Foucault Pendulum Electronics Kit.

D07_Description_Firmware

www.foucaultpendulum.nl

Document version	2026-05_24
Related Documents	Firmware Source Code. D02_Options. D05_Description Electronic Circuits. D06_Arduino_PinUse_Messages. D08_Description_GUI_Software. D14_Development_Environment, Coding Rules. ATMega2560_Manual

In brief:
This document describes the processes in the firmware which detect the Center- and Rim Passes, generate the DrivePulses and does the Automatic Amplitude Control.
On the end some words about the averaging algorithm used in several places.

General.
The system has 4 methods to synchronize with the Bob.
This can be selected in the panel Drive Parameters on the GUI.

CenterPass_Magnetic: The timing of the Drive Pulses is synchronized with CenterPasses detected by a separate CenterPass Detection coil or by the signal coming from the DriveCoil itself, when the bob flies over it.

CenterPasses Capacitive: The timing of the Drive Pulses is synchronized with CenterPasses detected by an electrode in the center below the bob, using the 465 kHz signal on the wire and bob.

Charron Ring: The electronics are prepared to detect the wire touching a Charron Ring, but no further implementation is done.

Resonance: The electronics and software can deliver DrivePulses at a very stable frequency which can be adjusted in extremely small steps. This is for future experiments where I will try to drive the pendulum with the frequency of the major ellipse axis and try not to excite the slightly higher frequency of the minor axis. This to limit ellipse growth.

The Detectors:
In the firmware we have three detectors: for CenterPasses_Magnetic, CenterPasses_Capacitive and RimPasses_Magnetic. The signal for Detecting RimPasses_Capacitive is implemented in the electronics, but in the firmware not beyond the A/D conversion.

The three detectors function simultaneously, but quite independent of each other.
Each detector has its own time base: a *PositionCounter*. The PositionCounters for the CenterPass detectors are set to zero when their CenterPass is detected.
The PositionCounter for the RimPass detector is zeroed by the CenterPass detector which is selected for synchronizing the Rim Passes.
Besides that we have a PositionCounter for the Drive Pulses which is zeroed by the CenterPass Detector selected for synchronizing the DrivePulses, and a

PositionCounter for the PMS. This counter is zeroed at the CenterPass of the detector synchronzing the DrivePulses, but only when *LastPoint = true*. So this counter runs for a full swing, where the others do a HalfSwing.

StateMachine *DetectCenter_Cap*.

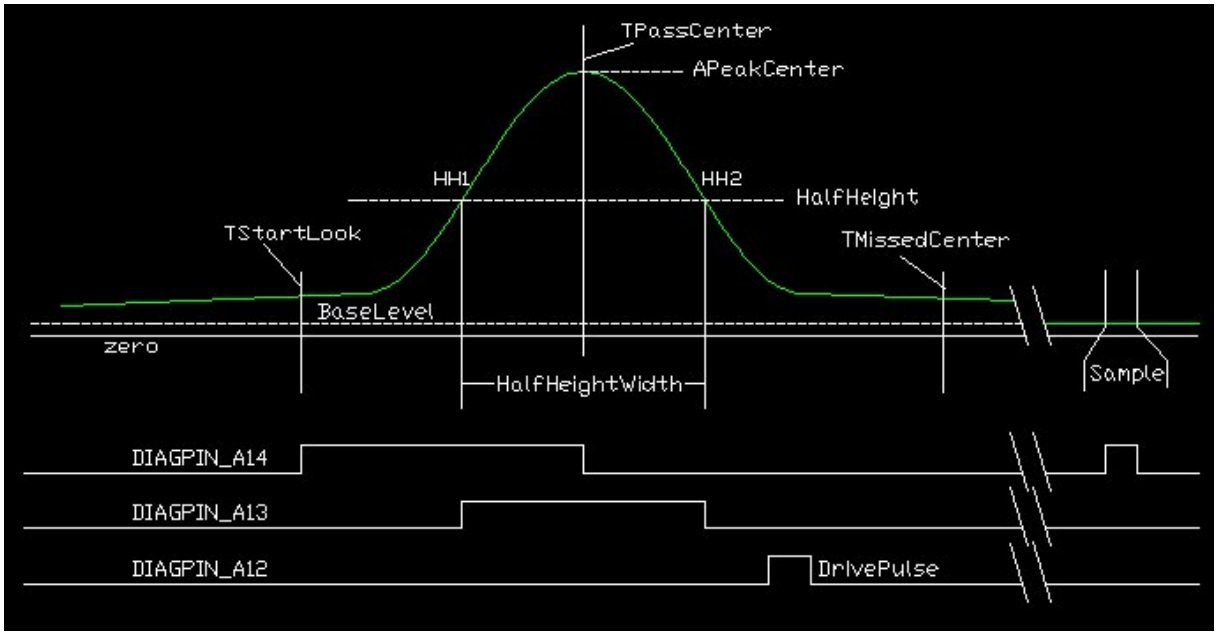

Fig. 1 The signal from the Center electrode and the steps of the detection process. (Not to scale)

The 465 kHz signal from the Center electrode is amplified, rectified and fed to the analog input channel *Adc_Center_Cap*. (green trace) When the bob is far away there is a small signal called BaseLevel.

The statemachine *switch (stCenter_Cap) { }* in the function *DetectCenter_Cap ()* starts for the first time after booting in the state *ccap_Idle*. Here we give some variables a sense making initial value and jump to the state *ccap_Pulse* where we wait until the signal *adc_Center_Cap* rises such that we may expect the bob te be close to the center. We jump to the state *ccap_WaitStart* where we wait until the *PositionCounter_Center_Cap > TStartLookForCenter_Cap.*
As *TStartLookForCenter_Cap* is set to some 80% of the half-period time we skip one CenterPass, but after that we are close to the center again and in the state *ccap_WaitHalfHeight1* we wait until the adc signal rises above the *AHalfHeigthCenter_Cap* level. This level is initially set to a sense making value, but will be augmented later on when we are in sync. The *HalfHeightWidthCounter* is now set to zero to be able to do the counting work.

Next state is *ccap_WaitCenter* where we track the rising *adc_Center_Cap* signal until it starts falling again. That is the moment of the CenterPass. We give *TQuarterSwing* half the value of the *TPassCenter_Cap*, which identifies the far end of the swing.
Depending on the *DriveSyncMode* and the *RimSyncMode* we set some values and jump to the state *ccap_WaitHalfHeight2*. Here we follow the signal until it drops below *AHalfHeigthCenter_Cap*, and now the value of the *HalfheightWidth* of the pulse is known.

The last steps are to wait for the *PositionCounter_Center_Cap* to reach the value *TQuarterSwing*. We are now at the far end of the swing and here we sample the base level of the signal from the center electrode. We take 20 samples and average them into

avBasecenter producing *ABaseCenter_Cap*. Also the peakvalue of the centerpulse is averaged into *avPeakCenter*. Finally a new, augmented value is calculated for *AHalfHeightCenter_Cap* which is used in the centerpasses to come.
From here we jump to the state *ccap_WaitStart* for the next cycle.

Outside the statemachine, actually the *switch-case* tatement, a test is done for the *PositionCounter_Center_Cap* to reach the value *TMissedCenter_Cap*. In that case the whole synchronisation procedure is repeated from the state *ccap_Idle*.

Also when the signal *ForceResync* is given from the GUI the statemachine is kicked to the *Idle* state to try for a new synchronisation, but only when this detector is selected to synchronize the Drive Pulses. This part is handled in the body of th Interrupt Service Routine *ISR (TIMER1_COMPA_vect).*

The background for the measurement of the half-height width of the capacitively detected center pulse is that it is a measure for the velocity of the bob and so for the amplitude of the pendulum. (A correction may be needed if there is ellipticity).

On the Arduino Monitor we have the following output when in *TestMode1*:

```
15:29:47.209 -> CC_Start  17001, 173
15:29:47.293 -> CC_HH1    17672, 309
15:29:47.312 -> CC_Center 18704, 605
15:29:47.331 -> CC_HH2      864,  1896, 304
15:29:47.738 -> CC_Qswing  9353,   6
15:29:47.738 -> CC_Sampl   9373,   6, 307
```

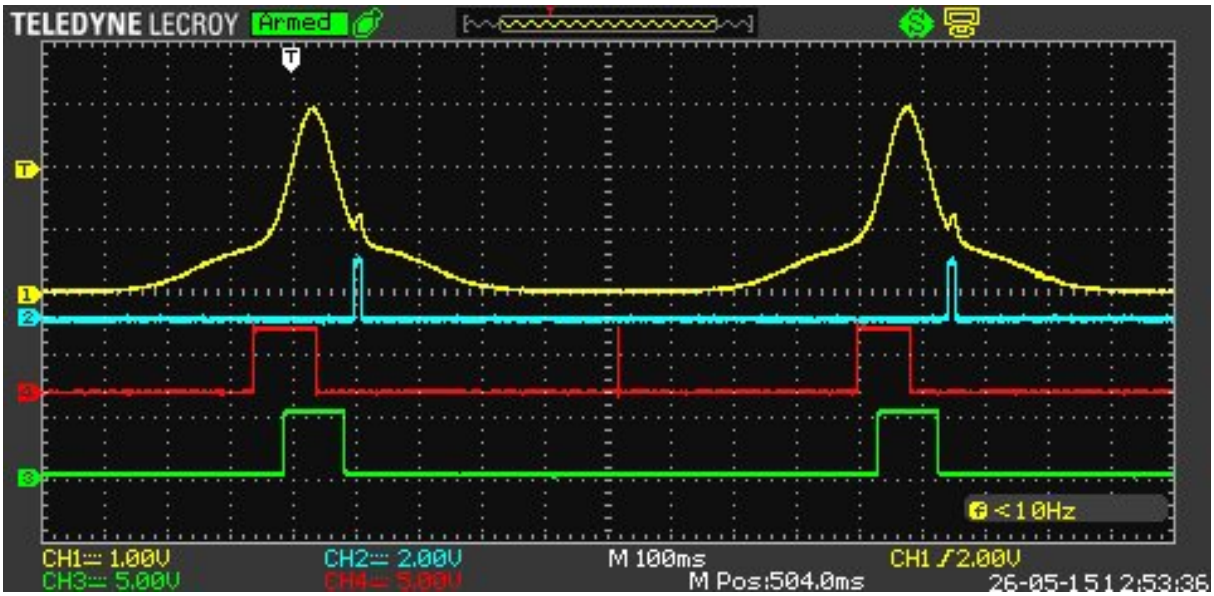

Fig 2. The signals from the Capacitive Center detection on the oscilloscope.

