4B25 Coursework 4 - Accelerometer-Based Activity Classifier with Uncertainty Estimates

Name: Leon Brindley

College: Gonville and Caius

CRSid: lpb32

Summary

This project determines the number of steps taken by an individual over a **10-second** period. It then infers whether they are **stationary**, **walking** or **running** from this. The classification is determined using a five-step algorithm and a **Freescale MMA8451Q** accelerometer. The design was implemented using a **Freescale FRDM-KL03Z** development platform, which contains its own integrated MMA8451Q.

Step 1: Magnitude Calculation

Firstly, the **14-bit** acceleration measurements from the X, Y and Z axes are parsed using bit shift operations. Any unexpected **type A uncertainty** can be identified by statistically analysing the variance of the results and excluding any points outside of an acceptable confidence interval. The magnitude of these three readings is calculated using Pythagoras' theorem across three cartesian axes. The **sqrt()** function in **math.h** is too large relative to the FRDM-KL03Z's **2 KB** of SRAM, so an integer-based implementation using the Newton-Raphson method is executed in **sqrtInt()** instead. By default, **19** results are stored in the **AccelerationBuffer**.

Step 2: Low-Pass Filter

Secondly, the AccelerationBuffer is multiplied by an array of coefficients (**LPFWeights**) to low-pass filter the signal. This removes high-frequency noise, so an accurate speed can be calculated. By default, **39** results are stored in the **LPFBuffer**.

Step 3: Frequency Calculation

Thirdly, the frequency of the signal is extracted by counting the number of **inflection points** (**maxima** and **minima**) in a known time period. The function **simpleDiff()** compares the readings on either side of each array element to identify these inflection points. To ensure that LPFBuffer[0] and LPFBuffer[38] are not excluded, the **last** and **second-to-last** elements of each cycle are retained for comparison.

Step 4: Step Counting and Speed Calculation

Fourthly, the **number of steps** in a particular period of time is extracted from the data. By estimating the length of each step (equal to 0.415 and 0.413 times the user's height for men and women, respectively), the algorithm also estimates

the **speed** at which the device is moving. Please note that the speed results will only become valid once the AccelerationBuffer and LPFBuffer has been filled at least twice (due to the low-pass filter, hence why steps 3, 4 and 5 are ignored after the first cycle is concluded).

Due to the strict memory requirements of the FRDM-KL03Z, only 39 elements are stored in the AccelerationBuffer and LPFBuffer. Waveforms with **different frequencies** can still have the **same number of inflection points** over this short time period, causing a significant rounding error. Hence, the number of inflection points is retained over **multiple** 39-element cycles using the variable **cumulativeInflectionPoints**. If the program is run for longer, its rounding error decreases substantially (and the speed calculation is more accurate). It is recommended that steps should be counted for at least **10 seconds** for the speed to be determined with sufficient accuracy.

Step 5: Activity Classification

Finally, the aforementioned speed calculation is converted to an activity (**ActivityRunning**, **ActivityWalking** or **ActivityStationary**). Long and Srinivasan found that the characteristic velocity of running, the average velocity with equal amounts of walking and running (a running fraction of 0.5), equals about **2.2 m/s** (**7.92 km/hr**) on average, and Saibene and Minetti found that the characteristic velocity calculated in previous literature varied between **1.80** and **2.50 m/s** (**6.48** and **9.00 km/hr**, respectively). By assuming that the characteristic velocity of individual users follows a uniform distribution between these two figures, the activity (and its confidence level) is classified.

```
1. Acceleration Magnitude: 707mms^-2.
2. AccelerationBuffer[38] = 707, LPFBuffer[38] = 48529.
simpleDiff(): 53104 < 53988 and 53104 < 55147 - MINIMUM detected in LPFBuffer[11].
simpleDiff(): 52869 > 52750 and 52869 > 52276 - MAXIMUM detected in LPFBuffer[23].
simpleDiff(): 47503 < 47659 and 47503 < 47937 - MINIMUM detected in LPFBuffer[35].
3. numberOfInflectionPoints: 3, cumulativeInflectionPoints: 3.
4. Distance (mm): 1490 / Time (ms): 2400 = Speed (mm/s): 620, Speed (m/hr): 2235.
5. Activity = Walking (Confidence Level = 67 Percent), Stationary (Confidence Level = 33 Percent).
EXECUTION TIME of classiferAlgorithm() = 16874 - 16776 = 98ms.
```

Figure 1: Terminal Output When Walking

The same logic was used to discern between the walking and stationary states, as a threshold of 0 m/s will always be exceeded due to fluctuations in the user's position or the measurement setup. Graham et al. have shown that the mean walking speed of older adults in geriatric rehabilitation settings is **0.23 m/s** (**0.828 km/hr**), so this is used as the lower bound. Meanwhile, Murtagh et al. have shown that the typical walking speed of healthy adults when purposefully walking slowly is **0.82 m/s** (**2.952 km/hr**), so this is used as the upper bound.

