MicroCockpit
Custom and design for simulation

RR Electronics

GUIDE DE L'UTILISATEUR

Quicklinks:

addpost, addr, ahit, althit, analog, bcd, breakpoint, button, byte, call, chit, clearflag, clearpoint, clrp, connector, ctrldel, ctrlhit, defbtn, definebutton, defmod, definemodule, defineqp, defineqproc, defmod, delay, delayscan, delscan, device, disablescan, discan, display, enablescan, enque, enque16, enscan, fclr, flag, flippoint, fset, gendebugtrap, hid_rpt_snd, if, #include, int, interk, jump, keyhit , keypress, keyrelease, khit, kpr, krl, nq, nqw, output, ph, postk, pulsew, random, rawkey, rvar, scanrow, sd, senddata, sendrawdata16, sendser16, sendser32, setp, setpoint, setctrldelay, setflag, setinterkey, setmaxdelta, setpostkey, setqlogon, shifthit, shit, switch/case , timer, urvar, var, var8, var16, word

EEPROM loading procedure

Output Module Commands

Display Module Commands

Mouse commands

Device Descriptor Syntax

==========

EPL Syntax Reference

This is a description of the EPIC Control Language (EPL) as it stands right

now; it is by no means a complete reference, and the specification is still

evolving. See manual for any more recent command additions.

General EPL Syntax

All text between angle brackets "< >" is descriptive and should not be taken literally. All options separated by vertical lines "I" are mutally exclusive.

All options within square brackets "[ ]" are optional, options not within square brackets are required.

// <comment>

A comment. All text following the double slashes and preceding the end-of-line is ignored.

Comments may appear anywhere in your program.

<label>

<flag>

<var>

block comment:

/* start of block comment

.....block to be commented

*/ end of block comment

Identifiers (ie. string literals), 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. Indentifiers can only start with an Alpha (A-Z) or underscore (_)

: <label> { <function(s)> }

as of EPICenter Version 45 the format:

<return type> <label>(<arg type> <arg>,....)

can be used.

:ThisLabel{ }

is the same as:

void ThisLabel(void) { }

This is in preparation for argument pass and return values.

The only supported form is, for example:

void GetAirSpeed(void){ ....} //no arguments, no return value

A procedure block. The colon should be followed by a unique block label.

Function calls are placed between curly braces and are executed in the order listed. A "return" is assumed at the end of the procedure block. Comments and line breaks can be interspersed as desired.

:INIT{ }

INIT is a special procedure that is executed whenever the project is loaded to EPIC. :INIT must be spelled with capitol letters.

Function name spelling:

keypresslkpr(<key>);

means this function can be KEYPRESS(<key>) or KPR(<key>)

There is a predefined label called RETURN that can be used in place of any

label reference. A "jump" to RETURN exits the level of local block by jumping to the "}" that terminates the block.

:MyFunction

{

if(myFlag) jump RETURN;

.....other code .....

}

<<<<<if myFlag is set, there will be a jump to this point, exiting "MyFunction

```
:MyFunction
{
if(myFlag)
{
if(myFlag1) jump RETURN;
....other code
}
    <<<< if myFlag1 is set the jump will be here bypassing the rest of the local block and
    executing ...more code...
.....more code....
}
```

#define <string> <value>

Preprocessor directive to replace string with value.

```
#define AAAA 0x25
:MyFunction
{
if(myVariable == AAAA) {myVariable++;}
}
```

if myVariable is equal to 0x25 execute the local block and increment myVariable.

```
;*****************************************
```

Preprocessor conditional directives

```
#ifdef <keyword>
```

if <keyword> is defined include the lines between
the #ifdef and the matching #endif

```
#ifndef <keyword>
```

if <keyword> is NOT defined include the lines
between the #ifndef and its matching #endif

```
#else
```

include the statements following #else if the
previous #ifdef or #ifndef was false

```
#endif
```

terminator for conditional blocks

```
;*****************************************
```

Macros

A macro is defined in a block that terminates with the #endmac directive.

The macro is expanded by the #expand directive

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

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

;**********

General Program Form

EPL programs should have the following form:

```
#include <epicdefs.hpl>
```

common definitions. This is in the EPIC\INCLUDE directory as noted by the < >

... module definitions ... These can be at the top of the device descriptor file

```
#include “deviceName.hpl”
```

deviceName is your device descriptor file name and is in your project directory as noted by the “ “

```
#include “<header file name1>.hpl”
```

one line for each header file. These will include function prototypes, #defines , and other data common to the source file of the same name.

