Embedded Software and Systems (ESS) Assignment

22 - 26 January 2024

Total Number of Pages: 17

a) Derive a set of requirements and specifications for your product

My partner and I in a few months time will get a puppey Japanese Spitz which we are naming after the Japanese baseball player Shohei Ohtani. Inspired by this, I am going to create a digital dog collar that will not only display the name of the dog and contact information in case he gets lost just like a typical dog collar, but also have the additional tracking functionality of a fitness tracker like a Fitbit[1].

Functional Requirements

- R1: Display dog name and owner contact information if the user pushes a button.
- R2: Measure location on a dog walk etc. and distance travelled if the user puts the collar into walk mode.
- R3: Display on phone app past and live dog walk maps
- R4: Measure location with respect to geo-fences for safe areas for dogs to roam where (in Event-Condition-Action style) if the dog leaves the safe zone and in geo-fence mode produce a push notification to the user's phone.
- R5: Measure dog movement to estimate daily dog calorie expenditure/activity level
- R6: Display on phone app summary calorie/activity information.
- R7: Store up to 10 days data locally.
- R8: Upload data wirelessly to the phone.
- R9: Notify user when battery is less than 10%.

Extra-functional Requirements

Constraints

- R10: Battery powered lasting one week for normal use.
- R11: Low-weight and small size less than 50g and 5cm by 2.5cm to be suitable for both small and large dogs to wear[2].
- R12: Low cost of production at less than £25 to enable retail price of £50 although we want this to be a 'luxury' dog product

Robustness & Dependability

- R13: Rugged-integration to IP67 standards for use by dogs that live active lifestyles with exposure to water, mud, dirt etc.

b) Design and discuss your architectural and component choices

(i) Which microcontroller is most suitable and why?

The first choice we have to make is which microcontroller family we want to work with. Because of the tight coupling between hardware and software inherent in embedded systems[3] this is likely not a decision that we will be able to easily change our mind on down the road, unlike some other decisions later on in the VEE process[3].

There are four commonly used microcontroller families we can consider[4, 5]:

- Arduino: This seems to be the most accessible microcontroller family suitable for hobbyists but as we are a startup that wants to go into production this is not the best option for us[4] especially as location and distance travelled (R2) and geo-fencing with GPS-type support (R4) will put significant demands on the controller.

- Espressif Systems ESP32 series: ESP is notable for its built in WiFi and bluetooth capabilities. We do need a form of wireless communication in order to fulfill the location and distance travelled (R2) especially as we want to be able to see the dog walk live (R3) and the upload data wirelessly to the phone (R8) requirements. This makes WiFi not suitable as people tend to walk their dogs outside where WiFi is not available!! As we will discuss in the wireless section, WiFi consumes much more power than Bluetooth (especially Bluetooth Low Energy BLE)[5] so built-in WiFi would not be used.

- STMicroelectronics STM32 series: STM also offers BLE offerings[6] although there seem to concerns about them being buggy[7]. Perhaps the most significant difference seems to be the suite of development tools where STM32CubeIDE allows easy testing on the hardware itself[8], especially beneficial as we used this IDE and the STM32F4 Discovery board in the course[3] so I have more experience with the STM ecosystem and would be able to prototype something much faster.

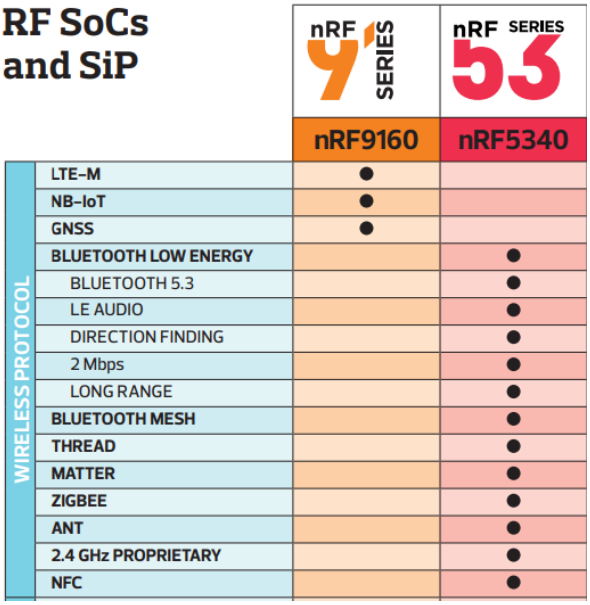
- Nordic nRF series: Nordic seems to be the market leader when it comes to BLE products and lower-power wireless devices[7] although some users cite the significant learning curve associated with Zephyr their open-source RTOS and its devicetree data structure[7] which it uses to describe the hardware available on board[9]. nRFConnect SDK offers a simpler abstraction on top of Zephyr however which might reduce the learning curve[10]. Perhaps most signficantly both Zephyr and nRFConnect support POSIX (Portable Operating System Interface)[11] which means we will be able to test our application code without the hardware (unlike STM which is optimised more for testing on the board itself). Although this is not explicitly listed as a requirement, as a startup with engineers (me!) new to the embedded space the additional time testing in a more familiar Linux environment is very signficant.

In conclusion, the combination of POSIX testing and the fact that Nordic is the leader in BLE products means I shall choose them. The next step is to choose a specific Nordic microcontroller.

(ii) Which wireless interface have you chosen and why?

The next step is to choose which Nordic product we want specifically. There are two main series we can consider the nRF9 series and the nRF53 series[12] (there is also the older nRF52 series). The primary difference is that the nRF9 series nRF9160 offers integrated support for cellular IoT and GPS whilst the nRF53 series nRF5340 offers built in BLE[12].

Which product we choose will be primarily driven by which wireless interface we decide to choose.



To make this decision we need to consider our requirements where the most significant is that we want to measure the location of the dog on a walk (R2), be able to display this information live (R3) and have push notifications with a geo-fence (R4). The most obvious solution for these requirements is to use Global Navigation Satellite System (GNSS) which in the US is Global Positioning System (GPS) but also has other international counter-parts such as the European Galileo[13]. We could also use blue-tooth and in fact that is how Apple AirTags work[14] where they exploit BLE to communicate with nearby Apple devices using the signal strength to triangulate the location, and in fact the range on these bluetooth devices can be quite significant up to 1km[15]. However, nonetheless GPS is clearly still the preferred option especially as our requirements include fairly precise dog walk visualisations that would not be possible with the bluetooth positioning.

