Upgrade of the X-Band receiver control system.

A. Martínez-Parra.
P. García, F. Beltrán, M. Patino, J.A López-Pérez, F. Valle, A. Alonso, C. Almendros, S. Henche.

IT-CDT 2020-20
09/09/2020

INDEX:

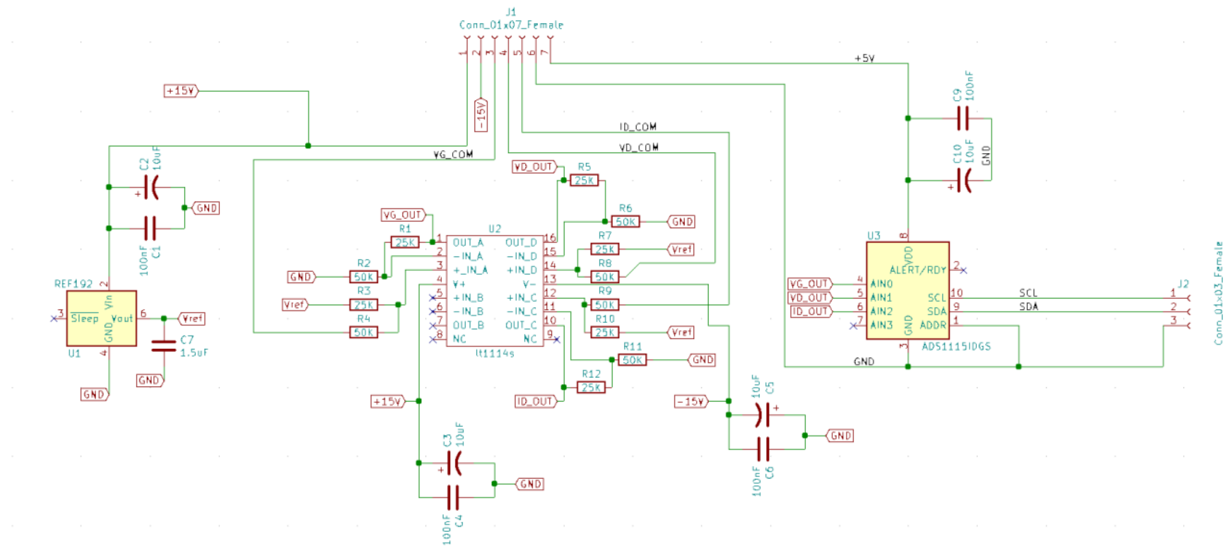
1. INTRODUCTION.

MicroIOC is the system used to control and monitor many of the receivers used in Yebes 40 meters radio-telescope. The aim of changing the microIOC for a Raspberry Pi 3, is to improve and correct errors of the previous control. This report indicates the changes made to carry out this replacement, the improvements that were made to achieve a better response and the changes made on some of the modules that compose the FI module of X-Band receiver.

2. DESIGN OF A BIAS MONITORING AND CONTROL PCB.

By reviewing the technical report IT-OAN 2007-22, it was concluded that, for Raspberry to control the Hemt monitoring cards (Bias monitoring), an auxiliary PCB card had to be made to change the voltage range from [-5V, 5V] to [0V, 5V]. This change is due to the fact the ADC (ADS1115) has an operating range of 0V to 5V.

The auxiliary card was made with KiCad program, whose schematics are as follows:



This PCB card consists of a voltage reference of 2.5V (REF192), four operational amplifiers (LT1114S) and a 16-bit ADC configured in single ended mode (ADS1115). With these three elements and the configuration of resistors, it is possible to make the change of the voltage range that was mentioned above.

After the creation of the schematic, the board was routed and subsequently created with the LPKF. *(Photo 1)* This PCB communicates with the Raspberry from the I2C port and is capable of controlling all voltages and currents of Bias polarizations.

