CIS 721

RTOS, Raspberry Pi, and Sensor Interfacing

Dan Wagner, Hunter Goddard
Kansas State University
College of Engineering
Department of Computer Science

Dr. Mitchell Neilsen
Professor
Department of Computer Science

May 8 2018

1 Introduction

A Raspberry Pi is a modular, advanced computing device that can be used in many projects. It comes with 40 General Purpose Input/Output (GPIO) pins which can be attached to sensors, motors, and other similar devices (Foundation, 2018). By default, the Pi ships with the Raspian OS, which is a flavour of Debian Linux. This OS is not a Real-Time OS (RTOS) but can be loaded with one (or a microkernel) on top of the existing system. Once loaded with such software, code can be developed that allows for real-time constraints and modules. In this project, the Xenomai 3 RTOS was installed on top of Raspbian to incorporate these characteristics.

Two sensors were attached to the system for real-time reporting: a DHT11 humidity/temperature sensor and a rotary encoder module. Using the Xenomai 3 API, tasks were scheduled to periodically poll the sensors within specified deadlines. A static, cyclic scheduling algorithm was used to coordinate all the sensor readings. Xenomai handled the scheduling with predetermined priorities for each task.

The sections to follow will outline the project details. First, the Xenomai 3 RTOS will be discussed, particularly outlining the functionality that is incorporated into the sensor software. Second, the hardware specifications will be listed and discussed for the Raspberry Pi, rotary encoder, and DHT11. Third, the software will be described in detail along with any design decisions. Then, the project's experimental results and potential real-world applications will be discussed and analyzed. Finally, conclusions will be drawn from the previous discussions.

2 Xenomai 3

Xenomai 3 is an RTOS developed by numerous engineers as an Free Software project for Linux machines (Xenomai, 2018). It allowed for a versatile framework to be developed for real-time capabilities in Linux. The RTOS provides an in-depth, detailed API for programmers to utilize when managing tasks. In particular, several routines were used to create, initialize, schedule, and delete tasks.

Xenomai was responsible for creating the sensor reading tasks and scheduling them in a static, cyclic manner. Both sensor tasks were scheduled with their respective periods. Each was given a priority, and Xenomai scheduled them accordingly. In the static, cyclic scheduling system, the priorities were statically decided and the schedule is known a priori. This allowed Xenomai to determine the explicit schedule and have it planned, which reduced any overhead in the software.

3 Hardware Specifications

The Raspberry Pi 3B, shown below, was used in this project; its relevant specifications are listed in the table below. Its processor and RAM were more than capable of the project's requirements, since the sensors were polled and there were only two of them. Numerous input/output pins allowed multiple sensors to be wired up.

Figure 1: Raspberry Pi 3B.

Table 1: Raspberry Pi Specs

Processor	Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
RAM	1GB
Data Pins	40 GPIO
Operating System	Raspian
RTOS	Xenomai 3

Both sensors were distributed in a kit by Sunfounder. The rotary encoder has a resolution 20 cycles per revolution, which means that twenty electrically high pulses will be present on the square wave per revolution (ArtOfCircuits, 2018). When the encoder is turned to the left, a "negative" turn is recorded; negative refers to a reverse turn in the software, and is represented as a decrement on the counter. If it is turned to the right, then a positive turn is recorded.

The DHT11 recorded both humidity and temperature characteristics. Both readings have a 16-bit resolution. The sampling period is more than two seconds, and was the determining factor in the scheduling of tasks in Xenomai. The data is retrieved after a brief set of delays that allow the bits to propagate into memory. Once the data is in memory, its checksum is validated; bad data is discarded and results are defaulted to the previous reading while valid data is reported to the system, and the previous reading is updated to the fresh information.

