**Raytheon ISO Block Functional Test**

Setup, Maintenance & Troubleshooting

Contents

[Introduction – Operation & Flow 1](#_Toc445736031)

[Raspberry Pi Operating System Back-up 1](#_Toc445736032)

[Explanation of Arduino Usage & interaction with the I2C SDA (data) bus line 2](#_Toc445736033)

[Test Program Code – Flow diagram (Include reference to Raytheon test requirements) 2](#_Toc445736034)

[Test program Code - Revision Control Repository (github.com) 2](#_Toc445736035)

[Updating ISO Block firmware &/or configuring the PICkit 3 Programmer 2](#_Toc445736036)

[Hardware Modifications 2](#_Toc445736037)

[Making Python Scripts Globally Executable (+ creating desktop icon) 2](#_Toc445736038)

## Introduction

Refer to “ISO Block Test Requirements 030716” document provided by the customer. The document outlines the test operation and flow and the test program code follows this outline very close.

Documentation which details the software implementation can be found on the test PC under the “/home/pi/ISOBlock” directory or in the repository on github.com : <https://github.com/inovardes/ISOBlock.git>

Files pertaining to the design of the hardware and mechanical interface (the fixture) can be found on Inovar’s ecabinet server: ” 10.1.1.20:\Design Projects\Raytheon\ISO Block test fixture\Rev 2\Inovar Files”

## Raspberry Pi Operating System Back-up

* Cloning the SD card (create an image):

This step only needs to be done if an image hasn't already been saved for the Raspberry pi where the test test is run. An image has already been saved and can be found in one of 3 places: Inovar's ecabinet server, InoNet and the Raspberry Pi's external USB storage.

10.1.1.20:\Engineering\Test\Mckay\Customers\Raytheon\ISOBlock\Rpi\_Backup.img

Download and install Win32DiskImager on a Windows computer

Get everything set up just the way you want it on your Raspberry Pi, whatever you're using it for. Then shut down the Pi and remove the SD card. Insert the SD card into your computer. Startup Win32DiskImager, (If you're on OS X or Linux, you'll have to use the dd command as described here instead of these steps).

In the "Image File" box, enter the path of your soon-to-be image file. For example, I put mine in C:\Users\Whitson\images\myraspbmc.img

Under the "Device" box, select your SD card.

Click the "Read" button to create the image file from your card.

When it's done creating the image file, you can eject your SD card and put it back in your Raspberry Pi. Keep that IMG file in a safe place.

* Restoring a Corrupted SD card:

Now, if anything ever goes wrong with your Pi, you can restore your fully-set-up image using the instructions above in reverse:

Insert the SD card back into your computer.

Head to the start menu or screen and type "disk management." Open the disk management program and find your SD card in the list.

Right-click and delete all the partitions on your SD card. When it's empty, right-click on it and format it (it doesn't matter what filesystem you format it to, your computer just needs to recognize it).

Open Win32DiskImager again and browse for your image file. Select your device from the Device dropdown just as you did before. Click "Write" to write the image to the SD card.

When it finishes, eject the SD card and re-insert it into your Raspberry Pi. When you boot it up, it should be in the exact same state it was in when you first cloned the SD card. The only exception will be that you will need to update the test program files

## Explanation of Arduino Usage & interaction with the I2C SDA (data) bus line

The original test system design made use of the native I2C communication provided on the raspberry pi. As the design unfolded it was discovered that the raspberry pi used a subset of the I2C protocol, a higher level layer called the “SMBus”. The I2C commands within the SMBus library functions for Linux platforms that have I2C hardware installed are not completely compatible with other subsets of the I2C higher level variations. More specifically, the PIC used on the ISO Block utilizes the PMBus subset of the I2C protocol which introduces has some differences in the management of multibyte read/write operations. One of the significant problems surrounding the multibyte read command:

“read\_i2c\_block\_data(address, command, #of bytes to read)”

In this transaction the SMBus protocol sends two bytes (two frames: one contains the address and the other is a write request byte) at the beginning of the transmission which begin the read operation which is then followed by a stop and start bit (called a restart) which notifies the slave device that a multibyte read operation is requested. The problem with this is that the PMBus protocol is not expecting a stop bit after the first two bytes and only a start bit is expected. This essentially restarts the transmission and the data received is unknown data coming from an unknown register from the ISO Block PIC chip.

To overcome this problem an Arduino is used to mask over the unwanted stop bit sent from the raspberry pi. The Arduino watches the Bus data line (SDA) and counts the clock cycles in the first two bytes leading up to the stop and start bits. The Arduino is attached to some external hardware which controls the SDA line on the I2C bus and holds the line high for about 20u seconds until the stop bit passes. This works relatively well but does introduce some bugs in the test program which causes false failures related to communication failure. Within the test program source code, inside the following function:

“def RetryI2CReadMultipleBytes(command, bytesToRead):”

The Arduino sends a command to the Arduino, via an external interrupt, to notify of a upcoming multibyte read operation. The Arduino puts itself in a loop and another external interrupt attached to the clock line counts the clock edges and on a specific count pulls the SDA line high and then releases the line just before the start bit.

