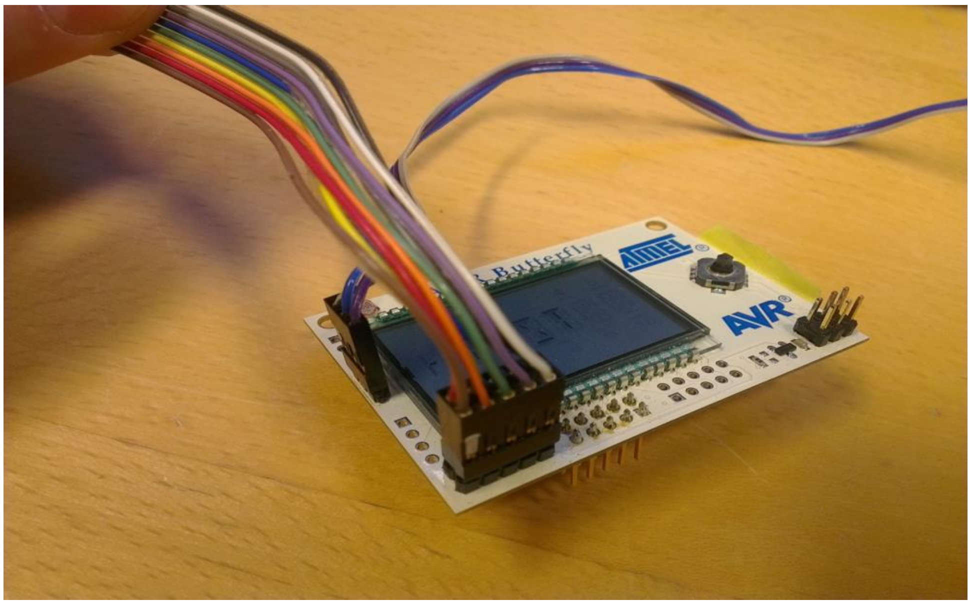
Exercise 5: BRTT and Linux

In this exercise we will use the Butterfly Real-Time Tester to test the Linux operating system installed in the real-time lab. We will do similar tests as we did in exercise 3 and 4 with the AVR and FreeRTOS.

1. Using I/O card

The interface to the BRTT consists of digital I/O signals, so to do the test the computer must be able to send and receive signals. The computers in the real-time lab have a National Instrument I/O card that we will be using. To use this (and many other cards) in Linux, we use the Comedi driver. The files io.h and io.c have been prepared with the necessary functionality to communicate with the Butterfly. See comments in the header file to get the details.

In the Real-Time lab the IO cards inputs and outputs are available on a green rack. Each digital IO has an LED, showing the current value of the channel. Among the cables going out of the green rack is a 5x2 header cable that has been prepared for use with the Butterfly. When connecting this header cable to the Butterfly, its white dot should be on bottom left pin of the PortB of the Butterfly. This is shown below.



To get Comedi to work, the program must be linked with the comedi library, as well as the math library. When adding pthreads to the mix, the final list of linker flags should be
`-lpthread -lcomedi -lm`
In the later parts of this exercise, we will be using some library functions that only compile with GNU extentions enabled. You will need to use the compiler flag `-std=gnu11` to enable these.

2. Reaction test using busy-wait

As in the previous exercise using FreeRTOS, we will start by doing a reaction test where we busy-wait of the input pin.

Task A:

Create 3 POSIXs threads, one for each of the tests A, B and C. The threads are created with the functions you learned to use in exercise 2. Each thread waits until it receives its test signal, and then it will send its response signal back. Run a test of 1000 subtests and store the results.

If you get valid results in many subtests, but then suddenly there is an "overflow", it means that the system was not able to reply within the 65 milliseconds that the BRTT waits for replies. You can probably run another set of tests and that will be able to complete. If this is a consistent problem, there is a problem with your code.

2.1. Comparing single-core and multi-core

The computers at the real-time lab have quad-core CPUs. The speed of the CPU and the fact that it can run three processes or threads in parallel will to some degree mask poor real-time scheduling for us. To get more relevant results that differentiate between good and bad real-time performance, we need to make some changes to our program.

Task B:

First of all we want to run everything on the same CPU core. The following function can be called in the start of a spawned thread function, and will ensure that the thread only will be executed on the specified CPU core.

```
#include <sched.h>

int set_cpu(int cpu_number){
	cpu_set_t cpu;
	CPU_ZERO(&cpu);
	CPU_SET(cpu_number, &cpu);

	return pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpu);
}
```

Run the same test as in A, with all threads on the same CPU core, and record the results for an A+B+C test with 1000 iterations.

Compare the results between running on a single core and all four. Compare the shape of the distributions: do any of the histograms have any particular spikes, and why? Which one has the best worst-case response time, and why?

2.2. Adding disturbance

In addition to forcing all threads to run on the same CPU-core, we will also create additional threads that run on the same core just to disturb the application. Each of these disturbance threads should set themselves to run on the same core, before starting an infinite loop of either some simple calculations, or just

```
while(1){
    asm volatile("" ::: "memory");
}
```

Task C:

Spawn 10 disturbance threads running an infinite loop, on the same CPU core as the response task threads. Record those results - again, an A+B+C test with 1000 subtests.

We will get back to comparing these results in task D.

3. Periodic POSIX threads

In order to make a thread execute periodically, we will use the function

```
clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &waketime, NULL);
```

Calling this function will put this thread to sleep until the time specified by `waketime`, which is of the type `struct timespec`. We get the start time with clock_gettime, then add a fixed period to the waketime each iteration of our periodic task.

```
struct timespec waketime;
clock_gettime(CLOCK_REALTIME, &waketime);

struct timespec period = {.tv_sec = 0, .tv_nsec = 500*1000*1000};

while(1){
    // do periodic work ...

    // sleep
    waketime = timespec_add(waketime, period);
    clock_nanosleep(GLOCK_REALTIME, TIMER_ABSTIME, &waketime, NULL);
}
```

With these functions you should be able to create a periodic pthread, test this by creating a thread that prints a message every 500ms or similar.

3.1. Reaction test with periodic POSIX threads

Task D:

Create three tasks that periodically poll the test signal from the BRTT, and set its response signal as soon it receives it. The period should be 1ms, but feel free to experiment with shorter periods. What happens if the period is too short?

As before, you should run all threads on the same CPU. Do an A+B+C reaction test with 1000 subtests, with and without disturbance.

Compare the four results: busy-polling the input vs periodic execution, both with and without disturbance:

- Without the disturbance enabled - which one has the fastest response, and why?
- Which one is less impacted by the addition of the disturbance, and why?
- When the disturbance is enabled - do any of these have a predictable and consistent worst-case response time?