September 23rd, 2022

Lancaster University

“Develop hardware and software to digitally control a DDS, PLL and VCO RF source frequency”

For STFC and Lancaster University

Benjamin Green

Supervisors: Dr Nirav Joshi, Dr Harry Marks

UKRI Science and Technology Facilities Council

The Cockcroft Institute of Accelerator Science and Technology

1 Table of Contents

2 Table of Figures

3 The Project

3.1 CLARA

CLARA or 'Compact Linear Accelerator for Research and Applications' is an advanced electron accelerator test facility with plans to expand to be the European test bed for accelerator and FEL research and development [1]. CLARA has three planned phases of which phase one has already been operational, and the process of moving to phase two has begun. Phase two of CLARA intends to deliver a beam energy of 250MeV, 250pC of beam charges at 100HZ repetition rate and intends to be capable of performing experiments beyond what phase one was capable of by virtue of having a dedicated full energy exploitation beamline [1]. The FEL diagnostics, the subject of this work, will be implemented as part of CLARA phase 3.

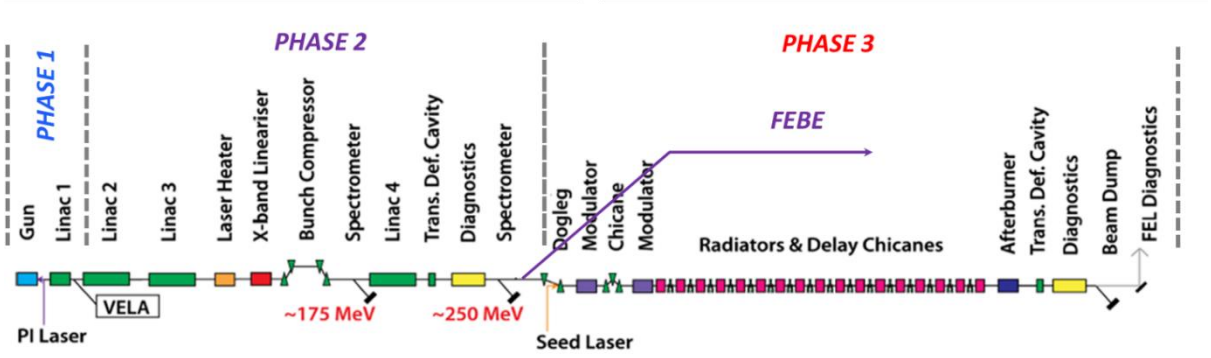

Figure 1 CLARA Timeline [1]

3.2 Wakefield Monitor

One issue that can happen to an electron beam on a system such as CLARA is beam degradation due to Wakefields. Wakefields occur due to electron bunches leaving a trail of electric field which influences particles coming after [2]. Wakefields can occur for a number of reasons [2] but by aligning the X-band lineariser and the linear accelerator, the degradation caused by Wakefields can be reduced [3]. A Wakefield monitor measures the effect of the Wakefield by "coupling to the transverse higher order modes (HOM) excited by the offset beam" generating a signal to correct misalignment of the electron beam [3]. This is especially useful for an FEL due to the pronounced beam degradation this effect can cause on such a system.

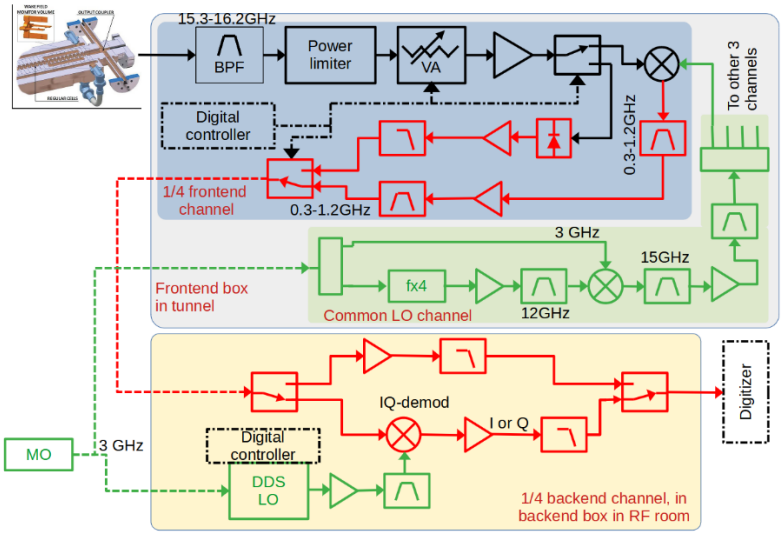

Figure 2 CLARA Wakefield Monitor Design

Figure 2 shows the full Wakefield monitor design currently under development at STFC's Daresbury site. This work only aims to focus on the back end of this design.

3.3 Summer Project

The back end of the Wakefield monitor has a narrowband spectrum analysis path which is intended for the four-channel output of the previous sections of the system. To obtain useful amplitude and phase information from the previous sections, an IQ demodulator is used which takes an input from the front-end spectrum analysis path and a stable LO signal. This demodulator is the last section of signal processing before feeding into a digitiser.

The project was to develop measurement and control software capable of controlling four oscillators responsible for providing the stable LO signal to the four channels of the IQ demodulator. The control software should eventually be able to be ported to EPICS for integration into the CLARA EPICS control system or a specific EPICS IOC should be developed to accomplish the task.

4 The Approach

4.1 AD9914 Evaluation Board

The Analog Devices AD9914 evaluation board is a direct digital synthesizer capable of synthesising a waveform of up to 3GHz. This was chosen as the LO for the demodulator for a number of reasons which will be explained later. Four of these boards are required for the four channels of the IQ demodulator and each must be individually controlled.

Analog Devices states "The AD9914 is a direct digital synthesizer (DDS) featuring a 12-bit DAC. The AD9914 uses advanced DDS technology, coupled with an internal high-speed, high-performance DAC to form a digitally programmable, complete high frequency synthesizer capable of generating a frequency-agile analogue output sinusoidal waveform at up to 1.4 GHz. The AD9914 enables fast frequency hopping and fine-tuning resolution (64 bit capable using programmable modulus mode). The AD9914 also offers fast phase and amplitude hopping capability. The frequency tuning and control words are loaded into the AD9914 via a serial or parallel input/output port" [4].

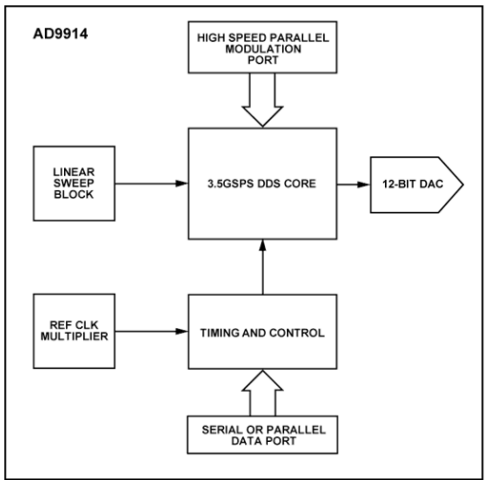

Figure 3 AD9914 Functional Block Diagram [4]

The AD9914 offers a controllable, high bandwidth, highly stable LO signal for a reasonable price. The final design will see the reference clock input come from the labs master oscillator at 2998.5MHz which of course means the maximum output frequency of the board is limited to this frequency. This is not a problem however, as such a high output frequency is not required. One major advantage of the AD9914 is that it can be controlled serially using the SPI protocol, a parallel programming mode or by using Analog devices software over a USB link.

4.2 Raspberry Pi

The Raspberry Pi is a versatile, compact inexpensive single board computer only costing around £35 for the Pi3 used in this project. The Raspberry Pi has a 40 pin GPIO along with SPI and I2C communication busses as well as the ability to communicate on networks via ethernet or wirelessly. Coupled with the large amount of online user support and previous projects and EPICS support, it was clear at the time of selection that the Raspberry Pi 3 was an excellent choice as a middle controller board with the job of taking information from a larger network via EPICS and communicating that information to the four evaluation boards. The Pi 3 specifically has been used in EPICS projects before such as in these two papers [5] [6] using both SPI and I2C communication methods as well as full control of the GPIO in EPICS. Given the precedent, the low cost and wide functionality, the Pi 3 was chosen as the middle board for this project.

4.3 Design Idea

Since the Raspberry Pi has been successfully integrated into an EPICS control system using the SPI protocol before and the AD9914 boards can be controlled with SPI the idea was to use the RPi to control the boards with this protocol. The logic of the system is as follows: the Raspberry Pi operates as an EPICS IOC on the wider network, a command from any authorised PC is received to change the frequency of one board, the Raspberry Pi interprets this data and communicates the frequency change to the AD9914 in question over SPI.

5 SPI

5.1 What is SPI?

SPI or serial peripheral interface is a serial communication method usually used to communicate with devices such as sensors from devices such as microcontrollers. SPI can operate in a three or four wire mode but the only one that will be explained here is the four-wire mode.

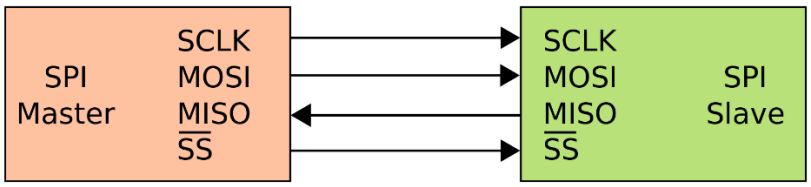

Figure 4 SPI Diagram [7]

The four lines of SPI communication are labelled as follows:
SCLK – Clock
MOSI – Main out secondary in
MISO – Main in secondary out
CS – Chip Select

SPI is called a synchronous communication protocol which means the data transfer is synchronised with a clock signal [8]. The device that generates the clock signal is labelled as the main device and all subsequent connected devices are called the secondary devices [8]. The MOSI line is data transmission from the device generating the clock to the secondary devices and the MISO line is data transmission from a secondary back to the main usually in response to stimuli from the main device. Chip select is a signal sent from the main device to open or close communication with a secondary device. By default, the chip select line is active low so for example, if there are two secondary devices and the main device only wants to send data to one of them, the main would pull a chip select for secondary one down, indicating communication start, and leave CS for secondary two high. This means that even if a device receives data on the CLK and MOSI lines, if the CS is still high

the device remains ‘deaf’ to the communication. This does require an individual chip select line for each secondary device however.

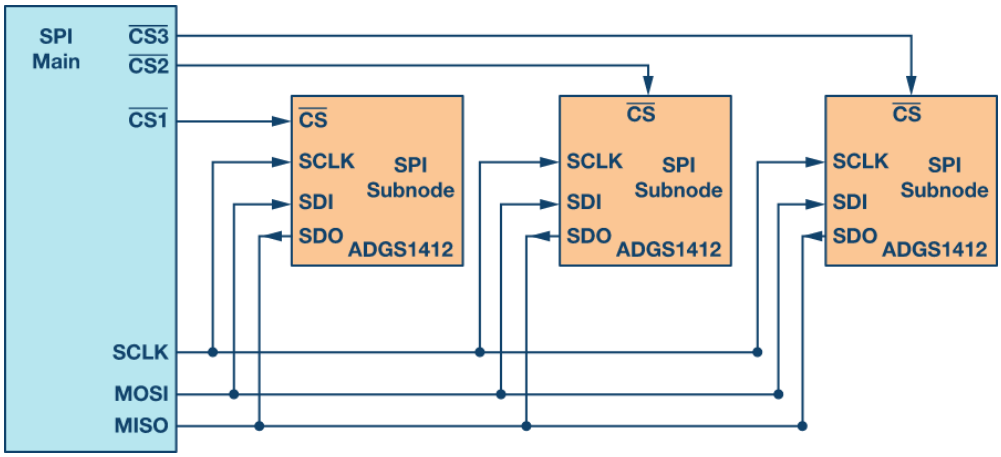

Figure 5 Multi Secondary SPI Network [8]

A few more points about SPI communication are: SPI allows for simultaneous reading in on the MISO and writing out from the MOSI line and the clock edge “synchronizes the shifting and sampling of the data” [8].

5.2 Modes of SPI Communication

There are four modes of SPI communication that controls the polarity and phase of the clock signal depending on what the secondary device can accept. The AD9914 can only communicate with SPI mode zero and so that mode will be explained here.