< global flag declarations >

< global variable declarations >

```
#include “<source file name”.epl>”
```

< procedure blocks >

Procedure blocks are the “meat” of the program-where the work actually gets done. All button actions (ie. key and button presses) are defined in this section.

The <button definitions> section defines which procedure block is called for individual button actions (ie. button presses and button releases) and is required.

Numbers can be expressed in decimal, hexidecimal, or binary. If a number is to be a hexidecimal number it must be distinguished by preceding it with “0x”. If a number is to be in binary it must be distinguished by preceding it with “0b”

Example: the number 128 can be:

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

Data Types

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

Declare a flag. <flag> must be a unique flag identifier. The initial value of <flag> is undefined unless the intial “true” or “false” is specified.

Example:

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

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

Declare an 8 bit (BYTE) variable. <variable name> must be a unique variable identifier. The initial value of <variable name> is undefined if not specified.

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

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

Declare an unsigned 16 bit variable

range 0 to 65535 (0xFFFF)

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

int <variable name> [=] [initial value];

Declare a signed 16 bit variable.

range -32768 (0x8000) to +32767 (0x7FFF)

********** **Rotary Variables (16bit)** **********

rvar(<name>,<lower limit>,<upper limit>,<preset value>,[no wrap flag]); (signed. range -32768 to 32767)

urvar(<name>,<lower limit>,<upper limit>,<preset value>,[no wrap flag]); (unsigned. range 0 to 65535)

Rotary variables wrap around if the min/max limits hit

example- Rotary variable 1 with min=10 and max=359

rvar(rvar1,10,359,100);

if rvar1=358 and 2 added to it the new value = 10 (358+1=359, 359+1=10)

If the “wrap flag” is set to true, the rvar will NOT rotate, but will stay in the specified bounds.

Wrap flag default is false (wrap)

rvar(testRvar,000,255,000,true);

if

testRvar = 200;

testRvar += 100; testRvar will = 255. 200+100 is greater than 255 so testRvar is set to the upper limit.

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

binary coded decimal

range +-9,999,999.9999

STTTUUUU.FFFF

S=sign (V1.082 and above)

TTT = tens of thousands

UUUU = units

FFFF=fractional part

example:

bcd testBCD;

testBCD = -9876543.1234;

tth(<bcd variable>) extract tth word from bcd variable (this will include sign bit)

units(<bcd variable>)

fracts(<bcd variable>)

example:

testWord = units(testBCD); //extract units part (0 to 9999) of bcd number to variable testWord.

testWord will now = 0x6543

This is useful for sending a BCD number to Flight Sim.

addr <name>; **EPICenter version 1.0.7.6 and above**

A pointer type. “name” can take the address of another variable, array, etc.

example:

```
  addr word_ptr;
  word myWord;
:MyFunc
{
  word_ptr = @myWord; //set word_ptr to the address of myWord
}
```

random

Note- Random number is generated every ½ second

Example:

```
If(testByte == random) {....}
```

Procedure Block Execution

Execution of a procedure block begins with the first command in the block and continues with successive commands until a RETURN is encountered. A RETURN is assumed at the end of a procedure block and need not be explicitly declared.

Please note that execution continues through all referenced procedure blocks until a RETURN is encountered, and that it is possible to recurse (ie. Call the same procedure block more than once from itself).

```
;*********************************
```

---EEPROM loading procedure---

1. Unplug USB cable

Remove power from EPICUSB to remove any existing running program.

2. Replug power to EPICUSB.

3. Replug USB cable to EPICUSB.

4. EPICenter- Tools I Hardware I LoadEEPROM

5. Select the EEPROM file. This will normally be EPICUSB.BIN or USBxxxx.Bin where xxxx is the checksum.

6. 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 the there will be a final loaded ok message.

Steps 1-3 should not be necessary, but it is best to be safe.

If it looks like the loading gets hung up and stops responding with the “sanity” light flashing fast and “USB ISR”, hit CTRL-ALT-DELETE, task manager, Applications, EPICenter, End Task, or wait until EPICenter comes back with the “Not Responding” message.

