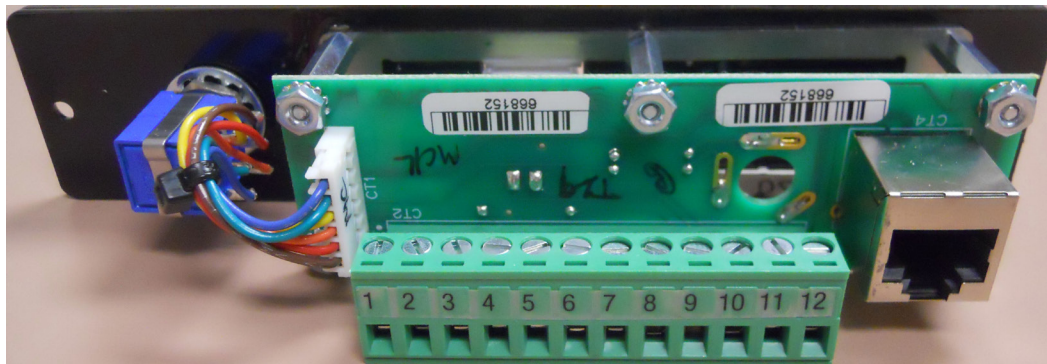
## GP-3-SK Headphone Panel (W# 008220)

The GP-3-SK panel is comprised of a switch and a level pot.

All user wiring to the GP-3-SK panel takes place at the 12-position plug terminal and the RJ-45 connector mounted on the GP-3PCB.



Front View



Rear View

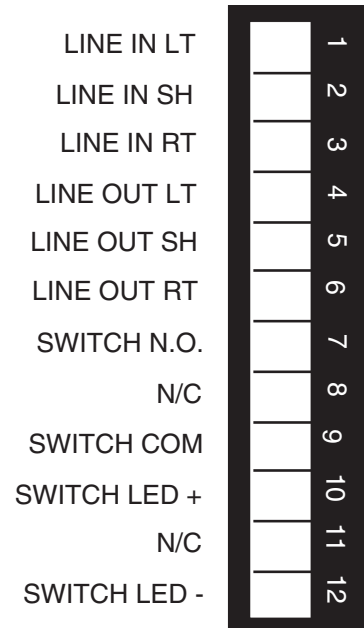
### REPLACEMENT PARTS

PART NAME	W#
FACEPLATE	008793
GP-3 SWITCH BARRIER LEFT	008714
GP-3 SWITCH BARRIER RIGHT	008719
SWITCH	510109
CLEAR FLAT TOP CAP WITH WHITE BASE AND WHITE INSERT	530109
10K DUAL AUDIO POT	500029
21MM GRAY COLLET KNOB	520023
21MM BLACK CAP WITH WHITE LINE	530319
6 PIN PLUG	230031
6 PIN HEADER	250065
12-POSITION PLUG ON BARRIER STRIP	260045
12-POSITION BOXED HEADER	260046
RJ-45 CONNECTOR	260048
SWITCH LED RED	600027

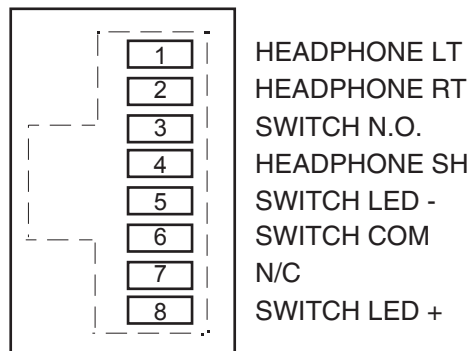


## GP-3-SK Pinouts

### *12-pin Plug Terminal*



### *RJ-45 Connector*



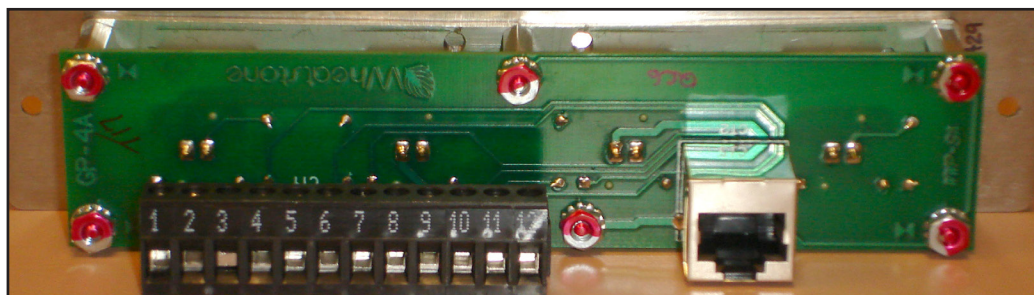
## GP-4S 4 Switch Mic Control Panel (W# 008706)

The GP-4S panel has “ON,” “OFF,” “COUGH,” and “TB” (talkback) switches for use as a microphone input remote control.

All user wiring to the GP-4S panel takes place at the 12-position plug terminal or the RJ-45 connector mounted on the GP-4PCB.



Front View



Rear View

### REPLACEMENT PARTS

PART NAME	W#
FACEPLATE	008726
GP-4 SWITCH BARRIER	008715
SWITCH	510109
RED TRANSP CAP FOR SWITCH	530097
ORANGE TRANSP CAP FOR SWITCH	530098
CLEAR FLAT TOP CAP WITH WHITE BASE & WHITE INSERT	530109
12-POSITION PLUG ON BARRIER STRIP	260045
12-POSITION BOXED HEADER	260046
RJ-45 CONNECTOR	260048
SWITCH LED RED	600027
SWITCH LED YELLOW	600031

## GP-4S Pinouts

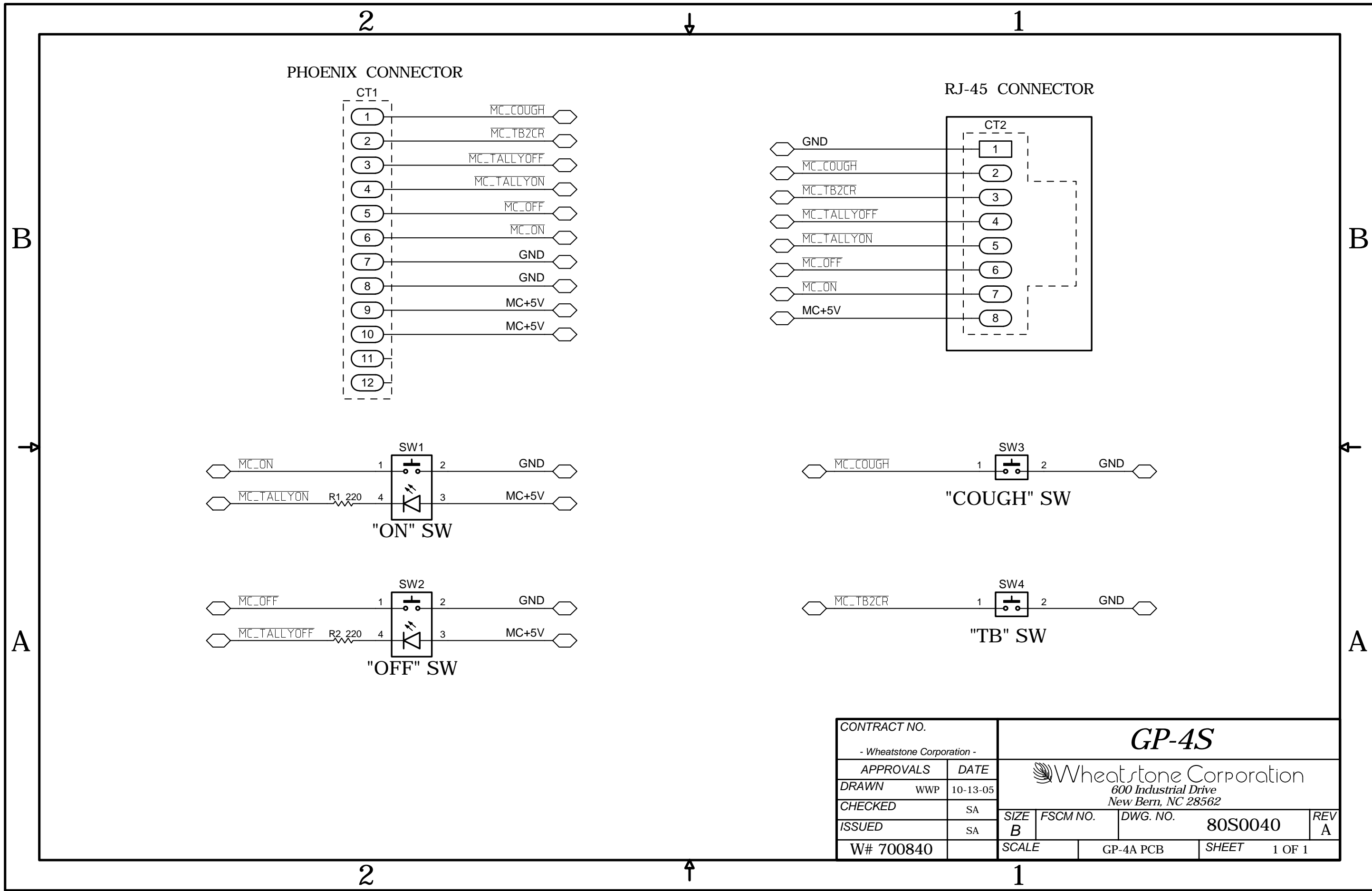
Wire these connections to the console mic input channel or WheatNet-IP logic port.


### *Plug Terminal*

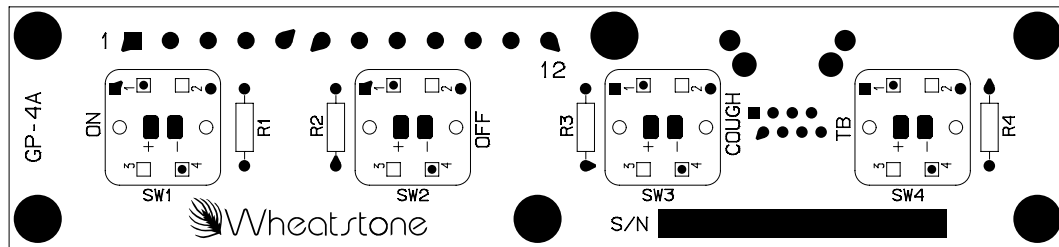
COUGH	1
TALK BACK	2
OFF TALLY	3
ON TALLY	4
REMOTE OFF	5
REMOTE ON	6
GROUND	7
GROUND	8
+5V DIGITAL	9
+5V DIGITAL	10
N/C	11
N/C	12

### *\*RJ-45 Connector*

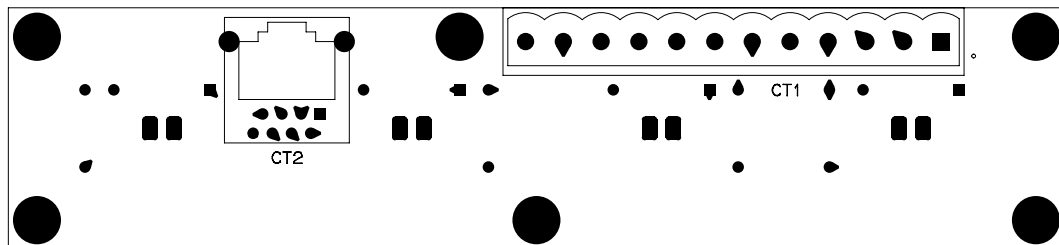
1	GROUND
2	COUGH
3	TALK BACK
4	OFF TALLY
5	ON TALLY
6	REMOTE OFF
7	REMOTE ON
8	+5V DIGITAL



CONTRACT NO.		GP-4S			
- Wheatstone Corporation -		 Wheatstone Corporation 600 Industrial Drive New Bern, NC 28562			
APPROVALS	DATE				
DRAWN WWP	10-13-05				
CHECKED	SA	SIZE	FSCM NO.	DWG. NO.	REV
ISSUED	SA	B		80S0040	A
W# 700840		SCALE	GP-4A PCB	SHEET	1 OF 1



Top



Bottom

## GP-4S 4 Switch Mic Control Panel Load Sheet

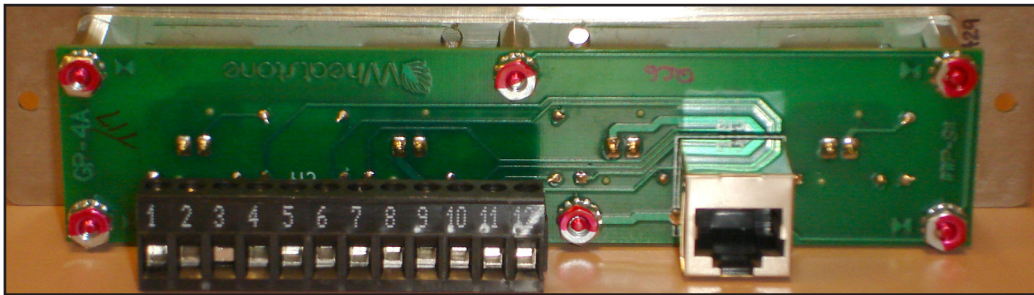
## GP-4W 4 Switch Control Panel (W# 008707)

The GP-4W panel has four general purpose illuminated switches.

All user wiring to the GP-4W panel takes place at the 12-position plug terminal or the RJ-45 connector mounted on the GP-4PCB.