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State
0	0	0	Logic low

Figure 6 SPI Mode 0 Table [8]

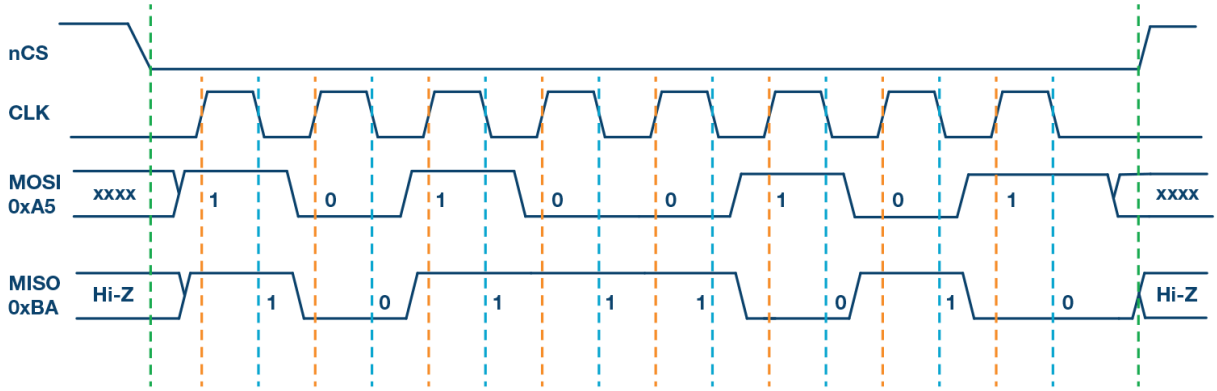

Figure 7 SPI Mode 0 Timing Diagram [8]

Figure seven shows one full communication cycle for SPI mode 0. The idea of the communication is as follows: Cs line goes low to start communication, CLK goes from low to high to send or receive

information, data is sent out on the MOSI line on the rising edge of the clock, data is received on the MISO line on the falling edge of the clock and the communication ends once the CS line goes high.

5.3 Why was SPI Chosen for this Task?

As previously stated, the Raspberry Pi has been used in EPICS systems with the SPI protocol before and the AD9914 has the ability to communicate via SPI. Another reason was that, for testing purposes SPI has many drivers and libraries like for example spiDev for Python and so code for this communication method could be very quickly developed unlike the other methods. Given the short duration of this project, the time factor was a large consideration. Another large advantage is the easy implementation of multi secondary device communication networks. As shown in figure 5 SPI can be configured to have one main device and many secondary devices so long as each secondary device has its own chip select line. The idea for the system is to have one Raspberry Pi acting as a main and four AD9914 boards acting as secondary devices so the ease of implementation was another large advantage here. The Raspberry Pi has a 40 pin GPIO which can be used for additional chip select lines and is therefore made for this type of configuration. Overall, SPI was chosen because it is a simple and effective communication method that fits the task well.

6 How to Communicate with the AD9914

6.1 Overview

The AD9914 evaluation board has multiple modes of operation as well as multiple methods of control. For this project the single tone mode operating out of one register is all that is required. The method of communication chosen will be using a Raspberry Pi 3 and the SPI communication protocol.

6.2 Hardware Setup

To set up the AD9914 to communicate in this manor some physical pins on the board must be set using jumper pins. To start, P205, P203 and P204 must be set to disable to tell the DDS board not to communicate over the USB port and to instead expect communication via the parallel port. Next, there are four function pins called IOCFG0-3 or pins one to four on the parallel interface. For serial communication they must be set in the manor of 0001 or IOCFG0 set high and the rest low. The P202 pin must be connected also, this comes as standard out of the box. The next thing to do is to short the external power down pin (P102 first pin) to ground. This stops the device from shutting down by default. Next is to set the register to read from for single tone mode. For this work the register selected was register 11 which corresponds to shorting all of the pins to ground (P102 from 8 to 10 or PS0-BUF to PS2-BUF).

This amounts to all of the physical pin setup. The final hardware setup is to connect the SPI interface. The SPI interface lies on the final four pins of the parallel port. They are as follows: CS = MPI00, SCLK = MPI01, MOSI = MPI02 and MISO = MPI03. Aside from the SPI interface, three more pins also require inputs. They are reset, I/O update and I/O sync and they are on the parallel port numbered 5, 6 and 7 NOT MPI05 etc. The equivalent pin numbers for the SPI pins are 40, 39, 38 and 37.
For any further hardware setup such as the RF I/O or power, please refer to the data sheet [9].

6.3 Serial Communication

Please note, all information in the following section is referencing the data sheet provided by analogue devices found here [9].
The serial communication with the DDS board is very specific with regards to its timing. Firstly, each register contains four bytes of information and for the purposes of this work, when communicating one must send the full four bytes of information or the timing will be out. For simplification in future the main device will be the Raspberry Pi and the secondary device will be the DDS board. The main

device needs to send an instruction byte to the secondary to initiate communication. From this, the secondary discerns whether it should be reading or writing.

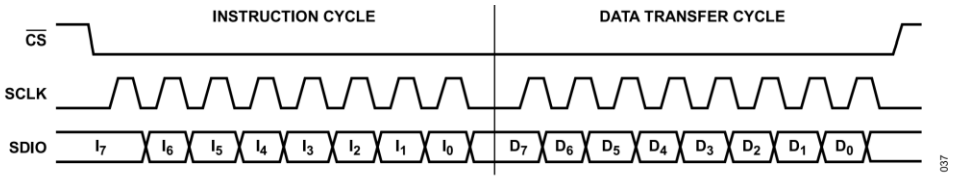

Figure 43. Serial Port Write Timing, Clock Stall Low

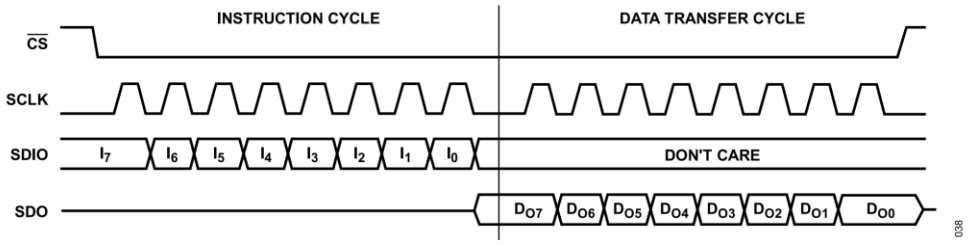

Figure 44. 3-Wire Serial Port Read Timing, Clock Stall Low

Figure 8 Serial Read/Write Cycle [9]

The write cycle is as follows: chip select goes low, the first byte of the instruction byte tells the chip if its reading (logic 1) or writing (logic 0). The following bits of the first byte tell the chip from which register to operate from. The main device then sends four bytes of data which the secondary stores in a serial buffer. A high pulse on the I/O update pin is required to move the data from the serial buffer into the register. The chip select then goes high to end communication. Note, the communication style is that of SPI in mode 0, please refer to the earlier section on SPI communication for further information.

The read cycle is as follows: chip select line goes low, the first bit of the instruction byte is a one and the following bits dictate the address of the register to be read from. The secondary receives this information and relays the information via the MISO line back to the main device. Chip select line goes high to end communication. Note, the I/O update pulse is not required for a reading operation because the secondary sends information directly from the registers, bypassing the serial bus.
A few more things to note are as follows: on the write cycle the information is sent out on the leading edge of the clock and when reading on the falling edge of the clock. The timing is critical, if the chip select goes high or low earlier than expected, this can cause the timing to be such that the device will no longer communicate until reset.

6.4 Setup for Single Tone Mode

Please note, all information in the following section is referencing the data sheet provided by analogue devices found here [9].
As stated before, the AD9914 has multiple modes of operation and for this work only the single output tone is required. Which mode the board is operating in is controlled by the first four registers known as the control function registers. The data that is in these registers determines the mode of operation. Therefore, to set up the board to operate in single tone mode one must set these registers with the correct information. Before that however, the board must have all of its registers cleared to remove any spurious data lingering. This is done with a short reset pulse. Next, an I/O

sync pulse must be performed. I/O sync tells the board that the next incoming byte is an instruction byte not data and is required on start up to confirm the timing. However, it is not required after the initial setup so long as the timing remains correct with the information stated in the previous section.

After resetting and syncing the board one can then send data to the secondary. The first register that should be set is address four or 0x003 to set the DAC calibration. The data sheet states the DAC must be calibrated each time the board is powered off or when the internal reference clock is changed [9]. The calibration optimises the setup and hold times for the internal DAC timing [9]. The fourth register has one bit inside which initiates this calibration. Essentially the information for the fourth register must be sent with the DAC calibration bit turned high for the duration of $tCal = \frac{469632}{fs}$ and then the information inside the fourth register must be sent again with the calibration bit set to zero. This concludes the calibration step and has now set the fourth register with its required data. The fourth control register has no more controls of interest linked to it, most bits are open or set specifically by the manufacturer. The full information of what is in each control register will be conveyed at the end.

After the calibration the remaining information in the control registers should be sent. Control register one contains the most important information. Bit zero should be set to zero to allow for MSB first when communicating serially. Bit one must be set as 1 to enable four wire mode. Initially, the AD9914 is set to communicate in three wire serial mode. Essentially, this is SPI but instead of the MISO line being the read back line, the MOSI line writes to and reads from the secondary device. This is a feature not supported by the raspberry pi. It is therefore important that this bit be set high to enable four wire SPI communication.

The next bit of mention is in control register 2 which has its bit 23 dictate weather or not profile mode is activated or not. This is the bit that sets the board operating in its single tone out mode and so should be set high. The board will only change its operating register by changing the external pins. The remaining control bits are unimportant and mostly consist of open, manufacturer set not to be changed or default values which could be changed but are unneeded.

CFR1 Address 0x00 – 0x00,0x00,0x01,0x00,0x0A
CFR2 Address 0x01 – 0x01,0x00,0x80,0x90,0x00
CFR3 Address 0x02 – 0x02,0x00,0x00,0x19,0x1C
CFR4 Address 0x03 (DAC Cal enable) – 0x03,0x03,0x05,0x21,0x20
CFR4 Address 0x03 (DAC Cal disable) – 0x03,0x00,0x05,0x21,0x20

6.5 Changing the Frequency

With the AD9914 now operating in single tone mode with the external pins setting the output profile to read from register 11, simply by changing the value inside register 11 will change the output frequency. One thing to mention is that the AD9914 requires a reference clock to operate and for this project this comes from a master oscillator operating at 2998.5MHz. For profile mode, the exact frequency is not stored inside the register and instead a frequency tuning word or FTW is stored inside the register. Essentially, the direct digital synthesis works by storing this tuning word inside of the active register where the board will then take it and use the following equation to produce a correct output frequency $Fout = \left(\frac{FTW}{2^{32}}\right) * Fsysclk$. The intricacies of the direct digital synthesis itself will not be explained here and for simplification the board takes the FTW and uses the equation to convert to an output frequency.

Therefore, to set the board to a desired frequency one must send the appropriate tuning word to the active register. The reverse of the previous equation then is $FTW = round\left(2^{32} * \left(\frac{Fout}{Fsysclk}\right)\right)$.
Note, the equation is rounded because only an integer can be stored inside of the register.
The idea then is that the output frequency is input into the FTW equation and then the generated FTW is sent via SPI to the AD9914 where it will produce the desired frequency via direct digital synthesis.

7 Python Code and Single Board Connection

7.1 Single Board Testing

As stated before, the final system aims to have one Raspberry Pi control four AD9914 boards in the single tone output mode via SPI. The previous section detailed the theory of communicating to such a board via SPI. The first prototype of the system involved controlling one AD9914 board using a Raspberry Pi and Python. The idea being to test the theory with a simpler setup that could be easily debugged verses a four board EPICS approach.
The test rig was simple, connect the relevant pins on the AD9914 to the SPI 0 bus pins on the Raspberry Pi and use a Python library called spidev to send and receive the SPI information. The test would be to confirm that the frequency of one AD9914 could be reliably changed with this method.