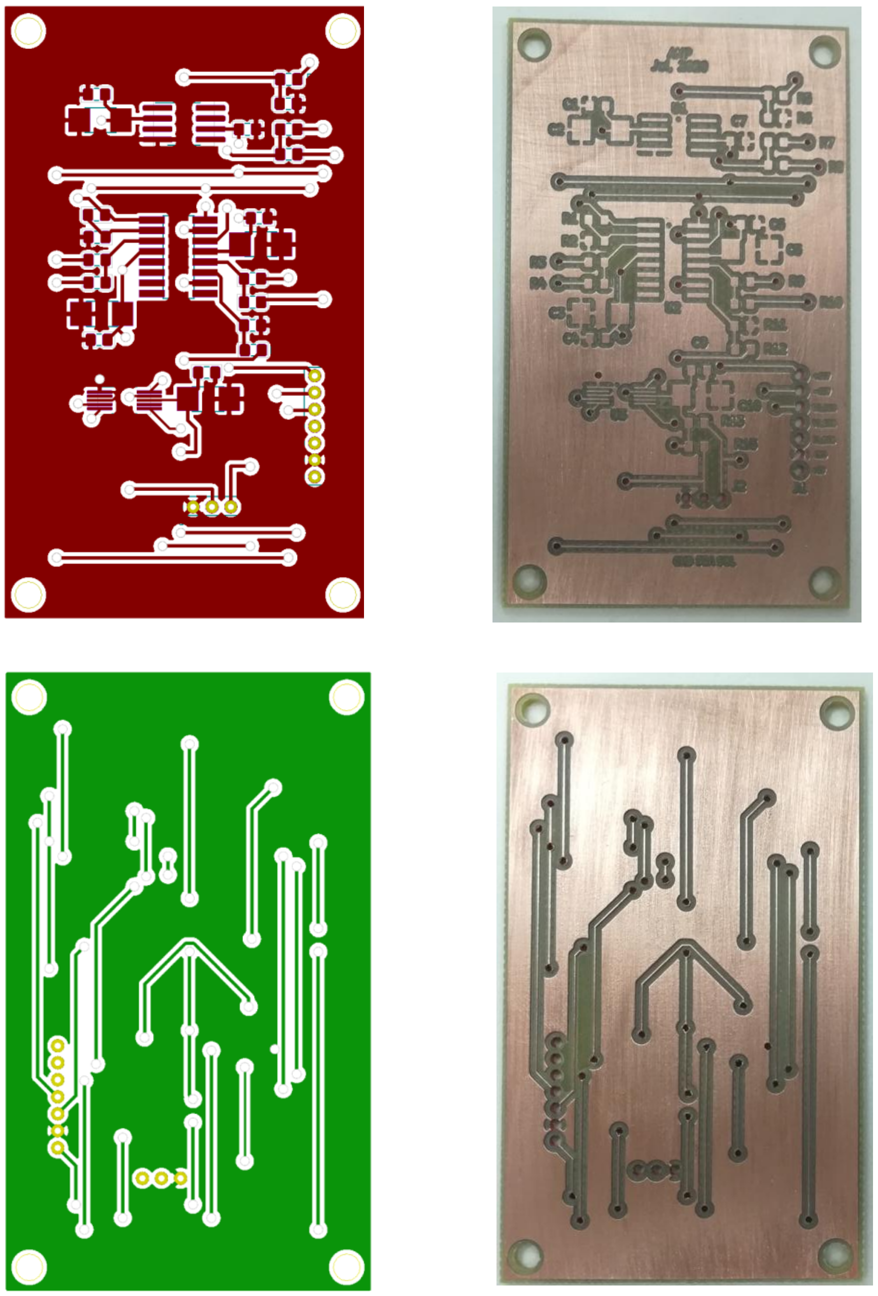


Photo 1

3. COMMANDS USED TO CONTROL AND MONITORING.

The program was created with the commands that would make the control system work correctly.
For this, the program that was already created in C++ for the microIOC was used, modified and changed to Python so that the Raspberry could perform and control the system as necessary.

The commands used are as follows:

- get_xBand7650Locked.
- get_xBand450Locked.
- cleanupExtRef.
- cleanupLocked.
- setAtt_R.
- getAtt_R.
- setAtt_L.
- getAtt_L.
- setAmplifierStage.
- getAmplifierStage.
- setAmplifierChannel.
- getAmplifierChannel.
- polarizationGateVoltage.
- polarizationDrainVoltage.
- polarizationDrainCurrent.
- setNoiseDiodeMode.
- getNoiseDiodeMode.
- setPhasecalMode.
- getPhasecalMode.

get_xBand7650Locked and get_xBand450Locked:

These functions are used to know if the OL module is working properly. The code used is as follows:

```
if str_rec == 'get_xBand7650Locked\r\n':

if GPIO.input(22):

sc.send(str.encode('1. OK\r\n'))

else:

sc.send(str.encode('0. OK\r\n'))
```

```
elifstr_rec == 'get_xBand450Locked\r\n':

if GPIO.input(27):

sc.send(str.encode('1. OK\r\n'))

else:

sc.send(str.encode('0. OK\r\n'))
```

If the result is 1, it means it is working and if the result is 0 it fails.

setAtt_R, getAtt_R, setAtt_L and getAtt_L:

