

Building a Simulator with The Extended Programmable Input/Output Controller (EPIC)



Developer's Manual

**Compiled by:
Johnny "JAM01" Martin
June 2003**

Table of Contents

<u>TABLE OF CONTENTS</u>	I
<u>INTRODUCTION</u>	1
<u>PART 1: HARDWARE ARCHITECTURE</u>	3
<u>1.1 R&R EPIC Hardware Descriptions</u>	3
<u>1.1.1 EPIC Expansion Module</u>	3
<u>1.1.2 EPIC 32 Point Output Module</u>	4
<u>1.1.3 EPIC 32 Digit Display Module</u>	4
<u>1.1.3 EPIC Rotary Module</u>	4
<u>1.1.4 EPIC Gauge/Meter Module</u>	4
<u>1.2 Micro Cockpit hardware</u>	5
<u>1.2.1 Micro Cockpit ABA Module</u>	5
<u>1.2.2 Micro Cockpit 64BTN Module</u>	5
<u>1.3 CSI Cockpit Simulations Hardware</u>	6
<u>1.3.1 CSI EMDA Module</u>	6
<u>1.3.2 CSI Pro32/40 Module</u>	7
<u>1.4 Cockpit Panels</u>	8
<u>1.4.1 Advance Graphics Technologies L.L.C.</u>	9
<u>1.4.1.1 F-16 Panels</u>	9
<u>1.4.1.2 Commercial Aircraft Panels</u>	10
<u>1.4.2 Aimsworth Simulations</u>	11
<u>1.4.2.1 F-16 Panels</u>	11
<u>1.5 Cockpits</u>	12
<u>1.5.1 Homebuilt Cockpits</u>	12
<u>1.5.2 Commercial Cockpits</u>	12
<u>PART 2: EPIC INFRASTRUCTURE</u>	13
<u>2.1 Powering Modules</u>	13
<u>2.1.1 Distributed power</u>	13
<u>2.1.2 Central power</u>	13
<u>2.1.3 Power distribution</u>	13
<u>2.2 Mounting Modules</u>	14
<u>2.2.1 Open Mounting</u>	14
<u>2.2.2 Central Mounting</u>	14
<u>2.2.3 Modular Mounting</u>	14

<u>2.3</u>	<u>Wiring to Modules</u>	15
2.3.1	<u>Distributing Power</u>	15
2.3.2	<u>Wiring modules</u>	15
2.3.3	<u>Wiring panels</u>	15
 <u>PART 3: HARDWARE AND SOFTWARE INSTALLATION</u>		20
<u>3.1</u>	<u>Keyboard Support</u>	20
<u>3.2</u>	<u>Multiple EPIC USBs</u>	20
<u>3.3</u>	<u>EPIC Hardware Installation</u>	20
3.3.1	<u>Installing the EPIC USB drivers</u>	20
3.3.1.1	<u>EPIC USB LEDs</u>	21
3.3.2	<u>Software for R&R EPIC Modules</u>	21
3.3.3	<u>Software for Micro Cockpit Modules</u>	22
3.3.4	<u>Software for CSI Modules</u>	22
 <u>PART 4: SOFTWARE ARCHITECTURE</u>		21
<u>4.1</u>	<u>Aftermarket Software</u>	21
4.1.1	<u>EPICmapper</u>	21
4.1.2	<u>Flight Simulator Universal Inter-Process Communication (FSUIPC)</u> ...	21
<u>4.2</u>	<u>Homegrown EPL Software</u>	21
4.2.1	<u>EPICenter</u>	21
4.2.2	<u>Updating EPIC software</u>	22
4.2.3	<u>Installing new EPICenter software</u>	22
4.2.4	<u>Downloading EEPROM software to EPIC</u>	23
4.2.5	<u>Compiling EPL to EPIC</u>	23
4.2.6	<u>Downloading EPL to EPIC</u>	24
<u>4.3</u>	<u>EPL Software</u>	24
4.3.1	<u>EPL Program Syntax</u>	24
4.3.2	<u>EPL Program Structure</u>	25
4.3.2.1	<u>Main EPIC Project File: <name>.epf</u>	25
4.3.2.2	<u>Assignment of analogs and modrows: mydevices.hpl</u>	27
4.3.2.5	<u>Assignment of hardware devices: devices.hpl</u>	30
4.3.2.5	<u>Assignment of procedures to hardware devices: procedures.hpl</u>	31
4.3.3	<u>EPL Commands</u>	39
4.3.4	<u>Pre Processor Directives</u>	47
4.3.5	<u>Macros</u>	48
4.3.6	<u>Data Types</u>	48
4.3.7	<u>Device Descriptors</u>	49
 <u>PART 5: SAMPLE PIGEONHOLE CODE</u>		21

<u>5.1</u>	<u>C++ software to interface to Pigeon Holes and Q-Procs</u>	21
<u>5.1.1</u>	<u>Generic Pigeon Hole code</u>	21
<u>5.1.2</u>	<u>Generic Q-Proc code</u>	21
<u>5.1.3</u>	<u>Falcon4SP3 Memory Reader: JAMmemreader Project</u>	21

Introduction

When I bought my EPIC card a couple of years ago and starting reading the web site forums, one theme came across very clearly: “there was insufficient documentation on EPIC.” This goal of this manual is to provide a single place for the new users of EPIC to gather information to help them develop their simulator systems.

This manual will not answer all of your questions, nor will it address all configurations or uses of EPIC hardware or software. Hopefully, this manual does provide enough basic information for the novice to design a simulator architecture; select and install an EPIC hardware suite; and write the EPIC software that provides the level of enjoyment they need from their simulator system.

Before we start there is some basic terminology you need to know. You should know that EPIC basically listens for external events (Input Events) from buttons, switches, joysticks, or even computer programs and reacts to them by executing code that we’ll call Procedure Blocks. EPIC can also cause things to happen in the outside world (Output Events) by causing lights to light, buzzers to buzz, motors to turn etc., and these can be initiated by the Input Events discussed earlier.

The Procedure Blocks are part of the overall EPIC software Project you will create in a language call EPIC Programming Language or EPL. The EPIC developers make it easy for you to write EPL by giving you an integrated development environment (IDE) call EPICenter. With EPICenter you’ll be able to write, compile and download EPL to EPIC. You’ll even be able to update the EPIC’s Electronically Erasable Programmable Read Only Memory (EEPROM) using the EPICenter and new EEPROM code provided by the EPIC developers.

Disclaimer:

This manual is a compilation of information from various sources and is provided to the simulator community for the personal use of individuals or companies developing simulators around the EPIC system. It may not be sold, but may be provided free of charge to aid users of EPIC and related hardware and software.

Trademarks:

The following trademarks may be registered by their manufacturers with all rights reserved:

R&R Electronics:	EPIC	
EPICMapper:	EPICMapper	
Micro Cockpit:	ABA	64BTN

CSI Cockpit Simulations: EMDA
HASBRO:
Advanced Graphics Technologies, L.L.C.
Aimsworth Simulations:

PRO34
Falcon4
F-16, Airbus, B747 Panels
Aimsworth Cockpit

- The document is only used for non-commercial purposes.
- The document is only used for informational purposes.
- Any copy of this document, or portion thereof, must include this copyright notice.

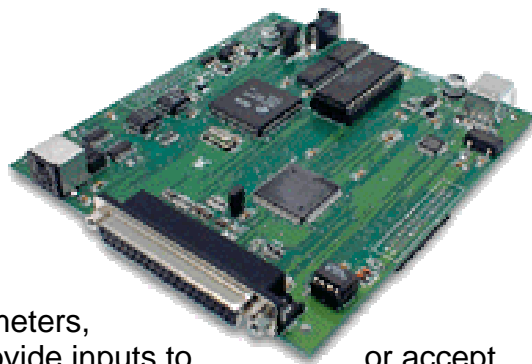
This documents is provided "as is", without warranty of any kind, either expressed or implied. This publication may include typographical or technical inaccuracies. Information in this document may periodically change without prior notice.

Part 1: Hardware Architecture

The follow information is provided to help the simulator builder choose a hardware architecture for their simulator. This section and this entire document is written around the EPIC system ([EPIC Home Page](#)) and the modules designed to connect to EPIC and the software written to work with EPIC. I will attempt to include information on third-party EPIC hardware and software when I can find it.

1.1 R&R EPIC Hardware Descriptions

[EPIC](#) is the **Extended Programmable Input/Output Controller** . EPIC USB is a USB interface card designed for computer enthusiasts, home cockpit builders, and for specialized analog/digital I/O applications. EPIC USB, when used with the appropriate modules, allows the use of analog potentiometers, switches, LED's, motors, meters, etc., to provide inputs to or accept outputs from a computer program such as a flight simulator. Using EPIC USB's advanced interface, you can read the position values of analogs 0 through 15 in approximately 28 μ s. EPIC USB also supports bi-directional command and data queuing and can generate hardware interrupts. Button events can be queued by EPIC USB until retrieved by a program, and programs can "fake" button events, initiate execution of EPL procedures, and transfer data to the EPIC for autonomous processing by multiple "smart" modules.

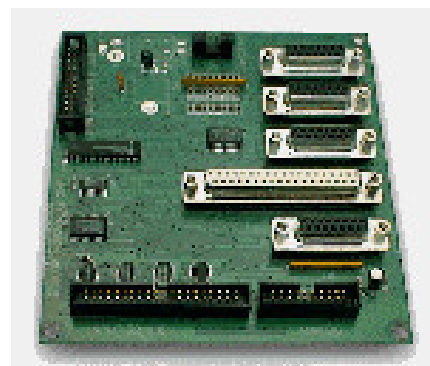


Features:

- Control up to 22 analogs (potentiometers) when used with the EPIC [Expansion Module](#)
- Control up to 255 buttons (when used with the EPIC [Expansion Module](#))
- Completely programmable using the [EPICenter](#) environment to produce EPL code
- USB Plug and Play interface (compatible with Windows 2000 and XP)

1.1.1 EPIC Expansion Module

The EPIC Expansion Module provides connectors for analog potentiometers, switches and the EPIC Bus. The EPIC Bus is used to connect additional EPIC modules (such as the Output Module) to EPIC USB. Standard analog and Thrustmaster joysticks can connect to EPIC USB via the DB15 connectors on the Expansion Module; however, some rewiring of the joystick may be required.

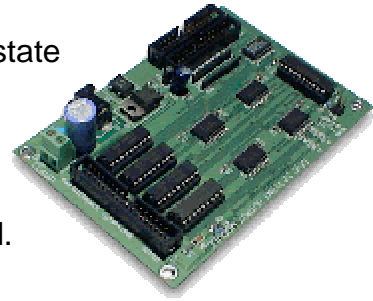


Features:

- Access to 16 analogs (using a 16 pin connector)
- Access to 255 buttons (using a 40 pin connector)
- Access to the EPIC Bus (using a 26 pin connector)

1.1.2 EPIC 32 Point Output Module

The 32 Point Output Module allows up to 32 dual state (off/on) devices to be controlled by the EPIC USB. Output devices include light emitting diodes (LEDs), incandescent lights, buzzers, and even stepper motors. Multiple Output Modules can be daisy chained if more than 32 outputs are required.

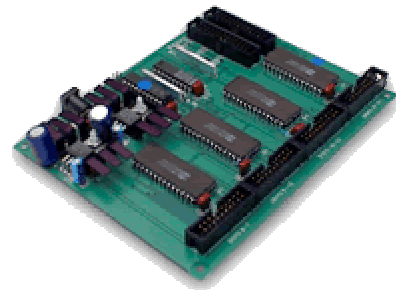


Features:

- Control 32 dual state functions per Output Module
- Daisy chain to increase the number of outputs beyond 32

1.1.3 EPIC 32 Digit Display Module

The EPIC 32 Digit Display Controller allows the EPIC USB to drive up to 32 numerical LED displays. These displays can be used in any panel which requires numerical readouts.

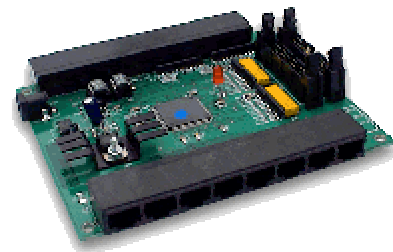


Features:

- Control up to 32 LED numerical digits

1.1.3 EPIC Rotary Module

The EPIC Rotary Module allows you to connect up to 16 mechanical and 4 optical rotary switches to the EPIC USB. Additional Rotary modules can be added if additional inputs are required.



Features:

- Access to up to 16 mechanical and 4 optical rotary switches

1.1.4 EPIC Gauge/Meter Module

The EPIC Gauge/Meter Module allows you to connect up to 8 instruments to the EPIC USB. Trim potentiometers mounted on the board allow for precise calibration of each instrument.

Features:

- Control up to 8 instruments simultaneously

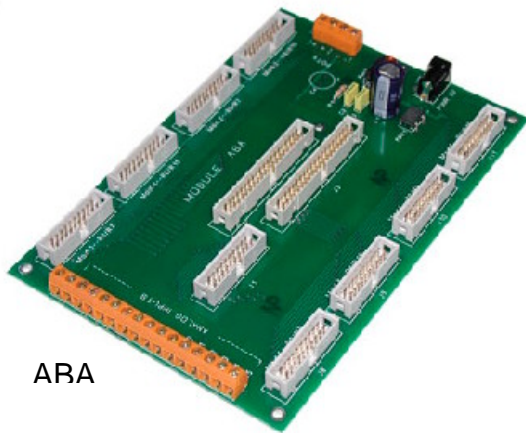
1.2 Micro Cockpit hardware

1.2.1 Micro Cockpit ABA Module

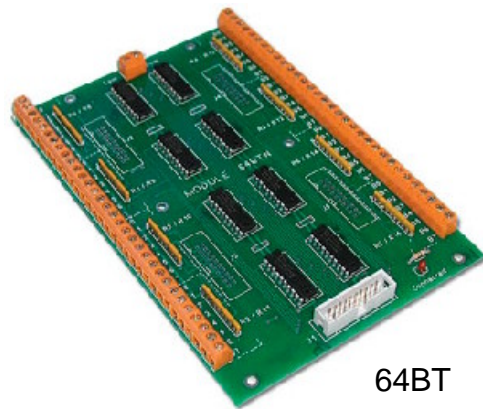
The Micro Cockpit ABA and 64BTN modules simplify EPIC wiring systems significantly ([Micro Cockpit Home Page](#)). The ABA module connects to the EPIC Expansion Module and provides screw-type connectors for 16 analog channels and 8 connectors for 64BTN Modules (discussed later). A screw connector provides 5Vdc power (supplied by EPIC USB) to supply the potentiometers (10 microamps 100K ohms). An external power supply is needed for the ABA only if more than 50 switches will be closed at one time across the 8 64BTN modules. Only one ABA Module (and therefore only 8 64BTN Modules) can be connected to an EPIC Expansion Module.

Features:

- Access to 16 analog channels
- Connectors for 8 64BTN Modules
- Onboard connector for power connection



ABA



64BT

1.2.2 Micro Cockpit 64BTN Module

Up to eight 64BTN Modules can be connected to the ABA Module at one time. Each 64BTN Module provides access to 64 (ModRowBit) digital inputs to EPIC USB. By using the ABA/64BTN combination, you may not need to use diodes, which should simplify your wiring job. Your wiring system should be very simple: you just need to connect all the common wires of each button to the screw connector marked "Common" on the 64BTN, then, connect the other pin of each button to one of the eight groups of screw connectors marked 0 to 7, on the 64BTN (no diodes required). With this wiring system, you can have wire runs longer than 100 feet, between your buttons and the 64BTNs. Since you can connect a maximum of 8 x 64BTNs to the ABA, you

can have a total of 512 digital inputs. 48 more digital inputs can be wired to the Expansion module DB15 connectors (FLCS, RUDDER, A, B) if you need them.

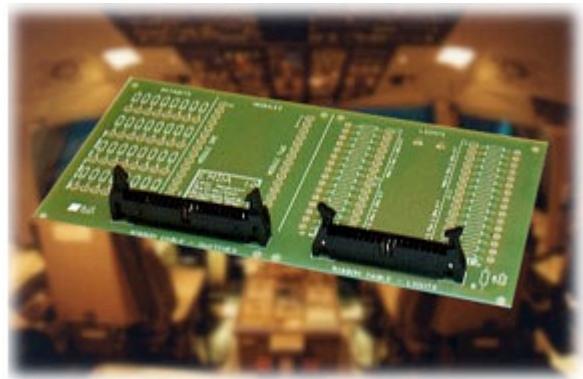
Features :

- Access to 512 digital inputs
- No diode required for switches connected to the 64BTN
- Up to 8 64BTNs can be connected to an ABA Module
- Power for the 64BTNs is supplied by ABA Module

1.3 CSI Cockpit Simulations Hardware

1.3.1 [CSI EMDA Module](#)

The EPIC Modular Distribution Architecture (EMDA) works in conjunction with the [EPIC](#) card to provide a truly modular and simple solution to the complex wiring job faced by any cockpit builder. The EMDA solution is also an easy way to make wiring your EPIC based cockpit affordable. The newest version of the EMDA card includes an extra set of plugs, that allow daisy-chaining several cards. Since diodes are soldered onto the EMDA card, they no longer have to be soldered directly in line with the wiring or your switches. Everything is kept [neat and orderly](#) on the EMDA Wiring Interface Card. The same goes for resistors used in line with LEDs on the lighting circuits. Resistors can be soldered directly to the EMDA Wiring Interface Card, saving frustration both in assembly and in later troubleshooting.