7.2 Python Code and Test Rig

```
import spidev
import time
import binascii
import RPi.GPIO as GPIO

spi = spidev.SpiDev()
spi.open(0, 0)
spi.max_speed_hz= 10000000
spi.threewire = False
spi.mode = 0

reg = bytearray(b'\x80\x81\x82\x83\x8B\x8C')

GPIO.setmode(GPIO.BCM)
GPIO.setup(17,GPIO.OUT)
GPIO.setup(27,GPIO.OUT)
GPIO.setup(22,GPIO.OUT)

def I_O_Update():
        GPIO.output(22,GPIO.HIGH)
        time.sleep(0.001)
        GPIO.output(22,GPIO.LOW)
        time.sleep(0.001)
        #IO Update Pulse

def start_up() :
        GPIO.output(17,GPIO.HIGH)
        time.sleep(0.001)
        GPIO.output(17,GPIO.LOW)
        #Reset Pulse
        #time.sleep(0.001)

        GPIO.output(27,GPIO.HIGH)
        time.sleep(0.001)
        GPIO.output(27,GPIO.LOW)
        #IO Sync Pulse

        spi.writebytes([0x03,0x03,0x05,0x21,0x20])
        #DAC calibration

        I_O_Update()
        #I/O Update Pulse

        spi.writebytes([0x03,0x00,0x05,0x21,0x20])
        #DAC calibration clear bit

        I_O_Update()
        #IO Updtae Pulse

        spi.writebytes([0x00,0x00,0x01,0x00,0x0A])
        #Write register 00

        spi.writebytes([0x01,0x00,0x80,0x09,0x00])
        #Write register 01

        spi.writebytes([0x02,0x00,0x00,0x19,0x1C])
        #Write register 02

        spi.writebytes([0x0B,0x0F,0x1C,0x8E,0xEB])
        #Write register 0B

        I_O_Update()
        #IO Updtae Pulse
```

```
def read_back() :
    int = i = 0
    for x in reg:
        spi.writebytes([reg[i]])
        print(reg[i])
        resp=spi.readbytes(4)
        print('Received 0x{0}'.format(binascii.hexlify(bytearray(resp))))
        print(resp[0:5])
        i=i+1

def input_frequency() :
    while True:
        frequency = float(input("Please enter a frequency in MHz "))
        frequency = frequency * 1000000
        FTW = int(round(2**32*(frequency/((2998.5*10**6)))))
        FTW_hex = hex(FTW)
        print(frequency,FTW,FTW_hex)
        print(type(FTW_hex))
        FTW_bytes = FTW.to_bytes(4, 'big')
        print(FTW_bytes)
        b = list(FTW_bytes)
        print(b)
        c = bytes(b)
        print(c)

        k = int(0)
        spi.writebytes([0x0B])

        for x in FTW_bytes:
            spi.writebytes([c[k]])
            k=k+1
        I_O_Update()
        read_back()

try:
    start_up()
    GPIO.output(27,GPIO.HIGH)
    time.sleep(0.001)
    GPIO.output(27,GPIO.LOW)
    #IO Sync Pulse
    read_back()
    input_frequency()

finally:
    spi.close()
```

Figure 9 Single Board Python Control Code

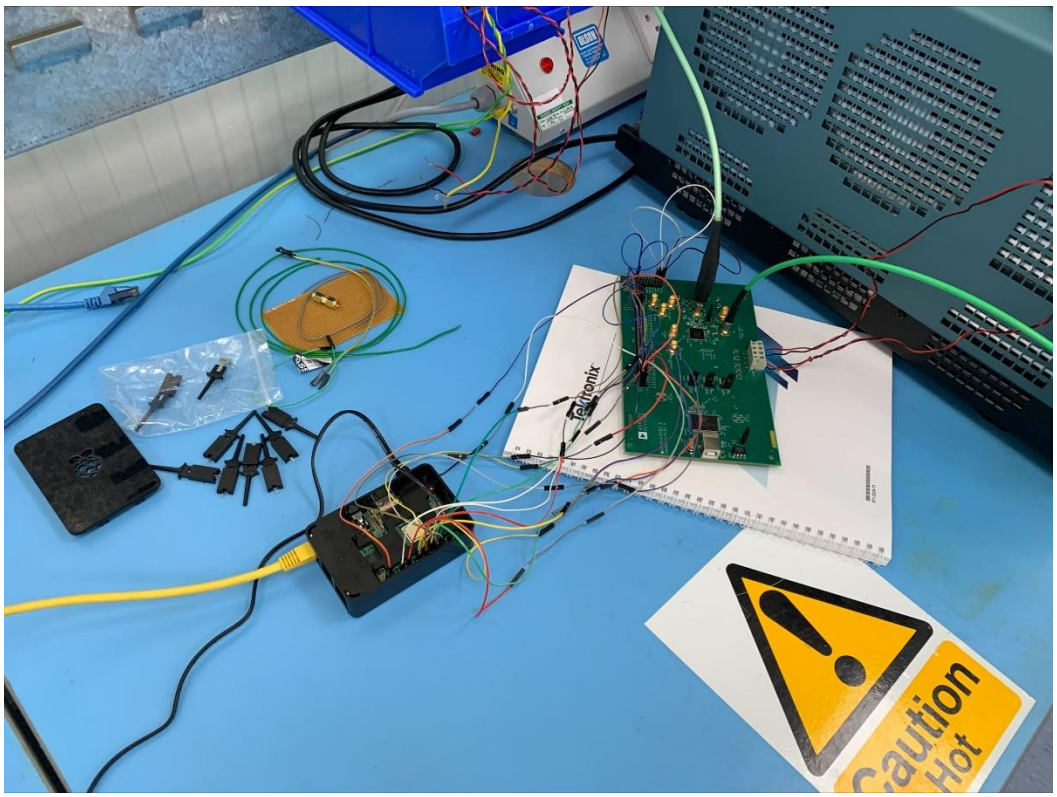

Figure 10 Single Board Test Rig Connection

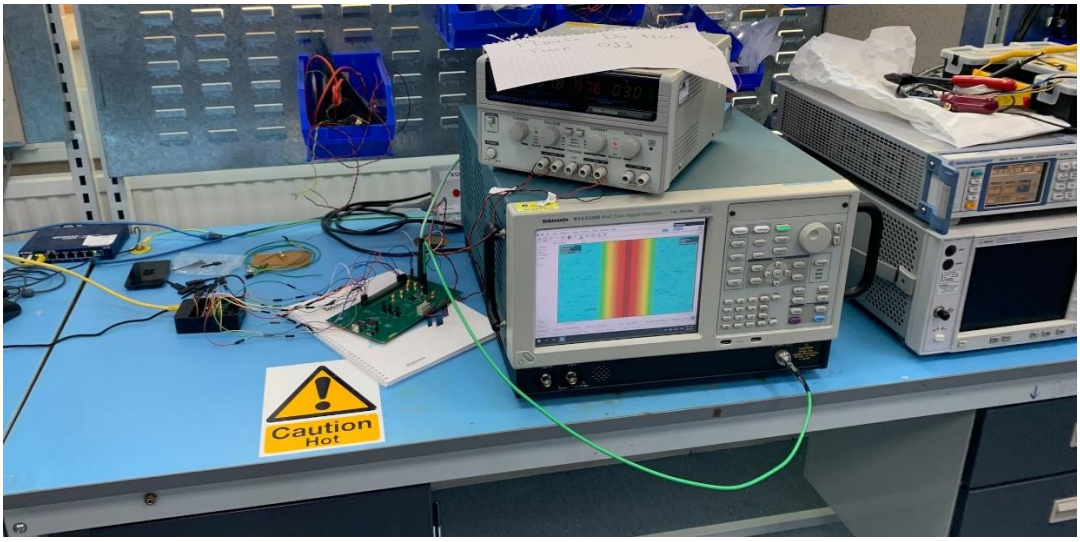

Figure 11 Single Board Test Rig

The Python SPI code to control one AD9914 can be seen in figure 9. The first section of the code opens communication with SPI bus 0 on the Raspberry Pi and sets up three additional GPIO pins for the control of the Reset, IO update and sync pins on the AD9914 not controlled by SPI. A byte array is also declared which contains the addresses of the four control registers and the register which controls the current frequency. The I/O update function simply sends a pulse to the I/O update pin on the AD9914 whenever it is called. The start up function performs the start up task described in section 6.4 to set the AD9914 up for single tone output mode. The spidev function SPIwritebytes and SPIreadbytes open the SPI communication line and either sends the data in the function or reads the data back for however many bytes it has been set to receive. The writebytes function will leave the chip select line low until it has sent all of the data stored in the function verses pulsing the CS line high and low after each byte.
The read back function performs the read operation also described in section 6. It writes one instruction byte with the addresses stored in the byte array in the beginning to tell the AD9914 which address is being read from. The code then opens the SPI communication and waits for the four bytes stored in the register it is trying to read from. The read function reads from all of the control registers and the register that contains the FTW to ensure the correct data is contained in each register.
The input frequency function allows a user to input a new frequency to the board in MHz from which the code will calculate an appropriate FTW to send to the register. It converts this value into four bytes and stores it in a byte array. The code then writes the instruction byte telling the AD9914 that the next operation is writing to register eleven, the register responsible for frequency change. The function then sends the FTW stored in the byte array, sends an I/O update pulse to move the data from the serial buffer to the register and then reads back the data to ensure a successful transfer. The try function simply calls the previous declared functions and ‘finally’ closes the SPI connection when the program is terminated.

7.3 Robustness Test

When the Raspberry Pi was chosen for the task of controlling the AD9914 boards some anecdotal evidence was put forward by my supervisor stating that he had issues with the Raspberry Pi dropping communications or simply crashing when attempting to do similar control jobs in the past. This was most likely down to for example, an undervoltage or poor network connection, however, should the Raspberry Pi be used in this project its reliability had to be tested.
Thusly, a robustness test was developed to change the output frequency of the AD9914 from 177MHz to 200MHz and back for twenty-four hours straight. This should allow any obvious reliability issues to surface and be identified. The Raspberry Pi was set to count the number of times it

changed the frequency from 177MHz to 200MHz and back calling this one full cycle with the code setting the time it takes to complete one cycle equal to 20 seconds.

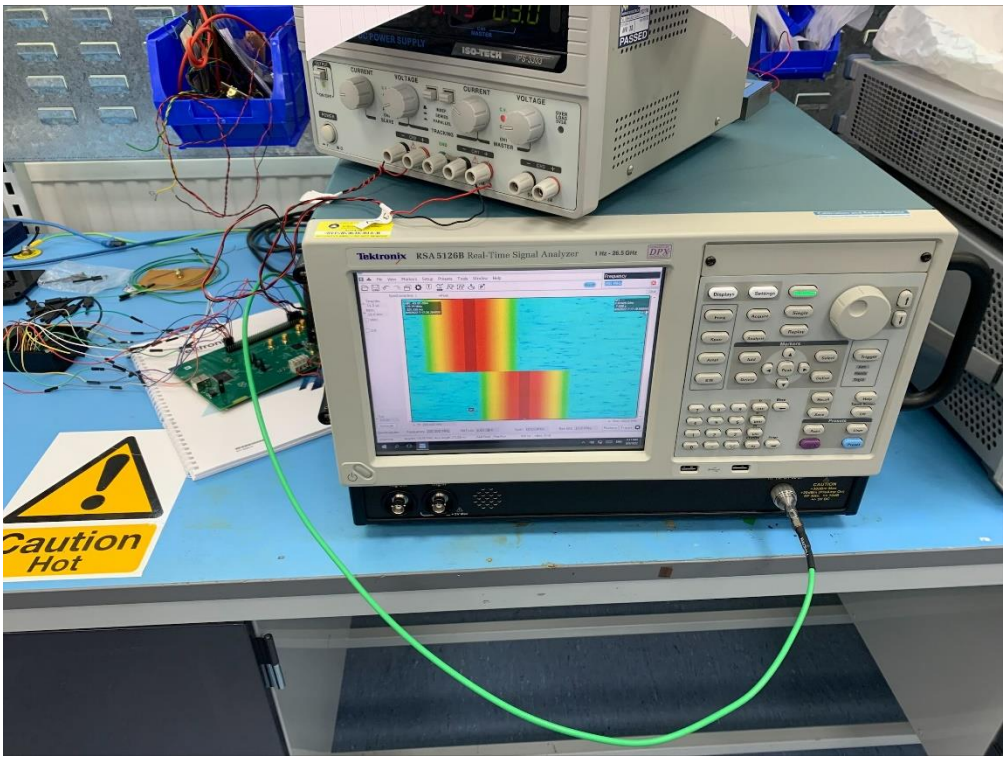

Figure 12 Robustness Test

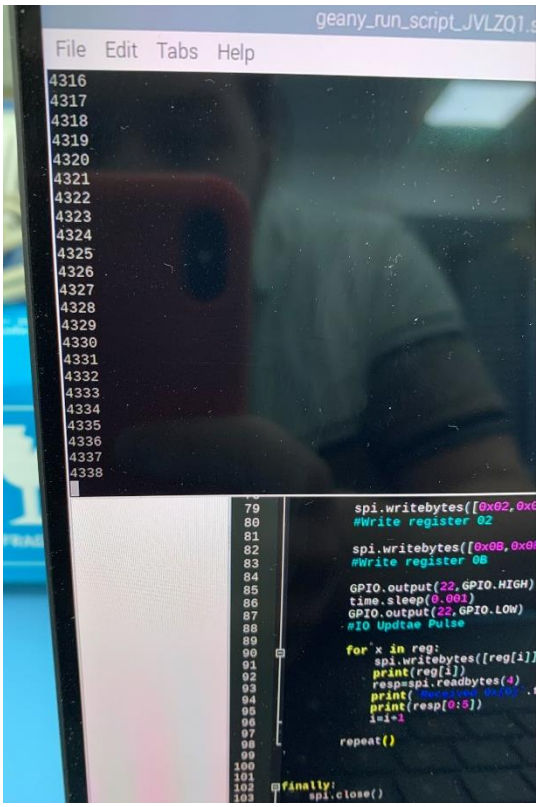

Figure 13 Robustness Test Final Cycle Count

As can be seen from figure 13 the final cycle count was 4338 cycles. Therefore, the runtime was $Runtime\ in\ Hours = \frac{4338\ cycles * 20\ \text{seconds}}{60} = 24.1\ hours$. This confirms the Raspberry Pi ran the code for over 24 hours without any errors occurring.
As stated, this would have triggered any immediate reliability issues however, more niche ones would not be found with this method. A future test to 100% confirm the reliability of the Raspberry Pi for this use would be to monitor the temperature of the Raspberry Pi and other parameters and run the test over a much longer period of time. This would yield much more data with which to make an accurate assessment of the Raspberry Pi's robustness. However, for this stage of the project the Raspberry Pi has proven its reliability enough to progress it to the next stage.