Nonetheless this does not automatically mean we should go with the nRF9 series because it is possible to integrate a GPS module with the nRF53. The second choice is how we want to upload our data wirelessly to the phone (R8). We have a few options

- As we can see in the Nordic product guide[12] we could use LTE-M (Long Term Evolution for Machines), the big disadvantage here is that it typically requires a SIM card subscription to a cellular provider which would either be an additional cost to the consumer or to us as a startup. For example, PitPat's Dog GPS tracker is significantly more expensive at £159 because they include a factory-installed SIM and use LTE-M to provide coverage in the UK. Tractive on the other hand is significantly cheaper at £44.99 up front but requires a subscription of £6/month when billed annually
- The other option is to use bluetooth which would not require these same costs and has reduced power consumption requirements. The disadvantage would be the fact that the range of bluetooth is much less and that we would require an additional bluetooth module to integrate with. One advantage of bluetooth is that it can offer higher data rates with bluetooth classic even support data-intensive applications like audio streaming but given that we are just sending some location and accelerometer data this is not necessary.
- There are a few other options including device mesh network with gateway[16] such as Zigbee or Thread[12] but given that we want to be able to have live data updates when going on dog walks (R2) which are often in remote areas without other devices I do not think this works well. There is also NB-IoT (Narrow-band) but this is not appropriate for our use-case as they are better suited to static, ultra-low bandwidth sensors not ones that need to move around with the dog!

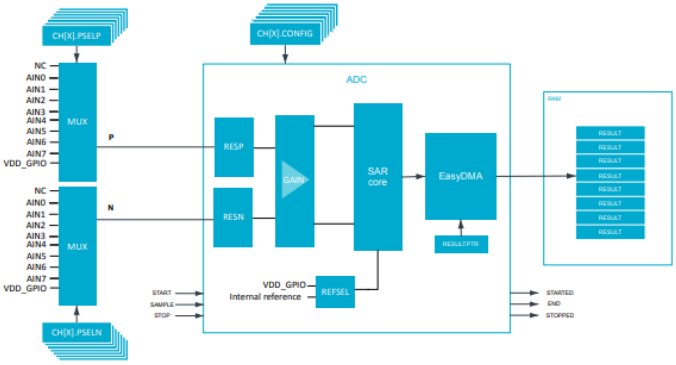
Ultimately, even though blue tooth can have long-range of several hundred metres a distance which most dog owners will not let their dog get further away from them then, in order to implement the geo-fences requirement (R4) we stshall opt for using LTE-M in line with the other GPS dog trackers in the market. This will likely have an affect on the cost of production (R12) where we may need to have an additional subscription in order to make the economics work in addition to the one-off retail price target of £50.

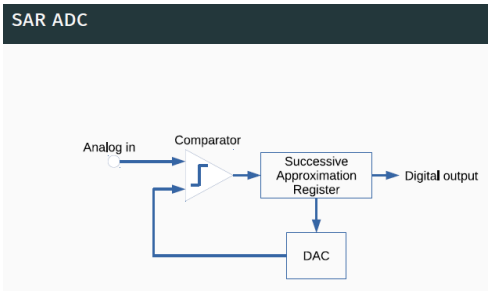
(iii) How are you going to obtain your sensor readings?

In order to fulfill the requirements we need to get two types of sensor readings.

The first is the GPS signal required for the location measurement (R2) and geo-fencing (R4). Because we have chosen an integrated GPS received[17] we can send and receive signals via the Univeral Asynchronous Received Transmitter (UART) interface which offers full-duplex communication which means that it allows data to be transmitted and received simultaneously[16]. As we can see on the datasheet our microcontroller support 4x UART with EasyDMA for direct memory access[17] with One pulse per second (1PPS) which is more than sufficient accuracy for our dog walk location mapping[18].

The other is the accelerometer for dog activity estimation and calorie consumption estimation (R5). We have a choice between whether we use a digital or an analog accelerometer where the advantage of the former is that they often offer higher resolution and precision, and more robustness to noise. The disadvantage is that they have higher cost and power consumption[19]. The alternative is an analog accelerometer which has a simpler interface and lower power consumption, although it can be more sensitive to noise sensitivity[19]. As we are trying to build a wearable device with cost (R12) and power (R10) requirements we shall choose the analog accelerometer (also just as importantly we are required to do so for the assignment!!). In order to obtain the readings we will need an analog to digital converter fortunately the nRF9160 microcontroller offers this with a Successive approximate analog-to-digital converter (SAADC)[18] (see below) which uses a successive approximation register (SAR) to find the closest approximation of the input voltage where iteration is used to refine the resolution where the main advantage of this approach is that it is power-efficient and relatively simple compared to more complicated ADCs like sigma-delta but with higher resolution than a flash converter[20, 3]. Given that we do not need high resolution data like we would for headphone speakers etc. we just need approximate accelerometer data this should be more than sufficient where we have 8/10/12-bit resolution with 14-bit resolution with oversampling[18]. We will want a three-axis accelerometer however to capture all the possible movement of the dog.





If we were using a digital accelerometer we might communicate over I2C or SPI (more on these later) but the nRF9160 offers direct sample transfer to RAM using EasyDMA[18] (see diagram above). However, we still get the flexibility to choose our sampling frequency which is either via a task from software or a Programmable Peripheral Interconnect (PPI) which allows interaction with peripherals without the need for CPU intervention[18]

(iv) What sampling frequency and resolution will you require?

Unlike human activity recognition (HAR)[21] there is not a lot of data available for dog (or even pet) related activity tracking. Whistle, a pet tracking device, has done a major study with over 100,000 dogs to help power the AI behind their own activity tracker[22], but for commercial reasons have not made the data widely available.