These four functions are used to control attenuation, from programmable attenuators, that can be set in the IF module.

```
elif str_rec[0:8] == 'setAtt_R':

at=str_rec.split()

aux=bin(int(at[1]))

a1=len(aux)

x1=7-a1

var=[0,0,0,0,0]

for i1 in range((a1-1), 1, -1):

        if a1 == 7:

                var[i1-2]=int(aux[i1])

       elif a1 == 6:

           var[i1-1]=int(aux[i1])

       elif a1 == 5:

           var[i1]=int(aux[i1])

       elif a1 == 4:

           var[i1+1]=int(aux[i1])

       elif a1 == 3:

           var[i1+2]=int(aux[i1])


for j1 in range (a1,x1):

        var[j1]=0

if var[4]:

     GPIO.output(5, 1)

else:

     GPIO.output(5, 0)

if var[3]:

     GPIO.output(6, 1)

else:

     GPIO.output(6, 0)

if var[2]:

     GPIO.output(13, 1)

       else:

     GPIO.output(13, 0)

if var[1]:

     GPIO.output(19, 1)

else:

     GPIO.output(19, 0)

if var[0]:

     GPIO.output(26, 1)

else:

     GPIO.output(26, 0)

sc.send(str.encode('OK\r\n'))



elif str_rec == 'getAtt_R\r\n':

att=0

if GPIO.input(26):

      att=att+16

if GPIO.input(19):

      att=att+8

if GPIO.input(13):

      att=att+4

if GPIO.input(6):

      att=att+2

if GPIO.input(5):

      att=att+1

print (att)

sc.send(str.encode('%d\r\n' % att))
```

```
elif str_rec[0:8] == 'setAtt_L':
atL=str_rec.split()
auxL=bin(int(atL[1]))
a=len(auxL)
x=7-a
varL=[0,0,0,0,0]

for i in range((a-1), 1, -1):
     if a == 7:
        varL[i-2]=int(auxL[i])
     elif a == 6:
        varL[i-1]=int(auxL[i])
      elif a == 5:
         varL[i]=int(auxL[i])
      elif a == 4:
         varL[i+1]=int(auxL[i])
      elif a == 3:
         varL[i+2]=int(auxL[i])


for j in range (a,x):
varL[j]=0
   if varL[4]:
       GPIO.output(18,1)
   else:
       GPIO.output(18,0)
   if varL[3]:
       GPIO.output(23,1)
   else:
       GPIO.output(23,0)
   if varL[2]:
       GPIO.output(24,1)
   else:
       GPIO.output(24,0)
   if varL[1]:
       GPIO.output(25,1)
   else:
       GPIO.output(25,0)
   if varL[0]:
       GPIO.output(12,1)
   else:
       GPIO.output(12,0)
   sc.send(str.encode('OK\r\n'))

elif str_rec == 'getAtt_L\r\n':
attL=0
if GPIO.input(12):
       attL=attL+16
if GPIO.input(25):
       attL=attL+8
if GPIO.input(24):
       attL=attL+4
if GPIO.input(23):
        attL=attL+2
if GPIO.input(18):
        attL=attL+1
print (attL)
sc.send(str.encode('%d\r\n'     %     attL))
```

From the 5 bits set on the Raspberry for each polarization (RCP or LCP) you can set attenuation and read the attenuation that is programmed on those pins.

cleanupExtRef:

This function is used to determine whether the 5MHz reference is on or not. Its code is as follows:

```
elifstr_rec == 'cleanupExtRef\r\n':
    if GPIO.input(17):
sc.send(str.encode('1. No loss of Reference.\r\n'))
    else:
sc.send(str.encode('0. Loss of Reference.\r\n'))
```

If the answer is 0, the reference is not on and your information does not reach the module. However, if the answer is 1, means that the 5MHz reference is properly powered on.

cleanupLocked:

This function is used to determine whether the 5Mhz reference is connected to the module or not.

```
elifstr_rec == 'cleanupLocked\r\n':
   if GPIO.input(4):
sc.send(str.encode('1. Unlocked\r\n'))
   else:
sc.send(str.encode('0. Locked\r\n'))
```

If the answer is 1 it means that the reference signal is not connected to the module. But if the answer is 0, it is that the reference signal is connected.

setAmplifierChannel and getAmplifierChannel:

These functions are used to set and read which polarization is set out of the two possible (LCP and RCP).

```
elifstr_rec[0:19] == 'setAmplifierChannel':
   a6=str_rec.split()
   aux6=a6[1]
   if aux6 < '0' or aux6 > '1':
print("Introducir canal 0 o canal 1.")
else:
      if aux6 == '0':
GPIO.output(16, 0)
sc.send(str.encode('OK\r\n'))
     if aux6 == '1':
GPIO.output(16, 1)
sc.send(str.encode('OK\r\n'))

elifstr_rec == 'getAmplifierChannel\r\n':
```

```
if GPIO.input(16):
sc.send(str.encode('1. OK\r\n'))
```

```
     else:
sc.send(str.encode('0. OK\r\n'))
```

LCP polarization is set to 0, and RCP polarization is set to 1.So, if the LCP polarization is wanted, it is needed to set (or read) a 0 and if the RCP polarization is wanted, it is needed to set (or read) a 1.