Front View



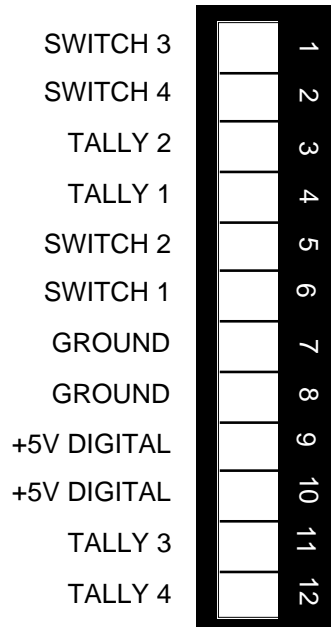
Rear View

### REPLACEMENT PARTS

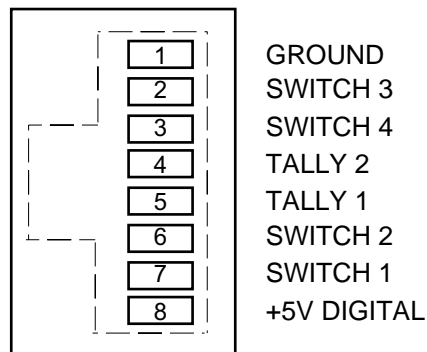
PART NAME	W#
FACEPLATE	008726
GP-4 SWITCH BARRIER	008715
SWITCH	510109
CLEAR FLAT TOP CAP WITH WHITE BASE & WHITE INSERT	530109
12-POSITION PLUG ON BARRIER STRIP	260045
12-POSITION BOXED HEADER	260046
RJ-45 CONNECTOR	260048
SWITCH LED RED	600027

## GP-4W Pinouts

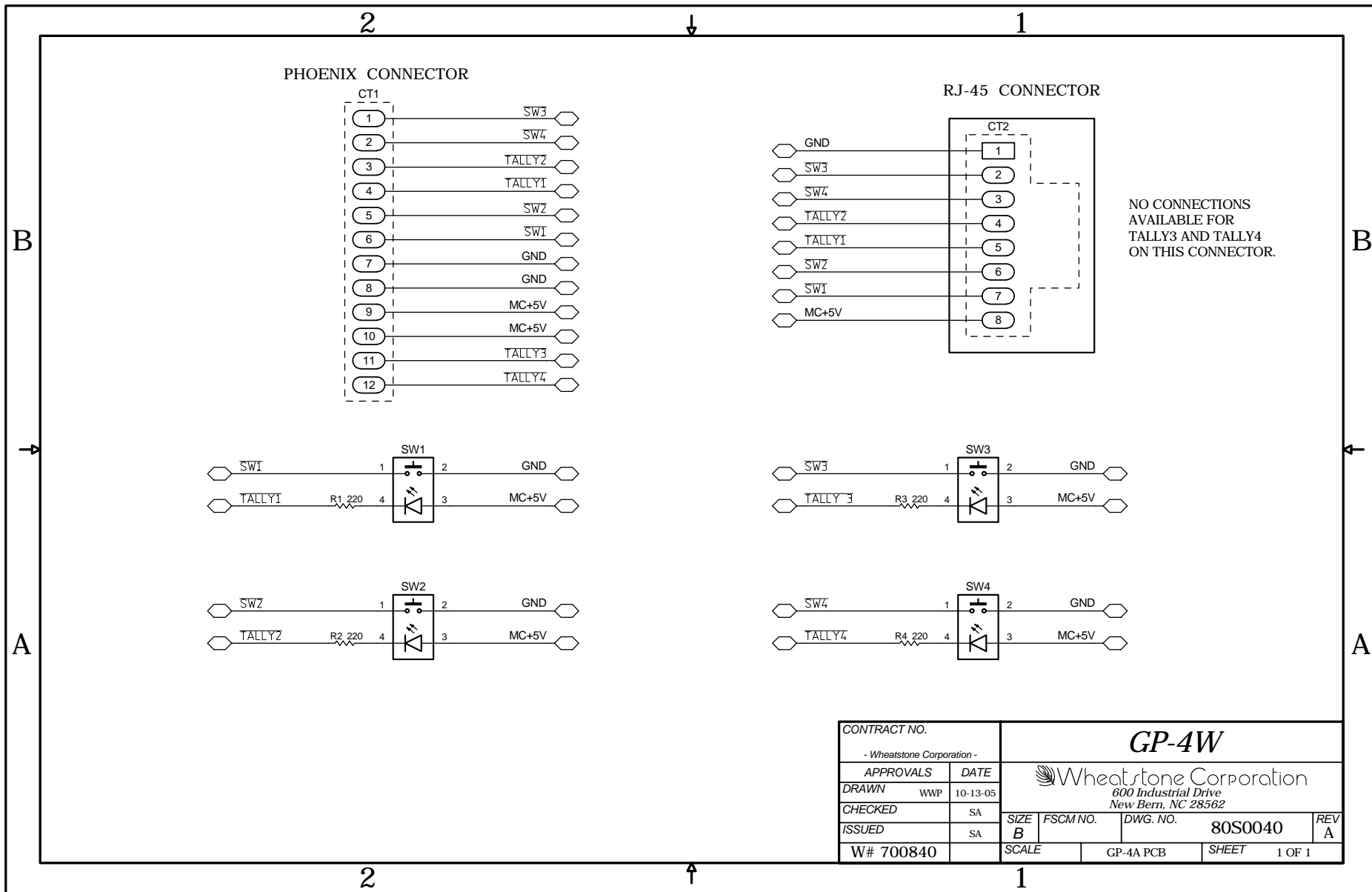
### *Plug Terminal*



### *RJ-45 Connector*

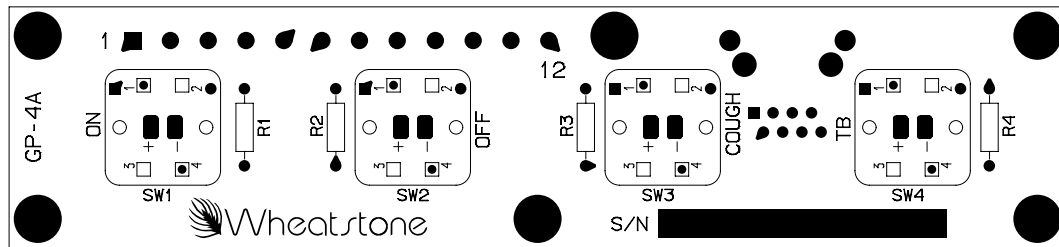


No connections available for Tally3 and Tally4 on this connector.

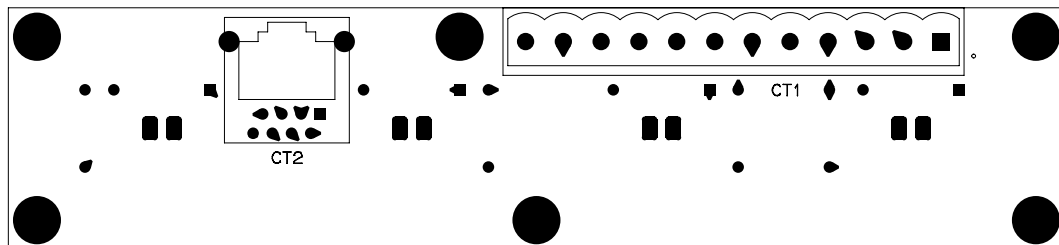


CONTRACT NO.			<b>GP-4W</b>			
- Wheatstone Corporation -			<b>Wheatstone Corporation</b> 600 Industrial Drive New Bern, NC 28562			
APPROVALS	WWP	DATE				
DRAWN	SA	10-13-05				
CHECKED	SA					
ISSUED	SA					
W# 700840			SIZE B	FSCM NO.	DWG. NO. 80S0040	REV A
			SCALE	GP-4A PCB	SHEET	1 OF 1





Top



Bottom

## GP-4W 4 Switch Control Panel Load Sheet

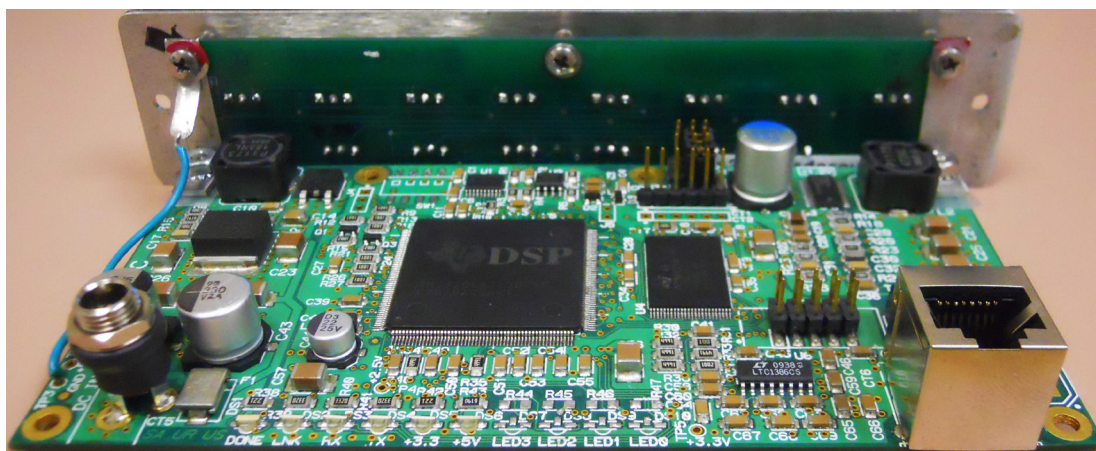
## GPIP-8 8 Switch Programmable Switch Panel (W# 008703)

The GPIP-8 panel has eight switches that can be programmed for a variety of functions by using the WheatNet IP GP-16P software (described in Appendix).

The unit has an RJ-45 connector for Ethernet connections and a DC power jack mounted on the GPC-1PCB.



Front View



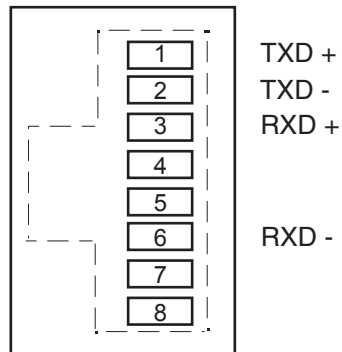
Rear View

### REPLACEMENT PARTS

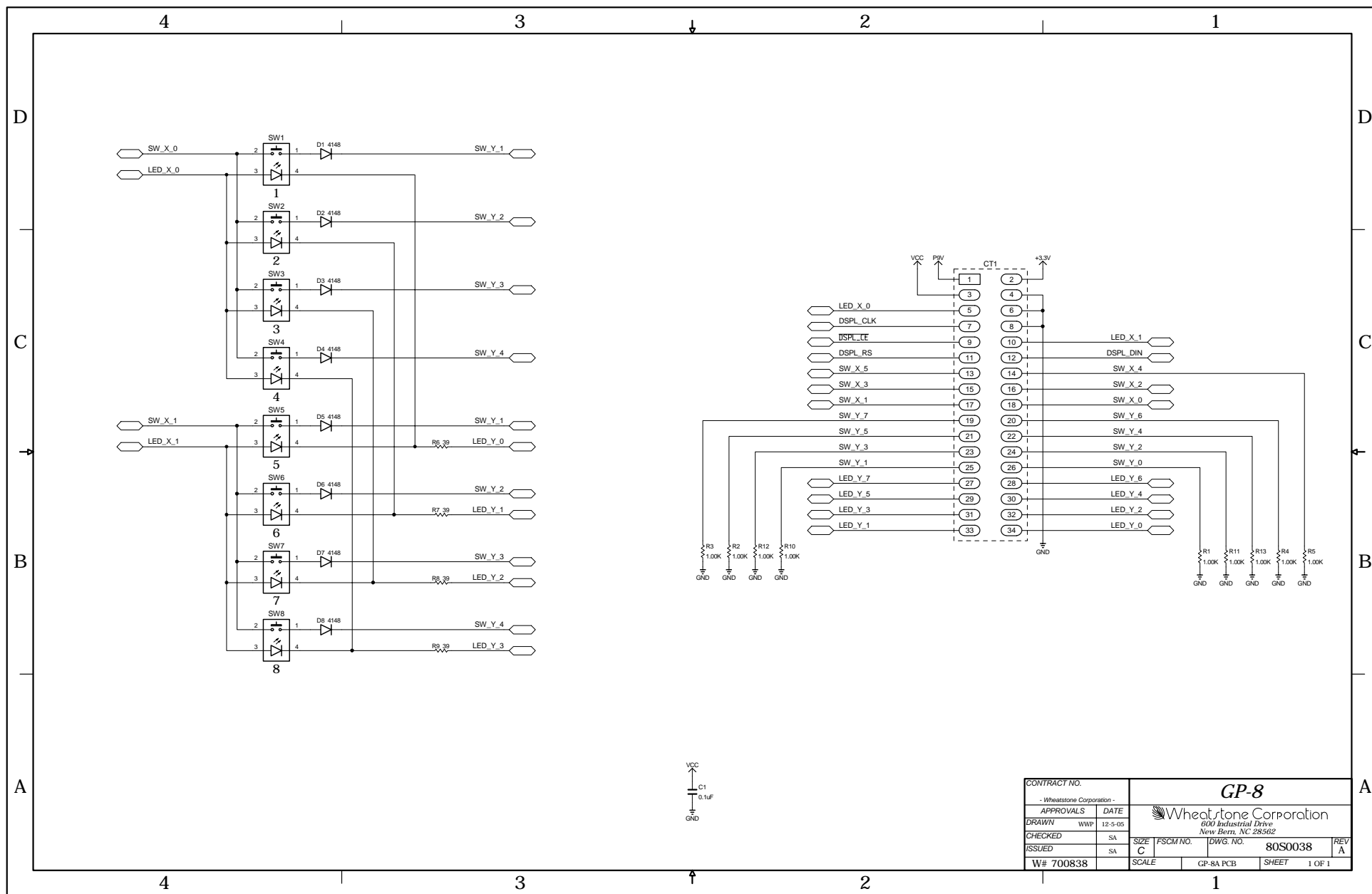
PART NAME	W#
FACEPLATE	008728
GP-8 PCB "L" BRACKET	008745
COAXIAL POWER JACK	260054
RJ-45 CONNECTOR UPRIGHT	260048
SWITCH NKK W/BRIGHTED RED LED	510290
WHITE CAP FOR SWITCH	530004
POWER WALL ADAPTER	980035
PLUG KIT FOR POWER ADAPTER	980037

