class: u3Base for C++ (g++) Linux

Started by Carl Friis-Hansen (c) 2006 (carl.friis-hansen@carl-fh.com). Revised 20091125

This class uses some functions from the Labjack company written for the original C function library called u3.h and u3.c.

The original functions are really very low level functions and not very friendly for larger applications. I chose to do it all in C++ as this makes the overview better and the C++ class objects offers better ways to document the functionality.
The hello test applications are good starting points, but bare in mind
that the test application expects a U3-USB with an RB12 termination board with the following modules mounted:
EIO0..EIO7 output relay, CIO0..CIO3 logic input.

Please find the youngest version at any time on:
http://computingconfidence.com/u3

Tested on harware:
U3 Firmware Version 1.170

U3 Bootloader 0.110
U3 Hardware 1.200
Upgraded with:
LJSelfUpgrade V1.09
U3firmware_117_08152006.hex
Driver Version:
labjackusb 2.01

All original development and test is done on Ubuntu Linux version 9.10.

Change log:

20060905 - Carl Friis-Hansen
Added logic table functions and structures to stimulate logic outputs
according to logic inputs. The stimulation happens at the end of each
call to checkAllInputs().

20060915 - Carl Friis-Hansen
Added delay timers for relays (logic outputs).
Added Hysteresis for analog inputs changing logic state.
Added U3 counter integration.
Corrected error in logEval concerning string operations strncpy(...)

20060923 – Carl Friis-Hansen
Added delay timers for inputs' logic state.
Added shared memory interface to serve independent external programs.

20061019 – Carl Friis-Hansen
Added SSW (Software Switch) functionality with additional implementation
in shared memory structure structSharemem { int ssw[10]; }.
The logic keywords are SSW0..SSW9 when used with logic equations.

20091125 – Carl Friis-Hansen
A Modified to use LabJack's new driver **labjackusb** which is installed as a C library
and interfaces to libusb-1.0-0 .

Index to methods

```
//
//  The default path to the LabJack U3 (1..n) where first dev will be 1
//
#define U3_DEVICE_PATH  1
//
//  The path/name for the shared memory ID data file
//
#define SHM_FILE_NAME   "u3shmID.txt"
//
//  1=Print errors to terminal - 0=Save last error (retrieve with getLastError())
//
#define PRINT_ERRORS    1
//
//  Structure for storing conversion table for an analog input
//
//  User need this structure in order to define a conversion table  for analog inputs.
//
typedef struct STRUCT_ANACON {
  double  vInput;          //  Input value
  double  vOutput;         //  Output value
} structAnaCon;
//
//  Structure holding data shared memory and acquired with checkAllInputs method
//  assuming that useSharedMemory method was previously called.
//
typedef struct share_mem    //  This structure can hold all input/output in shared memory
{
  int             newDataAvailable;       //  Parent sets this to 1 and any monitor can set it to 0
  structConfTable confTable[U3IOALEN+1];  //  All used inputs, sequential indexing
  structRelayOut  relayOut[U3IOALEN+1];   //  All outputs, absolute indexing
  int             ssw[10];                //  10 logic input software switches for boolean equations
} structSharemem;
```

class u3Base methods

```
  public:
    //  -------- Setup section --------

                          // Constructor
            u3Base(       int devicePath );    //  First U3 device has number 1.
```

Description

Constructor to initialize class. During initialization, connection to the U3 module is tested as a connection is opened.
In order for your program to detect success or failure, use the method getConnectID.
devicePath should be set to **1** for first LabJack device.
The default is defined in u3Base.hh under the name U3_DEVICE_PATH.

Example

```
u3Base  u3b( 1 );
if( u3b.getDeviceID() == NULL ) {
  return  2;       // Huston, we have a problem!
}
```

```
                          // Destructor
            ~u3Base(      void );
```

Description

Destructor to destroy class. When called, the connection to the U3 is closed.
This method is also called upon normal program termination.

// Verification of connection to hardware

```
HANDLE      getConnecID( void );              //  Returns NULL on connection error during construction
```

Description

Get the connection ID (handle) for the U3. On success, this method will return the handle different from NULL.

Example

```
u3Base  u3b( 1 );
if( u3b.getConnecID() == NULL ) {
  return  2;       // Error.
}
```

```
                        //  Write values to current state only. This is the most reliable way.
    int      writeConf(  char *FIOAnalog,       //  "11111111" 1=Analog 0=logic/digital
                         char *EIOAnalog,       //  "00000000" 1=Analog 0=logic/digital
                         char *DirFIO,          //  "00000000" 1=out 0=in (for logic/digital only)
                         char *DirEIO,          //  "00000111" 1=out 0=in (for logic/digital only)
                         char *DirCIO,          //      "0011" 1=out 0=in
                         const char *cntIO0,    //  ""=no counter0 else FIO0..EIO0
                         const char *cntIO1 ); //  0=no counter1 1=also counter1
```

Description

Configure the U3 device's channels. Thus determine what channels are analog or logic and the direction of these at run-time..
The method return 0 on success and non zero on failure.
FIOAnalog determines which channels are to be analog inputs. Data is to be given as a binary string where LSB is FIO0 and MSB is FIO7. A "1" means analog and a "0" means logic.
EIOAnalog determines which channels are to be analog inputs. Data is to be given as a binary string where LSB is EIO0 and MSB is EIO7. A "1" means analog and a "0" means logic.
DirFIO determines the direction of logic channels (input or output). Data is to be given as a binary string where LSB is FIO0 and MSB is FIO7. A "1" means out and a "0" means in.
DirEIO determines the direction of logic channels (input or output). Data is to be given as a binary string where LSB is EIO0 and MSB is EIO7. A "1" means out and a "0" means in.
DirCIO determines the direction of logic channels (input or output). Data is to be given as a binary string where LSB is CIO0 and MSB is CIO7. A "1" means out and a "0" means in.

Example

```
u3Base  u3b( 1 );
if( argc > 1 && strcmp( argv[1], "--config" ) == 0 ) {
  u3b.writeConf(        "11111111",   //  FIOAnalog all
                        "00000000",   //  EIOAnalog none
                        "00000000",   //  DirFIO don't care
                        "00000111",   //  DirEIO EIO2..EIO0 are logic out
                        "00000011" ,  //  DirCIO CIO1..CIO0 are logic out
                        "" , 0 );     //  No counters
}
```

```
                    //  Write default startup values to flash (EEPROM). Not working stateCIO!
int     writeConf(  char *FIOAnalog,      //  "11111111" 1=Analog 0=logic/digital
                    char *EIOAnalog,      //  "00000000" 1=Analog 0=logic/digital
                    char *DirFIO,         //  "00000000" 1=out 0=in (for logic/digital only)
                    char *DirEIO,         //  "00000111" 1=out 0=in (for logic/digital only)
                    char *DirCIO,         //  "00000011" 1=out 0=in
                    char *stateFIO,       //  "00000000" 1=hi/on 0=lo/off
                    char *stateEIO,       //  "00000000" 1=hi/on 0=lo/off
                    char *stateCIO,       //      "0000" 1=hi/on 0=lo/off
                    const char *cntIO0,   //  ""=no counter0 else FIO0..EIO0
                    const char *cntIO1 ); //  0=no counter1 1=also counter1
```

Description *** (***This method doesn't work well for stateCIO. Use the setRelayComplex method to set the state of these***) ***

Configure the U3 device's channels. Determine what channels are analog or logic and the direction of these at power-up.
This method does not need to be called if the device is already configured as the configuration is stored in EEPROM.
Also bare in mind that the device might **fail after 10.000 configurations**. The method returns 0 on success and non zero on failure.
See method above this one for: **FIOAnalog**, **EIOAnalog**, **DirFIO**, **DirEIO**, **DirCIO**.
stateFIO sets the hi/lo (on/off) state of logic (non-analog) outputs where “1” is hi (on) and “0” is lo (off).
The stateCIO seems to be flawed either in this class, the driver or the hardware. I would not put trust in the state configuration of CIOx at the moment and rather call setRelayComplex method after the buildInputCommand... sequence.