setAmplifierStage and getAmplifierStage:

These functions serve to be able to set and read at what stage, of the three possible, each polarization is configured.

```
elifstr_rec[0:17] == 'setAmplifierStage':
  a5=str_rec.split()
  aux5=a5[1]
  if aux5 < '0' or aux5 > '4':
print("Introducir stage 1, 2, 3 o 4.")
  else:
     if aux5 == '1':
GPIO.output(20, 0)
GPIO.output(21, 0)
sc.send(str.encode('OK\r\n'))
     if aux5 == '2':
GPIO.output(20, 0)
GPIO.output(21, 1)
sc.send(str.encode('OK\r\n'))
    if aux5 == '3':
GPIO.output(20, 1)
GPIO.output(21, 0)
sc.send(str.encode('OK\r\n'))
    if aux5 == '4':
GPIO.output(20, 1)
GPIO.output(21, 1)
sc.send(str.encode('OK\r\n'))

elifstr_rec == 'getAmplifierStage\r\n':
if GPIO.input(21):
if GPIO.input(20):
sc.send(str.encode('4. OK\r\n'))
       else:
sc.send(str.encode('2. OK\r\n'))
  else:
       if GPIO.input(20):
        sc.send(str.encode('3.OK\r\n'))
       else:
        sc.send(str.encode('1.OK\r\n'))
```

Stage 1 is set to 00, stage 2 as 01, and stage 3 is set to 10. Although stage 4 is also referenced, it is not used in this module.

polarizationGateVoltage, polarizationDrainVoltage and polarizationDrainCurrent: These three functions are used to read the gate voltage, drain voltage, and drain current of amplifiers. These values are obtained from the PCB created (explained in the previous section) and from the I2C port.

```
elifstr_rec == 'polarizationGateVoltage\r\n':

   RATE = 860

   SAMPLES = 1000

   i2c   =   busio.I2C(board.SCL,   board.SDA,
frequency=1000000)

   ads = ADS.ADS1115(i2c)

ads.mode = Mode.CONTINUOUS

ads.data_rate =RATE

ads.gain=2/3

   chan0 = AnalogIn(ads, ADS.P0)

   vout0=float(chan0.voltage)

   vin0=(vout0-2.5)*1.96

sc.send(str.encode('%.5f,%.5f\r\n'           %
(vin0,vout0)))



elifstr_rec== 'polarizationDrainVoltage\r\n':

    RATE = 860

    SAMPLES = 1000

    i2c   =   busio.I2C(board.SCL,   board.SDA,
frequency=1000000)

    ads = ADS.ADS1115(i2c)

ads.mode = Mode.CONTINUOUS

ads.data_rate =RATE

ads.gain=2/3

chan1 = AnalogIn(ads, ADS.P1)

    vout1=float(chan1.voltage)

    vin1=((vout1-2.5)*1.96)+0.02

sc.send(str.encode('%.5f,%.5f\r\n'           %
(vin1,vout1)))



elif str_rec== 'polarizationDrainCurrent\r\n':

    RATE = 860

    SAMPLES = 1000

    i2c   =   busio.I2C(board.SCL,   board.SDA,
frequency=1000000)

    ads = ADS.ADS1115(i2c)

ads.mode = Mode.CONTINUOUS

ads.data_rate =RATE

ads.gain=2/3

     chan2 = AnalogIn(ads, ADS.P2)

     vout2=float(chan2.voltage)

     vin2=((vout2-2.5)*1.96)+0.04

sc.send(str.encode('%.5f,%.5f\r\n'           %
(vin2,vout2)))
```

setNoiseDiodeMode and getNoiseDiodeMode:

These functions are used to set and read in which state the Noise Diode is.

```
elifstr_rec [0:17] == 'setNoiseDiodeMode':

  a3=str_rec.split()

  aux3=a3[1]

  if aux3 < '0' or aux3 > '2':

print("Introducir Modo 0, Modo 1 o Modo 2.")

else:

     if aux3 == '0':

         cad="DOF\r\n"

elif aux3 == '1':

           cad="DON\r\n"

  elif aux3 == '2':

           cad="D80\r\n"

  s.write(str.encode(cad))

  s.read(5)

  sc.send(str.encode('OK\r\n'))

  elifstr_rec == 'getNoiseDiodeMode\r\n':

  s.write("D??\r\n".encode())

      recibido1 = s.read(5)
```

```
    recibido=recibido1.decode()
if "DOF" in recibido:
sc.send(str.encode('DOF. OK\r\n'))
elif "DON" in recibido:
sc.send(str.encode('DON. OK\r\n'))
elif "D80" in recibido:
sc.send(str.encode('D80. OK\r\n'))
```