According to the Nyquist-Shannon theorem in order to accurately capture a signal you should at sample at least twice the highest frequency present in the signal[23]. Human activity trackers typically vary their signal as low a 1-10Hz when sleeping to 10-30Hz for step counting and as high as 100Hz for exercise tracking[21]. As we want to classify activity all day long it makes sense to also be able to vary the sampling rate in order to preserve battery. Given that 30Hz was sufficient for a lion-tracking study[24] 50Hz should be more than sufficient for our pet tracker - and also in line with the 50Hz maximum that is used by Whistle[25]. Rather than using software to do activity detection which involves expensive CPU

usage we can instead use PPI channels which can use to analyse the accelerometer data at a baseline sampling rate of perhaps 1Hz for sleeping and low activity and then if a threshold is breached the PPI can adjust to increased the sampling rate. This would work where each sampling rate has its own corresponding timer. Our microcontroller offers very accurate 1/16 MHz timer[18] but this is orders of magnitude more than we need. Instead we can use the lower-power 32.768 kHz Real-Time Clock (RTC) to calculate our two required sampling rate.

The ADC can be run in one of three modes: one-shot mode, continuous mode and scan mode[18]. As we have only one analog device that we want to continuously get data from we should use continuous mode (scan allow switching between analog devices). We can achieve continuous sampling by triggering the SAMPLE task from one of the general purpose timers through the PPI system[18]. We are advised the key formula we need to abide by is that

$$f_{SAMPLE} < \frac{1}{t_{ACQ} + t_{conv}}$$

i.e. that the maximum sampling rate cannot exceed the inverse of the acquisition time added to the conversion time. This can vary based upon the source resistance but acquisition time is typically order of magnitude $10\mu s$ and conversion time typically less than $2\mu s$ so plugging into the equation we get

$$f_{SAMPLE} < \frac{1}{10\mu s + 2\mu s} = 83kHz$$

This is obviously not a concern as we are considering max sampling rates of just 50Hz.

Nonetheless to achieve our goal sampling rates we need to use a pre-scaler[16, 3]. As we have decided to use the lower-power 32.768 kHZ RTC we need to be able to scale this to 1Hz and 50Hz rates for our two use-cases. Using the pre-scaler formula[3] we need

$$Prescaler for 1Hz = \frac{32.768kHz}{1Hz} = 32,768$$

$$Prescaler for 50Hz = \frac{32.768kHz}{50Hz} = 655.36$$

This are significantly smaller than the max PRESCALER size that is raised as a concern in Making Embedded Systems[16] where sometimes we have to consider constraints on the max value on the prescaler, for the nRF9160 the max prescaler is $2^{12} - 1$ so this is not a problem[18], and we certainly do not need to generate a faster clock from a slower oscillator using something like phase lock loop (PLL)[16] as we are sampling at significantly lower rates than the clock already.

Nonetheless, changing the prescaling to give us different sampling rates does not work because with the nRF9160[18] you can only write to the PRESCALER register when the RTC is stopped, which is not ideal as we may need the RTC for other components. Instead we can calculate the required PRESCALER for a shared counter frequency[18] of 50Hz

$$PRESCALER = round(\frac{32.768kHz}{50Hz}) - 1 = round(655.36) - 1 = 654$$

We can then use this shared PRESCALER of 654 to generate an interrupt frequency for both sampling rates where we vary by changing the compare value.

$$interruptFrequency = \frac{clockIn}{prescaler \times compare}$$

If we set the compare value to 1, then we achieve $\frac{32.768kHz}{654 \times 1}$ the 50Hz frequency and if we set it to 50 we achieve $\frac{32.768kHz}{654 \times 50}$ the 1 Hz.

(v) Datasheet requirements and other components

Features:

Microcontroller:

- ARM® Cortex® -M33
  - 243 EEMBC CoreMark score running from flash memory
  - Data watchpoint and trace (DWT), embedded trace macrocell (ETM), and instrumentation trace macrocell (ITM)
  - Serial wire debug (SWD)
  - Trace port
- 1 MB flash
- 256 kB low leakage RAM
- ARM® Trustzone®
- ARM® Cryptocell 310
- Up to 4x SPI master/slave with EasyDMA
- Up to 4x I2C compatible two-wire master/slave with EasyDMA
- Up to 4x UART (CTS/RTS) with EasyDMA
- I2S with EasyDMA
- Digital microphone interface (PDM) with EasyDMA
- 4x pulse width modulator (PWM) unit with EasyDMA
- 12-bit, 200 ksps ADC with EasyDMA - eigth configurable channels with programmable gain
- 3x 32-bit timer with counter mode
- 2x real-time counter (RTC)
- Programmable peripheral interconnect (PPI)
- 32 general purpose I/O pins
- Single supply voltage: 3.0 – 5.5 V
- All necessary clock sources integrated
- Package: 10 × 16 x 1.04 mm LGA

LTE modem:

- Transceiver and baseband
- 3GPP LTE release 13 Cat-M1 and Cat-NB1 compliant
  - 3GPP release 13 coverage enhancement
- 3GPP LTE release 14 Cat-NB2 compliant
- GPS receiver
  - GPS L1 C/A supported
  - QZSS L1 C/A supported
- RF transceiver for global coverage
  - Up to 23 dBm output power
  - -108 dBm sensitivity (LTE-M) for low band, -107 dBm for mid band
  - Single 50 Ω antenna interface
- LTE band support in hardware:
  - Cat-M1: B1, B2, B3, B4, B5, B8, B12, B13, B14, B18, B19, B20, B25, B26, B28, B66
  - Cat-NB1/NB2: B1, B2, B3, B4, B5, B8, B12, B13, B17, B19, B20, B25, B26, B28, B66
- Supports SIM and eSIM with an ETSI TS 102 221 compatible UICC interface
- Power saving features: DRX, eDRX, PSM
- IP v4/v6 stack
- Secure socket (TLS/DTLS) API

Current consumption @ 3.7 V:

- Power saving mode (PSM) floor current: 2.7 µA
- eDRX @ 82.91s: 18 µA in Cat-M1, 37 µA in Cat-NB1 (UICC included)

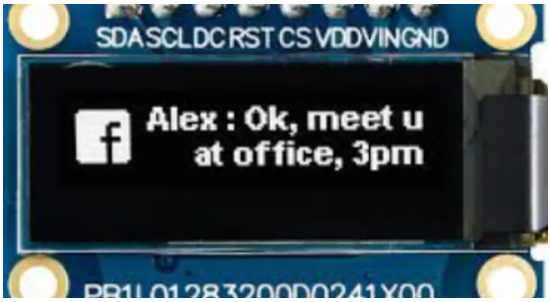
Before finalizing our decision to go with the nRF9160 we need to check the datasheet to make sure that it can suit all our other requirements and we need to choose the other components we need to make sure they can work with the nRF9160