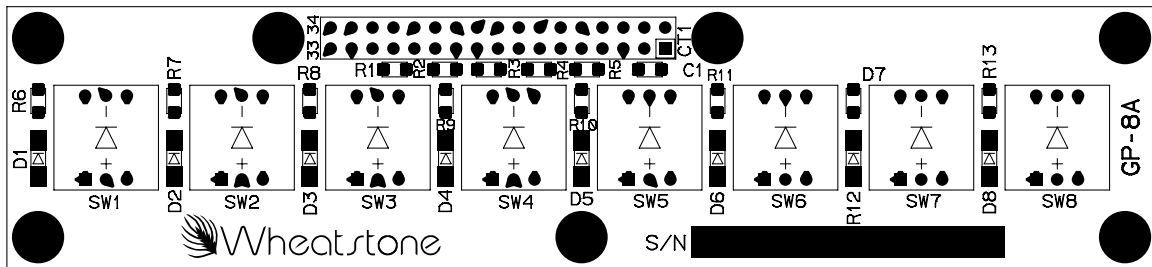
## GP-IP-8P Pinouts

### *RJ-45 Ethernet Connector*

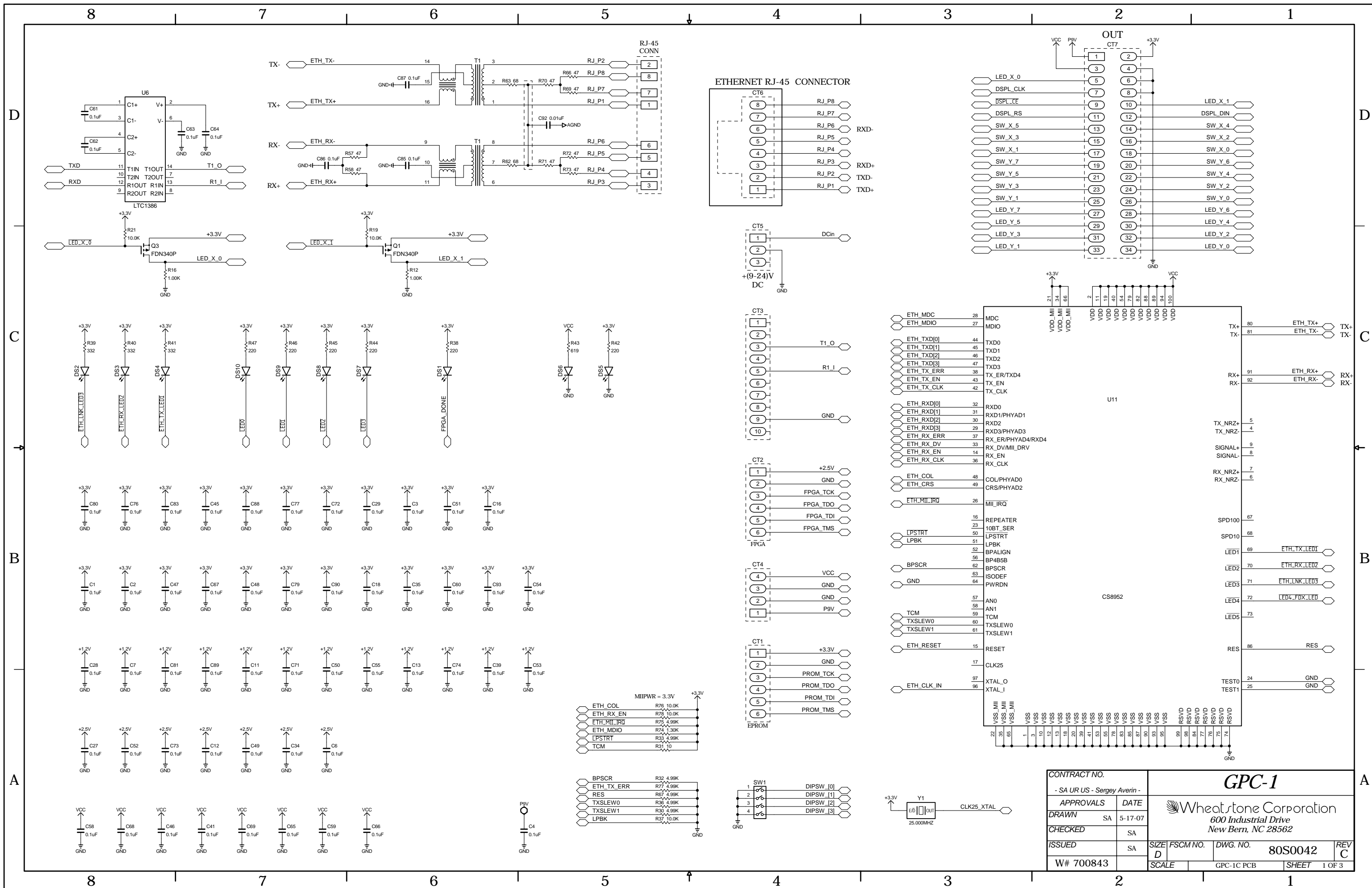


Plug the supplied AC adapter into the AC mains and into the DC IN power jack on the GPC-1PCB to power-up the panel.



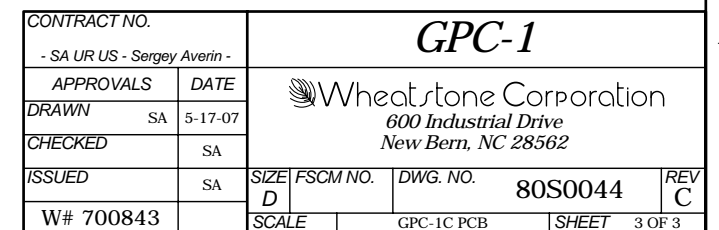
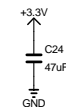
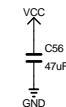


## GP-8 8 Programmable Switch Panel Load Sheet



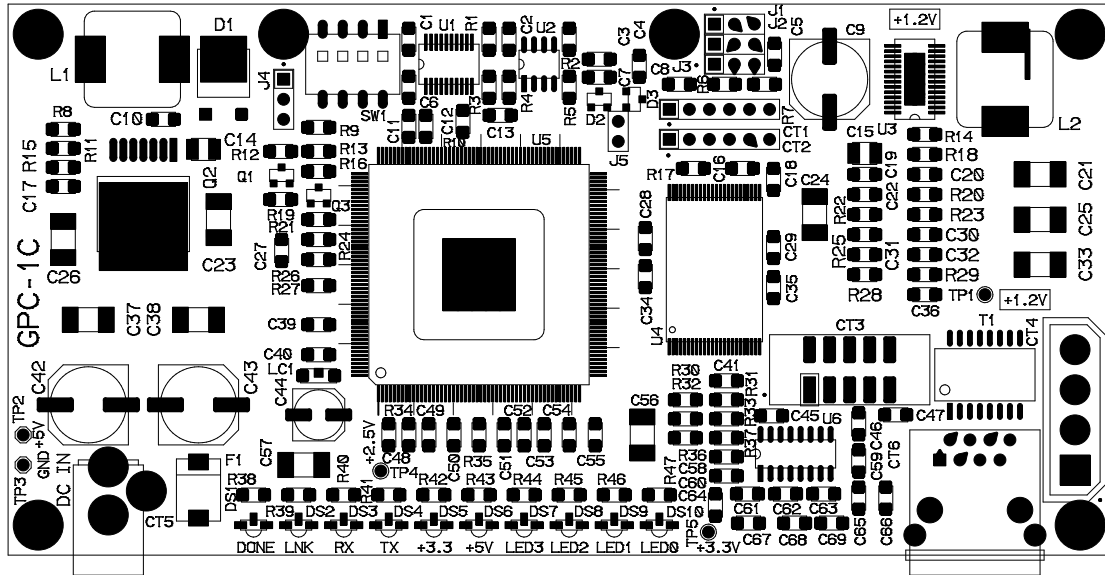
GPC-1 Controller Schematic - Sheet 1 of 3



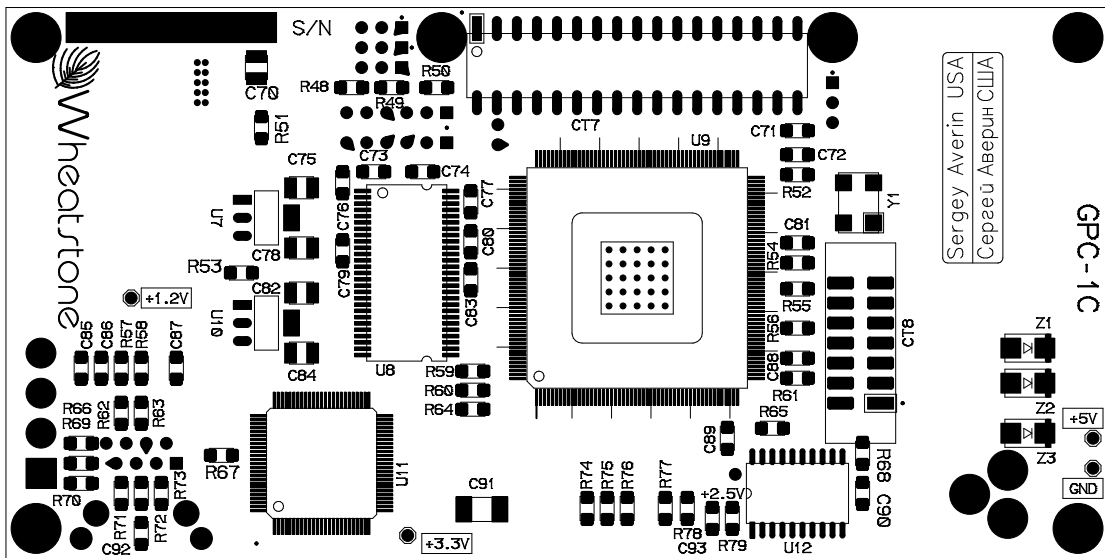


Page 1 - 21





Top



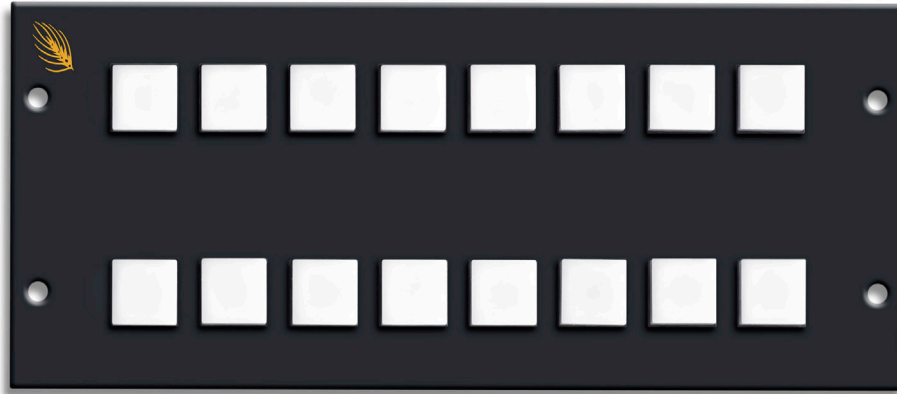
Bottom

## GPC-1 Controller Load Sheet

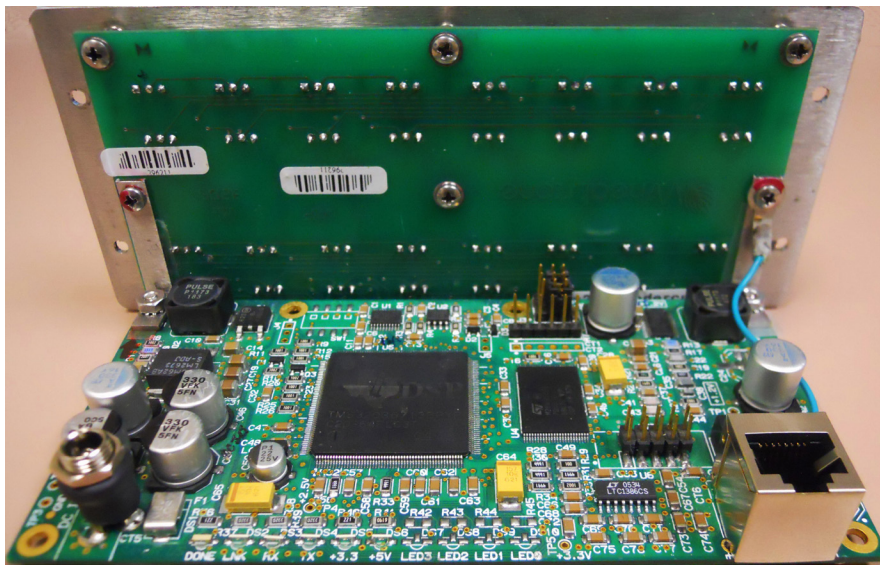
## GPIP-16 16 Switch Programmable Switch Panel (w# 008704)

The GPIP-16 panel has sixteen switches that can be programmed for a variety of functions by using the WheatNet IP GP-16P software (described in Appendix).

The unit has an RJ-45 connector for Ethernet connections and a DC power jack mounted on the GPC-1PCB.