This communication is established from a USB to RS232 serial converter. If DOF is set (or read) means that the Noise Diode is disabled (0V), if DON is set (or read), the Noise Diode is on (15V), and if D80 is set (or read), the Noise Diode is switched (between 0 and 15 V) at 80 Hz of frequency.

setPhasecalMode and getPhasecalMode:

These functions are used to set and read in which state the Phasecal is. This communication is done via USB to RS232 serial converter.

```
elif str_rec [0:15] == 'setPhasecalMode':
a4=str_rec.split()
aux4=a4[1]
if aux4 < '0' or aux4 > '1':
   print("Introducir Modo 0 o Modo 1.")
else:
   if aux4 == '0':
      cad3="POF\r\n"
      s.write(str.encode(cad3))
      s.read(5)
      sc.send(str.encode('OK\r\n'))
elif aux4 == '1':
      cad4="PON\r\n"
    s.write(str.encode(cad4))
    s.read(5)
    sc.send(str.encode('OK\r\n'))

elif str_rec == 'getPhasecalMode\r\n':
     s.write("P??\r\n".encode())
     recibido2 = s.read(5)
     recibido3=recibido2.decode()
     if "POF" in recibido3:
        sc.send(str.encode('POF. OK\r\n'))
     elif "PON" in recibido3:
        sc.send(str.encode('PON. OK\r\n'))
```

If PON is set (or read) means that the Phasecal is on. However if POFF is set (or read), the Phasecal is turned off.

4. NEW CONNECTION BETWEEN THE X-BAND MODULE AND THE RASPBERRY.

After creating all the functions required for control and monitoring of modules, they had to be identified in the technical report IT – OAN 2007 – 22 and found in which backplane they were for the system to work as expected.

To do this, it has to be looked at the "monitoring and control signals" part of the report. All the control signals are located in backplane 1 distributed in its three columns a, b, and c.

The signal distribution is as follows:

FUNCTION	RASPBERRY PORT	NAME OF THE COTROL SIGNAL	CONNECTORS	
			ROW	PIN
get_xBand7650Locked()	15	LO_AL_7650	C	13
get_xBand450Locked()	13	LO_AL_450	C	14
cleanupExtRef()	11	NW_SW_MON0	C	11
cleanupLocked()	7	NW_SW_MON1	C	12
setAtt_R y getAtt_R	29	IF_RCP_VAR_ATT_B1	C	15
	31	IF_RCP_VAR_ATT_B2	C	16
	33	IF_RCP_VAR_ATT_B4	C	17
	35	IF_RCP_VAR_ATT_B8	C	18
	37	IF_RCP_VAR_ATT_B16	C	19
setAtt_L y getAtt_L	12	IF_LCP_VAR_ATT_B1	C	20
	16	IF_LCP_VAR_ATT_B2	C	21
	18	IF_LCP_VAR_ATT_B4	C	22
	22	IF_LCP_VAR_ATT_B8	C	23
	32	IF_LCP_VAR_ATT_B16	C	24
setBiasStage	36	MUX_TRT_B2	C	27
	38	MUX_TRT_B1	C	26
	40	MUX_TRT_B0	C	25
VG_MON	------	------	A	10
VD_MON	------	------	A	11
ID_MON	------	------	A	12

5. PHYSICAL IMPLEMENTATION OF THE NEW CONTROL SYSTEM.

After control signals were created, implemented, and configured, the module was physically deployed. To do this, the microIOC was removed and the Raspberry was inserted.

The Raspberry had to be put in the same module as its power supply and the created PCB powered between ±15V. *(Photo 2, Photo 3 and Photo 4)*