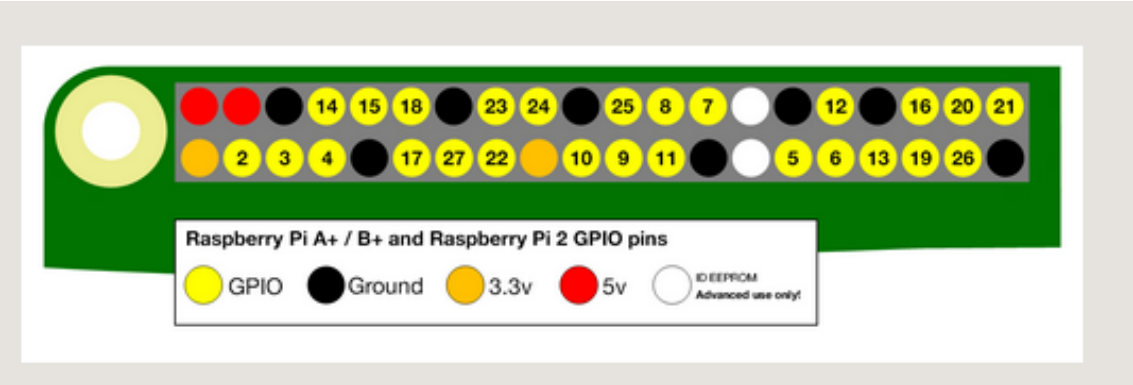

Figure 2: Pi 3 GPIO Pinouts. Courtesy of Raspberrypi.org.

Above is an image that labels each numbered pin of the Raspberry Pi 3. The rotary encoder required five pins: 5V, GND, Clock, Data, and Switch. Pins 5V and GND were attached to the top row, second and third pins (red and black). Clock was wired to GPIO 16, Data to GPIO 27, and Switch to GPIO 22. These pins were denoted by the hardware; as it will be discussed, in software, the pins were referred to by a different numbering mechanism.

The figure below shows the system's configuration. Three pins were used for the DHT11: 5V, GND, and Data. The 5V and GND signals were wired to the same pins as the rotary encoder through the use of a breadboard. Data was wired to GPIO 23. The following image shows the final setup of the system.

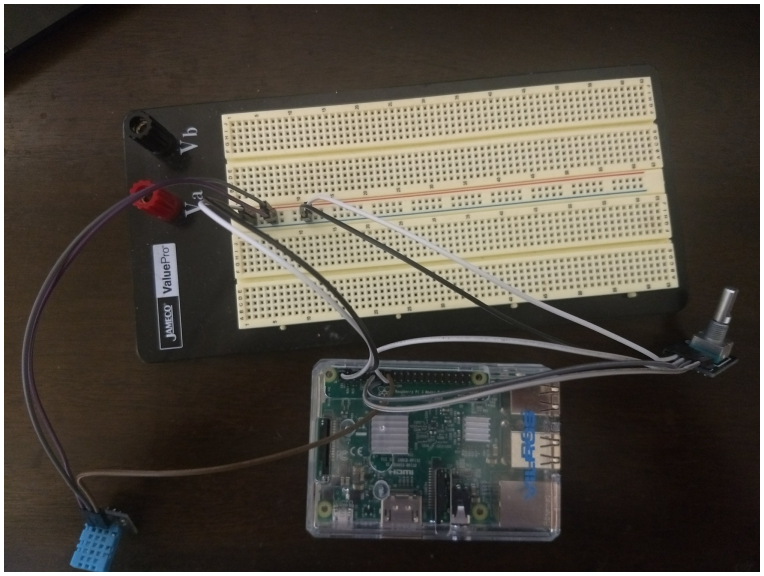

Figure 3: Sensors and Pi Wiring.

A third sensor, an infrared receiver (shown below), was considered for this project. It would have an ISR similar to the encoder and would be polled periodically for any new infrared detections. It had an inherent refresh delay of 600 ms (Sunfounder, 2018). This delay proved to be too large when scheduling the other two sensors, and caused many missed deadlines. As a result, no IR sensor was used in the project.