Front View



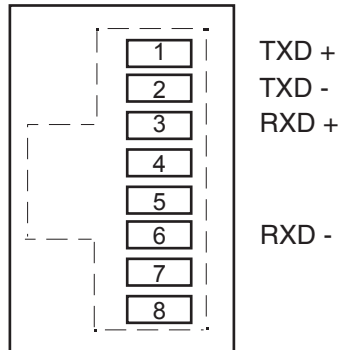
Rear View

### REPLACEMENT PARTS

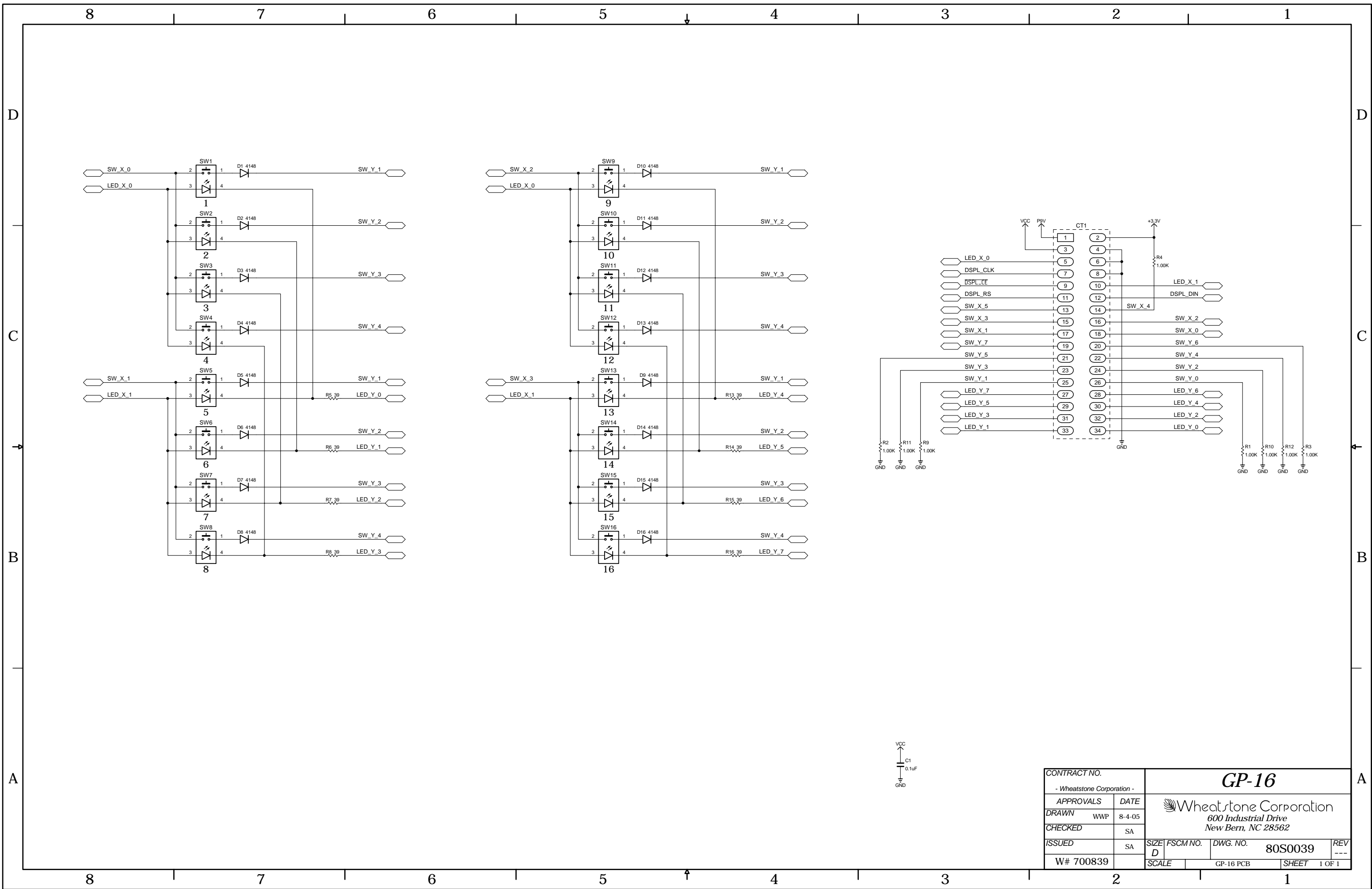
PART NAME	W#
FACEPLATE	008729
GP-16 PCB "L" BRACKET	008746
COAXIAL POWER JACK	260054
RJ-45 CONNECTOR UPRIGHT	260048
SWITCH NKK W/BRIGHTED RED LED	510290
WHITE CAP FOR SWITCH	530004
POWER WALL ADAPTER	980035
PLUG KIT FOR POWER ADAPTER	980037

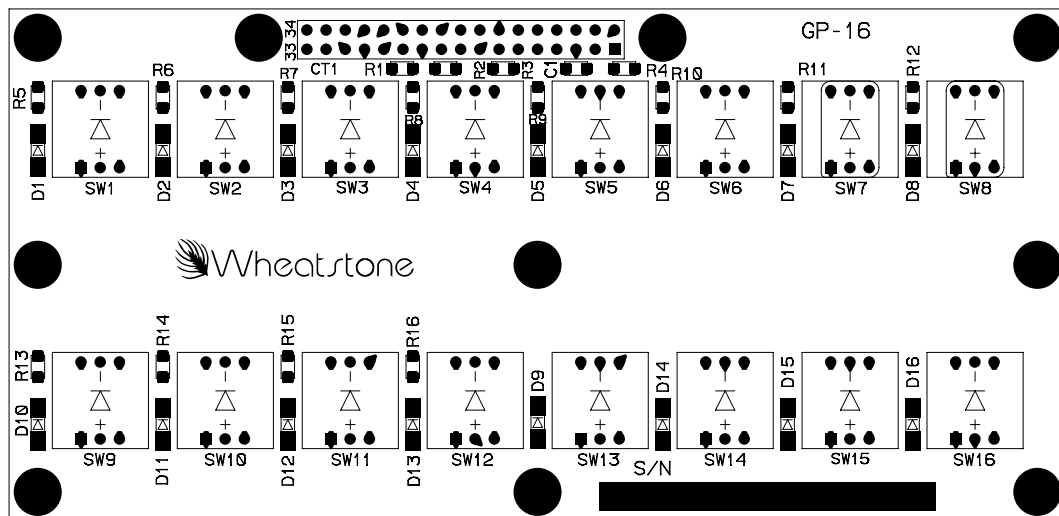
## GP-IP-16 Pinouts

### *RJ-45 Ethernet Connector*

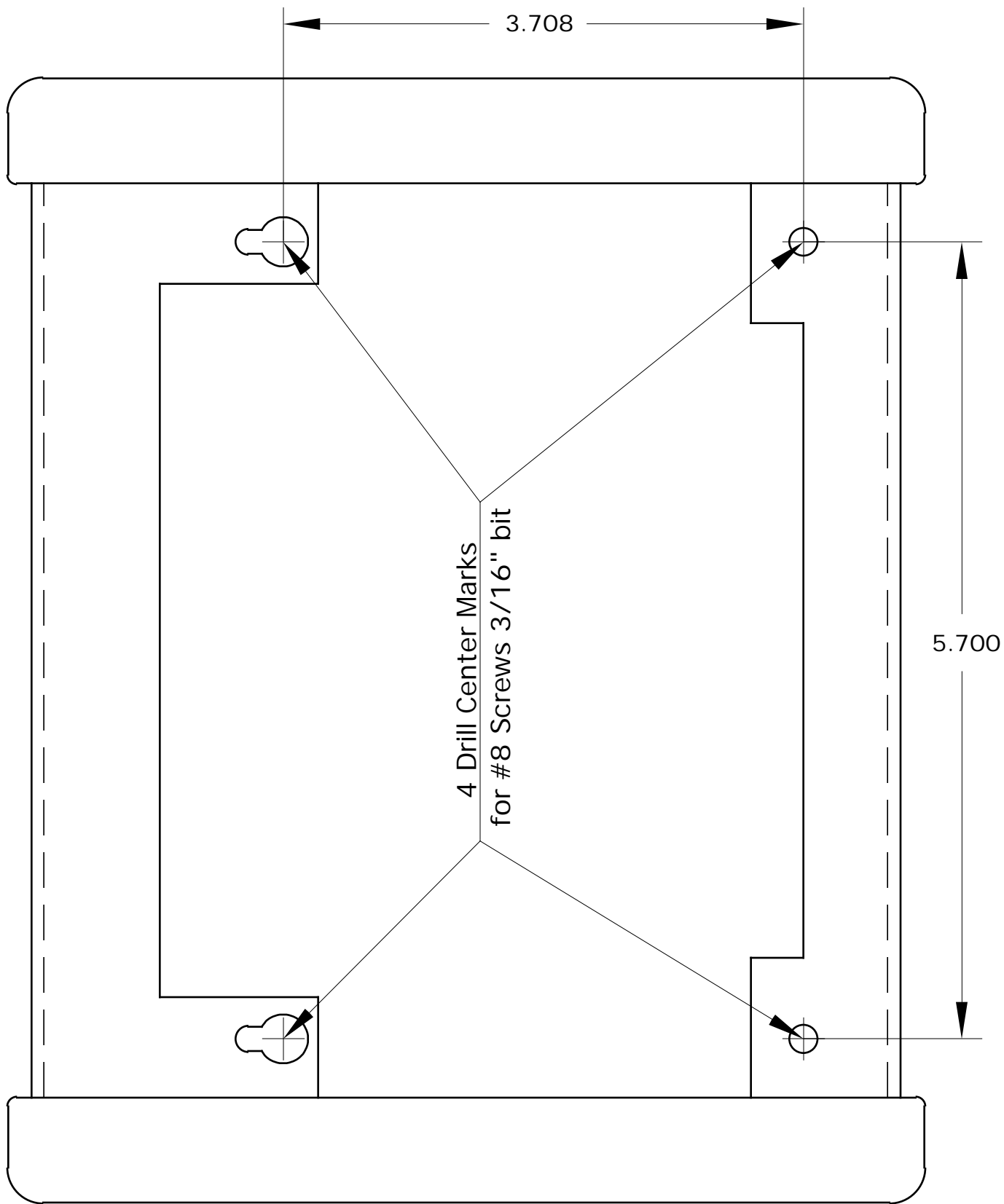


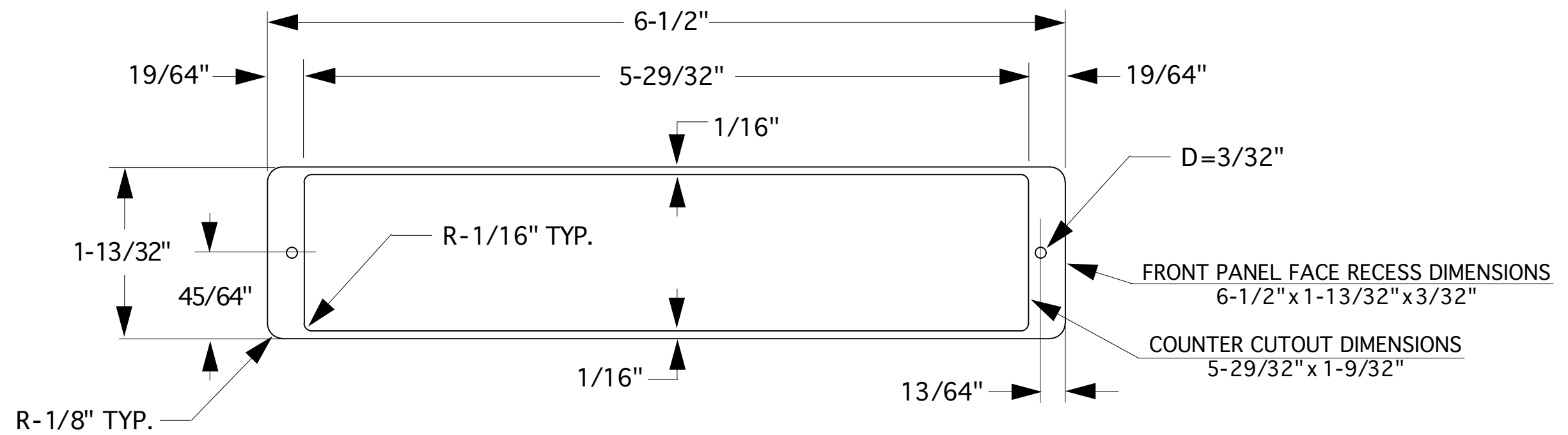
Plug the supplied AC adapter into the AC mains and into the DC IN power jack on the GPC-1PCB to power-up the panel.