Configuration: MMA8451Q Registers

The MMA8451Q accelerometer is configured by writing to the registers **F_SETUP**, **CTRL_REG1**, **HP_FILTER_CUTOFF** and

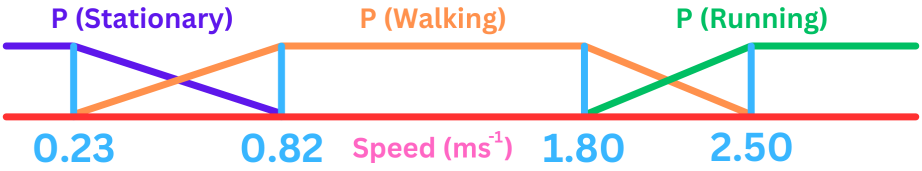

Figure 2: 4B25 Uncertainty Graph

XYZ_DATA_CFG. The structures of each register are explained in Freescale's MMA8451Q datasheet.

Firstly, F_SETUP is set to **0x00** to disable the internal first-in, first-out (FIFO) buffer, as **AccelerationBuffer** is used instead.

Secondly, CTRL_REG1 is set to **0x05** to select a **14-bit** resolution and activate the **reduced noise mode**. In contrast, if the **F_READ** bit is set to **1**, then an **8-bit** resolution can be used for faster data transfer.

Thirdly, HP_FILTER_CUTOFF is set to **0x03** to ensure the MMA8451Q's high-pass filter is unbypassed and configure its cut-off frequency as **2 Hz**. The output data rate (ODR) is also set to the default value of **800 Hz**. This high-pass filter removes the acceleration due to **gravity** (g), which forms a DC offset.

Finally, XYZ_DATA_CFG is set to **0x12** so the high-pass filter is enabled and the accelerometer's full-scale range equals **8g** (but note that if the LNOISE mode is active, the range is effectively limited to **4g** anyway).

Configuration: Sampling Rate

It is insufficient to simply configure the time period (for example, **20ms**) using the argument of **OSA_TimeDelay()**, as this does not account for the time required to execute the data processing steps. Furthermore, this duration can vary between samples (for example, because more iterations are required in the **sqrtInt()** function using the Newton-Raphson method).

Fortunately, this execution time can be recorded by calling **OSA_TimeGetMsec()** before and after running the necessary code (and calculating the difference). Subsequently, this is subtracted from the nominal time period before being fed into the OSA_TimeDelay() function.

Configuration: Low-Pass Filter Cut-Off Frequency

To set the low-pass filter frequency, you can use a program such as WinRFCalc. You must make sure that the number of taps is **odd** and that the sampling frequency is **at least twice** the cut-off frequency (to fulfil the Nyquist Criterion). The cut-off frequency should be **greater than** the MMA8451Q's **output data rate** (**ODR**) divided by **2**. Therefore, for the default ODR of **800 Hz**, the cut-off frequency must equal at least **400 Hz**. The default LPFWeights array gives **39** taps, a maximum attenuation of **130dB** and a cut-off frequency and

sampling frequency of **450 Hz** and **16,384 Hz**, respectively. These parameters were chosen as the resulting coefficients were all positive and there were minimal rounding errors after scaling.

