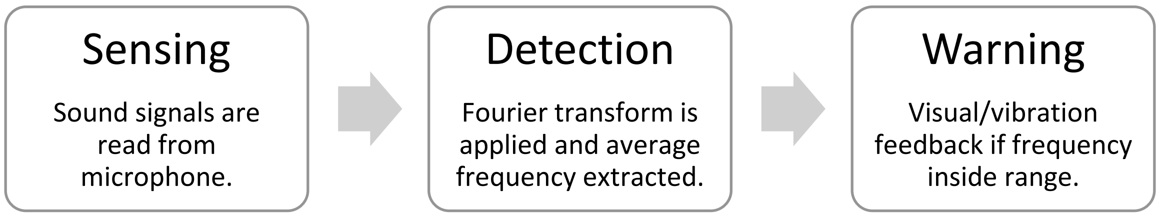
Approaching Traffic Haptic Information System (ATHIS)

Technical Documentation

ATHIS is a helmet/backpack mounted warning system for cyclists to signal approaching traffic. It uses an audio signals to detect approaching objects and relays detections via vibrations to the wearer. The system is meant for use by cyclist and pedestrian traffic. This document outlines the features of the prototype, issues with implementation, and future improvements.

1. Overview

ATHIS is implemented on the BeagleBone Black kit. It is comprised of three stages.


Sensing
Sound signals are read from microphone.
Detection
Fourier transform is applied and average frequency extracted.
Warning
Visual/vibration feedback if frequency inside range.

For the first stage, sound amplitudes are sampled from a single microphone at a prescribed rate. Samples are grouped into blocks. When one block is sampled, a Fourier transform is applied to decompose the signal into constituent frequencies. The average frequency of each block is stored for a certain time. If the average frequencies show a change over time which is greater than a set threshold, a warning is issued as output to signal an incoming vehicle.

2. Theory of Operation

ATHIS operates on the principle of doppler effect. The doppler effect is the change in frequency of a source moving relative to an observer:

$$\Delta f = \frac{\Delta v}{c} f_0$$

Where:

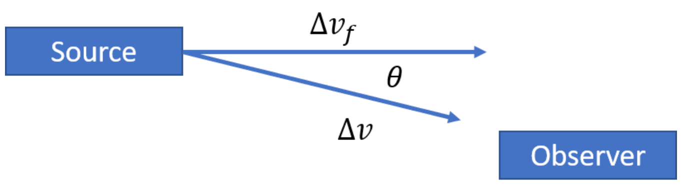
Δv is the relative radial velocity of the source and the observer. It is positive when the distance is decreasing. If the relative forward velocity is given by Δv_f, the angle between the forward direction and line of sight between source and observer is given by θ, then the radial velocity is $\Delta v = \Delta v_f \cdot \cos(\theta)$.

c is the speed of sound in air.

f_0 is the stationary frequency of the source.

At large distances between source and observer, θ is close to zero. However, as the source approaches the observer in parallel, θ approaches $\pi/2$. Assuming constant Δv_f, Δv approaches 0. This means that the observed frequency of the source decreases as it approaches at a constant speed.

