CHAPTER 1

Prototype Design

As discussed in the Literature Review, using an Infrared Array Sensor (IAR) appear to be the most viable way to achieve the high-level goals of this project. Thermosense [4], the primary occupancy sensor in the IAR space, used the low-cost Panasonic Grid-EYE sensor for this task. This sensor, costing around $30USD, appears to be a prime candidate for use in this project, as it satisfied low-cost criteria, as well as being proven by Thermosense to be effective in this space. However, while still available for sale in the United States, we were unable to order the sensor for shipping to Australia due to export restrictions outside of our control. While such restrictions would be circumventable with sufficient effort, using a sensor with such restrictions in place goes against an implicit criteria of the parts used in the project being relatively easy to acquire.

This forced us to search for alternative sensors in the space that fulfill similar criteria but were more broadly available. The sensor we settled on was the Melexis MLX90620 (*Melexis*) [5], an IAR with similar overall qualities that differed in several important ways; it provides a 16×4 grid of thermal information, it has an overall narrower field of view and it sells for approximately $80USD. Like the Grid-EYE , the *Melexis* sensor communicates over the 2-wire I^2C bus, a low-level bi-directional communication bus widely used and supported in embedded systems.

In an idealized version of this occupancy system, much like Thermosense this system would include wireless networking and a very small form factor. However, due to time and resource constraints, the scope of this project has been limited to a minimum viable implementation. Appendix Chapter **??** on page ?? discusses in detail how the introduction of new open standards in the Wireless Personal Area Network space could be used in future systems to provide robust, decentralized networking of future occupancy sensors. This prototype architecture has been designed such that a clear path to the idea system architecture discussed therein is available.

Analysis Tier	Raspberry Pi B+
Preprocessing Tier	Arduino Uno R3
Sensing Tier	Melexis MLX90620 & PIR

Table 1.1: Hardware tiers

1.1 Hardware

As reliability and future extensibility are core concerns of the project, a three-tiered system is employed with regards to the hardware involved in the system (Table 1.1). At the bottom, the Sensing Tier, we have the raw sensor, the Melexis MLX90620 (*Melexis*), which communicate over I^2C . Connected to these devices via those respective protocols is the Preprocessing Tier, run an embedded system. The embedded device polls the data from these sensors, performs necessary calculations to turn raw information into suitable data, and communicates this via Serial over USB to the third tier. The third tier, the Analysis Tier, is run on a fully fledged computer. In our prototype, it captures and stores both video data, and the Temperature and Motion data it receives over Serial over USB.

While at a glance this system may seem overly complicated, it ensures that a sensible upgrade path to a more feature-rich sensing system is available. In the current prototype, the Analysis Tier merely stores captured data for offline analysis, in future prototypes this analysis can be done live and served to interested parties over a RESTful API. In the current prototype, the Analysis and Sensing Tiers are connected by Serial over USB, in future prototypes, this can be replaced by a wireless mesh network, with many Preprocessing/Sensing Tier nodes communicating with one Analysis Tier node.

Due to low cost and ease of use, the Arduino platform was selected as the host for the Preprocessing Tier, and thus the low-level I^2C interface for communication to the *Melexis*. Initially, this presented some challenges, as the *Melexis* recommends a power and communication voltage of 2.6V, while the Arduino is only able to output 3.3V and 5V as power, and 5V as communication. Due to this, it was not possible to directly connect the Arduino to the *Melexis*, and similarly due to the two-way nature of the I^2C 2-wire communication protocol, it was also not possible to simply lower the Arduino voltage using simple electrical techniques, as such techniques would interfere with two-way communication.

A solution was found in the form of a I^2C level-shifter, the Adafruit "4-channel I2C-safe Bi-directional Logic Level Converter" [1], which provided a cheap method to bi-directionally communicate between the two devices at their own preferred voltages. The layout of the circuit necessary to link the Arduino