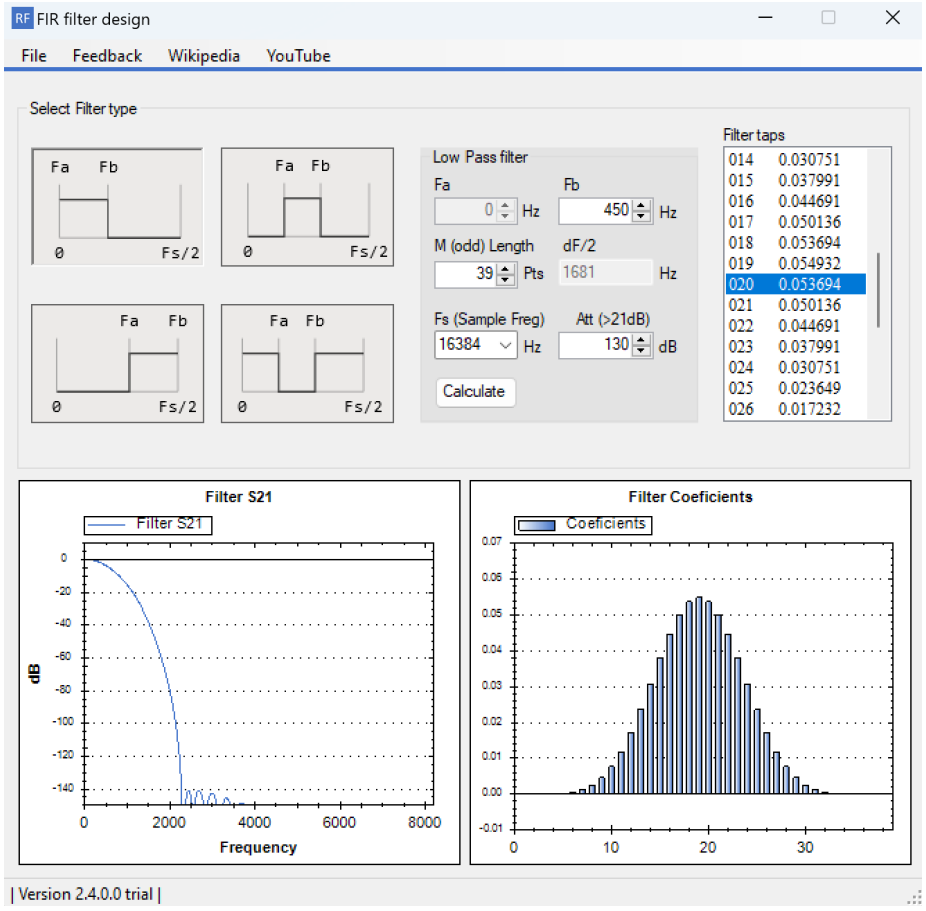

Figure 3: WinRFCalc39

In contrast, the more compact LPFWeights array gives **19** taps, a maximum attenuation of **120dB** and a cut-off frequency and sampling frequency of **450 Hz** and **8,192 Hz**, respectively.

Low-Pass Filter Effect

Below are some raw acceleration magnitudes from the MMA8451Q. During this test, the FRDM-KL03Z was gently moved from side to side. If using the signal without low-pass filtering, the **simpleDiff()** function would detect **9** inflection points.

After this data was low-pass filtered, the **simpleDiff()** function only detected