Yellow: The CenterPass signal at analog input A4.
Blue: The drive pulse at TP_1_DRV.
Red: The *DIAGPIN_A14* goes high at the Treshold and Low at the CenterPass.
Green: The *DIAGPIN_A13* shows the timing of the HalfHeight transitions for the HalfHeight Width determination.
And yes, there is a small amount of crosstalk from the drive pulse, caused by an inadequate shielding.

These signals are from my sub-meter pendulum with a period time of ca. 1.8 seconds.

Detecting Rim Passes Capacitive.

In my sub-meter pendulum I have installed a ring shaped electrode on the floor to detect rim passes with the capacitive method. However, the signal from this electrode turned out to have a shape which makes it very difficult to recognize the passes of the bob. The cause is the combination with the shape of the bob which is disk-like with a diameter close to the diameter of the ring electrode. In fig 3. one can see this signal. The “shoulders” represent the ring passes, but are drowned in the signal from the whole bob over the ring.
With a bob shaped more like a tender cylinder the signal would probably be more useable. For the time being I have decided not to put any efford in Capacitive Rim detection.

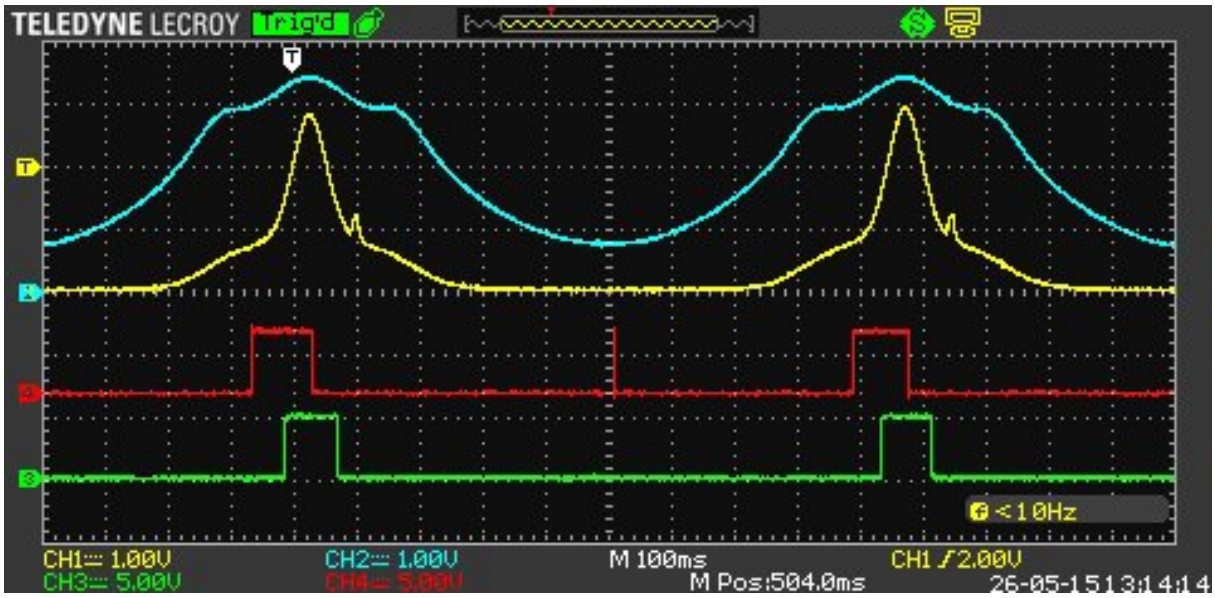

Fig 3. Much like Fig 2. but the blue trace now shows the signal from the ring shaped Rim electrode.

StateMachine *DetectCenter_Mag*.

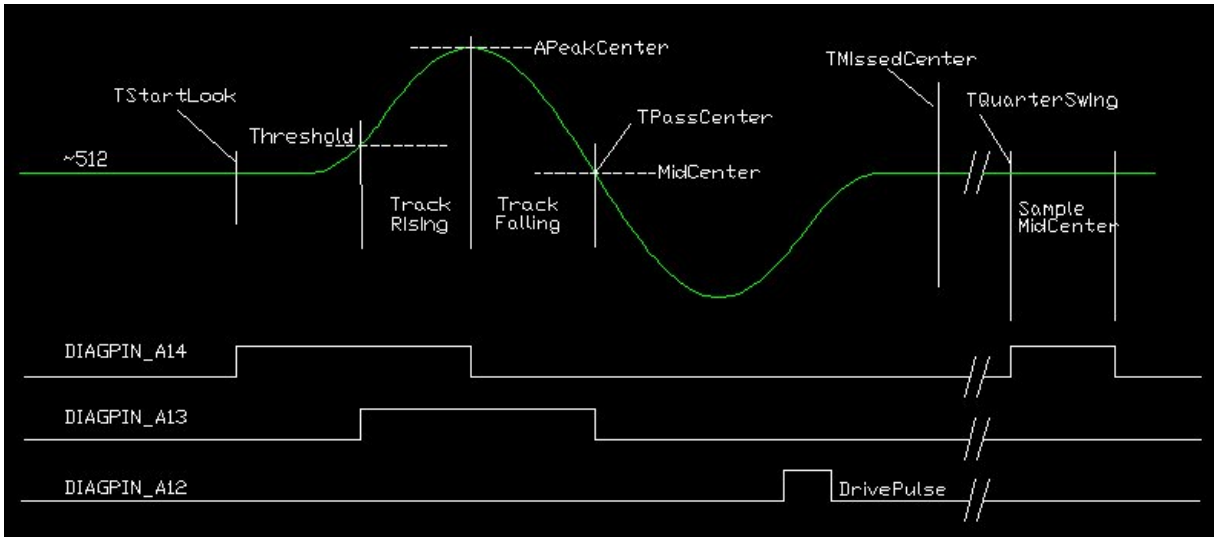

Fig 4.
The signal from the CenterCoil, the steps in the detection process and the diagnostic signals. (not to scale)