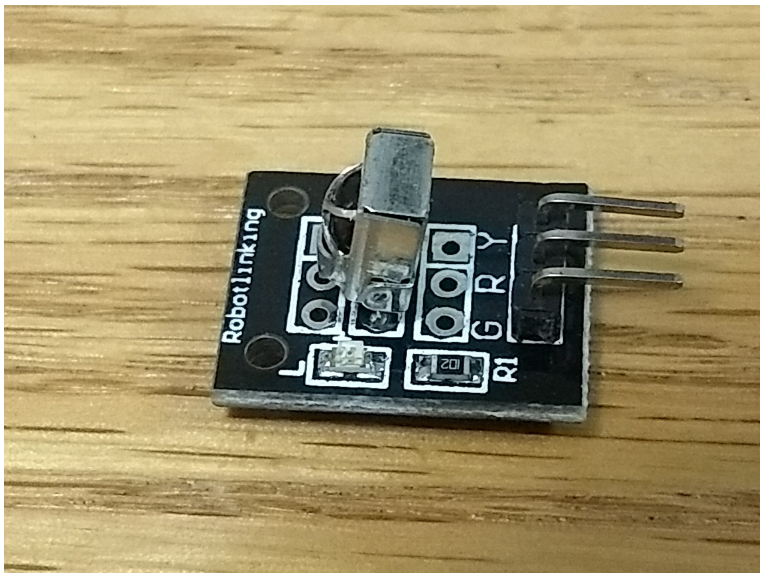

Figure 4: Infrared Receiver.

4 Software Specifications

The system's real-time constraints were considered before any software was developed. Sunfounder, the sensors' distributor, provided several code examples on their Github account to test each sensor (Sunfounder, 2018). These constraints were derived from examining their source code.

The DHT11 (pictured below) experienced high amounts of latency, as each reading required one second to be valid. First, the sensor's pin was pulled low for 18ms. Then, it was pulled high for 40us in preparation to read the data. After a slight delay from verifying the data's integrity via checksum, it was read, after which the next reading was not available for a tenth of a second. Any reduction in this delay caused a reading to be invalid, or missed. As a result, failures of the DHT11 were defined as not receiving data from the sensor after one tenth of a second.

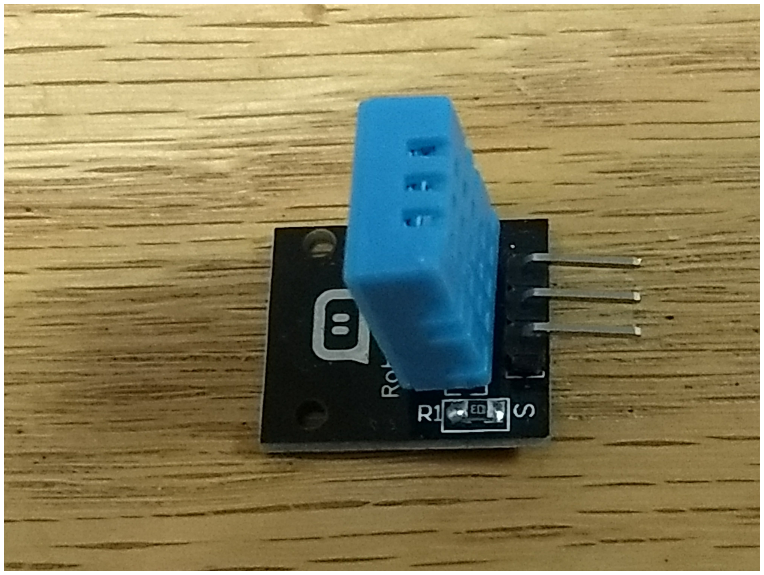

Figure 5: DHT11 Sensor.

For the rotary encoder (pictured below), the latency was very short. From Sunfounder's source code, there was no delay explicitly defined. After experimenting with the sensor, it was possible to retrieve information every 5ms without issue. Thus, any failure of the encoder was defined as not recording a reading within five milliseconds.