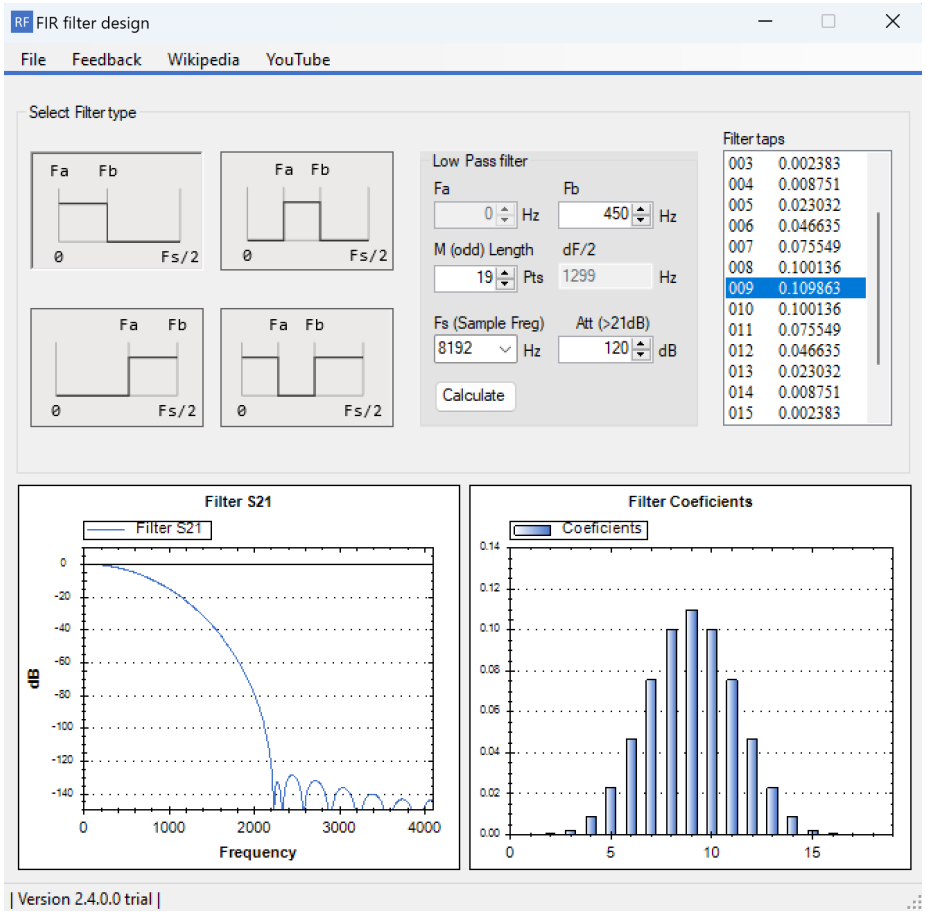

Figure 4: WinRFCalc19

3 inflection points. The sinusoidal shape below was far more representative of the activity being performed.

If the **LPFWeights** array is changed to **19** elements long, the **SAMPLING_PERIOD** should be **doubled** to maintain a roughly equal test duration. The resulting waveform is less smooth, but often an identical number of inflection points are detected (when an approximately identical activity was performed).

Other Sources of Uncertainty

As shown in the report, there was a significant offset when the accelerometer was stationary, and so the algorithm was rarely 100% confident that the device wasn't moving. Furthermore the standard deviation of either 347 or 572 m/hr, depending on the movement being performed, was large and led to some erroneous classifications.

As fixed-point computations are widely used to eliminate any rounding errors, the main sources of error are due to variations in people's characteristic transition velocities (as accounted for in **step 5** of the algorithm) and variations in people's step lengths.

The variation in people's step lengths has not been accurately characterised in prior literature, and the 0.414 * HEIGHT estimate is frequently used across the industry, so this form of uncertainty was negated. Furthermore, this result can vary with numerous factors, such as the presence of rain, any incline of the ground and your footwear, so the calculation of the associated type B uncertainty is unfeasible.

Finally, the confidence level is also affected by the duration of the experiment if the full runtime is used in the speed calculations. In the example below, if the time period equals 2 seconds, then for the same number of inflection points (**3**), the time can vary between **2** and **4** seconds. In contrast, if **7** inflection points are covered, the time can vary between and **6** and **8** seconds (causing a smaller **percentage** uncertainty).

To avoid this, the duration excludes any time before the first inflection point or after the final inflection point, and the **cumulativeInflectionPoints** variable is decremented to avoid overestimating the speed due to starting and finishing on an inflection point.

OLED Display

The function **printCharacter()** in **devSSD1331.c** is used to display the numbers and units of the measurements. This calls the function **printLine()** in **devSSD1331.c** to display the numbers **0 to 9**, the letters **K**, **M**, **H**, **R**, **W**, **A**, **L**, **I**, **N**, **G**, **U**, **S**, **T** or **B**, a full stop (**.**) or a forward slash (**/**). This allows for the classifications **RUNNING**, **WALKING** and **STILL** to be printed, as well as the unit **KM/HR**. Alternatively, for bolder text with a width of greater than one pixel, the function **printRect()** can be used instead. Further work is