Example

```
u3Base  u3b( 1 );
if( argc > 1 && strcmp( argv[1], "--wrconfig" ) == 0 ) {
  u3b.writeConf(        "01111111",   //  FIOAnalog all except FIO7
                        "00000000",   //  EIOAnalog none
                        "00000000",   //  dirFIO don't care
                        "00000111",   //  dirEIO EIO2..EIO0 are logic out
                             "0011",  //  dirCIO CIO1..CIO0 are logic out
                        "00000000",   //  stateFIO don't care (nearly all are analog)
                        "00000111",   //  stateEIO EIO2..EIO0 are high
                            “0000”,   //  stateCIO don't care (not working for output)
                            “FIO7”,   //  Counter0 at channel FIO7
                            0 );      //  No counter1
```

```
}
```

```
                    // Starting point for building the functionality.
                    // The whole buildCommand..... section must be completed before
                    // any call is done to the main loop function checkAllInputs().
    void    buildInputCommandStart( void );
```

Description

Used together with buildInputCommandEnd, buildInputCommandAIN, buildInputCommandCIN and buildInputCommandDIN to configure individual inputs in preparation to a sampling loop.
The buildInputCommandANC and buildInputCommandHYS is supposed to come after buildInputCommandAIN of the channel it is intended for, but before buildInputCommandEnd.

Example

```
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIO0", 4.0, 9.0, 9.6, "V FIO0 Analog 0-10V DC – 9V batt. test" );
u3b.buildInputCommandAIN("FIO2", 1.00,  0.0, 0.0,  "V FIO2 Analog 0-2.44V DC");
u3b.buildInputCommandAIN("FIOT”, 1.00,  0.0, 0.0,  "C Ambient temperature");
u3b.buildInputCommandDIN("EIO3", "  EIO3 Logic input  24V DC" );
u3b.buildInputCommandDIN("CIO2", "  CIO2 Logic input 230V AC" );
u3b.buildInputCommandEnd();
```

```
                        //  Command and configure an analog channel.
                        //  Only one buildInputCommandAIN(...) per analog channel.
                        //  If neg channel=SE/31 then you can use the next function instead.
                        //  The limitLow and limitHigh are set points of output at which a
                        //  logic state is recognized and can be used by means of logicTableAdd(...)
                        //  to set state of logic outputs.
    int     buildInputCommandAIN( char    *sFECio,    //  FIO0..7, FIOT, EIO0..7
                                  int     nch,        //  If neg channel
                                  double  calibrat,   //  deviation from 2.44V (default 1.0)
                                  double  limitLow,   //  Value where "FILx" or "EILx" becomes hi
                                  double  limitHigh,  //  Value where "FIHx" or "EIHx" becomes hi
                                  char    *remarks    //  Limited to 256 characters
                                );
```

Description

Used together with buildInputCommandStart and buildInputCommandEnd to configure individual inputs in preparation to a sampling loop. This method handles the configuration of one analog input.

sFECio	channel name “FIO0..FIO7”, “FIOT”, “EIO0..EIO7”, “CIO0..CIO3” to be configured.
nch	determines which channel to use for the negative part of the analog input (reference point). The default is 31 (Signal Earth).
calibrat	Analog input with SE reference is nominal 2.5V. Thus, to produce 0 to 10V return, the calibrat should be 10 / 2.5 = 4.0
limitLow	When voltage comes under this limit the logic level XILx is becomes hi, otherwise lo. See method logicTableAdd.
limitHigh	When voltage comes over this limit the logic level XIHx is becomes hi, otherwise lo. See method logicTableAdd..
remarks	Free text for later display usage.

Example

```
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIO0", 31, 4.0, 9.0, 9.6, "V FIO0 Analog 0-10V DC – 9V batt. test" );
u3b.buildInputCommandAIN("FIO2", 31, 1.00,  0.0, 0.0,  "V FIO2 Analog 0-2.44V DC");
u3b.buildInputCommandAIN("FIOT”, 31, 1.00,  0.0, 0.0,  "C Ambient temperature");
u3b.buildInputCommandEnd();
```

```
                        // Command and configure an analog channel.
                        // The version of buildInputCommand assumes negChannel is SignalEarth (31).
                        // Only one buildInputCommandAIN(...) per analog channel.
                        // The limitLow and limitHigh are set points of output at which a
                        // logic state is recognized and can be used by means of logicTableAdd(...)
                        // to set state of logic outputs.
int     buildInputCommandAIN( char    *sFECio,    //  FIO0..7, FIOT, EIO0..7
                              double  calibrat,   //  deviation from 2.44V (default 1.0)
                              double  limitLow,   //  Value where "FILx" or "EILx" becomes hi
                              double  limitHigh,  //  Value where "FIHx" or "EIHx" becomes hi
                              char    *remarks    //  Limited to 256 characters
                            );
```

Description

Used together with buildInputCommandStart and buildInputCommandEnd to configure individual inputs in preparation to a sampling loop.
This method handles the configuration of one analog input.

sFECio	channel name “FIO0..FIO7”, “FIOT”, “EIO0..EIO7”, “CIO0..CIO3” to be configured.
calibrat	Analog input with SE reference is nominal 2.5V. Thus, to produce 0 to 10V return, the calibrat should be 10 / 2.5 = 4.0
limitLow	When voltage comes under this limit the logic level XILx is becomes hi, othervise lo. See method logicTableAdd.
limitHigh	When voltage comes over this limit the logic level XIHx is becomes hi, othervise lo. See method logicTableAdd..
remarks	Free text for later display usage.

Example

```
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIO0", 4.0, 9.0, 9.6, "V FIO0 Analog 0-10V DC – 9V batt. test" );
u3b.buildInputCommandAIN("FIO2", 1.00,  0.0, 0.0,  "V FIO2 Analog 0-2.44V DC");
u3b.buildInputCommandAIN("FIOT”, 1.00,  0.0, 0.0,  "C Ambient temperature");
u3b.buildInputCommandDIN("EIO3", "  EIO3 Logic input  24V DC" );
u3b.buildInputCommandDIN("CIO2", "  CIO2 Logic input 230V AC" );
u3b.buildInputCommandEnd();
```

```
                        //  Command to include a user conversion table.
                        //  The table will do conversion on final calibrated output value and
                        //  before any call to logicTable or other methods that might depend
                        //  on the analog output value.
    int     buildInputCommandANC( char        *sFECio,    //  FIO0..7, FIOT, EIO0..7
                                  structAnaCon  *anaCon,   //  Conversion table
                                  int     anaConLen       //  Size of table (number of entries)
                                );
```

Description

Used after buildInputCommandAIN for the channel to have conversion table implemented.
The conversion table must use the structure structAnaCon defined in u3Base.hh. An array with this structure is to be defined in the user application.

sFECio	channel name “FIO0..FIO7” or “EIO0..EIO7” to be configured.
anaCon	Pointer to first record in conversion table.
anaConLen	Number of entries in conversion table.