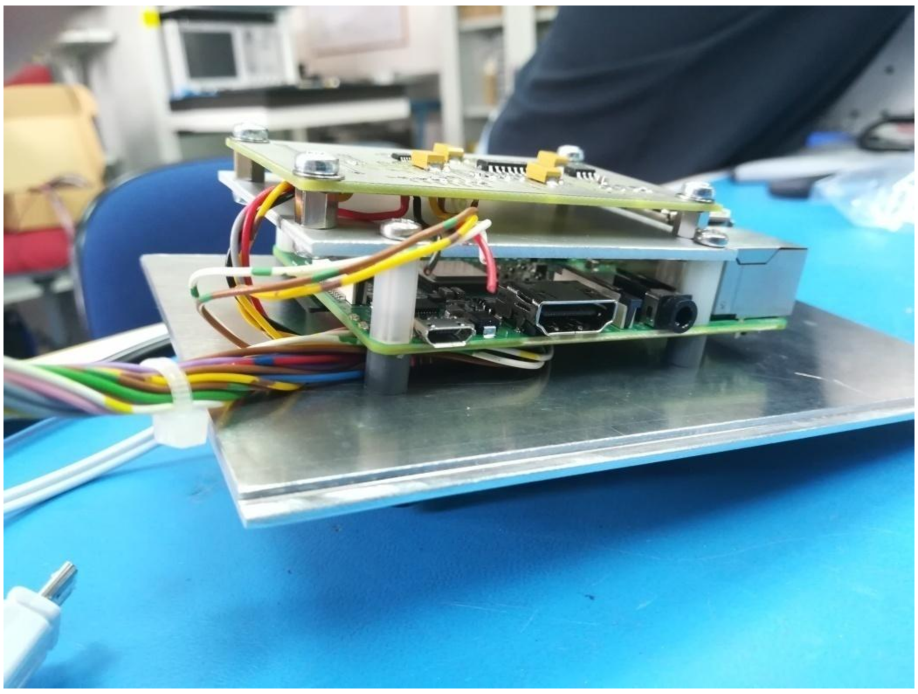


Photo 2

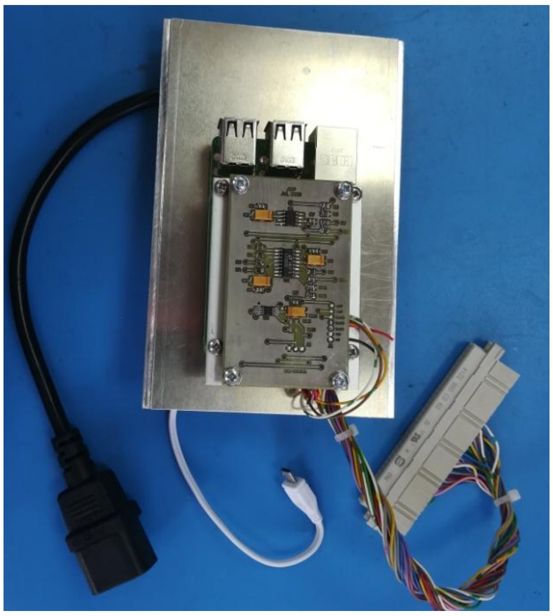
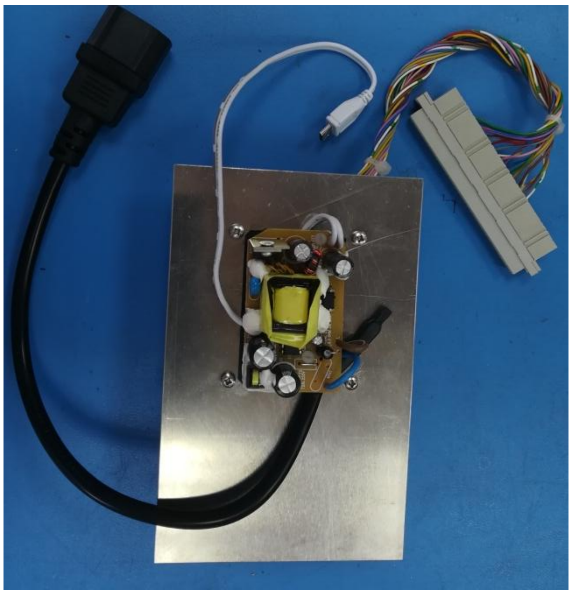



Photo 3 *Photo 4*

After the Raspberry was connected to its power and the created PCB, it had to be connected to backplane 1 as it was explained above. (Photo 5)