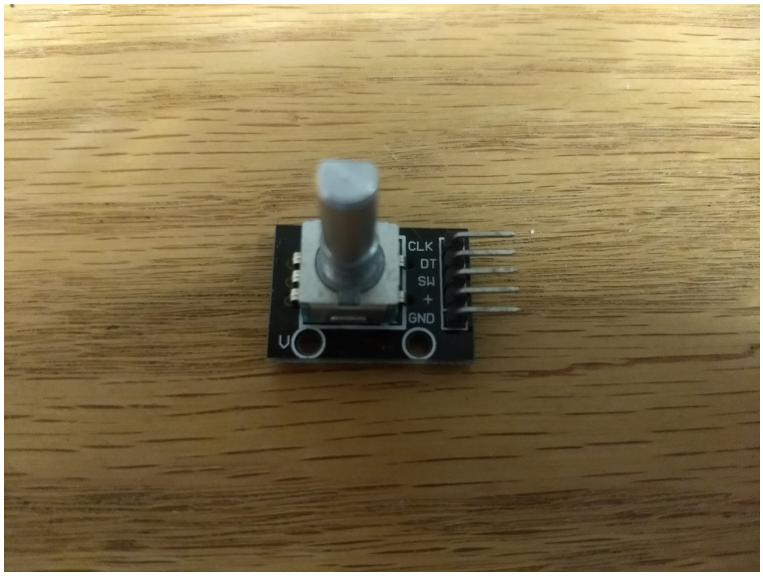

Figure 6: Rotary Encoder.

These real-time constraints were modeled in UPPAAL for verification. Below is a visual representation of this model.

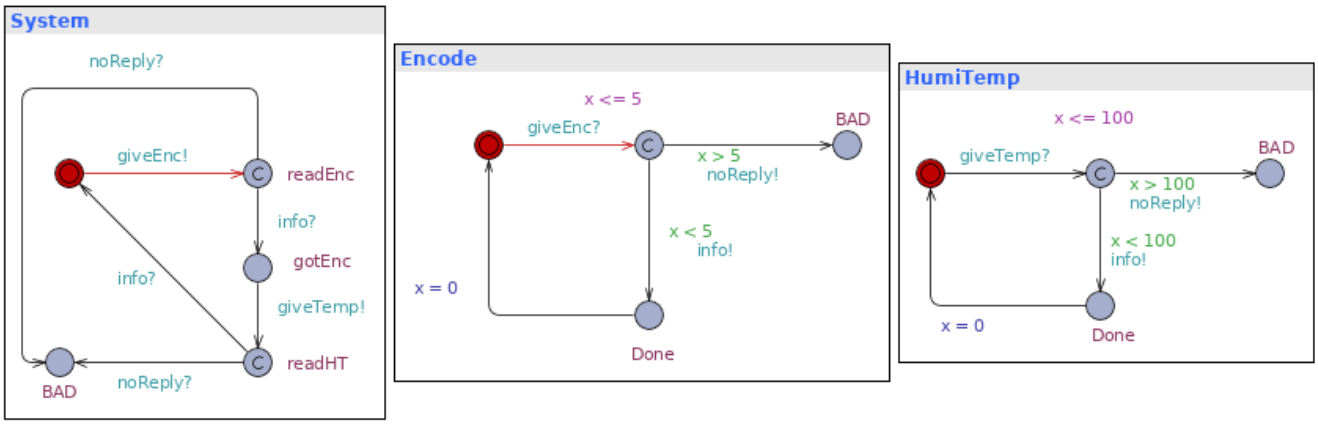

Figure 7: UPPAAL Model of the System.

One process represented the Raspberry Pi, with the encoder and DHT11 being additional processes. First, the Raspberry Pi would request the encoder's data by signaling it along the giveEnc channel. Upon receipt of this message, the encoder would enter an intermediate state with invariant of $x <= 5$; this modeled the latency in its operation. If a fault occurred (i.e. $x > 5$), then a message on the noReply channel would be sent, the bad state would be entered and the Raspberry Pi would experience a failure. Otherwise, the encoder sent data back to the Pi (via the info channel) and proceeded to a done state before resetting its local clock for the next request. After receiving the encoder data, the Pi would ask the DHT11

for data by sending a message on the giveTemp channel. The DHT11 functioned similarly to the encoder, but with a 1 second latency (100 time units). If results were not available within one second, the sensor would report a fault on the noReply channel and enter a bad state. Otherwise, it would send a reply on the info channel, enter a done state, and reset its local clock before awaiting the next poll from the Raspberry Pi.