## GP-16 16 Programmable Switches Load Sheet





GP-3 Headphone Panel Full Size Template

## GPC-IP SYSTEM PARTS LIST

PART NAME	W#
GPC DESK TURRET	008700
GP-U1 UNDER COUNTER MOUNT ASSY	008701
GP-3 HEADPHONE PANEL ASSY	008705
GP-4S SWITCH PANEL ASSY	008706
GP-4W SWITCH PANEL ASSY	008707
GPIP-8 SWITCH PANEL ASSY	008703
GPIP-16 SWITCH PANEL ASSY	008704
GP-BK BLANK PANEL	008720
GP DUAL RACK FACE	008744
GPC INSTALL KIT	008711
GP PANEL ROUTING TEMPLATE	008718

## GPC-IP INSTALLATION KIT PARTS LIST

PART NAME	W#
GPC TURRET MOUNTING TEMPLATE	008712
GPC-IP MANUAL	008727
440X3/16 PHILLIPS PANHEAD S/S SCREW	820019
832X5/8 PHILLIPS PANHEAD S/S SCREW	820127



# **Appendix**

## **WheatNet-IP GPIP-16P Configuration Tool**

### **Setup and Programming Guide**

# Wheatstone Corporation

## Technical Documentation

---

### WheatNet-IP

### GPIP-16P Configuration Tool

#### Setup and Programming Guide

- Configuring IP Addresses
- 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](http://www.wheatstone.com)

# Table of Contents

---

## 1 Introduction

1.1 GPIIP-xx Hardware Compatibility.....	3
1.2 Panel types.....	3
1.3 Power Supply .....	3
1.4 LED's .....	3

## 2 What You Need to Get Started

2.1 WheatNet-IP GP-16P Configuration Tool Software .....	4
2.2 Physical Network Connection .....	4
2.3 IP Address Settings .....	4
2.3.1 Changing the GP panel's IP Address .....	5
2.4 WheatNet-IP Navigator Software .....	5
2.5 WheatNet-IP GP-16P Help File.....	6

## 3 Using GPIIP-16P Configuration Tool Software

3.1 Programming Procedure Summary .....	7
3.2 Adding Devices .....	7
3.3 Selecting Devices .....	7
3.4 Create a New Script File .....	8
3.5 Script Wizard Button Functions.....	9
3.6 Script Wizard Output LIO Functions .....	10
3.7 Script Wizard Custom Action Hook .....	10
3.8 Compile the Script .....	10
3.9 Starting the Script .....	11
3.10 Testing .....	11
3.11 Reviewing the Script Wizard Code .....	12

## 4 Configuring Device Properties

4.1 Device Properties Tab .....	13
4.2 Host Blade Setting .....	13
4.3 Surface Configuration .....	13
4.4 Audio Processors .....	14
4.5 Soft LIO Configuration .....	14

## 5 LIO Example Using Soft LIO's

5.1 Configure the Source Signal in Navigator.....	17
5.2 Assign GPIIP Soft LIO's .....	17
5.3 Create the Mic Control Script Using Script Wizard .....	18
5.4 Reviewing the Script Wizard Code .....	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

---

## Table of Contents (continued)

7.5 Example Script Structure .....	23
7.6 Example Script – Variables and Constants .....	24
7.7 Example Script – Subroutines .....	25
7.8 Example Script – Actions .....	26
7.9 Custom Scripting Suggestions .....	28
7.10 Scripting Router Control .....	28
7.11 Scripting Surface Control .....	28
7.12 Basic Surface functions .....	28
7.13 Advanced Surface Functions .....	29
7.14 Example surf_talk Commands .....	29

## 8 GPIB-16 Scripting Language Overview

8.1 Case Sensitivity .....	30
8.2 Comments .....	30
8.3 Actions .....	30
8.4 Global Variables .....	30
8.5 Local & Static Local Variables .....	31
8.6 Constants .....	31
8.7 Arrays .....	31

## 9 GPIB-16 Scripting Language Structure

9.1 Script Structure .....	32
9.2 Constant Declarations .....	32
9.3 Global Variable Declarations .....	32
9.4 Global Array Declarations .....	33
9.5 Local & Static Local Variable Declarations .....	33
9.6 Action Bodies .....	33
9.7 Action Parameters .....	34
9.8 Subroutine Bodies .....	34
9.9 Subroutine Parameters .....	34

## 10 Script Debugging

10.1 Finding Compiler Errors .....	36
10.2 Third Party Editors .....	37
10.3 Using “Print” and Telnet to Debug .....	37

## Appendix A

A1 - Example Custom Script File – interlock16.ss .....	39
--	----

# 1 Introduction

---

This manual will guide you through the process of configuring and programming a GPIIP-8 or GPIIP-16 panel using the WheatNet-IP GP-16P Configuration Tool software. This primer is aimed at familiarizing you with the software's fundamentals and quickly getting your GPIIP-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.

Extensive custom scripting tools are provided to accommodate user applications that go beyond basic audio signal and logic control. The custom script writer will want to make use of the comprehensive Help File.

Certain sections of this document use material located in the software's extensive Help file.

## 1.1 - GPIIP-xx Hardware Compatibility

The GPIIP-xx hardware is loaded with firmware and software designed to be used with a WheatNet-IP based system exclusively. These models are physically similar to the devices that connect to a legacy TDM WheatNet or Bridge router system; however the software is not compatible. Legacy GP panels may be updated to work with a WheatNet-IP system.

## 1.2 - Panel Types

The GPIIP-8 and GPIIP-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.

## 1.3 - Power Supply

GPIIP-xx panels use a wall wart style power supply rated at 9V DC and 1000 mA. The DC output connector is type - "2.5mm x 5.5mm," female, with center positive wiring.

Onboard regulators provide the +5 and +3.3 VDC voltages required by the GPC-1 circuit board.

## 1.4 – LED's

The following diagnostic LED's are mounted to the main GPC-1 circuit card.

- **Done** - momentarily lights during the boot sequence, then is normally OFF.  
Note - if DONE Stays lit the firmware failed to load. Contact Wheatstone for repair.
- **Link** - lights when Ethernet connectivity is established.
- **Rx/Tx** - signals the reception and transmission of Ethernet traffic.
- **+3.3 & 5V** - indicate the presence of these power supply voltages.
- **LED 0 - 3** - factory use only.

# 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 - WheatNet IP GP-16P Configuration Tool Software

Make sure you have installed the WheatNet IP 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 1.0.0 but the general process will apply to all versions.

## 2.2 - Physical Network Connection

The setup and editing of GPIIP-xx devices requires an Ethernet network connection. While the overall WheatNet-IP BLADE system requires a 1000BASE-T Gigabit network, GPIIP panels may be programmed and operated using a 10/100BASE-TX Ethernet connection. There are two ways to connect:

*1000BASE-T or 100BASE-TX LAN* - the GPIIP-xx device and PC are connected to a common 10/100 or 10/100/1000 Ethernet switch 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. A drawback to this type of connection is that when the GPIIP-xx device is re-booted, the PC momentarily loses the network connection and takes a moment to recover. This loss of connectivity may be a source of trouble when configuring IP addresses.

## 2.3 - IP Address Settings

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

- All GPIIP devices in the system are assigned a unique static IP address, no DHCP.
- The device's factory supplied IP address is printed on a label affixed to the front panel.
- The default factory IP address for GP devices starts at 192.168.87.221.
- ***GPIIP-16P Programming Tool software*** is used to change the unique static IP addresses of GPIIP-xx panels. You will need the GPIIP panel's MAC address. It is printed on a label and affixed to a chip on the GPIIP panel's circuit board.
- The PC can use a standard 10/100 (Fast Ethernet) or 10/100/1000 (Gigabit) adapter.
- The PC running the GPIIP-16 Configuration Tool **must be on the SAME subnet** as the GPIIP-xx device. For example if your GP-xx IP address is 192.168.87.221 then the PC's Network Interface Card must be given a unique IP address on the 192.168.87.xxx subnet.

**Important:**  
**GPIIP-xx panel addresses are easily changed using the**  
***WheatNet IP GP-16P Programming Tool* software.**  
**Please refer to the next section for details on *changing* a GP panel's IP address.**

### 2.3.1 - Changing the GP panel's IP Address

If you need to change the IP address of a GPIIP panel, start up the WheatNet-IP GP-16P Programming Tool Software. Choose the menu item "*Hardware>Assign IP Address.*" The following dialog box will appear:

The image shows a software dialog box titled "IP Assignment" with a green header bar and a close button (X) in the top right corner. Inside the dialog, there are several input fields. Under "Device Filter:", the "MAC Address:" field contains "00:50:C2:23:C2:74". Under "New IP Settings:", the "Name:" field contains "Host-GP", the "IP Address:" field contains "192.168.87.222", the "Subnet:" field contains "255.255.255.0", and the "Gateway:" field contains "192.168.87.1". At the bottom, there is a "Requests:" label next to a counter box showing the number "1". Below the counter is a large "START" button. A mouse cursor is pointing at the "START" button.

Fill in the required information as follows:

*MAC Address:* get this twelve character hexadecimal number from the label affixed to a chip on the GPIIP panel's circuit card. Wheatstone MAC addresses begin with the sequence 0050C223xxxx.

*Name:* any 8 character name for this device.

*IP Address:* any *unique* static IP address on the WheatNet-IP system's subnet. (i.e. 192.168.87.221)

*Subnet:* Standard Subnet Mask for your WheatNet-IP subnet. (i.e. 255.255.255.0)

*Gateway:* enter the WheatNet-IP system's Gateway or 255.255.255.255 if no gateway is used.

Once you have filled in the *IP Assignment* form:

- Press the START button located at the bottom of the form.
- Cycle the power on the GPIIP panel - while re-booting, the GPIIP will request its network settings.
- The "*Requests*" counter box should increment indicating that the new network settings were requested by the GPIIP panel.

To test the new network settings:

Open a command prompt window and use the "ping xxx.xxx.xxx.xxx" command to see if the GPIIP panel responds to the new IP address setting.

To open a command prompt window:

From the Windows XP Start Menu select Run and type "cmd" without quotes then press ENTER.

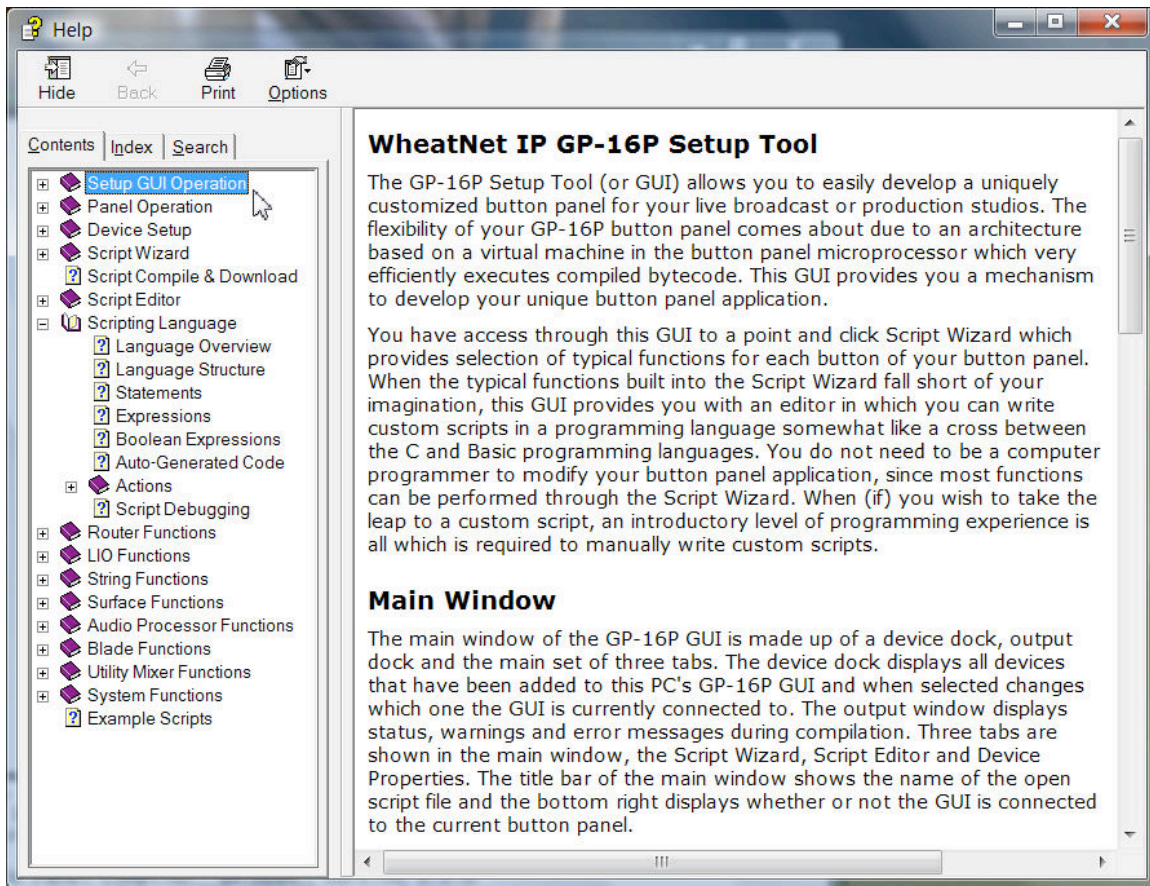
From Windows Vista Start Menu type "cmd" or "prompt" without quotes in the *Start Search* box.

## 2.4 - WheatNet-IP Navigator Software

The GPIIP-xx panels may be programmed to control audio and logic signal cross-points, fire Salvos, activate surface presets, and many other functions. Depending on your application, the *GPIIP-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 the WheatNet-IP system. You will need access to the *WheatNet-IP Navigator* software to access the required information. Navigator is also used to create optional "Soft LIO" signals for certain applications.

## 2.5 - WheatNet-IP GP-16P Help File

The *WheatNet IP 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 GPIP-16P Configuration Tool Software

OK, now that we have the network connection issues taken care of we can start the GPIP-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 GPIP-xx device are listed below - let's review them and then perform each in turn.

- Add the Device info to the GPIP-16P Software Tool
- Connect to the Device in Online Mode
- Open 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 GPIP-16P Configuration Tool Software.

## 3.2 - Adding Devices

Before you can program a GPIP panel you need to define each panel to be a "device" in your system.

Use the Menu item **Hardware->Add New Device...**

Enter an 8 character Name for the panel and its IP address. The IP address tells the software which device to talk to when you choose a device name, it doesn't change the IP address of the panel. To change IP addresses please see Chapter 2.



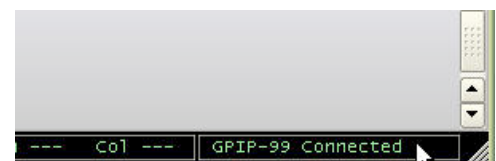
## 3.3 - Selecting Devices

As you add each new GPIP panel using the *Add Device* form, they appear in alphabetical order in the Devices list located on the left side of the main screen.



- You can mouse over the Device name in the list to see its IP address.
- When you wish to program a GPIP panel, you simply select it from the devices list.

Check that you are connected in the *Status Bar* at the bottom.