Unplug USB cable, unplug and replug power cable, plug USB cable back in, rerun EPICenter.

Tools I EPIC_info

If Checksum is ok, you are done, if not, retry the EEPROM write again.

COMMANDS

Commands must be in lower case.

Note - **definebutton** command is in the syntax for backwards compatibility. We recommend defining a switch in a device and using **deviceName.switchName.On** or **.Off** instead.

definebuttonIdefbtn(<button>, onloff, <label>)

Associate a procedure block with a specific button action. <button> is the

button being assigned. This can be in the formats:

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

MRM- Module,Row,Mask example:

M1R3M0xFF module1 row3 mask all bits

M1R3M0x0F module1 row3 include bits 0-3

, on (press) and off (release) are the only two actions allowed, and <label> is the procedure block to execute when the action occurs.

As of Version44 of EPICenter:

deviceName.buttonName.On{.....}

and

deviceName.buttonName.Off{.....}

can be used.

example:

APpanel.APon.On{.....}

Version 46 and up

void deviceName.buttonName.On(void) {.....}

definemoduleIdefmod(<module number>,<mtype>,<start row>,<numrows>)

mtype = 0 (FASTSCAN) for high priority scan (maximum 16 rows)

mtype = 1 (SLOWSCAN) for low priority scan

mtype = 2 (OUTPUT)for output module

startrow = first row to scan in this group

numrows = number of rows to scan starting with startrow

OUTPUT modules must be defined to allow setting and reseting indiviual bits

DO NOT define display modules using definemodule. Define a display in the device descriptor file instead.

keyhit|khit(<key>);

Press and then release <key>. <key> must be a valid key identifier; see TOKENS.DOC for all the possibilities. Keynames **MUST** be in upper case

keypress|kpr(<key>);

Press but do not release <key>. <key> must be a valid key identifier; see TOKENS.DOC for all the possibilities. A keypress with HID keys will keep repeating “key” until a keyrelease is executed.

keyrelease|krl(<key>);

Release <key> without pressing it. <key> must be a valid key identifier;

see TOKENS.DOC for all the possibilities. <key> should have been previously “pressed” using the “keypress” command.

rawkey(<value>);

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

shifthit|shit(<key>);

Press and then release <key> while holding down the shift key. <key> must be a valid key identifier; see TOKENS.DOC for all the possibilities. To pressand/or release one of the shift keys by itself, use “keypress” and “keyrelease”.

ctrlhit|chit(<key>);

Press and then release <key> while holding down the Control key. <key> must be a valid key identifier; see TOKENS.DOC for all the possibilities. To press and/or release the Control key by itself, use “keypress” and “keyrelease”.

althit|ahit(<key>);

Press and then release <key> while holding down the Alt key. <key> must be a valid key identifier; see TOKENS.DOC for all the possibilities. To press and/or release the Alt key by itself, use “keypress” and “keyrelease”.

pulsew(<value>); **set interval for “pulse” keyword**

value = number of 20ms intervals.

example:

pulsew(5); //5 * 20ms = 100ms

will cause

DeviceName.ButtonName = pulse;

to go “on” for 100ms, then “off”

setinterkey|interk(<delay>);

An advanced hardware control command. Set the delay between “make” (ie. press) and “break” (ie. release) codes to <delay> * 10ms. Default is 40 milliseconds.

setpostkey|postk(<delay>);

An advanced hardware control command. Set the “post-break” delay (ie. The delay between “break” and “make” codes) to <delay> * 10ms.

Default is 30 milliseconds.

addpost(<delay>,<delay1>) ;

Insert delay(10 millisecond intervals) between key release and CTRL,ALT,

or SHIFT for the next ctrlhit,althit, or shifthit. This applies only to

the next command only. Range 0-15. <delay1> applies to AFTER the CTRL,ALT, or SHIFT release.

Example-

```
:ThisFunction
{
addpost(5,0);
ctrlhit(A);
ctrlhit(B);
}
```

will add 50milliseconds between the “A” release and the CTRL release, but does not apply to the ctrlhit(B).

Example-

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

will add 50milliseconds between the “A” release and the CTRL release and a 20 millisecond delay after the CTRL is released (before the CTRL press for the “b” key, but does not apply to the ctrlhit(B).

setctrldelay|ctrldel(<delay>);

Will insert 10millisecond delays as above but applies to all ctrlhit,althit,

and shifthit commands. Range 0-15

example-

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

will inject 50 extra milliseconds between the keyrelease and the **CTRL**,**ALT**,or **SHIFT** hits.