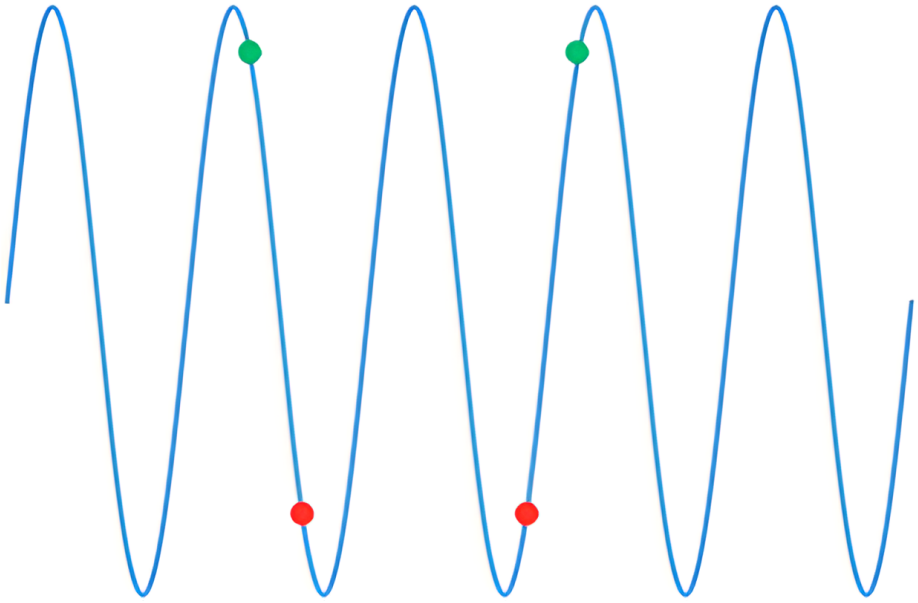

Figure 5: Uncertainty From Sampling Times

needed to finish the pedometer display, as the algorithm itself was prioritised instead.

File Structure

`src/boot/ksdk1.1.0/activityClassifier.c` - Implements the five steps of the activity classifier algorithm as explained above.

`src/boot/ksdk1.1.0/devMMA8451Q.c` - Driver for communicating with MMA8451Q accelerometers. For example, the function **configureSensorMMA8451Q()** will write to the MMA8451Q configuration registers explained above. The I2C address of the MMA8451Q must be set to **0x1D** when calling initMMA8451Q in boot.c with the FRDM-KL03Z. When using this development board, as opposed to the more complicated Warp platform, the MMA8451Q's operating voltage cannot be dynamically altered, so the **operatingVoltageMillivolts** member variable has no effect.

`src/boot/ksdk1.1.0/devSSD1331.c` - Driver for communicating with SSD1331 OLED displays. For example, the function **printLine()** will print a 2D line between two coordinates in a colour defined by its blue, green and red content. Meanwhile, the function **printRect()** will print a solid rectangle bounded by two coordinates in a colour defined by its fill and its line (border). This source file also includes the function **printCharacter()** so the final results can be printed on an SSD1331 OLED display. These numbers are implemented by calling the function **printLine()** to mimic multiple seven-segment displays alongside each

other.

`src/boot/ksdk1.1.0/boot.c` - Contains the **main()** function that runs when the program boots. For example, the function **classifierAlgorithm()** is repeatedly called here using a for loop to execute the activity classifier algorithm over multiple cycles.

`src/boot/ksdk1.1.0/config.h` - Configures the Warp firmware for the platform in question (in this case, the FRDM-KL03Z). For the activity classifier algorithm to work, **WARP_BUILD_ENABLE_DEVMMA8451Q** must be set to **1**. Furthermore, as two bytes are required for each axis (X, Y and Z), **kWarpSizesI2cBufferBytes** must be set to **at least 6**.

Baseline firmware for the Warp family of hardware platforms

This is the firmware for the Warp hardware and its publicly available and unpublished derivatives. This firmware also runs on the Freescale/NXP FRDM KL03 evaluation board which we use for teaching at the University of Cambridge. When running on platforms other than Warp, only the sensors available in the corresponding hardware platform are accessible.

Prerequisites: You need an arm cross-compiler such as `arm-none-eabi-gcc` installed as well as a working `cmake` (installed, e.g., via `apt-get` on Linux or via MacPorts on macOS). On Ubuntu, the package you need is `gcc-arm-none-eabi`. You will also need an installed copy of the SEGGER JLink commander, `JlinkExe`, which is available for Linux, macOS, and Windows (here are direct links for downloading it for macOS, and Linux tgz 64-bit).

1. Compiling the Warp firmware

First, edit setup.conf to set the variable `ARMGCC_DIR` and `JLINKPATH`. If your `arm-none-eabi-gcc` is in `/usr/local/bin/arm-none-eabi-gcc`, then you want to set `ARMGCC_DIR` to `/usr/local`. In the following, this `README.md` will refer to the top of the repository as `$TREEROOT`. On some platforms, you might need to also, in addition, set the `ARMGCC_DIR` environment variable in your shell (using `setenv` or `export` as appropriate), to point to the same path as you set in setup.conf.