The signal (green) from the CenterCoil is amplified and lifted to aproximately half the range of the A/D converter (0..5V, 10 bit resolution, so 0..1023 in range. Halfscale is 512)

We start in the state *cmag_Idle* where some preparations are made and we jump to the state *cmag_WaitPulse*. Here we wait until the signal rises above a certain level, default 550, which means that the bob is approaching the coil. Then we zero the *PositionCounter* and jump to the state *cmag_WaitStart* where we wait until the *PositionCounter > TStartLookForCenter*. This is a value of approximately 90% of the HalfSwing time, so we will miss the first CenterPass. No problem, this is just the start of the synchronistion process, and now we are close to synchronisation.
The next step is to wait for a threshold of some 50 units above the center level of 512. This is to prevent false triggers when the signal is noisy. If your signal is less noisy you may reduce this margin.
When the threshold is found we start tracking the rising signal in the state *cmag_TrackRising*. We track until the signal is falling again. Here we freeze the adc value as *APeakCenter_Mag*. Then we jump to the state *cmag_WaitCenter* where we wait until the signal falls below the midlevel *AMidCenter_Mag*. This is the moment of the CenterPass and we freeze that position as *TPassCenter_Mag*. In fact we are in sync now. We put half the value of *TPassCenter_Mag*. in *QuarterSwing* and wait in the next state until that position is reached. That is at the far end of the bob's swing. Here we take some 20 samples of the adc value and average them to update the *AMidCenter* level.
Then we return to the state *cmag_WaitStart* and wait for the next HalfSwing.

Outside the statemachine is constantly checked if *PositionCounter_Mag > TMissedCenter_Mag.* In that case we decide that the CenterPass has been missed and we are out of sync. We kick the statemachine to *cmag_Idle* for a new try to find synchronisation.
Also when the signal *ForceResync* is given from the GUI the statemachine is kicked to the *Idle* state, but only if the DriveSyncMode is Center_Magnetic. This part is handled in the body of th Interrupt Service Routine *ISR (TIMER1_COMPA_vect).*

When TestMode2 is activated some diagnostic messages appear on the USB serial output of the Arduino which can be viewed with the Arduino Monitor (ctrl-shift-M) or on a suitable terminal program. Also the diagnostic signals pictured in fig 5 can be viewed with an oscilloscope.
DIAGPIN_A14 is high from *TStartLookForCenter_Mag* to the moment where the peakvalue is found, and during the sampling at the far end of the swing.
DIAGPIN_A13 is high from passing the threshold to finding the CenterPass.

On the Arduino Monitor we'll see the following output when we are in TestMode2: (example 4m pendulum)

```
16:44:15.396 -> CM Start  38001
16:44:15.429 -> CM Thres  39208
16:44:15.528 -> CM Top    40536, 727, 754
16:44:15.561 -> CM CPass  41272, 512, 515
16:44:16.586 -> CM Sample 20637, 515
```

Legend: See the relevant statements in the frirmware.