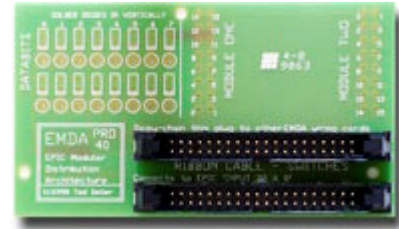
Read our online [PDF documentation](#) for detailed information on using these Wiring Interface Cards.



The modular concept means EMDA cards can be easily daisy-chained as your needs grow. Each EMDA breakout card along a ribbon cable provides connections for 32 switches. The 34-wire ribbon cable for lights can also be attached to these EMDA Wiring Interface Cards, providing access to any of 32 lights at each card location. When you need more lights or switches, just add another length of ribbon cable, and another EMDA card. The idea is to keep the spaghetti-bowl wiring to a minimum. Using two or three of the EMDA

distribution cards along a length of ribbon cable allows you to keep the breakout cards near specific groups of switches. For example, a

communications panel with four rotary switches and eight LEDs can be operated with one EMDA Wiring Interface Card. The landing gear and autobrake panel further down the line can be operated with another EMDA Wiring Interface Card. If either cockpit panel needs to be repaired or replaced at a later date, the whole panel -- including its EMDA Wiring Interface Card -- can simply un-plug from the main ribbon cable, and another panel can be plugged in. True modularity!



The EMDA Wiring Interface works in conjunction with the *EPIC* card, manufactured by R&R Electronics.

1.3.2 CSI Pro32/40 Module

A revolution in simplicity! Our modular solution to cockpit wiring just got even better. The EMDA PRO-Series is designed to fit into small spaces, and the cramped quarters behind actual airliner cockpit panels. These breakout cards are just 6.0 x 10.6 cm (2 3/8 x 4 3/16 inches), and yet they offer almost all of the functionality of a full-size EMDA V2 card! The PRO-40, provides up to 16 easy connections for switches, including the diodes required for their operation with the EPIC system. To save space, the diodes can be mounted vertically instead of horizontally as you would find on a full-sized EMDA card.





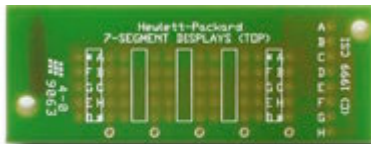
The PRO-34, provides connections for all 32 outputs available from the EPIC Output Module. You can wire up 32 lamps, LEDs, buzzers or what-have-you right from this tiny 6.0 x 10.6 cm (2 3/8 x 4 3/16 inch) card. To help save space, we have designed the card so that resistors (if you need any), are mounted vertically.

FEATURES:

- Daisy-chaining plugs on every EMDA wiring interface card
- Full backward-compatibility as well with older EMDA cards

1.3.3 CSI 7-Segment Displays

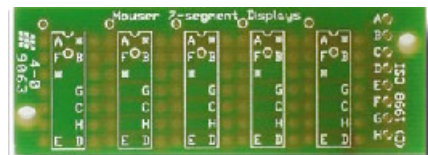
These little circuit boards take all the hard work out of wiring seven-segment LED displays. All you need to add are the numeric displays of your choice.



We offer two versions of the wiring card, accommodating the [most popular seven-segment LED choices](#) among cockpit builders (the QT MAN3xxxA series and Hewlett-Packard HDSP-series).

These displays come ready for you to solder on five digits. If you only require three digits and need to keep space to a minimum, just cut two of the digits off the left side of the card, using a hacksaw or table saw. The basic mounting card is 2.3 x 6.2 cm (29/32 x 2 1/2 inches)

Strip apart the EPIC Display Card's ribbon cable and solder in the segment wires at the right hand end of the display mount. The holes are arranged in the same order as the wires on the ribbon (A-H), to keep the whole operation neat and simple to solder.



The commons to each digit go in the holes marked with a donut around them (just above and to the left of the letter A on each digit). That's all there is to the wiring!! These Seven-Segment Display Mounts are designed to work in conjunction with the *EPIC* Seven-Segment Display Module, manufactured by R&R Electronics.

1.4 Cockpit Panels

The advantage of using EPIC is the ability to send keystrokes or data to your simulator program or use data from your simulator. To do this, you need

some physical hardware or panels. I use the term panels as a generic term since panels come in all types: lighted enunciators, gangs of switches, gauges, displays and numerous combinations of the above. One choice you'll need to make is what panels you need and how you will get them. I've included some commercial panels below to give you a feel for what's available on the market, some of these companies will even sell you the panel without the hardware or make custom panels to your drawings.

1.4.1 [Advance Graphics Technologies L.L.C.](#)

1.4.1.1 F-16 Panels

F-16C Caution Lights Panel

Dimensions

- Width: 114 mm (4.5 in)
- Height: 83 mm (3.25 in)
- Thickness: 3.175 mm (0.125 in)

Panel Hardware:

- Lighting panel, which allows each annunciator to be illuminated independently of the others.



F-16C Integrated Control Panel (ICP)

Dimensions

- Width: 140 mm (5.5 in)
- Height: 105 mm (4.125 in)
- Thickness: 3.175 mm (0.125 in)

Panel Hardware:

- 12 illuminated, square pushbutton switches
- 6 illuminated, round pushbutton switches
- 1 nonilluminated, black pushbutton switch
- 3 miniature toggle switches
- 2 rocker switches
- 4 white thumbwheels (Not shown. These thumbwheels can be made functional with the addition of rheostats behind the panel.)



F-16C Landing Gear Panel

Dimensions

- Width: 202 mm (7.9 in)
- Height: 171 mm (6.7 in)
- Thickness: 3.175 mm (0.125 in)

Panel Hardware:

- 3 standard toggle switches
- 3 lever lock toggle switches
- 3 green indicator lamps (12VDC)
- 3 round pushbutton switches
- Landing gear lever with integral switches (not shown)

1.4.1.2 Commercial Aircraft Panels

Airbus EFIS Panel

Dimensions

- Width: 177.8 mm (7.0 in)
- Height: 101.6 mm (4.0 in)
- Thickness: 3.175 mm (0.125 in)

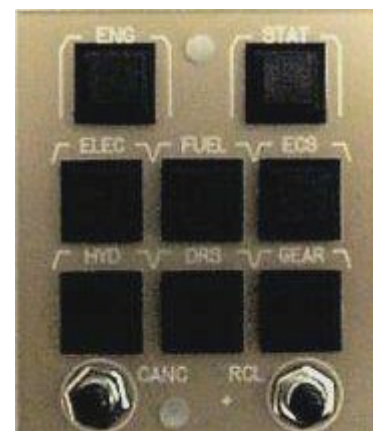
Panel Hardware:

- 7 non-illuminated pushbutton switches
- 3 black knobs (can easily be painted to match knobs shown)
- 4 LED displays
- 1 miniature toggle switch
- 2 toggle switches
- 1 rotary encoder with integral pushbutton switch
- 2 6-position rotary switches



747-400 EICAS Panel

Dimensions



- Width: 63.50 mm (2.50 in)
- Height: 76.20 mm (3.00 in)
- Thickness: 3.175 mm (0.125 in)

Panel Hardware:

- 10 non-illuminated pushbuttons

1.4.2 [Aimsworth Simulations](#)

1.4.2.1 F-16 Panels



1.5 Cockpits

1.5.1 Homebuilt Cockpits

Building your own cockpit can be as simple or as complex as you want to make it. Your choice of materials will determine how hard a job your cockpit construction will be. Using fiberglass or sheet metal may give a better or more realistic look and feel; however, plywood is easier to cut and cheaper to buy. Whether you decide to construct a cockpit or purchase one, you need to think about how you'll be routing your wires and whether all of your panels will fit the way you want. I use PVC pipes to route my wiring and I fully built my cockpit before I made any cutouts for panels.



1.5.2 Commercial Cockpits

Commercial cockpits are available and range for frontal only to a full cockpit with canopy. [Aimsworth](#) is one company that sells F-16 cockpits in the US and overseas.



Part 2: EPIC Infrastructure

2.1 Powering Modules

2.1.1 Distributed power

Many of the EPIC modules and add-on hardware require between 9 and 12 volts dc. You can purchase individual wall plug type power supplies and use one per EPIC module and one for each Output module for the lights. This distributed power systems has the obvious advantage of allowing you to purchase your power supplies and simply plug them into your EPIC modules. A disadvantage of the distributed is system is the need to run 120 volts alternating current (vac) to each of these power plugs throughout your simulator.

2.1.2 Central power

Another way to power you EPIC system is via a centralized power distribution systems. With a centralized system, you might have a single power supply capable of handling all of your power needs and you would run a cable to each system requiring power. With a central system, only 12vdc is run throughout your simulator and by using fuses, short circuits can be handled without much danger. Another advantage to the central system is the ability to get used personal computer power supplies that would have sufficient +12vdc and +5vdc power available to run your EPIC system.



2.1.3 Power distribution

Whether you choose distributed or centralized, you need to determine how you'll get the power where it needs to go. You need to consider how to get 120vac and 12vdc/9vdc around your simulator. You need to ensure you keep your power cable runs away from sharp objects or pinching objects that could damage the insulation and cause a short circuit. You should also consider keeping all of your power connections protected by shielding and providing fuses on all power cables.

2.2 Mounting Modules

All of your EPIC modules need to be mounted to something (or in something). Whether you choose to mount them in a PC-like cabinet together, or mount them free in your cockpit or in separate boxes, you need to do some planning before you start wiring things up.

2.2.1 Open Mounting

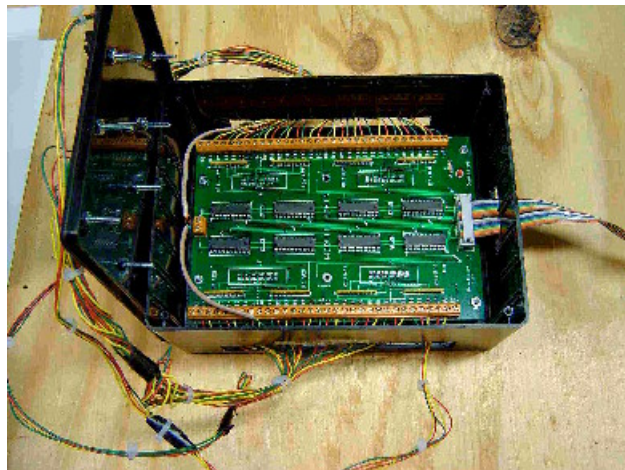
I've read some forum postings on the advantages of mounting your EPIC Modules against the walls of the simulator. This has the advantage of free access to all connectors, good airflow and it makes it easier to find a location since EPIC Modules can be placed wherever free space can be found. The major disadvantage for me would be the danger of accidental damage and the need to feed power to Modules located all around the simulator. My fear with this mounting concept is that something would come loose and fall on a Module or a liquid would spill and short something out.

2.2.2 Central Mounting

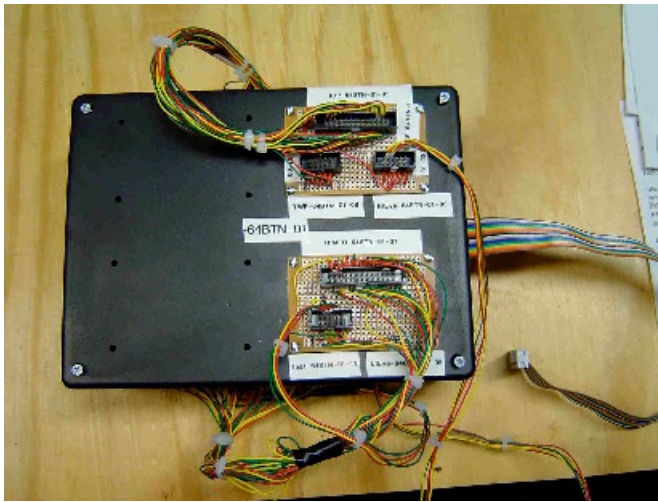
Central mounting involves mounting all of your EPIC Modules in a single case, like a computer case. One of the advantages to central mounting is it is easier to connect all of your Modules since they are all in one case and you'll have access to a 12vdc power supply. Another advantage is your Modules have protection for damage and probably have a ventilating fan to help cool them. The main disadvantage with Central mounting is all of your panel wiring must come to one enclosed location to feed all of your Modules. Unless you plan wisely, this could become very cumbersome during construction, maintenance and troubleshooting.

2.2.3 Modular Mounting

I chose modular mounting of my Modules. I placed one or two Modules in a case (I used Radio Shack plastic cases) and cut holes for the wires or connectors. I also mounted connectors to the cases for my panels to plug into. This mounting concept allows me to remove my modules for testing or repair, without removing any of my simulator wiring. I also plan to mount most of my Module cases centrally (under my pilot seat) to allow easy connection of the power. Mounting my modules



under my seat is ideal since most of my panels are located to the left or right of the seat anyway. For my lights, I'm mounting my EMDA PRO34 Modules on boards and mount the boards in cases. I'll place the PRO34 cases on the left, right, and front walls of my simulator, using ribbon cable running back to the Output Modules located under the pilot seat. The major disadvantage of this mounting concept is, it provides the worse air flow of the three discussed because all the Modules are in small enclosed cases with poor circulation.



2.3 Wiring to Modules

2.3.1 Distributing Power

The main consideration here is how you plan to run your wiring around your simulator. If all of your wiring is simply laying on the floor, it may be difficult to troubleshoot or make repairs down the road. By using some form of conduits, you can keep your wiring organized and protect it from damage. You also need to decide how you will distribute the power to your EPIC, and it's Modules.

2.3.1.1 If you're using a distributed power architecture, you'll need to supply 120vac (US) power throughout your cockpit to power all of your individual power supplies. Running 120vac is probably easier since you can buy as many wall-style power supplies as you need and use a power strip in a central location to power them.

2.3.1.2 If you're using a central power architecture, you'll need to design a power box that has the connections for all of your EPIC Modules. My power box has a small circuit card mounted inside with 2-pin connectors for +12vdc and ground. The 2-pin connectors allow me to connect a cable that has the EPIC power plug (center negative) on the other end and I run these cables to each of my EPIC modules.

2.3.2 Wiring modules

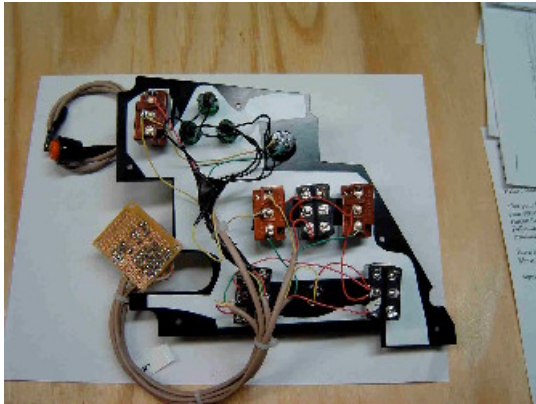
I plan to locate all of my EPIC modules in one place so connecting them to my power box won't require very long cables. Also, since all of the EPIC Modules come with pre-made 1-foot cables, centralizing their location makes connecting them easier. You'll need to decide where you will place your EPIC Modules so you can determine if the ribbon cable length will cause problems for your EPIC hardware.

2.3.2.1 In my simulator, any wiring that goes to any module, but not to one of the Module's connectors, goes to a small circuit card that has an attached to a connector. This way, all connections to every one of my EPIC Modules has a physical connector. This wiring system makes my EPIC Modules (mounted in their cases) more like the line replaceable units (LRUs) in a real aircraft.

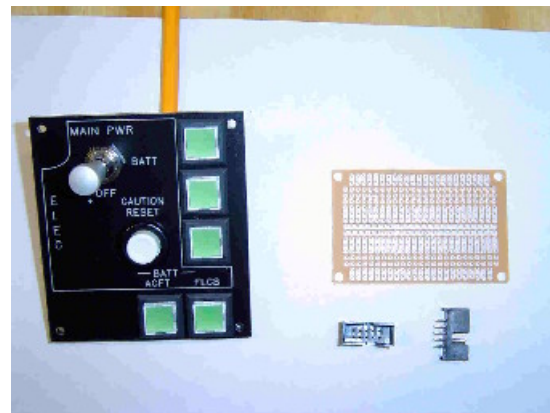
2.3.3 Wiring panels

Modularity and ease of troubleshooting are a goal of my simulator design. For this reason, I wanted all of my panels to be removable without much hassle. To accomplish this, I run wires from all of my switches, lights, rotaries, etc., from each of my switch panels, to a small circuit card where the

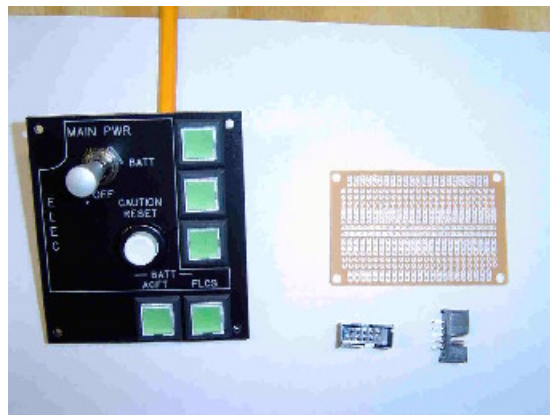
wires are attached to a multi-pin ribbon cable connector. Switches and lights go to separate connectors since they will terminate at different EPIC Modules. I use the ABA/64BTN for switches and the EMDA PRO34 for my lights. This wiring system allows me to disconnect a panel very easily and troubleshoot or test the individual components without disturbing the wiring inside my simulator.