Second, edit `tools/scripts/glaux.jlink.commands` and `tools/scripts/warp.jlink.commands` to replace `<full-path-to-warp-firmware>` with the full path to your Warp firmware directory.

Third, build the Warp firmware by

```
make warp
```

Fourth, load the Warp firmware to hardware by

`make load-warp`

To build for the Glaux variant, use `make glaux` and `make load-glaux` in steps three and four instead.

The build process copies files from `src/boot/ksdk1.1.0/` into the `build/`, builds, and converts the binary to SREC. See `Warp/src/boot/ksdk1.1.0/README.md` for more. *When editing source, edit the files in `src/boot/ksdk1.1.0/`, not the files in `build` location, since the latter are overwritten during each build.*

To connect to the running hardware to see output, you will need two terminal windows. In a separate shell window from the one in which you ran `make load-warp` (or its variants), launch the JLink RTT client See note 1 below:

`JLinkRTTClient`

2. Using the Warp firmware on the Freescale FRDMKL03 Board

The SEGGER firmware allows you to use SEGGER's JLink software to load your own firmware to the board, even without using their specialized JLink programming cables. You can find the SEGGER firmware at the SEGGER Page for OpenSDA firmware.

To build the Warp firmware for the FRDM KL03, you will need to modify this line in `src/boot/ksdk1.1.0/config.h`.

3. Editing the firmware

The firmware is currently all in `src/boot/ksdk1.1.0/`, in particular, see `src/boot/ksdk1.1.0/warp-kl03-ksdk1.1-boot.c` and the per-sensor drivers in `src/boot/ksdk1.1.0/dev*.[c,h]`.

The firmware builds on the Kinetis SDK. You can find more documentation on the Kinetis SDK in the document doc/Kinetis SDK v.1.1 API Reference Manual.pdf.

The firmware is designed for the Warp and Glaux hardware platforms, but will also run on the Freescale FRDM KL03 development board. In that case, the only sensor driver which is relevant is the one for the MMA8451Q. For more details about the structure of the firmware, see src/boot/ksdk1.1.0/README.md.

4. Interacting with the boot menu

When the firmware boots, you will be dropped into a menu with a rich set of commands. The Warp boot menu allows you to conduct most of the experiments you will likely need without modifying the firmware:

```
[ *             W   a   r   p   (rev. b)            * ]
[                     Cambridge / Physcomplab                    ]
```

```
    Supply=0mV, Default Target Read Register=0x00
    I2C=200kb/s,    SPI=200kb/s,    UART=1kb/s, I2C Pull-Up=32768

    SIM->SCGC6=0x20000001       RTC->SR=0x10        RTC->TSR=0x5687132B
    MCG_C1=0x42         MCG_C2=0x00     MCG_S=0x06
    MCG_SC=0x00         MCG_MC=0x00     OSC_CR=0x00
    SMC_PMPROT=0x22         SMC_PMCTRL=0x40     SCB->SCR=0x00
    PMC_REGSC=0x00          SIM_SCGC4=0xF0000030    RTC->TPR=0xEE9

    0s in RTC Handler to-date,  0 Pmgr Errors
Select:
- 'a': set default sensor.
- 'b': set I2C baud rate.
- 'c': set SPI baud rate.
- 'd': set UART baud rate.
- 'e': set default register address.
- 'f': write byte to sensor.
- 'g': set default SSSUPPLY.
- 'h': powerdown command to all sensors.
- 'i': set pull-up enable value.
- 'j': repeat read reg 0x00 on sensor #3.
- 'k': sleep until reset.
- 'l': send repeated byte on I2C.
- 'm': send repeated byte on SPI.
- 'n': enable SSSUPPLY.
- 'o': disable SSSUPPLY.
- 'p': switch to VLPR mode.
- 'r': switch to RUN mode.
- 's': power up all sensors.
- 't': dump processor state.
- 'u': set I2C address.
- 'x': disable SWD and spin for 10 secs.
- 'z': dump all sensors data.
Enter selection>
```

Double echo characters

By default on Unix, you will likely see characters you enter shown twice. To avoid this, do the following: - Make sure you are running `bash` (and not `csh`) - Execute `stty -echo` at the command line in the terminal window in which you will run the `JLinkRTTClient`.

