12. Conversational Software

NOTE - September 27, 2023: Having achieved the results described below in the log entry of 9/27/2023 I am pushing this hobby project down onto my stack (so-to-speak) in order to give my time to several other projects and activities that have fallen a bit to far by the wayside. Do not yet have a sense of when I'll return back to this project.

Purpose: Take the software to the next level so that it is reasonably suitable for long-term field test monitoring.

Goal: Create software in both the ATTiny84 (ATtiny) and the Raspberry Pi (RPi) that enables the RPi to send commands to the ATTiny for it to carry out. Establish a comms protocol that enables a back-and-forth 'conversation' of sorts between the two devices.

> **Sub-Goal**: Redesign the code on the ATTiny84 side to make it '*event-driven*' and '*non-blocking*' in order to facilitate it being responsive to commands from the RPi. This will also allow for more responsive health and error signalling via LEDs on the sensor board.

Design Notes:

Assumptions:

1. The raspberry pi will be plugged in to the mains and always on and available as seen from the perspective of the moisture sensors.
2. Raspberry Pi functions as central server for the sensor network.
3. Raspberry Pi provides instructions to the sensors. A very limited set of possible instructions, but the RPi is in control. Possible instructions:
    * Send battery level.
    * Got to sleep for x minutes.
    * Send capacitance readings continuously until I say to stop.

Communications Structure:

Here is how I'm thinking it goes. We start with the raspberry pi sniffing the airwaves just listening for a moisture sensor to tell it: "Hey raspberry pi I'm here, I'm awake, and I'm awaiting instructions." Then the RPi sends an instruction. The moisture sensor acknowledges the instruction, which might include in the ACK results from performing that instruction - e.g., a moisture reading. Then the RPi acknowledges the moisture sensor's completion of the instruction. Then the cycle can then begin again.

Sensor: I'm awake.
RPi: Acknowledged, send me capacitance reading.
Moisture sensor: [performs a soil moisture reading action] Capacitance is ###.
RPi: Acknowledged, sleep for 1 hour.
Moisture sensor: Acknowledged. [Goes to sleep]

Considerations:

* *Comms Errors*.Now what I also have to consider is that either device may not hear the commands or acknowledgements. So there has to be some facility for repetition and time-outs. But I also know that the RPi seems to pretty much always get the messages from the ATtiny even though the ATTiny will often not hear the acknowledgement back from the RPi. So I have to account for that situation as well.

  - The fall-back for the moisture sensor sleep time, in the event of an ack timout, will simply be that the moisture sensor will use its last known sleep time. On the RPi side it will also know that value so if the RPi never gets an acknowledgement from the moisture sensor that it heard the latest sleep command the raspberry pi will also fall back to the prior one.

Key References --

Raspberry Pi Login --

Pi user is: pi. So SSH login is:

```
ssh pi@[ip address of the RPi on my LAN]
```

password to the pi is: `pi9012`

NOTE: I have created an alias command in my .bash_profile for the login: '**`pilogin`**'

And then once in, get into the working directory for the nRF24 utilities and programs:

```
cd /home/rf24libs/RF24/examples_linux/build
```

The run the RPi receiving program:

```
./RPi_CapDataReceive
```

Good reference for running the process detached from terminal: https://linuxconfig.org/detach-process-program-from-current-shell-to-keep-it-alive-after-logout

Based on this, the command to run in an SSH terminal to keep the RPi programming forever, even after detaching the SSH session, is:

```
nohup /home/rf24libs/RF24/examples_linux/build
/RPi_CapDataReceive  &
```

To Modify RPi side .cpp code --

Use Kate on my desktop.

Then copy to the RPi using below command:

```
scp /home/jroc/Dropbox/projects/MoistureSensor/Software
/RPi/RPi_CapDataReceive.cpp pi@[ip-address-of-the-RPi-on-my-
LAN]:/home/rf24libs/RF24/examples_linux/
```

NOTE: My bash alias command for the above is: '**`picodecopy`**'

Then on the RPi, cd into the build directory and execute 'make'

BUT if this is the first iteration of the program, you need to first add it to CMakeLists.txt in the examples_linux directory (above the build directory).

NOTE: *All the code is under git version control*. So the executables for both the ATTiny and RPi are on my desktop.

GitHub Repo: https://github.com/JeffRocchio/Wireless-Soil-Moisture-Sensor

Next Step: Several steps upon completion of this milestone: *One*, Fabricate a few different capacitor sensors to test. *Two*, modify code to measure and TX moisture (capacitance) data once/hour instead of continuously. *Three*, setup sensor in a real flower/plant pot in my house and run field tests on the sensors.

9/27/2023 --

Results: **Success**. Today I implemented writing the sensor readings to a log file on the RPi (which can be viewed or monitered via an SSH terminal session using `cat` or `tail` commands). I set the logging frequency to once every 2 hours for now. So now I shall deploy this into a real plant pot and just see how it goes for awhile.