R1: Dog name display. Going through the options[26]:

- ElectroPhoretic Display (EPD) like in a Kindle would consume less power but it has a paper like display appearance
- Liquid Crystal Displays (LCD) like in TVs offer a very sharp display but require backlighting so is not suitable for low energy/non-wired use case like ours. Also we only wish to display limited textual information so don't need a great screen, furthermore being visible in direct sunlight is more relevant if outdoors.
- Organic Light Emitting Diode (OLED) like in the iPhone, has the advantage that each pixel is individually lit saving energy making them well suited to low-energy cases like battery powered devices. This also makes them a good fit for always-on type use-cases, for example, although not an explicit requirement, one could imagine that it would be nice to have the name of the dog on constant display when on a walk. One downside is that they can be susceptible to water damage - something that will be an obvious problem when on walks outside.
- Reflective Segmented LCDs like in boiler displays have the advantage of very low energy consumption as they don't need to be backlit if there is light in the room that can be reflected.
- LEDs - we could opt for a simple LED but this would not fulfill the requirement of displaying dog name and contact information.

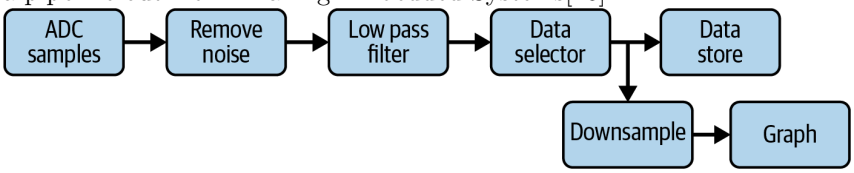
On balance we shall use a small OLED display where we will choose Passive Matrix OLED (PMOLED) over Active Matrix OLED (AMOLED) as we do not need the precise control and faster refresh rates that a TV etc. might need. However, to choose the specific component we need to consider both the cost and the connection type we want.

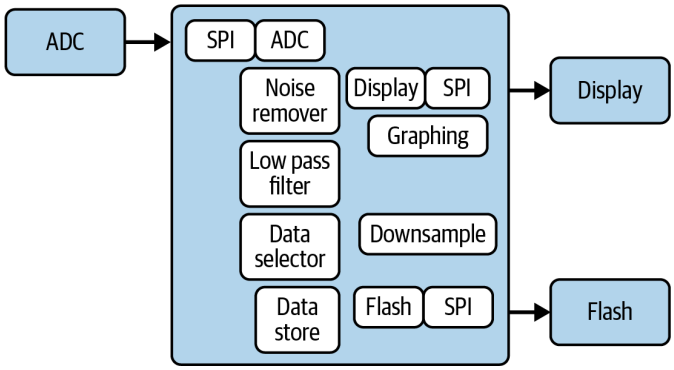
There are two main ways we can connect our PMOLED screen: 1. Serial Peripheral Interface (SPI) 2. Inter-Integrated Circuit (I2C or I^2C). I2C has simpler wiring with just a serial data (SDA) and serial clock (SCL) compared to four wires for SPI with chip select (CS), master-out slave-in (MOSI), master-in slave-out (MISO) and a clock (CLK)[3]. I2C is only half duplex unlike SPI which is full duplex allowing simultaneous communication in both directions. The main difference for our purposes is that bus speeds on I2C are limited to 400kHz[3] whereas SPI can go faster and SPI requires more GPIO pins. Our nRF9160 has 32 GPIO pins so this is not an issue, and because we are using the screen to show very low-refresh information like the dog's name or the owner's contact information I thought I would go for the cheaper I2C. However, if I compare products for example the the Midas 0.91in white passive matrix OLED display 128x32 pixels which is £5.93 (excluding VAT) for the I2C version in the left picture below but £6.88 (excluding VAT) for the SPI version in the right picture you can see a significant difference in the quality of the image which given that we are aiming for a luxury product (R12) we shall go with the SPI version (right picture). The data bandwidth for SPI should be more than sufficient here because we will update the screen quite infrequently[16]



We will also need two buttons: 1. in order to display the different contact information and 2. start/stop a dog walk. In either case buttons can suffer from 'bouncy signal'[16] so we will need to debounce the signal looking for the rising edge of the signal where the user releases the button. They will be connected to the GPIO boards which the nRF9160 has 32[18]

R3: Phone app display. As discussed we will use LTE to send data live to the user's phone (R8) and a phone app will display this graphically. We have a choice about how much data processing we do on the dog collar vs on the phone. To minimize battery we will want to do most of it on the phone, however we also want to minimize how much data is sent over the network so one could imagine following the data pipeline outline in Making Embedded Systems[16].





To make this work we may need some buffering as each filter takes some processing time and we need to be careful about the possible latency effects of this on our windowed data[16]. With the GPS data we will want to send it in a stream, or with low latency but with the activity data we can probably down sample significantly and batch them into 10 minute updates as the activity scores will be summaries over multi-minute/hour level granularity. Furthermore, flash, unlike EEPROMS which allow bytes to be

written individually, is made up of sectors that must be erased all at one time and then written one at a time[16] so we need to buffer some of the data in RAM (256 KB according to the datasheet [18]) before writing.

R7: 10 days data storage: For this reason, and because one of our requirements is to be able to store up to 10 days data locally (R7) we will need a pipeline that stores to flash. We can estimate the space requirements of the GPS data where a minimal representation might be two floats of up to 7 decimal places precision which is 4 bytes x 2, and then an additional 4 bytes for a timestamp making 12 bytes in total.

$$Latitude(4bytes) + Longitude(4bytes) + Timestamp(4bytes) = 12bytes$$

If the user goes on a walk for 2 hours a day with their dog (which is likely towards the upper limit) that is $2hours \times 60mins/hour \times 60seconds/minute = 7,200seconds$, and if we assume one GPS signal per second (as discussed we have a 1PPS signal [18]) that is $7,200 \times 12bytes = 86.4KB$ which given our 1 MB space gives us 11.57 days worth of storage! This is sufficient for our storage requirements of 10 days data locally (R7) but only just! And we would also have to consider the activity data, and perhaps need to down-sample and process this significantly on the dog collar itself. As discussed above, flash must be written to in batches where the datasheet[18] describes that there are 32 regions of 32KB each which would be $32KB/12bytes = 2666s$ or 45 minutes of data at at time! There is thus a risk that we could lose some data but given that this is dog walk data rather than something more significant such as health data etc. I think this is an acceptable risk. Of course, we need to have a policy for if we run out of flash memory where the best option is probably to write over the oldest data (circular buffer style) the idea being that the owner will want to retain the latest dog walks the most.