Introduction to using the menu

You can probe around the menu to figure out what to do. In brief, you will likely want:

1. Menu item `b` to set the I2C baud rate.
2. Menu item `r` to switch the processor from low-power mode (2MHz) to "run" mode (48MHz).
3. Menu item `g` to set sensor supply voltage.
4. Menu item `n` to turn on the voltage regulators.
5. Menu item `z` to repeatedly read from all the sensors whose drivers are compiled into the build.

NOTE: In many cases, the menu expects you to type a fixed number of characters (e.g., 0000 or 0009 for zero and nine) See note 1 below. If using the ***`JLinkRTTClient`****, the menu interface eats your characters as you type them, and you should not hit RETURN after typing in text. On the other hand, if using* ***`telnet`*** *you have to hit return.*

If you see repeated characters, you can set your terminal to not echo typed characters using `stty -echo`.

Example 1: Dump all registers for a single sensor

- `b` (set the I2C baud rate to `0300` for 300 kb/s).
- `g` (set sensor supply voltage to `3000` for 3000mV sensor supply voltage).
- `n` (turn on the sensor supply regulators).
- `j` (submenu for initiating a fixed number of repeated reads from a sensor):

```
Enter selection> j

    Auto-increment from base address 0x01? ['0' | '1']> 0
    Chunk reads per address (e.g., '1')> 1
    Chatty? ['0' | '1']> 1
    Inter-operation spin delay in milliseconds (e.g., '0000')> 0000
    Repetitions per address (e.g., '0000')> 0000
    Maximum voltage for adaptive supply (e.g., '0000')> 2500
    Reference byte for comparisons (e.g., '3e')> 00
```

Example 2: Stream data from all sensors

This will perpetually stream data from the 90+ sensor dimensions at a rate of about 90-tuples per second. Use the following command sequence: - `b` (set the I2C baud rate to `0300` for 300 kb/s). - `r` (enable 48MHz "run" mode for the processor). - `g` (set sensor supply voltage to `3000` for 3000mV sensor supply voltage). - `n` (turn on the sensor supply regulators). - `z` (start to stream data from all sensors that can run at the chosen voltage and baud rate).

5. To update your fork

From your local clone:

```
git remote add upstream https://github.com/physical-computation/Warp-firmware.git
git fetch upstream
git pull upstream master
```

If you use Warp in your research, please cite it as:

Phillip Stanley-Marbell and Martin Rinard. "A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation". ArXiv e-prints (2018). arXiv:1804.09241.

BibTeX:

```
@ARTICLE{1804.09241,
    author = {Stanley-Marbell, Phillip and Rinard, Martin},
    title = {A Hardware Platform for Efficient Multi-Modal
    Sensing with Adaptive Approximation},
    journal = {ArXiv e-prints},
    archivePrefix = {arXiv},
    eprint = {1804.09241},
    year = 2018,
}
```

Phillip Stanley-Marbell and Martin Rinard. "Warp: A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation". IEEE Micro, Volume 40 , Issue 1 , Jan.-Feb. 2020.

BibTeX:

```
@ARTICLE{8959350,
    author = {P. {Stanley-Marbell} and M. {Rinard}},
    title = {Warp: A Hardware Platform for Efficient Multi-Modal
    Sensing with Adaptive Approximation},
    journal = {IEEE Micro},
    year = {2020},
    volume = {40},
    number = {1},
    pages = {57-66},
    ISSN = {1937-4143},
    month = {Jan},
}
```

Acknowledgements

This research is supported by an Alan Turing Institute award TU/B/000096 under EPSRC grant EP/N510129/1, by Royal Society grant RG170136, and by EPSRC grants EP/P001246/1 and EP/R022534/1.

Notes

1 On some Unix platforms, the `JLinkRTTClient` has a double echo of characters you type in. You can prevent this by configuring your terminal program to not echo the characters you type. To achieve this on `bash`, use `stty -echo` from the terminal. Alternatively, rather than using the `JLinkRTTClient`, you can use a `telnet` program: `telnet localhost 19021`. This avoids the JLink RTT Client's "double echo" behavior but you will then need a carriage return ( ) for your input to be sent to the board. Also see Python SEGGER RTT library from Square, Inc. (thanks to Thomas Garry for the pointer).