The following is OBSOLETE:

ifactive (<module,row,mask>) callljump <label1>; else callljump <label2>;

ifactive (<MRB button>) callljump <label1>; else callljump <label2>;

<mask> is a button bit mask ; If the button specified by <button> or any of the buttons specified by <module,row>, <mask> are active (ie. pressed) <label1> is executed, otherwise <label2> is executed. If no “else” condition is declared a “return” is assumed.

Ifactivematch(<Module>,<Row>,<Mask>,<Value>) CallIJump <Label1>; else callljump <label2>;

Scanrow bits are masked (only bits where <mask> has a one in it are looked at),

and masked bits compared to bits in <value>. If they match <label1> is executed,

if no match <label2> is executed.

0 in value = switch closed.

example- mask=0b00001111 value=0b00001111 scan=0b11110111 would not match

example- mask=0b00001111 value=0b00000111 scan=0b00000111 would match

example- mask=0b00001111 value=0b00000111 scan=0b00000101 would not match

example- mask=0b10000001 value=0b00000000 scan=0b11110111 would not match

example- mask=0b10000001 value=0b00000000 scan=0b01110110 would match

example- mask=0b10000001 value=0b00000000 scan=0b11110110 would not match.

>>>>>>>>>>>>>USE<<<<<<<<<<<<<<<<<

1. for ifactive(....)

use:

```
if(deviceName.ButtonName) ..... else ......
```

2. for ifactivematch(.....)

use:

```
byte scan;
scan = scanrow(0,0);    //get current state of switches on module 0 row 0
scan &= 0x0F;           //mask bits of interest
if(scan == 0b00000001) jump Proc0; //active switches (on) are 1’s
if(scan == 0b00000010) jump Proc1;
if(scan == 0b00000100) jump Proc2;
if(scan == 0b00001000) jump Proc3;
if(scan == 0b00000101) jump Proc4; //if switches 0&2 are on and 1&4 are off
```

or

instead of the if(scan ==..... lines

```
switch(scan)
{
```

```
case 1 : jump Proc0;
case 2 : jump Proc1;
case 3 : jump Proc2;
case 4 : jump Proc3;
case 5 : jump Proc4;
}
```

scanrow(<module>,<row>)

returns the value at mod,row 1's = active(on) switches

module and row must be defined as being scanned using definemodule command, otherwise and 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>);

A hardware control command. Disable scanning of <button> or masked row until reenabled. See SCAN.DOC for more information.

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>);

A hardware control command. Enable scanning of <button>. See SCAN.DOC for more information.

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

A hardware control/execution flow command. Delay for <delay> * 20ms, then scan <button>; if <button> is not pressed, execute <label1>; otherwise, execute <label2>. The "delayscan" should be preceded by a "disablescan" of <button> and both <label1> and <label2> should "enablescan" <button>.

clearflag|fclr(<flagname>) ;

```
flagname = false; (preferred)
```

Clear (ie. make "false") <flag>.

setflag|fset(<flagname>);

flagname = true; (preferred)

Set (ie. make “true”) <flag>.

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

example:

```
if(testByte > 5) jump Function1; else jump Function2;

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

A conditional. Datatype1 can be any data type except “constant”. If the comparison evaluates to “true”, <label1> will be executed. If the expression evaluates to “false” <label2> will be executed. If no “else” condition is declared a “return” is assumed. If “call” or “jump” is missing a “call” is assumed.

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

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>);

delay range 1 to 65535

Pause for <delay> * 20ms, then continue execution. All loops should include some sort of delay or else realtime 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;
}
```

would be ok if loop was not too large a value. The problem is that the key buffer is being loaded faster than unloaded and could overflow if loop is too fast and too many keys are being loaded.

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

This loop would continue until the key 0 was hit and

would overflow the buffer(with interkey=40ms and postkey=30ms) as the key takes 70ms total for each key to get out and is being loaded with a key(and delays) every 40ms or about 2 key entrys for every key sent. A delay of about 4 (80ms) per key hit in a loop would be safe.

delayv(<20ms * 16bitvalue>);

Obsolete see delay( );

call(<label>);

alternate use <label>( );

Execute the referenced procedure block <label>, then return to next line when the called procedure is completed.