Three queries were used to verify these characteristics. First, $A[\,]$ *not Encode.BAD* would test for a failure with the encoder. Next, $A[\,]$ *not HumiTemp.BAD* would test for DHT11 failures. Finally, $A[\,]$ *not System.BAD* tested for a system failure, with both sensors failing within the same reading cycle. UPPAAL's verifier satisfied the final condition, and found the first two queries unsatisfied, which translates to a successful, fault-tolerant system.

P1: The Main GPIO connector

WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		3.3v	1	2	5v		
8	Rv1:0 - Rv2:2	SDA	3	4	5v		
9	Rv1:1 - Rv2:3	SCL	5	6	0v		
7	4	GPIO7	7	8	TxD	14	15
		0v	9	10	RxD	15	16
0	17	GPIO0	11	12	GPIO1	18	1
2	Rv1:21 - Rv2:27	GPIO2	13	14	0v		
3	22	GPIO3	15	16	GPIO4	23	4
		3.3v	17	18	GPIO5	24	5
12	10	MOSI	19	20	0v		
13	9	MISO	21	22	GPIO6	25	6
14	11	SCLK	23	24	CE0	8	10
		0v	25	26	CE1	7	11
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

Figure 8: WiringPi Pinouts. Courtesy of Wiringpi.com.

Once hardware connections were established and real-time constraints were verified, the software needed to interface with each device. This was made possible using the WiringPi library (Henderson, 2018). The GPIO designations in Figure 2 were not the same as WiringPi used. Figure 8 shows these obvious differences. The software refers to each pin by its WiringPi number. For the rotary encoder, the Clock, Data, and Switch pins were referred to as pins 1, 2, and 3 respectively. The DHT11's Data pin was denoted as pin 4. WiringPi used these numbers to interface with the Raspberry Pi through wiringPiSetup(). This function initialized the system to use its numbering scheme and allowed the data to be accessed on those pins. Additionally, wiringPiISR() was used to initialize the encoder's ISR, which was not run until the task was released for running. A summary of the wiring configuration is listed below.

Xenomai's API was utilized to create and initialize the tasks. First, rt_task_create() was called to make each sensor reading a task: each was given a task object, the sensor name, a stack size, priority, and mode (Xenomai, 2018). The stack size and mode were set to their defaults (0, 0), and the DHT11 was given a higher priority than the encoder due to its significantly higher delay time; since the encoder had less complex data than the DHT11,

Table 2: Summary of Hardware Pin Configuration

GPIO #	WiringPi #	Sensor Pin
2	-	5V
6	-	GND
18	1	Encoder CLK
27	2	Encoder DATA
22	3	Encoder SW
23	4	DHT11 Data

it was deemed not as important. After task creation, each task was started by rt_task_start(), which was given the task, its function starting point, and any arguments (Xenomai, 2018). The arguments were the default setting of 0; DHT11's starting point was the read_temp() function, and the encoder started within its ISR, read_encoder().

Within the starting functions, two additional Xenomai routines were used to ensure periodic scheduling: rt_task_set_periodic() and rt_task_wait_period(). First, the task was set to be periodic with a given period starting at sensor task's invocation, defined as the constant TM_NOW (Xenomai, 2018). These periods were defined in nanoseconds, as the period must be converted to ticks via rt_timer_ns2ticks() (Xenomai, 2018). As a result, the encoder's period was 5×10^6 ns and the DHT11's was 1×10^8 ns. After making each task periodic, its code segment would execute and report the sensor readings to console. Following this, the sensors would be put to sleep (by using rt_task_wait_period()) until their next scheduled time slot, which is when Xenomai would schedule the re-execution of this cycle.

5 Experimental Results

Scheduling tasks via the Xenomai routines was simple and effective. Every tenth of a second, the DHT11 would output the current temperature and humidity measurements. Meanwhile, the encoder's ISR would be run every 5ms without failure and report changes it detected. If the DHT11's period was reduced below one tenth of a second, then the scheduler would fail to meet the deadlines and report a communication error. Similarly, the encoder would throw the error if reduced below a 5ms period.