2.3.3.1 Wiring panels is one of the most time consuming and critical parts of building a simulator in my view. If not done properly, it will lead to a lot of time troubleshooting when something stops working. Below is a chronology of the wiring of my F-16 Main Power panel. I hope this chronology gives you some ideas of how you should perform this critical step when building your own simulator. A good place to start is to gather up the parts you'll need just to make sure you don't have to stop to order a part in the middle of your wiring project.



2.3.3.2 Before touching any hardware, I made a wiring plan for my panel. Below is an excerpt from my LeftCableTrack.xls excel spreadsheet. As you can see, I track the position of the switch on the panel; the actual cable and wire connected to the switch; the cable wire position on my ribbon cable connector; and the module, row, and bit of the switch on my 64BTN. All of this data can be



predefined once I know how many, and what type of switches are on a panel and therefore what size ribbon cable I'll need. I also keep my rotary encoders, potentiometers and lights on separate connectors so I must know how many of these I have also. For a given panel, I will have from 1 to 4 connectors, depending on what devices are included on the panel. Here are a couple of pointers:

- a. An on/off switch or two-position rotary only needs one row in the table
- b. A multi-position rotary (example: 4-position rotary) needs the same number of rows, as there are positions on the rotary (except for a 2 position rotary that only needs 1 row)
- c. When using the 64BTN, you can use diodes with Rotary switches to reduce the number of bits required
- d. Potentiometers are programmed directly into EPICenter, therefore they appear in a separate part of the table
- e. Lights and LEDs are kept separate since they connect to a different ribbon cable
- f. Rotary Encoders are kept separate since they connect to a different ribbon cable

Connector Cable #:

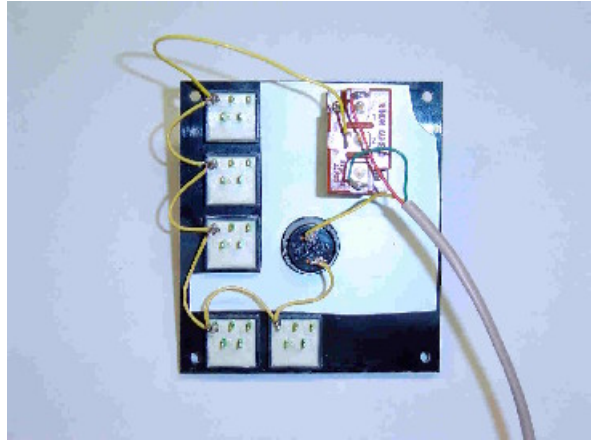
Button Name	64BTN Position	Module	Scan	Bit	Panel Row	Row Position	Panel Cable #	Wire Color	Connector Position
Main_Pwr_On	8	2	0	7	1	1	1	G	1
Main_Pwr_Off	9	2	1	0	1	1	1	R	2
FLCS_Pkg	10	2	1	1	1	2	1	Y	3
Caution_Reset	11	2	1	2	2	1	2	G	4
STBY_Gen	12	2	1	3	2	2	2	R	5
EPU_Gen	13	2	1	4	3	1	2	Y	6
Batt_Acft	14	2	1	5	4	1	3	G	7
Batt_FLCS	15	2	1	6	4	2	3	R	8
Common							3	Y	10

Module: Main_Pwr

Connector Cable #:

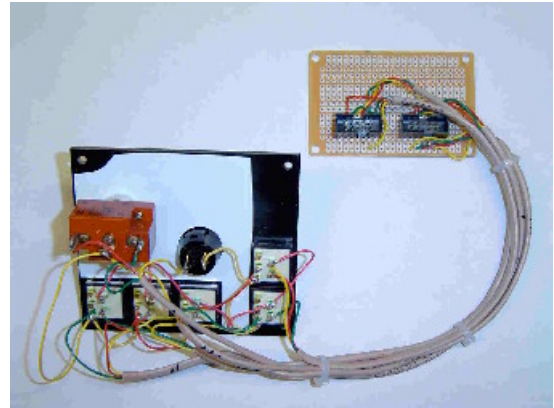
Light Name	PRO32 Position	Module	Scan	Bit	Panel Row	Row Position	Panel Cable #	Wire Color	Connector Position
FLCS_Pwr_Led		2	2	2			4	G	1
STBY_Gen_Led		2	2	3			4	R	2
EPU_Gen_Led		2	2	4			4	Y	3
Batt_Acft_Led		2	2	5			5	G	4
Batt_FLCS_Led		2	2	6			5	R	5
Common							5	Y	10

2.3.3.3 Once the cable track is done, it's time to work on the hardware. I start with a Radio Shack printed circuit board (PCB 276-150) that has holes sized for my ribbon connectors (mouser.com). These PCBs are **almost** perfect; however, when I plug in my ribbon connectors, I have to solder a small wire to the pins on one row of connector pins (see photo below). Once I install the connector and install my jumper wires, then I'm ready for the wires from the panel.



2.3.3.4 Now I wire my panel according to the cable track. I wire the common terminals together on my switches first; then I connect all the switch-able terminals to the cables. Next I wire the positive terminals of my LEDs together and then wire the negative side to my cables. Since I use a 3-conductor cable with a green, red, and yellow wire, I can keep things straight by always using the wires in the same order. I also mark the cables numerically so I am able to find the exact wire for any switch if I have a problem. I now attached the other end of the cable wires to the PCB. I use cable ties to bind all of the cables together along the whole length of the cable, usually 1ft max (see photo below).

2.3.3.5 Next I test all of the connections at the connector, moving the switches through all positions. By thoroughly testing my panel now, I can rest assured later-on that my switches are wired correctly and that problems lie in other areas. Once tested, I build a label for the panel showing, where it's from (the panel name), where it's going (64BTN or EMDA number), and what its order is.



2.3.4 Using label makers

I have begun to use a label maker to make my wiring systems that much easier to create and maintain. My label name contains the Panel, Module, and a counter for the total number of panels connecting to a given Module. Using this information, I can create unique names for my cables that are very descriptive: LandGear-64BTN-01-03 (Landing Gear panel to 64BTN number 1, 3rd panel connection). I place my labels close to the connector on the Module, panel and on both sides of the cable connecting the Panel to the Module.

2.3.4.1 The cable name to connect the switches on my Main Power panel is “MainPwr-64BTN03-02” and I place this on my panel’s PCB connector, on both ends of the connecting cable and on another PCB connector that’s wired to the screw terminals of my 64BTN. This cable wiring system really takes the guess work out of troubleshooting my simulator and will keep my job easy when I’m sorting out problems in my simulator with 256 switches, 15 pots, 8 rotary encoders and 64 lights.

Part 3: Hardware and Software Installation

3.1 Keyboard Support

EPIC USB, with the latest versions of EPICenter software, no longer requires you to connect the keyboard to the EPIC to send keystrokes. The newer versions of the EPIC USB driver installs additional mouse and keyboard devices in your windows configuration.

3.2 Multiple EPIC USBs

EPIC USB, with the latest drivers, allows you to use multiple EPIC USBs in a simulator. The EPIC USBs will communicate with each other and pass data via reserved memory called Pigeonholes (discussed later in the Software section).

3.3 EPIC Hardware Installation

3.3.1 Installing the EPIC USB drivers

3.3.1.1 Clean EPIC USB Install:

1. Ensure the EPIC hardware is disconnected
2. Install the EPICenter software
3. Verify the install of: windows\system32\drivers\EPICUSB.SYS
4. Verify the install of: windows\system32\epicio.dll
5. Connect your EPIC USB and ensure it finds the EPICUSB.SYS driver
6. Update the EEPROM to new version.
7. Close EPICenter.
8. Check a switch operation using EPICInfo
9. Check that keystrokes are working with at simple test project. If the keystrokes do not work check the FAQ on the R&R website.

3.3.1.2 Installing EPIC USB Updates

1. **Update EPIC USB EEPROM to new version first.**
2. Unzip the update. The main files for updating are EPICenter.exe, EPL.DLL, Vercheck.exe, \windows\system32\epicio.dll, and windows\system32\drivers\EPICUSB.SYS
3. Go to EPICenter \ tools \ Clean Registry. Check boxes "remove collection enumerations" and "remove WDM/driver info. Then click "OK"
4. Close EPICenter.
5. Unplug and replug USB cable from EPICUSB, this will find new hardware and reinstall the drivers. Direct the Windows driver install to the epic\install directory if necessary.

6. Check that keystrokes are working with at simple test project. If the keystrokes do not work check the FAQ on the R&R website.

3.3.1.1 EPIC USB LEDs

The EPIC USB has 6 LEDs to provide status of your EPIC system. These LEDs are from left to right looking at the LEDs: USB ISR, TP2 (test point), Sanity, TP1, Configured and Loaded.

"**USB ISR**" will light whenever the PC is transmitting or receiving data from EPIC USB. This should never be on solid but could be blinking.

"**TP2**" will come on for a second when EPIC USB is unplugging itself to tell the system a file has been loaded and to reconfigure. This will be steady on when a trap is hit. This can be used by the user to test to see if an area in their EPL was hit. This is used by setting bit 4 of the Qlog flag bits. "setqlogon(0x11)" will allow user control of TP2 and will light when an exec(138) command is hit and go out when exec(139) is hit. (bit 0 in the qlog flags will trace epl execution in the backtrace buffer).

"**Sanity**" means the card is running and sane. This should be blinking at about a 1/2 second rate after the correct speed settings are sent.

"**TP1**" can be used for different things in different versions, but in earlier versions it meant the card had data in the output buffer for the PC. Ignore this unless asked to check it during troubleshooting.

"**Configured**" means the PC found the card and exchanged configuration data with EPICUSB and has the drivers loaded (this will also light if only the HID drivers were loaded and the WDM (EPICUSB.SYS) is missing.

"**Loaded**" means the logic loaded ok. This happens at power up time.

3.3.2 Software for R&R EPIC Modules

3.3.2.1 Neither the EPIC Expansion nor the Output Modules require any special software to be loaded for them to operate properly with the EPIC USB. Once they are correctly connected to the EPIC, they are available for you to connect other hardware and will support your hardware based on your EPL code. The EPIC Output Module has some additional considerations like address jumpers; the ability to daisy-chain them on the EPIC Bus; and separate power supply for the module and its outputs. The EPIC Output module jumpers tell the Output module which position it is in and thus it's address. Take a look at the Output Module and set your jumper(s) according to you EPL programming. Also, on power-up of your EPIC with an attached Output Module, all of your EPIC controlled lights should turn "on". For this

reason, I place a procedure in my EPL “INIT” procedure to turn all of my lights “off”. By turning my lights “off”, I know my EPL was downloaded correctly and that my code is now running in my EPIC hardware.

3.3.2.2 A discussion of the software for the Rotary Module will be Included in a future update.

3.3.2.3 A discussion of the software for the Gauge Module will be Included in a future update.

3.3.3 Software for Micro Cockpit Modules

The Micro Cockpit ABA and 64BTN Modules do not require any additional software to operate properly with the EPIC USB. Once they are connected, you can address the switches connected to the 64BTNs without regard to the ABA hardware, you simply need to know the Mod/Row/Bit (Module, Row and Bit) of the 64BTN the switches are connected to. The EPIC USB can power the ABA; however, the ABA will require an independent power supply if you believe over 50 switches may be activated at one time. In any event, the 64BTNs will be powered from the ABA Module and no diodes are required for switches connected to the 64BTNs. Since the ABA is numbered for 64BTN Modules 1 – 8, and the bit connections on the 64BTNs are labeled for Rows 0 – 7 with Bits 0 – 7, it’s easy to determine the address of each switch you connect to a 64BTN (example: 64BTN Module 3 with a switch on Row 5, bit 2:

```
device(your_device_name) //-----any name you choose-----  
{  
connector(Mod3);//The module the switch is connected to  
  
button(5,2,your_button_name);//Button definition statement: Name is any  
name you choose  
//From this point forward you could refer to  
“your_device_name.your_button_name” to react to a button action like “on”  
or “off”.  
}
```

This same EPL code is used regardless of whether you hardwire your switches to the Expansion module, use the ABA/64BTN or use the EMDA Modules discussed next.

3.3.4 Software for CSI Modules

The CSI EMDA Modules do not require any additional software to operate properly with the EPIC USB. Once they are connected, you can address the switches and lamps connected to the EMDA Modules without regard to the actual hardware, you simply need to know the Mod/Row/Bit (Module, Row

and Bit) the switches or lights are connected to. The EMDA Modules require no separate power supply, however, all switches connect via the EMDA Modules will require diodes. The EMDA Modules make it easy to install inline resistors for your LEDs and diodes for your switches.

Part 4: Software Architecture

4.1 Aftermarket Software

4.1.1 [EPICmapper](#)

A discussion of the EPICmapper software will be Included in a future update.

4.1.2 [Flight Simulator Universal Inter-Process Communication](#) (FSUIPC)

FSUIPC is effectively a successor to FS6IPC.dll. Both modules are designed to allow external (i.e. *separate*) programs to communicate with and perhaps control Microsoft Flight Simulator. Keep in mind that Flight Simulator Panels (including their Gauges), Aircraft, Scenery and other graphics, and pretty much everything else *within* Flight Simulator, are *mostly* NOT correctable or influenced in any way by FSUIPC. Apart from some assistance in providing weather data to adventures, making adjustments in the weather itself, and enabling better access to some engine variables for some Gauges, FSUIPC can only help external applications talk to FS, nothing more.

4.1.3 [Falcon4 Shared Memory Interfaces](#)

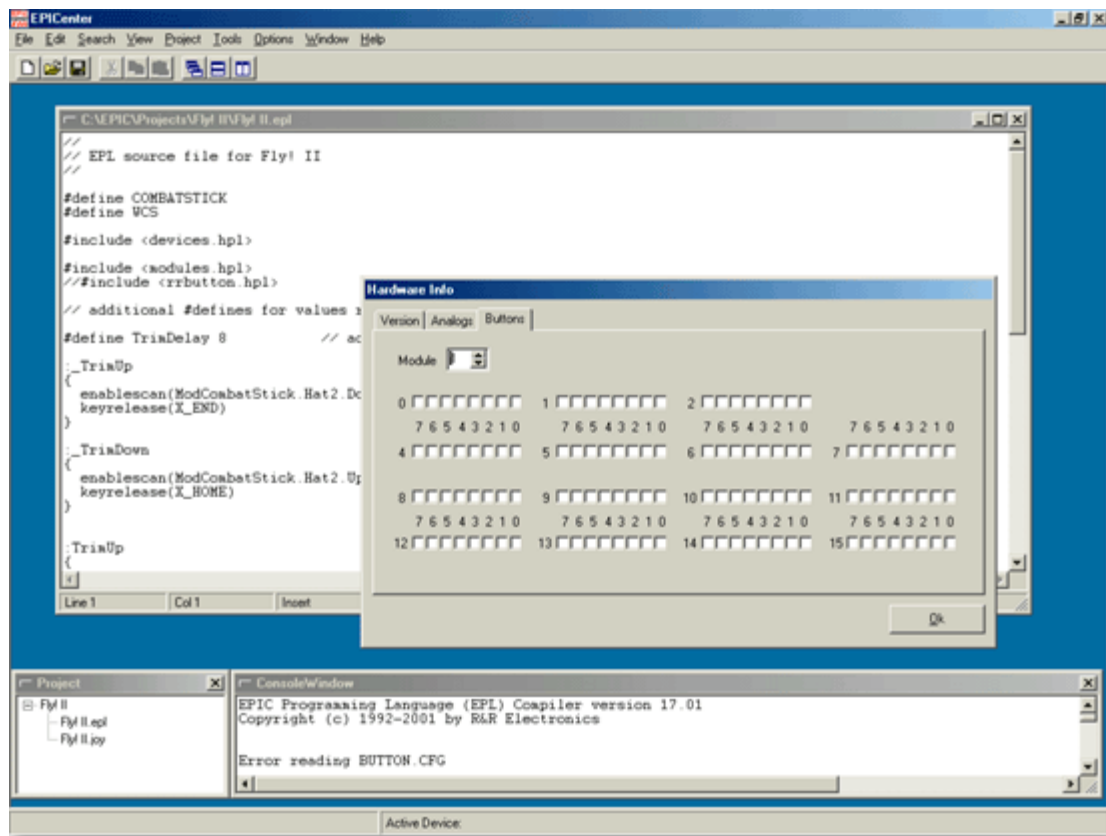
There are a number of hobbyists who have written interfaces to Falcon4 and its updated versions SP2 and SP3. Falcon4 shares some of its internal data and even allows inputs to orientate the view when using the virtual cockpit in the 3D world. This data sharing is implemented via a shared memory interface provided via the Windows environment. The Falcon4 shared memory interface outputs aircraft data, data about the light indicators, and other data that will make your simulator more realistic. This data can be accessed via standard C++ function calls. To obtain one of these Falcon4 shared memory interfaces or memreaders, check the EPIC forums or other Falcon4 simulator websites like the one in the link on the title to this section.

4.2 Homegrown EPL Software

4.2.1 EPICenter

[EPICenter](#) is a Windows based Integrated Development Environment (IDE) for configuring, editing, debugging, loading and managing your EPIC USB simulator files. If you've ever used Borland C++ it should look very familiar. EPICenter is included on the EPIC USB Installation CD-ROM provided with your EPIC USB interface card.