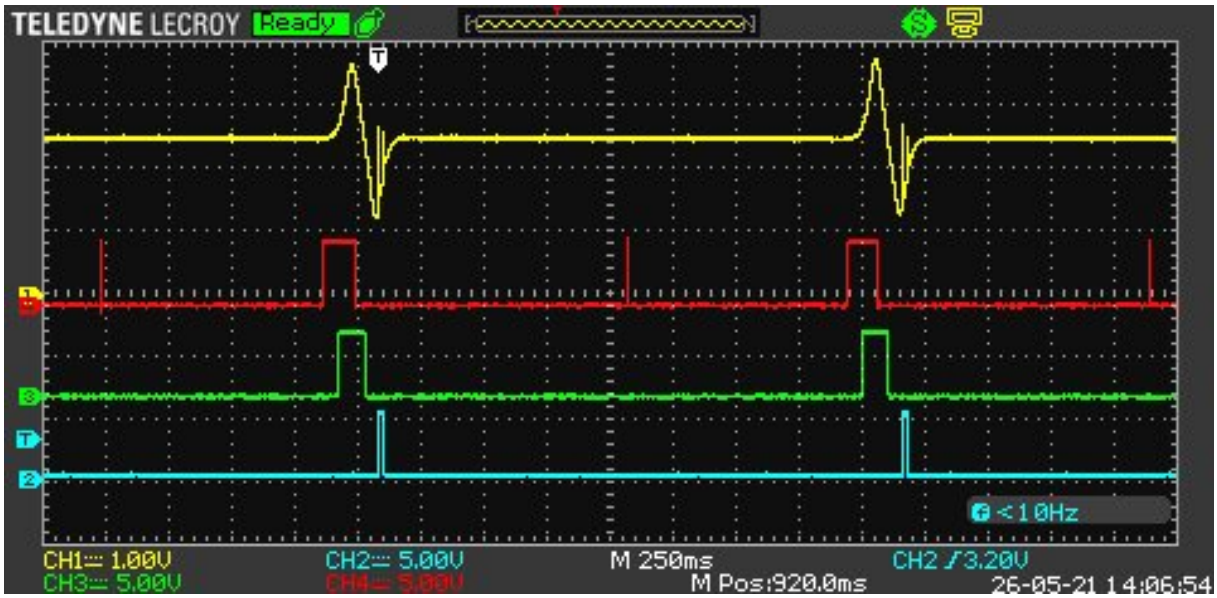

fig 5. The signals from the 4m pendulum on the oscilloscope.
Yellow: CenterCoil signal on pin A5.
Red: signal on *DIAGPIN_A14* is high from TStart to Detect Peak, and Sample Midlevel..
Green: sigal on *DIAGPIN_A13* is high from Threshold to Center.
Blue: Drive Pulse on *DIAGPIN_A11*.
We also see the induction of the DrivePulse.

StateMachine *DetectRim_Mag*.

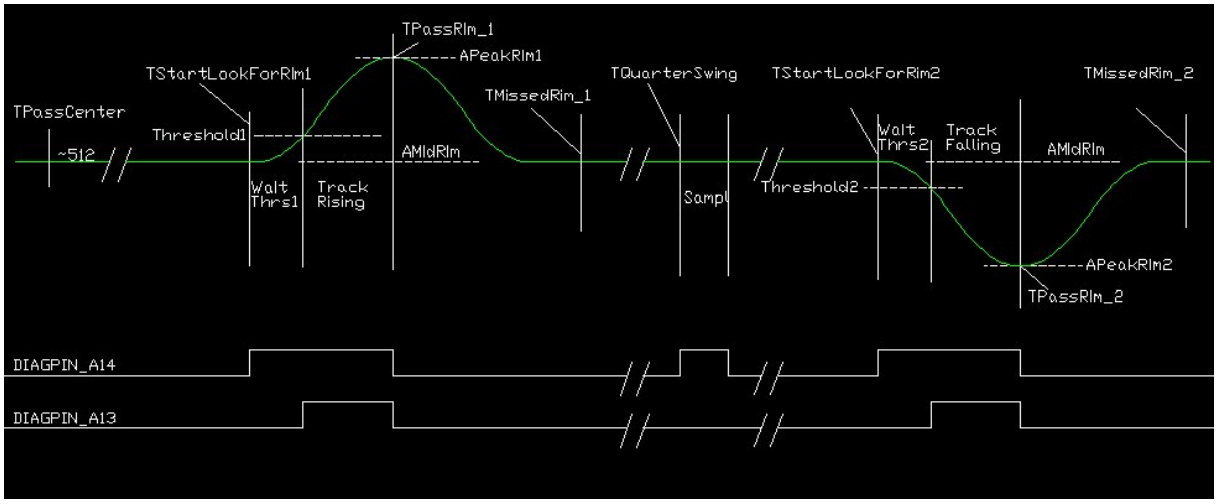

Fig 6. The signal from the RimCoil and the steps of the detection process and the diagnostic signals. (not to scale)**.**

The signal (green) from the RimCoil is amplified and lifted to aproximately half the range of the A/D converter (0..5V, 10 bit resolution, so 0..1023 in range. Halfscale is 512) Starting at the left at the time of a CenterPass we expect a positive going signal when the bob passes the RimCoil outwards and a negative signal when the bob goes back inwards to the center. During some time after a CenterPass we ignore the signal as it may be distorted by the DrivePulse.

After Power-ON we start in the state *rmag_Idle* and wait until the signal *HaveCenter* becomes true. This signal is generated by the CenterPass detector which is assigned to

synchronize the RimPasses (*RimSyncMode*) When that CenterPass detector is in sync it also zeroes the *PositionCounter_Rim_Mag* at each CenterPass. On *HaveCenter* we jump to the state *rmag_WaitStart1* and wait until
PositionCounter_Rim_Mag > TStartLookForRim1_Mag. This “dont look” time is because somewhere during that time the DriveCoil is engaged and will induce a disturbing signal.
After TStart we wait until the Rim signal goes above a certain lthreshold level indicating that the RimPass is soon to come. This is to prevent false triggers when the signal is a bit noisy.
From then on we are in the state *rmag_TrackRising*, where we track the signal until it starts falling again. That is the moment of RimPass1.
If that does not happen for whatever reason the PositionCounter will increase beyond *TMissedRim1*, a *MissedRim1* is reported and we start over again from state *rmag_Idle*.

Normally a jump is made to *rmag_WaitTQuarterSwing* where we wait for the PositionCounter to pass that value. TQuarterSwing is set by the CenterPass detector assigned to the DrivePulse synchronisation, and at a valid CenterPass it is set to half that TCenterPass value, which identifies the farest position of the bob. Here the bob is far away from the coils, so we can safely take samples to find the exact value of *AMidRim_Mag*. This value should theoretically be 512, but due to tolerances it may be a bit off.
We need that *AMidRim_Mag* value to reliably calculate the peak amplitudes of the RimPass signals. It can tell us about the horizontality of the RimCoil.
Note that on the GUI these peak amplitudes are given as the abs difference with this MidLevel.