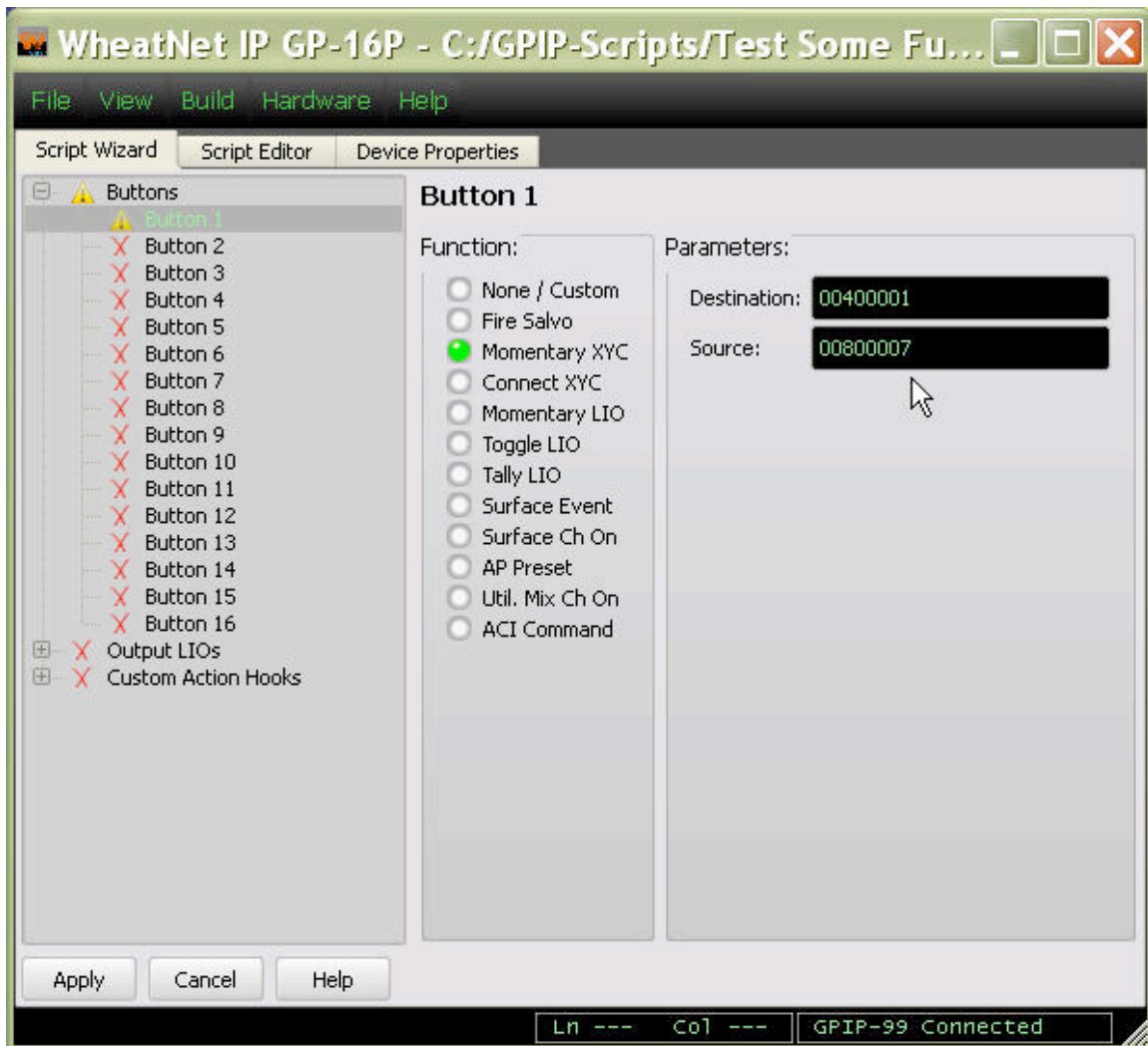
If this field is stuck on "Connecting..." then you have an Ethernet issue, perhaps related to a wrong IP address.

### 3.4 - Create a New Script File

Select the Menu item **File->New** and the *Script Wizard* opens automatically.

The *Buttons* list in the scroll pane on left side of the Wizard is where you select which GPIIP panel button you would like to program. Simply click on the button name 1-16 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 BLADE signal ID numbers or Salvos as configured in the WheatNet-IP Navigator software.



### 3.5 - Script Wizard Button Functions

The following functions may be mapped in any combination to the GPIIP-xx buttons. Note that in some cases a button may perform actions on the press, release, and over-press of the switch. The software's Help file includes all the pertinent 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 Navigator. Enter the Salvo's Index number in the Press and Release Parameters fields. Salvos are created in Navigator and are simply a stored set of one or more routes and/or disconnects. The first Salvo in the Navigator 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 XYZ** – XYZ 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 the previous Source when the button is released. Enter the Destination and Source signal ID numbers from your Navigator configuration's Detail window. You can display Signal ID numbers in the Detail Dock at any time by left clicking on the desired signal in the System Crosspoint grid. Make sure the Detail Dock is visible on the lower left side of Navigator by clicking on the Detail Dock button at the top of Navigator.

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

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

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

**Tally LIO** – use this to turn the button into an indicator lamp. The LED in the button will light when the logic condition is met. Button presses are ignored. This function requires mapping of Soft LIO's in *Device Properties* – see section 4 or Help for specific details.

**Surface Event** – 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 *Device Properties* tab. 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 the IP address associated with **Surface 1:** in the list.

*Event:* - 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.

**Surface Ch On** – Turn a surface's input fader channel On and Off.

*Surf ID:* Selects a surface, 1 through 8, as defined in the *Device Properties* tab.

*Input Ch:* Select the input fader channel, 1- 48.

**AP Preset** – Use this function to change Presets on a Vorsis Audio Processor.

*Audio Proc:* Selects the processor, 1-4, as defined in the *Device Properties* tab.

*Preset:* Select the Preset number to take on the selected processor.

**Util. Mix Ch On** – Use this function to turn a fader or main mix *ON* in a BLADE's Utility Mixer.

*Mixer ID:* Select Utility Mixer 1 or 2 in the host blade defined in *Device Properties* tab.

*Channel:* Select input fader 1-8 or output A or B.

**ACI Command** – ACI Device: Host BLADE or any surf 1-8 from dev props

Press: ACI string sent on button press Release: ACI string sent on button release.

### 3.6 - Script Wizard Output LIO Functions

The GPIIP panel provides direct access to the logic states of the 64 Soft LIO's that reside inside the Host BLADE specified on the Device Properties tab. The GPIIP panel "subscribes" to the Host BLADE and is automatically updated by the Host BLADE whenever the state of the 32 input or 32 output Soft LIO's changes. GPIIP scripts may read and change the state of the Soft LIO's in the Host BLADE.

By using the functions in this section you can trigger specific actions based on the high or low state of any given Soft LIO. Please see the Help File for Details.

### 3.7 - Script Wizard Custom Action Hook

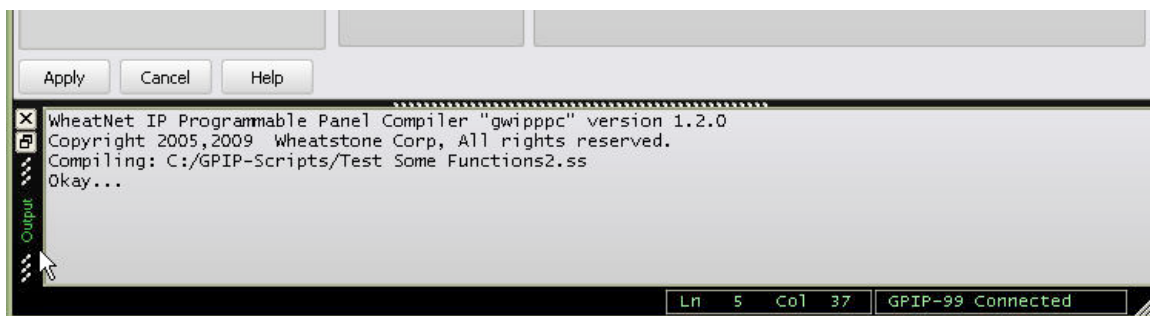
Some GPIIP panel applications may require that a user's subroutine be run at boot time to reset or read values from the system in order to synchronize the panel to the "real world". The *Custom Action Hook* allows you to define this subroutine call. Using this function allows you to easily mix custom actions with script wizard actions. Please see the Help File for Details.

### 3.8 - 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 GPIIP-xx panel. To compile, select the **Build > Compile & Download** menu choice. If this is the first time you have compiled this script, you will be prompted to save it. Give your script a name and click *Save* to complete this process.

If the compile is successful, you will see the following feedback in the compiler "Output" window located at the bottom of the screen.

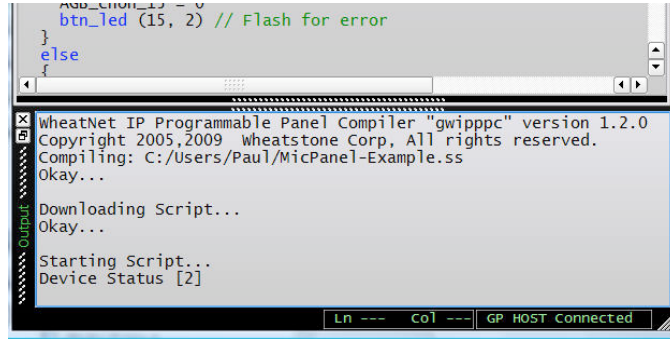
If you do not see the "Output" window on the screen, select the menu Item **View > Output**.



### 3.9 - Starting the Script

When the *Compile & Download* processes are completed the GPIIP panel will automatically start the new script.

Depending on your script you may need to press a button(s) to get the LED's to light up.



Note that once the code is transferred into the GPIIP-xx non-volatile flash memory, it will boot your Script every time the unit is powered up.

### 3.10 - Testing

Now its time to see the results of the code you have downloaded to the GPIIP-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 Navigator 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.

### 3.11 - 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 the *Script Editor Tab*.

Here is a sample Script and its code descriptions:

Wizard code starts here >  
// precedes all Comments.

Button types are listed as >  
Comments

Variables defined >

Startup action calls a timer >  
function to configure startup state.

Auto-generated action code >

```
WheatNet IP GP-16P - C:/Users/Paul/test some functions.ss
File View Build Hardware Help
Script Wizard Script Editor Device Properties

//AG_START
// All code between the AG_START and AG_END tags is auto generated and should not
// be modified
// wheatNet IP Script Wizard - GUI v1.0.0
//AG_BTN1 TYPE="SALVO" PRS="1" REL=""
//AG_BTN2 TYPE="XYC_MOMENTARY" DST="0040022" SRC="0040119"
//AG_BTN3 TYPE="XYC_CONNECT" DST="0040033" SRC="0040300"
//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"
//AG_BTN8 TYPE="SURF_CHON" SURF="1" CH="12"
//AG_BTN9 TYPE="AP_PRESET" AP="1" PSET="7"
//AG_BTN10 TYPE="UMIX_CHON" UMIK="1" CH="7"
//AG_BTN11 TYPE="ACI" DEV="1" PRS="INPUT:7|FADER:192" REL="INPUT:7|FADER:0"

variable: AGB_old_src_2 // Storage for button 2 old source signal
variable: AGB_toggle_5 = 0 // Storage for button 5 toggle state.
variable: AGB_chon_8 // Storage for button 8 mixer ch ON flag
variable: AGB_chon_10 // Storage for button 10 mixer ch ON flag
variable: AG_temp // temporary variable

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

action: AG_TIMER_FUNC
{
    btn_led (6, lio_get (6))
    AGB_chon_8 = surf_get_input_on (1,12)
    if (AGB_chon_8 == -1)
    {
        AGB_chon_8 = 0
        btn_led (8, 2) // Flash for error
    }
    else
    {
        btn_led (8, AGB_chon_8)
    }
    AGB_chon_10 = umix_get_input_on (1,7)
    if (AGB_chon_10 == -1)
    {
        AGB_chon_10 = 0
        btn_led (10, 2) // Flash for error
    }
}
```



# 4 Configuring Device Properties

Some applications may require the GPIIP-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 an Event or turn a channel ON and OFF on a control surface. You might also wish to use the GPIIP panel at a talent microphone location in a studio. These applications require you to “tell” the GPIIP panel some information about the surface and logic signals. This is what the Device Properties form is for.

## 4.1 - Device Properties Tab

While ON LINE and connected to the button panel, open the *Device Properties* dialog by clicking on the *Device Properties* tab in the main window. See the dialog box below.

## 4.2 - Host BLADE Setting

If you intend to use the Soft LIO functions to interface with physical logic signals you will need to identify the BLADE in your system that the GPIIP panel will talk to. Enter the IP address of any BLADE in the system. A BLADE may “host” multiple GPIIP panels.

## 4.3 - Surface Configuration

If you are using your GPIIP button panel to interface with a Wheatstone surface, you will need to identify the IP address of each surface in the *Device Properties* tab of the button panel. The setup steps only need to be performed once since the setup information will be stored in the button panel’s flash memory and also on your PC.

Select the GPIIP device you wish to setup, then make sure you are Connected. Use the “Device Properties” dialog box to specify the surface IP addresses.

Enter the IP address of each mixer in the *Surfaces* tree on the left side of the dialog box.

You may specify up to 8 surface IP addresses. The *Surface 1:* address corresponds to surface ID “1” in a script.

For example, when you specify “Surface 1” in a *Surface Function* in the Script Editor, or a surface-related function in the Script Wizard, the Surface 1 IP address will be used.

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.

The screenshot shows the 'Device Properties' tab of the 'WheatNet IP GP-16P' dialog. The 'Blade' section has an 'IP address' field set to '192.168.87.101'. The 'Surfaces' section lists eight surfaces, with 'Surface 1' set to '192.168.87.11' and 'Surface 2' set to '192.168.87.16'. The 'Audio Processors' section has four fields labeled 'AP 1' through 'AP 4'. The 'LIO Map' section contains two tables: 'Outputs' and 'Inputs', both with 'Soft LIO Index' columns. The 'Outputs' table has indices 1 through 16, with index 4 highlighted. The 'Inputs' table has indices 1 through 16, with index 16 highlighted. At the bottom are 'Apply', 'Cancel', and 'Help' buttons.

**Important Note:** The *Device Properties* controls will be disabled if you are **not** connected to the GPIIP 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.

## 4.4 - Audio Processors

If you are using your GPIIP button panel to control Vorsis Audio Processors, you will need to identify the IP address of each processor in the *Device Properties* tab of the button panel. The setup steps only need to be performed once since the setup information will be stored in the button panel's flash memory and also on your PC.

- Select the Vorsis device you wish to setup.
- Make sure you are Connected.
- Use the *Device Properties* tab to specify the processor's IP addresses.

