Foucault Pendulum Electronics Kit.

D14_Development Environments & Coding rules.

www.foucaultpendulum.nl

Document version	2026-05-25
Related Documents	All

In brief:
The software in this kit consists of 3 parts: the Arduino Firmware, the PC program for control and logging the pendulum and a PC program to analyse the logfiles.
The user needs to understand and be able to modify all 3 parts to use the system and fit it to his/her own requirements.
In this document I give some tips about the software development environments I used and coding rules I try to stick to.

Arduino environment:
For the Firmware I used the classical Arduino environment version 1.8.16. There exist more modern versions, I expect them to work as well, but I have no experience with it.
Link: https://www.arduino.cc/
Note: Quite recently Arduino was bought by Qualcomm. I have no idea what consequences this may have for the availability of hardware, software environments, libraries and other support.

Set the board to "Arduino Mega 2560". To use the Arduino Monitor set it to baudrate 115200
The special libraries needed are in the .ZIP file with the software.
Store them in your .....<version>/libraries directory.

Some notes for users of Linux Mint (I use version 21):
- To work with Arduino's you need to be member of the group *dialout*.
In a command window type: *sudo usermod -aG dialout <your user name>*
Do not forget the option *-a* for append. Without it you lock yourself out of the system because it makes you ONLY member of *dialout*, and no more of a.o. *sudo users.*

- Mint 21 comes with some stuff pre-installed for a Braille Keyboard. This interferes with the USB serial ports. Type: *sudo apt remove brltty* .
- When compiling you may encounter an error message about *unknown mmcu=avr6'*
Remedy: update Arduino AVR boards with the Arduino board manager.

Pascal environment:
For the PC programs I use Free Pascal version 3.2.2 with the Lazarus IDE version 2.2.6 on a Linux-Mint 21 PC. Other versions may work as well. Link: https://www.lazarus-ide.org/
Lazarus + FPC is an Integrated Development Environment (IDE) to make programs with a Graphical User Interface or GUI (windows like). It is available for many platforms:
M$ Windows, Linux, Mac, Raspberry-Pi, Android and more...
It is free, open source, living (maintained by voluntiers and has active forums) requires almost no configuration, has all features of modern object oriented languages (classes, multithreading, function and operator overloading, etc...) and compiles to native code (no run time environment required for an executable).
The slogan is "Write Once, Compile Anywhere" and to my experience that is close to the truth. However, expect cosmetic differences across platforms.
Laz-FPC is also close to the commercial Delphi IDE and language.
People who have worked with M$ Visual Basic will recognise many similarities in setting up GUI forms.

Some tips for Linux Mint users:
- Do not install Lazarus/FPC via the Mint software manager, you will get a rather old version.
Instead download a recent version from the Lazarus website. You need 3 debian packages. Type *sudo dpkg -i *.deb* in the directory where you downloaded them.
- You may encounter error messages about libgtk.
Install that library with *sudo apt-get install libgtk2.0-dev* .
You may need to type: *sudo apt --fix-broken install .*
- The lazarus editor has standard file-close boxes in the tabs. This is clumsy, because you easily close a file when you only want to switch to it and then you need several mouseclicks to get it back.
Do: *Tools > Options > Pages and Windows > Show close buttons in notebook:* Uncheck.
-There may be a problem with the Alt key to place controls on a smaller grid.
On your linux system maneuvre to: *System Settings > Windows > Behavior > Special key to move and resize windows:* set to Disabled. This works for other programs too.
To start the IDE from a file manager set the file association for the <projectname>.lpi file to: */usr/share/lazarus/<version>lazarus*. Other settings may cause lazarus to start with an empty project.

Some tips for working with the Lazarus IDE.
To change the name of your project: DO NOT change filenames manually outside the IDE.
Do *Project >Save Project As >* change the name of the *.lpi* file. All relevant changes are made automatically. Afterwards delete the files with the old name.
When you have imported new units or deleted unwanted units into/from a project some strange error reports may result from old stuff in the directories *lib* or *backup*..
Remedy: delete the directories *lib* and *backup*. This gives a clean recompile-all.
If you do a clean compile like this there will be a lot of warnings from the synapse library. Leave it like that, it will cause no problems.

When you have your system working I advise to still let the GUI run from within the IDE. If something goes wrong better error messages and diagnostic tools are available.

Some Coding Conventions I try to stick to.
- Variables are defined (given memory) in the module where they first get their value.
If needed elsewhere they are declared (tell that they exist) in the module Globals.h.
In Pascal this is realized by defining them in the *Interface* section of the unit.
- Where applicable variables have a postfix identifying the unit, like _mm for millimeter, _deg for degrees, _ticks for timer ticks, etc.
- I tried to use variable and function names which as close as possible reflect the purpose of that item. This often results in quite long names. Don't be afraid of that, we have ^C and ^V.
- I tried to use identical names for the message fields on both sides.
- Textboxes to/from which values are written/read are named like Tbx<VariableName>
- Same for the naming of other controls and the variables they operate on.
- In the firmware all timing is generated by the 16 bit Timer1, running on 20 kHz. So we have timeslices of 50 microseconds. Several actions which need to be started at a certain moment last much longer. If that is done within the interrupt handler interrupts would be missed. To overcome this problem we set a flag *Do_<action>* in the interrupt handler.
In the infinitely running *loop ()* such flags are tested and if found true a function with the *action* is executed and the flag is cleared. The action code itself must be interruptable of course, but this generally requires no special precautions.
- I like the {opening and closing} braces of a statement block on the same ident.
This makes the block structure much better visible. Same in Pascal with *begin* and *end.*
Procedures and functions have a comment telling from where they are called.
- I throw a lot of comments in between the code.
- I throw a lot of diagnostic statements in the code. OutCommented if not needed anymore.