Example

```
 //  Table for conversion of germanium diode voltage to degrees Celsius
 //  This is not calibrated at all, but gives a good idea of how to use
 //  the analog conversion table.
 structAnaCon  anaConGDiode[] =
 {
   { 0.20,  60.0 },
   { 0.28,  20.0 },
   { 0.36, -20.0 },
 };
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIO2", 1.00,  -19.9, 59.9,  "C FIO2 temperature measured with germanium diode @ 1mA");
u3b.buildInputCommandANC("FIO2", &anaConGDiode[0], 3 );
```

```
                        // Command to implement hysteresis in connection with change og logic
                        // state between limitLow to mid range and limitHigh to mid range.
                        // The hysteresis is an absolute value proportional to the output.
int     buildInputCommandHYS( char        *sFECio,    //  FIO0..7, FIOT, EIO0..7
                              double      hysteresis  //  Hysteresis for changing logState
                            );
```

Description

Used after buildInputCommandAIN for the channel to have hysteresis in logic states implemented.
The hysteresis takes effect when an analog level goes from below limitLow to above limitLow and
when going from above limitHigh to below limitHigh. Set in other words: The hysteresis is the extra value needed
to change the logic state into limitMid.

Example

Let's say that the internal temperature sensor FIOT is defined as input, the limitLow is 20 degrees,
the limitHigh is 30 degrees and the hysteresis is 2 degrees. When FIOT goes above 30 degrees, then
FIHT becomes true. After that the temperature has to fall below 28 degrees before FIHT becomes false.
When FIOT goes below 20 degrees, then FILT becomes true. After that the temperature has to rise
above 22 degrees before FILT becomes false:

buildInputCommandAIN(“FIOT”, 1.0, 20.0, 30.0, “Internal temperature”);
buildInputCommandHYS(“FIOT”, 2.0);

```
                        // Set the time in seconds it take for an input to go from
                        // hi-->lo and from lo-->hi or if analog hi/lo-->mid and mid-->hi/lo
                        // This is not applicable to Counter Inputs
    void    buildInputCommandDelay(const char *sFECio,      //  FIO0..7, EIO0..7, CIO0..3
                                   time_t      offDelay,    //  Propagation delay time in seconds (or off alarm)
                                   time_t      onDelay );   //  Propagation delay time in seconds (or to alarm)
```

Description

Change of logic state for either a logic input or an analog input can be delayed using this method.
The delay is in seconds. The offDelay determines the time it takes an input to acknowledge lo state
(mid state if analog). The input has to be logic stable in in the delay period or the timing will be reset.
The same goes for going from lo (mid) to hi (!mid) using the onDelay timing.
This feature is not to be interpreted as propagation delay time.
Any logic equations reacts on the state after the delay has timed out.

Example

Let's take a scenario where a fire sprinkler must start (EIO1) when the room temperature becomes too high (FIH1) for more than 20sec and go off (!EIO1) when temperature goes below high (!FIH1) for more than 240sec or 4min. 0..2V ~ 0..200degrees and max allowed temperature is 70degrees.
Then some of the setup instructions would be:

```
u3Base u3b( 1 );
.....
u3b.buildInputCommandAIN(   ”FIO1”, 100.0, 0.0, 70.0, ”FIO1 - Temperature sensor” );
u3b.buildInputCommandDelay( ”FIO1”, 240, 20 );
.....
u3b.setRelayText( ”EIO1”, ”EIO1 – Sprinkler closed”, ”EIO1 – Sprinkler active” );
.....
u3b.logicTableAdd( ”EIO1”, ”FIH1” );
u3b.setRelayComplex( ”!EIO1” );
```

```
                        // Command and configure a counter channel.
                        // Only one buildInputCommandCIN(...) per counter channel.
int     buildInputCommandCIN( char         *sFECio,    //  FIO0..7, FIOT, EIO0..7
                              int          cntNum,     //  0=counter0 1=counter1
                              unsigned long cntLimit,  //  Set logState at this point
                              const char   *remarks    //  Limited to 256 characters
                            );
```

Description

Two counter are available on the U3. Counter0 can be located on any of the channels FIO0..7, EIO0..1. Counter1 must be placed on a location right after counter0. The location must have been set in one of the wrConfig methods.
The counters react on logic signal going from high to low state and can count to 2^32 (a very large number indeed). Mechanical switches should not be used directly, but rather through an adequate circuit or through an XOR function (see addLogicTable(...)).
When the counter has reached cntLimit, then the logic state changed from 0 to 1. Let's say that you have chosen to locate counter0 at channel FIO7, then the logic state is !FIO7 before cntLimit and FIO7 after cntLimit has been reached. The method resetCounter can be used to reset the counter or the output RCT0/RCT1 can be set hi in logic equations (see example below).

Example

If we have a machine to package 6 items in every box: We count the items with an XOR function using EIO6 and EIO7.
We advance to a new empty package by sending a 4 second signal with CIO0. FIO3 is configured as logic output and used as output from the XOR and connected with a wire to the counter input FIO7:

```
buildInputCommandStart();
buildInputCommandDIN( "EIO6", "EIO6 logic input" );
buildInputCommandDIN( "EIO7", "EIO7 logic input" );
buildInputCommandCIN( "FIO7", 0, 6, "Count 6 things on FIO7 using counter0" );
buildInputCommandEnd();
setRelayText( "CIO0", "CIO0 Package filling", "CIO0 Package filled with items (throw 4sec)" );
setRelayDelay( "CIO0",  4, 0 );   //  Four second delay from hi to lo
logicTableAdd( "CIO0", "FIO7" );   //  Package filled with items
logicTableAdd( "FIO3", "EIO6 * !EIO7 + ( !EIO6 * !EIO7 + EIO6 * EIO7 ) * FIO3" );  //  XOR function
logicTableAdd( "RCT0", "CIO0" );  //  Reset counter after box is filled so we can start over.
```

```
                        // Command and configure a logic/digital channel.
                        // Only one buildInputCommandDIN(...) per logic channel.
    int     buildInputCommandDIN( char    *sFECio,    //  FIO0..7, EIO0..7, CIO0..3
                                  char    *remarks    //  Limited to 256 characters
                                );
```

Description

Used together with buildInputCommandStart and buildInputCommandEnd to configure individual inputs in preparation to a sampling loop.
This method handles the configuration of one logic input.

sFECio channel name “FIO0..FIO7”, “FIOT”, “EIO0..EIO7”, “CIO0..CIO3” to be configured.

remarks Free text for later display usage.

Example

```
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIOT”, 1.00,  0.0, 0.0, "C Ambient temperature");
u3b.buildInputCommandDIN("EIO3", "  EIO3 Logic input  24V DC" );
u3b.buildInputCommandDIN("CIO2", "  CIO2 Logic input 230V AC" );
u3b.buildInputCommandEnd();
```

// Call this to conclude and activate the command sets.

```
void    buildInputCommandEnd( void );              //  Must conclude the build commands
```

Description

Used together with buildInputCommandStart, buildInputCommandAIN, buildInputCommandCIN and buildInputCommandDIN to configure individual inputs in preparation to a sampling loop.
The buildInputCommandANC and buildInputCommandHYS is supposed to come after buildInputCommandAIN of the channel it is intended for, but before buildInputCommandEnd.