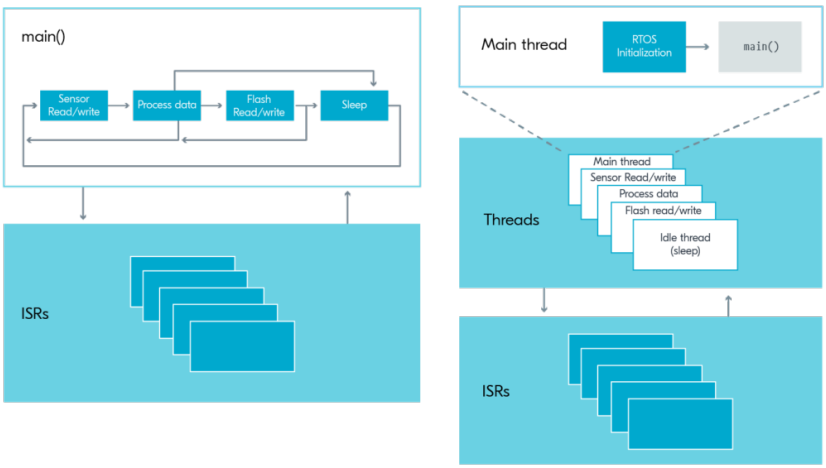
R10: Perhaps most importantly we will need a power source which we are hoping will last one week (R10). For this there are two main types of battery that are commonly used: Lithium Ion and Lithium Polymer. Both have high energy density and low-discharge rates which means they do not leak energy when y are not being used. Lithium Polymer can be molded into more flexible form factors but they come at a higher cost[27] so that for reason a standard lithium ion battery button cell battery as used in digital watches[28] like the CR2016 which can provide the 3.5V we need for the microcontroller[18] and does not violate our size and weight constraint (R11) at 20mm in diameter (just inside our 5cm by 2.5cm shape) and at 1.9g[28]

c) Design your firmware

(i) Will you use an RTOS? Why/Why not?

Real-time operating systems (RTOS) provide a layer of abstraction to 1. ease programming in complex systems 2. provide the illusion of concurrency 3. provide modularity and isolation[3]. This is in contrast to running a bare-metal for loop which is typically associated with greater power efficiency, reduce memory usage and potentially faster performance although as applications get more complicated this is not necessarily the case[29].

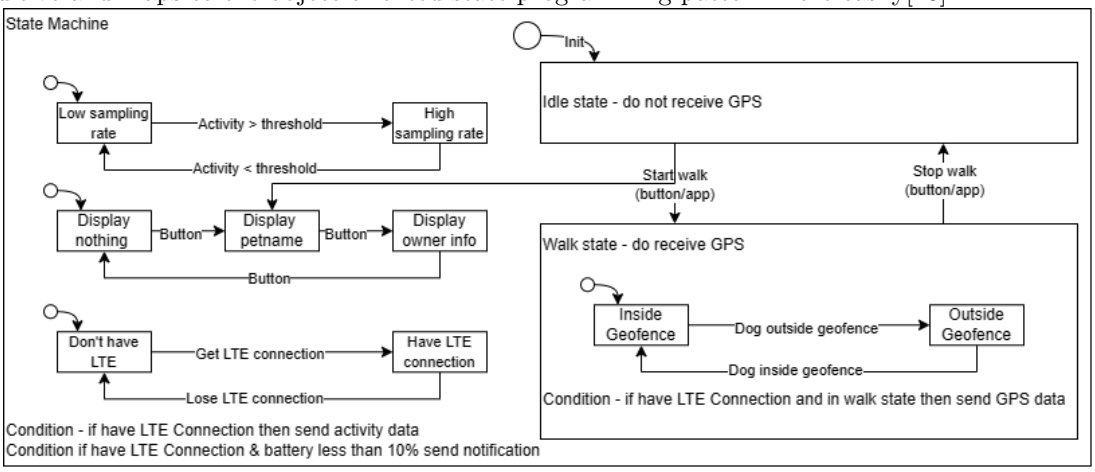
The core of an RTOS is the kernel which controls everything and most notably is schedules the order of tasks that are run on the CPU[3]. Both bare metal and RTOS offer interrupt service routines but RTOS offer a thread abstraction to make it easier to work with these (see bare metal vs RTOS visualization below[29]



For our use-case we need to process data from multiple data sources, display this data, store it in flash and send information to the user over LTE which is a lot of complexity to manage with just a bare-metal implementation. Our microcontroller nRF9160 is built to work with Nordic's Zephyr RTOS[29] and even offers abstractions on top of that including the nRFConnect SDK. As cited in Making Embedded Systems, one of the major advantages of RTOS is they give you access to a large number of libraries and packages including flash file systems which cand handle things like being power-loss resilient, handle wear leveling using all the blocks equally and identifying bad blocks and working around them[16], although there are bare-metal file systems like littlefs[16] this is much easier with an RTOS. Other advantages including using modules to help with security, not insignificant when we have real-time location data of not just the dog but presumably also the accompanying dog-walker[16]! It is worth noting that an RTOS can introduce additional complexities however, for example if two tasks are running an neither is an interrupt you can use a mutex to indicate who owns the resource, but this now means that ownership handover should be atomic i.e. uninterruptible[16]

(ii) In particular, show a detailed state machine for your prototype

In visualising our state machine we have a choice between state-centric state machines (potentially hiding the transitions) or we can choose an event-centric state machine - we shall choose the latter as it is more intuitive and maps to the object-oriented state programming pattern more easily[16].