Text boxes which are to be filled with default values or values from the parameter file initially (design time) have a value consisting of some 9's. This is an extra check to see that they are really filled with the defaults or file values.
Text boxes whose values are calculated at runtime or display measured values initially are filled with '----' . This is an extra check to see that they are really filled with the intended values.
- In each module I list the procedures and functions in that module as forward declarations. (except those in a class if present) This makes finding a function much easier.

Numeric values in the messages between Arduino and GUI.
are transfered as bit patterns. This eliminates the need for conversions from and to human readable ASCII and keeps the messages short. This method is possible because both Arduino and PC processors use the so called "little endian" format to store integer numbers in memory.
Floating point numbers cannot be transfered this way because the formats on these platforms are likely to differ. Floats are converted to apropriate multiples of 10, 100 etc, before being transfered as integers.
Booleans are stored as individual bits in a Command or Status word.
Yes, I know that these are dirty methods, but it works fast and reliable.

On the Pascal side (GUI) the messages are sent and received as type *shortstring*, an array of char, where element [0] represents the string length. The length itself is not sent, but a received string needs to have the proper length byte.
On the C++ side (Arduino) the messages are transfered as arrays of byte. The messages are sent and received with a predefined length (fixed length).
Because of the length byte in the message from Arduino to GUI there is a difference in byte numbering on the two sides.
Note: Pascal is quite strongly typed. Byte and Char are compatible but need to typecasted.

Writing and Reading the setting of RadioButtons to / from the parameter file.
This turned out to be tricky.
The values of the RadioButtons for Drive_SyncMode, Rim_Sync Mode and AmplitudeControlMode are enummerated types. These cannot as such be written to the parameter file, a typecast to the ordinal value is required.
Reading back directly into the enummerated type is also not allowed, must be done into an integer type. Assigning this integer to the enummerated type via a typecast can be done, but then the controls on screen do not follow.
My solution was to "click" the appropriate button via a case statement where the temporary integer was typecasted to the appropriate enumerated type.
Note that "clicking" on a radiobutton that is already checked does NOT trigger the change event.
One of the radiobuttons in a group seems to be "clicked" on startup. Probably that is the first one which was first put on the form during form design. Be sure that that is the one which is checked by default.
Using radiobuttons is a bit tricky in another way too. The standard event is the OnChange event. Mind that clicking a radiobutton triggers two OnChange events, one for the button which is activated and on for the deactivated button. The order in which these events are handled is not predictable. I have decided to use the OnClick event which only triggers for the activated button.

Network connections.
The Arduino and the GUI communicate via the local area network using UDP. You must give the Arduino a static (fixed) IPv4 address which is never used by other equipment in the network. You should therefore program your lan-router such that there is a (small) range of IPv4-addresses excluded from DHCP, give the Arduino an IP in that range and assure that no other equipment with the same fixed IP is ever connected to the LAN.
The Arduino's IP address is assigned in the module messages.cpp.

There is a way to circumvent these requirements on the network: Give the PC or laptop also a fixed IP in the same sub-domain as the Arduino, and connect it only to the Ardiuno. This will work, but the PC will be inaccesible through the network.
Connecting the PC/Laptop to the Arduino via the big world wide internet may introduce big problems, because of the time delays which may occur.

Firmware Version identification.
In the Firmware you should fill in the macro's BUILD_YY, BUILD_MM and BUILD_DD with the current data. From these data the VersionNumber is calculated as (YY-26) * 10000 + MM * 100 + DD. This gives a number fitting an unsigned int_16, which goes into the message to the GUI. The trick will work upto the year 2032.
In the GUI it is decomposed into a readable format and also logged.
Numeric values in C++ should not start with a leading zero. In that case they will be interpreted as octal and e.g. 08 will produce an error message.

Arduino Programming.
I generally do not use the typical Arduino statements like *Pinmode*, *digitalWrite*, etc.
I like to directly program the registers of the I/O peripherals. A consequence is that my software will generally not work on other Arduino's. The advantage is that I have full control and that the code executes faster.
A typical example are the A/D conversions. The Arduino statement for that is *result = analogRead (channel) .* This statement involves a complete setup of the A/D converter hardware, and then wait for the conversion result for each conversion again and again.
In an Interrupt handler you don't want to waste that time. We therefore prepare the A/D converter in the *setup()* section. In the interrupt handller we set the multiplexer (mux) to the channel we want to convert and start the conversion. In the next interrupt (timeslice) we then can fetch te result, the converter has done its work while we did something else. We only have to take care which cannel we started the conversion for, to put the result in the right place.

Schema's and board layouts are made with Eagle 7.7.0.
This is the latest free version before Eagle came in the hands of Autode$k. Several boards are larger than the limitation of the freeware version of Eagle, and were made using my licence from 2014.
I am willing to provide the Eagle files, but you cannot do very much with them unless you also have a licence for PCB's upto 16x10 cm in minimal 2 layers.