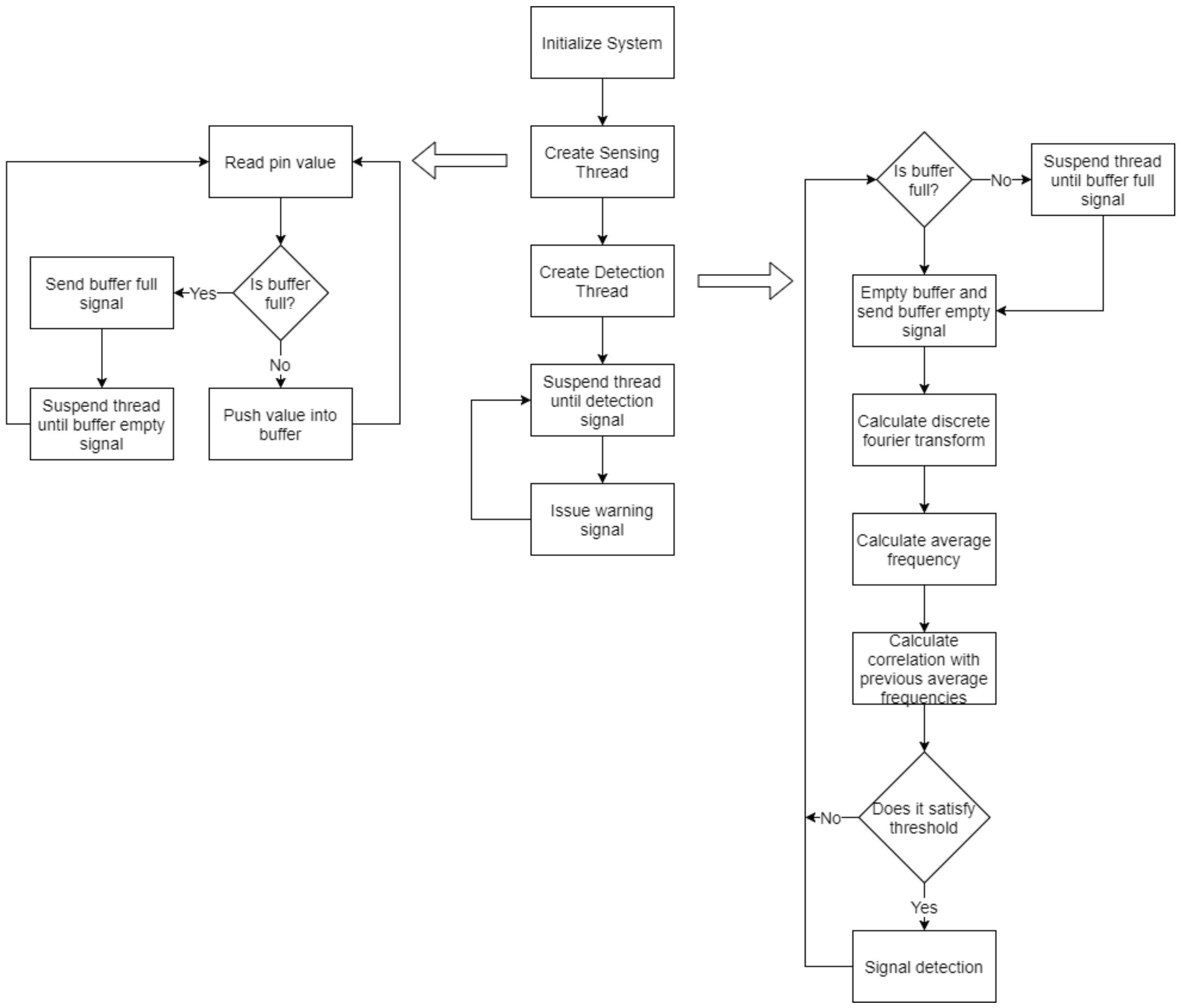
Note that for head-on approach, as in case of same-lane vehicular traffic, there will be no change in frequency since the angle of approach is 0.



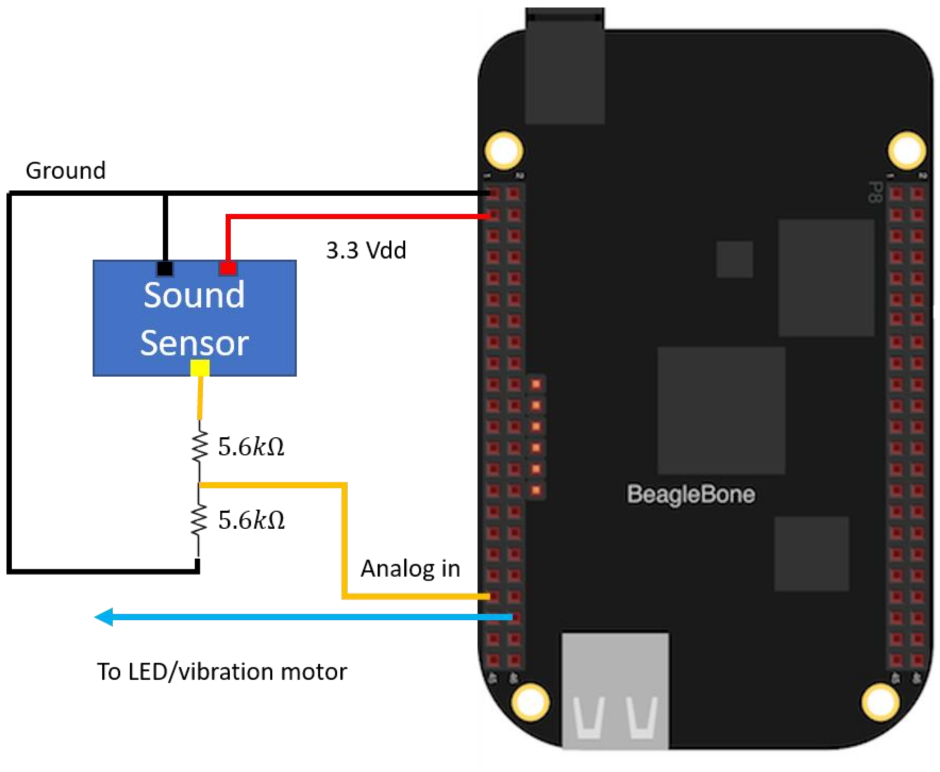
The change in frequency over time can then be used as a measure of distance between approaching vehicles and the observer.

3. Implementation

The system was implemented in C on a Linux operating system (kernel version 4) hosted on a BeagleBone device. The flowchart of system logic is shown below:


Initialize System
Create Sensing Thread
Create Detection Thread
Suspend thread until detection signal
Issue warning signal
Read pin value
Is buffer full?
Yes
No
Send buffer full signal
Suspend thread until buffer empty signal
Push value into buffer
Is buffer full?
No
Suspend thread until buffer full signal
Empty buffer and send buffer empty signal
Calculate discrete fourier transform
Calculate average frequency
Calculate correlation with previous average frequencies
Does it satisfy threshold
No
Yes
Signal detection

The circuit schematic of the system is as follows:



3.1 Sensing

Sensing is done by sampling sound sensor voltage levels periodically using analog inputs. By default, the board does not enable analog inputs. At every operation the following command must be entered. It modifies the beaglebone's device tree overlay to enable analog inputs:

```
echo BB-ADC > /sys/devices/platform/bone_capemgr/slots
```

The operating system stores the voltage values inside files for each analog pin. The sensing thread periodically opens, reads, and closes the file associated with the pin. File paths for each pin are included in `config.h` in the source code.

After reading the pin value, the sensing thread pushes it into a buffer. The buffer is an integer array protected by a condition variable shared between sensing and detection threads.

Once the data has been sent, the sensing thread sleeps for the specified period (see `config.h`) and repeats the process.

3.2 Detection

The detection thread runs concurrently with the sensing thread. It uses the shared condition variable to wait until the buffer is full and consumes it all at once. This ensures that the sensing thread does not sleep for long intervals.

Using voltage levels consumed from the buffer, the thread computes their Fourier transform. The transform is naively implemented on the CPI using floating point arithmetic and calls to math library functions. It runs with a complexity of $O(n^2)$ where n is the number of samples.

The transform returns weights associated with each constituent frequency of the signal. An average of the frequencies using the weights is calculates to give a representative estimate of the sound sample at that instant.

The detection thread maintains a fixed trailing record of prior average frequencies. At each update, it computes the Pearson correlation coefficient of the frequencies as a time series. A coefficient of 1 means the frequencies are reliably increasing with time. A coefficient of -1 indicates a decrease. Values close to 0 indicate signal noise or no change.

The correlation coefficient is compared against a threshold value provided at compile time (see `config.h).` If the comparison is positive, a detection is signaled to the main thread using a condition variable.

3.3 Warning

Warning and feedback logic is implemented in the main thread after secondary threads are spawned. The main thread suspends itself until signaled by the detection thread. Once active, it sends a logic high to the output pin connected to a vibration motor or LED. Feedback is turned off after an interval. Finally, the thread suspends itself again to await another detection. The `libsoc` library is used for digital pin signaling.

4. Issues with Implementation

The system as it stands now is not functional. The rate of false positives is high. A possible list of contributing factors is documented in this section.

4.1 Weak theoretical correlation

The theoretical basis of this project makes several assumptions. First, the relative forward velocity is assumed constant. Second, ambient noise is assumed insignificant. Third, physical properties of signals like rest frequency and speed of sound in air are abstracted away when relative change is measured. The detection algorithm relies on a monotonic change in frequency to positively identify an approaching vehicle. However, the discrepancy between actual readings and system thresholds based on assumptions make detection of approach unreliable.

4.2 Non-uniform sampling rates

The non-uniformity in sampling rates steps from two sources: unreliable task scheduling by the host operating system and the computationally intensive nature of the algorithm.

The sensing thread uses the standard `time` library to schedule periodic pin readings by putting the sensing thread to sleep between measurements. However, there is an inherent time error when the thread wakes up again. This means that audio samples are not uniformly sampled which affects their Fourier transform.