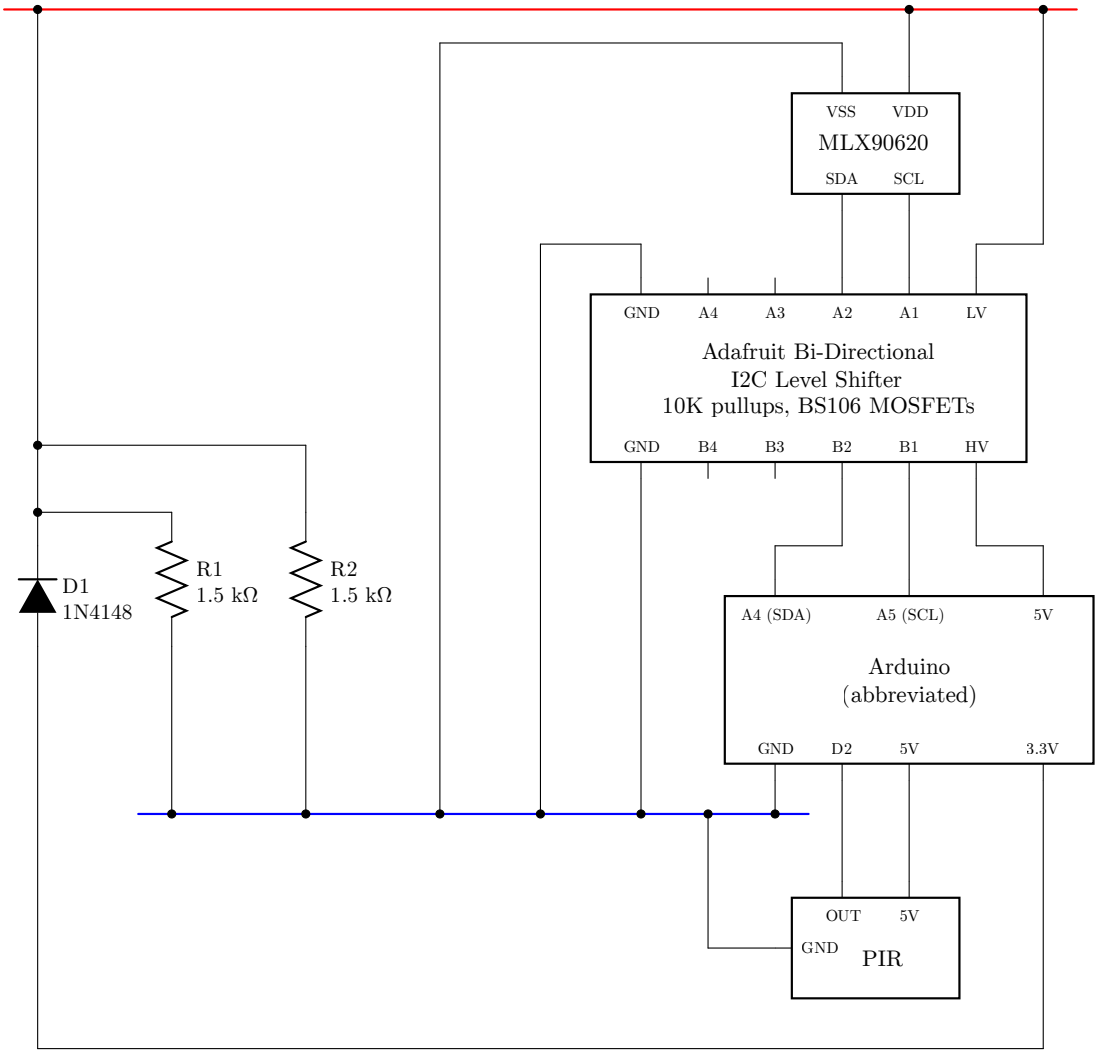

Figure 1.1: MLX90620, PIR and Arduino integration circuit

and the *Melexis* using this converter can be seen in Figure 1.1 on the preceding page.

Additionally, as used in the Thermosense paper, a Passive Infrared Sensor (PIR) motion sensor [2] was also connected to the Arduino . This sensor, operating at 5V natively, did not require any complex circuitry to interface with the Arduino . It is connected to digital pin 2 on the Arduino , where it provides a rising signal in the event that motion is detected, which can be configured to cause an interrupt on the Arduino . In the configuration used in this project, the sensor's sensitivity was set to the highest value and the timeout for re-triggering was set to the lowest value (approximately 2.5 seconds). Additionally, the continuous re-triggering feature (whereby the sensor produces continuous rising and falling signals for the duration of motion) was disabled using the provided jumpers.

For the Analysis Tier, the Raspberry Pi B+ was chosen, as it is a powerful computer capable of running Linux available for an extraordinarily low price. The Arduino is connected to the Raspberry Pi over USB, which provides it both power and the capacity to transfer data. In turn, the Raspberry Pi is connected to a simple micro-USB rechargeable battery pack, which provides it with power, and subsequently the Arduino and sensors.

1.2 Software

At each layer of the described three-tier software architecture (pictured in greater detail in Figure 1.2 on page 6), there must exist software which governs the operation of that tier's processing concerns. Software in this project was written in two different languages.

At the Sensing Tier, it was not necessary for any software to be developed, as any software necessary came pre-installed and ready for use on the aforementioned sensors.

At the Preprocessing Tier, the Arduino, the default C++ derivative language was used, as careful management of memory usage and algorithmic complexity is required in such a resource-constrained environment, thus limiting choice in the area.

Finally, at Analysis Tier, a computer running fully-fledged Linux, choice of language becomes a possibility. In this instance, Python was settled on as the language of choice, as it is a quite high-level language with excellent library support for the functions required of the Analysis Tier, including serial interface, the use of the Raspberry Pi's built in camera, and image analysis. The 2.x branch

of Python was chosen over the 3.x branch, despite its age, due a greater maturity in support for several key graphical interface libraries.

1.2.1 Pre-processing: `mlx90620_driver.ino`

On the Arduino, once large program was developed, termed `mlx90620_driver.ino`. This program's purpose was to take simple commands over serial to configure the Melexis MLX90620 (*Melexis*) and to report back the current temperature values and Passive Infrared Sensor (PIR) motion information at either a pre-set interval, or when requested.

To calculate the final temperature values that the *Melexis* offers, a complex initialization and computational process must be followed, which is specified in the sensor's datasheet [5]. This process involves initializing the sensor with values attained from a separate on-board I^2C EEPROM, then retrieving a variety of normalization and adjustment values, along with the raw sensor data, to compute the final temperature result.

The basic algorithm to perform this normalization was based upon the provided datasheet [5], as well as code by users "maxbot", "IIBaboomba", "nseidle" and others on the Arduino Forums [3] and was modified to operate with the newer Arduino "Wire" I^2C libraries released since the authors' posts. In pursuit of the project's aims to create a more approachable thermal sensor, the code was also restructured and rewritten to be both more readable, and to introduce a set of features to make the management of the sensor data easier for the user, and for the information to be more human readable.