Note: Procecure formats must be defined ahead of their usage. If a procedure exists after the calling procedure the called procedure's prototype must be defined before the calling procedure. This is usually done in a header file (.hpl) and this file included at the top of the main source (.epl) file.

example:

In MySource.epl:

```
void MyFunction(void) //note that :MyFunction is the same as void MyFunction(void)
{
Function2( ); //call Function2.
}
```

If Function2 is not prototyped or before MyFunction, the compiler will not know how to call Function2.

Function2 can be defined in MySource.hpl as

```
void Function2(void);
```

note the semicolon.

In the project main source (say MyProject.epl)

```
#include <epicdefs.hpl> //common definitions in EPIC\INCLUDE directory
#include "mydevices.hpl" //device descriptor file
#include "mysource.hpl" //header file for MYSOURCE.EPL
.....other included header files....
#include "mysource.epl" //source code
.....other included source files...
.....source for "MyProject" here.....
```

all source files can now see the function prototypes for "mysource.epl"

You should avoid recursion (i.e. a procedure block calling itself).

jump(<label>);

An absolute JUMP to the procedure block <label>. It is possible to "jump" to the current procedure block; in this case execution begins again at the first command in the procedure block.

setmaxdelta(<channel>,<maxDelta>);

channel range 0-16

maxDelta range 1-255

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

```
: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.

; destID is the destination ID of the target program. For example if FS Communicator is logged in with a destID of 25 then nq(15,ON,25); will send the 15 "ON" command to FS Communicator

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

Will send event(0-255) to the PC program with ON or OFF code.

Event is a value that the program interprets.

Example-Event of 5 is gear control ON code is gear up, OFF code is gear down

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

enque16|nqw(<event #>,<data>,<destID>));

event # in range of 0-1023 (0-0x3FF)

data is 16 bit data. This can be any datatype except BCD

destID is the ID of the program to send the data to.

If destID is missing, destID will default to 9. This is for backwards compatibility with Peter Dowson's EPICINFO.DLL.

for event# = **BtnOn**, **BtnOff**, or **BtnP** **deviceName.buttonName = on/off/pulse/toggle** ; can be used

example:

APevents. AP_APR_HOLD = pulse; will send a pulse event to device “APevents” button “AP_APR_HOLD” . If device **APevents has options(SEND_BUTTON_DLL)**; then this event will go to the DLL that has requested the device data from that device.

APevents.AP_APR_HOLD = toggle; will flip the event bit, if it was “on” it will now be “off”, and vice-versa.

********** OUTPUT MODULE COMMANDS **********

module,row,data can be var8,var16,constant

senddata | sd(<module>,<row>,<data>);

Send raw data to module row (updates internal image)

sendrawdata16(<mod>,<row>,<data>);

sends 16 bit data to mod,row and mod,row+1 (no image update)

setpoint | setp(<module>,<row>,<bit(s)>);

(Prefered: use deviceName.outputRowName.pointName = on;)

clearpoint | clrp(<module>,<row>,<bit(s)>);

(Prefered: use deviceName.outputRowName.pointName = off;)

flippoint | flipp(<module>,<row>,<bit(s)>) ;

Reverses current state of bit(s)

(obsolete, use

deviceName.outputRowName ^= bits;

or

deviceName.outputRowName.pointName ^= on;)

ifpoint | ifpnt (<module,row,bits>) CALL|JUMP <label1>; else CALL|JUMP <label2>;

bit(s) only acts on bits that are set to 1.

(obsolete, use

if(deviceName.outputRowName.pointName) .....

********** **DISPLAY MODULE COMMANDS** **********

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

example:

defined in device descriptor sections

```
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
display(five,2,21,"-9999999.99",true); //(V1.082) a signed number with a range of +-
                                         // 9,999,999.99 with 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>);

Use this for critical timing only. For general pupose timing use DELAY command

example-

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

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

Normally use-

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

This 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....
};
```

define pigeon hole- allocates memory space for a PC program to load data to

NOTE total definitions must total to no more than 4 bytes. Combinations 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>) Note: no terminating ;

index # (range 0-1023) is assigned by the PC programmer

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);}
```

breakpoint(1); or gendebugtrap(1);