Next we go to *rmag_WaitStart2* from where the steps to find RimPass2 are similar as for RimPass1, with the difference that we now have to track a down going pulse.
When *TestMode3* is true some diagnostic messages appear on the USB serial output and on the diagnostic pins.
DIAGPIN_A14 is high from *TStartLookForRim* to the moment where the peakvalue is found, and during the sampling at the far end of the swing.
DIAGPIN_A13 is high from passing the threshold to finding the peakvalue.

On the Arduino Monitor we can see the following output: (example 4m pendulum)

```
16:30:30.787 -> RM Start1  8001
16:30:30.787 -> RM Trsh1   8002
16:30:30.854 -> RM Pass1   9370, 681
16:30:31.416 -> RM Sample 20605, 515
16:30:31.813 -> RM Start2 28001
16:30:31.979 -> RM Thrsh2 31338
16:30:32.045 -> RM Pass2  32818, 353
```

Fig 7. The signals from the 4m pendulum on the oscilloscope.
Yellow: RimCoil signal on the Arduino input A7.
Red: *DIAGPIN_A14* shows the times from TStartLook to TPass, and the time of sampling.
Green: *DIAGPIN_A13* shows the times form Pass-Threshold to TPassRim.
Blue: The DrivePulse indicates the position of the CenterPass. Also visible is the induction in the RimCoil.

DrivePulse Generation.

The Drive Pulse time is generated in the function *ISR (TIMER1_COMPA_vect),* the 20 kHz interrupt handler for Timer 1, which also calls the Center- and Rim Pass detectors.
See the document “DO4_Description_Electronic_Circuits” for the circuit description.
The Drive Current is set to either a maximum or a minimum level by the boolean *MaxDrive*, which is set by the active Amplitude Control mechanism, or forced from the GUI. This boolean selects either Drive_MaximalCurrent or Drive_MinimalCurrent as given from the GUI. This is the PWM value for Timer4 which runs on some 16 kHz and produces a PWM output on pin 6 of the Arduino. See DO5_Description_Electronic_Circuits.

The DrivePulse timing is given by the values of *Drive_Start* and *Drive_Stop*, which in the GUI are calculated from the parameters *Drive_Position* and *Drive_Width*. They are calculated such that *Drive_Position* is in the mid of *Drive_Width.*

The option to use the signal from the DriveCoil to detect CenterPasses requires two other signals to be set, just before the DrivePulse begins and a longer time after it stopped. This gating prevents the sensitive pre-amplifier for the CenterPass signal to be overdriven by the DrivePulse.

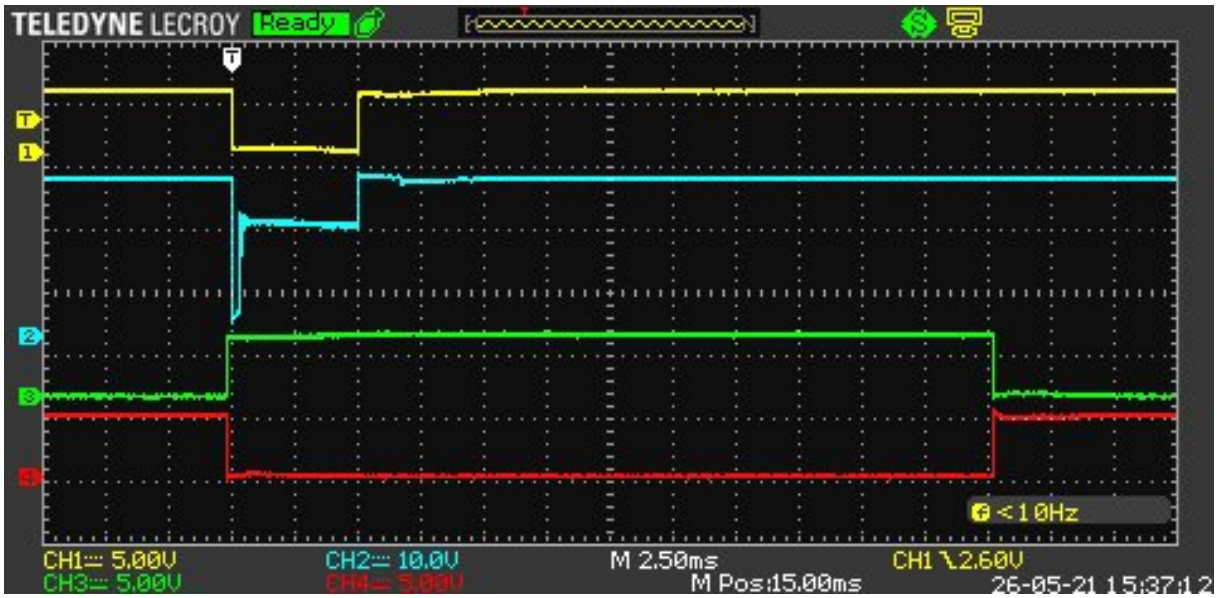

Fig 8. Signals in the DrivePulse circuit.
Yellow: The /DRV signal at Mega pin 41. The pulse width was set to 100 ticks of 20 kHz.
Blue: The signal at the tab of transistor Q3. We can see that it is going down ca. 9 Volt during the pulse. The overshoot at the start and ending of the pulse are due to the inductance of the coil. This is a normal phenomenon.
Green: The signal at Mega pin 45, shorting the amplifier IC1C.
Red: The signal at Mega pin 49, gating the signal with Q6 and Q7.

Note: The signal on the Q3-tab might show strong oscillations. In that case apply the snubber circuit mentioned in DO10, Assembly instructions.