6 Applications

The versatility of the Xenomai 3 API combined with the large number of GPIO pins on the Raspberry Pi provides systems such as this with a great deal of extensibility. New sensors can be added to augment existing functionality while keeping the device compact and portable.

A common use of the humidity and temperature sensors present in this system is greenhouse environment monitoring. Greenhouses require maintaining a specific set of atmospheric conditions for the plants they contain to thrive, as the plants grown in one are often native to foreign climates. Adding a second rotary encoder would allow a user to easily set the acceptable range for both temperature and humidity; additionally, the inclusion of a UV sensor would allow the monitoring of sunlight exposure, activating electric UV bulbs to compensate for plants that require more sunlight than is available in the region. The size and relatively low cost of the system would also make it reasonable to include one in each of an array of greenhouses, each tuned to various atmospheric conditions.

This system could also be extended to be applied in a safety-critical context. In hospitals, various air quality sensors (such as particulate concentration and bacterial presence) could be added to the existing temperature and humidity sensors to keep patients safe and comfortable by producing alerts when conditions go beyond acceptable levels. Systems such as this could be placed in infectious disease wards, operating rooms, and maternity wards to continuously monitor the air quality for abnormalities.

An application of this system that takes advantage of its portability is for hiking and mountain climbing. Temperature and humidity are important information for wilderness explorers to avoid overheating or dehydrating. Rapid changes in these measurements can also act as indicators for inclement weather. Additionally, extending the system with altitude and air pressure sensors would provide much-needed information to keep climbers safe and aware. Since the system is small and light, it could be strapped to a backpack without significantly affecting the strict weight limits that hikers have for their gear.

7 Conclusion

The Xenomai 3 API proved to be very simple and powerful. The small number of functions necessary to create a static, acyclic task schedule were intuitive and mostly effective - however, there was a problem with scheduling task sets that had significantly different periods. The refresh delay for the infrared sensor was too large for Xenomai to schedule it wit the shorter tasks for the encoder and humidity/temperature sensor. The Raspberry Pi 3B was easily able to support and monitor the sensors used and ran Xenomai without issue. Lastly, the simplicity and extensibility of this system show the high potential it has for being adapted to a wide variety of fields and responsibilities.

References

ArtOfCircuits. (2018). Encoder module ky-040. Retrieved from http://artofcircuits.com/product/encoder-module-ky-040

Foundation, R. P. (2018). Raspbery pi 3 specifications. Retrieved from https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Henderson, G. (2018). Wiringpi. Retrieved from http://wiringpi.com/

Sunfounder. (2018). Sunfounder sensorkit for rpi2. Retrieved from https://github.com/sunfounder/SunFounder_SensorKit_for_RPi2

Xenomai. (2018). Xenomai task management services. Retrieved from https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy__task.html