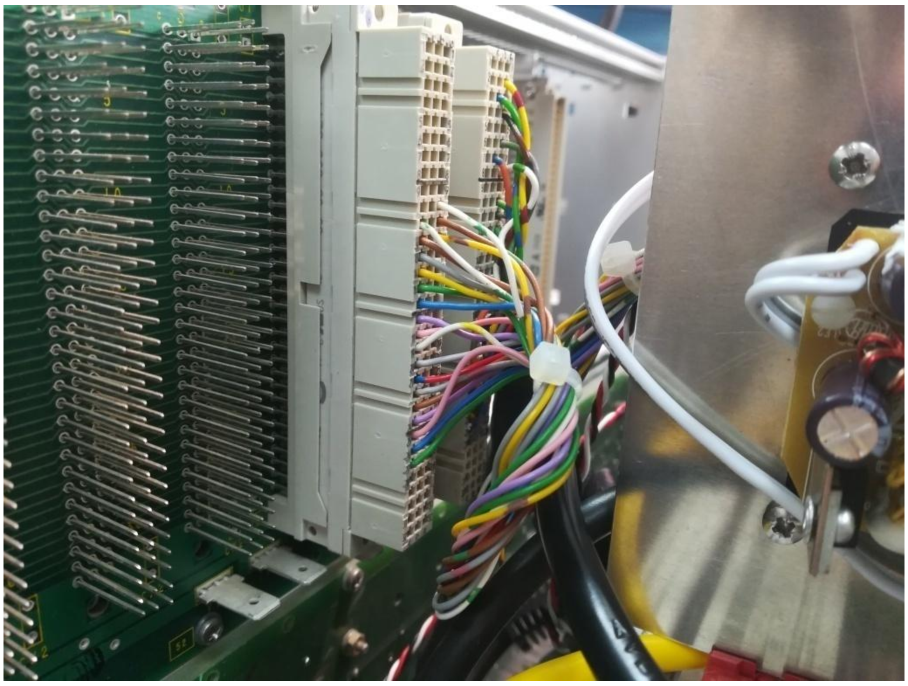


Photo 5

In order to power the PCB created between ±15V, the backplane 2 was used, from which +15V, -15 V and GND were taken. In addition, the ADC of the created PCB is powered by +5V, this voltage was taken from the Raspberry.

The final implementation of the control system can be seen in *Photo 6, Photo 7 and Photo 8.*