Freeze DEBUG buffer. Code of 0008 appended to DEBUG buffer.

setqlogon(<flags>);

bit 0= Log EPIC command flow

bit 1= Log input queue and interrupt queue (XQ) entries(bit 7 = 1) to DEBUG buffer. Entries are CMD/Data BYTE(1)

Example entry of 1005 = cmd 10h (load PH) data 05 LOADPH #5. PH data is not logged

9001= XQ loaded PH #1

bit 2 = trap time hogging routines (this will put a code of 000A in the debug buffer)

bit 3 = trap EXCESSIVE time hogging routines (> ½ second) puts a code of

0009 in debug buffer.

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

Serial module commands

Sends serial data to serial module. This data can be used to send data to a second EPIC. With DLL version 1.700 and above data can be sent to another EPIC using nqw(PH#,data16,EPICUSBx); where x=EPICUSB # 0-3

sendser16(<PH#>,<data>);

where data is a byte,word, or constant (range 0-65535 (0xFFFF))

data appears in bytes 0,1 of pigeon hole, bytes 2,3 are 0

```
sendser16(25,0x1234);
```

sendser32(<PH#>,<data0>,<data1>);

data0 appears in pigeon hole bytes 0,1

data1 appears in pigeon hole bytes 2,3

sendser32(15,0x1234,0x4567);

********** **M O U S E** **********

V53 and greater

You must have **MouseDevice.hpl** included **As The LAST** device your device descriptor file or after the #include for the device descriptor file:

#include “xxxxx.hpl” where xxxxx is the name of your device descriptor file

#include <Mousedevice.hpl> This is in your \epic\include directory

Absolute mouse:

Mouse.X = xxxx; Where xxxx = 0 to 0x7FFF

Mouse.Y = yyyy;

0,0 is upper left corner, 0x4000,0x4000 is center, 0x7FFF,0x7FFF is lower right

Relative mouse:

Mouse.rel.X= xx; where xx is –127 to 127

Mouse.rel.Y= yy;

mousebuttons:

right

middle

left

Mouse.right = on; // right button down

Mouse.right=off; //right button up

To hide mouse cursor, move to 0x7FFF,0x7FFF

==========

```
switch(argument){
case <constant0> :
execute code here if argument == constant0 for this case.
case <constant1>:
execute code here if argument == constant1
.....other cases(128 maxium total cases ).....
default:
optional. Execute code here if no cases are matched.
}
```

DEVICE DESCRIPTOR SYNTAX

#pragma hid_rpt_snd x

will send x number of the devices starting with device0 (the first one described)

example:

#pragma hid_rpt_snd 7

will send devices 0-6 to Direct Input (Game controllers)

connector(<name>)

{

analog(<input analog number>);

modrow(<module number>,<row number>);

}; Note the terminating ;

Elements in this list are relative. The first analog in the list is 0, second 1, etc. Same for modrow.

Note: Direct Input devices with axes, buttons, and POVs defined MUST come before physical devices as the first hid_rpt_snd devices will be sent to Direct Input

device(<name>)

{

options(<option type>,<option type>,<option type>);

option types-

SEND_BUTTON_DLL will send a button event in this device to EPICIO.DLL IF the button has no procedure attac

(not implemented) SEND_ANALOG_DLL will send output analogs data to EPICIO.DLL if there is a change in any o

(not implemented) SEND_POV_DLL will send POV data for this device if there is a change in the POV position

connector(<connector name>);

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

//if relative analog = Virtual axis is not associated with connector.

//if real axis, defaults for type=Analog8, source=0,min=0,max = 255(0xFF)

//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 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 in to bits or 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}
```

//not associated with “connector”

```
pov(<point of view name>);
display(<name>,<module>,<start digit>,<format>,<DP type flag>);
}; Note the terminating ;
```

displays can be blanked by:

```
deviceName.displayName = blank;
```

displays can be dashed by:

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

or