Note: The option to derive the CenterPass pulse from the DriveCoil itself has never been tested so far.

FirstHalf, LastPoint, timing of Messages.

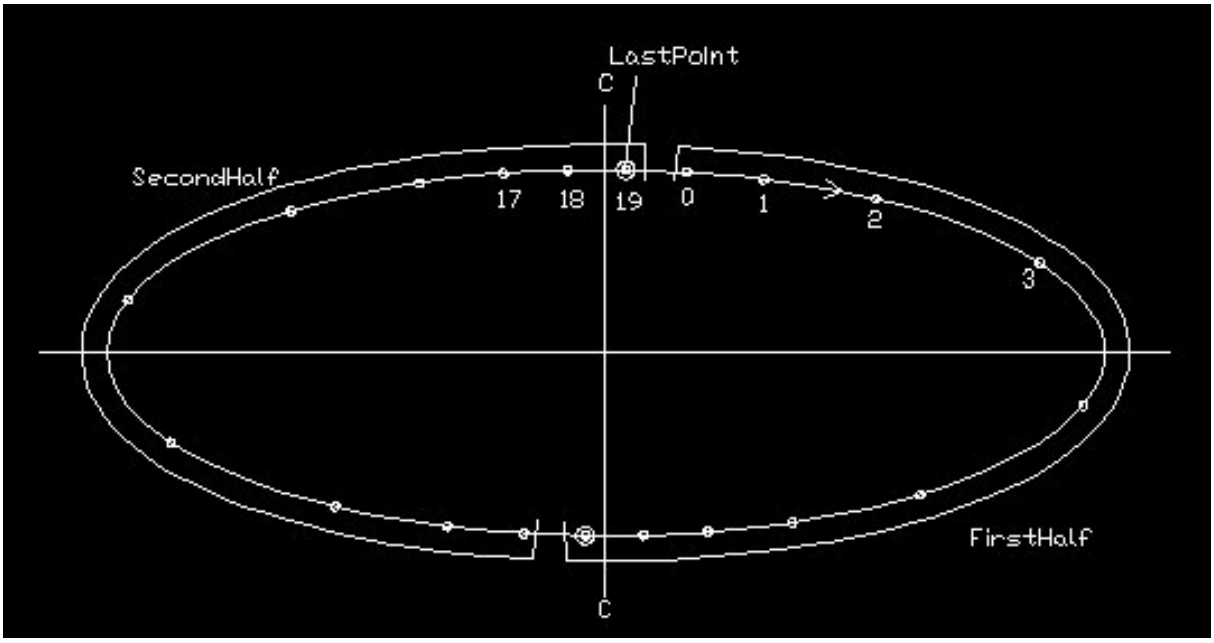

Fig 9. The elliptical path and the messages.

In fig 9. an (exaggerated) ellipse is drawn with dots on the moments where messages are sent. The rotational direction is CW in this example. The moments of sending messages is equidistant in time, but not synchronized with the movement of the bob.
CenterPasses are detected when the bob crosses the line C-C, but it will take a short time before the message with CenterPass notion is sent (double circles)
We identify the message with a CenterPass as the last one from a halfswing, and invert the signal *FirstHalf* after the message has been sent. Which half becomes the first is arbitrary, but as long as the system is in sync we continue to count 0-1, 0-1, 0-1, etc.

The important thing is that we do all the calculation for finalizing a whole swing during the processing of the message with LastPoint = true. During that processing also the Point numbering is zeroed. During FirstHalf the points are tracked until the farest distance from the center has been found for an estimate of the precession angle. The farest point in this example will be nr 5, as 4 has the largest distance, and 5 is found on the way back.

Each message also contains PMS information from the time the message was sent.
The 8 analog input channels are one by one sampled by the 20 kHz sample clock, so each channel is sampled with 2.5 kHz. So there wil be a latency of maximal 400 usec before the data is sent. Because of the random nature of this latency the resulting errors can be regarded as random noise.

Automatic Amplitude Control.
Two methods of keeping the pendulum's amplitude constant are implemented.
Method 1: Using a Rim Coil.
Here the time from CenterPass (either Capacitive or Magnetic) to the first, outgoing RimPass is used as a measure for the actual amplitude.
The RimPass time is taken as the mean of two succesive RimPass1 events to cancel the effect of a not well centered rimcoil. and is compared with a precalculated value *SetPointAmplitude_ticks*. When it takes the bob to long to reach the RimCoil we decide the pendulum's amplitude is to small and we set the boolean *MaxDrive,* which causes the maximum drive strength to be used. Otherwise *MaxDrive* is set to false and the minimum drivestrength is used.
The value of *SetPointAmplitude_ticks* is calculated in the GUI from the settings for the desired pendulum amplitude, the radius of the RimCoil and the measured PeriodTime of the pendulum. The PeriodTime is constantly recalculated as the sum of two succesive CenterPass times, (half-swing times) and averaged with the leaking bucket algorithm. The sum of succesive passes is taken to eliminate the effect of a not well centered detection coil or electrode.

Method 2: Using the width of the CenterPulse Capacitive.
The Half-Height width of this pulse is determined at each pass, averaged and compared with the precalculated value *SetPointAmplitude_ticks*. When the witdh is to large we decide the pendulum's amplitude is to small and we set the boolean *MaxDrive,* which causes the *Drive_MaximalCurrent* to be used. Otherwise *MaxDrive* is set to false and the *Drive_MinimalCurrent* is used.
So in both cases (magnetic or capacitive) the compare action is the same, only the calculation of the target value differs. The compare action is done in the firmware, the precalculation is done in the GUI.