Our state machine is a hierarchical state machine that allows us to visualize and model nested and independent states without a state explosion[3]. In our state machine there are two key states 'Idle state' and 'Walk state'. Our initial transition is to the former but we can transition between states based upon either a button or app click to start or stop the walk. Crucially in the walk state we initialize with the inside geofence state and then we continually test whether the dog is outside the geofence. You could imagine that we might add additional functionality where in addition to trigger a push notification to the user's phone that the dog is outside the geofence (R4) we might also have other custom behaviour

like saving battery on other features in case the dog gets lost or perhaps displaying a custom message on the collar to anyone who finds the dog.

We also have a few other states such as the different sampling rates and conditions such as low battery push notifications. We could potentially enhance the state machine with timeouts etc. where if the consumer does not charge their collar maybe they get sent another message or perhaps if in walk state it switches to idle state to save battery. We already have something similar to this functionality where the default display on the collar is nothing - no point using the battery to show the pet's name at home, but we default to displaying the petname once we go into walk mode. We can see the use of parallel state machines where you need to have both an LTE connection and a walk state to send the GPS data - and only if you are in the outside geofence state will a push notification be sent!

In order to make sure we have not missed any states we can use a state transition table[3, 16]. Because we have a hierarchical state machine this is slightly more complicated so we will focus on the most interesting behaviour which is around LTE and walk state which gives four possible combined states. Note, in Making Embedded Systems there is the example of considering invalid states because if there is a toy with 30 buttons every possible combination will be pressed[16], in our case these do not need to be considered as LTE connection and geofence detection is not dependent upon the user. We do though have to consider the invalid action where a user might start a walk on the dog collar and then start another walk on their phone where we will do nothing.

LTE state	Walk state	Event	Next LTE state	Next Walk state
Init	Init	Nil	No LTE	Idle state
No LTE	Idle state	Get connection	Have LTE	Idle state
No LTE	Idle state	Start walk	No LTE	Walk state.inside geofence
No LTE	Idle state	Stop walk	No LTE	Idle state
Have LTE	Idle state	Lose connection	No LTE	Idle state
Have LTE	Idle state	Start walk	Have LTE	Walk state.inside geofence
Have LTE	Idle state	Stop walk	Have LTE	Idle state
No LTE	Walk state.inside geofence	Get connection	Have LTE	Walk state.inside geofence
No LTE	Walk state.inside geofence	Start walk	No LTE	Walk state.inside geofence
No LTE	Walk state.inside geofence	Stop walk	No LTE	Idle state
No LTE	Walk state.inside geofence	Dog outside geofence	No LTE	Walk state.outside geofence
No LTE	Walk state.outside geofence	Get connection	Have LTE	Walk state.outside geofence
No LTE	Walk state.outside geofence	Start walk	No LTE	Walk state.outside geofence
No LTE	Walk state.outside geofence	Stop walk	No LTE	Idle state
No LTE	Walk state.outside geofence	Dog inside geofence	No LTE	Walk state.inside geofence
Have LTE	Walk state.inside geofence	Lose connection	No LTE	Walk state.inside geofence
Have LTE	Walk state.inside geofence	Start walk	Have LTE	Walk state.inside geofence
Have LTE	Walk state.inside geofence	Stop walk	Have LTE	Idle state
Have LTE	Walk state.inside geofence	Dog outside geofence	Have LTE	Walk state.outside geofence
Have LTE	Walk state.outside geofence	Lose connection	No LTE	Walk state.outside geofence
Have LTE	Walk state.outside geofence	Start walk	Have LTE	Walk state.outside geofence
Have LTE	Walk state.outside geofence	Stop walk	Have LTE	Idle state
Have LTE	Walk state.outside geofence	Dog inside geofence	Have LTE	Walk state.inside geofence

d) Discuss, with the aid of examples, how you would go about implementing and unit testing your code.

(i) Code implementation)

Now we have the finite state machine we can directly map the code that we might write to it where each entry is represented by a row in our table above without the need for overly complex and perhaps unhandled state/event combinations that you would get a

```
enum class tLteState {
    InitLte,
    NoLte,
    HaveLte,
}

enum class tWalkState {
    InitWalk,
    IdleState,
```

```
    WalkStateInsideGeofence,
    WalkStateOutsideGeofence,
}

enum class tEvent {
    GetConnection,
    LoseConnection,
    StartWalk,
    StopWalk,
    DogInsideGeofence,
    DogOutsideGeofence,
}

struct StateTransitionTableEntry {
    tLteState lteState;
    tWalkState walkState;
    tEvent event;
    tLteState nextLteState;
    tWalkState nextWalkState;
}
```

We can then have an event handler[16] which looks up in the stable table map what the next states are based upon the event. To do this we can create a composite key of the two states and an event. Here the using keyword allows us to define a type alias. Then inside the HandleEvent function we create the key for the particular event and use it to look up the iterator which the compiler with auto can deduce is of type std::map<StateKey, StateTransitionTableEntry>::iterator

```
using StateKey = std::tuple<tLTEState, tWalkState, tEvent>;
std::map<StateKey, StateTransitionTableEntry> stateTransitionTable;

void HandleEvent(struct stateTableEntry *currentState, tEvent currentEvent) {
    StateKey key = std::make_tuple(currenState->lteState,
        currentState->walkState, currentState->event);
    auto it = stateTransitionTable.find(key);
    if(it != stateTransitionTable.end()) {
        currentState->lteState = it->second.nextLteState;
        currentState->walkState = it->second.nextWalkState;
    }
}
```

Eventually, this tabular approach might get unwieldy[16], so we might move to a more traditional finite state machine design pattern with if-else statements for each state. Especially if we want to add conditionals like if the battery is less than 10% (R10) then send a notification to the user etc.

(ii) Testing

There are three types of tests most commonly seen in embedded systems[16]

- Power-on self-test (POST) = This runs every-time we boot the system even after the code is released. For our use-case the most obvious things to test are the main hardware components like the GPS receiver and the analog accelerometer.
- Unit tests: As we are using an RTOS with modules we can separate the code into different parts and test each one individually. We probably do not want to run these tests when the product is being used/or on boot, but they can be vital in the development process to catch bugs and isolate malfunctioning software. Unit tests map to the design step in the VEE diagram[3]
- Bring up tests = These are tests that are performed on new or modified components[16] most often hardware. For example, in our use-case we have smalled OLED screens so it would make sense to test the screen for powering on and displaying something before integrating with our system.

Another example,. might be flash test to see if we can read existing data, get byte access and block access[16]. In a production process this should be done on every component to help detect manufacturing defects and to ensure product quality, a digital multimeter can be helpful here[16].

Other types of testing we can do include

- Functional and extra-functional tests = Of course, it makes sense to actually test our system against our functional and non-functional requirements. As our device is meant to be used outdoors by dogs this should include adherance to IP67 standards with related exposure to water and dirt, and of course vigorous play (R13).
- Integration, system and regression tests = Whenever, a component or new piece of software is added we need to test whether the components can work together using integration tests For example, is the accelerometer data successfully processed by the ADC and sent to the user via LTE or if there is no LTE connection is it successfully stored in flash. Similarly, regression tests check whether existing behavour is not affected by some new change. Integration and system tests map in our VEE diagram[3] to the specification and architecture steps.
- Verification and validation: We could potentially try to verify correctness either by construction, or by checking including both physical and virtual prototypes which in turn include formal verification methods and simulations[3], although these are not widely used in industry as they do not scale well to large systems. Static analysis particularly in a compiled typed language like C can be particularly helpful.
- Debugging: We can also use a debugger such as JTAG to step through our code and see what is happening[3]
- Probabilistic and chaos-monkey testing: Rather than writing individual tests we can test probabilistically or explicitly look for errors with something like chaos monkey[3]
- Performance and profiling tests = On embedded systems we are highly constrained by RAM, Flash, CPU time etc. so testing our systems e.g. how many hours of GPS data can our system store before running out of space would be very valuable especially as we have requirements around data storage (R7) and battery life (R10)
- Security testing = We are dealing with semi-sensitive location data and so we should test basic security vulnerabilities.
- Beta testing = One final type of testing we might consider is actually using the product ourselves in the world with a small set of users, perhaps just the startup employees to see if the product is working as expected.

(iii) In particular, how would you go about testing the analog sensor without having access to it (e.g. whilst it is being fabricated)

There are a number of ways that we can test our components before having access to them.

- Dev kits: Even though we might not have access to the component itself it is possible we can get a dev kit or an evaluation board for the related component to test on some hardware early in the software development lifecycle[5]. At the very least we can use the dev kit to set up the compiler and debugger before the custom hardware comes in[16]
- POSIX support: One of the advantage of Nordic was that Zephyr and nRFConnect support POSIX[11] which means we can test our application code without the hardware by compiling it on Linux. Variations include hardware-in-the-loop simulation[3] where the hardware thinks it is controlling a real plant or system but instead a PC is just running a simulation.

e) Carefully consider possible failure modes of your system and what risk mitigation you could take

Our dog tracker is a soft real-time system where not meeting a constraint can lead to a degradation of service but it is not a hard real-time service where something like death/injury/explosion could happen[3]. We can divide possible failure modes into a few categories

- Total failure: The system failures to fulfill any of the functional requirements (i.e. a bricked device[3]) for example, perhaps the battery cannot hold a charge and so does not turn on. This is most likely due to some sort of hardware failure and we can try to mitigate this with rigorous bring up and POST tests. Of course, one advantage of total failure is that it should be obvious to the consumer and therefore they can contact our support team to help try and solve the problem. Although not a mitigation, having good error logs that we can access can help significantly with the debug process[16], although as our primary communicated method is via LTE this does not work if that is down, perhaps we might need a backup USB port as a way to download data manually. If there is some power but it is very low than a Brownout reset in which the system is put in a reset state if the supply voltage drops below the brownout threshold can help[18]

- Intermittent failure: This is probably the most likely type of failure, for example, there could be a software bug which only comes up in some edge case perhaps when the collar is in some unusual state like resetting to dog inside the geo-fence with an intermittent connection to GPS so it does not reliably send the push notification to the user. Or perhaps there is a rare buffer overflow in a variable in RAM which most of the time does not matter but sometimes affects the timing of components. This is particularly insipid if it is not part of normal operation, for example if the push notifications for outside the geofence are not working but the dog stays within the geo-fence for 3 months and so this is not detected. The solution to this is lots of testing to at least make sure the most common patterns of usage do not have these issues.

- Constant partial failure: This could be something subtle like a timer (or even a rounding) error compounding over time, for example the accuracy of the RTC depends on the accuracy of the clock it uses for input, but this will run slower when the temperatures drop[16]. This does not directly prevent the key requirements being fulfilled and the consumer can still use the product but in a slightly degraded fashion. One choice we would have to make in this is between graceful degradation or failing loudly and immediately[16]. Something like a timer being subtly out of sync potentially could be reset, but other errors like battery life slowly degrading, or the OLED screen having burn-in (even after mitigations like changing pixel locations) might be something that we just have to accept as part of the user experience over time.

- Connection failure: This type of failure involves being unable to send data over LTE or perhaps being unable to receive GPS transmissions. To some degree, these types of failures are part of normal operating experience and in fact we have added the ability to store data locally (R7) as one of our key requirements partly in case of such a connection failure.

- Wear and tear: Given that our product is going to be worn by dogs outside the chance of water damage, dirt, chewing and tearing is quite high, especially as we have a potentially fragile OLED screen. Using high quality products and properly testing their water-proof, dirt-proof and other IP67 standards will be very important. Another factor worth considering is the fact that this is a wearable and that it needs to be comfortable for the dogs etc even if the collar is wet or dirty etc.

In addition to potential mitigations, being able to recover from failures involves having

- Watchdog: In the case of catastrophic error a watchdog is a timer capability that will reset the processor if the processor fails to perform an action, usually something simple like toggling an I/O line[16], known as 'kicking the dog' or perhaps more appropriately given our pet-related product 'petting the dog'[16]. This acts as a form of heartbeat which if it fails can trigger a fail-safe system to run. In order for this to work we will need to leave space to store our golden image[3] which of course does not work if it also has bugs so perhaps reduced functionality such as just enough to send errors logs over LTE/USB etc. is sufficient.

- Error logs: We should use an error library[16] to store the errors in RAM although as discusssed getting access to these logs, particular once the product is out in the wild might be quite difficult.
- Over-the-air updates: As we have LTE in our system the ability to transfer new code in case of a bug is of course very tempting, although there are obvious limitations if the LTE is not working and the fact that it might be difficult to test much in advance[16]. If we want to do a firmware update this is more complicated as we need to have a separate bootloader and application code 1. putting the code in auxiliary flash 2. where if the flash's code version is the same as the processor's current version call the current code normally but otherwise copy the flash's code across to the processor and reboot[16]. The obvious downside to all this is it requires flash space which we have already established is at a premium.
- Data loss: If the power is lost in the middle of a system reset it might not have had time to mark the flash as ready to erase[16]. Some data like timestamped GPS data could probably be deduplicated on the phone app side to one data value per second but other mights not be able to. One solution is to have a modification list in a different part of flash with the program address block that is about to be erased, and then on boot we can check the list for all any marks that have not been completed[16]. Alternatively we can also use a checksum to see if all the data has been written completely[16].

f) Discuss various approaches that could be used to reduce power consumption for your device. Consider in particular how power optimization techniques impact sensing, computation and communication

As discussed in Making Embedded Systems[16] power is a function of voltage and current.

$$P = I^2 \times R \tag{1}$$

$$P = I \times VW \qquad = P \times T \tag{2}$$

This means we have a few approaches to lowering the amount of energy a system uses. First we can reduce the voltage by using lower voltage parts as lower V means lower P and therefore lower W. We can also reduce the resistance as lower R reduces P as well, or current I. And finally we can reduce the time T by turning the system off as much as possible[16]. Starting with building a power budget and after profiling to understand what the biggest consumers of power are we then can iterate on reducing the consumption[3]

- Turn-off: The most obvious way to reduce power consumption is to turn the whole system off although this has the obvious downside of reducing the capabilities of the system somewhat! Variations of this include just turning off some components, perhaps triggered if the battery level is at less than 10% (R9) and could include turning off the OLED display or turning off the LTE connection.
- Slow down: Lowering the clock frequency saves power although this also reduces the amount of processor cycles you have to run code in (writing optimal code to reduce the cycles is crucial here)[16]. Our nRF9160 has a slow clock to drive the RTC which is used in the acceloremeter data sampling saving energy over the 1/16 MHz timer[18]. We could potentially also run processing at this lower clock rate although often it is more energy efficient to do computation at the maximum clock rate and then sleep instead[3] The nRF9160 has a number of other power saving states including LTE-M has an extended discontinuous reception (eDRX) in which the device can enter a lower-power mode by extending the time intervals that they need to check in with the network for potential incoming signals and data. In the nRF9160's case it has eDRX @ 82.91s: 18 µA in Cat-M1, 37 µA in Cat-NB1 (UICC included) which means that it waits 82.91s to check, and that it uses just $18A$ per cycle for LTE-M Cat-M (or slightly more if used narrow band including the Universal Integrated Circuit Card more commonly known as a SIM card)[17] It also has a power saving mode (PSM) floor current of just $2.7A$[18].

- Sleep: Sleeping involves turning the processor core off but keeping timers, peripherals and RAM alive, allowing any interrupt to quickly return the processor to normal running. Some work is even being done to run program code from RAM rather than flash although the volatile nature of the memory makes this tricky[3] Other variations include turning off some of the peripherals as well. This is what the IdleState would involve when the dog is not on a walk. This can be optimised to avoid frequent wake-ups by batching low priority house-keeping together[16]
- Off-CPU: Our nRF9160 comes with a number of helpful power saving functionlaties. These include being able to use PPI to get the accelerometer data meaning that we don't need to use the CPU.
- Simplify computation: One example is how we reduce the sampling rate of the accelerometer based upon the activity level which not only means less data to receive but also less data to process, transform and store. Another example is we can simplify the mathematics where for example division can be very expensive but multiplication and addition are cheap, so storing a rolling sum or block average rather than a rolling average[16] on the microcontroller and doing any graph processing on the much more powerful mobile phone could save a lot of power. Another example, could be compression with run-length encoding replacing repeated occurrences of a symbol or Huffman coding that builds a dictionary of symbol frequency[3]

References

[1] Dall e Image Generator. Generate an image of a fitbit like black digital dog collar around the neck of a japanese spitz dog - wher the dog collar will display the name of the dog 'ohtani'.

[2] smartbark.co.uk. Best gps trackers for dogs 2024 : Tested reviewed'.

[3] University of Oxford. Embedded software systems lectures slides jan 2024.

[4] Embedic. Esp32 vs arduino: What are differences and how to choose.

[5] Embedic. Esp32 vs stm32: Which is better and how to choose?

[6] ST. Bluetooth low energy.

[7] Reddit. Nordic vs stm for ble (advice required).

[8] ST. Stm32cubeide.

[9] Nordic Semi. Devicetree.

[10] Nordic Semi. What shall i choose zephyr or nrf connect sdk.

[11] Nordic Semi. The posix architecture.

[12] Nordic Semi. Nordic product guide.

[13] Baseline Equipment. Gnss vs gps: Examining the differences.

[14] PC Mag. The best bluetooth trackers for 2024.

[15] Novel Bits. Coded phy: Bluetooth's long-range feature.

[16] Elecia White. *Making Embedded Systems*. O'Reilly Media, Inc, 2nd edition.

[17] Nordic Semi. nrf9160, low power sip with integrated lte-m/nb-iot modem and gnss.

[18] Nordic Semi. nrf9160 product specification v2.1.

[19] Digikey. What you need to know to choose an accelerometer.

[20] Wikipedia. Successive approximation adcr.

[21] Muhammad Haris Arshad, Muhammad Bilal, and Abdullah Gani. Human activity recognition: Review, taxonomy and open challenges. *Sensors*, 22(17):6463, 2022.

[22] Whistle. Powered by the pet insight project.

[23] Wikipedia. Nyquist-shannon sampling theorem.

[24] Byron Du Preez Simon Chamaillé-Jammes Andrew J. Loveridge David W. Macdonald Matthew Wijers, Paul Trethowan and Andrew Markham. Vocal discrimination of african lions and its potential for collar-free tracking. *Bioacoustics*, 30(5):575–593, 2021.

[25] Whistle. How does whistle ™ estimate how far my dog has traveled?

[26] ynvisible. 6 of the best iot display technologies compared.

[27] Battery Specialists. What is the difference between a li-polymer and li-ion battery.

[28] Wikipedia. List of battery sizes.

[29] Nordic Semi. Bare-metal vs rtos programming.