4.2.2 Updating EPIC software



The main window is a multi-document interface. You can edit multiple files at the same time, compile your project, edit your analogs, load your files into the EPIC, and debug them all from the same main screen. The Project Window has three entities (your project, your main EPL source file, and your JOY file). Double clicking on the EPL entity will bring up that file in a text editor; double-clicking on the JOY entity will bring up the Analog Settings Window. The Debug window (View | Debug), will tell you more than you ever wanted to know about the innards of EPICenter and the compiler. The Console window has replaced the old DOS screen. This window reports any errors while compiling or loading your EPL file.

4.2.3 Installing new EPICenter software

To install new EPICenter software simply unzip the distribution zip file with folder names enabled (checked). This should place all of the new EPICenter files in the correct directory, overwriting your old EPICenter files. If the new EPICenter distribution is accompanied by a new epicio.dll or ,sys file, ensure these files are unzipped into the proper directory also.

4.2.4 Downloading EEPROM software to EPIC

Once you have the EPIC USB connected to your computer and you have EPICenter running, you need to download new code to your EPIC EEPROM. The makers of EPIC are constantly upgrading the capabilities of EPIC and downloading new EEPROM software will allow you to use these new capabilities. One caution: I back up my computer before any major upgrade to ensure I can recover from any inadvertent problems. By performing a backup, if I find my simulator configuration is not currently compatible with the new EPIC update, I can always revert back to my previous configuration until the issue is sorted out. To perform the EEPROM update, go to EPICenter| tools| hardware| load EEPROM. Select the EEPROM file you wish to load, remember, the checksum is the last 4 digits of the name.

To load the EPIC EEPROM use the following steps:

1. Unplug the EPIC USB cable
2. Remove power from EPICUSB (to remove any running program)
3. Restore power to EPICUSB
4. Plug in the EPIC USB cable
5. Go to EPICenter- Tools | Hardware | LoadEEPROM
6. Select the EEPROM file. This will normally be EPICUSB.BIN or USBxxxx.Bin where xxxx is the checksum.
7. Enter the checksum in the dialog and click ok. The “sanity” led will blink faster and the “USB ISR” led will blink with the blocks being loaded. If all goes well there will be a final loaded ok message.
8. Unplug the EPIC USB cable
9. Remove power from EPICUSB
10. Restore power to EPICUSB
11. Plug in the EPIC USB cable
12. Run EPICenter – Tools| EPIC_info
13. If Checksum is ok, you are done, if not, retry from step 1 again

Although steps 1-3 above should not be necessary, they represent a safer way to update your EEPROM. If it looks like the loading gets hung up and stops responding with the “sanity” light flashing fast and “USB ISR”, close the EPICenter by pressing CTRL-ALT-DELETE, task manager, Applications, EPICenter, End Task, or wait until EPICenter comes back with the “Not Responding” message.

4.2.5 Compiling EPL to EPIC

After writing your EPL code, compile it using the menu item on the top of the EPICenter window, or use F9. EPICenter will review every (non-comment) line in your EPL Project and identify any errors. Keep in mind, your error may not be on the line identified, but may be in a preceding line or the last line in the file compiled prior to the line that is being reported by EPICenter. If an error is identified, correct the error and then recompile (F9). When you get a clean compile, you'll get a pop-

up window with an “OK” to acknowledge, then you’re ready to download the code to your EPIC USB.

4.2.6 Downloading EPL to EPIC

Once you get a clean compile, download the EPL code to your EPIC by using the menu item on the top of the EPICenter window or by pressing F10. EPICenter will now download your EPL to EPIC while giving you a progress window so you can track the progress. You’ll also hear EPICenter disconnect and reconnect the EPIC from/to Windows as part of this process. Once the download is complete, a special procedure, called “INIT” in your <name>.epi file is executed. The “INIT” procedure is where you can place code (i.e. setup routines) that you want executed whenever your EPIC is first loaded. Now that code has been successfully loaded into your EPIC, you’re ready to use it, so close the EPICenter since you won’t need it again until you need to modify your EPL code.

4.3 EPL Software

4.3.1 EPL Program Syntax

4.3.1.1 All text between angle brackets “< >” is **descriptive** and should not be taken literally. All options separated by vertical lines “|” are **mutually exclusive**. All options within square brackets “[]” are **optional**, options not within square brackets are required. All text following the double slashes “//” **<comment>** and preceding the end-of-line is ignored. Comments may appear anywhere in your program.

4.3.1.2 Identifiers (ie. string literals <label> <flag> <var>), are used to “name” flags, variables, and procedure blocks. Identifiers are case sensitive (ie. “Flag1” is not the same as “FLAG1”) and must be unique. Reserved words cannot be used as identifiers. Also, identifiers can only start with an Alpha (A-Z) or underscore (_) characters. If you reuse a label, flag or variable by defining it twice, you will get the “Ambiguous use of <item name>” error since all labels, flags and variables must be unique.

4.3.1.3 : <label> { <function(s)> } This is the syntax for a label and it’s accompanying function statements. You can now use the syntax <return type> <label>(<arg type> <arg>,....) in preparation for passing arguments and return values in later versions of EPICenter. Currently however, the only supported form is: void <label>(void){} which has no arguments (leading void) and no return values (trailing void). There is also a predefined label called “**RETURN**” that can be used in place of any label reference. A “jump” or “call” to “RETURN” does nothing.

4.3.1.4 EPICenter allows the use of predefined shortcuts in some of the names of the functions. These shortcuts are defined as follows: **keypress/kpr(<key>);** This

function definition means that this function can be **KEYPRESS(<key>)** or **KPR(<key>)** where **KPR(<key>)** is the shortcut version of this function.

4.3.1.5 The preprocessor directive, “**#define**”, allows you to use a string to represent a value (**#define <string> <value>**). In the example if Trigger is equal to module 0, row 0, button 0 and if the Auto pilot switch is module1, row 5, bit 6, we could use the following statements in our EPL file:

```
#define TRIGGER M0R0B0
#define APSWITCH M1R5B6
```

Once defined, we only need to refer to TRIGGER and APSWITCH for actions involving these devices (mod, row, bits).

4.3.1.6 Numbers can be expressed in decimal, hexadecimal, or binary. If a number is to be a **hexadecimal** number, the numerical value must be preceded by “**0x**”. If a number is to be in **binary**, the numerical value must be preceded by “**0b**”. Therefore 123 can be represented as follows:

```
decimal: 128 range 0-255(byte),0-65535(word)
hexidecimal: 0x80 range 0-0xFF(byte), 0-0xFFFF (word)
binary: 0b10000000 range 0-0b11111111(byte), 0-0b1111111111111111
```

4.3.2 EPL Program Structure

To write an effective EPL Project, you must define quite a number of items for EPIC about your hardware and how you want EPIC to respond to hardware actions. I use the following structure to organize my EPL code to ensure I tell EPIC everything it needs to know:

```
<name>.epi //my main EPIC EPL file
EPICHID.INC //contains global definitions
mydevices.hpl //explains assignments of analogs and modrows
devices.hpl //explains my hardware devices (physical)
procedures.hpl //explains how procedures tie to devices
```

4.3.2.1 Main EPIC Project File: <name>.epi

When you create an EPIC Project using EPICenter, it will create the project with a default name or the <name> you choose. We’ll use <name> as the project name in this document. One of the files created when you start a project will be <name>.epi, the main EPIC EPL file for your Project. In <name>.epi, you’ll need to tell the EPICenter compiler about all the other EPL files you want to include in your EPIC project. We use the “**#include**” command to include other EPIC code in our projects:

```
#include <EPICHID.INC> //the < > mean “/EPIC/Include” directory
```

```
#include "mydevices.hpl" //the " " mean "/EPIC/Projects/<name>" directory
#include "devices.hpl"
#include "procedures.hpl"
```

An EPIC EPL program needs to include some or all of the following items to operate properly. The placement of these items (into separate files) is somewhat arbitrary; however, you should maintain this order in your EPL Project or you may receive errors when you compile the Project. The key to a good EPIC Project is including all of these definitions and code blocks in a set of files (like the ones I discussed above) that you can manage as your Project grows to thousands of lines of code.

- module definitions
- flag declarations
- variable declarations
- button definitions
- procedure blocks

I also include in my <name>.epi file the definitions of my modules. Using the “**definemodule**” command, we tell the compiler that we have a module attached to EPIC. The order that the define module statements are encountered will determine the relative numbering of the modules (the first module defined is module 0; the second is module 1; etc..) The syntax for the define module command is:

```
Definemodule|defmod(module<name>,moduletype,StartRow,NumberRows)
```

The module types are 0=high priority scan; 1=low priority scan; 2=output module; and 3=seven segment display. To ease the burden on our memory, we can “#define” the module types and simply use the names from that point on:

```
#define FASTSCAN 0
#define SLOWSCAN 1
#define OUTPUT 2
#define 7SEGDISP 3
```

To make life even easier, the EPICHID.INC file already defines these along with other information so if you “#include” it in your Project, you don’t have to define them again. Take a look inside the EPICHID.INC file to see what other goodies it has for you, it’s in your “/EPIC/include” directory. With the module types defined we can now write:

```
definemodule (FirstModule,FASTSCAN,0,5)
```

This statement defines a module named “FirstModule” that is a “high priority scan” type module with rows starting at 0 and going through row 5. Keep in mind that you can only have 16 rows as Fastscan rows, so use Fastscan wisely.

There is another unique part of the <name>.EPL file we need to discuss since it also has a special purpose. But first, I need to introduce procedure blocks. Procedure blocks are the “meat” of an EPL program, since this is where the work actually gets done. Procedure blocks begin with a label that is followed by the curly bracket “{”. Other procedure calls, functions, etc, fall within the initial curly bracket and a closing curly bracket “}”. Statements between the curly brackets are executed in the order they are encountered, and each statement must end with a semi-colon (“;”). A “return” is assumed at the end of the procedure block and therefore not required (the closing curly bracket “}” serves this function). If a procedure block calls another procedure block, execution continues with the called procedure block until a closing curly brace is encountered (then execution returns to the previous procedure block). It is also possible for a procedure block to recurse by calling itself. Comments and blank lines can be interspersed as desired throughout the procedure block and will be ignored by the compiler. The following is an example of a procedure block:

```
:This_is_a_Label
                //This is a comment line
{
                //This is the first line of the procedure block
keyhit(g);      //Function statement
                //Blank line
}
                //This is the last line of the procedure block
```

The “INIT” procedure is a special procedure that is executed whenever a project is downloaded to EPIC USB. The “:INIT” label must be spelled with capitol letters and appear in your EPL code. I place the :INIT procedure in my <name>.epl file since it is the first procedure block executed. Other procedure blocks are tied to button definitions, pigeon hole definitions and qproc definitions, all of which will be discussed later.

4.3.2.2 Assignment of analogs and modrows: mydevices.hpl

With EPICenter and EPICUSB, the concept of “connectors” and “devices” was introduced. This section is aimed at explaining how to use connectors and devices in your EPL project. A “device” is a collection of switches, buttons, displays, rotaries, analogs, etc. My Threat Warning Prime panel is an example of a device. It has both buttons and lights defined for Threat Handoff; Missile Launch Warning; Primary Mode; Unknown Threat; Naval Threat; and Target Separation functions.

```
device(Threat_Warn_Prime)//---Threat Warning Prime---
{
    connector(Mod1);
    button(4,6,Handoff);
    button(4,5,Msl_Launch);
    button(4,4,Pri_Mode_Open);
    button(4,3,U_Unk);
    button(4,2,Naval);
    button(4,1,Tgt_Sep);
}
```

```

output(Leds1,0,1)
{
    Handoff_Led =      0b00000001;
    Msl_Launch_Led =  0b00000010;
    Pri_Mode_Open_Led = 0b00000100;
    U_Unk_Led =        0b00001000;
    Naval_Led =        0b00010000;
    Tgt_Sep_Led =      0b00100000;
};
};
};

```

4.3.2.3 A device is defined using 1 connector that the device would “plug” into. MOD1 is the connector for the Threat Warning Prime device above. The connector does not have to be a physical plug connection, and in reality may be just the wires going to various analogs, and modrows. Devices can be used to pass events to Direct Input (which show up in Gaming Options); to pass events to programs communicating with EPICIO.DLL; or internally to execute procedures when a button changes state (on to off or off to on). If a button is not attached to an internal procedure, it will generate an automatic event. An automatic event can go to Direct Input, the DLL, or both based on keywords “SEND_BUTTON_DLL”, “SEND_ANALOG_DLL”, “SEND_POV_DLL” (EPROM V5.2 and above) in the device definition.

#pragma hid_snd_rpt xx where xx is the number of devices to send to HID, controls which devices will send automatic events to Direct Input. (EPROM V5.1 and above).

The EPIC developer recommends ordering devices as follows:

- Devices to send to Direct Input and alternately to EPICIO.DLL
- Device to send to EPICIO.DLL
- Devices with physical switches to be used to execute EPIC procedures.
- Displays, outputs, etc.
- Mousedevice

4.3.2.4 Connectors are a collection of analogs and modules and rows to be scanned (checked by EPIC for changes). In EPIC USB, there are physical and EPIC USB internal connections that we must keep straight in order for our simulators to work as we plan. Physical connections are the module, row, bits where our switches are physically connected:

```

button(1,6,Handoff)//on module 1
button(2,3,Handon)//on module 2

```

4.3.2.4.1 In EPIC USB however, we then connect this physical mod, row, bit to an internal connector called "connector" which is tied to the internal EPIC USB mod, row, bit. Below is the internal EPIC USB connector , "Mod1", definition.

```

connector(Mod1) //Internal EPIC connector

```

```

{
  analog(0);
  modrow(0,1);// EPIC module 0, Row 1, Bits 0 - 7
}

```

4.3.2.4.2 We must tie our physical switch definitions to the EPIC USB internal connectors before our hardware can communicate with EPIC. We tie our physical switches to a connector by assigning our hardware to a device and then assigning the device to a connector. These will be relative assignments, which mean the assignment order will determine the module, row, and bit values for each button, etc.. Below is the device "first" which is assigned to connector Mod1 and has button "Handoff" assigned to it.

```

device(first)//
{
  connector(Mod1);
  button(1,6,Handoff);//assigned to Mod1 modrow (0,1)
}

```

The result of this code is that the button "Handoff" is now assigned to EPIC USB's internal Module 0, Row 1, Bit 0 since it is the first button assigned. The next assignment would be Module 0, Row 1, Bit 2, and so on. In reality, you don't need to worry about this since EPIC will keep track of it for you, just remember you have to use connectors and devices correctly and you can only assign 8 devices to a modrow.

4.3.2.4.3 If you want, you can make life easy by assigning your physical devices to the EPIC USB internal connectors in the same order you have them connected physically. This simply means that the switch assigned to module 0, row 0, bit 0, is assigned to a device/connector first and then the switch assigned to module 0, row 0, bit 1 is assigned. If you do this for your entire project, your physical and internal EPIC USB connections will both be the same. On the other hand, if you assign your physical devices to internal EPIC USB connectors in a random order, your final EPIC module, row, bits values may not match your actual physical module, row, bits values since they will all be relative.

4.3.2.4.4 For the advanced EPL users, here's another thought on connectors. Different connectors can have the same analogs and module/row definitions as other connectors; however, the end definition (the device level) of the actual switch (Mod, row, bit) must be unique. This makes it easy to just use the same modrow and analog definition for all of your connectors and not try and keep track of which modrows are used with which connectors. Again, you just need to make sure you only assign 8 devices per modrow across your connectors.

```

connector(Mod1)
{

```

```

    analog(0);
    modrow(0,1);
    modrow(0,2);
    modrow(0,3);
}
connector(Mod2)
{
    analog(0);
    modrow(0,1);
    modrow(0,2);
    modrow(0,3);
}
device(first)//
{
    connector(Mod1);
    button(1,6,Handoff);//assigned to Mod1 modrowbit (0,1,0)
}
device(second)//
{
    connector(Mod2);
    button(2,3,Handon); //assigned to Mod2 modrowbit (0,1,1)
}

```

In this example two connectors use the same definitions for the analogs and modrows. Also shown are two device definitions using the connectors I defined. Notice that the button definitions use the same modrow (0,1) because they are the first ones defined and therefore are assigned to the first modrow in the connector definition (0,1). The button definitions however use different connectors (Mod1 and Mod2) and have different button row bit values. In the end, “Handoff” is assigned internally to module 0, row 1, bit 0, while “Handon” is assigned internally to module 0, row 1, bit 1.

