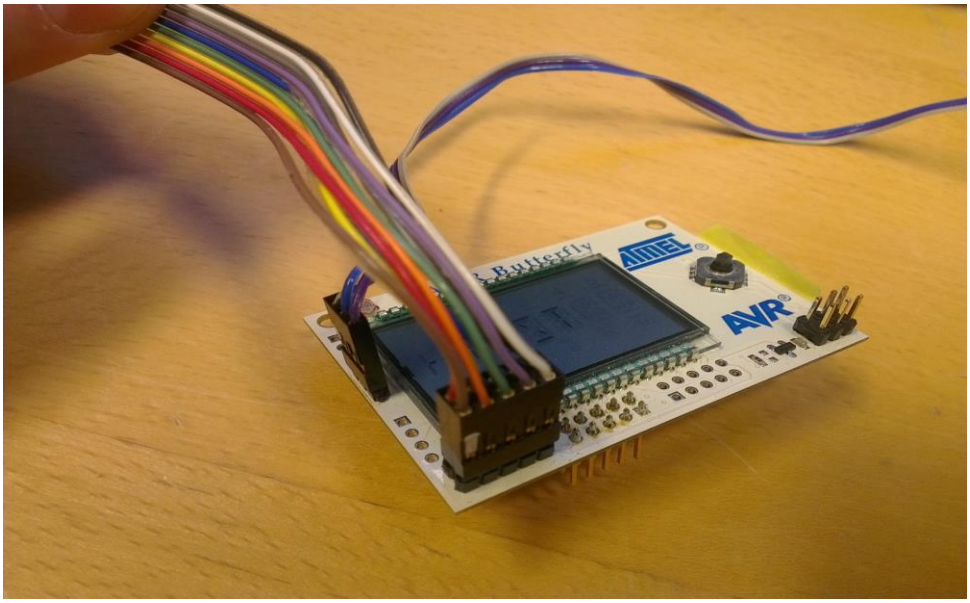
Exercise 5: BRTT and Linux

In this exercise we will use the Butterfly Real-Time Tester to test the Linux Mint operating system installed in the real-time lab. We will do similar tests as we did in exercise 5 with the FreeRTOS system on AVR.

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` from itslearning 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 a LED, showing the current value of the channel. Among the cables going out of the green rack is a 5x2 header cable that have 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.



You should create a small test program using `io.h` and `io.c` to test that you can control the I/O card. The digital outputs are easy to test since changing its value will turn its LED on or off.

To get Comedi to work the program must be linked with the comedi library (`-lcomedi`). In addition you will need `-lrt`, `-pthread` and `-lm`.

There is a known bug with -lm, *the solution is simply adding it last among the included libraries. Also if the linker doesn't find the included libraries, make sure the libraries are listed at the end of the linker command.*

2. Reaction test using busy-wait

In the same way as in exercise 5 we will start by doing a reaction test using a busy-wait strategy.

Assignment 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 were not able to reply within the 65 milliseconds that the BRTT waits for replies. You can probably run another tests and that will be able to complete. If it is a consistent problem, it might be a problem with your code.*

2.1. Adding disturbance

The computer on the real-time lab has a quad-core CPU. The speed of the CPU and the fact that it can run three processes or threads in parallel will to a 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 do some changes to our program.

Bind to CPU-core

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

```
int set_cpu(int cpu_number)
{
      // setting cpu set to the selected cpu
      cpu_set_t cpu;
      CPU_ZERO(&cpu);
      CPU_SET(cpu_number, &cpu);

      // set cpu set to current thread and return
      return pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t),
&cpu);
}
```

To set which CPU a thread should run on, you need to include `sched.h` and add `-D_GNU_SOURCE` to your compiler flags in the Makefile.

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 some simple calculations (busy_wait).

Assignment B:

Run the same test as in A, with all threads on the same CPU core. Together with the 3 threads, you should also run 10 disturbance threads in an infinite loop on the same CPU core.

Try to run an A+B+C test with 1000 subtests. If you are not able to complete such a test without overflow, you might want to add a short sleep (nanosleep or usleep), or use the `pthread_yield()` function after each time you read IO.

3. Periodic POSIX threads

Earlier we have created POSIX threads (pthreads) in Linux, and now we will make these periodic in a similar way as we did with the tasks in FreeRTOS.

To make a periodic task in FreeRTOS we used `vTaskDelayUntil()` to sleep until a future point in time. Now we will use the following function call:

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

When running this function, the thread will sleep, allowing other threads to execute, until a future point in time. This point in time is defined by `next`, which is a timespec struct. The struct is initialized to the current time with the following function:

```
clock_gettime(CLOCK_REALTIME, &next);
```

To wait until a future time, the timespec must be increased. Below is a simple function for increasing a timespec with a number of microseconds that you can use.

```
void timespec_add_us(struct timespec *t, long us)
{
	// add microseconds to timespecs nanosecond counter
	t->tv_nsec += us*1000;

	// if wrapping nanosecond counter, increment second counter
	if (t->tv_nsec > 1000000000)
	{
		t->tv_nsec = t->tv_nsec - 1000000000;
		t->tv_sec += 1;
	}
}
```

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.