Additionally, support for the PIR's motion data was added to the code, with the PIR configured to perform interrupts on one of the Arduino's digital pinsnd the code structured to take note of this information and to report it to the user in the "MOTION" section of the next packet.

The first of the features introduced was the human-readable format for serial transmission. This allows the user to both easily write code that can parse the serial to acquire the serial data, as well as examine the serial data directly with ease. When the Arduino first boots running the software, the output in Figure 1.3 on page 7 is output. This specifies several things that are useful to the user; the attached sensor ("DRIVER"), the build of the software ("BUILD") and the refresh rate of the sensor ("IRHZ"). Several different headers, such as "ACTIVE" and "INIT" specify the current millisecond time of the processor, thus indicating how long the execution of the initialization process took (33 milliseconds).

Once booted, the user is able to send several one-character commands to

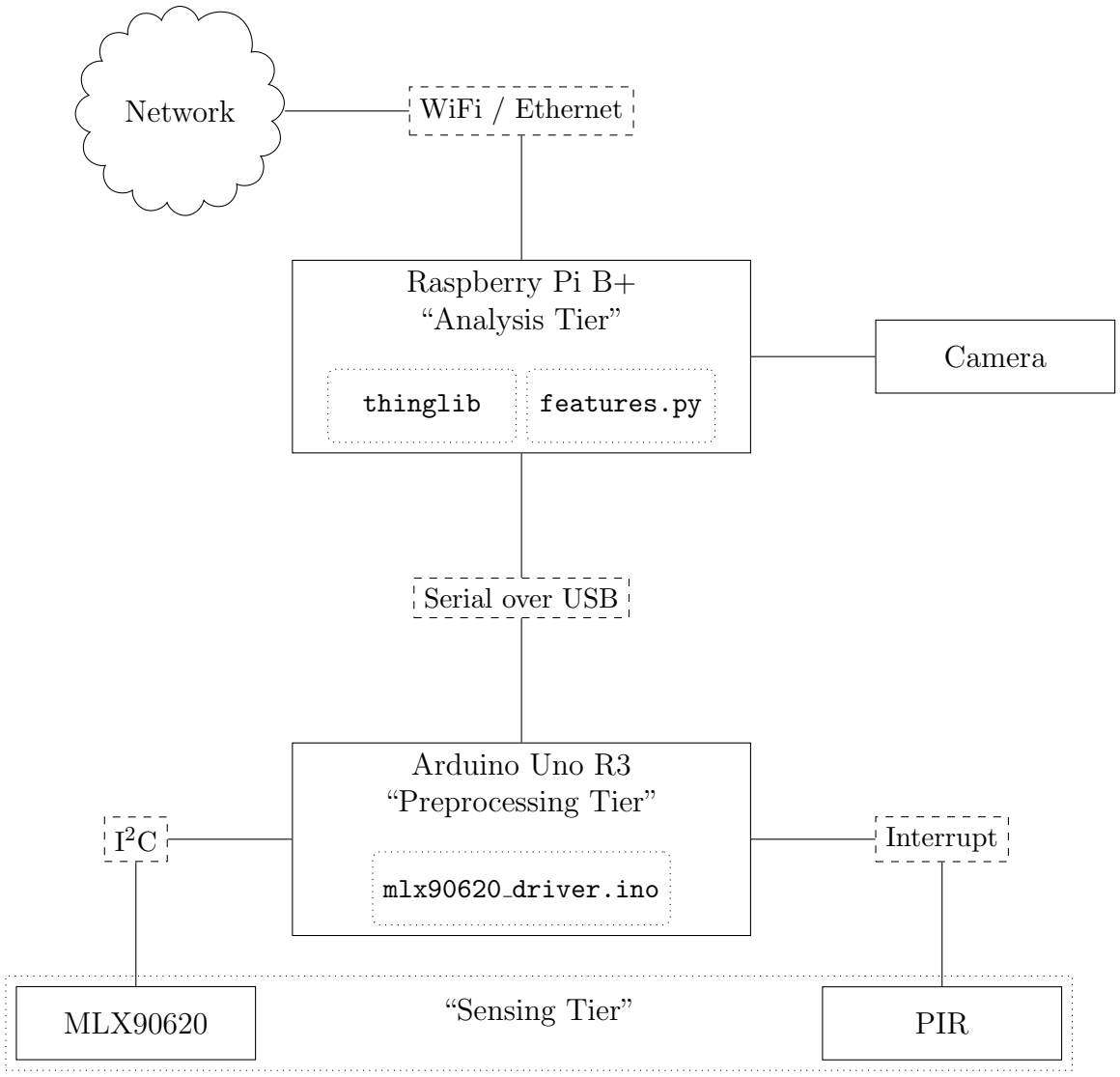

Figure 1.2: Prototype system architecture

```
INIT 0
INFO START
DRIVER MLX90620
BUILD Feb  1 2015 00:00:00
IRHZ 1
INFO STOP
ACTIVE 33

START 34
MOVEMENT 0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
STOP 97
```

Figure 1.3: Initialisation sequence and thermal packet

the sensor to configure operation, which are described in Table **??** on page ??. Depending on the sensor configuration, IR data may be periodically output automatically, or otherwise manually triggered. This IR data is produced in the packet format described in Figure 1.3. This is a simple, human readable format that includes the millisecond time of the processor at the start and end of the calculation, if the PIR has seen any motion for the duration of the calculation, and the 16x4 grid of calculated temperature values.

1.2.2 Analysis: `thinglib`

On the analysis tier a set of Python libraries and accompanying utility scripts were developed to interface with the Arduino, parse and interpret its data, and to provide data logging and visualization capabilities. Most of this functionality was split into a reusable and versatile Python module called `thinglib`.

`thinglib` provides 4 main feature sets across 3 files; the `Manager` series of classes, the `Visualizer` class, the `Features` class and the `pxdisplay` module.

1.2.2.1 `Manager` classes

The Manager series of classes are the direct interface between the Arduino and the Python classes. They implement a multi-threaded serial data collection and parsing system which converts the raw serial output of the connected Arduino into a series of Python data structures that represent the collected temperature and motion data of each captured frame. Several different versions of the `Manager` class exist to perform slightly different functions. When initializing these classes

the sample rate of the *Melexis* can be configured, and it will be sent through to the Arduino for updating.

`BaseManager` is responsible for the implementation of the core serial parsing functions. It also provides a threaded interface through which the *Melexis*'s continuous stream of data can be subscribed to by other threads. The primary API, the `subscribe_` series of functions, return a thread-safe queue structure, through which thermal packets can be received by various other threads when they become available.

`Manager`, the primary class, provides access the *Melexis*'s data at configurable intervals. When initializing this class, you may specific 0.5, 1, 2, 4 or 8Hz, and the class will configure the Arduino to both set the *Melexis* to this sample rate, and to automatically write this data to the serial buffer whenever it is available. This serial interface is multi-threaded, as at higher serial baud rates if data was not polled continuously enough the internal serial buffer would fill and some data would be discarded. By ensuring this process cannot be blocked by other parts of the running program this problem is mostly eliminated.

`OnDemandManager` operates in a similar way to `Manager`, however instead of using a non-blocking threaded approach, the user's scripts may request thermal/motion data from the class, and it will poll the Arduino for information and block until this information is parsed and returned.

Finally, `ManagerPlaybackEmulator` is a simple class which can take a previously created thermal recording from a file, and emulate the `Manager` class by providing access to thread-safe queues which return this data at the specified Hz rate. This class can be used as a means to playback thermal recordings with the same visualization functions.

1.2.2.2 `pxdisplay` functions

The `pxdisplay` module is a set of functions that utilize the `pygame` library to create a simple live-updating window containing a thermal map representation of the thermal data. One can generate any number of `pxdisplay` objects, which leverage the `multithreading` library and `multithreading.Queue` to allow thermal data to be sent to the display.

The class also provides a set of functions to set a "hotest" and "coldest" temperature and have RGB colors assigned from red to blue for each temperature that falls between those two extremes.

1.2.2.3 `Visualizer` class

The `Visualizer` class is the natural compliment to the `Manager` series of classes. The functions contained within can usually be provided with a Queue object (generated by a `Manager` class) and can perform a variety of visualization and storage functions.

From the recording side, the `Visualizer` class can "record" a thermal capture by saving the motion and thermal information to a simple `.tcap` file, which stores the sample rate, timings, thermal and motion data from a capture in a very straightforward format. The class can also read these files back into the data structures `Visualizer` uses internally to store data. If `Visualizer` is running on a Raspberry Pi, it can also leverage the `picamera` library an the `OnDemandManager` class to synchronously capture both visual and thermal data for ground truth purposes.

From the visualization side, `Visualizer` can leverage the `pxdisplay` module to create thermal maps that can update in real-time based on the thermal data provided by a `Manager` class. The class can also generate both images and movie files from thermal recordings using the PIL and ffmpeg libraries respectively.

1.2.2.4 `Features` class

In Thermosense [4], an algorithm was demonstrated that allowed the separation of "background" information from "active" pixels, and from that information, the extraction of the features necessary for a classifier to correctly determine the number of people in an 8×8 thermal image. This algorithm involved calculating the average and standard deviations of each pixel while it is guaranteed that the image would be empty, and then when motion is detected, considering any pixel "active" that reaches a value more than 3 standard deviations above the pixel when there was no motion.

From these "active" pixels, it was established that a set of three feature vectors were all that were required to correctly classify the number of people in the thermal image. These feature vectors were;

1. **Number of active pixels**: The total number of pixels that are considered "active" in a given frame

2. **Number of connected components**: If each active pixel is represented as an node in an undirected graph where adjacent active pixels are connected, how many connected components does this graph have?

3. **Size of largest connected component**: The number of active pixels contained within the largest connected component

In accordance with the pseudo-code outlined in the Thermosense paper, the algorithm described in Listing 1.1 on the next page was created to extract these figures. The portion of this code dealing with scaling the thermal background for rooms without motion was not implemented, as in all experiments tested, there exists a significant interval of time during which the no motion is guarenteed and the thermal background can be generated. The `networkx` library was used to generate the connected components information.

```
import networkx, itertools

nomotion_wgt = 0.01
n_rows = 4
n_cols = 16
background = first_frame
means = first_frame
stds = [ [0]*16 ]*4
stds_post = [ [None]*16 ]*4

def create_features(new_frame, is_motion):
  active = []
  g = networkx.Graph()

  for i, j in itertools.product( range(n_rows), range(n_cols) ):
    prev = background[i][j]
    cur = new_frame[i][j]
    cur_mean = means[i][j]
    cur_std = stds[i][j]

    if not is_motion:
      background[i][j] = nomotion_wgt * cur + (1 - nomotion_wgt) * prev
      means[i][j]      = cur_mean + (cur - cur_mean) / n
      stds[i][j]       = cur_std + (cur - cur_mean) * (cur - means[i][j])
      stds_post[i][j]  = math.sqrt(stds[i][j] / (n-1))

    if (cur - background[i][j]) > (3 * stds_post[i][j]):
      active.append((i,j))
      g.add_node((i,j))

      # Add edges for nodes that have already been computed as active
      for ix, jx in [(-1, -1), (-1, 0), (-1, 1), (0, -1)]:
        if (i+ix, j+jx) in active:
          g.add_edge((i,j), (i+ix,j+jx))

  comps = list(networkx.connected_components(g))
  num_active = len(active)
  num_connected = len(comps)
  size_connected = max(len(c) for c in comps) if len(comps) > 0 else None

  return (num_active, num_connected, size_connected)
```

Listing 1.1: Core feature extraction code

1.3 Sensor Properties

In order to best utilize the Melexis MLX90620 (*Melexis*), we must first understand the properties it exhibits, and their potential affects on our ability to perform person related measurements. These properties can be broadly separated into three different categories; bias, noise and sensitivity. A broad range of data was collected with the sensor in a horizontal orientation using various sources of heat and cold to determine these properties. This experimental setup is described in Figure 1.4 on the following page.

1.3.1 Bias

When receiving no infrared radiation, the sensor should indicate a near-zero temperature. If in such conditions it does not, that indicates that the sensor has some level of bias in its measurement values. We attempted to investigate this bias by performing thermal captures of the night sky. While this does not completely remove the infrared radiation, it does remove a significant proportion of it.

In Table 1.2 on page 14 the thermal sensor was exposed to the night sky at a capture rate of 1Hz for 4 minutes, with the sensing results combined to create a set of means and standard deviations to indicate the pixels at "rest". The average temperature detected was 11.78°C, with the standard deviation remaining less than 0.51°C over the entire exposure period. The resultant thermal map shows that pixels centered around the four "primary" pixels in the center maintain a similar temperature around 9°C, with temperatures beginning to deviate as they became further from the center.

The most likely cause of this bias is related to the physical structure of the sensor. The *Melexis* is a rectangular sensor which has been placed inside a circular tube. Due to this physical arrangement, the sides of this rectangular sensor will be significantly closer to these edges than the center. If these sides are at an ambient temperature higher than the measurement data (as they were in this case) thermal radiation from the sensor package itself could provide significant enough to cause the edges to appear warmer than the observed area of the sky. Such issues with temperature could be controlled for using a device that cools the sensor package to below that of the ambient temperature being measured, however, this is not a concern in this project, as the method of calculating a thermal background will compensate for any such bias as long as it remains constant.

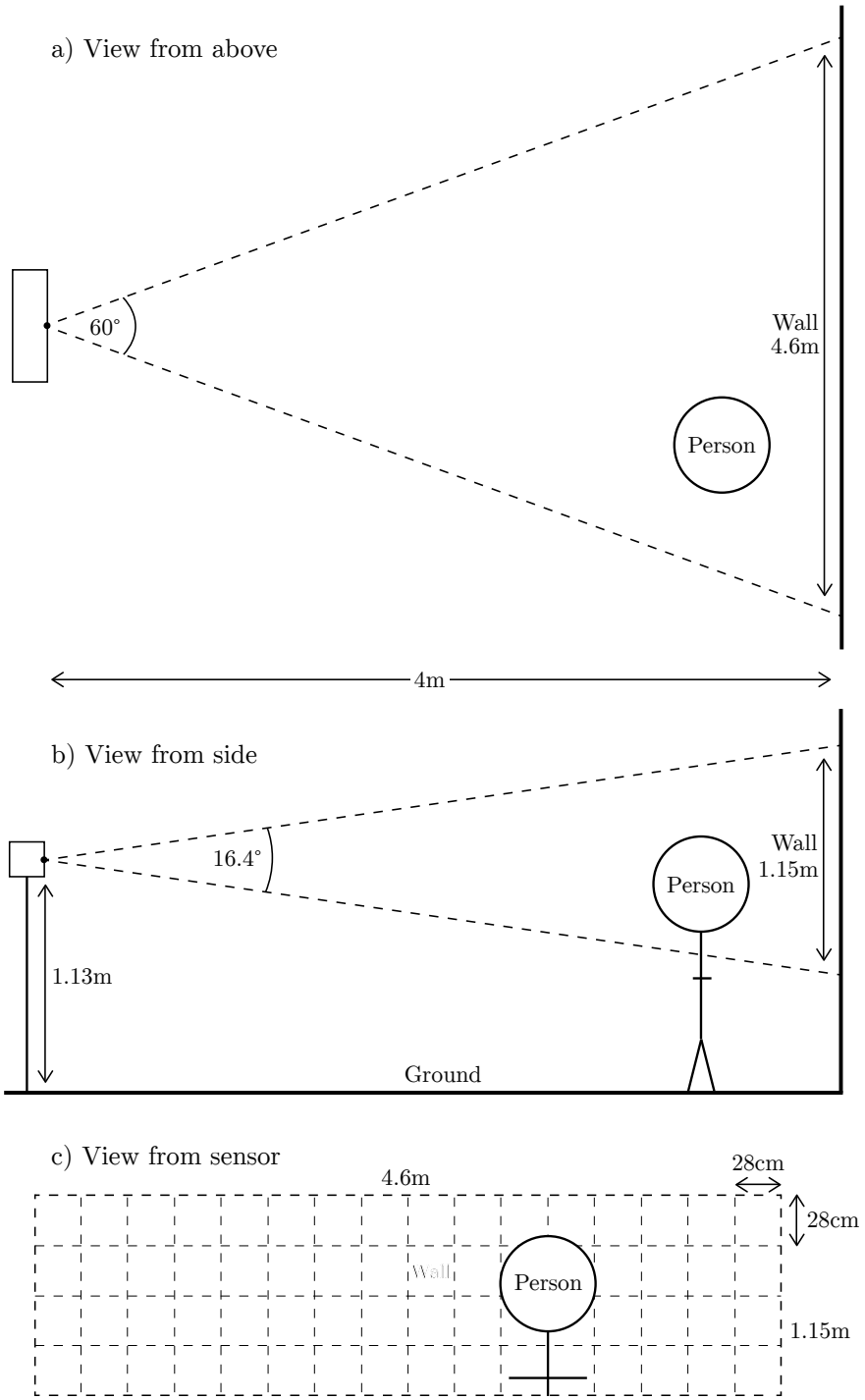

Figure 1.4: Experiment setup to determine sensor properties

14.95 0.51	14.33 0.27	12.34 0.27	8.77 0.33	8.15 0.31	10.84 0.38	9.02 0.26	7.79 0.37	6.67 0.27	9.63 0.29	9.29 0.26	8.24 0.27	9.84 0.25	14.28 0.33	14.92 0.3	13.16 0.25
14.54 0.34	15.62 0.31	12.73 0.23	11.51 0.27	11.79 0.26	11.47 0.27	11.43 0.29	9.02 0.35	8.57 0.23	11.15 0.23	10.64 0.22	10.3 0.24	12.09 0.22	14.49 0.26	14.88 0.31	14.71 0.36
18.25 0.45	16.62 0.31	14.15 0.24	11.97 0.34	13.11 0.3	12.64 0.22	10.66 0.23	9.15 0.24	9.58 0.28	11.95 0.28	11.22 0.24	11.52 0.36	11.11 0.23	12.59 0.25	14.44 0.31	13.35 0.28
16.02 0.28	16.81 0.36	15.0 0.25	11.53 0.28	10.18 0.29	12.2 0.25	11.78 0.29	8.36 0.31	8.15 0.33	10.36 0.32	10.74 0.31	8.25 0.36	9.99 0.35	12.42 0.38	11.39 0.4	11.06 0.34

Table 1.2: Mean and standard deviations for each pixel at rest

1.3.2 Noise

One of the features of the *Melexis* is the ability to sample the thermal data and a variety of sample rates between 0.5Hz and 512Hz. However, it was noted in early experimentation that a higher sample rate resulted in an increase in the noise contained within the resultant images. As our experiments focus on separating objects of interest from a thermal background, it is important to determine the maximum level of noise tolerable before our algorithms are unable to separate the background from the objects of interest.

Figure 1.5 on the following page plots one of the central pixels of the sensor in a scenario where it is merely viewing a background (shown in green), and when it is viewing a person (shown in red), at the 5 different sample rates achievable with the current hardware. We can see in these plots that the data becomes significantly more noisy as the sample rate increases, and we can also determine that the sensor uses a form of data smoothing at lower sample rates, as the variance in data increases with sample rate.

If the sample rate were to increase, it is likely that the ability for the sensing system to disambiguate between objects of interest and the background would diminish. However, in the current project, even the slowest sampling rate of 0.5Hz is sufficient, as occupancy estimations at a sub-second level present little additional value and would require significant reforms in the efficiency of the software used.

1.3.3 Sensitivity

The *Melexis* is a sensor composed of 64 independent non-contact digital thermopiles, which measure infrared radiation to determine the temperature of objects. While they are bundled in one package, Figure 1.6 on page 17 shows that they are in fact wholly independent sensors placed in a grid structure. This has important effects on the properties of the data that the *Melexis* produces.

Figure 1.7 on page 18 shows a graph of the temperatures of the top row of 16 pixels of the *Melexis* as a hot object is moved from left to right at an approximately similar speed. One of the most interesting phenomena in this graph is the apparent extreme variability of the detected temperature of the object as it moves "between" two different pixels; there is a noticeable drop in the objects detected temperature. Further analysis of each of the pixel's lines on the graph shows each pixel exhibiting a bell-curve like structure, with the detected temperature increasing from the baseline and peaking as the object enters the center of the pixel, and the detected temperature similarly decreasing

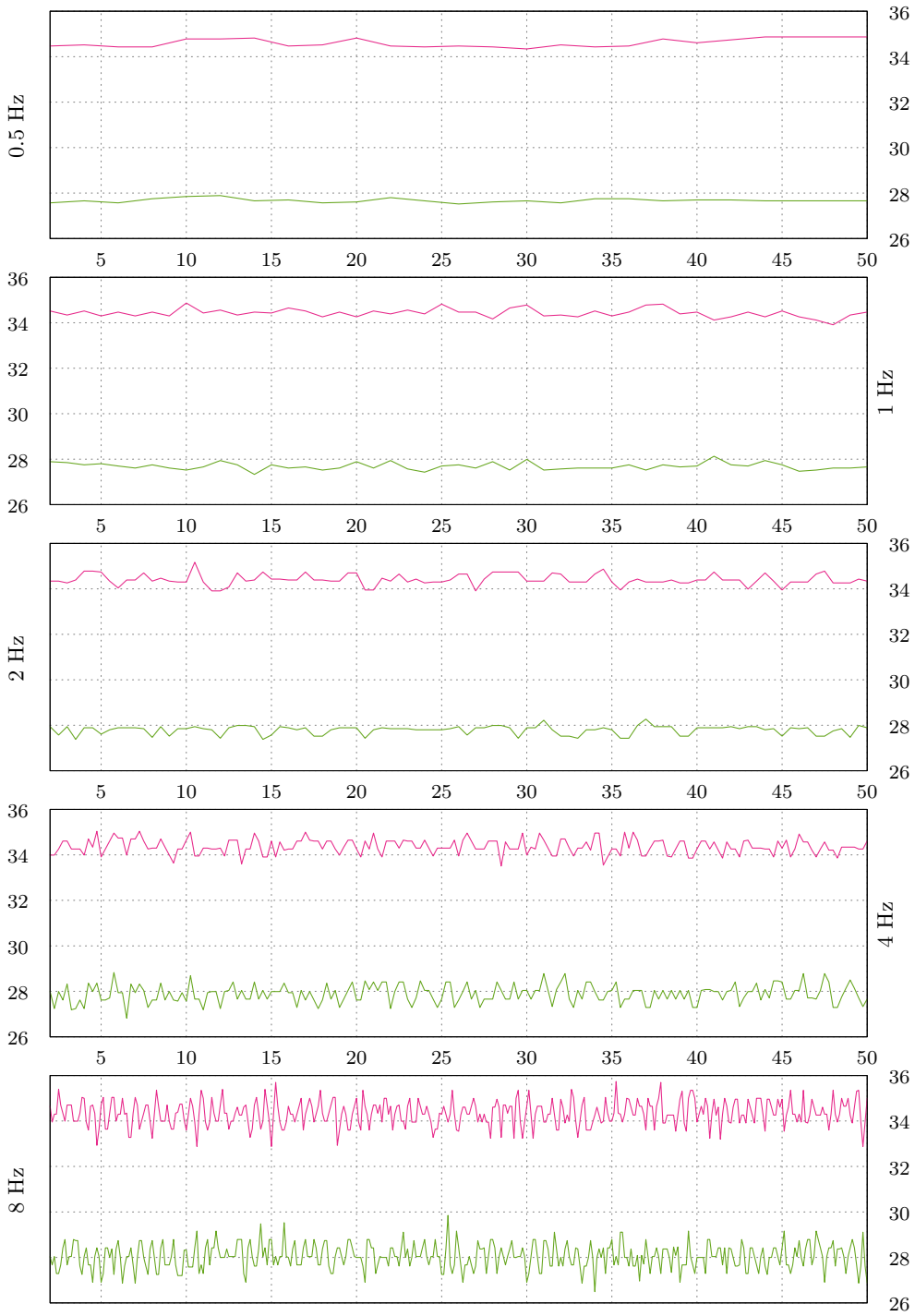

Figure 1.5: Comparison of noise levels at the *Melexis*' various sampling speeds

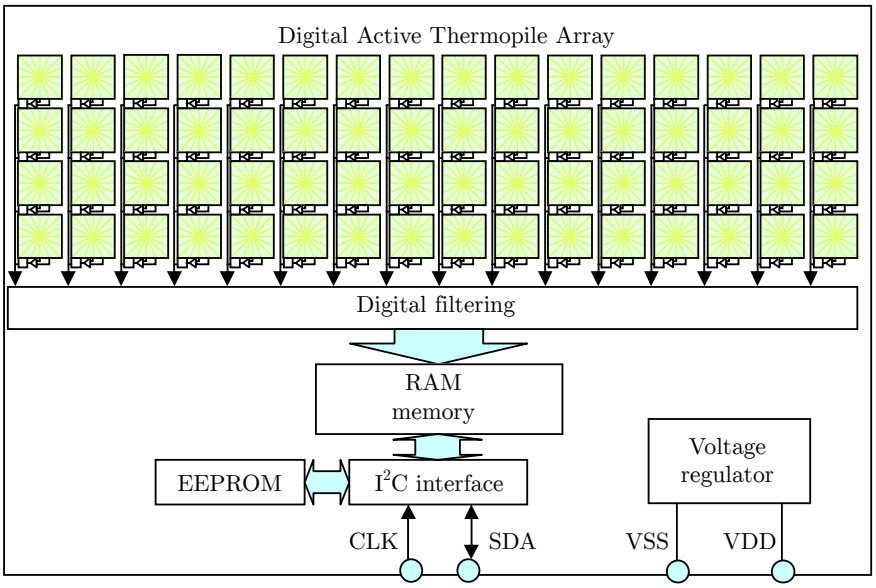

Figure 1.6: Block diagram for the *Melexis* taken from datasheet [5]

as the object leaves the center.

This phenomenon has several possible causes. One likely explanation is that each individual pixel detects objects radiating at less favorable angles of incidence to be colder than they actually are: As the object enters a pixel's effective field of view, it will radiate into the pixel at an angle that is at the edge of the pixel's ability to sense, with this angle slowing decreasing until the hot object is directly radiating into the pixel's sensor, causing a peak in the temperature reading. As the object leaves the individual elements field of view, the same happens in reverse.