No Automatic Amplitude Control.
If this option is selected one should set the Drive strength to a fixed value with one of the checkboxes *ForceMinimum* or *ForceMaximum* in the Drive Parameter pane.
Changes in the amplitude can stil be seen and logged as changes in *TPassRim1_Mag* and / or *THalfHeightWidth*, if enabled.

Communication with the GUI.
Communication with the GUI (the Pascal PC program) goes by Ethernet UDP messages. The GUI sends messages to the Arduino and the Arduino responds on each message with a return message.
Note: You should configure the IP address of the Arduino te be in your local LAN domain as a fixed IP (no automatic IP assignment). Modify your LAN router settings so that there is a (small) range of IP addresses excluded from DHCP and be sure that no other device in your LAN uses the address of the Arduino. (Try with the ping command).

In the firmware the messge handling is in the module messages.cpp.
In the function *Init_Messages ()* the setup for the ethernet connection is done.
The function *Update_Messages ()* is called very frequently from *loop ()* in the .ino file.
First the ETH interface is polled for an available packet. If found and if it has the proper size the message is read and decoded, and after that the outgoing message is composed and sent to the GUI.
Both messages have a simple checksum mechanism to verify the sanity of the data transport.

DecodeInMessage () first does a check on the *CheckSum*. If found OK it copies the values from the *InMessageBuffer* to local variables and takes a number of actions.
At a *CheckSumError* the contents of the message are not used and the error is reported in the OutMessage. There is no special action if messages continue to have errors. If there are no incoming messages for a few seconds the Blue Led COMM starts to fast flash. Normally it flashes at a slow rate.
Basically, if the pendulum runs neatly it will continue to do so when the messages stop.

PrepareOutMessage () copies the values from local variables into the OutMessageBuffer, and calculates the CheckSum.

The way numeric values are stored in the messages is described in DO14: Development_ Environment.

Diagnostic Features.
In the Arduino firmware many diagnostic features are built-in. One way is by messages on the USB serial connetion, to be viewed with the Arduino Monitor (Ctrl-Shft-M) or any suitable TTY program. The other way is by observing signals on an oscilloscope.
Scope signals can be taken from any accessible point on the electronics boards, from the analog inputs A0 .. A7 and from the DIAGPIN's A8 .. A15 on the Arduino connector.
By entering the digits 0 .. 9 on the monitor these features can be activated.

Digit	Signals from	DIAGPIN_A	USB-Serial	Remark
0	-	-	-	Switches all OFF
1	Center Pass Capacitive.	14: StartLook to Center, Sample Baselevel 13: Half Hight Pulse Width. 12: Drive pulse.	An Example of the output is given above.	
2	Center Pass Magnetic.	14: StartLook to Center, Sample Midlevel. 13: Threshold to Center. 12: Drive pulse.	An Example of the output is given above.	
3	Rim magnetic.	14: StartLook to Rim Pass, Sample Midlevel. 13: Threshold to Rim Pass.	An Example of the output isgiven above.	Both Rim Passes

4	A/D results.	-	ADC 0 .. 7	Each time a messageis sent by the Arduino
5	Amplitude Control System.	-	SetPoint and actual value.	
6	Timing of the Messages.	14: FirstHalf. 13: LastPoint. 12: Toggle on Msg Sent. 11: Process InMsg.	-	
7	t.b.d.			Make your own
8	t.b.d.			Make your own
9	t.b.d.			Make your own

The DIAGPIN_A15 shows the frequency and the duration of the 20 kHz interrupt handler *ISR (TIMER1_COMPA_vect)* which does most of the work.

DIAGPIN_ A8, 9 and 10 are currently not in use.

OptionJumpers.
The BobControl Board has locations for 4 OptionJumpers OPT1-1 (should read OPT1-4). Currently only OPT1 is in use to suppress the LedTest during startup of the Arduino. These jumpers are read at regular intervals. In the firmware they are called *OptionJumper1 .. 4.*

Leaking Bucket Averager.
On several locations in the firmware and in the GUI an averaging algorithm is used, known as the “Leaking Bucket Averager”. The basic algorithm is:

`av = av * (1 – f) + New * f, where 0 <= f <= 1`.

In words: to get the new average we subtract a fraction from the old average and add the same fraction of the new sample.

When we deal with integer numbers and we chose the fraction f as $1/2^n$, then the implementation becomes very effective, because division and multiplying by a power of two involves only bitwise shifting of the number. So we do:

```
av = av – (av >> n) + (new >> n);
```

Unfortunately if we do it this simple the least significant n bits would drop off by the shifts. Therefore we think everything shifted n bits to the left and it becomes.

```
avs = avs – (avs >> n) + (new);
av = avs >> n;
```

In fact we use fixed point arithmetic, where the separating point is n bits from the right. Often a 32 bit long integer is needed for avs to prevent overflow.

Such a filter behaves as a first order low pass filter with a time constant of $Fs / 2^n$ where Fs is the sample frequency.
The -3dB corner frequency of such a filter is $F_{-3dB} = Fs / (2 \pi 2^n)$.

On a PC platform we can just use floating point aritmetic because of the much faster CPU’s there.