8 Source Code

```
#include <wiringPi.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <signal.h>
#include <unistd.h>
#include <math.h>

#include <alchemy/task.h>
#include <alchemy/timer.h>

#define MAXTIMINGS 85

#define ENC_CLK_PIN 1
#define ENC_DT_PIN 2
#define ENC_SW_PIN 3
#define DHTPIN 4

// Tasks
RT_TASK humitemp;
RT_TASK encoder;

RTIME humitemp_period = 1e8;
RTIME encoder_period = 5e6;

// Encoder variables
unsigned char last_enc_status;
unsigned char flag;
unsigned char current_enc_status;

// Humitemp sensor data
int dht11[5] = {0, 0, 0, 0, 0};
int old[5] = {0, 0, 0, 0, 0};

float timer = 0;
float last_timer = 0;
int count, prev;

void quit(int signal) {
    // Suspend tasks, then delete them.
    rt_task_suspend(&humitemp);
    rt_task_suspend(&encoder);
```

```
    printf("Tasks suspended. Deleting...\n");
    rt_task_delete(&humitemp);
    rt_task_delete(&encoder);

    printf("Tasks deleted. Exiting...\n");
    exit(0);
}
/* Attempts to read the encoder data every 50 ms. */
static void read_encoder() {
    rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(encoder_period));  //50ms perio
    while (1) {
        last_enc_status = digitalRead(ENC_DT_PIN);
        while(!digitalRead(ENC_CLK_PIN)) {// timing constraint here?
            current_enc_status = digitalRead(ENC_DT_PIN);
            flag = 1;
        }

        if (flag == 1) {
            flag = 0;
            if (last_enc_status  == 0 && current_enc_status == 1) count++;
            if (last_enc_status == 1 && current_enc_status == 0) count--;
        }
        if (count != prev) printf("Encoder count: %d\n", count);
        prev = count;
        rt_task_wait_period(NULL); // wait for the next period to run the following
  }
}

/* Attempt to retrieve the data from the humidity/temperature sensor. */
static void read_temp() {
    rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(humitemp_period)); // 1s period
    while (1) {
    uint8_t laststate = HIGH;
    uint8_t counter = 0;
     uint8_t j = 0, i;
     float f; // fahrenheit

     dht11[0] = dht11[1] = dht11[2] = dht11[3] = dht11[4] = 0;

     // pull pin down for 18 milliseconds
     pinMode(DHTPIN, OUTPUT);
     digitalWrite(DHTPIN, LOW);
     delay(18);
     // then pull it up for 40 microseconds
     digitalWrite(DHTPIN, HIGH);
```

```
 delayMicroseconds(40);
 // prepare to read the pin
 pinMode(DHTPIN, INPUT);

 // detect change and read data
 for ( i=0; i< MAXTIMINGS; i++) {
 counter = 0;
 while (digitalRead(DHTPIN) == laststate) {
 counter++;
 rt_task_sleep(1);
 if (counter == 255) {
 break;
 }
 }
 laststate = digitalRead(DHTPIN);

 if (counter == 255) break;

 // ignore first 3 transitions
 if ((i >= 4) && (i%2 == 0)) {
 // shove each bit into the storage bytes
 dht11[j/8] <<= 1;
 if (counter > 16)
 dht11[j/8] |= 1;
 j++;
 }
 }

 // check we read 40 bits (8bit x 5 ) + verify checksum in the last byte
 // print it out if data is good
 if ((j >= 40) &&
 (dht11[4] == ((dht11[0] + dht11[1] + dht11[2] + dht11[3]) & 0xFF)) ) {
 f = dht11[2] * 9. / 5. + 32;
 printf("Humidity = %d.%d %% Temperature = %d.%d *C (%.1f *F)\n",
 dht11[0], dht11[1], dht11[2], dht11[3], f);
        old[0] = dht11[0];
        old[1] = dht11[1];
        old[2] = dht11[2];
        old[3] = dht11[3];
}

    // Data bad -- print last good reading
 else
 {
 printf("Humidity = %d.%d %% Temperature = %d.%d *C (%.1f *F)\n",
```

```
     old[0], old[1], old[2], old[3], f);
     }
        int error_code = rt_task_wait_period(NULL); // wait for the next period to run t
        if (error_code) { printf("Error from rt_task_wait_period for read_encoder %s\n",
    }
}

/* Initializes the ISR for the Encoder. */
void init_encoder() {
pinMode(ENC_SW_PIN, INPUT);
pinMode(ENC_DT_PIN, INPUT);
pinMode(ENC_CLK_PIN, INPUT);

pullUpDnControl(ENC_SW_PIN, PUD_UP);
    if (wiringPiISR(ENC_SW_PIN, INT_EDGE_FALLING, &read_encoder), 0) {
        printf("Unable to initialize ISR\n");
    }
}

int main(int argc, char* argv[]) {
    signal(SIGINT, quit);
    printf("Initializing tasks.\n");
    if (wiringPiSetup() < 0) printf("WiringPi setup error\n");
    init_encoder();
    printf("Initialization complete.\n");

    // &task, name, stack size, priority, mode);

     rt_task_create(&encoder, "Encoder", 0, 50, 0);
     rt_task_start(&encoder, &read_encoder, 0);

     rt_task_create(&humitemp, "Humitemp", 0, 49, 0);
     rt_task_start(&humitemp, &read_temp, 0);
     while(1);
    return 0;
}
```