7.4 Why this Method is Not Suitable for the Final Design

The approach shown here is much simpler than the final design and so the question of why not use this approach for the final design may arise. The main reason is this code could not easily be embedded in an EPICS control network. Granted, the Python library PyEPICS does exist which would allow some degree of integration into an EPICS system however, and indeed the library will be used as a sequencer later on, although the aim is to remove it entirely for the final system for a lower-level EPICS approach. The main issue is that EPICS would be communicating to another program to use the GPIO instead of having access to the GPIO itself. This means if a successor to this project wanted to make changes or add functionality, they would need to edit the code in Python. However, giving EPICS control over the GPIO and setting it up as an IOC means one can simply delete the current sequencer and rewrite it in any language suitable, connect it to the IOC and have full control over the GPIO. One can move the sequencer inside or outside the IOC, have multiple sequencers in different languages access the IOC for different functionalities. Essentially, it makes the system highly customisable and very easy to add functionality in a variety of ways.

8 EPICS and the Four Board Connection

8.1 What is EPICS and why is it Necessary?

EPICS or experimental physics and industrial control system is "a set of software tools and applications which provide a software infrastructure for use in building distributed control systems to operate devices such as Particle Accelerators, Large Experiments and major Telescopes" [10]. The use for EPICS is extremely large and can be used for tasks such as: closed loop feedback control, data analysis, access security and modelling and simulation among many others [11]. EPICS control networks consist of many devices over one network using a client and server based system.

The basic structure of an EPICS network contains three major components being the IOC (input output controller), CWS or client workstation and a LAN or local area network [11].

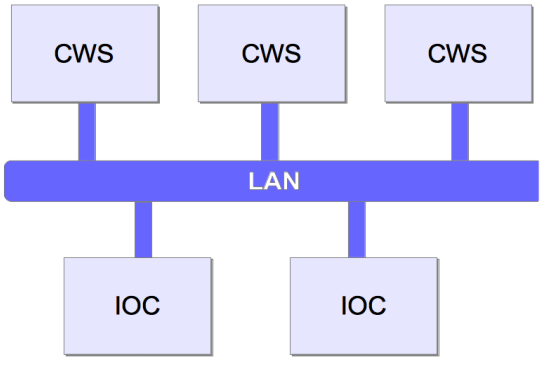

Figure 14 EPICS Control Structure [11].

The idea of the EPICs network is that the IOCs act as the server components of the network and the CWSs act as clients with the LAN facilitating the communication. IOCs contain the databases and device support to perform an operation in response to a stimulus, usually connected to another device such as a sensor. The CWS is usually what provides the stimulus to the IOC to stimulate and action or a response. An IOC can be placed on a regular PC however, it is more often installed on devices such as the Raspberry Pi. The CWS on the other hand is more than likely a desktop PC running a consumer operating system. This distributed network allows for easy additions of IOCs or CWSs to the network allowing for easy expansion or reduction of systems and system functionality. STFCs Daresbury lab has an EPICS control network to control the CLARA system and so for consolidation purposes and for all of the benefits EPICS provides, the control system will be built as an EPICS IOC for an easy addition to the existing network.

8.2 What is an EPICS IOC?

An EPICS IOC contains multiple components that will be explained in this section and throughout the rest of the paper.

The IOC database contains a number of records which in essence dictates the whole functionality of the IOC. The records define what type of information is incoming and outgoing, what to do when a certain piece of data is sent or received, what limits and data type should be contained in one record or another and weather or not to raise an alarm flag in response to an event. The record in the database must each have a unique name across not only the IOC but the whole TCP/IP subnet because of the communication style. The concept of linking records is also apart of the database. If for example when one record changes and there is a desire to update another record with a modified version of the record that has changed, a link can be used to send the received information to a record that will modify it (CALC record) and then link that information to the output record. By using linking and specialised records such as CALC records, a form of programming logic can be achieved, although to a limited capacity [11].

Another topic to discuss is database scanning. A record being processed means the record performs its task and if it is linked to any other records sets off a chain of commands. A record can be processed as a result of a CWS changing the value inside of the record, but it can also be processed by scanning. The scan field is a condition on which the record will process. This can be periodic, when an event occurs, or passive. Passive scan is the most important because if a record is set to passive scan that record will process after another record linking to it processes. The logic is as follows: CWS changes the information inside of a record, record processes, the processed record has a forward link to another record with a passive scan, that record also now processes.

Device support is a device specific driver allowing EPICS access to hardware functionality of the computer it is running on.

The network protocols such as channel access are how the EPICS IOC communicates with the CWS over the LAN. This will be explained in a later section.

Further information can be found here [11].

8.3 Four Board – IOC Connection Design

After confirming the SPI communication method between the Raspberry Pi and the AD9914 the next part of the project was to be able to control four AD9914 boards with the RPi using EPICS. The Raspberry Pi will be acting as an EPICS IOC connected to the four AD9914 boards. The idea of the system is that a CWS will send a request over the LAN to change the frequency of one AD9914 and the Raspberry Pi will respond to that request saying it contains that record. It will then use a sequencer (another part of EPICS that sequences complex events) and relevant device support to determine the information to send to the board in question and then send the SPI information over the RPis GPIO. It will then read back the current value of the active frequency register to confirm the successful write operation to that specific board. The CWS will be able to confirm by checking the read record of that board.

9 IOC, Device Support and How to Install

9.1 Overview

The IOC itself uses two main device support packages named epics-devgpio [12] and drvAsynSPI [13]. The first allows EPICS to control the GPIO pins on the Raspberry Pi and the second allows EPICS to send information over the Raspberry Pi's SPI bus. For the IOC to work, EPICS, Stream Device and asyn must be installed on the Raspberry Pi.

9.2 EPICS, Stream Device and asyn Installation

Firstly, EPICS and the two support packages Stream Device and asyn must be installed on the Raspberry Pi. The installation process of EPICS on the Raspberry Pi is fairly straightforward and one can use the following guides for this part [14] [15]. This work uses a Raspberry Pi 3 as the controller and since the RAM on this board is somewhat limited there is one major issue with EPICS installation. The issue is that, when installing EPICS, the Pi will get so far through installation and then run out of memory before finishing. To prevent this issue from occurring, the Swap file on the Raspberry Pi must be edited to allow the Pi to use half of its total RAM capacity, a guide for this can be found here [16].

After installing EPICS on the RPi and running the softIOC test in the guide to confirm installation, the next step is to install the asyn package. Note, one must install asyn first as Stream Device is dependent on asyn. Asyn is a required module for the device support to work and can simply be installed using the previous two guides.

Once EPICS base and asyn have been installed, Stream Device is the next required package. The Paul Scherrer Institut says Stream Device is 'generic EPICS device support for devices with a "byte stream" based communication interface. That means devices that can be controlled by sending and receiving strings' [17]. Stream Device is used for the SPI device support and facilitates the data streaming. Note, the previous quote states Stream Device communicates by sending and receiving strings, the strings by default are in ASCII format. To install Stream Device the following guide states the correct steps [18]. The only difference is that one must rename the directory of the Stream Device support package to 'stream' because by default, the make files point to a directory called 'stream' and should it be named something different when installing, the file won't be able to locate the support package. Of course, the make files could be edited however, there are many instances where the Stream Device support package directory is called 'stream' and so it is much easier to simply rename the directory than it is to hunt for every instance of this.

9.3 devGPIO and drvAsynSPI Installation

Before continuing, it should be noted here that these device support packages have been installed on a 64-bit Raspberry Pi 3 and that this install method may vary depending on the hardware and OS type. As of the time of this paper, the most reliable version of the devGPIO device support is version R1-0-6 [19] and explicitly states its support for the Raspberry Pi 3. There is a newer version of the device support [12] however, it proved difficult to get working. It may be a better device support to use however, if one decides to use a board not explicitly stated on the support documentation for the previous version.

After downloading the device support, place the file in the support files folder then go to the root of the devGPIO folder. Navigate to the congfigure/RELEASE file and change the EPICS/base path to where it is installed on the device. Return to the top directory and run 'make'. The devGPIO device support has now been installed.

Much like the previous device support, drvAsynSPI should also be downloaded and placed inside of the support folder for EPICS. Edit the configure/RELEASE file and change the EPICS/base path and asyn path to where it is installed on the device. Before making the device support, a patch file which

can be found here [20] must be installed. By default, there is a problem with the read and write handler of Stream Device's asyn interface and this patch fixes these issues. It comes in a .patch file format which is supposed to be an automated patching process in Linux, for which a guide can be found here [21]. However, during my own installation I found the files the .patch file was looking to edit were in different locations than where the code was trying to point it to. As a result, I manually implemented the patch by going to the specific file to be edited and copy pasted the changes across. After patching the file, return to the top directory and run 'make'. The device support is now installed.

9.4 How to use devGPIO

After making a fresh IOC navigate to the configure/RELEASE file and ensure asyn, Stream Device, devGPIO and drvasynSPI have correct paths set, pointing to their location in the support section. Also another command called 'STREAM_WORKARROUND=1' must be added underneath the device support path pointers. It simply allows for the previous patch in drvAsynSPI to work.

```
record( bo, "GPIO:P9:12:OUT" ) {
  field( DTYP, "devGpio" )
  field( OUT,  "@P9_12 H" )
}
```

Figure 15 devGPIO Record Example [22]

To create records to access the GPIO pins navigate to the db file inside of the new IOC and create a new binary output record. Binary output simply takes a 1 or a 0, for an active high record a one corresponds to a high output and a 0 corresponds to a low output. However, for readability one can add the following fields to the record 'field (ZNAM, "OFF")' and ' field(ONAM, "ON") to add names to the binary states. For example, now "ON" corresponds to a 1 and a "OFF" corresponds to a 0 in the binary record. The data type field must be declared as the devGPIO type to declare the record is trying to access the GPIO device support. The device support does support an input field however it is not used here. To output information over the GPIO a field with an OUT link must be specified along with the "@pin active" [22]. The pin pointer can be of the form name ("GPIO1_28"), key ("P9_12") or the number of the pin which is the approach used when writing the records in this work ("17"). The final part of this field is to declare whether the pin is active high "H" or active low "L". Figure 15 shows an example of a record which is active high and controlling the pin with a key of "P9_12". Only binary in and binary out are supported for this version and this work will only use binary out.