## 4.5 - Soft LIO Configuration

If you are using a GPIIP button panel to interface with Logic I/O in the WheatNet IP system, you will need to configure Soft LIO's in the *Device Properties* tab. Each GPIIP Panel will store its own *Device Properties* settings.

### *Soft LIO Features*

- Soft LIO's are 64 virtual logic ports residing inside the *Host BLADE*.
- Map soft LIO's to GPIIP buttons for interfacing to physical Blade logic ports.
- Custom scripts can access any of the 64 soft LIO's in a *Host BLADE*.
- Each GPIIP panel utilizes up to 16 inputs and outputs.
- Create logic Source signals in Navigator for any Soft LIO Inputs.
- Create logic Destination signals in Navigator for any Soft LIO outputs.
- Associate Soft LIO's with Audio or Logic I/O Only signals.
- Route Soft LIO's just like hardware logic.
- Use LIO\_HI and LO *Actions* to perform any system function when a specific Soft LIO Output is Hi or Lo.

Note: Some applications may not need the Navigator signals; it all depends on whether you need to route soft logic to/from a GPIIP panel and a BLADE's physical logic ports.

You may map up to 16 Input LIOs and 16 Output LIOs, one for each switch on a GPIIP-16.

Input LIOs correspond to Logic I/O values which are fed IN to the router matrix. Typical types of input LIOs are for switch functions like Start/Stop or ON-OFF-Cough-Talkback remote logic signals associated with a microphone source. In a discrete hardwired system these signals would typically then be fed into a physical logic input line.

Custom scripts for your GPIIP-16 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 GPIIP-16 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 GPIF 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.

# 5 LIO Example Using Soft LIO's

This example describes a method for creating a microphone fader control panel using four buttons on a GPIIP panel. The GPIIP *Soft LIO* mapping feature is used to allocate resources to “virtually” wire up the four switches and two LED signals we will use on the panel. The Soft LIO's mapping the remote control signals will also be assigned to an audio signal using Navigator.

Before we get on with the following example it is useful to understand that there are two primary ways to approach remote control of a surface channel using the GPIIP panel. You can write a custom script using ACI commands to control a *specific fader channel* on a surface or you can use the Device Properties to “point” the GPIIP panel to a *specific source signal*, which has been configured in WheatNet-IP Navigator with Soft LIO logic associations. The ACI based script will only work on a single fader strip while the Soft LIO approach will work on any fader in the system. The path chosen also dictates how your overall script will be written. The former uses surface functions in a custom script while the latter uses the Script Wizard.

For the sake of this example, let's assume that we have a microphone source named “HOST MIC” in our WheatNet-IP system. We will be placing a GPIIP button panel next to the host announcer in a talk studio. We would like to use some of the GPIIP buttons to provide the host with remote ON/OFF, Cough, and Talkback capability. We would also like the GPIIP panel's ON/OFF button LED's to follow the console's fader status.

We will program the following functions:

Audio Signal	This GPIIP panel will map GP buttons 1-4 to remotely control any surface fader that has the the “Host Mic” audio Source signal. Blade Input Signal ID - 00400001				
GP Panel			Soft LIO		
Switch #	Function	Signal Type	Index	Direction	
1	Remote ON	Toggle switch	21	Input	
2	Remote Off	Toggle switch	22	Input	
3	Cough	Momentary switch	23	Input	
4	Talkback	Momentary switch	24	Input	
1	On Tally	External LED	25	Output	
2	Off Tally	External LED	26	Output	

The toggle type Remote ON and OFF switches will have LED indicators that follow the state of the fader channel on the surface. Cough and Talkback are momentary switches that are only active when pressed and have LED indicators driven internally.

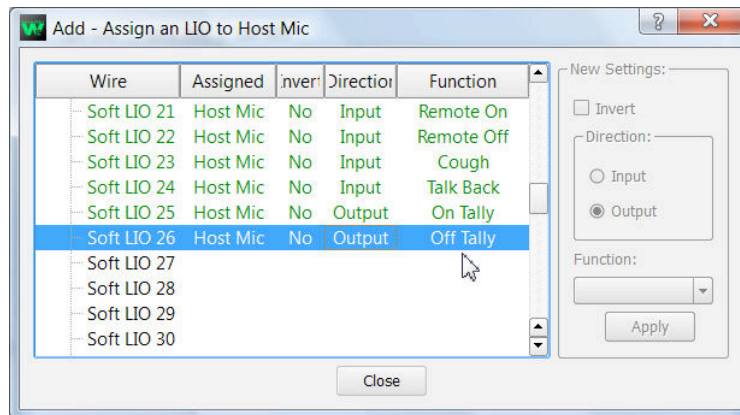
Proceed with the setup in the following order:

- Assign Soft LIO “ports” to the HOST MIC audio signal with WheatNet-IP Navigator software.
- Assign Soft LIO “ports” to the GPIIP panel with WheatNet-IP GP-16P Configuration Tool software.
- Script Wizard is used to generate the script.
- Compile Script and Download to the GPIIP panel.

This approach has two benefits- the resulting script is very clean and the GPIIP panel follows the microphone source signal to whichever surface it is connected to. Let's get started.

## 5.1 - Configure the Source Signal in Navigator

The first thing we need to do is configure the “Host Mic” source signal with some Soft LIO signals to perform the desired functions. The following figure shows how the LIO’s will be defined for “Host Mic” in the Navigator GUI.



### Edit LIO's In Navigator

- Right Click on the Signal Name in the Crosspoint grid.
- Select *Modify Signal*.
- Click *LIO Info* Tab.
- Click *Add*.

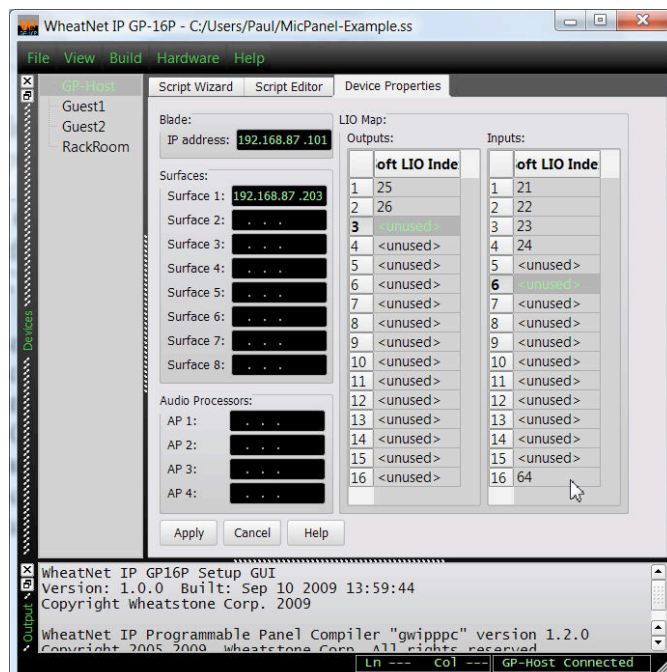
Choose any free Soft LIO's and add the 4 switch inputs and two LED outputs.

## A Bit About Soft LIO's

Sixty four *Soft LIO*'s reside inside each BLADE and are available system wide to any GPIIP panel. When you define a *Host BLADE* in the *GPIIP Programming Tool* you are making the Soft LIO's from that BLADE available to the GPIIP panel. Defining a *Soft LIO* signal only differs from defining a real physical LIO signal in that we do not require real physical hardware for the I/O. *Soft LIO*'s may be mapped in Navigator to audio or logic i/o only signals and routed just like hardware logic ports. A GPIIP panel can access up to 32 soft I/O's in the defined Host BLADE.

## 5.2 - Assign GPIIP Soft LIO's

Let's assume that we want to use the first four buttons on our panel to perform these functions. We need to map the LIO's from the first step above to the panel's first four switches. The Script Wizard assumes a one-to-one correlation between the physical switch on the panel and the *Soft LIO Index* numbers in the LIO Map on the Device Properties tab. For proper operation, we need to match the desired Soft LIO *values* (1-64) with each switch's LIO Index number (1-16). The following figure shows how we will configure the LIO Map properties for this example.



### Assign Soft LIO Indexes

- Make sure you are connected to the GPIIP panel and have selected a *Host BLADE* IP address.
- Define the first four **Input** LIOs to match the Soft LIO's chosen for the Remote On, Remote Off, Cough, and Talkback LIOs for the "HOST MIC" signal.
- Define the first two **Output** LIOs to match the On-Tally and Off-Tally LIOs for the "HOST MIC" signal.
- Click *Apply*.

## Important Distinctions

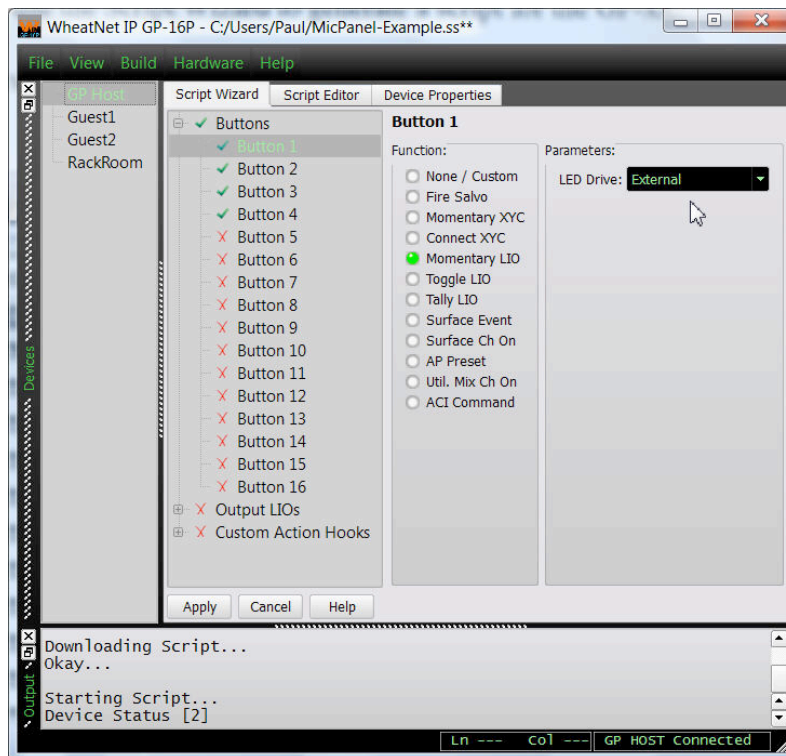
When talking about virtual logic i/o it is easy to get confused over the “direction” – in or out – of the signal. Direction is determined from the point of view of the crosspoint matrix. The following guidelines apply:

- "Input" LIOs are sent **into** the router matrix. These are switch closures from physical logic input ports, GPIP panel switches, or even some Navigator created signals like silence Detect logic.
- "Output" LIOs are sent **out** of the router matrix. These signals may cause a physical output “relay” to close, cause a LED to light on a GP panel, or even provide a virtual output port whose state can be monitored and acted upon in a GP script.

One way to keep it all straight is to consider the switch signals from a GPIP panel as contact closures that must be wired to the crosspoint matrix inputs. Output closures from the crosspoint matrix connect to switch LED’s on a GP panel.

### 5.3 - Create the Mic Control Script Using Script Wizard

Now we want to use the Script Wizard to generate a script for the GPIP panel. All code will be generated automatically. You will be able to see the results in the Script Editor tab later. Using the Script Wizard is easy. Just click on the Button you want to configure, select a function for the button, set any optional parameters, and click Apply at the bottom. Don’t worry if you make a mistake, you can always go back and change or remove button programming selections. Program buttons 1-4 – our ON-OFF-COUGH-TB switches - for *Momentary LIO*. The ON OFF switches will have the LED driven from an external LIO (state of the surface fader channel). Proceed as follows:



- 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
// All code between the AG_START and AG_END tags is
// auto generated and should not be modified.
// WheatNet IP Script Wizard - GUI v1.0.0
//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"
action: STARTUP
{
  tmr_create_periodic (3, "AG_TIMER_FUNC") //check LED states every 300mS.
}
action: AG_TIMER_FUNC
{
  btn_led (1, lio_get (1)) // check Soft LIO Out #1 & light ON led id 1.
  btn_led (2, lio_get (2)) // check Soft LIO Out #2 & light OFF led if 1
}
```

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 shown above.