For more information, see the Arduino source code found at at Github: <https://github.com/inovardes/ISOBlock/Masking_SMBus_stop_bit>

Or located locally on the tester @: “/home/pi/ISOBlock/Masking\_SMBus\_stop\_bit/Masking\_SMBus\_stop\_bit.ino”

## Test Program Code – Flow diagram (Include reference to Raytheon test requirements)

## Test program Code - Revision Control Repository (github.com)

The test program code (.py files) and other documentation can be found on the test PC under the following directory: “/home/pi/Python Projects/ISOBLock”. If the files ever become lost or corrupted, visit github via the url: <https://github.com/mckaylund/ISOBlock.git> or search github.com for “ISOBlock”.

When modifications are made to the code, please update the repository by opening a terminal window on the PC and typing in the following commands:

1. cd “/home/pi/Python Projects/ISOBLock”
2. git add .
3. git commit –m “add comments about the update”
4. git push origin master
5. When prompted, type in the account credentials:
   1. username: inovardes@gmail.com[inovardes@gmail.com](mailto:inovardes@gmail.com)
   2. password: des@inovar1

## Updating ISO Block firmware &/or configuring the PICkit 3 Programmer

Please reference “PICkit\_3\_User\_Guide\_52116A.pdf” found on the web. If the document doesn’t exist on the web any longer, check the following repository: <https://github.com/inovardes/ISOBlock.git> (click the “Download ZIP” button to view files)

Follow the instructions outlined in steps 5.4 – 5.6 of the document mentioned above.

The new firmware file will be provided by the customer. A variable constant called “firmwareVersion” will need to be edited in the main test program code. This variable value must be changed in the ProgConstants.py file, found in “/home/pi/Python Projects/ISOBLock”. The value of this variable must match the value found in “BOARDID2LO” byte field (see “ISO Block Test Requirements 030716 “ section 5.4 for more details”.

## Hardware Modifications

ISO Block interface PCBA mods (Modifications apply only to rev 1):

1. J13, J14, J16, J17, J18 and J20 must have their pins sanded down to fit in their through holes. The PCB hole diameter is too small for the part’s pins.
2. Cut traces on J21, J12 and J19 between pins 1 & 2 (there are two sets of pin 1 & 2, cut both). Then, cut the VOUT\_RTN\_KELVIN trace that connects to pin 2 of J21 and solder a jumper to J21, pin 1, bypassing J22, pin 2. Also, cut the VIN\_RTN\_KELVIN trace that connects to pin 2 of J12 and solder a jumper to J12, pin 1, bypassing J12, pin 2. These changes are to overcome a pin numbering error in the part creation.
3. Solder a jumper across R1, effectively shorting across this resistor. The fan won’t work otherwise.
4. Cut the +5V trace attached to pin 1 of R5. Next, solder a jumper wire from RPi +3.3V to pin 2 of Q1 (the source pin). A design mistake was made which prevented the MOSFET from turning on since Vgs couldn’t overcome the threshold voltage.
5. Solder a jumper wire from J22 pin 8 to pin 1 of Q1 (the gate pin). An error occurred during the creation of the RPi header and pin 8 was not attached to the header properly and, as a result, the SYNC\_nEnable pin was not connected

## Making Python Scripts Globally Executable (+ creating desktop icon)

* .py Executable:

Normally, in order to run a Python program you have to tell the Python software to open the file. However it is possible to execute the file without having to call upon Python first. This allows you to call upon your own programs (that you created in Python) at the terminal by simply typing it's name.

First you need to tell Linux to mark your Python file as executable, which means that the file is a program. For this example the target file to be made executable will be called example.py. When you come to doing it yourself simply replace this with your own file name. We use the chmod +x command to make a file executable. In the terminal type the following:

chmod +x example.py

You can now try running the program directly by typing:

./example.py

Even though you didn't call upon Python the program should still run the same as if you'd typed python example.py. The program can only be run by calling it with it's full location /home/pi/example.py or from the current directory by using ./ as the location.

To make the file accessible in the same way as any other command in the terminal, it needs to be copied (using the command cp) to /usr/local/bin with the following command:

sudo cp example.py /usr/local/bin/

With the file now located in /usr/local/bin it can be executed from any directory by simply typing it's name. Try changing to another directory and then run the program again by typing the following:

example.py

To make your custom-made programs seem more like native utilities, you can rename (using the command mv) them to remove the .py file extension. To change example.py in this way type the following line at the terminal:

sudo mv /usr/local/bin/example.py /usr/local/bin/example

Now the program can be run by simply typing example at the terminal!

* Creating a Desktop Icon in Linux: