CHAPTER 1

Design and Implementation

With the Literature Review concluding that a design based on ThermoSense would be most appropriate, a software and hardware prototype (the “sensing system”) must now be constructed to provide a platform for experimentation and evaluation of the sensor chosen, as well as to capture, store, visualize and replay sensor data for those purposes. We will first discuss the hardware foundations of the project, then the architecture of the software developed to run on those foundations.

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 sensors themselves. Connected to the sensors via those respective protocols is the Preprocessing Tier, hosted on an embedded system. The embedded device polls the data from these sensors, performs necessary calculations to turn the raw sensor information into actionable 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 temperature and motion data it receives over Serial over USB, as well as visual data for ground truth purposes.

While at a glance this system may appear to be complex, the system has been designed to ensure 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.

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

Table 1.1: Hardware tiers

1.1.1 Sensing

As discussed in the Literature Review, using a Thermal Detector Array (TDA) appear to be the most viable way to achieve the high-level goals of this project. ThermoSense [4], the primary occupancy sensor in the TDA space, used the low-cost Panasonic Grid-EYE sensor for this task. This sensor, costing around $50, 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], a TDA 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 $80. Like the Grid-EYE, the *Melexis* communicates over the 2-wire I^2C bus, a low-level bi-directional communication bus widely used and supported in embedded systems.

We envision a further advanced prototype to have wireless networking in a smaller form factor, much like ThermoSense. However, due to time and resource constraints, the scope of this project has been limited to a minimum viable implementation. This prototype architecture has been designed such that a clear path to an idea system architecture involving each Pre-Processing Tier and Analysis Tier being connected by a wireless mesh network to enable easy installation in households.

1.1.2 Pre-Processing

Due to low cost, broad support and ease of development, the Arduino platform was selected as the host for the Preprocessing Tier, and thus will handle the I^2C communication with the *Melexis*. Initially, this presented some challenges, as

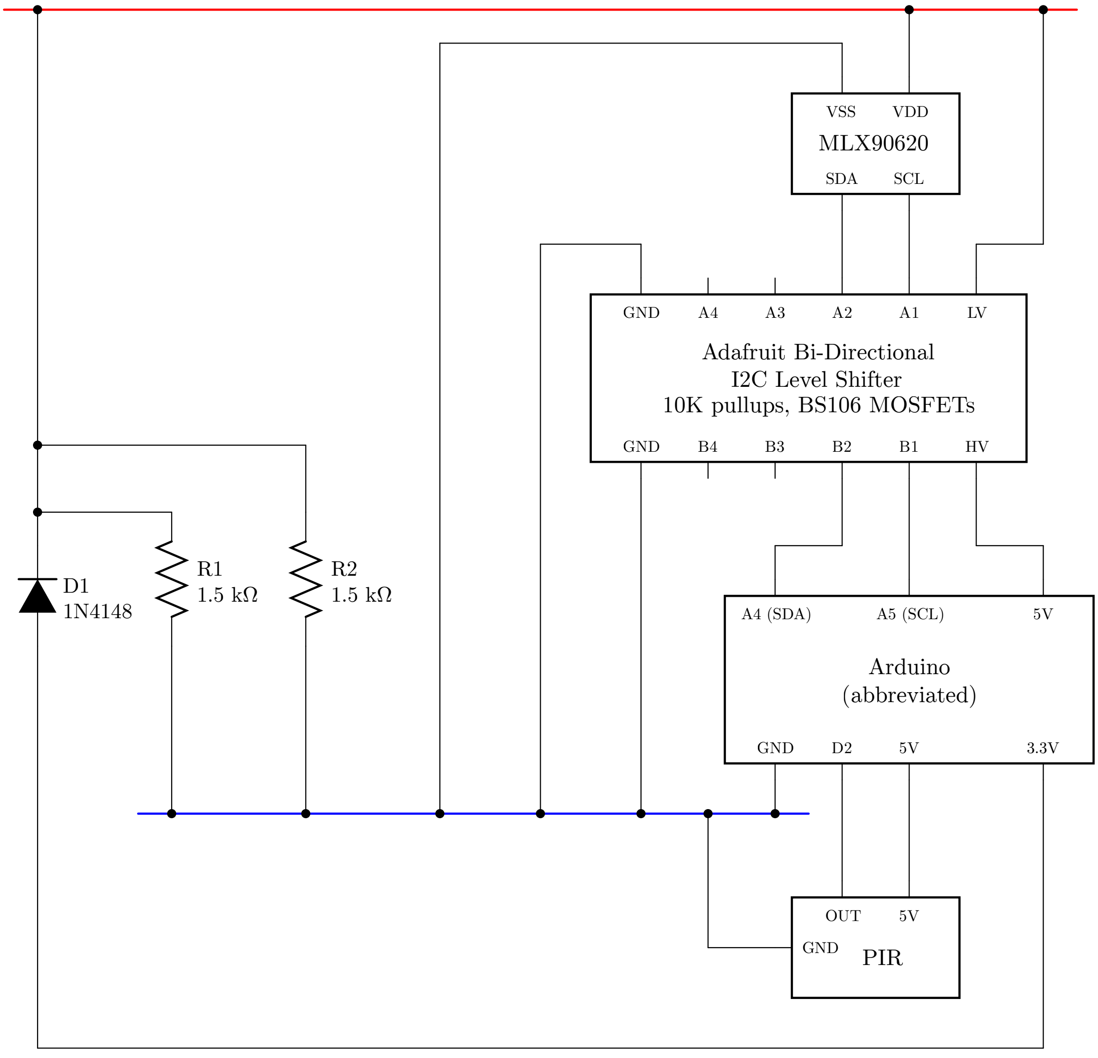

Figure 1.1: MLX90620, PIR and Arduino integration circuit

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 and the *Melexis*, including the use of this converter, can be seen in Figure 1.1.

Additionally, as used in the ThermoSense paper, a Passive Infrared Sensor (PIR) motion detector [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 (a "trigger") 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 (the trigger reoccurring) 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.

1.1.3 Analysis / Classification

For the Analysis Tier, the Raspberry Pi B+ was chosen, as it is a powerful and inexpensive computer capable of running Linux. 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.

1.1.4 Component Costs

As being low-cost is one of the project's goals, we have summarized the cost of each of the components of the prototype in Table 1.2a. We believe that for a prototype, this cost is sufficiently low. In the envisioned system, there would only be one Raspberry Pi in the system, and it would not require a camera, lowering

Part	Cost
MLX90620	$80
Raspberry Pi B+	$50
Arduino Uno R3	$40
Passive Infrared Sensor	$10
TOTAL	**$180**

(a) Our project

Part	Cost
TMote Sky	$110
Grid-EYE	$50
Passive Infrared Sensor	$10
TOTAL	**$170**

(b) ThermoSense (estimated)

Table 1.2: Minimum viable component cost comparison

Category	SLOC
TArL Python	674
`cam`	425
`features`	191
`pxdisplay`	58
TArL Arduino	492
`mlx90620_driver`	492
Analysis Scripts	147
Capture Scripts	234
Total	*1,624*

(a) Source Lines Of Code written

Library	Version
Arduino	
SDK	1.6.4
`SimpleTimer`	1.0
Python	
`networkx`	1.9.1
`numpy`	1.8.0
`matplotlib`	1.3.1
`picamera`	1.10
`Pillow`	2.8.1

(b) Libraries used

Table 1.3: Overview of code used in project

the cost to around $\$50 + \$130n$ where n is the number of sensors. Similarly, as technology improves, sensor technology expected to continue to fall in price, causing the most expensive component, the infrared sensor, to become increasingly cost-effective.

When we compare this to the estimated cost of the ThermoSense system (Table 1.2b), we believe that it achieves a suitably comparable cost for a prototype. When removing the aspects of the prototype that would be unnecessary in the final version, the difference is only $10.

1.2 Software

At each layer of the described three-tier software architecture (pictured in greater detail in Figure 1.2), software exists to govern the operation of that tier's functionality. For the sensing tier, the Melexis MLX90620 (*Melexis*)'s own software

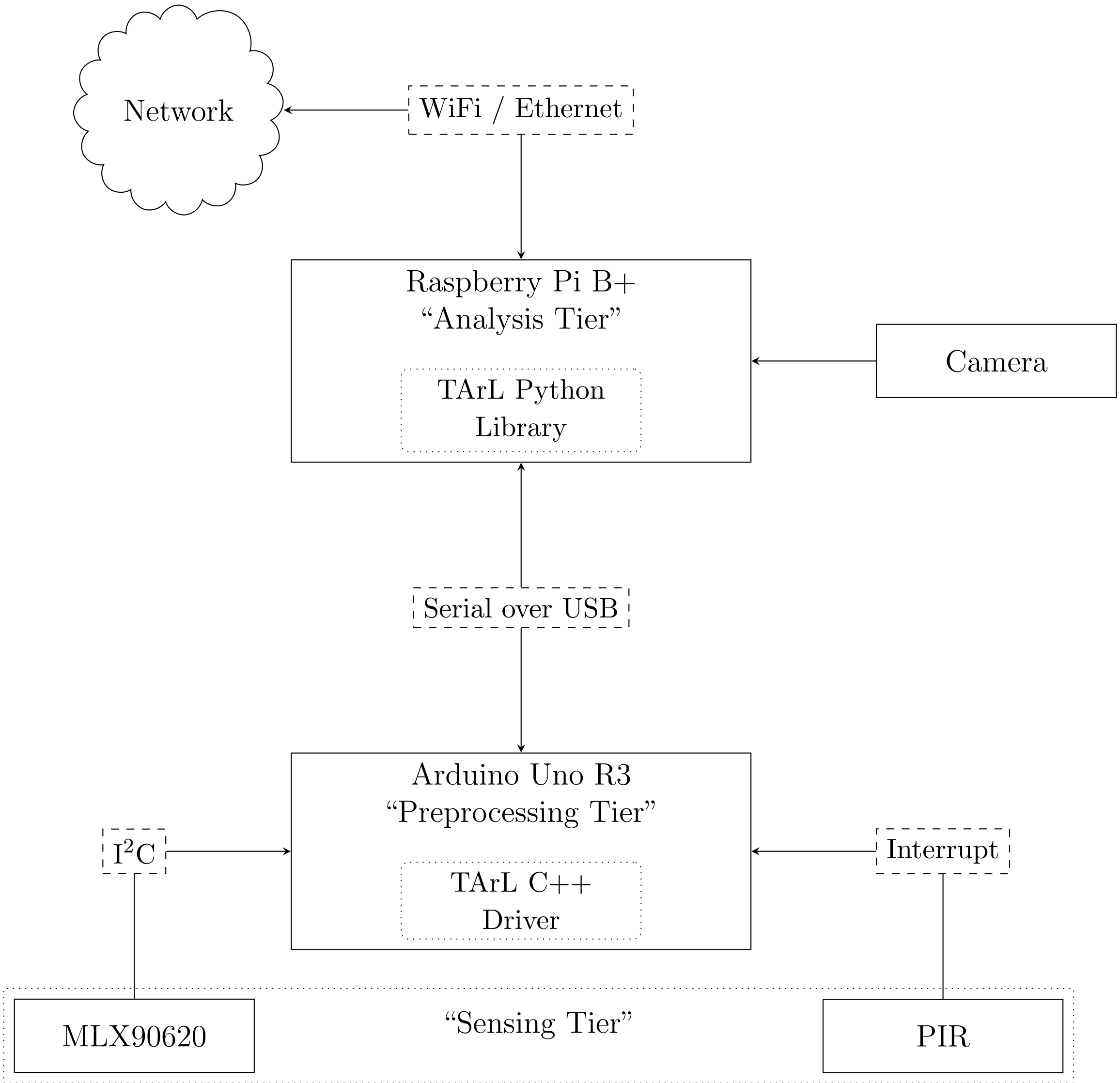

Figure 1.2: Prototype system architecture

is used, while for the other tiers, a bi-lingual software library, the Thermal Array Library (TArL), was developed to provide a suite of functions to enable the easy data collection and analysis of information from the hardware prototype. TArL is split into two parts:

At the Preprocessing Tier, the Arduino, the TArL *Melexis* driver is found, which is written in the default Arduino C++ derivative language. The used of a low-level language is important at this tier as careful management of memory usage and algorithmic complexity is required in such a resource-constrained environment.