After creating the Db file, navigate to "The Name of Your IOC/App/src. Now edit the makefile to add the following libraries and database definitions [23]:

```
DBD += YourIOCName.dbd

# YOURIOCNAME.dbd will be made up from these files:
YourIOCNAME_DBD += base.dbd

# Include dbd files from all support applications:
#YOURIOCNMAE_DBD += xxx.dbd
YOURIOCNAME_DBD += asyn.dbd
YOURIOCNAME_DBD += stream.dbd
YOURIOCNAME_DBD += devgpio.dbd
YOURIOCNAME_DBD += drvAsynSPI.dbd

# Add all the support libraries needed by this IOC
#YOURIOCNAME_LIBS += xxx
YOURIOCNAME _LIBS += asyn
YOURIOCNAME _LIBS += stream
YOURIOCNAME _LIBS += devgpio
YOURIOCNAME _LIBS += drvAsynSPI

# YOURIOCNAME _registerRecordDeviceDriver.cpp derives from SPI_test.dbd
YOURIOCNAME _SRCS += SPI_test_registerRecordDeviceDriver.cpp

# Build the main IOC entry point on workstation OSs.
YOURIOCNAME _SRCS_DEFAULT += SPI_testMain.cpp
YOURIOCNAME _SRCS_vxWorks += -nil-

# Add support from base/src/vxWorks if needed
# YOURIOCNAME _OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary

# Finally link to the EPICS Base libraries
YOURIOCNAME _LIBS += $(EPICS_BASE_IOC_LIBS)
```

Figure 16 "The Name of Your IOC/App/src" makefile

Now navigate to the TOP of the IOC directory and 'make' the IOC. If the IOC has been made correctly then new folders in the directory should have been made. Navigate to iocBoot/iocYOURIOCNAME and edit the st.cmd file. Inside the st.cmd file add the following lines:

```
## Load record instances
#dbLoadRecords("db/xxx.db","user=greenbt2")
GpioConstConfigure("RASPI B+")
drvAsynSPIConfigure( "SPI", "/dev/spidev0.0", 0, 10000000, 1 )
dbLoadRecords( "$(TOP)/db/YOURIOCNAME.db","head=YOURUSERNAME" )
```

Figure 17 iocBoot/iocYOURIOCNAME ./st.cmd

The GPIOConstConfigure function loads the GPIO lookup table for a specific board. The documentation states only three boards the: Raspberry Pi 3, Raspberry B REV 2 and the Beaglebone Black are compatible with this device support however, since the Pi 3 and Pi 4 have the same GPIO layout it may also work on this version of the device support, although this is untested. The function underneath is for the following device support. The final function adds a macro called 'head'.
To activate the GPIO pins, send a channel access command 'caput' to the device once the IOC is running along with the name of the record and a "ON" or "OFF" to change the state of the pin named in the record.

9.5 How to use drvAsynSPI

As stated before, this device support uses a basic 'Stream Device template' in terms of how it works. Another point to reiterate is that this device support is based off another one by the name of drvAsynI2C which has the exact same setup as this one and so guides linking to this will be referenced here. This device support requires the user to create a 'protocol' file in the top of the IOC directory. A protocol file "contains protocols for each function of the device type and variables which affect how the commands in a protocol work. It does not contain information about the individual device or the used communication bus" [24]. A protocol is in a way like a C function which can be called by a record in the database when that record is processed. The subject of protocols is wide so this work will only cover the relevant sections.
The first part of the protocol file contains system variables which influences the behaviour of the input and output commands [24].

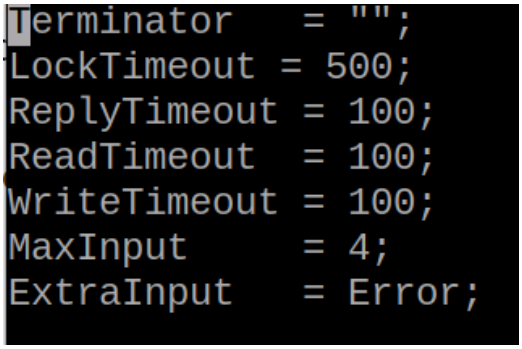

Figure 18 System Varaibles

Figure 3 contains the system variables used in the final IOC and the following descriptions come from the following source [24].

Terminator – Some devices expect a terminator after each communication cycle and by default, Stream Device adds one at the end of an output. The AD9914 requires no such terminator and so it has been set blank.

Lock Timeout – If the device is busy with other records, how many milliseconds will the record wait for exclusive access to the device before aborting operation. In other words, if a record attempts to run a function inside the protocol file before the device has finished executing its previous function, how long should it wait before aborting. In two example protocol files [25] [26] it states the default is 500ms which is what is used in the final IOC.

Reply Timeout – How long to wait for the first byte of a response from an input device before aborting operation. This must be less than the lock timeout because other records may ask to use functions inside of the protocol file before the device has responded and thus should be aborted.

Read Timeout – Such as is the case for this device, some protocol functions may receive information in multiple bytes over the same communication cycle. This value sets the maximum time the system will wait to read information between breaks in communication before aborting the read cycle.

Write Timeout – If the output can't be written immediately, how long to wait to send the information before aborting the cycle.

Max Input – How many bytes to read before terminating the input. Here, the AD9914 has a register size of four bytes so the system has been set to expect four bytes before terminating the input communication.

Extra Input – Should any extra inputs be received; they will be treated as a parsing error.

With the system variables now describing how the device should be read from and written to, the next section describes how to declare the protocols themselves.

```
send {
        out "%(TEST:BYTE1.VAL)r", "%(TEST:BYTE2.VAL)r", "%(TEST:BYTE3.VAL)r"
        "%(TEST:BYTE4.VAL)r", "%(TEST:BYTE5.VAL)r";

}


read {
        out 0x8B;
        in "%04r";
}
```

Figure 19 Protocol

Stream Device is capable of holding data or pointing to data inside of other records inside of its protocols. In figure four two C like functions can be seen in the form of 'send' and 'read'. These functions will be called when a record pointing to them is processed. Looking at the first function 'send' it has the syntax: out "% record value r". This means when send is called, it will output the value inside of whichever record it is pointing at. The 'r' at the end of the command stands for raw data. Because stream device by default looks to send and receive ASCII data, to send the raw binary data of a record (stored as a decimal integer in the record from which the raw format converter converts it to binary) the 'r' format converter is required at the end of the command. Since there is no limit on the number of bytes that can be sent out in one time, simply adding a comma and then pointing to a new variable will send another byte without sending the chip select line high. Looking again at 'send' this sends five bytes, each pointing to a different value, without pulling the chip select line high between bytes.

The 'read' function displays how a protocol can have both a out and an in as a part of the same function. Recall back to the previous section which explains how to read from a register, the board requires one instruction byte telling the board to send the contents of a specific register. The only register that requires a read back is register eleven which contains the frequency value which the board is operating at. This will be explained further later. Since only one register is being read from, the instruction byte is the same every time which is why the 'out' command contains a fixed hex value. This hex value contains the read bit at the front of the instruction byte and the address for register eleven. After the instruction byte is sent the AD9914 will send back the four bytes stored inside of register eleven. The in command has the format "%04r". This states the input is expecting four bytes in a raw format. The record which called 'read' automatically updates its value with the value read in from the register.

How the protocol file works with the database will be described in the next section.
One final thing to mention is in the iocBoot/iocYOURIOCNAME, edit the st.cmd file again. Inside the st.cmd file add the following lines below the GPIOConstConfig command:

```
drvAsynSPIConfigure( "SPI", "/dev/spidev0.0", 0, 10000000, 1 )
```

Figure 20 drvAsynSPIConfigure

This function has the format "drvAsynSPIConfigure("NAME", "SPIbus", spi_mode, spi_max_speed, autoConnect)". These variables will be broken down in the following section and information was taken from the following source [13].

NAME – used by Asyn to identify the driver
SPIbus – the location of the SPI bus on the Raspberry Pi, in this case SPI bus zero
SPI_mode – Mode zero is the only mode the AD9914 will accept
SPI_max_speed – Maximum clock speed of the SPI communication.
autoConnect – should the driver auto connect to the device

The final part to mention is how the records in the database link to the functions in the protocol file.

```
record(mbboDirect, "TEST:SEND"){
        field(DTYP, "stream")
        field(OUT, "@test.proto send() SPI")
        field(SCAN, "Passive")
        #field(FLNK, "READ:BACK")
        #field(DRVH, "256")
        #field(DRVL, "0")
        #field(NOBT, "1" )
}
```

Figure 21 Send Record

```
record(stringout, "TEST:BYTE1"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE2"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE3"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE4"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE5"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
        field(FLNK, "TEST:SEND" )
}
```

Figure 22 Byte Values

To link the protocol file to a record the data type filed must be set to type “stream” followed by an “IN” or “OUT” field. Since the record in figure six is linking to an output function in the protocol file the record type must be set as a multiple binary output along with a data type of “OUT”. In the field declaring the record is an “OUT” type the syntax is as follows “@PROTOCOL_FILE_NAME.proto FUNCTION_NAME() ASYN_NAME”. This links the record to the specific function inside of the protocol file along with the bus name to send or receive the data from. When this record is processed, this field links to the function in the protocol file which then sends the data. Figure seven shows the records containing byte values for the output function. Figure 4 shows how to link a protocol variable to a value inside of a record.

The process of the output function is as follows: Send record is processed, send record calls the send function inside of the protocol file, the send function pulls the byte values from the records linked in the database and sends them over the SPI bus”.

```
record( mbbiDirect, "READ:BACK"){
field(DTYP, "stream")
#field(DISV, "1")
field(INP, "@test.proto read() SPI")
field(SCAN, "Passive")
}
```

Figure 23 Read Back

To compare the input record against the output record, there is only one major difference which is that the output record holds no useful value and is only responsible for calling the function inside of the protocol file. The input record calls the 'read' function in the protocol file but also holds the value obtained from the read back operation. This is in the form of a decimal number.

10 Py EPICS, Sequencer and Final Operation

10.1 Overview

EPICS records, specifically using calc records and links, can recreate a great deal of functionality found in regular programming languages. However, when a program needs to sequence complex events with complex timing and or needs complex functions, a sequencer is the best option [27] [28].

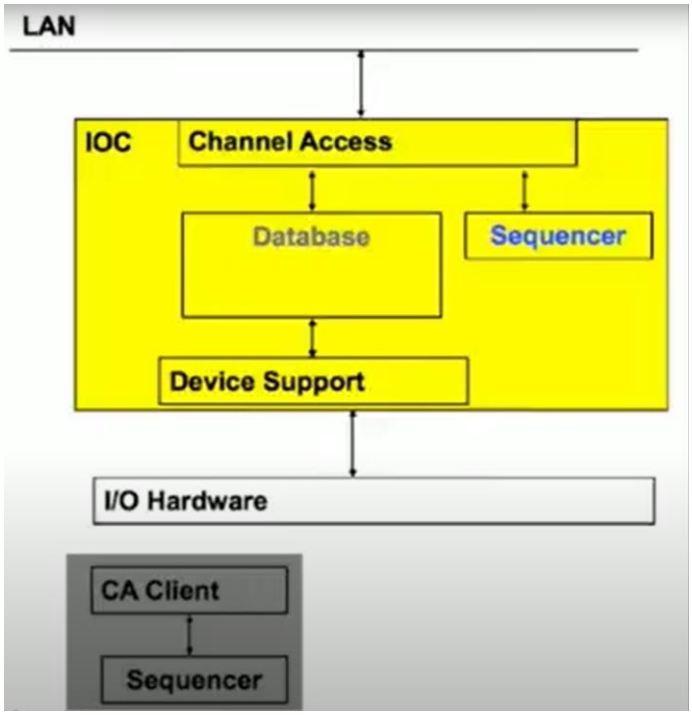

Figure 24 Sequencer Diagram [27]

A sequencer can exist either inside of the IOC or outside of the IOC, either way both use the same method to access the data and functions inside of the IOC. Channel Access is the protocol by which a client sends a UDP broadcast to all servers (usually IOC's) on the network asking for a specific process variable [29] [29]. If a server has the PV name in question it will send a UDP reply letting the client know which server contains the requested PV. The client will then send another request to the server with the requested PV to set up a TCP connection from which all future communication between client and server will occur [29]. This is in effect a virtual circuit. Should the client request

more than one PV from the server, it may send several UDP broadcasts however only one TCP connection will ever be established, possibly for multiple PV's. This allows the client to access multiple PVs contained inside of a server and is used not only for sequencers but also for any kind of external viewing or control of PV's inside of a server on the network.

A sequencer inside of the IOC generally uses a programming language called 'State Notation Language' or SNL and uses state machine logic for its operation. If the sequencer is inside of the IOC this is always called a virtual channel access client, which means the client accessing the PV's inside of the IOC is software running on the same machine as the IOC. External sequencers can use a variety of programming languages and can be virtual channel access clients or hardware channel access clients.

The basic idea of a sequencer is: the sequencer monitors a PV inside of an IOC for changes, when that PV changes it will use channel access to obtain the changed value of the PV, it will perform some logic and return some values to the IOC (not necessarily to the same PV it has been monitoring) via channel access.