```
action: BTN_1_PRESS //mapped as ON
{
  lio_set (1,1)
}
action: BTN_1_RELEASE
{
  lio_set (1,0)
}
action: BTN_2_PRESS //mapped as OFF
{
  lio_set (2,1)
}
action: BTN_2_RELEASE
{
  lio_set (2,0)
}
action: BTN_3_PRESS //mapped as COUGH
{
  lio_set (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
{
  btn_led (4,1)
  lio_set (4,1)
}
action: BTN_4_RELEASE
{
  btn_led (4,0)
```

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.

### 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.

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?

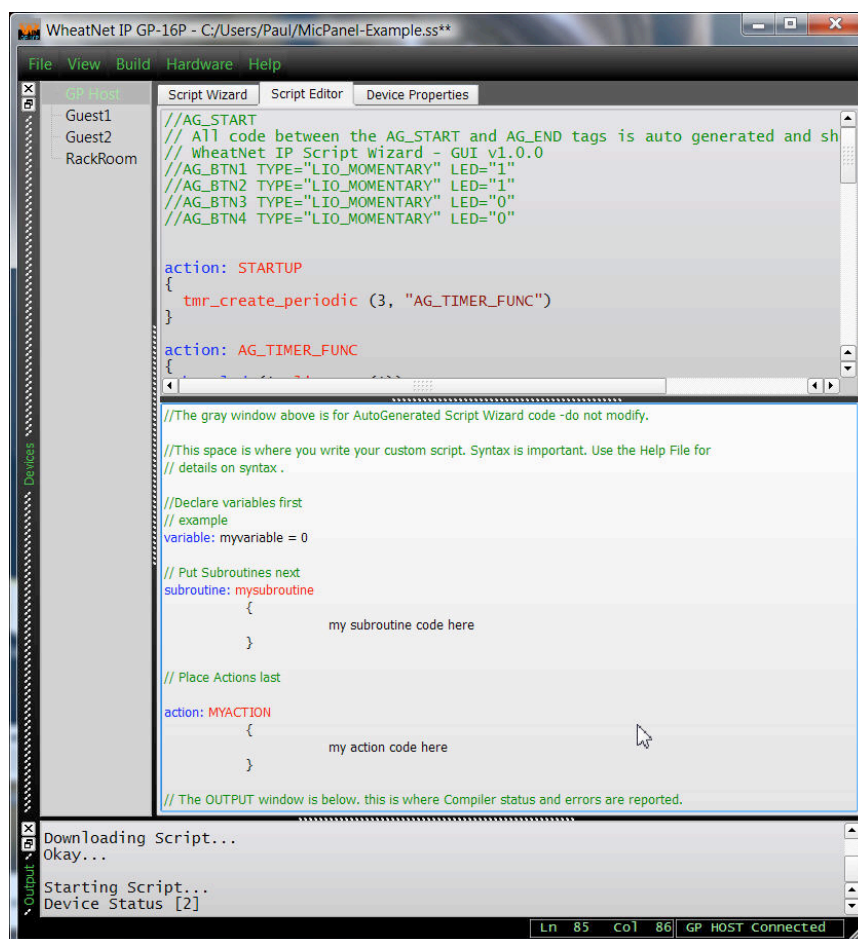
The Script Editor is a specialized text editor built into the GPIIP Programming tool. This editor provides a convenient way to write custom scripts and also view Script Wizard code.

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



Auto-generated code Window

User Script Editing Window

Compiler Output Window

## 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. Also, when using a third party editor, make sure you do NOT make any edits in the area between the *//AG\_START* and *//AG\_END* tags. The editor built into the GPIIP-16P tool prevents you from editing this area, but a third party editor will NOT do so.

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 GPIIP-16P 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.

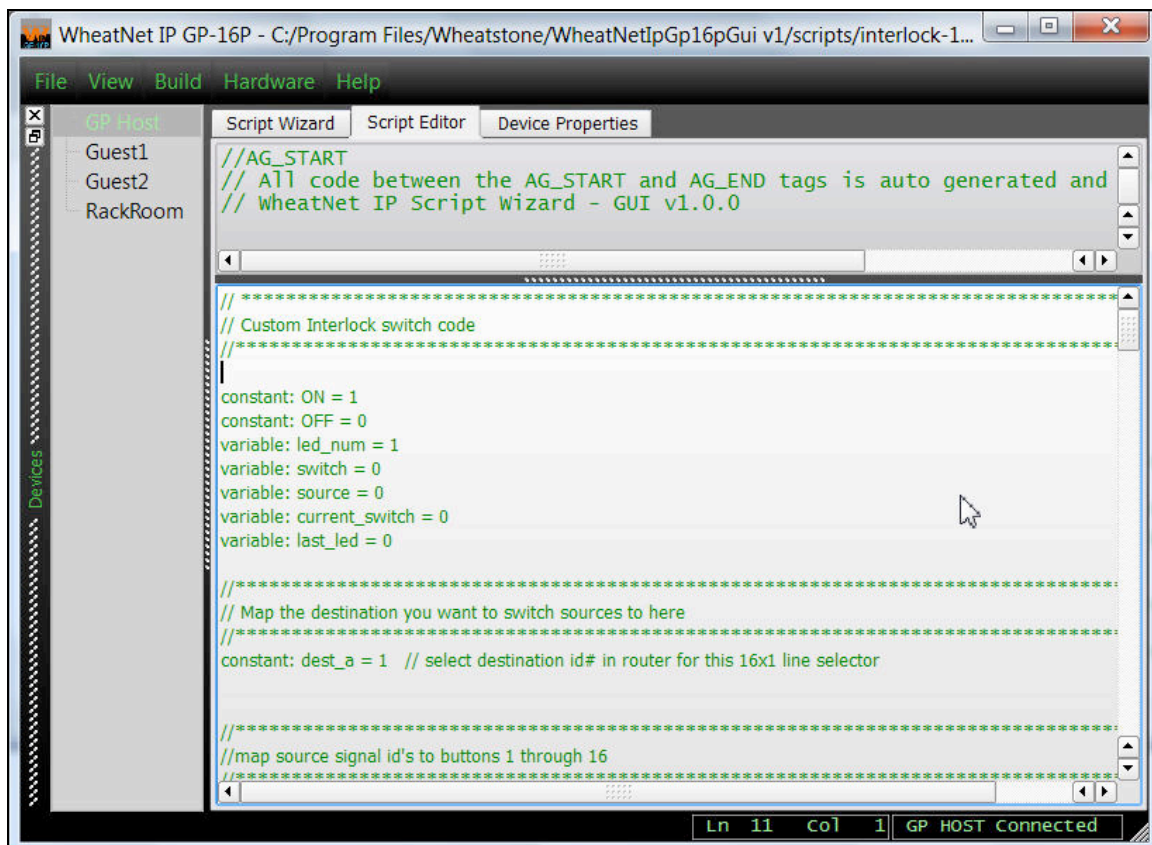
## 7.1 - Getting the Example File

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. If you are reading this from a printed manual you can get this manual in electronic form from the [www.wheatstone.com](http://www.wheatstone.com) website under Application Notes or User Manuals.

## 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" a unique 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".

With the Script Editor Tab open, and assuming you have copied/pasted the example file per Appendix A, you will see the following:





### 7.3 - Auto-generated Script Components

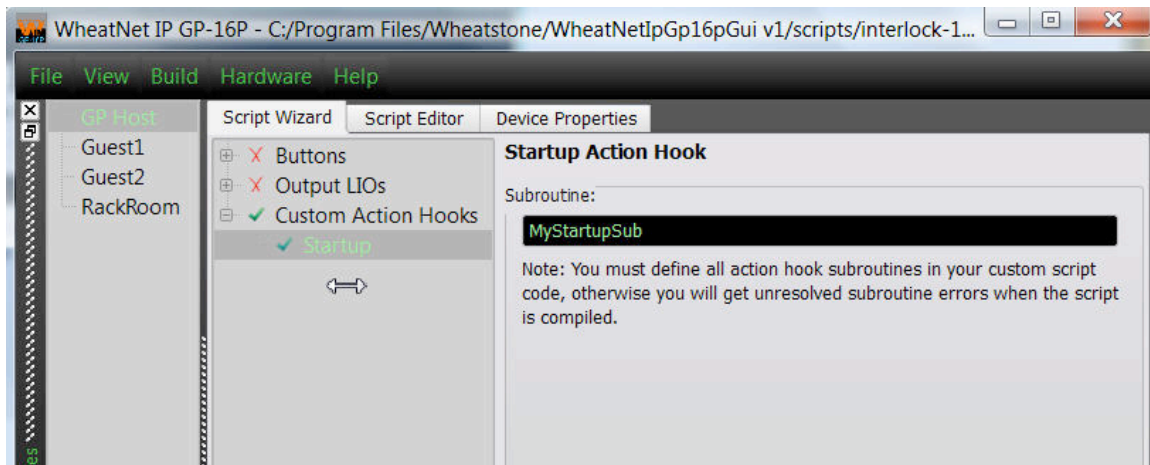
Notice that the first section of the custom script has a few lines of auto-generated code. These are minimum startup lines and must not be altered or deleted.

```
//AG_START
// All code between the AG_START and AG_END tags is auto generated and should not be modified.
// WheatNet IP Script Wizard - GUI v1.0.0
//AG_HOOK TYPE="STARTUP" ACTION="MyStartupSub"
```

### 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 GPIIP panel.

Use the Script Wizard's *Custom Action Hooks* dialog to point to your startup subroutine. In the case below we will call "MyStartupSub" subroutine when the GPIIP panel powers up.



### 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 them 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 come 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 Navigator 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 = 00400001 // select any valid Destination id# in router for this 16x1 line selector

//*****
//map source signal id's to buttons 1 through 16
//*****
constant: source1 = 00400001 //change constants to any other valid Source signal id# as required
constant: source2 = 00400002
constant: source3 = 00400003
constant: source4 = 00400004
constant: source5 = 00400005
constant: source6 = 00400006
constant: source7 = 00400007
constant: source8 = 00400008
constant: source9 = 00400009
constant: source10 = 00400010
constant: source11 = 00400011
constant: source12 = 00400012
constant: source13 = 00400013
constant: source14 = 00400014
constant: source15 = 00400015
constant: source16 = 00400016

```

## 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 the 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 incorrect syntax 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 these and many more, like Utility Mixer, Surface, and Blade control.

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 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\_event – 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 setting, 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\_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

## GPIP-16P Scripting Language Overview

---

The following Script Language overview may be found in the GPIP Configuration 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 GPIP 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 GPIP-16P 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 (i.e. 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 GPIIP-16P 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)
}

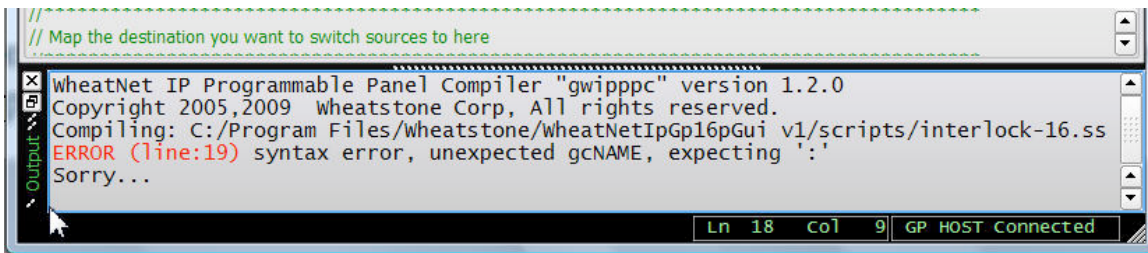
//-----
--
// This action will result in the following message on the console:
// SVM: Hello World
// SVM: Sum = 100
//-----
--
action: test_action
{
    call sum_up_1 ("Hello", " ", "World")
    print ($0)
    call sum_up_2 (10, 20, 30, 40)
    call print_sum ($0)
}
```

# 10 Script Debugging

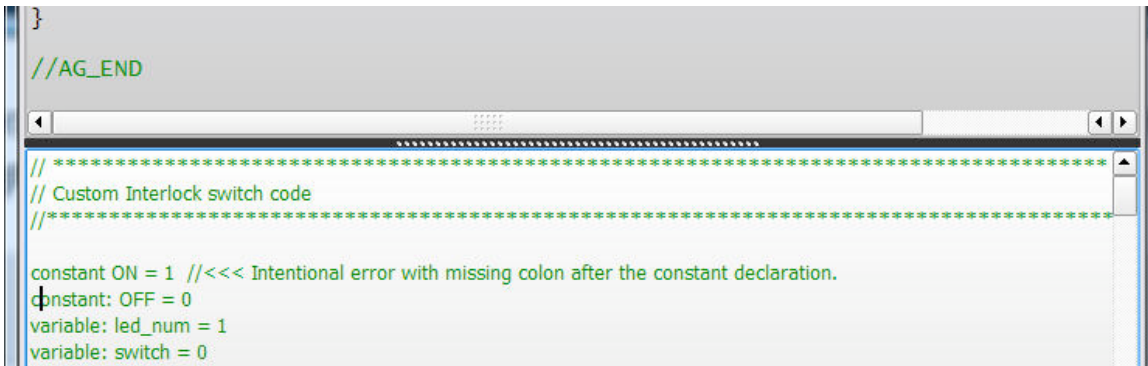
If you have delved into writing your own scripts you will inevitably have to debug them - if only to root out spelling or other minor syntax errors. Programming and debugging go hand in hand. Fortunately there are a couple of very useful tools to aid you in your time of need.

## 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 GPIP tool’s Output 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.



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



The marked line is ok – error is just above it.

## 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 GPIIP 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 GPIIP-16P 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 GPIIP-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 GPIIP panel you need to know three things:

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

Use any Telnet client or open a Command Prompt Window and type:

telnet 192.168.87.221 (or whatever the IP address of your GPIIP-xx panel is).

Toggle the ECHO OFF and enter the user name 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   exit     nds      sstart   svar
mac       telnet  props    sdbg     slvlist  cdbg
dbghdw    ip      dstlist  srclist
srun      sshow

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>
```



```
C:\> Telnet 192.168.8.221

-----
Type "help <command>" for help on a specific command.
Type "!!" to repeat the last command.

->sdbg 1
SCRIPT DEBUG is ON
-> SUM: Subroutine-handle_sw_press
SUM: connecting Source ID 11 to Dest 1.
SUM: Subroutine-handle_sw_press
SUM: connecting Source ID 112 to Dest 1.
```

Now when you press a button on the GPIF-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 GPIIP-16P Configuration Tool do the following:

- 1 - Start the GPIIP-16P Configuration Tool software.
- 2 - Click **File-New...**
- 3 - Click **File-Save as...**, enter interlock16 as the filename, and click **Save**.
- 4 - Select the **Script Editor** tab.
- 5 - Copy and paste everything between the *//START HERE* and *//END HERE* lines directly into the bottom window of the Script Editor.
- 6 - Click **File-Save**.

```
//*****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 = “00400001” // select YOUR destination id# in router for this 16x1 line selector.

//*****
//map source signal id's to buttons 1 through 16
//*****
constant: source1 = “00400001” //change each Source signal id# as required for YOUR system.
constant: source2 = “00400002”
constant: source3 = “00400003”
constant: source4 = “00400004”
constant: source5 = “00400005”
constant: source6 = “00400006”
constant: source7 = “00400007”
constant: source8 = “00400008”
constant: source9 = “00400009”
constant: source10 = “00400010”
constant: source11 = “00400011”
constant: source12 = “00400012”
constant: source13 = “00400013”
constant: source14 = “00400014”
constant: source15 = “00400015”
constant: source16 = “00400016”
//*****
```

```

// 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 ****
```