4.3.2.5 Assignment of hardware devices: devices.hpl

Rather than placing all of my EPL code in one file, I break it up into multiple files. I use the “devices.hpl” file for all of my hardware device definitions. In this file I’ll keep all of the definitions for all types of hardware on a given panel together. Therefore, switches, lights, buzzers, motion control, etc. will all be defined together so I know exactly where to go if something fails or doesn’t work properly. I also separate these files by region of my cockpit, so I have a frontdevices.hpl, leftdevices.hpl, and rightdevices.hpl. Here’s the example I used earlier showing part of my Threat Warning Prime panel device definitions, this code appears in my frontdevices.hpl file.

```

device(Threat_Warn_Prime)
{
connector(Mod1); // Switches on the Threat Warning Prime
button(4,6,Handoff);
}

```

```

button(4,5,Msl_Launch);
//the rest of my button definitions
output(Leds1,0,1) // Lights on the Threat Warning Prime
{
    Handoff_Led = 0b00000001;
    Msl_Launch_Led = 0b00000010;
    Pri_Mode_Open_Led = 0b00000100;
    // the rest of my output definitions
};
//Other devices could be added with the appropriate code
};

```

4.3.2.5 Assignment of procedures to hardware devices: procedures.hpl

I use the procedures.hpl file as a companion to the devices.hpl file; therefore, I have a frontprocedures.hpl, leftprocedures.hpl and rightprocedures.hpl file in my EPIC project. The procedures file is where I have the actual functions for EPIC to execute based on a button action. Again, like my devices file, segmentation makes it easier to manage my project. Below is an excerpt from my frontprocedures.hpl file for the Threat Warning Prime panel.

```

//-----Procedures for Threat Warning Prime-----
void Threat_Warn_Prime.Handoff.On(void){shifhit(PGDN);}
void Threat_Warn_Prime.Msl_Launch.On(void){keyhit(INS);}
void Threat_Warn_Prime.Pri_Mode_Open.On(void){shifhit(PGUP);}

```

4.3.2.6 Special procedures: Rotaries

In some software simulators not written to support a product like EPIC, virtual switches may not behave like they would in the real world. In the Falcon4 virtual aircraft, some 3-position switches will follow this sequence: position 1 to 2, to 3, to 1, to 2 and so on (i.e. it rolls over). In the real world, this switch would go from: position 1 to 2, to 3, to 2, to 1 and so on (i.e. must reverse direction). To keep your software simulation synced with your switches sometimes requires special code. I place this code in my rotaries.hpl file since once I write a routine, I can duplicate it whenever I come across another switch with the same behavior. Below is the code for a switch on my HSI that controls selection of navigation modes. This code excerpt comes from my rotary.hpl file that is the only other file that contains procedures tied to buttons.

4.3.2.6.1 This code supports a 4-position rotary switch. since I use diodes on this rotary, I only need 3 bits since I can check for bits 1 and 2 being active to represent position 3. In Falcon4, this rotary switch rolls over from position 4 to position 1 and has no reverse direction command. I can make the Falcon4 virtual switch go backward by sending the forward command 3 times (ILSRoate3). To force the Falcon4 virtual switch to behave like my real-life switch, I have to track the last position of the real switch and then send the forward command three times for a reversal, unless I'm in position 4.

```

byte ILSPos;

```

```

byte Last_ILSPos = 1;
:ILSRotate
{keyhit(I);Last_ILSPos = ILSPos;}//Increment virtual switch once
:ILSRotate3
{keyhit(I);keyhit(I);keyhit(I);Last_ILSPos = ILSPos;}
//to make virtual switch go backward, increment it 3 times
:ILSPos1
{jump (ILSRotate3);}
:ILSPos2
{if (Last_ILSPos == 1) jump ILSRotate;if (Last_ILSPos ==3) jump ILSRotate3;}//don't forget to track
last position
:ILSPos3
{if (Last_ILSPos == 2) jump ILSRotate;if (Last_ILSPos ==4) jump ILSRotate3;}
:ILSPos4
{jump (ILSRotate);}
// Check to see which button was pushed
void Instru_Mode.TCN_ILS.On(void){jump (CheckILSRotary);}
void Instru_Mode.TCN_ILS.Off(void){}
void Instru_Mode.TCN.On(void){jump (CheckILSRotary);}
void Instru_Mode.TCN.Off(void){}
void Instru_Mode.NAV_ILS.On(void){jump (CheckILSRotary);}
void Instru_Mode.NAV_ILS.Off(void){}
//-----Start Check Rotary Procedure-----
:CheckILSRotary{
ILSPos = 0; //---Counter-----
// Set a variable based on which button was pushed
if (Instru_Mode.TCN_ILS) {ILSPos = 1;} //If TCN_ILS is active add 1 to ILSPos
if (Instru_Mode.TCN) {ILSPos += 2;}
if (Instru_Mode.NAV_ILS) {ILSPos = 4;}
if(ILSPos==Last_ILSPos) jump RETURN;// If nothing happened do nothing
// Jump to the correct Simulator Routine
if(ILSPos == 1) jump ILSPos1;
if(ILSPos == 2) jump ILSPos2;
if(ILSPos == 3) jump ILSPos3;
if(ILSPos == 4) jump ILSPos4;}
//-----End ILS Rotary Procedure-----

```

4.3.2.7 Special procedures: Pigeonholes

Most of you may not have to use pigeonholes, but unless you use commercially available software, you'll probably need some special software to send data from your software simulator to EPIC so you can control output devices like lights, buzzers, motion control and gauges. By using pigeonholes, I can pass data back and forth between my software simulator (Falcon4 SP3 for me) and my simulator hardware panels using EPIC USB. Below is a code excerpt of my pigeonholes code that controls two of my eyebrow lights. This code requires a companion routine to send data to the pigeonhole, I use C++ to write the companion code.

```

//-----Pigeon Hole 1-----
byte F4SharedMem_lightBits; //Temporary data store
byte F4SharedMem_lightBits2; //Temporary data store
byte F4SharedMem_lightBits3; //Temporary data store
byte F4SharedMem_HSIBits; //Temporary data store
byte F4SharedMem_GearBits; //Temporary data store
int checkBits

```

```

ph Falcon4SharedMem (0) //Define the Pigeon Hole as a 16 bit word;
    //will only really need 1
{
    byte F4share1; //First PH store
    byte F4share2; //Second PH store
    byte F4share3; //Second PH store
    byte F4share4; //Second PH store
    F4SharedMem_lightBits = Falcon4SharedMem.F4share1; //transfer data
    F4SharedMem_lightBits2 = Falcon4SharedMem.F4share2; //transfer data
    F4SharedMem_lightBits3 = Falcon4SharedMem.F4share3; //transfer data
    F4SharedMem_HSIbits = Falcon4SharedMem.F4share4; //transfer data
    call (dataXfer);
};
:dataXfer
//Eye Brow Master Caution
LeftEye_Lgts.Lamps2.Master_Cau_Lamp=off;
checkBits = F4SharedMem_lightBits & 0x1;
if (checkBits == 0x1){LeftEye_Lgts.Lamps2.Master_Cau_Lamp=on;}
//Eye Brow Engine Fire
RightEye_Lgts.Lamps1.EngineFire_Lamp=off;
checkBits = F4SharedMem_lightBits & 0x20;
if (checkBits == 0x20){RightEye_Lgts.Lamps1.EngineFire_Lamp=on;}
};

```

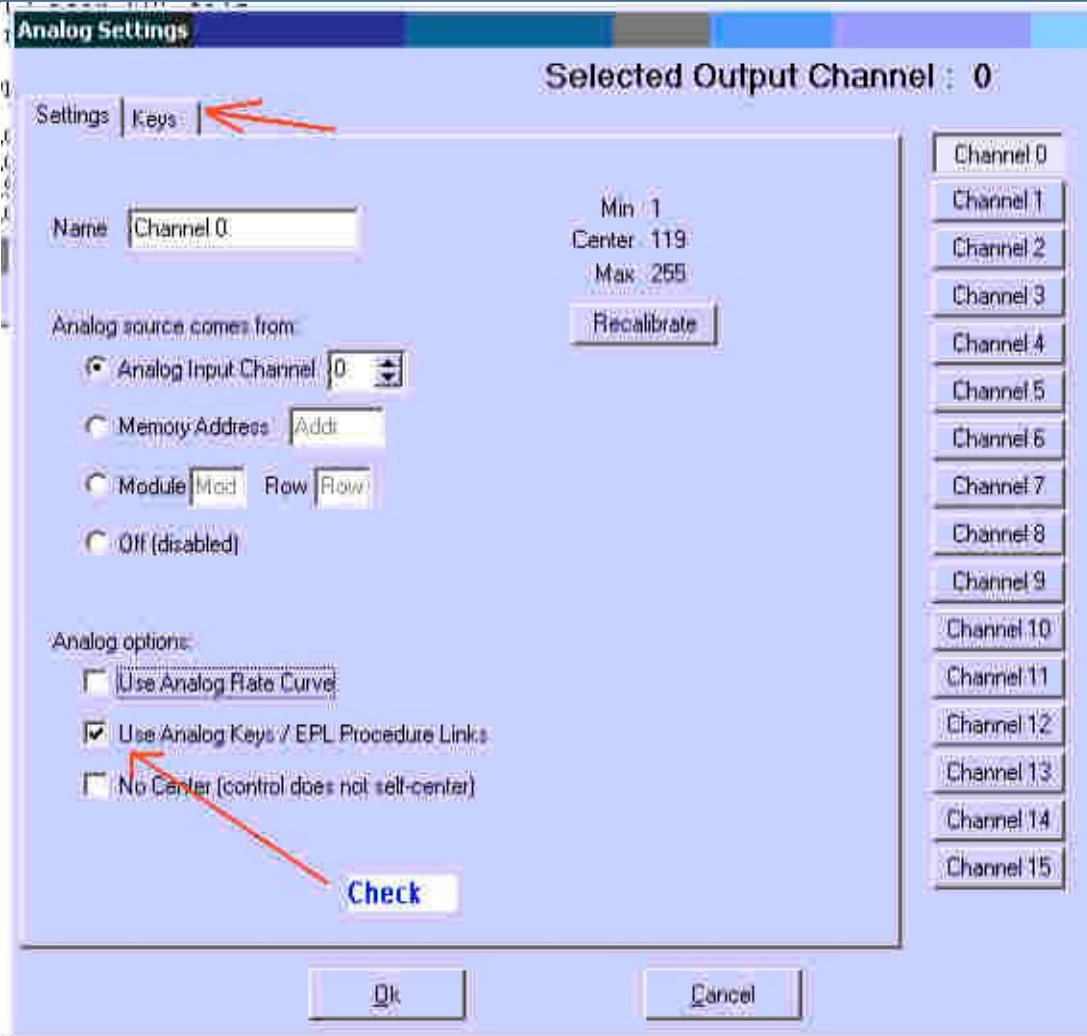
If the appropriate bits in the pigeonhole are on “1”, then EPIC will turn the appropriate light on.

4.3.2.8 Keystrokes and Procedure calls from Analogs

Analog potentiometers or pots, can generate keystrokes, link to EPL procedures and have their response altered by rate curves. Below are the steps to take to attach keystrokes and procedures to your pots.

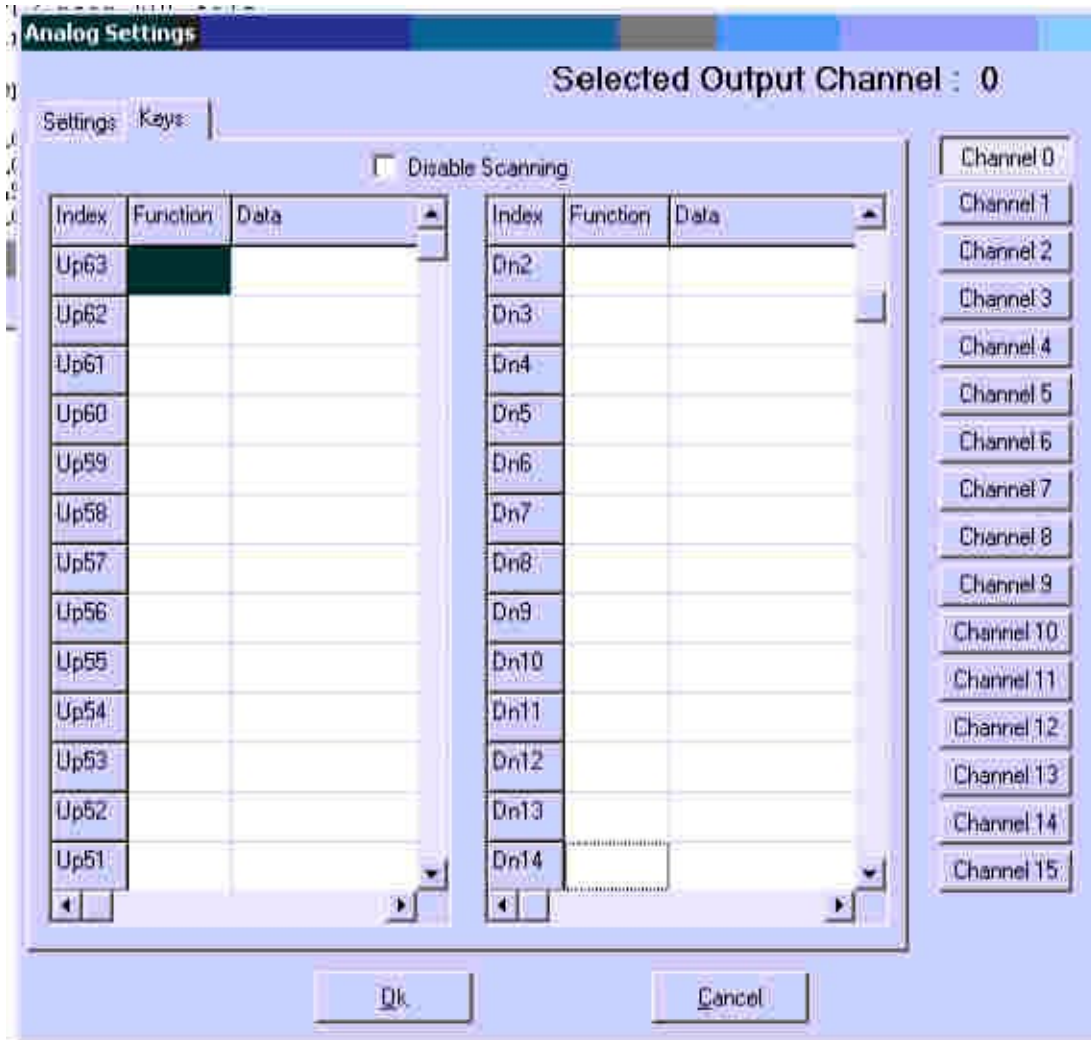
First, open the Project window and double click the <project>.joy file.

Double click <project>.JOY from project manager Project tab.
 Check “Use Analog Keys / EPL Procedure Links”, The “Keys” tab will appear.

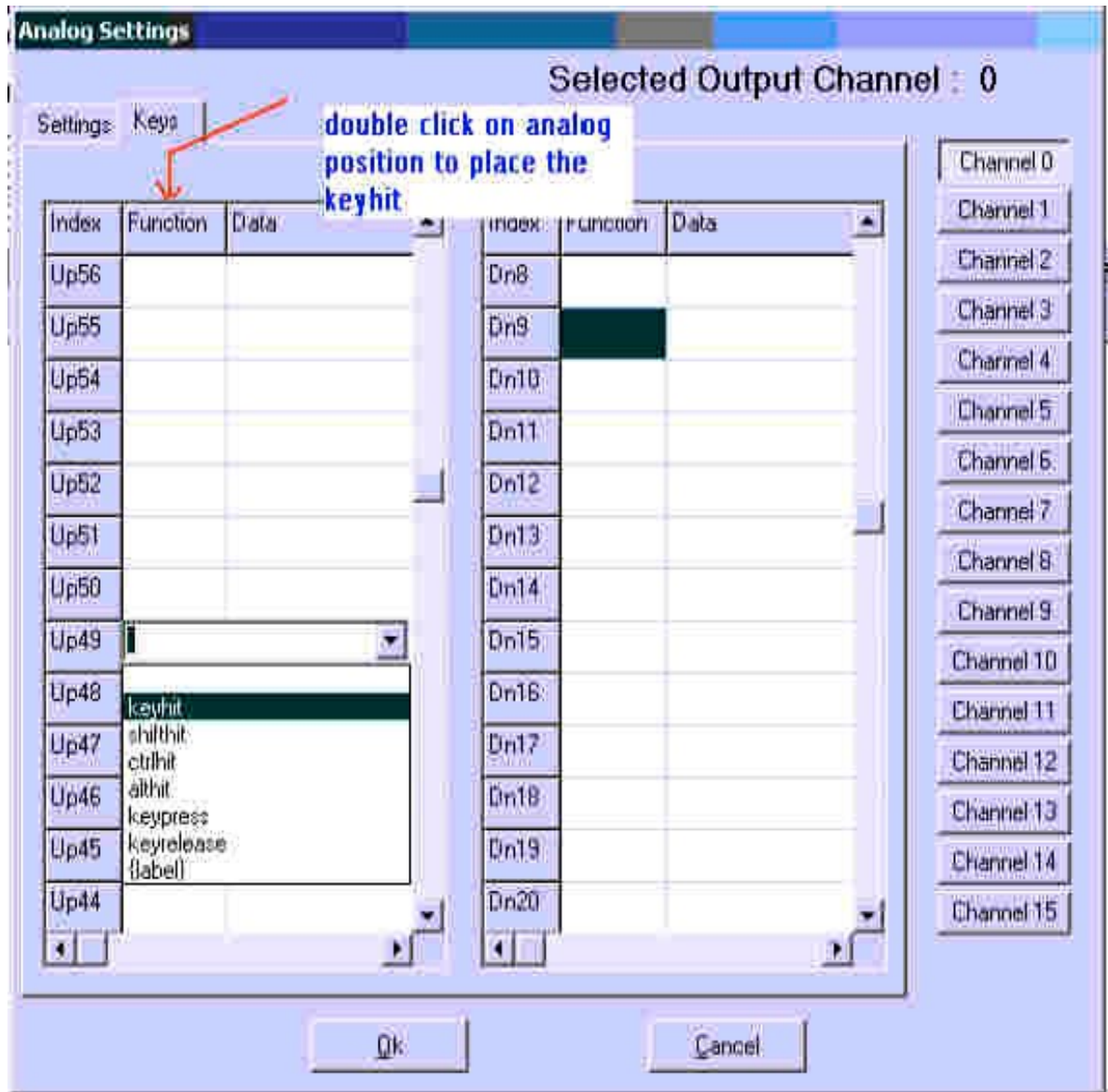


Click “Keys” tab.