10.2 PyEPICS

PyEPICS is a package for Python that gives Python an interface to access EPICS Channel Access protocol and EPICS process variables [30]. As stated, before a sequencer communicates via EPICS channel access, takes values from an EPICS server, performs some logic and returns something to the server. PyEPICS gives Python functions to access the data inside of an IOC and so the logic of the program is built in Python. The reason this approach was chosen is because Python is an easy-to-read language that is much faster and simpler to code and debug when compared to SNL which is notoriously difficult to debug. For the timespan of this project, it made sense to go with the simpler approach. However, one drawback is that this type of sequencer does not run from within the IOC itself and instead operates running as a Python script parallel to the IOC on the same hardware as a virtual channel access client or on separate hardware as a regular channel access client. Should one want to move the sequencer inside of the IOC for lower-level communication then SNL would be a better approach.

For installation on the Raspberry Pi 3 to run the sequencer as a virtual channel access client on the same hardware as the IOC, the operating system must be 32-bit because PyEPICS does not have libraries supporting 64-bit ARM. Should one wish to use a Raspberry Pi with a 64-bit operating system, SNL or another sequencer approach is required. For installation simply install as if installing a regular Python package by following this guide [31].

10.3 Final Program

```
import time
import epics
from epics import caget, caput, cainfo
from epics import PV
import binascii

first = ["3", "3", "0", "1", "2"]
second = ["3", "0", "0", "0", "0"]
third = ["5", "5", "1", "128", "0"]
fourth = ["33", "33", "0", "9", "25"]
fifth = ["32", "32", "10", "0", "28"]
#Byte values for the setup function. Read virtically

c = ['11', '0', '0', '0', '0'] #Byte values to set register 11 to 0
d = 0
e = 0 #Will be used as counters later
temp = 0
pwr = 2**32 #Quicker to declare this here


def I_O_UPpdate():
        caput('greenbt2:GPIO22:OUT', 'ON')
        time.sleep(0.0001)
        caput('greenbt2:GPIO22:OUT', 'OFF')
        #IO Update Pulse

def start_up():
        global first, second, third, fourth, fifth
        caput('greenbt2:GPIO17:OUT', 'ON')
        time.sleep(0.0001)
        caput('greenbt2:GPIO17:OUT', 'OFF')
        time.sleep(0.001)
        #Reset Pulse

        caput('greenbt2:GPIO27:OUT', 'ON')
        time.sleep(0.0001)
        caput('greenbt2:GPIO27:OUT', 'OFF')
        time.sleep(0.0001)
        #IO Sync Pulse

        i = 0

        for x in first:
                caput('TEST:BYTE1', first[i])
                caput('TEST:BYTE2', second[i])
                caput('TEST:BYTE3', third[i])
                caput('TEST:BYTE4', fourth[i]) #Sets four of the byte values in EPICS
                caput('greenbt2:GPIO18:OUT', 'ON')
                caput('greenbt2:GPIO23:OUT', 'ON')
                caput('greenbt2:GPIO24:OUT', 'ON')
                caput('greenbt2:GPIO25:OUT', 'ON') #Turns on all the CS lines for four AD9914 boards
                caput('TEST:BYTE5', fifth[i]) #After the fifth byte is set in EPICS, it sends all five bytes
                time.sleep(0.001) #Makes sure no timing errors occur
                caput('greenbt2:GPIO18:OUT', 'OFF')
                caput('greenbt2:GPIO23:OUT', 'OFF')
                caput('greenbt2:GPIO24:OUT', 'OFF')
                caput('greenbt2:GPIO25:OUT', 'OFF') #Turn off all CS lines
                I_O_UPpdate() #Moves the values from the serial buffer to the registers
                i = i+1
start_up()

def onChanges(pvname=None, value=None, char_value=None, **kw):
        print('PV Changed!', pvname, char_value, time.ctime())
        Value_PV = float(char_value) #When the value of the frequency PV changes
        print(Value_PV)
        FTW = int(round(2**32*(Value_PV/((2998.5*10**6))))) #Calculates the FTW to be sent via SPI
        print(FTW)
        FTW_bytes = FTW.to_bytes(4, 'big') #Converts to four bytes
        print(FTW_bytes)
        b= list(FTW_bytes)
        print(b)
        global c
        c[1] = str(b[0])
        c[2] = str(b[1])
        c[3] = str(b[2])
        c[4] = str(b[3]) #Moves the byte values to a global array
        global d
        global e
        d = e
        d = d+1 #Makes d != e to indicate the value has been changed
mypv = epics.PV('INPUT:FREQ') #Links the PV to the Input Frequency EPICS record
mypv.add_callback(onChanges) #Uses a callback to activate the onChanges function when the PV changes
```

```
def send():
        global d
        global e
        global temp
        if d == 100000:
                d = 1
                e = 0 #Stops the values increasing without bound
        if d != e:
                #When d != e, need to send the new value
                caput('TEST:BYTE1.VAL', c[0])
                caput('TEST:BYTE2.VAL', c[1])
                caput('TEST:BYTE3.VAL', c[2])
                caput('TEST:BYTE4.VAL', c[3])
                #Set the first four bytes
                if caget('CS1.VAL') == 1:
                        caput('greenbt2:GPIO18:OUT', 'ON')
                        #Puts CS line 1 on to write to DDS board 1
                if caget('CS2.VAL') == 1:
                        caput('greenbt2:GPIO23:OUT', 'ON')
                        #Puts CS line 2 on to write to DDS board 2
                if caget('CS3.VAL') == 1:
                        caput('greenbt2:GPIO24:OUT', 'ON')
                        #Puts CS line 3 on to write to DDS board 3
                if caget('CS4.VAL') == 1:
                        caput('greenbt2:GPIO25:OUT', 'ON')
                        #Puts CS line 4 on to write to DDS board 4
                caput('TEST:BYTE5.VAL', c[4])
                #Sets the fifth byte which sends the SPI signal
                caput('greenbt2:GPIO25:OUT', 'OFF')#25
                caput('greenbt2:GPIO24:OUT', 'OFF')#24
                caput('greenbt2:GPIO23:OUT', 'OFF')#23
                caput('greenbt2:GPIO18:OUT', 'OFF')#18
                #Must be turned off in this order due to timing
                I_O_UPpdate()

                #Read back procedure
                if caget('CS1.VAL') == 1: #18
                        caput('greenbt2:GPIO18:OUT', 'ON') #Puts CS line 1 on
                        temp = caget('READ:SWITCH.VAL') #Gets the current value of the read switch
                        temp = temp+1 #When temp changes, EPICS has a forward link to the record which activated the read function
                        time.sleep(0.001) #Timing here is critical, must give the CS line extra on time to account for timing variations in the read pulses
                        caput('READ:SWITCH.VAL', temp) #Changes the read switch value to activate the read function in EPICS
                        time.sleep(0.01) #Critical for timing variation
                        caput('greenbt2:GPIO18:OUT', 'OFF') #Turn of CS line 1

                if caget('CS2.VAL') == 1: #23
                        caput('greenbt2:GPIO23:OUT', 'ON')
                        temp = caget('READ:SWITCH2.VAL')
                        temp = temp+1
                        time.sleep(0.001)
                        caput('READ:SWITCH2.VAL', temp)
                        time.sleep(0.01)
                        caput('greenbt2:GPIO23:OUT', 'OFF')

                if caget('CS3.VAL') == 1:
                        caput('greenbt2:GPIO24:OUT', 'ON')
                        temp = caget('READ:SWITCH3.VAL')
                        temp = temp+1
                        time.sleep(0.001)
                        caput('READ:SWITCH3.VAL', temp)
                        time.sleep(0.01)
                        caput('greenbt2:GPIO24:OUT', 'OFF')

                if caget('CS4.VAL') == 1:
                        caput('greenbt2:GPIO25:OUT', 'ON')
                        temp = caget('READ:SWITCH4.VAL')
                        temp = temp+1
                        time.sleep(0.001)
                        caput('READ:SWITCH.VAL4', temp)
                        time.sleep(0.01)
                        caput('greenbt2:GPIO25:OUT', 'OFF')

                if temp > 50000:
                        caput('READ:SWITCH.VAL', 0)
                        caput('READ:SWITCH2.VAL', 0)
                        caput('READ:SWITCH3.VAL', 0)
                        caput('READ:SWITCH4.VAL', 0)
                        #Stops values increasing without bound
                e = e+1
                #Makes e = d, read/write is now finished
```

```
def main():
        while True:
                global pwr
                send()
                time.sleep(0.001)
                if caget('CS1.VAL') == 1:
                        #Performs the calculation to change the FTW into a readable frequency, must be performed in this function
                        caput('CHANNEL1:FREQ', (caget('READ:BACK.VAL')/(pwr))*(2998.5*10**6))
                        time.sleep(0.001)
                if caget('CS2.VAL') == 1:
                        caput('CHANNEL2:FREQ', (caget('READ:BACK2.VAL')/(2**32))*(2998.5*10**6))
                        time.sleep(0.001)
                if caget('CS3.VAL') == 1:
                        caput('CHANNEL3:FREQ', (caget('READ:BACK3.VAL')/(2**32))*(2998.5*10**6))
                        time.sleep(0.001)
                if caget('CS4.VAL') == 1:
                        caput('CHANNEL4:FREQ', (caget('READ:BACK4.VAL')/(2**32))*(2998.5*10**6))
                        time.sleep(0.001)
                send()



main()
```

Figure 25 Sequencer Code

```
Terminator   = "";
LockTimeout = 500;
ReplyTimeout = 100;
ReadTimeout  = 100;
WriteTimeout = 100;
MaxInput     = 4;
ExtraInput   = Error;

send {
       out "%(TEST:BYTE1.VAL)r", "%(TEST:BYTE2.VAL)r", "%(TEST:BYTE3.VAL)r"
       "%(TEST:BYTE4.VAL)r", "%(TEST:BYTE5.VAL)r";

}


read {
       out 0x8B;
       in "%04r";
}
```

Figure 26 Full Protocol File Code

```
record(mbboDirect, "TEST:SEND"){
        field(DTYP, "stream")
        field(OUT, "@test.proto send() SPI")
        field(SCAN, "Passive")
}

record(stringout, "TEST:BYTE1"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE2"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE3"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE4"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
}

record(stringout, "TEST:BYTE5"){
        field(DTYP, "Soft Channel" )
        field(VAL, "0")
        field(FLNK, "TEST:SEND" )
}

record(bo, "$(head):GPIO17:OUT") {
field(DTYP, "devgpio")
field(OUT, "@17 H") # or GPIO17, Active High
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bo, "$(head):GPIO27:OUT") {
field(DTYP, "devgpio")
field(OUT, "@27 H") # or GPIO27, Active High
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bo, "$(head):GPIO22:OUT") {
field(DTYP, "devgpio")
field(OUT, "@22 H") # or GPIO22, Active High
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(ai, "INPUT:FREQ"){
field(VAL, "0")
}

record( mbbiDirect, "READ:BACK"){
field(DTYP, "stream")
field(INP, "@test.proto read() SPI")
field(SCAN, "Passive")
}

record( mbbiDirect, "READ:BACK2"){
field(DTYP, "stream")
field(INP, "@test.proto read() SPI")
field(SCAN, "Passive")
}

record( mbbiDirect, "READ:BACK3"){
field(DTYP, "stream")
field(INP, "@test.proto read() SPI")
field(SCAN, "Passive")
}

record( mbbiDirect, "READ:BACK4"){
field(DTYP, "stream")
field(INP, "@test.proto read() SPI")
field(SCAN, "Passive")
}

record(bo, "$(head):GPIO18:OUT") {
field(DTYP, "devgpio")
field(OUT, "@18 L") # or GPIO18, Active Low
field(ZNAM, "OFF")
field(ONAM, "ON")
}
```

```
record(bo, "$(head):GPIO23:OUT") {
field(DTYP, "devgpio")
field(OUT, "@23 L") # or GPIO23, Active Low
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bo, "$(head):GPIO24:OUT") {
field(DTYP, "devgpio")
field(OUT, "@24 L") # or GPIO24, Active Low
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bo, "$(head):GPIO25:OUT") {
field(DTYP, "devgpio")
field(OUT, "@25 L") # or GPIO25, Active Low
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bi, "CS1") {
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bi, "CS2") {
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bi, "CS3") {
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(bi, "CS4") {
field(ZNAM, "OFF")
field(ONAM, "ON")
}

record(longin, "READ:SWITCH") {
field(VAL, "0")
field(FLNK, "READ:BACK")
}

record(longin, "READ:SWITCH2") {
field(VAL, "0")
field(FLNK, "READ:BACK2")
}

record(longin, "READ:SWITCH3") {
field(VAL, "0")
field(FLNK, "READ:BACK3")
}

record(longin, "READ:SWITCH4") {
field(VAL, "0")
field(FLNK, "READ:BACK4")
}

record(ai, "CHANNEL1:FREQ") {
field(VAL, "0")
}

record(ai, "CHANNEL2:FREQ") {
field(VAL, "0")
}
record(ai, "CHANNEL3:FREQ") {
field(VAL, "0")
}
record(ai, "CHANNEL4:FREQ") {
field(VAL, "0")
}
```

Figure 27 Full Database File

The EPICS solution has three sections the sequencer, database and the protocol. The circuit connection between the RPi and the four AD9914 boards is, each signal pin is connected to each AD9914 in a parallel configuration except for the chip select line for which the RPi has four individual CS pins connecting to its own AD9914. As stated back in section seven the AD9914 requires the control register to be set in a certain way with certain values for the board to operate in single tone mode. The byte records hold decimal numbers stored as strings and so that is what will be passed to and from the record. Looking at the beginning of figure 10 a new version of the start-up function can be seen. The functions 'caput' and 'caget' inside of the sequencer program use channel access to obtain and replace the information of a record inside of the database.