Example

```
u3Base  u3b( 1 );
u3b.buildInputCommandStart();
u3b.buildInputCommandAIN("FIOT”, 1.00,  0.0, 0.0,  0, 0, "C Ambient temperature");
u3b.buildInputCommandDIN("EIO3", "  EIO3 Logic input  24V DC" );
u3b.buildInputCommandDIN("CIO2", "  CIO2 Logic input 230V AC" );
u3b.buildInputCommandEnd();
```

```
                        // Use this function to automatically set/reset logic outputs each time
                        // checkAllInputs() is called. Only one call per output or output hazard
                        // would occur (there is a build in check for this though).
                        // The condition/equation string sFECi must be less that 1024 characters and
                        // have at least one parameter.
                        // The evaluation of sFECi is done with '*' operators first and '+' operators last
                        // unless '(' and ')' are used. All parameters MUST be separated by one space ' '
                        // like: logicTableAdd("CIO1","( !EIO6 + FIL5 ) * FIO3")
int     logicTableAdd(      char    *sFECo,     //  FIO0..7, EIO0..3 (Logic digital output)
                                                //  In addition: RCT0..1 for counter reset.
                            char    *sFECi);    //  FIL0..7, FILT, EIL0..7,   (logic analog Low input)
                                                //  FIM0..7, FIMT, EIM0..7,   (logic analog Mid input)
                                                //  FIH0..7, FIHT, EIH0..7,   (logic analog High input)
                                                //  FIO0..7, EIO0..7, CIO0..7,(logic digital input/output)
                                                //  *, +, (, ), !             (logic operators)
```

Description:

With this method one can avoid a lot of complicated coding and thus minimizing the chances of fatal errors. The method is attempting to follow standard boolean algebra and should therefore be pretty familiar to the technical user. As this protocol describes, an output is ONLY hi/on/true/1 if the boolean equation evaluates to true. There is a shortcut when it comes to the logic evaluation of the analog inputs. If we for example have the definition buildInputCommandAIN(“FIOT”, 1.0, 18.0, 29.0, “Internal temperature”), then the “FILT”, “FIMT” and “FIHT” logic flags are affected like this:

“FILT” is true below 18.0
“FIMT” is true above 18.0 but below 29.0
“FIHT” is true above 29.0

Thus the equation “!FILT * !FIHT” gives the same result as “FIMT”.

The use of brackets is limited to a depths of one. However, most (or really all) expressions can be reduced mathematically to satisfy this limitation.

The issue of interlocking (making sure that two outputs doesn't go hi at the same time can be solved, in a scenario where CIO0 and CIO1 are the two outputs and FIH0 and FIH1 are the inputs, like this:
logTableAdd(“CIO0”, “FIH0 * !CIO1”);
logTableAdd(“CIO1”, “FIH1 * !CIO0”);
However, if the involved outputs have been manipulated directly and not through methods in u3Base, the actual state of CIO0 and CIO1 might not be known. It is further a very good idea to initialize all logic outputs to a known state by means of setRelayComplex. In our scenario the interlock will set both output lo/false between two calls to chackAllInputs in the case FIH0 goes hi at the same time as FIH1 goes lo and visa versa. This means that if the method chackAllInputs is called once every second, the off-before-on time is one second.

Below is an example of how to half the toggling frequency of three relays (3 bit binary counter):

```
logicTableAdd( "EIO2", "!EIO2"  );                                  //  Toggle relay EIO2 watchdog
logicTableAdd( "EIO1", "!EIO1 * EIO2 + EIO1 * !EIO2"  );            //  Toggle half frequency of EIO2
logicTableAdd( "EIO0", "!EIO0 * EIO1 * EIO2 + EIO0 * !EIO1 + EIO0 * !EIO2" );     //  Toggle half frequency of EIO1
```

Yes, I know it has no practical use. It is really just to show the importance of checking every condition at which the output is allowed to toggle and the conditions at which it is not allowed to leave it's hi state.
For complicated logic designs, it might be a help to utilize Carnot cards to build the equations.

Below is an example of an XOR assuming FIO3 is output and EIO6, EIO7 are two inputs:

```
logicTableAdd( "FIO3", "EIO6 * !EIO7 + ( !EIO6 * !EIO7 + EIO6 * EIO7 ) * FIO3" );
```

The state of FIO3 will only change state when one input is hi and the other lo and vise versa.
In this way one can avoid ringing from a mechanical switch.

```
                        // Associate a text with the hi and lo state of a logic output
                        // After checkAllInputs() the current state can be read as text with
                        // the method getRealayText(...)
    void    setRelayText(   const char  *sFECio,      //  FIO0..7, EIO0..7, CIO0..3
                            char        *offText,     //  Like: "The relay is OFF"
                            char        *onText );    //  Like: "The relay is ON"
```

Description

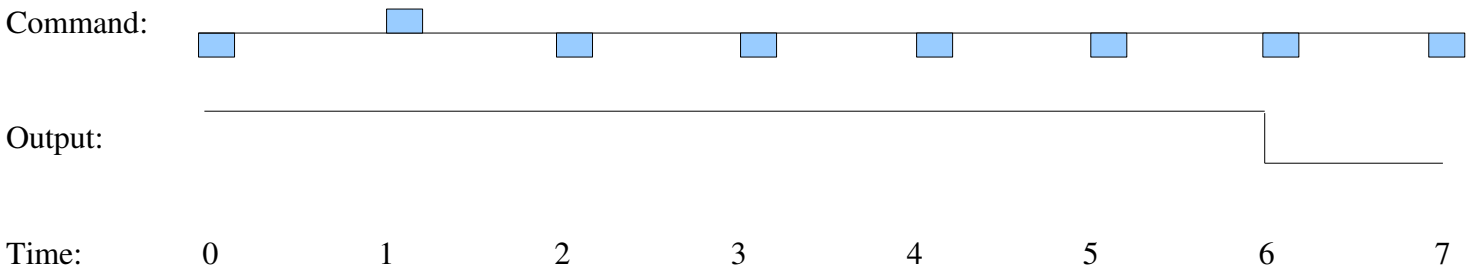
Optionally apply off and on text to a relay/logic output. During execution the appropriate text can then be retrieved with getRelayText for printing purpose. This text can be more meaningful than the result of getRelayState.

```
                    // Set the time in second it take for a logic output to go from
                    // hi-->lo and from lo-->hi
void    setRelayDelay(  const char  *sFECio,      //  FIO0..7, EIO0..7, CIO0..3
                        time_t      offDelay,     //  Propagation delay time in seconds
                        time_t      onDelay );    //  Propagation delay time in seconds
```

Description

When a logic output is ordered to lo and the current state is hi, a timer of offDelay seconds is initiated, and the output is left in it's current state. Next time the same order is issued and if the timer has run out, the logic state is changed and the timer is reset. If the output is ordered to go to the same state as it already is, then all timers for this output are reset.
When a logic output is ordered to go hi and the current state is lo, a timer of onDelay second is initiated and all the rest is the same as for the hi-->lo scenario.