At Analysis Tier, a computer running fully-fledged Linux, the bulk of TArL can be found. As the processing environment has become less constrained, the choice of language becomes a possibility. In this instance, Python was chosen as TArL's language on the Analysis Tier. Python was chosen 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 Sensing

The *Melexis* itself is its own computer (see Figure 1.3), containing EEPROM storage, RAM and unspecified code to perform "digital filtering" on the 16×4 array of digital active thermopiles. We are able to communicate with the *Melexis* through the provided I^2C interface, which offers commands to read both the EEPROM, and the sensor's RAM directly.

The sensor's EEPROM contains configuration values that the interfacing device is required to input into the device's RAM as part of a multi-step initialization sequence, and also contains constants used as part of the raw data to °C conversion process. The sensor's RAM contains the partially-filtered raw data, which is updated with reference to a clock frequency set between 0.5Hz to 512Hz in the initialization process.

The sensor's documentation offers no information regarding reconfiguration of the sensor's internal programming code, nor what code exists on the sensor when purchased. As such, we refer to the sensor's dataset [5] and use only the documented commands to interface with the sensor.

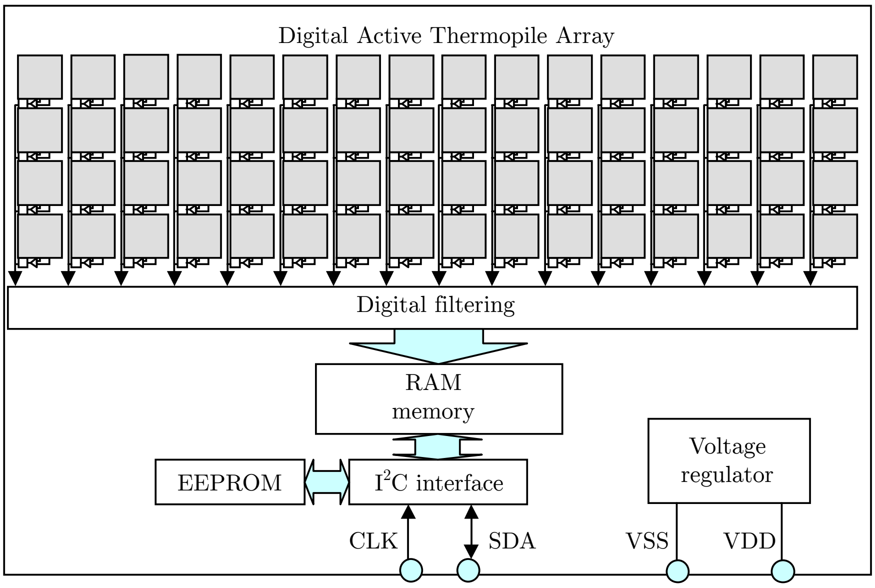

Figure 1.3: Block diagram for the *Melexis* taken from datasheet [5]

1.2.2 Pre-Processing

On the Arduino, the TArL C++ Driver is written as one monolithic Arduino program, termed `mlx90620_driver.ino`. This program's purpose is to take simple commands over serial to configure the *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 the on-board 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' original posts. To ensure the driver can be adapted and extended in the future, the code was also restructured and rewritten to be both more readable, 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.

```
INIT 0                        # Initialization sequence begins at time=0 milliseconds
INFO START                    # Information section starts
DRIVER MLX90620               # MLX90620 driver is being used
BUILD Feb  1 2015 00:00:00  # Driver was compiled at specified date
IRHZ 1                        # Infrared data is being sampled at 1Hz
INFO STOP                     # Information section ends
ACTIVE 33                     # Sensor is active at ready to send data at time=33 milliseconds

START 34                      # Thermal packet begins at time=34 milliseconds
MOVEMENT 0                    # No movement in this frame
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                       # Thermal packet ends at time=97 milliseconds
```

Figure 1.4: Annotated initialization sequence and thermal packet

The first of the features introduced was the human-readable format for serial transmission. This allows easy debugging of the sensor with only a rudimentary serial console. When the Arduino is powered up, an initialization sequence is output (Figure 1.4). 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 the sensor to configure operation. Depending on the input sensor configuration, IR data may be periodically output, or otherwise manually triggered. This IR data is produced in the packet format described in Figure 1.4.

1.2.3 Analysis / Classification

On the Analysis Tier, TArL's set of Python libraries and accompanying capture and analysis scripts were developed to interface with the Arduino, parse and interpret its data, and to provide data logging and visualization capabilities.