The start-up function inside of the sequencer turns the reset line and then the IO sync line on for all AD9914 boards by using channel access to turn the relevant GPIO pin on and off with the devGPIO device support. At the start of the sequencer program there are five arrays with the data required for the control registers. There is a 'for' loop that uses channel access to put this data into the respective byte records inside the database. Looking at figure 12 it can be seen that the record holding the data for byte five has a forward link to a record called send which has the link to the send function inside of the protocol file. Looking back to the for function in figure 10, the first four byte records are updated with the values from the start up arrays. Then all four chip select lines are set low. This is because all four AD9914 boards require the same information on start-up and so the information can be written to all four boards simultaneously. After the fifth byte record is written to the forward link to the send record allows that record to process. This then calls the send function inside of the protocol file, figure 11, which sends the raw binary data, converting the stored decimal to binary, from the individual byte records over SPI bus 0. The chip select lines are all set high to signify end of SPI communication, an I/O update pulse is sent to all four boards to move the information from the buffer to the register and then the cycle begins again until the setup is complete.

With all four boards now set up all that needs to be changed to change the frequency is the frequency tuning word inside of register eleven. Inside of the database there is a record called input frequency which is one of the few records one would actually input data to. The idea is, a GUI containing input fields for: input frequency, chip select one, chip select two, chip select three and chip select four would allow a user to change the frequency of any of the four boards, individually or together and then there would be four outputs corresponding to the output records: channel 1 frequency, channel 2 frequency etc. When the record input frequency changes, that signifies that the user wants to change the frequency of at least one of the boards.

The function 'on changes' inside of the sequencer monitors this PV's value for changes. Since the record input frequency has been declared as a PV in the sequencer, it does not require the use of 'caput' 'caget' to obtain the value of the record. When the PV changes the sequencer calculates the new FTW, converts the value to four bytes in the form of decimal numbers of type string and sets two counter variables as unequal to each other to signify to another function that the PV has changed. The call back function calls the 'on changes' function when the PV changes.

The send function activates when the counter variables are unequal, signifying a need to send new data to the board. Inside of the function, the first four byte values are updated via channel access commands, same as in the start-up functions. Then four 'if' statements occur which is if the chip select value for board one is on then turn chip select line 1 on etc. For example, if one only wishes to write to board one then put the value of record CS1 to 'ON' then when the value changes CS1 will go low before the SPI information is sent, allowing board one to 'receive' the SPI data. For the other three boards, because the respective 'CS' record is set to 'OFF' the chip select line will stay high throughout the SPI send cycle meaning that respective board will not 'receive' or stay deaf to the SPI

data stream. One can turn the CS lines on or off in any order, together or singular and the respective boards will update accordingly. The fifth byte is then updated, and the SPI data is sent. All CS lines are now set high to indicate end of communication and the I/O update line moves any information in the serial buffers to move to the registers.

Whilst all four boards can be written to simultaneously, they can't be read from simultaneously and instead must read back in order. This time the if statement is: if CS is high then put the CS line for that board low, caget the value of the read switch record, add one to the read switch record, wait, send the new value of the read switch to trigger the read function, wait, turn the CS line off, is the next CS line on, repeat. The read switch record is merely a counter record that has the job of forward linking to the read back record when it changes so the read back record processes and calls the read function inside of the protocol file. The read function sends and receives the SPI data and places the FTW it received from whichever board it was reading from inside of the respective read back record as a decimal of type integer. If the CS line for that particular board is set to 'OFF' the code will not read from it. At the end of the send function, it makes the temporary counters equal to each other again, so the function only runs once.

Finally, the main function runs the send function as well as four caget commands. They pull the new FTW from whatever board has changed frequency and converts it into a readable frequency. This is what will be viewable as the current frequency of each board on the GUI.

11 Future Work

11.1 Input Frequency Issue and Four Input Frequency PVs Solution

One problem currently facing the system is that, since the system only uses one PV for input frequency if one wants to change the frequency of only board one to for example 100MHz then the only CS with the value on would be CS1. If now one wants to set board two to the same frequency the only CS to be set to on is CS2 but since the value of the input frequency is the same as last time the 'On Changes' function does not trigger thus not updating board two to the new frequency. Since all four channels do not need to be changed simultaneously one can circumvent this issue by changing the code to work with four input frequency PVs.

To begin, one would first need to duplicate the 'Input Frequency' record inside of the database three more times and give the records unique names. Also, the removal of the 'CS' records from the database and any mention of them in the sequencer will be required. The code should add the three new records as PVs all with call-back to the 'on changes' function. Next, the 'On Changes' function logic should be changed slightly with a chain of 'if' statements at the end. Each PV should have two unique counter variables, for example, input frequency one has the counter names 'a' and 'b' input frequency two has counter names 'c' and 'd' etc. At the end of the code a set of 'if' statements should implement the logic: if PV_name = input frequency 1 then set counter a!=b, if PV_name = input frequency 2 then set counter c!=d etc. Finally, having one extra counter pair for example, I and J, should be set unequal to each other to tell the send function that an input PV in general has changed with the other logic telling the function which PV in in particular has changed. The send function will change the most firstly, the large if d!=e statement should be changed to if I!=J, telling the function that one of the four PVs has changed. Next, simply replacing the 'if caget('CS1.VAL) ==1' logic to ' if a!=b' for each instance of this logic, being sure to change each instance to the correct counter names for each respective PV. The only thing to change is at the end of the send function, the counters I and J should be set equal, indicating that the frequency of whatever PV changed has now been updated, but the other counters should not be set equal in the send function. Because the specific counter pair being unequal tells the code which PV has changed, this can't be set equal until all functions requiring this logic have been executed. The 'if' statements inside of the 'main' function require this logic to convert the FTW into a readable frequency. The respective identifier counter

pairs should be set equal at the end of their respective ‘if’ statements inside of this function. This is the easiest way to implement four PVs with the current sequencer logic.

11.2 Ability to Reset Each Board Individually

Currently, there is no ability for the system to reset each AD9914 board individually without restarting the IOC and sequencer. This poses a problem in that, should any errors occur with the boards individually, there would be no simple way just to reset one board. Currently, all four boards are connected to the same reset line in parallel which means when one board receives a reset pulse, all boards receive a reset pulse. This will need to change so that each board is connected to its own reset pin on the raspberry pi. The I/O update and I/O sync can all be left in parallel. This means adding three new GPIO records inside of the database for three new reset lines. Also, having four new binary input records called for example Reset 1 should be added. Firstly, the ‘start up’ function will need to change to include a reset pulse for each of the boards. The rest of the start-up function can remain the same. This sets up all four boards on start-up of the IOC and sequencer. Next, one will need to create four new functions to reset each of the boards individually. This will be very similar to the start-up function but will only send a pulse to one of the AD9914 boards and will only write the control registers of one AD9914. Next, add the four new Reset records as PVs with a call-back to one of the new start-up functions. The idea of the logic is as follows: set reset 1 record to ‘ON’, PV reset 1 has changed call-back to the reset 1 function, perform the reset logic ONLY for AD9914 board 1, set the input frequency record to zero for that specific board, set the reset record to ‘OFF’ for that specific board. This imitates a ‘reset button’ approach for each individual board.

11.3 Move Sequencer Inside the IOC

As stated in previous sections it may be beneficial to move the sequencer inside of the IOC verses having it run parallel to it, as is the case with the current system. Aside from the database additions mentioned in the earlier future work sections, the IOC itself will not need to change. The main advantage of using a sequencer-based approach with channel access is that the sequencer can be removed and rewritten in any language, so long as it has access to channel access, without changing the IOC. To move the sequencer inside the IOC the program must be written in SNL or state notation language and the sequencer must be pointed to in the start-up script of the IOC. A guide to SNL sequencers can be found here [32]. For this to work the same as the current system, the exact logic and timing should be ported from the Python sequencer, and additions mentioned in future work, to a new SNL sequencer.

11.4 PCB Connection

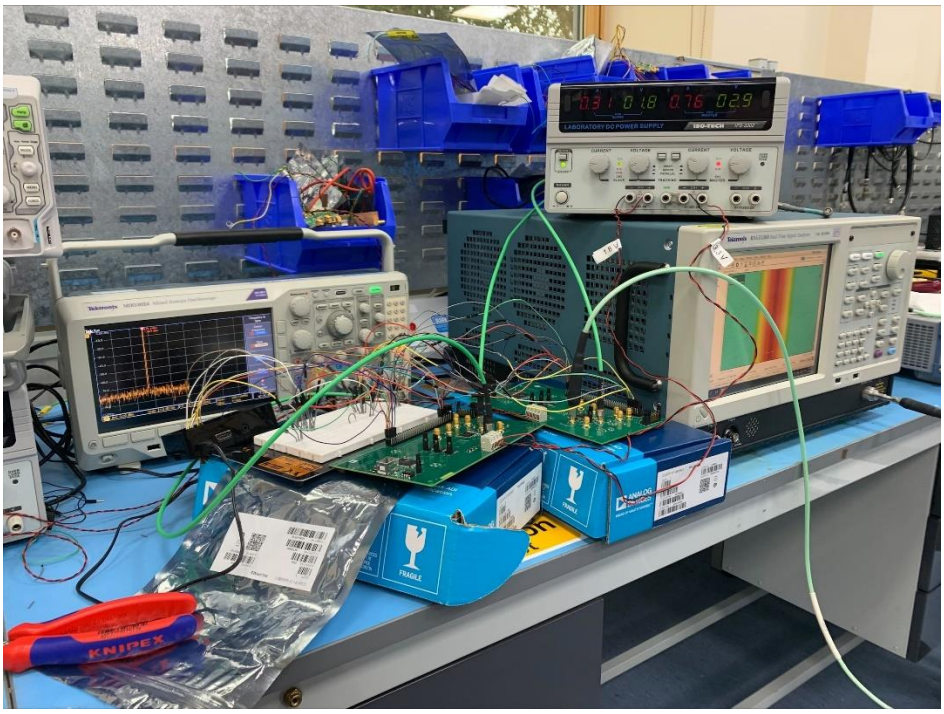

Figure 28 Multi-Board Test Rig

Figure 13 shows the current test setup to implement the circuit from the Raspberry Pi to two AD9914 boards using a breadboard as a connection interface. This is by no means a tidy solution and it is difficult to trace circuit problems because of that. Attempting to use this approach with four boards would compound the problem and so a different solution is required for the final system.