Additionally, all pin measurements are done on the CPU. The sensing thread has to use the operating system's file input/output interface to read analog pins. This puts additional overhead during sampling which may change the rate at which pins are read.

Finally, the Fourier transform is a computationally intensive operation. Given that all three threads are running concurrently on a single processor at equal priority with shared resources, the detection thread becomes a bottleneck at high sampling rates. This delays task execution in other threads which has a compounding effect on the following readings.

5. Future work and improvements

5.1 Robust sensing

Currently samples are not preprocessed before their Fourier transform. This allows for noise to obfuscate the desired measurements. An improvement would be to connect the microphone to a lowpass filter. This will suppress high frequency noise which will reduce aliasing effects by guaranteeing that the sampling rate is faster than the Nyquist frequency for the input signals.

Multiple analog pins may also be used to provide redundancies and fault-tolerance. Sound sensors placed in different orientations and connected to different input interfaces may allow for detection and isolation of environmental and system faults. For example, if a connection to one analog pin is getting noise from debouncing, it can be diagnosed via comparison to a parallel set of readings.

5.2 Performance optimization

Optimizing performance may alleviate the problem of non-uniform sampling rates. In particular, using the Fast-Fourier Transform (FFT) using fixed point arithmetic and with look-up tables for mathematical functions will speed up calculations. The FFT provides an optimization of $O(\log n)$ over the "textbook" version of the transform. This means a 3.5x speedup to start with for a sample size of 32.

Analog input operations can be offloaded to the Programmable Real-time Unit (PRU) on the BeagleBone. The PRU can execute a more limited instruction set but with guaranteed run times. This means that faster and uniform sample rates can be achieved.

Appendix – Source Code

config.h

```
#ifndef INC_CONFIG_H_
#define INC_CONFIG_H_

#define AINROOT "/sys/bus/iio/devices/iio:device0/"
#define AIN0 AINROOT "in_voltage0_raw"
#define AIN1 AINROOT "in_voltage1_raw"
#define AIN2 AINROOT "in_voltage2_raw"
#define AIN3 AINROOT "in_voltage3_raw"
#define AIN4 AINROOT "in_voltage4_raw"
#define AIN5 AINROOT "in_voltage5_raw"
#define AIN6 AINROOT "in_voltage6_raw"

#define NUMAIN 7                              // number of analog input pins
extern char *pins[NUMAIN];            // declares a global array of pin
addresses defined in io.c

#define OUTPIN 7                              // pin number of the digital
output pin used for detection (P9.42)
#define INPIN 0                               // index of analog pin from pins
array defined in io.c

#ifdef TEST

#define PERIOD 100000000                      // duration (ns) between
consecutive samplings (for TESTING)
#define THRESH 0.1                            // correlation threshold at
which to signal detection (for TESTING)
#define NSAMPLES 32                           // number of pin readings per
sample
#define TAIL 5                                // number of prior average
readings to store
#define ALERTDURATION 1000000000      // duration (ns) of detection alert

#else

#define PERIOD 1000000                // duration (ns) between  consecutive
samplings (for USE)
#define THRESH 0.1                            // correlation threshold at
which to signal detection (for USE)
#define NSAMPLES 20                           // number of pin readings per
sample
#define TAIL 10                               // number of prior average
readings to store
#define ALERTDURATION 1000000000      // duration (ns) of detection alert

#endif

#define BILLION 1000000000

#endif /* INC_CONFIG_H_ */
```

buffer.h

```
#ifndef INC_BUFFER_H_
#define INC_BUFFER_H_

#include <pthread.h>
#include <config.h>

typedef struct Buffer {
        int *q;                                 // array of integer values
        int r;                                  // index of Element at front of
queue
        int w;                                  // index where new Element is
written
        int N;                                  // size of queue
        int n;                                  // number of Elements enqueued
        pthread_mutex_t lock;
        pthread_cond_t cond;
} Buffer;

void buffer_init(Buffer *b, int N);

void buffer_destroy(Buffer *b);

void enqueue(Buffer *b, int value);

void dequeue(Buffer *b, int *values);

#endif /* INC_BUFFER_H_ */
```

detect.h

```
#ifndef INC_DETECT_H_
#define INC_DETECT_H_

#include <io.h>

#define PI 3.14159265358979323846

void init_detector(PERIODIC *p);
void _detector(void *arg);

#endif /* INC_DETECT_H_ */
```