TArL's Python portion provides 4 main feature sets across 3 files; the `Manager` series of classes, the `Visualizer` class, the `Features` class and the `pxdisplay` module.

`Manager` classes

The Manager series of classes are the direct interface between TArL's C++ Arduino driver and TArL's Python components. 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 at the same rate. This serial interface is multi-threaded, as at higher serial baud rates if data is not polled continuously the internal serial buffer fills and data is 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 it uses a polling model instead of the periodic model of the other classes. Scripts may request thermal/motion data from the class at any interval, and `OnDemandManager` 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.

`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 green to blue for each temperature value that falls between those two extremes.

`Visualizer` class

The `Visualizer` class is the natural compliment to the `Manager` series of classes. The functions contained within can 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 simple, plain-text 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 and the `OnDemandManager` class to synchronously capture both visual and thermal data for the purposes of ground truth verification.

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.

`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 that pixel's average when there was no motion.

From these "active" pixels, it was established that a set of three feature vectors could be generated 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 joined with its immediate active neighbors, how many "islands" of active pixels (termed connected components in graph theory) exist.
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 was created to extract these features. Given the scope restriction to a minimum viable implementation, the portion of the ThermoSense code dealing with scaling the thermal background for rooms without motion was not implemented. For connected components determination, we leveraged the `networkx` graph library.

1.3 Summary

We believe that the hardware and software architecture presented here lays a solid foundation on which experimental data can be collected. The hardware architecture, as discussed, as been specifically selected to ensure that there is a transition path from the current USB Serial Pre-Processing/Analysis connection to one which does this wirelessly. The software library, TArL, has been written to be robust and general, so that its functionality is both useful in the current situation, and also for future experiments with this and other prototypes.

```python
# INITILISATION: Import libs, set up variables
import math, itertools, networkx

w, h        = 16, 4        # Get thermal image dimensions
wgt         = 0.01         # Weighting for exp. weighted moving avg.
fst_frame   = get_frame()  # 1st thermal frame, set elsewhere (2D array)
back        = fst_frame    # Thermal background b (2D array)
means       = fst_frame    # Per pixel x̄ (2D array)
pstds       = [[0]*w]*h    # Per pixel intermediate σ (2D array of 0)
stds        = [[0]*w]*h    # Per pixel complete σ (2D array of 0)
n           = 1            # Processed frames counter

# f: New frame received from sensor, starting at the 2nd frame (2D array)
# is_motion: If there has been motion detected over given time window.
def get_features(f, is_motion):
  n      += 1               # Increment frame counter
  active  = []              # Init empty active list
  g       = networkx.Graph() # Init graph structure

  # BACKGROUND UPDATE: Iterate over every pixel and update if no motion
  for i, j in itertools.product( range(w), range(h) ):
    # If no motion update b_{i,j}, x̄_{i,j}, & σ_{i,j}  with f_{i,j}
    if not is_motion:
      back[i][j]   = wgt * f[i][j] + (1 - wgt) * back[i][j]      # b_{i,j}
      means[i][j]  = means[i][j] + (f[i][j] - means[i][j]) / n  # x̄_{i,j}
      pstds[i][j]  = pstds[i][j] + (new[i][j] - means[i][j])
                   * (c - means[i][j])
      stds[i][j]   = math.sqrt(pstds[i][j] / (n-1))             # σ_{i,j}

    # GRAPH GENERATION: If (f_{i,j} − b_{i,j}) > 3σ_{i,j} add pixel to active & graph
    if (f[i][j] - back[i][j]) > (3 * stds[i][j]):
      active.append((i,j))

      # Link all adjacent active pixels in graph structure
      for ix, jx in [(-1, -1), (-1, 0), (-1, 1), (0, -1)]:
        g.add_edge((i,j), (i+ix,j+jx)) if (i+ix, j+jx) in active

  # CONNECTED COMPONENTS: Get connected comps. from graph & gen features
  cons           = list( networkx.connected_components(g) )
  num_active     = len(active)
  num_connected  = len(cons)
  size_connected = max(len(c) for c in cons) if len(cons) > 0 else None

  return (num_active, num_connected, size_connected)
```

Listing 1.1: Core feature extraction code

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.