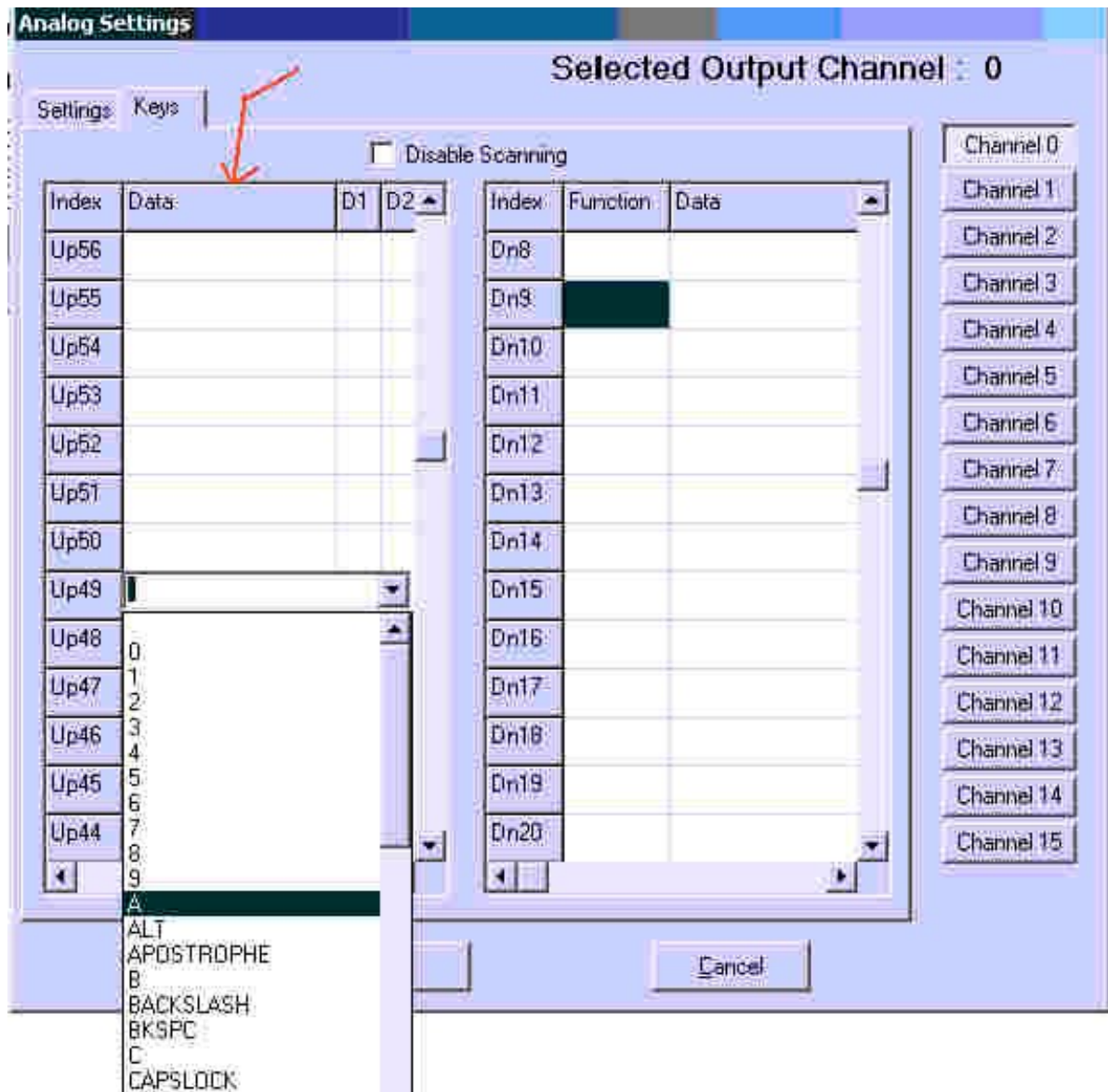
Note: Your selected output channel should be the “Input channel” in this window.



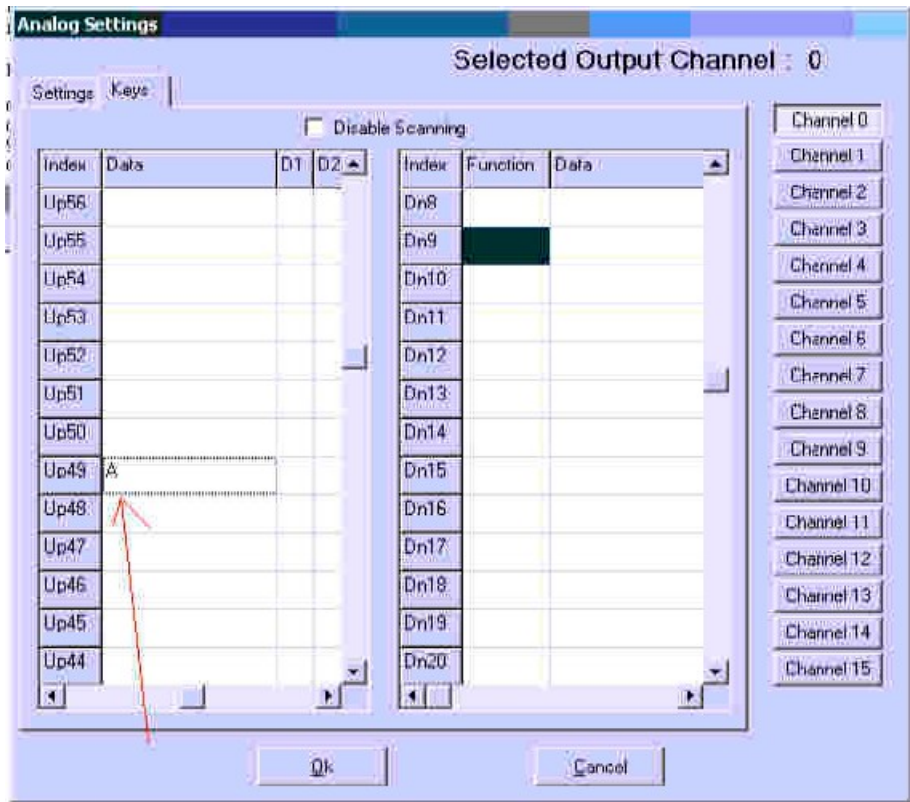
When the analog is going up the scroll bar and box will follow on the left side and when going down will follow on the right side. To keep the analog from interfering check “Disable Scanning”, if needed. Double click on the position to place the keystroke, or procedure link you want sent when the analog reaches this position. If the simulator functions you’re controlling with an analog has a present number of positions, you can evenly distribute these the commands across those positions.



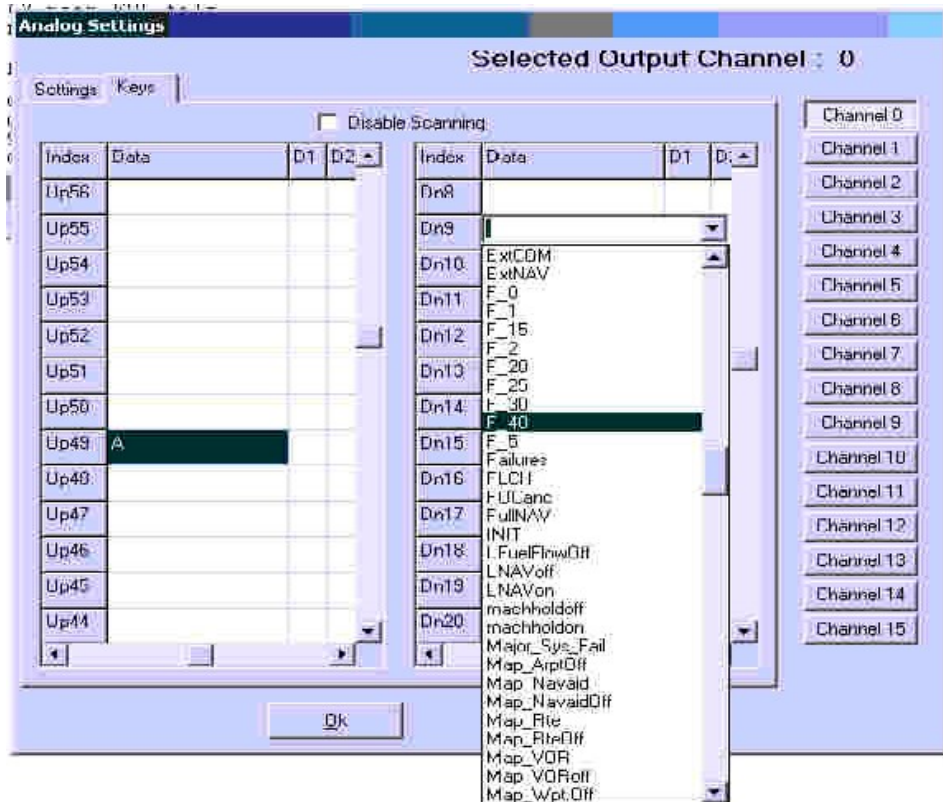
Select keyhit, then double click the "Data" box and select the key you wish to send.

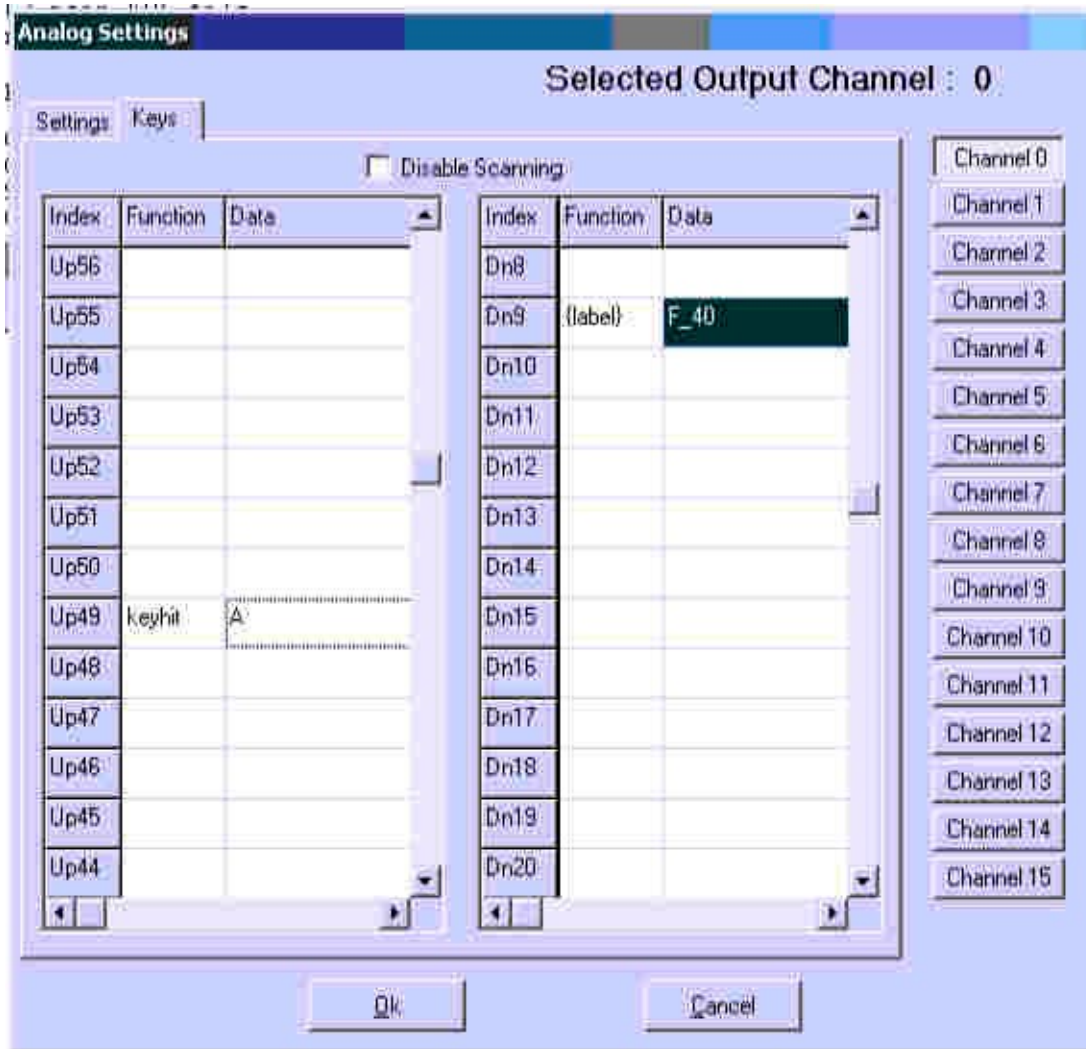


In this case we wish to send an "A" whenever this position is reached.



Now whenever the analog is going in the up direction and hits this position a “keyhit(A)” will be sent. To link to an EPL procedure, double click on the “Function” for the position and select {label} , double click the “Data” box and a drop down of your EPL procedures will appear, then select the procedure you wish to send when this position is reached.





Now whenever this analog is going in the down direction and hits this position, EPL procedure F_40 will be executed. Click “Ok” to save, Ok at rate curve save screen, F9 to compile, F10 to load and then you’re ready to use your analogs.

4.3.3 EPL Commands

The following are explanations of the commands available to support your EPL programming needs.

definebutton|defbtn(<button>, on|off, <label>)

This command associates a procedure block, <label>, with a specific button action on|off. The <button> is the button being assigned and label must be a defined label/procedure block or “RETURN”. This command can be in the formats:

MRB- Module,Row,bit ex: M1R3B2 module1 row3 bit2

MRM- Module,Row,Mask ex: M1R3M0xFF module1 row3 mask all bits

M1R3M0x0F module1 row3 include bits 0-3

After defining the button it can be used as follows:

void deviceName.buttonName.On(void) {.....} . The void(s) are optional at this time; therefore, I use: **Threat_Warn_Prime.Msl_Launch,On** (press) or **“.Off”** (release) <label>. On|Off are the only two actions allowed in this command, and <label> is the procedure block to execute when the action occurs.

keyhit|khit(<key>);

Tells EPIC to press and then release a <key>. All <key> must be a valid key identified in the [TOKENS.DOC](#) and they must be capitalized. Please note that although <key> is a capital letter such as “A”, a non capital letter will be sent i.e. “a”.

Example:

Keyhit(A);// sends → a

keypress|kpr(<key>);

Tells EPIC to press, but not release the <key>. All <key> must be a valid key identified in the [TOKENS.DOC](#) and they must be capitalized.

keyrelease|krl(<key>);

Tells EPIC to release <key> without pressing it. All <key> must be a valid key, identified in the [TOKENS.DOC](#) and they must be capitalized. The <key> should have been previously “pressed” using the “keypress” command.

rawkey(<value>);

Tells EPIC to send <value> to the keyboard port. This can be used to send extended codes to a program that handles them. Usually the value is 128 or greater.

shifthit|shit(<key>);

Tells EPIC to press and then release <key> while holding down the shift key. All <key> must be a valid key, identified in the [TOKENS.DOC](#) and they must be capitalized. To press and/or release one of the shift keys by itself, use “keypress” and “keyrelease”. Use shifthit <key> when you need to send a capitalized character, or other shifted key.

ctrlhit|chit(<key>);

Tells EPIC to press and then release <key> while holding down the Control key. All <key> must be a valid key, identified in the [TOKENS.DOC](#) and they must be capitalized. To press and/or release the Control key by itself, use “keypress” and “keyrelease”.

Also, for compound key sequences like “control shift A”, use the keypress (CTRL) and then shifthit (A) then keyrelease (CTRL).

althishit(<key>);

Tells EPIC to press and then release <key> while holding down the Alt key. All <key> must be a valid key, identified in the TOKENS.DOC and they must be capitalized. To press and/or release the Alt key by itself, use “keypress” and “keyrelease”.

setinterkey|interk(<delay>);

This is an advanced hardware control command. It sets the delay between the “make” (ie. press) and “break” (ie. release) codes to <delay> * 10ms. The default <delay> is 40 milliseconds.

setpostkey|postk(<delay>);

This is an advanced hardware control command. It sets the “post-break” delay (ie. the delay between the “break” and the “make” codes) to <delay> * 10ms. The default <delay> is 30 milliseconds.

addpost(<delay>,<delay1>);

This command inserts a delay of <delay> * 10 millisecond between the key release and CTRL, ALT, or SHIFT for the next ctrlhit, althishit, or shifthishit. This applies only to the next command. The range is 0-15 for <delay> and <delay1>. The <delay1> applies to AFTER the CTRL, ALT, or SHIFT release. Example:

```
{addpost(5,2);
ctrlhit(A);
ctrlhit(B);
}
```

This command added 50milliseconds between the “A” release and the CTRL release and 20 milliseconds after the CTRL released (before the CTRL press for the “b” key, but does not apply to the ctrlhit(B).

setctrldelay|ctrldel(<delay>)

This command inserts a delay of <delay> * 10 millisecond like above, but applies to all ctrlhit, althishit, and shifthishit commands. The range for <delay> is also 0-15.