Below is a schematic of delayed hi-->lo of 4 seconds:



In the schematic it is shown that conflicting command to go hi caused a reset of the 4 second timer. In this example the interval of the commands is one second, but a shorter command interval will not change the timing, it will rather make it more precise.

```
                        // This function sets up a command in the U3 device to acquire fast reads.
                        // At this time the streaming functions are not completely implemented and
                        // should therefore not be used!
int     streamConfig(       int numChannels,        //
                            int samplesPerPacket,   //  1..16
                            int scanInterval );     //  1..65535
```

```
//  -------- Run section --------

                        //  Each time checkAllInputs() is called the inputs, defined with the
                        //  buildInputCommand... methods, are read from the U3 device, evaluated
                        //  and acted upon. The new data are available through various methods
                        //  described below.
int     checkAllInputs( void );
```

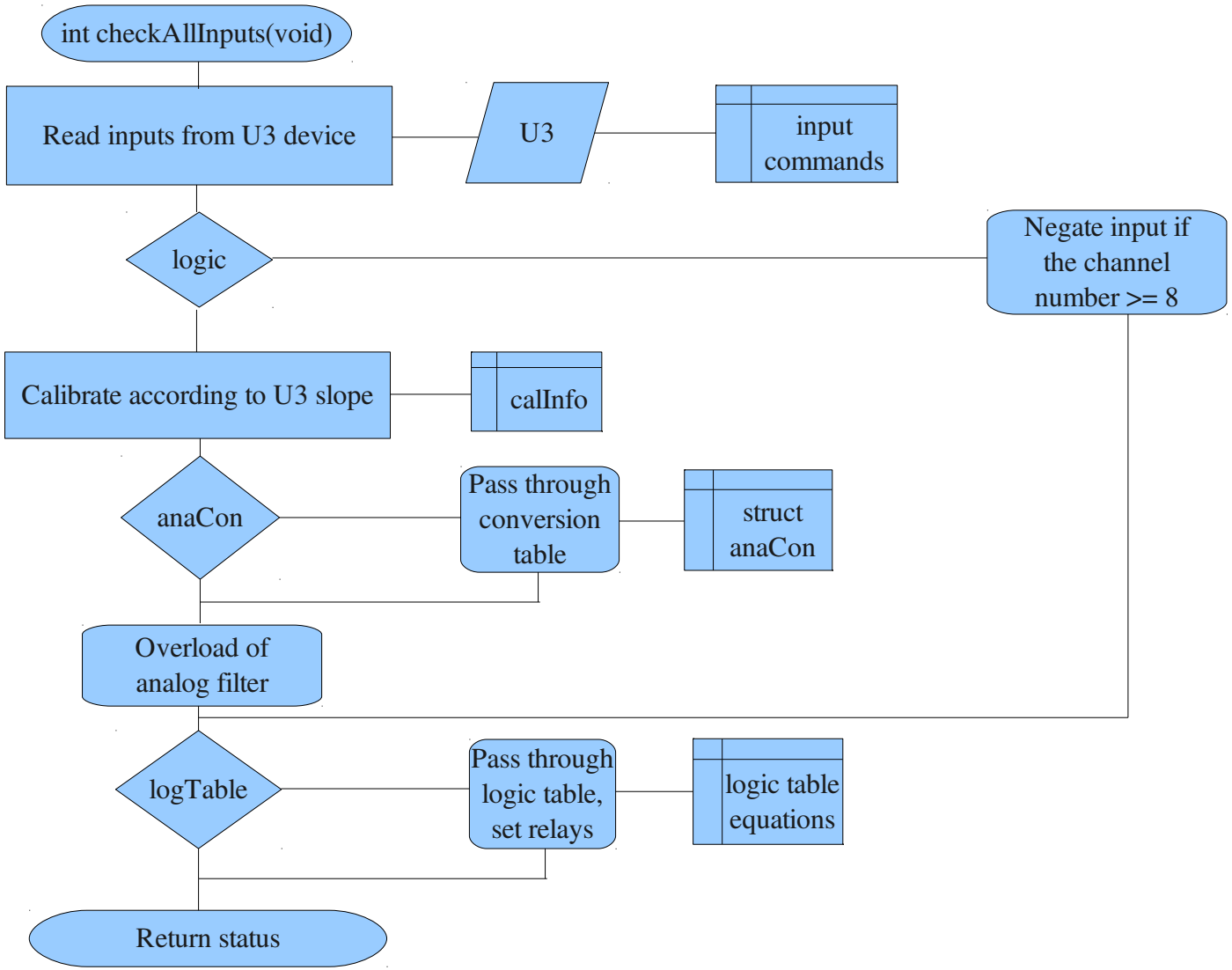
Description

It is the thought that this method should be called repeatedly for example at an interval of one second.
In multi threaded applications it is important to wait, until the return value of the method is received,
before any other methods in u3Base or any contact to the U3 device are made.
During the execution of checkAllInputs all inputs, defined by the buildInputCommand... methods, are
read and modified. After this, eventual conversion tables are processed and finally the boolean expressions
from any logTableAdd definitions is processed. The process of logTables results in logic outputs being
stimulated in the same sequence as they are defined with the method logTableAdd.

One of the big questions is how often one would call checkAllInputs? If the LabJack U3 is used for surveillance,
regulation of environment and processes of a fairly slow nature, then a cal every second would probably do very fine.
However, if a feeding machine, a roller coaster or other machinery that needs fast reaction, then a call every 20msec or
less is likely to be needed. Below is an example where method
checkAllInputs is called every 125msec and printing to terminal is done every 2sec:

```
while( !kbhit() ) {
   count++;
   u3b.checkAllInputs();       // every 1/8 second
   if( (count & 0x0F) == 0 ) {
      // various printing to terminal or other process every 2 second
   }
   usleep( 125 * 1000 ); //  1/8 sec
}
```

Functional draft of the checkAllInputs:



// Number of items in confTable. The items were defined with buildInputCommand..

```
int      getNumConfTable( void );
```

Description

Use the return value from this method in for example for(...) loops as the amount of defined inputs.

Example

```
u3Base u3b( 1 );
int n;
.....
 for( n=0; n<u3b.getNumConfTable(); n++ ) {
  u3b.getConfTable( n,        //  0..19, 31 Input
                    &table ); //  User variable to retrieve the lot
  switch( table.typeAD ) {
    case 'A':
      printf( "%6.2f %s\n", table.voltage, table.remarks );
      break;
    case 'D':
      printf( "%4d %s\n", table.logState, table.remarks );
      break;
    case 'C':
      printf( "%4d %6d %s\n", table.logState, (int)table.counter, table.remarks );
      break;
  }  //  switch(...)
 }  //  for(...)
```

```
                        // As all data are received they are put into a structured array in sequence.
                        // You can retrieve the type of channel that corresponds to this sequence number.
    char      getConfTable(      int inputNumber,            //  0..getNumConfTable()-1
                                 structConfTable *table ); //  User variable to retrieve the lot
```

Description

All aspects of an input can be retrieved in a record with a call to getConfTable.
Please see the u3Base.hh header file for available fields in the structConfTable.

Example

```
u3Base  u3b( 1 );
structConfTable table;
int         n;
.....
u3b.checkAllInputs();
for( n = 0; n < u3b.getNumConfTable(); n++ ) {
  u3b.getConfTable( n, &table );
  switch( table.typeAD ) {
    case 'A':
      printf( "%6.2f %s\n", table.voltage, table.remarks );
      break;
    case 'D':
      printf( "%4d %s\n", table.logState, table.remarks );
      break;
    case 'C':
      printf( "End=%4d  count=%6d  remarks=%s\n", table.logState, (int)table.counter, table.remarks );
      break;
  }  //  switch(...)
}  //  for(...)
......
```

```
                        // Get all info for an output in a structRelayOut variable
int     getRelayOut(    int         outputNumber, //  0..U3IOALEN
                        structRelayOut *table );  //  User variable to retrieve the lot
```

Description

All aspects of a logic output/relay can be retrieved in a record with a call to getRelayOut.
Please see the u3Base.hh header file for available fields in the structRelayOut.

Example

```
u3Base  u3b( 1 );
.....
structRelayOut  rtable;
.....
u3b.getRelayOut( "EIO4", &rtable );
printf( "%4d %s\n", rtable.logState, rtable.logState ? rtable.onText : rtable.offText );
printf( “Off time is set to %d and on time is set to %d\n”, rtable.offDelay, rtable.onDelay );
```

```
                    // As all data are received they are put into a structured array in sequence.
                    // You can retrieve the type of channel that corresponds to this sequence number.
    char    getInputType(   int inputNumber );         //  0..getNumConfTable()-1
```

Description

In a print or data acquisition loop it is important to know the type of data one is retrieving.
The getInputType can return the following type identifiers:
'A' Analog input.
'D' Logic/digital input.
'C' Counter input.
The parameter inputNumber can be any of 0 to getNumConfTable().