```
deviceName.displayName = Edash;     //for KR1 radio displays
```

Analog Primer

Analog processing is split into two sections: input side (i.e. the physical pot) and the output side (what Windows is going to see in Gaming Options).

At the moment, access to the input side is provided by the Analog Settings Window. Use this window to «turn on» your physical analogs and calibrate them.

The output side is controlled by device descriptors in your EPL code. We recommend separating your device descriptors from the rest of your EPL code by placing them in another file (devices.hpl is a good choice).

The Analog Settings window will eventually be replaced with a Device Editor that will also generate the device descriptors. Until then, you'll have to calibrate your analogs in the Analog Settings window and write your device descriptors by hand.

The JOY File

The JOY file contains all the information about your physical pots and their calibration. This file should contain proper calibration data for all of your pots, even if they're not used in a particular project. To save confusion, map your analogs «straight through» (i.e. 0 to 0, 1 to 1, etc.). This information is referenced by the connector and device descriptors and, if missing, could hose things up.

Device Descriptors

Device descriptors describe the logical implementation of your devices and create the data structures that make them show up in Windows' Gaming Options control applet. The general format is:

```
device(ModCombatStick)
{
                        // module & analog references are relative to selected connector
  connector(FLCS);     // must use FLCS connector
// analog(reference, name, WinAxis);
  analog(0, Aileron, X);
  analog(1, Elevator, Y);
// analog(VIRTUAL, name, WinAxis, type=MEM8, source=0, min=0, max=255);
  analog(VIRTUAL, Virtual, Rz, , , 0x02, 0xFD);
// button (modrow, bit, name)
  button(0, 0, Trigger);
  button(0, 1, GreyThumb);
  button(0, 2, RedThumb);
  button(0, 3, Pinky);
  button(0, 4, Btn4);
  button(0, 5, Index);
  button(0, 6, Dummy1);
  button(0, 7, Dummy2);
// hat4 (modrow, up-, down-, left-, right-bit, name)
  hat4(1, 3, 0, 1, 2, Hat1);
  hat4(1, 7, 4, 5, 6, Hat2);
// pov (name);
  pov(POV1);}
```

As of build 0.1.1.37 the analog statements in the device descriptors create structures that have the following elements:

```
byte type;
word source;
byte max_delta;
word pre_offset;
byte rate_index;
word post_offset;
word last_value;
byte scale_factor;
word min;
word max;
byte polling_cycle;
byte flags;             //bit 3=reverse axis;bit 4=disable antijitter
                        //
                        //bit 5 translate now bit. This will cause a value sent
                        // to a virtual analog to be translated when used,
                        // instead of waiting for analog processing schedule
                        // use when translating a gauge value from a pigeon hole
                        // and sending to the gauge in the pigeon hole handler
```

These elements are referenced within EPL as follows:

Device.Analog.type
Device.Analog.source
Device.Analog.max_delta
Device.Analog.pre_offset
Device.Analog.rate_index
Device.Analog.post_offset
Device.Analog.last_value
Device.Analog.scale_factor
Device.Analog.min
Device.Analog.max
Device.Analog.polling_cycle
Device.Analog.flags

Valid analog types are:

#define VIRTUAL 0xFF

#define ANALOG8 0

#define ANALOG10 1

#define ANALOG16 2

#define MEM8 3

#define MEM10 4

#define MEM16 5

#define MODROW8 6

#define MODROW10 7

#define MODROW16 8

As of build 0.1.1.37 only ANALOG8 and MEM8 are used.

Assigning a value to an analog channel defined as something other than a MEM8, MEM10, or MEM16 has no affect.

The source defines where the input comes from. For types MEM8, MEM10, and MEM16 the source is a memory address that will be calculated by the compiler. Changing this value from within an EPL program is not advisable and could cause the EPIC to crash, hang, or reboot.

Using Virtual Analogs

Defining

Before a virtual analog can be used it must be defined within a device descriptor. The complete syntax for defining a virtual analog is:

```
analog (    VIRTUAL,        // #defined constant, 0xFF
            name,           // name for this analog
            WinAxis,        // map analog to this Windows axis
            type=MEM8,      // analog type (default MEM8)
            source=0,       // source (default 0)
            min=0,          // min (default 0)
            max=255);       // max (default 255)
```

Parameters with default values can be omitted:

```
// the following lines are equivalent:
analog(VIRTUAL, Virtual, X);
analog(VIRTUAL, Virtual, X, MEM8);
analog(VIRTUAL, Virtual, X, MEM8, 0);
analog(VIRTUAL, Virtual, X, MEM8, 0, 0, 255);
analog(VIRTUAL, Virtual, X, MEM8, , , 255);
```