io.h

```
#ifndef INC_IO_H_
#define INC_IO_H_


#include <pthread.h>
#include <buffer.h>

typedef int PIN;

typedef struct {
       pthread_mutex_t lock;                  // lock that protects the
conditional variable
       pthread_cond_t cond;                   // conditional variable for
detection flag
       int detection;                                 // flag indicating
detection
       pthread_t th_pread, th_detect; // handles for pin reading and
detection threads
       struct timespec ts;                            // contains pin sampling
rate information
       PIN pin;                                               // handle to the
pin being read
       Buffer buff;                                   // Buffer object
containing read samples passed b/w read/detect
} PERIODIC;


int read_pin(PIN pin);

void init_periodic_read(PERIODIC *p);

void *_reader(void *args);


#endif /* INC_IO_H_ */
```

buffer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <buffer.h>


void buffer_init(Buffer *b, int N) {
        b->q = malloc(N * sizeof(int));
        b->N = N;
        b->n = 0;
        b->w = 0;
        b->r = 0;
        pthread_mutex_init(&b->lock, NULL);
        pthread_cond_init(&b->cond, NULL);
}

void buffer_destroy(Buffer *b) {
        free(b->q);
        pthread_cond_destroy(&b->cond);
        pthread_mutex_destroy(&b->lock);
}

void enqueue(Buffer *b, int value) {
        pthread_mutex_lock(&b->lock);
        while (b->n >= b->N) {
                pthread_cond_wait(&b->cond, &b->lock);
        }
        b->q[b->w] = value;
        b->w = (b->w+1) % b->N;
        b->n++;
        pthread_cond_broadcast(&b->cond);
        pthread_mutex_unlock(&b->lock);
}

void dequeue(Buffer *b, int *values) {
        pthread_mutex_lock(&b->lock);
        while (b->n < b->N) {
                pthread_cond_wait(&b->cond, &b->lock);
        }
        int i;
        for (i=0; i<b->N; i++) {
                values[i] = b->q[i];
        }
        b->r = 0;
        b->n = 0;
        pthread_cond_broadcast(&b->cond);
        pthread_mutex_unlock(&b->lock);
}
```

detect.c

```
#include <math.h>
#include <config.h>
#include <detect.h>
#include <stdio.h>
#include <pthread.h>


void init_detector(PERIODIC *p) {
	pthread_create(&p->th_detect, NULL, _detector, p);
}

void _detector(void *arg) {
#ifdef TEST
	printf("Detector thread running...\n");
#endif

	PERIODIC *p = (PERIODIC *) arg;
	pthread_mutex_init(&p->lock, NULL);
	pthread_cond_init(&p->cond, NULL);
	p->detection = 0;

	int vals[NSAMPLES] = {0};             // stores a set of sampled pin
readings
	double totals[TAIL][2] = {0};  // stores summed real and imaginary
components of fourier transform
	double means[TAIL];                           // stores mean fourier
magnitudes for sets of samples
	double meanofmeans = 0;                       // the average of the
mean frequencies of sample sets;
	int tailmean = TAIL *  (TAIL + 1) / 2;// average of 1,2,3...,TAIL
	double c, s, a, smt, sm, st;   // cos, sin, argument, covariance, std.
dev, std. dev
	double dm, dt, corr;                  // mean - meanofmeans, 1,2,3...
- TAIL, correlation
	int curr = 0;                                 // current index of
means buffer
	int i, j;                                             // indices

	while (1) {
		dequeue(&p->buff, vals);

		// take fourier transform. This is O(n^2).
		for (i=0; i<NSAMPLES; i++){
			totals[i][0] = 0;
			totals[i][1] = 0;
			for (j=0; j<NSAMPLES; j++) {
				a = 2*PI*i*j/NSAMPLES;
				c = cos(a);
				s = sin(a);
				totals[i][0] += vals[j] * c;
				totals[i][1] += vals[j] * s;
			}
		}
		// store average frequency for current samples
		curr = curr % TAIL;
```

```
                meanofmeans -= means[curr] / TAIL;
                means[curr] = 0;
                for (j=0; j<NSAMPLES; j++) {
                        means[curr] += j * totals[j][0] * totals[j][0] +
totals[j][1] * totals[j][1];
                }
                means[curr] /= NSAMPLES;
                meanofmeans += means[curr] / TAIL;
                ++curr;

                // calculate correlation among means of current and prior
sample sets
                smt = 0; sm=0; st=0;
                for (j=0; j<TAIL; j++) {
                        dt = (j - tailmean);
                        dm = (means[j] - meanofmeans);
                        smt += dt * dm;
                        st += dt * dt;
                        sm += dm * dm;
                }
                corr = smt / sqrt(st * sm);