Example

```
u3Base  u3b( 1 );
.....
u3b.checkAllInputs();
printf( “Last input is of type: %c \n”, u3b.getInputType( u3b.getNumConfTable()-1 ) );
```

```
                    // As all data are received they are put into a structured array in sequence.
                    // You can retrieve the channel name that corresponds to this sequence number.
    char*   getInputFECio(  int inputNumber );         //  0..getNumConfTable()-1
```

Description

This method has much the same properties as getInputType, the main difference is that instead of returning the type, it is returning the name of the input channel. Like ”FIO7”.

```
                    // Get actual logic state as text set by setRelayText(...)
char*   getRelayText(   const char  *sFECio );    //  FIO0..7, EIO0..7, CIO0..3 Logic text
```

Description

Returns the text assigned using method setRelayText. That is, it only returns the text that corresponds to the actual state of the logic output/relay.

```
                    // Get actual logic state where 1=hi/true/on and 0=lo/false/off
                    // Please note that this info depends on outputs only being modified by u3Base methods
int     getRelayState(  const char  *sFECio );    //  FIO0..7, EIO0..7, CIO0..3 Logic output
```

Description

Returns the logic state of output/relay as 1 for hi/on or 0 for lo/off.

```
                    // Get actual logic state where 1=hi/true/on and 0=lo/false/off
                    // Please note that this info depends on the last call to checkAllInputs
int     getInputState(  int         inputNumber );//  0..getNumConfTable()-1    Logic input

                    // Get actual logic state where 1=hi/true/on and 0=lo/false/off
                    // Please note that this info depends on the last call to checkAllInputs
int     getInputState(  const char  *sFECio );    //  FIO0..7, EIO0..7, CIO0..3 Logic input

                    // Get actual value of analog input
                    // Please note that this info depends on the last call to checkAllInputs
double  getInputVoltage(int         inputNumber );//  getNumConfTable()-1      Analog input

                    // Get actual value of analog input
double  getInputVoltage(const char  *sFECio );    //  FIO0..7, FIOT, EIO0..7   Analog input

                    // Get text associated with analog input
char*   getInputText(   int         inputNumber );//  0..getNumConfTable()-1          Input

                    // Get text associated with analog input
char*   getInputText(   const char  *sFECio );    //  FIO0..7, FIOT, EIO0..7, CIO0..3 Input

                    // Get value of counter
long    getInputCount(  const char  *sFECio );    //  FIO0..7, EIO0..1         Counter input

                    // Get sotfware input switch state
int     getSSW(         const char  *sFECio );    //  SSW0..9
```

Description

Returns the logic state of **s**oftware input **sw**itch as 1 for hi/on or 0 for lo/off. There are 10 such switches available. They are useful in boolean equations as an additional manual control through main program or a shared memory program. Simply if FIO1 was a digital/logic output then logicTableAdd(”FIO1”,”SSW3”), FIO1 could be toggled by the following: if(kbhit()=='3') if(getSSW(”SSW3”)) setSSW(”!SSW3”) else setSSW(”SSW3”);

```
                        // Set logic output to hi (1) or lo (0)
int     setRelay(       int         outputNumber, //  Channel 0..19
                        int         state );      //  0=Off, 1=On

                        // Set logic output to hi (1) or lo (0)
int     setRelay(       const char  *sFECio,      //  Channel FIO0..7, EIO0..7, CIO0..3
                        int         state );      //  0=Off, 1=On

                        // Set any number of logic outputs to hi (1) or lo (0)
int     setRelayComplex(char        *action );    //  Channel FIO0..7, EIO0..7, CIO0..3
                                                  //  Many actions separated with space (' ').
                                                  //  Use '!' in front of channel name to reset
                                                  //  instead of set.
```

Description

It is mandatory to set all utilized logic outputs to a known state.
The most efficient way to do this, is to use the setRelayComplex although the setRelay method can be used as well.
Please note that there must be one space and one space only between parameters in setRelayComplex.

Example

```
u3Base u3b( 1 );
if( u3b.setRelayComplex( "!EIO0 !EIO1 EIO2" ) == -1 ) {
  printf( "Error: failure to set one or more logic output states.\n" );
}
```

```
                        // Set/Reset sotfware input switch
                        // The initial state is !SSWn or in other words: off/lo/0
int     setSSW(         const char  *sFECio );    //  SSW0..9 (hi), !SSW0..9 (lo)
```

Description

Returns the logic state of **s**oftware input **sw**itch as 1 for hi/on or 0 for lo/off. There are 10 such switches available. They are useful in boolean equations as an additional manual control through main program or a shared memory program. Simply if FIO1 was a digital/logic output then logicTableAdd("FIO1","SSW3"), FIO1 could be toggled by the following:
if(kbhit()=='3') if(getSSW("SSW3")) setSSW("!SSW3") else setSSW("SSW3");

```
                              //  Reset counter0 or counter1
    int      resetCounter(    int          cntNum );      //  0=counter0 1=counter1
```

Description

If counter cntNum is defined using buildInputCommandCNT, then the counter can be reset using this method.
Alternatively a counter can be reset using ”RCT0” or ”RCT1” in the result of a logic equation.
This latter method is the most useful, please see the example.

Example

Let's say counter0 located at FIO3, then it's reset command name would be RCT0. In case of counter1 it would be RCT1. If we would like to reset the counter0 when logic input CIO1 goes lo/off/false and analog input FIO1 goes above 1.8V and counter if full (500), then the following lines could be part of the program:

```
u3Base u3b( 1 );
u3b.buildInputCommandAIN( ”FIO1”, 1.0, 0.0, 1.8, ”FIO1 Analog input” );
u3b.buildInputCommandDIN( ”CIO1”, ”Enable counter reset”, ”Block counter reset” );
u3b.buildInputCommandCNT( ”FIO3”, 0, 500, ”Counter0 is full by a count of 500” );
u3b.logicTableAdd( ”RCT0”, ”!CIO1 * FIH3 * FIO3” );
```

```
                    // Set green LED on main board on (1) or off (0). Careful it is using a relay
int     setLED(         int         state );      //  LED on: state=1 off: state=0

                    // Activate the internal buzzer on main board (not fully tested)
int     buzz(           int         continuously, //  1=cont 0=one_time
                        int         period,       //  duration
                        int         toggles );    //  number of periods if continuously

                    // Set the voltage of analog of DAC0 or DAC1
int     setDAC(         char        *sDAC,        //  Channel DAC0..1
                        double      value );      //  0..2.44V
```

```
                    // At this time the streaming functions are not completely implemented and
                    // should therefore not be used!
int     streamStart(    void );                 //  Start streaming (must be set with streamConfig(...) )

                    // At this time the streaming functions are not completely implemented and
                    // should therefore not be used!
int     streamStop(     void );                 //  Start streaming (must be started with streamStart() )

                    // At this time the streaming functions are not completely implemented and
                    // should therefore not be used!
int     streamData(     void );                 //  Start streaming (must be started with streamStart() )
```

```
                    // After a call to this method, other programs can read incoming data via shred
                    // memory. The ID of the shared memory is written to a file named SHM_FILE_NAME.
                    // The data in shared memory is structured according to the structSharemem struct.
                    // It is up to the connecting program to attach to the shared memory and to
                    // remember to deattach from it again before exit.
int     useSharedMemory(void );                    //  Method returns 0 if OK and -1 on error.
```