ATTiny Code: `tiny84_SensorAsSlave`. No changes made to this code today.

RPi Code: `RPi_CapDataReceive.cpp`. Added function to write to the log. Added logic to check time and make a log entry once every 2 hours.

Documentation: No Changes, except for this log entry itself.

Git: To merge the Issue-1 branch back into Main I used the below strategy so as to avoid having to manually deal with all the merge conflicts of a normal merge. The below approach had the effect of just taking all the changes made in Issue-1 branch and overwriting the files in main with those updates so that Main becomes a copy of Issue-1:

```
You should be able to use the “ours” merge strategy to overwrite master with seotweaks like
this:

git checkout master
git pull
git checkout seotweaks
git merge -s ours master
git checkout master
git merge seotweaks

The first two steps are a useful precaution to ensure your local copy of master is up-to-date.
The result should be that your master is now essentially seotweaks.

(-s ours is short for --strategy=ours)

From the docs about the 'ours' strategy:
```

```
    This resolves any number of heads, but the resulting tree of the merge is always that of
the current branch head, effectively ignoring all changes from all other branches. It is meant
to be used to supersede old development history of side branches. Note that this is different
from the -Xours option to the recursive merge strategy.

Update from comments: If you get fatal: refusing to merge unrelated histories, then change the
second line to this: git merge --allow-unrelated-histories -s ours master

See full post @ https://stackoverflow.com/questions/2862590/how-to-replace-master-branch-in-git-
entirely-from-another-branch
```

Next Step: Test ability for ATTiny to handle a command sent by the RPi in one of it's ack packets.

9/26/2023 --

Results: **Success**. I determined the trigger for Issue-1 and created a working version of the ATTiny and RPi code by re-redefining the RPi ackPacket struct on both the ATTiny and RPi. For test purposes I simplified it down to just two fields, and made each field a uint32_t type. My working theory is that the issue has to do with byte alignment/boundaries relative to the ATTiny/nRF24 chipset packing and unpacking the structs. See Issue# 1 on the github for further details.

9/25/2023 --

Results: **Failure**. Update to 9/24 notes. While I thought 9/24 was a success, I later noticed that I had a pernicious bug that caused the two radios to stop talking to each other (continuous error #2 reported on the ATTiny side). I logged this as Issue-1 on the github for this project. I created branch Issue-1 to work on this.

9/24/2023 --

Results: **Success** in implementing the base structure for RPi to send commands back to the ATTiny.

ATTiny Code: `tiny84_SensorAsSlave`. Modified rxAck struct to make it suitable for receiving commands back from the RPi.

RPi Code: `RPi_CapDataReceive.cpp`. Modified the Ack struct to match the ATTiny code change noted above. Also eliminated any timeout, so the RPi runs in an infinite loop. Also eliminated having the user initiate 'receive' mode. So now when the program starts up it immediately goes into receive mode, inside an infinite loop, doing it's back-and-forth with the ATTiny forever. This will facilitate auto-start upon RPi boot when I implement that later.

Documentation: No Changes, except for this log entry itself.

Next Step: Modify RPi code to log sensor readings into a text log file once per hour. And once that has

tested successfully implement auto-start for the RPi side program.

GitHub: Above code mods and documentation has been commited to the GitHub repo @ https://github.com/JeffRocchio/Wireless-Soil-Moisture-Sensor

9/20/2023 --

Results: **Success** with non-blocking architecture + proving I can get get data back from the RPi in the ACK packet.

However, I am getting a lot of "error-2" - meaning ATTiny nRF24 chip sends the data out but doesn't see the RPi ack coming back. When I first implemented the non-blocking RadioComms object I was getting nearly perfect Tx/ACK results. But I had bugs in the cap reading object whereby I was getting a divide-by-zero error, showing up as 'inf' (infinity) in the cap reading over to the RPi. Yet the transmissions were error-free. After I fixed the cap reading bug now the transmissions are highly error prone. Is this a coincidence? Don't know.

ATTiny Code: The code is now highly modularized; and non-blocking. From a code memory usage perspective this is pretty highly inefficient. As of this date we are using 71% of the code memory space on the ATTiny. Although fully 10% of that is consumed by one math operation(!): The calculation of capacitance from the pin-voltage-pulse. See footnote #3 in capSensor.ino. The code modules, as of this data, are:

tiny84_SensorAsSlave.ino Dispatcher.h / .ino RadioComms.h .ino	These classes are meant to be specific to this project. They are not designed general purpose reusable classes.
CapSensor.h / .ino	Could *maybe* be a general-use class.
ErrorFlash.h / .ino HeartBeat.h / .ino	Designed as a reusable general-purpose class I could use in all my Arduino projects.

RPi Code: I modified this code to 'hard code' the respective radio addresses so as not to ask at run time 'which radio this is.' I also removed the timeout so that the RPi side would run continuously.

Documentation: Made a few updates to the 1-page 'protocol' diagrams for non-blocking classes. See in the folder: *Software/Documentation*