Example:

```
:INIT{
setpostkey(2);
setinterkey(2);
setctrldelay(5);
}
```

The setctrldelay command will inject an extra 50 milliseconds between the keyrelease and the CTRL, ALT, or SHIFT hits.

scanrow(<module>,<row>)

This command returns the value at <module>, <row>. A “1” means the modrow is active (on for switches). Module and row must be defined as being scanned using the definemodule command, otherwise an undefined result will be returned. Result: example:

```
definemodule(3,FASTSCAN,0,8)           //scan module 3 rows 0-7
byte scan;
scan = scanrow(3,2); //will return the current state of switches at module 3, row 2
```

disablescan|discan(<MRB button>);

disablescan|discan(<MRM>);

These are hardware control commands that disable scanning of button <MRB button> or masked row <MRM> until enabled. Example:

```
disablescan(M0R2B4);
disablescan(M0R2M0b00010000); is equivalent to M0R2B4
disablescan(M0R2M0x10); is equivalent to M0R2B4
disablescan(M0R2M0b00001100); will disable bits 2&3 on M0R2
```

enablescan|enscan(<MRB button>);

enablescan|enscan(<MRM>);

These are hardware control commands that enable scanning of button <MRB button> or masked row <MRM> that have been disabled with the disablescan command.

```
enablescan(M0R2B4);
enablescan(M0R2M0b00010000); is equivalent to M0R2B4
```

delayscan|delscan(<delay>, <button>, <label1>, <label2>);

This is a hardware control/execution flow command that causes EPIC to delay for <delay> * 20ms before scanning <button>; if <button> is not pressed, execution will jump to <label1>; otherwise execution will jump to <label2>. The delayscan should be preceded by a disablescan of the <button> and both <label1> and <label2> should include the enablescan of the disabled <button>.

clearflag|fclr(<flagname>);

The clearflag command causes <flagname> to be set to "false". The preferred method of setting a flag to false is the simply to set the flag to false with an arithmetic statement: flagname = false;

setflag|fset(<flagname>);

The setflag command causes <flagname> to be set to "true". The preferred method of setting a flag to true is the simply to set the flag to true with an arithmetic statement: flagname = true;

if (<datatype1> <comparison operator> <datatype2>) call|jump <label1>; else call|jump <label2>;

This is the EPICenter implementation of the classic, conditional, "If" statement. With this command, <datatype1> can be any data type except "constant". If the comparison evaluates to "true", <label1> will be executed. If the comparison evaluates to "false" <label2> will be executed. If there is no "else" condition, a

“RETURN” is assumed. If the “call” or “jump” is missing, a “call” is assumed.

Example:

```
if(testByte > 5) jump Function1; else jump Function2;
if(testByte == testWord) Function1;           //call Function1 if the
comparison is equal otherwise execute next line.
```

```
if(<datatype>) call/jump <label1>; else call/jump <label2>;
```

With this command, if the comparison evaluates to “true”, <label1> will be executed.

If the comparison evaluates to “false” <label2> will be executed.

Example:

```
If(testFlag) jump Function1; // will jump to Function1 if testFlag is true.
If(!testFlag) jump Function2; //will jump to Function2 if testFlag is false
```

```
delay(<delay>);
```

This command will cause the EPIC to pause all execution for <delay> * 20ms, before continuing. All loops should include some sort of delay or else real-time interlacing will NOT occur and multiple functions will not be executed. The delay does not have to encompass the whole function execution time unless it is fairly endless. Example:

```
:notendless{
keyhit(1);
delay(2);
loop--;
if(loop) jump notendless;
}
```

This code would be ok if the variable “loop” was not too large a value. The problem is that the key buffer is being loaded faster than unloaded and could overflow if we execute this code too fast and too many keys are sent. Example2:

```
:endless{
keyhit(1);
delay(2);
if(active(M0R0B0) RETURN; else jump endless;
}
```

This loop would go on until the key 0 was hit and would overflow the buffer (if interkey = 40ms and postkey = 30ms) since a key takes 70ms total for each key to get out and the keyboard buffer is being loaded with a key (plus delays) every 40ms or about 2 key entries for every key sent. A delay of about 4 (80ms) per key hit in a loop would be safe.

```
call(<label>);
```

This command causes execution to transfer to the referenced procedure block <label>, then return after that procedure block completes. It is possible to recurse, i.e. for a procedure block to call itself if needed.

```
jump(<label>);
```

This command will cause execution to transfer to the procedure block <label>; however, execution does not return after that procedure block completes. It is possible to “jump” to the start of the procedure block that is currently being executed; in this case, execution begins again at the first command in the procedure block.

setmaxdelta(<channel>,<maxDelta>);

maxDelta range 1-255

This command sets the maximum analog change rate per unit of time. This has the effect of forcing the analog to change at a constant <maxDelta> rate if the physical analog has moved at a greater rate than the <maxDelta> limit. This is good for controlling aiming in programs (such as Mechwarrior) where the analog must be moved fast to position, then slowed down to fine tune. Example:

```
:aim{setmaxdelta(mydevice.torso,1)} ;slow move
:no_aim{setmaxdelta(mydevice.torso,0xFF)} ;fast move
definebutton(M0R1B0,on,aim)
definebutton(M0R1B0,off,no_aim)
```

Queue commands

Queue commands will be sent to the host computer through the PC USB interface. See appropriate program or interface documentation for meaning of values.

enqueue|nq(<event index#>,ON/OFF);

This command will send event <event Index#> (0-255) to the PC program, with an ON or OFF code. Event <event Index#> is a value that the program must interpret. An example event could be 5 as the gear control, ON code is gear up, OFF code is gear down:

```
:gear_up{enqueue(5,ON)};
:gear_down{enqueue(5,OFF)};
```

enqueue16|nqw(<event #>,<data>);

This command will send event <event#> to the PC program, with <data> which must be a valid data type for the <event#>. The event <event #> must be in the range of 0-1023 and <data> must be 16 bit data. The <data> can be any datatype. For example, for a given event# which equals BtnOn, BtnOff, or BtnP, deviceName.buttonName = “on, off, or pulse”; can be used as the <data>.

example:

APevents. AP_APR_HOLD = pulse; will send a pulse event.

setenqueuerow|nqrow(<scanrow>);

This command will report any changes on a <scanrow> to the PC program.

OUTPUT MODULE COMMANDS

SENDDATA|SD(<module>,<row>,<data>);

This command sends raw <data> to <module> <row> (updates internal image).

`SendRawData16(<mod>,<row>,<data>);`

This command sends 16 bit <data> to <mod> <row> and <mod> <row>+1 (no image update)

`deviceName.outputRowName.pointName = on;`

This command turns “on” the output bit defined by <deviceName.output RowName.pointName>.

`deviceName.outputRowName.pointName = off;`

This command turns “off” the output bit defined by <deviceName.output RowName.pointName>.

`deviceName.outputRowName ^= bits; or`

`deviceName.outputRowName.pointName ^= on;`

This command turns “flips” the output bit defined by <deviceName.output RowName> or <deviceName.output RowName.pointName>..

DISPLAY MODULE COMMANDS

`display(<name>, <module>, <start_digit>, <format>,<DP control flag>);`

This command, defined in the device descriptor section, defines the format of an output module. The <name> is the name of the display; <module> is the module it is connected to; <start_digit> is the starting digit after loading downloading code to EPIC; <format> defines the number of digits and the use of a sign (+/-) and decimal point; and the <DP control flag>. Example:

```
device(Display)
{
display(one, 2, 0, "0000", true);    //a 4 digit display with no decimal //point, and
leading 0 fill. Module 2 start digit=0
//DP control is true for Module 2 type //displays.
display(two, 2, 3, "999.99", true); // a 5 digit display with a DP at digit
// two. Leading 0 blanking. Module 2
// starting digit = 3.
display(three,2,8,"-000.0",true);   //a 4 digit display with zero fill and sign digit
display(four,2,16,"-9999",true);    //a signed 4 digit display will leading zero
blanking
}
```

example:

Display.one = 123; will show 0123 on the display

Display.two = 1; will show 1.00 on the display because the left two digits are zero and blanked

Display.two = 0; will show 0.00 on the display

```

Display.two =123.4    will show 123.40
Display.three = -5;   will show -005.0
Display.three = 5;    will show 005.0
Display.four = -5;    will show - 5
Display.four = 5;     will show 5

```

timer(<10 MS INTERVALS>)

This command should be used for critical timing only. For general-purpose timing, use the “delay” command.

example-

```

:loop1sec{
timer(100);
keyhit(A);
if(timing) jump loop1sec;}

```

This code will send an “a” at exactly 1 second intervals. It is preferable that you use the following code instead:

```

:loop1sec{
delay(50);
keyhit(a);
if(timing) jump loop1sec;}

```

This code will send an “a” at approximately 1 second intervals.

```

ph <pigeon hole name> (<PH#>){
    <byte/word> <element name>;    //element0
    ...code to be executed at PH load time....
};

```

Pigeonholes are memory locations that can hold data for either a PC program or EPIC to use. EPIC can send data to or read data from a Pigeonhole and a PC program can do the same. This command defines a pigeonhole to allocate memory space for a PC program to load data into. The total <byte> or <word> definitions can total no more than 4 bytes (a word equals 2-bytes). Valid <byte/word> combinations for this command are:

```

<byte>, <byte>, <byte>, <byte>
<byte>, <word>, <byte>
<byte>, <byte>, <word>
<word> , <word>
<word>, <byte>, <byte>

```

Example:

```

byte testByte;
ph TestPH (5){
    word el0;
    byte el1;
    byte el2;

```

```

testByte = TestPH.el2;

```

```
};
```

```
defineqproc/defqp(<index #>,<EPL procedure>)
```

Qprocs are a means for a PC program to direct EPIC to execute a set procedure based on a given <index #> (range 0-1023). Once <index#> is assigned by the PC programmer, the appropriate EPL procedure will be executed whenever the PC program sends <index#>. Example:

PC programmer has defined index #20 as turn on nose gear lamp and 21 as turn nose gear lamp off.

```
#define nose_gear_led 2,0,0b00000001 //nose gear led at  
//module 2 row 0 bit 0
```

```
defineqproc(20,nose_led_on)
```

```
defineqproc(21,nose_led_off)
```

```
:nose_led_on{setpoint(nose_gear_led);}
```

```
:nose_led_off{clearpoint(nose_gear_led);}
```

DEBUG

```
breakpoint( );
```

For debugging purposes, this command puts a code of 0008 in the backtrace buffer and freezes logging.

```
gendebugtrap(1);
```

For debugging purposes, this command freezes the DEBUG buffer and appends a code of 0008 to the DEBUG buffer.

```
setqlogon(<flags>);
```

This command uses <flags> bits to control functions in the EPIC hardware.

Bit 0 controls logging of the EPIC command flow

Bit 1 controls logging of the input queue and interrupt queue (XQ) entries (bit 7 = 1) to DEBUG buffer. Entries are CMD/Data BYTE(1)

Bit 2 controls trap time logging routines (this will put a code of 000A in the debug buffer)

Bit 3 traps EXCESSIVE time hogging routines (> ½ second) and puts a code of 0009 in DEBUG buffer.

Bit 4 allows user control of test point 2 (TP2). TP2 can be turned on with exec(138) and off with exec(139) command. This is useful to see if a point in your program was hit without having to stop and check the debug buffer breakpoints.

4.3.4 Pre Processor Directives

```
#ifdef <keyword> if <keyword>
```

```
#else
```

```
#endif
```

This is a preprocessor directive and it allows you to conditionally include procedure statements or functions in your EPL code. Statement between the `#ifdef` and matching `#endif` are executed if `<keyword>` is defined. The `#else` command allows you to include statements following the `#else` if the previous `#ifdef` was false.

```
#ifdef <keyword> if <keyword>
#else
#endif
```

This is a preprocessor directive and it allows you to conditionally include procedure statements or functions in your EPL code. Statement between the `#ifndef` and matching `#endif` are executed if `<keyword>` is not defined. The `#else` command allows you to include statements following the `#else` if the previous `#ifndef` was false.

4.3.5 Macros

```
#macro <macro name>(<parameter1,parameter2,parameter3,...>
Macro body
#endmac
#expand <macro name>(parameter list)
```

The macro command allows you to use macros in your EPL project. A macro defines a block of code that starts with the `#macro <macro name>(<parameter1, parameter2, parameter3,...)` directive and terminates with the `#endmac` directive. The macro must be expanded by the `#expand` directive.

Example

```
#macro TypeHello(key1,key2,key3,key4,key5)
keyhit(key1);
keyhit(key2);
keyhit(key3);
keyhit(key4);
keyhit(key5);
testvar = 23;
#endmac
//The line:
#expand TypeHello(h,e,l,l,o)
//Will send:
hello
//And set variable testvar to 23
```

4.3.6 Data Types

Flag <flagname>[=][true,false];

This data type declares a flag <flag> that must be a unique flag identifier. The initial value of <flag> is undefined unless it is initialized to “true” or “false”.

Example:

```
flag testFlag1;  
flag testFlag2 = true;
```

var|var8|byte <variable name>[=] [initial value];

This data type declares an 8 bit (byte) variable <variable name> that must be a unique variable identifier. The value of <variable name> is undefined until it is specified.

```
byte testByte1;  
byte testByte1 = 0x35;
```

var16|word <variable name>[=] [initial value];

This data type declares a 16 bit (word) variable <variable name> that must be a unique variable identifier. The value of <variable name> is undefined until it is specified.

```
word testWord1;  
word testWord2 = 23512;
```

rvar(<name>,<lower limit>,<upper limit>,<preset value>,[wrap flag]);

This data type declares a rotary variable. Rotary variables are unique since they have the ability to wrap around (wrap flag set to “false”) if the <lower limit> or <upper limit> is reached. If the wrap flag is set to “true”, the rvar will not rotate, but will stay within the specified bounds. An example for rotary variable “test” with lower limit = 10 and upper limit = 359, and a preset of 100:

```
rvar(test,10,359,100, false);  
if test=358 and 2 is added to it, the new value = 10 (358+1=359, 359+1=10)  
rvar(test,000,255,000,true);  
if test = 200 and we add 100,  
test will = 255 because 200+100 is greater than 255 so test is set to the upper limit.
```

bcd <name>[=] [initial value];

This data type declares a binary coded decimal in the form: SNNNNNNN.FFF; S is the sign (not implemented yet); NNNNNNN it the number part, and FFFF is the fractional part.

random

This command allows you to assign a random number to a variable. A different random number can be generated every ½ second. Example:

```
testByte == random;
```

4.3.7 Device Descriptors

Device descriptor syntax

```
#pragma hid_rpt_snd x
```

This device descriptor will send x number of the devices starting with device0 (the first one described). For example, #pragma hid_rpt_snd 7 will send devices 0-6 to Direct Input.

```
connector(<name>)
```

This device descriptor associates analogs and module and rows (modrows) with a connector. The elements in this list are relative; therefore, the first analog in the list is 0, the second 1, etc. The same relative ordering is true for modrows. You must define devices with axes, buttons, and POVs before devices with no buttons, axes, or POVs. Example:

```
connector(mod1)
{
    analog(<input analog number>);
    modrow(<module number>,<row number>);
};
```

```
device(<name>)
```

This device descriptor establishes a device <name> that connectors can be attached to. Below is the syntax that may follow the device<name>:

```
connector(<connector name>);
```

```
analog(<relative analog>,<name>,<WinAxis>,[type],[source],[min],[max]);
```

If this is a relative analog then the virtual axis is not associated with connector. For a real axis the defaults for type are Analog8, source=0, min=0, max = 255(0xFF). For WinAxis = X,Y,Z,,Rx,Ry,Rz ,Slider,Dial,Wheel for Gaming Options axis.??????
button(<relative mod,row>,<bit>,<name>);

```
toggle(row, start-bit, name, num_bits, num_positions, ["all_off",] "bit0", "bit1" ...);
```

If the num_positions > num_bits then "all_off" must be specified.

```
hat4(<relative mod,row>,<up bit>,<down bit>,<left bit>,<right bit>,<name>);
```

```
output(<name>,<module>,<row>) ;
```

A row can be broken into bits or a combination of bits.

```
output(<name>,<module>,<row>) {<name> = <bit definitions>; <name>=<bit definitions>;}; Example:
```

```
output(leds1,3,13){ //a row of output points on
                    //module 3 row 13.
    rightLEDred = 0x08; //right gear led red assigned to bit 3
    rightLEDgrn = 0x10;
```

```

noseLEDred = 0x20;
noseLEDgrn = 0x40;
leftLEDred = 0x80;    //left gear led red assigned to bit 7
};
In the source file to turn on the above: Mydevice0.leds1.rightLEDred = on;
to turn off that led                      Mydevice0.leds1.rightLEDred = off;
To turn off the whole row: Mydevice0.leds1 = 0;
To test an output point for being on:
if(Mydevice0.leds1.rightLEDred) { if "on" execute this} else {if "off" execute this}
or
if(!Mydevice0.leds1.rightLEDred) {if "off" execute this} else {if "on" execute this}

```

The following are not associated with the "connector".

```
pov(<point of view name>);
```

```
display(<name>,<module>,<start digit>,<format>,<DP type flag>);};
deviceName.displayName = blank;
```

Allows displays to be blanked.

```
deviceName.displayName = dash;           //for module 2 type displays
                                         // 32 digit display modules
```

Allows displays to be dashed.

```
deviceName.displayName = Edash; //for KR1 radio displays
Allows KR1 radio displays to be dashed.
```

Part 5: Sample Pigeonhole Code

5.1 C++ software to interface to Pigeon Holes and Q-Procs

5.1.1 Generic Pigeon Hole code

The EPIC “_SendPH” command is used to send data to EPIC. The SendPH command comes in two forms and uses a handle to identify which EPIC device you’re talking to, identified which Pigeonhole you want to send data to and allows you to send 2-16bit words or 4-8 bit bytes (32 bits total).