Description

On Linux/Unix and probably Mac OS/X programs can write to each other via shared memory. In the u3Base class we utilize this as a way of having totally independent programs picking up data acquired by the checkAllInputs method in u3Base. The structSharemem from u3Base.hh is used to define the structure in the shared memory. Programs connecting to this shared memory will have to look for a file named SHM_FILE_NAME ("u3shmID.txt") created in the same directory as the program running the U3 hardware (using u3Base). This file contains the shared memory ID (shmid) in character format. Below is an example:

```
 if( ( shmIDFile = fopen( SHM_FILE_NAME, "r" ) ) != NULL ) { //  SHM_FILE_NAME is defined in u3Base.hh
   fgets( charTemp, 16, shmIDFile );
   fclose( shmIDFile );
   shmid = atoi( charTemp );
   if( (shareptr = (structSharemem *) shmat( shmid, 0, 0) ) == (structSharemem *) -1 ) {
     perror("can't attach to shared memory\n");
   }
   printf( "Using shared memory %d\n", shmid );
 } else {
   shmid = 0;
   printf( "Error opening file %s in order to acquire shared memory ID\n", SHM_FILE_NAME );
 }
```

Data can be retrieved like this:

```
 shareptr->confTable[0].logState
```

Remember to use shmdt(shareptr) in order to detach before exit.

The newDataAvailable in the structSharemem structure is set to 1 every time checkAllInputs delivers data.

```
                        // Return last error text.
                        // Use this if a method returns -1
    char*   getLastError( void );
```

Description

A compiler flag #define PRINT_ERRORS (see u3Base.hh) determines weather or not to print error messages to the terminal.
However, the last error message can always be retrieved using getLastError().
By default error messages are printed:
#define PRINT_ERRORS 1

```
//  -------- General public utility functions --------

                    // Convert from channel name to channel number
int     FECioToNum(     const char  *sFECio );    //  FIO0..7, FIOT, EIO0..7, CIO0..3 ==> 0..19

                    // Convert from binary to integer
int     binToInt(       char        *s );         //  Convert text based binary to int ("0101"=>5)

                    // Convert from integer to binary
char*   intToBin(       int         i,            //  Convert int to text based binary (5=>"00000101")
                        int         isWord );     //  0: 8bit, 1: 16bit


//  -------- Some development public utility functions --------

void    printU3configuration( void );             //  Prints sendBuff contents thus describing configuration

void    printSendBuff2( int         nmax );       //  For test only

                    // Convert U3 error codes into the corresponding text
                    // as described in the LabJack U3 pdf document
char*   getErrorText(   int         code );       //  Convert error code to error text.
```

Keywords in u3Base

To ease the use of this class, all channels and logic ports use reserved strings that compare with the labeling and documentation provided by LabJack.

Reserved strings related to the U3 hardware:

```
Key   Description
----  ------------------------------------------------------
FIO0  Channel  0 on main board r/w/p
FIO1  Channel  1 on main board r/w/p
FIO2  Channel  2 on main board r/w/p
FIO3  Channel  3 on main board r/w/p
FIO4  Channel  4 on main board r/w/p
FIO5  Channel  5 on main board r/w/p
FIO6  Channel  6 on main board r/w/p
FIO7  Channel  7 on main board r/w/p
FIOT  Internal temperature sensor on main board r/o/p
EIO0  Channel  8 on U12 termination board r/w/p
EIO1  Channel  9 on U12 termination board r/w/p
EIO2  Channel 10 on U12 termination board r/w/p
EIO3  Channel 11 on U12 termination board r/w/p
EIO4  Channel 12 on U12 termination board r/w/p
EIO5  Channel 13 on U12 termination board r/w/p
EIO6  Channel 14 on U12 termination board r/w/p
EIO7  Channel 15 on U12 termination board r/w/p
CIO0  Channel 16 on U12 termination board (logic only) r/w/p
CIO1  Channel 17 on U12 termination board (logic only) r/w/p
CIO2  Channel 18 on U12 termination board (logic only) r/w/p
CIO3  Channel 19 on U12 termination board (logic only) r/w/p
FIL0  Analog input channel  0 low value alarm/logic state r/o
FIL1  Analog input channel  1 low value alarm/logic state r/o
FIL2  Analog input channel  2 low value alarm/logic state r/o
FIL3  Analog input channel  3 low value alarm/logic state r/o
FIL4  Analog input channel  4 low value alarm/logic state r/o
FIL5  Analog input channel  5 low value alarm/logic state r/o
FIL6  Analog input channel  6 low value alarm/logic state r/o
```