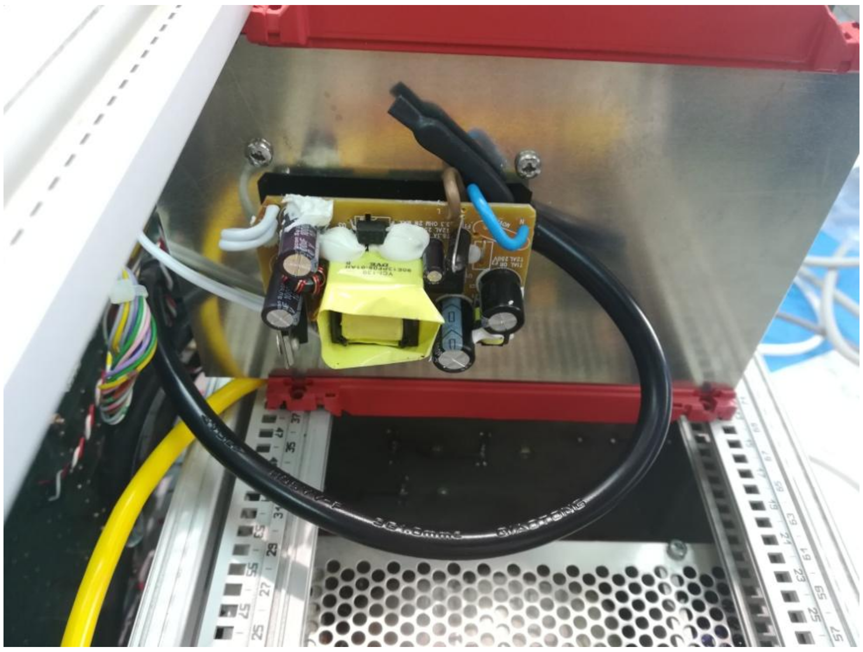


Photo 6

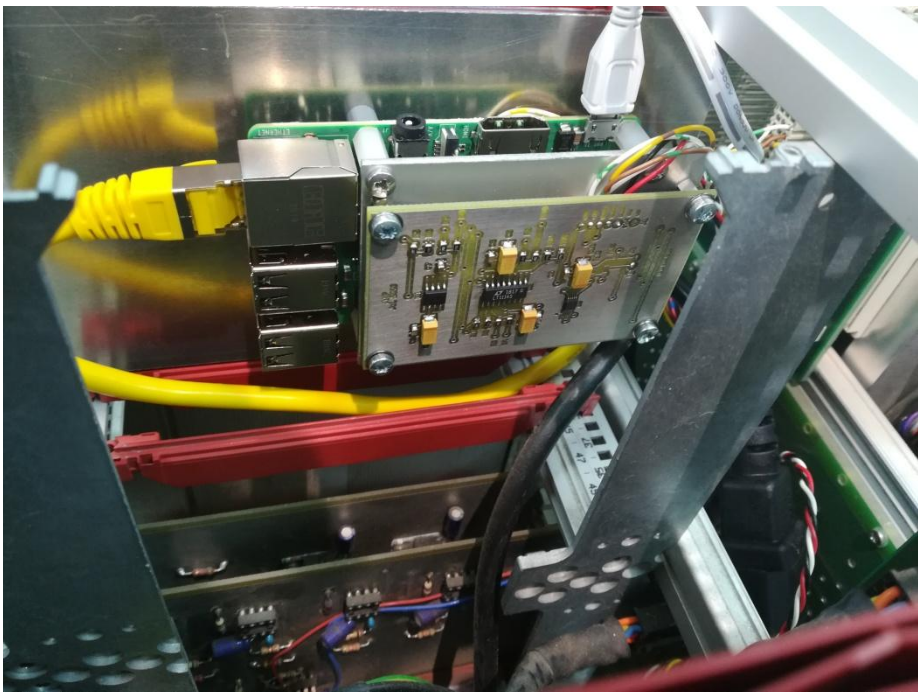


Photo 7

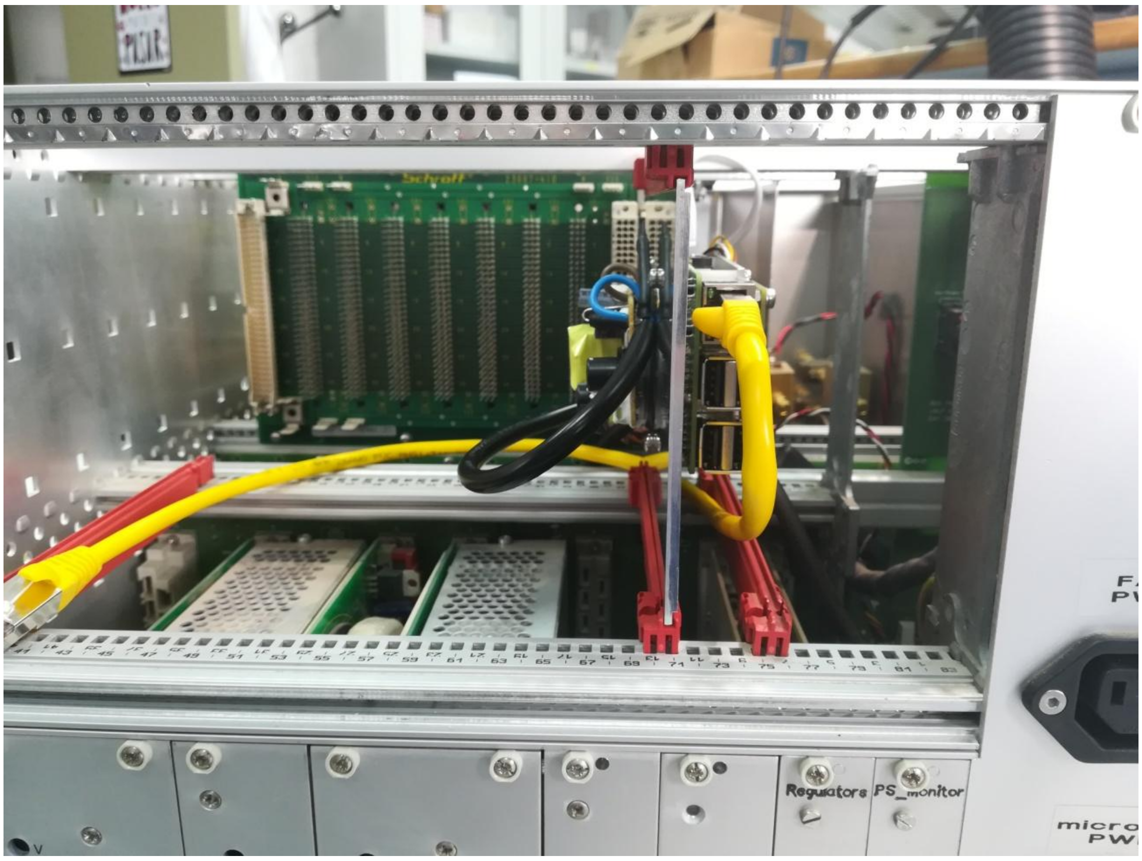


Photo 8

6. MODIFICATIONS MADE TO THE BIAS MONITOR CARD AND TO THE 'IF MODULE'.

After connecting the Raspberry, the module was checked for operation with the new control system.

It worked all correctly except the IF module and the monitoring voltages and polarization current.

In the IF module, the control system sets or reads the programmed attenuations in the variable attenuators. After checking these functions (setAtt_R, getAtt_R, setAtt_L, getAtt_L) the attenuation did not change, it was always 3dB.

To see what was going on, the IF module was opened and programmable attenuators were seen to be disconnected and a fixed 3dB attenuator was set. To solve the problem, fixed attenuators were removed and were reconnected programmable ones.

Having connected them, again, the attenuation control functions were rechecked and worked as expected. Attenuations from 0 to 31 dB could be set and read correctly.

It was also observed that the monitoring signals of polarization voltages and currents were not as expected. Therefore, Bias Monitor card module was opened and checked the values of currents and voltages in all multiplexers.

It was observed that the desired currents and voltages were set to RX_SELECT=1. So, for this, the 5 V of the Raspberry was taken and inserted in backplane 1, row C and pin 28.

Checking whether the voltages and currents were expected, it was seen that the signals had an unwanted offset. Therefore, the resistors used in the operational amplifiers of the Bias Monitor card were removed, since the voltages that entered these operational amplifiers were correct, but those that came out of them were unconfigured. By removing the resistors, the input and output signal of the operational amplifiers was the same and the polarization voltages and currents were properly monitored.