                // signal on detection
                if (corr >= THRESH) {
#ifdef TEST
                        printf("Corr: %10.3f\n", corr);
#endif
                        pthread_mutex_lock(&p->lock);
                        p->detection = 1;
                        pthread_cond_broadcast(&p->cond);
                        pthread_mutex_unlock(&p->lock);
                }
        }
}
```

io.c

```
#include <stdio.h>
#include <time.h>
#include <config.h>
#include <io.h>
#include <buffer.h>

char *pins[NUMAIN] = { AIN0,   // P9.39
                                              AIN1,   // P9.40
                                              AIN2,   // P9.37
                                              AIN3,   // P9.38
                                              AIN4,   // P9.33
                                              AIN5,   // P9.36
                                              AIN6    // P9.35
                                       };


int read_pin(PIN pin) {
        int val;
        int res;
        FILE *f = fopen(pins[pin], "r");
        if (f == NULL) return -1;
        res = fscanf(f, "%d", &val);
        fclose(f);
        if (res == 0) return -1;
        return val;
}

void init_periodic_read(PERIODIC *p) {
        pthread_create(&p->th_pread, NULL, _reader, p);
}

void *_reader(void *args) {
#ifdef TEST
        printf("Reader thread running...\n");
#endif
        PERIODIC *p = (PERIODIC *) args;
        int level;
        int i;
        while (1) {
                level = read_pin(p->pin);
                enqueue(&p->buff, level);
                nanosleep(&p->ts, NULL);
                ++i;
        }
        return NULL;
}
```

main.c

```
/*
 * main.c
 *
 *  Created on: Nov 5, 2017
 *      Author: Ibrahim Ahmed
 */


#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <config.h>
#include <io.h>
#include <buffer.h>
#include <detect.h>
#include <libsoc_gpio.h>

int main(void) {

#ifdef TEST
		printf("DEBUG\n");
#else
		printf("RELEASE\n");
#endif
	// acquire output pin for signalling
	gpio *outpin = libsoc_gpio_request(OUTPIN, LS_GPIO_SHARED);
	if (outpin == NULL) {
		printf("Could not acquire output pin.\n");
		exit(-1);
	}
	libsoc_gpio_set_direction(outpin, OUTPUT);
	if (libsoc_gpio_get_direction(outpin) != OUTPUT)
	{
		printf("Failed to set pin direction to output.\n");
		exit(-1);
	}
	struct timespec ts_alert = {ALERTDURATION / BILLION, ALERTDURATION %
BILLION};

#ifdef TEST
	printf("Testing analog pin access: \n");
	int i;
	for (i=0; i<NUMAIN; i++) {
		printf("AIN%1d:\t%4d\n", i, read_pin(i));
	}

	printf("\nTesting periodic pin read: \n");
#endif

	PERIODIC p;
	p.pin = INPIN;
	p.ts.tv_sec = (int) (PERIOD / BILLION);
	p.ts.tv_nsec = PERIOD % BILLION;
	buffer_init(&p.buff, NSAMPLES);
```

```
        // start periodic pin sampling thread
        init_periodic_read(&p);

#ifdef TEST
        int vals[NSAMPLES];
        int j = 0;
        for (j=0; j<10; j++) {
                dequeue(&p.buff, vals);
                printf("Periodic sample %d of 10: ", j+1);
                for (i=0; i<NSAMPLES; i++) {
                        printf("%d ", vals[i]);
                }
                printf("\n");
        }
        printf("\nTesting detection:\n");
#endif

        // start detection algorithm thread
        init_detector(&p);

        // start waiting on detection
        while (1) {
                pthread_mutex_lock(&p.lock);
                while (p.detection == 0) {
                        pthread_cond_wait(&p.cond, &p.lock);
                }
                p.detection = 0;
                libsoc_gpio_set_level(outpin, HIGH);          // set outpin
high on detection
#ifdef TEST
                printf("\nDETECTION!\n");
#endif
                pthread_mutex_unlock(&p.lock);
                nanosleep(&ts_alert, NULL);
        // outpin remains high for ALERTDURATION
                libsoc_gpio_set_level(outpin, LOW);                   // reset
outpin to low
        }

        pthread_join(p.th_pread, NULL);
        pthread_join(p.th_detect, NULL);

        return 0;
}
```