While interesting, this phenomenon has little consequence to the effectiveness of the techniques used, as in experimental conditions the sensor will not be sufficiently distant that humans could be detected as single pixels. However, this phenomenon could be leveraged in future work to perform sub-pixel localization, discussed later on.

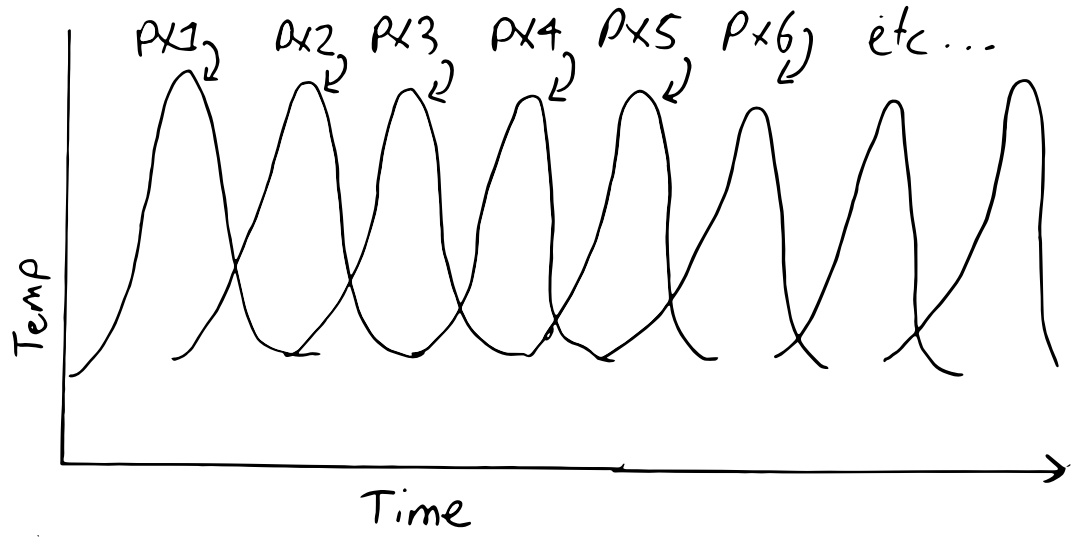

Figure 1.7: Different *Melexis* pixel temperature values as hot object moves across row

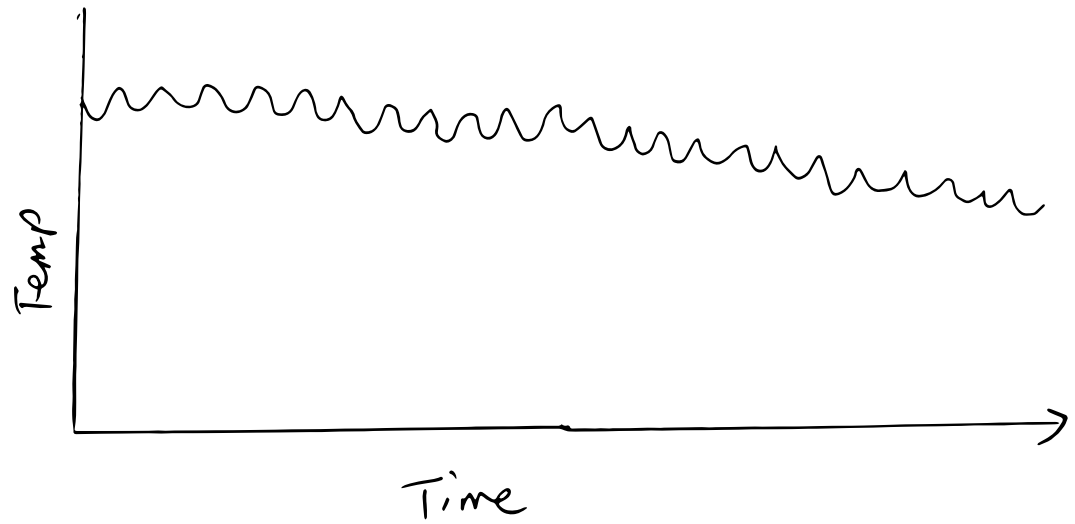

Figure 1.8: Variation in temperature detected for hot object at 1Hz sampling ration

Bibliography

[1] Adafruit. 4-channel I2C-safe bi-directional logic level converter - BSS138 (product ID 757). http://www.adafruit.com/product/757. Accessed: 2015-01-07.

[2] Adafruit. PIR (motion) sensor (product ID 189). http://www.adafruit.com/product/189. Accessed: 2015-02-08.

[3] Arduino Forums. Arduino and MLX90620 16X4 pixel IR thermal array. http://forum.arduino.cc/index.php/topic,126244.0.html, 2012. Accessed: 2015-01-07.

[4] Beltran, A., Erickson, V. L., and Cerpa, A. E. ThermoSense: Occupancy thermal based sensing for HVAC control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–8.

[5] Melexis. Datasheet IR thermometer 16X4 sensor array MLX90620. http://www.melexis.com/Asset/Datasheet-IR-thermometer-16X4-sensor-array-MLX90620-DownloadLink-6099.aspx, 2012. Accessed: 2015-01-07.