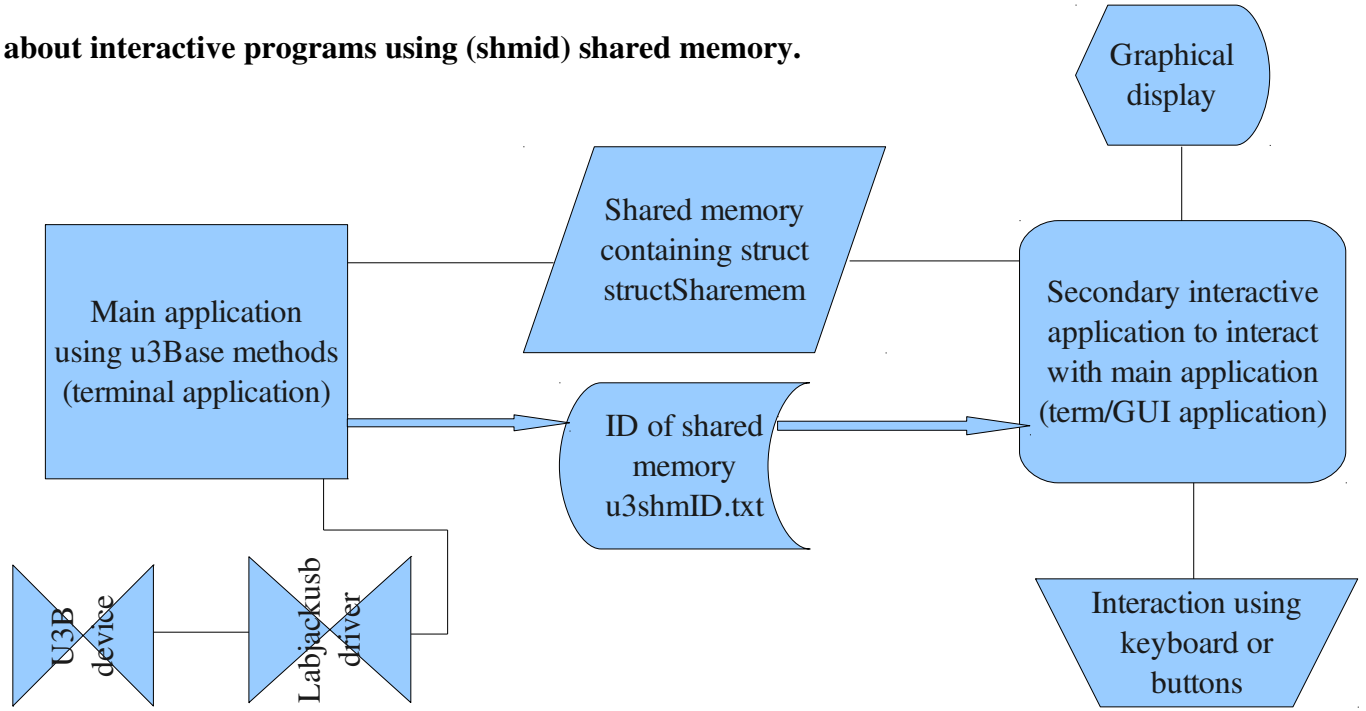
```
FIL7  Analog input channel  7 low value alarm/logic state r/o
FILT  Internal temperature    low value alarm/logic state r/o
EIL0  Analog input channel  8 low value alarm/logic state r/o
EIL1  Analog input channel  9 low value alarm/logic state r/o
EIL2  Analog input channel 10 low value alarm/logic state r/o
EIL3  Analog input channel 11 low value alarm/logic state r/o
EIL4  Analog input channel 12 low value alarm/logic state r/o
EIL5  Analog input channel 13 low value alarm/logic state r/o
EIL6  Analog input channel 14 low value alarm/logic state r/o
EIL7  Analog input channel 15 low value alarm/logic state r/o
FIM0  Analog input channel  0 mid value logic state r/o
FIM1  Analog input channel  1 mid value logic state r/o
FIM2  Analog input channel  2 mid value logic state r/o
FIM3  Analog input channel  3 mid value logic state r/o
FIM4  Analog input channel  4 mid value logic state r/o
FIM5  Analog input channel  5 mid value logic state r/o
FIM6  Analog input channel  6 mid value logic state r/o
FIM7  Analog input channel  7 mid value logic state r/o
FIMT  Internal temperature    mid value logic state r/o
EIM0  Analog input channel  8 mid value logic state r/o
EIM1  Analog input channel  9 mid value logic state r/o
EIM2  Analog input channel 10 mid value logic state r/o
EIM3  Analog input channel 11 mid value logic state r/o
EIM4  Analog input channel 12 mid value logic state r/o
EIM5  Analog input channel 13 mid value logic state r/o
EIM6  Analog input channel 14 mid value logic state r/o
EIM7  Analog input channel 15 mid value logic state r/o
FIH0  Analog input channel  0 high value alarm/logic state r/o
FIH1  Analog input channel  1 high value alarm/logic state r/o
FIH2  Analog input channel  2 high value alarm/logic state r/o
FIH3  Analog input channel  3 high value alarm/logic state r/o
FIH4  Analog input channel  4 high value alarm/logic state r/o
FIH5  Analog input channel  5 high value alarm/logic state r/o
FIH6  Analog input channel  6 high value alarm/logic state r/o
FIH7  Analog input channel  7 high value alarm/logic state r/o
FIHT  Internal temperature    high value alarm/logic state r/o
EIH0  Analog input channel  8 high value alarm/logic state r/o
```

```
EIH1  Analog input channel  9 high value alarm/logic state r/o
EIH2  Analog input channel 10 high value alarm/logic state r/o
EIH3  Analog input channel 11 high value alarm/logic state r/o
EIH4  Analog input channel 12 high value alarm/logic state r/o
EIH5  Analog input channel 13 high value alarm/logic state r/o
EIH6  Analog input channel 14 high value alarm/logic state r/o
EIH7  Analog input channel 15 high value alarm/logic state r/o
DAC0  Analog output         0 used as identifier in methods only
DAC1  Analog output         1 used as identifier in methods only
RCT0  Logic output port   (64) to reset counter0 w/o
RCT1  Logic output port   (65) to reset counter1 w/o
SSW0  Logig input sw switch 0 r/o
SSW1  Logig input sw switch 1 r/o
SSW2  Logig input sw switch 2 r/o
SSW3  Logig input sw switch 3 r/o
SSW4  Logig input sw switch 4 r/o
SSW5  Logig input sw switch 5 r/o
SSW6  Logig input sw switch 6 r/o
SSW7  Logig input sw switch 7 r/o
SSW8  Logig input sw switch 8 r/o
SSW9  Logig input sw switch 9 r/o


/p    Can be used to indicate physical position for both analog and logic
r/o   Read only for logic equations
w/o   Write only for logic equations
r/w   Read and write for logic equations if channel is output.
r/w   Read only for logic equations if channel is input.
```

Shared memory.

Information about interactive programs using (shmid) shared memory.



The main application is started first, usually in a terminal window. The main application writes a file called u3shmID.txt in the current directory for the main application. This file contains the ID of the shared memory in ASCII form like 6357011
The secondary interactive application must read this file in order to create a pointer to the shared memory.
Thereafter the secondary application can do controlled interactivity and display of the main application and thereby also the LabJack U3 device. All interaction are done via the shared memory which contains a structure called structSharemem. This structure can be found in u3Base.hh header file along with some important constants.
All secondary applications must be terminated before the main application is terminated, otherwise there is a risk that the shared memory segment will not be released before the computer is restarted.

The prerequisites in the main application are:

```
useSharedMemory();
```

Example of connecting to shared memory in a secondary application:

```
#include  <sys/shm.h>
#include  <stdio.h>
#include  "u3Base.hh"
int   main( int argc, char **argv )
{
 int                shmid;          //  Id of shmem to pass on to child program
 structSharemem     *shareptr;      //  Pointer to the shared memory assigned by shmget(...)
 FILE               *shmIDFile;     //  File data is the ID of the shared memory
 char               charTemp[255];
 if( ( shmIDFile = fopen( SHM_FILE_NAME, "r" ) ) != NULL ) { //  SHM_FILE_NAME is defined in u3Base.hh
  fgets( charTemp, 16, shmIDFile );
  fclose( shmIDFile );
  shmid = atoi( charTemp );
  if( (shareptr = (structSharemem *) shmat( shmid, 0, 0) ) == (structSharemem *) -1 ) {
   perror("can't attach to shared memory\n");
  }
  printf( "Using shared memory %d\n", shmid );
 } else {
  shmid = 0;
  printf( "Error opening file %s in order to acquire shared memory ID\n", SHM_FILE_NAME );
 }

 if( shmid > 0 ) {       //  Display the state of all software switches
  for( n = 0; n < 10; n++ ) {
   printf( "SSW%d: %d\n", n, shareptr->ssw[n] );
  }
 }
```

The structSharemem

```
typedef struct share_mem    //  This structure can hold all input/output in shared memory
{
  int                 newDataAvailable;          //  Parent sets this to 1 and any monitor can set it to 0
  structConfTable     confTable[U3IOALEN+1];     //  Varibale to hold retrieved with u3Base::getConfTable(...)
  structRelayOut      relayOut[U3IOALEN+1];      //  Variable to hold Logic output (relay) state
  int                 ssw[10];                   //  10 logic input software switches for boolean equations
} structSharemem;
```

See u3Base.hh header file for details about confTable[] and relayOut[] arrays.

The newDataAvailable is useful if the sampling frequency of the secondary application is faster that that of the main application. By setting newDataAvailable=0 upon reading the data in shared memory and only reading if newDataAvailable==1, one can avoid unnecessary readings.

The confTable[] array contains information about all defined inputs. That is, their actual value limits associated text and their type. The index of confTable[] is 0..n where 0 is the first buildInputCommandXXX(...) and n is the last buildInputCommandXXX(...) in the main application. One can also determine the correct index by searching for confTable[x].u3Pos string matching the input in question. **Writing to confTable[] should be avoided.**

The relayOut[] array contains information about all logic/relay outputs. Here the index corresponds to the absolute position on the U3 LabJack. Thus, FIO0=0, FIO7=7, EIO0=8, EIO7=15, CIO0=16 and CIO3=19. **Writing to relayOut[] should be avoided.**

The ssw[] array contains information about the state of the 10 possible software switches. The index is 0..9 corresponding to SSW0..SSW9. This array can be read in order to know the current value or it can be written to in order to set a new value. The values can be 0 for off/lo and 1 for on/hi. Any new value submitted will be registered during the next sample in the main application.