Since the lights of an Output Module are represented by individual bits, we have to provide the data in a special way to ensure we can command the correct bits. To accomplish this task using 4-8 bit bytes, we “AND” the set of bits we’re interested in and shift them to the correct position. As an example, for the 3rd byte of data, we must define it as a UCHAR and logically “AND” (“&”) it with 0xFF0000 and shift it to the 3rd byte position “(>>16”.

Below is the _SendPH command for sending 4-8 bit bytes to EPIC:

```
_SendPH(handle,PHnum,(UCHAR)(tempLight&0xFF),(UCHAR)((tempLight&0xFF00)>>8),(UCHAR)((tempLight&0xFF0000)>>16),(UCHAR)((tempLight&0xFF000000)>>24));  
_GetData(handle);
```

Since the current EPIC code does not start sending data automatically, we’re using the _GetData command to force EPIC to start the data transfer.

```
_GetData(handle); //Temporary fix of a bug in EPIC
```

Of course there is additional code you need so I’ve included my entire Memory Reader project below that’s shows how I interface to Falcon4.

5.1.2 Generic Q-Proc code

To be provided in a later update.

5.1.3 Falcon4SP3 Memory Reader: JAMmemreader Project

```
//Legal Stuff: The following is provided free to non-commercial simbuilders
//for their personal use.
// This is the code to send the Falcon 4's shared memory data to EPIC via
//pigeonholes
#include <windows.h>
#include <stdio.h>
#include <iostream.h>
#include "EPICIO.hpp"
#include "flightdata.h"

//The following defines the handle to access the Falcon4 shared memory
HANDLE gSharedMemHandle = NULL;//Falcon4 shared memory handle
void* gSharedMemPtr = NULL;

int main (void)    //starting point for C++ code
{
//First lets initialize some variables
int handle = -1, JAMCount=0;
int bit = 0;
CHAR JAMCheck = 'y';
int tempLight=0;    //tempLight is used for manually setting lights during testing
int PHnum = 1;//Pigeon Holes used with Falcon4SP3 will be 1, 2, 3 and 4
//The Falcon4 landing gears are not light bits so extra code is needed
int l =0, GearStatusLgt = 0;
int NoseGearTemp=0,LeftGearTemp=0,RightGearTemp=0,GearTrans=0;

//Next lets open the EPIC device
handle = _OpenDevice(1); // 0= EPICISA, 1=EPICUSB0, etc
if (handle <0)    //This would mean an error
{
    cout<<"Error opening EPIC device";
}
//We should either get a handle or an error code

//The following extracts the Falcon4 data from shared memory
FlightData* flightData;
gSharedMemHandle = OpenFileMapping(FILE_MAP_ALL_ACCESS, TRUE, "FalconSharedMemoryArea");
if (gSharedMemHandle)
```

```

        {
            gSharedMemPtr = MapViewOfFile(gSharedMemHandle, FILE_MAP_WRITE, 0, 0, 0);
        }
    else
        {
            printf ("Unable to open file. Is Falcon Running?\n");    //Falcon4 must be running
            exit (0); //Pops open a Dos Window with this message
        }
//The following code allows the manual checking of lights on an Output Module
teststart:
cout<<"Do you want to test lights? ";cin>>JAMCheck;
if (JAMCheck=='n') goto testend;
cout<<"What Output Module? ";cin>>PHnum;
cout<<"What Bit do you want to turn on/off? ";cin>>bit;
tempLight=1;
if (bit==1) goto special;
bit=bit-1;
do
{
    tempLight = tempLight*2;
    bit=bit - 1;
}while(bit>0);
//The following is the EPIC Pigeonhole send command
_SendPH(handle,PHnum,(UCHAR)(tempLight&0xFF),(UCHAR)((tempLight&0xFF00)>>8),(UCHAR)((tempLight&0xFF0000)>>16),(UCHAR)((
tempLight&0xFF000000)>>24));
_GetData(handle);    //Temporary fix of a bug in EPIC
if (JAMCheck == 'y') goto teststart;
else goto testend;
special:    //Required to handle the special case of testing bit "1"
{
    tempLight=1;
    _SendPH(handle,PHnum,(UCHAR)(tempLight&0xFF),(UCHAR)((tempLight&0xFF00)>>8),(UCHAR)((tempLight&0xFF0000)>>16),(UCHAR)((
tempLight&0xFF000000)>>24));
    _GetData(handle);
    if (JAMCheck == 'y') goto teststart;
}
testend:

//-----Now lets talk to Falcon4 shared memory-----

```

```

cout<<"/nDo you want to interface to Falcon4SP3? ";cin>>JAMCheck;
if (JAMCheck=='n') goto finish;
JAMCount = 1;
do
{
    flightData = (FlightData*)gSharedMemPtr;    //Get a copy of the Falcon4 shared memory
    NoseGearTemp = flightData->NoseGearPos; // Get the nose gear state
    LeftGearTemp = flightData->LeftGearPos;    // Get the left gear state
    RightGearTemp = flightData->RightGearPos; // Get the right gear state
    if (NoseGearTemp ==1) {GearTrans=1;}      //Build a word with the gear information
    if (LeftGearTemp ==1) {GearTrans=GearTrans+2;}
    if (RightGearTemp ==1) {GearTrans=GearTrans+4;}

//-----Now lets send all the Falcon4 data in 5 Pigeonholes-----
//-----Pigeonhole 1 has the lightBits data-----
// Look in the flightData.h file to find the data definitions
_SendPH(handle,1,(UCHAR)(flightData->lightBits&0xFF),(UCHAR)((flightData->lightBits&0xFF00)>>8),(UCHAR)((flightData->lightBits&0xFF0000)>>16),(UCHAR)((flightData->lightBits&0xFF000000)>>24));
//-----Pigeonhole 2 has the lightBits2 data-----
_SendPH(handle,2,(UCHAR)(flightData->lightBits2&0xFF),(UCHAR)((flightData->lightBits2&0xFF00)>>8),(UCHAR)((flightData->lightBits2&0xFF0000)>>16),(UCHAR)((flightData->lightBits2&0xFF000000)>>24));
//-----Pigeonhole 3 has the lightBits3 data-----
_SendPH(handle,3,(UCHAR)(flightData->lightBits3&0xFF),(UCHAR)((flightData->lightBits3&0xFF00)>>8),(UCHAR)((flightData->lightBits3&0xFF0000)>>16),(UCHAR)((flightData->lightBits3&0xFF000000)>>24));
//-----Pigeonhole 4 has the hsiBits data-----
_SendPH(handle,4,(UCHAR)(flightData->hsiBits&0xFF),(UCHAR)((flightData->hsiBits&0xFF00)>>8),(UCHAR)((flightData->hsiBits&0xFF0000)>>16),(UCHAR)((flightData->hsiBits&0xFF000000)>>24));
//-----Pigeonhole 5 has the landing gear data-----
_SendPH(handle,5,GearTrans,0,0,0);
GearTrans=0;
_GetData(handle);
//Use a counter to set up a delay before updating data again
for (I = 1; I<=50000; I++);
//This code will continue sending the data until you close the window with a Ctrl-C
}while(JAMCount = 1);
finish:
if (handle>=0) _CloseDevice(handle); //Close the EPIC device
// Close shared memory area
if (gSharedMemPtr)
{

```

```
        UnmapViewOfFile(gSharedMemPtr);
        gSharedMemPtr = NULL;
    }
    CloseHandle (gSharedMemHandle); //Close the Falcon4 shared memory
    return 1; //We're done
}
```

5.1.4 Falcon4SP3 Shared Memory definitions: FlightData.h

```

#ifndef _FLIGHT_DATA_H
#define _FLIGHT_DATA_H
// SP3
class FlightData
{
public:
    enum LightBits
    {
        MasterCaution = 0x1,

        // Brow Lights
        TF      = 0x2,
        OBS     = 0x4,
        ALT     = 0x8,
        ENG_FIRE = 0x20,
        CONFIG  = 0x40,
        HYD     = 0x80,
        OIL     = 0x100,
        DUAL    = 0x200,
        CAN     = 0x400,
        T_L_CFG = 0x800,

        // AOA Indexers
        AOAAbove = 0x1000,
        AOAOOn   = 0x2000,
        AOABelow = 0x4000,

        // Refuel/NWS
        RefuelRDY = 0x8000,
        RefuelAR  = 0x10000,
        RefuelDSC = 0x20000,

        // Caution Lights
        FltControlSys = 0x40000,
        LEFlaps       = 0x80000,
        EngineFault   = 0x100000,
        Overheat      = 0x200000,
        FuelLow       = 0x400000,
        Avionics      = 0x800000,
        RadarAlt      = 0x1000000,
        IFF           = 0x2000000,
        ECM           = 0x4000000,
        Hook          = 0x8000000,
        NWSFail       = 0x10000000,
        CabinPress    = 0x20000000
    };
};

```

```

enum LightBits2
{
    // Threat Warning Prime
    HandOff    =    0x1,
    Launch     =    0x2,
    PriMode    =    0x4,
    Naval      =    0x8,
    Unk        =    0x10,
    TgtSep     =    0x20,

    // Aux Threat Warning
    AuxSrch    =    0x1000,
    AuxAct     =    0x2000,
    AuxLow     =    0x4000,
    AuxPwr     =    0x8000,

    // ECM
    EcmPwr     =    0x10000,
    EcmFail    =    0x20000,

    // Caution Lights
    FwdFuelLow =    0x40000,
    AftFuelLow =    0x80000,

    //Engine lights
    EPUOn      =    0x100000,
    JFSON      =    0x200000,
    SEC        =    0x400000,

    OXY_LOW    =    0x800000,
    PROBEHEAT  =    0x1000000,
    SEAT_ARM   =    0x2000000,
    BUC        =    0x4000000,
    FUEL_OIL_HOT = 0x8000000,
    ANTI_SKID  =    0x10000000,
    TFR_ENGAGED = 0x20000000,
    GEARHANDLE =    0x40000000,
};

```

```

enum LightBits3
{
    FlcsPmg    =    0x1,
    MainGen    =    0x2,
    StbyGen    =    0x4,
    EpuGen     =    0x8,

```

```

    EpuPmg      = 0x10,
    ToFlcs     = 0x20,
    FlcsRly    = 0x40,
    BatFail    = 0x80,
    Hydrazine  = 0x100,
    Air        = 0x200,
    Elec_Fault = 0x400,
    Lef_Fault  = 0x800,
};

enum HsiBits
{
    ToTrue      = 0x01, // HSI_FLAG_TO_TRUE
    IlsWarning  = 0x02, // HSI_FLAG_ILS_WARN
    CourseWarning = 0x04, // HSI_FLAG_CRS_WARN
    Init        = 0x08, // HSI_FLAG_INIT
    TotalFlags  = 0x10, // HSI_FLAG_TOTAL_FLAGS ???
    ADI_OFF     = 0x20, // ADI OFF Flag
    ADI_AUX     = 0x40, // ADI AUX Flag
    ADI_GS      = 0x80, // ADI GS FLAG
    ADI_LOC     = 0x100, // ADI LOC FLAG
    HSI_OFF     = 0x200, // HSI OFF Flag
    BUP_ADI_OFF = 0x400, // Backup ADI Off Flag
    VVI         = 0x800, // VVI OFF Flag
    AOA         = 0x1000, // AOA OFF Flag
    AVTR        = 0x2000, // AVTR Light
};

```

```

// These are outputs from the sim
float x;      // Ownship North (Ft)
float y;      // Ownship East (Ft)
float z;      // Ownship Down (Ft)
float xDot;   // Ownship North Rate (ft/sec)
float yDot;   // Ownship East Rate (ft/sec)
float zDot;   // Ownship Down Rate (ft/sec)
float alpha;  // Ownship AOA (Degrees)
float beta;   // Ownship Beta (Degrees)
float gamma;  // Ownship Gamma (Radians)
float pitch;  // Ownship Pitch (Radians)
float roll;   // Ownship Pitch (Radians)
float yaw;    // Ownship Pitch (Radians)
float mach;   // Ownship Mach number
float kias;   // Ownship Indicated Airspeed (Knots)
float vt;     // Ownship True Airspeed (Ft/Sec)

```

```

float gs;          // Ownship Normal Gs
float windOffset; // Wind delta to FPM (Radians)
float nozzlePos;  // Ownship engine nozzle percent open (0-100)
float internalFuel; // Ownship internal fuel (Lbs)
float externalFuel; // Ownship external fuel (Lbs)
float fuelFlow;   // Ownship fuel flow (Lbs/Hour)
float rpm;        // Ownship engine rpm (Percent 0-103)
float ftit;       // Ownship Forward Turbine Inlet Temp (Degrees C)
float gearPos;    // Ownship Gear position 0 = up, 1 = down;
float speedBrake; // Ownship speed brake position 0 = closed, 1 = 60 Degrees
open
float epuFuel;    // Ownship EPU fuel (Percent 0-100)
float oilPressure; // Ownship Oil Pressure (Percent 0-100)
int  lightBits;  // Cockpit Indicator Lights, one bit per bulb. See enum

// These are inputs. Use them carefully
float headPitch; // Head pitch offset from design eye (radians)
float headRoll;  // Head roll offset from design eye (radians)
float headYaw;   // Head yaw offset from design eye (radians)

// new lights
int  lightBits2; // Cockpit Indicator Lights, one bit per bulb. See enum
int  lightBits3; // Cockpit Indicator Lights, one bit per bulb. See enum

// chaff/flare
float ChaffCount; // Number of Chaff left
float FlareCount; // Number of Flare left

// landing gear
float NoseGearPos; // Position of the nose landinggear
float LeftGearPos; // Position of the left landinggear
float RightGearPos; // Position of the right landinggear

// ADI values
float AdillsHorPos; // Position of horizontal ILS bar
float AdillsVerPos; // Position of vertical ILS bar

// HSI states
int  courseState; // HSI_STA_CRIS_STATE
int  headingState; // HSI_STA_HDG_STATE
int  totalStates; // HSI_STA_TOTAL_STATES ???

// HSI values
float courseDeviation; // HSI_VAL_CRIS_DEVIATION
float desiredCourse; // HSI_VAL_DESIREDCRS
float distanceToBeacon; // HSI_VAL_DISTANCE_TO_BEACON

```

```

float bearingToBeacon; // HSI_VAL_BEARING_TO_BEACON
float currentHeading;   // HSI_VAL_CURRENT_HEADING
float desiredHeading;   // HSI_VAL_DESIRED_HEADING
float deviationLimit;   // HSI_VAL_DEV_LIMIT
float halfDeviationLimit; // HSI_VAL_HALF_DEV_LIMIT
float localizerCourse;  // HSI_VAL_LOCALIZER_CRS
float airbaseX;         // HSI_VAL_AIRBASE_X
float airbaseY;         // HSI_VAL_AIRBASE_Y
float totalValues;     // HSI_VAL_TOTAL_VALUES ???

float TrimPitch; // Value of trim in pitch axis, -0.5 to +0.5
float TrimRoll;  // Value of trim in roll axis, -0.5 to +0.5
float TrimYaw;   // Value of trim in yaw axis, -0.5 to +0.5

// HSI flags
int hsiBits; // HSI flags

//DED Lines
char DEDLines[5][26]; //25 usable chars
char Invert[5][26]; //25 usable chars

//PFL Lines
char PFLLines[5][26]; //25 usable chars
char PFLInvert[5][26]; //25 usable chars

//TacanChannel
int UFCTChan, AUXTChan;

// RWR
int RwrObjectCount;
int RWRsymbol[20];
float bearing[20];
unsigned long missileActivity[20];
unsigned long missileLaunch[20];
unsigned long selected[20];
float lethality[20];

//fuel values
float fwd, aft, total;

void SetLightBit (int newBit) {lightBits |= newBit;};
void ClearLightBit (int newBit) {lightBits &= ~newBit;};
int IsSet (int newBit) {return ((lightBits & newBit) ? TRUE : FALSE);};

void SetLightBit2 (int newBit) {lightBits2 |= newBit;};
void ClearLightBit2 (int newBit) {lightBits2 &= ~newBit;};

```

```
int IsSet2 (int newBit) {return ((lightBits2 & newBit) ? TRUE : FALSE);};

void SetLightBit3 (int newBit) {lightBits3 |= newBit;};
void ClearLightBit3 (int newBit) {lightBits3 &= ~newBit;};
int IsSet3 (int newBit) {return ((lightBits3 & newBit) ? TRUE : FALSE);};

void SetHsiBit (int newBit) {hsiBits |= newBit;};
void ClearHsiBit (int newBit) {hsiBits &= ~newBit;};
int IsSetHsi (int newBit) {return ((hsiBits & newBit) ? TRUE : FALSE);};

int VersionNum; //Version of Memarea
};

extern FlightData cockpitFlightData;
#endif
```

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.