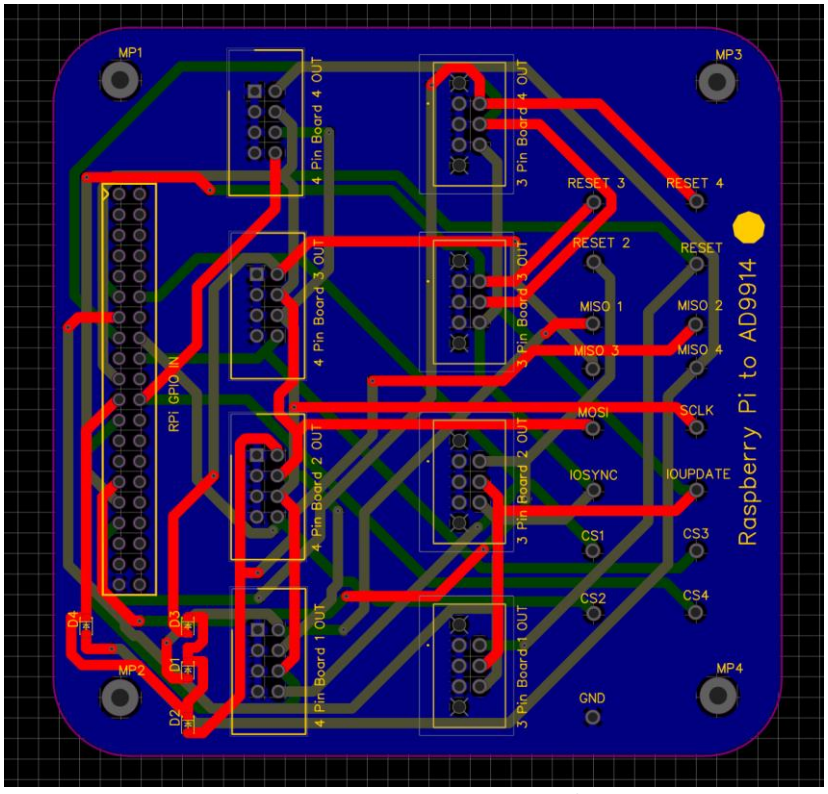

Figure 29 Connection PCB Trace Outline

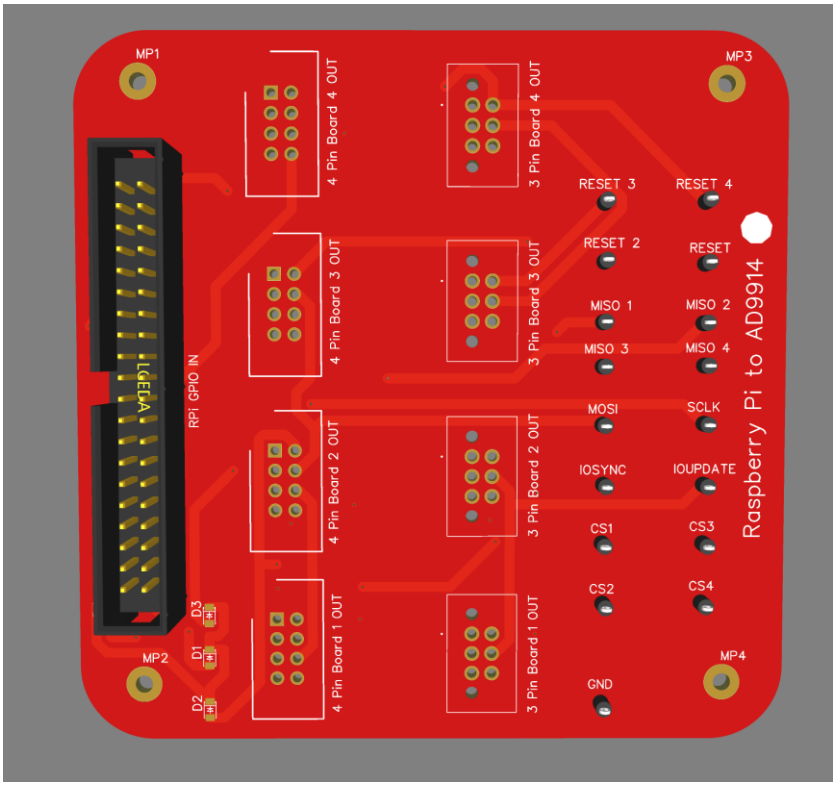

Figure 30 Connection PCB 3D Model

To circumvent the untidy breadboard approach, as seen in the test rig, a PCB has been developed to connect the Raspberry Pi to the four AD9914 boards. The board is designed to take a ribbon cable input from the Raspberry Pi's 40 pin GPIO. Each AD9914 board then has its own four pin connection and three pin connection. The four-pin connection is designed to connect via a ribbon cable to the SPI pins on the AD9914s parallel port and the three-pin connection is designed to connect via a ribbon cable to the reset, I/O sync and update pins on the opposite end of the parallel port. This PCB has been designed with the individual reset functionality in mind and so, to use this PCB design when adding the new reset pin records, they must be set as follows: Reset pin 1 = GPIO17, Reset pin 2 = GPIO 5, Reset pin 3 = GPIO6 and Reset pin 4 = GPIO 13. The connection board also contains testing posts for each of the communication lines for debugging purposes. One final thing to mention is the diodes for the MISO line. During testing it was discovered that connecting the MISO lines in parallel caused degradation of the return signal, causing the RPi to miss-read the signal. However, when removing the parallel connections so only one MISO line was connected, the signal was no longer degraded and the RPi read the correct information every time. To recreate this effect, four diodes are connected to each of the MISO lines and then in parallel with each other on the output side of the diode. This ensures the signal only travels from the MISO output of the AD9914 to the Raspberry Pi without any interference.

12 Conclusion

Although the project still has much work to be done in terms of the future work section, building and assembling the hardware and real-world testing with the IQ path and EPICS network the majority of the work required for this system is present here. If the work has not been completed a reasonable method has been suggested in the future work section.

The system was fully tested with two AD9914 boards using the EPICS-SPI communication method and was able to fully control the output frequency of both boards. There is no reason to think that the current system would not work with four boards however, it remains another point that hasn't been tested.

Overall, though given the short time period of this project the initial goals have been far exceeded and a viable solution has been presented to act as an EPICS controlled four channel LO input for the IQ demodulator on the upcoming Wakefield monitor for CLARA phase 3.

13 Bibliography

[1] STFC, “CLARA,” [Online]. Available: https://www.astec.stfc.ac.uk/Pages/CLARA.aspx. [Accessed 21 09 2022].

[2] P. E. C. P. S. Reiche1, “PULSE LENGTH CONTROL IN AN X-RAY FEL BY USING WAKEFIELDS,” SLAC-PUB-13180.

[3] M. L. S. H. M. Dehler, “FRONT END CONCEPT FOR A WAKE FIELD MONITOR,” in *IBIC2014,*, Monterey, 2014.

[4] Analog Devices, “AD9914,” 2022. [Online]. Available: https://www.analog.com/en/products/ad9914.html#product-overview. [Accessed 21 09 2022].

[5] H. N. Masako, “Construction of measuring instrument data collection system using Raspberry Pi. Easy and inexpensive DAQ construction,” Osaka, 2018.

[6] Y. E. S. U. Shiro Kusano, “DEVELOPMENT OF CONTROL SOFTWARE FOR EPICS COMPATIBLE DC MAGNET,” in *Proceedings of the 18th Annual Meeting of Particle Accelerator Society of Japan*, Takasaki, 2021.

[7] Wikipedia, “Single master to single slave: basic SPI bus example,” [Online]. Available: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface#/media/File:SPI_single_slave.svg. [Accessed 21 09 2022].

[8] P. Dhaker, “Introduction to SPI Interface,” Analog Dialogue, [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html. [Accessed 21 09 2022].

[9] Analog Devices, “AD9914,” 14 09 2022. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD9914.pdf.

[10] EPICS, “What is EPICS?,” 2022. [Online]. Available: https://epics-controls.org/. [Accessed 22 09 2022].

[11] EPICS, “EPICS Overview,” 2019. [Online]. Available: https://docs.epics-controls.org/en/latest/guides/EPICS_Intro.html. [Accessed 22 09 2022].

[12] F. Feldbauer, “epics-devgpio,” 18 08 2022. [Online]. Available: https://github.com/ffeldbauer/epics-devgpio. [Accessed 15 09 2022].

[13] KEK, “drvAsynSPI,” 25 09 2018. [Online]. Available: https://github.com/kek-acc/drvAsynSPI. [Accessed 15 09 2022].

[14] Sidekick EPICS Documentation, “Install EPICS v7 on Raspberry Pi (40 minutes),” [Online]. Available: https://sfeister.github.io/sidekick-epics-docs/pi-epics-install/. [Accessed 15 09 2022].

[15] EPICs, “Installation on Linux/UNIX/DARWIN (Mac),” [Online]. Available: https://docs.epics-controls.org/projects/how-tos/en/latest/getting-started/installation.html#install-epics. [Accessed 15 09 2022].

[16] Emmet, “Increasing Swap on a Raspberry Pi,” 30 01 2022. [Online]. Available: https://pimylifeup.com/raspberry-pi-swap-file/. [Accessed 15 09 2022].

[17] Paul Scherrer Institut, “EPICS Stream Device,” 2018. [Online]. Available: https://paulscherrerinstitute.github.io/StreamDevice/. [Accessed 15 09 2022].

[18] Sidekick EPICS Documentation, “Build and Install StreamDevice on Raspberry Pi (20 minutes),” 2022. [Online]. Available: https://sfeister.github.io/sidekick-epics-docs/pi-stream-device-install/. [Accessed 15 09 2022].

[19] F. Feldbauer, “R1-0-6,” 11 06 2019. [Online]. Available: https://github.com/ffeldbauer/epics-devgpio/releases/tag/R1-0-6. [Accessed 16 09 2022].

[20] KEK, “Using StreamDevice with drvAsynSPI,” 04 01 2018. [Online]. Available: https://github.com/kek-acc/drvAsynSPI/tree/master/streamDevice. [Accessed 16 09 2022].

[21] A. Maqbool, “How to Run “patch” Command in Linux?,” 2021. [Online]. Available: https://linuxhint.com/run-patch-command-in-

linux/#:~:text=In%20Linux%20operating%20system%2C%20%E2%80%9Cpatch,get%20the%20difference%20or%20patch.. [Accessed 16 09 2022].

[22] F. Feldbauer, “devgpio.pdf,” 02 03 2016. [Online]. Available: https://github.com/brunoluvizotto/epics-devgpio/blob/master/documentation/devgpio.pdf. [Accessed 16 09 2022].

[23] T. Obina, “hādowea e no akusesu,” KEK, 01 11 2018. [Online]. Available: https://cerldev.kek.jp/trac/EpicsUsersJP/raw-attachment/wiki/intro/20181101_KEK/day1_gpio.pdf. [Accessed 16 09 2022].

[24] Paul Scherrer Institut , “Protocol Files,” 2018. [Online]. Available: https://paulscherrerinstitute.github.io/StreamDevice/protocol.html. [Accessed 20 09 2022].

[25] F. Feldbauer, “ad7998.proto,” 27 03 2017. [Online]. Available: https://github.com/ffeldbauer/drvAsynI2C/blob/master/example/protocol/ad7998.proto. [Accessed 2022 09 2022].

[26] EPICS Users JP, “I2C setsuzoku no sensā o seigyo,” 15 06 2017. [Online]. Available: https://cerldev.kek.jp/trac/EpicsUsersJP/wiki/epics/raspberrypi/setup_epics_i2c. [Accessed 15 09 2022].

[27] A. Johnson, “SNL Programming,” 01 12 2014. [Online]. Available: https://www.youtube.com/watch?v=oMe7p__zLyQ. [Accessed 20 09 2022].

[28] EPICS Sequencer, “Introduction,” [Online]. Available: https://www-csr.bessy.de/control/SoftDist/sequencer/Introduction.html#overview. [Accessed 20 09 2022].

[29] A. Johnson, “CA Concepts,” 20 10 2014. [Online]. Available: https://www.youtube.com/watch?v=ZX5Nbm4tYdA. [Accessed 20 09 2022].

[30] PyEPICS, “EPICS Channel Access for Python,” 2021. [Online]. Available: https://cars9.uchicago.edu/software/python/pyepics3/. [Accessed 20 09 2022].

[31] PyEPICS, “Downloading and Installation,” 2021. [Online]. Available: https://cars9.uchicago.edu/software/python/pyepics3/installation.html#downloads-and-installation. [Accessed 20 09 2022].

[32] EPICS Sequencer, “Tutorial,” [Online]. Available: https://www-csr.bessy.de/control/SoftDist/sequencer/Tutorial.html. [Accessed 21 09 2022].

[33] F. Feldbauer, “drvAsynI2C,” 31 05 2022. [Online]. Available: https://github.com/ffeldbauer/drvAsynI2C. [Accessed 20 09 2022].

[34] M. Elfleet, “Science and Technology Facilities Council Summer Placement Project Report,” Unpublished, 2022.

[35] EPICS Users JP, “RaspberryPi ni EPICS o insutōru,” 15 06 2017. [Online]. Available: https://cerldev.kek.jp/trac/EpicsUsersJP/wiki/epics/raspberrypi. [Accessed 15 09 2022].