4. Reaction test with periodic POSIX threads

Assignment C:

Create three periodic threads with a period of 1 ms. Let each of these periodically poll for a test signal from the BRTT, and set its response signal as soon it receives it. The actions of this program should be as similar to Assignment C of exercise 4 as possible.

As in B you should run all threads on the same CPU. You should first test the program without disturbance threads, before testing it with the same 10 disturbance threads as in B.

Do an A+B+C reaction test with 1000 subtests, with and without disturbance. Is there a difference?

Do the same tests with a period of 100 us (with disturbance). If this does not give you significantly different results, try even shorter period.

5. Approving the exercise

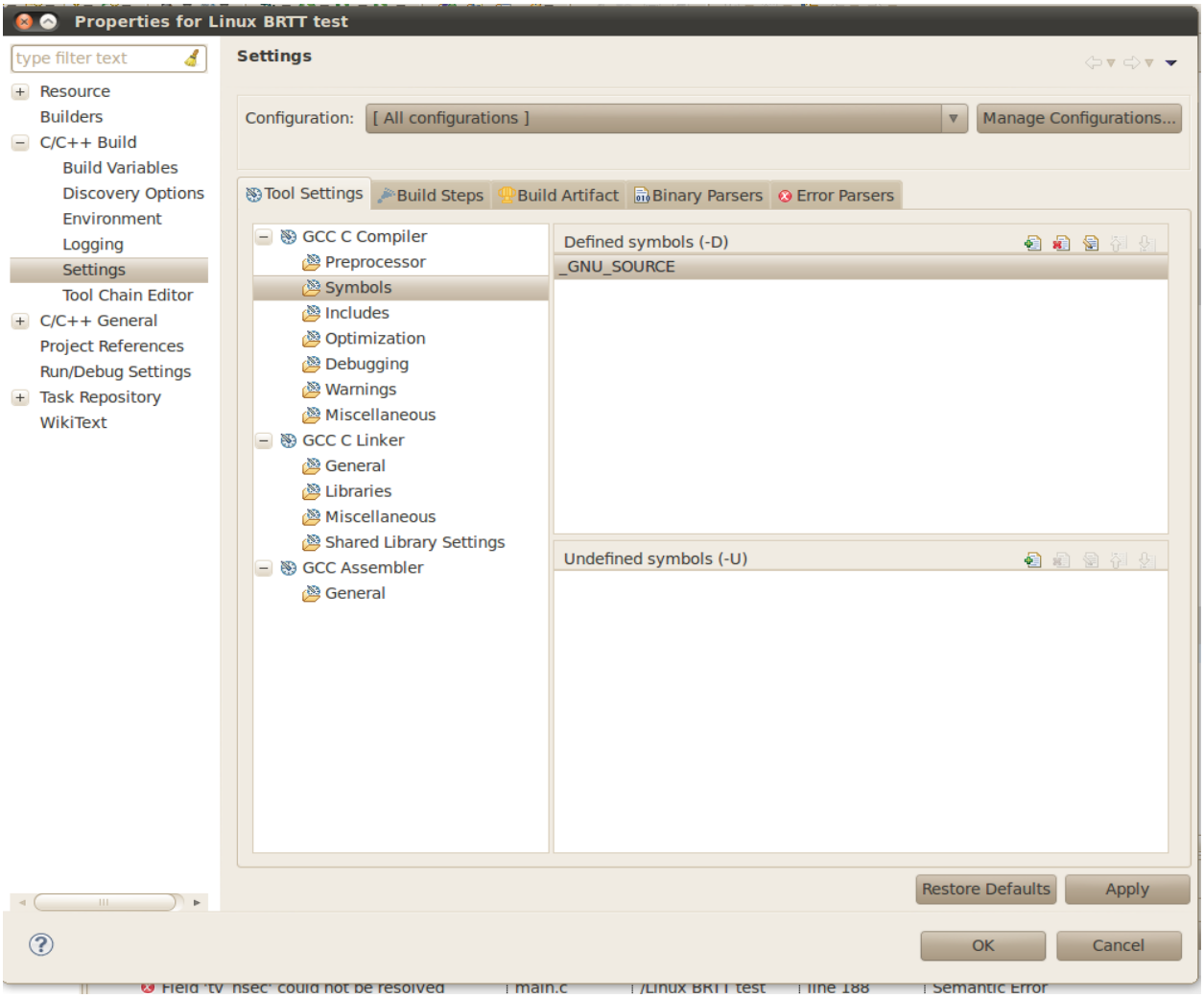
For this exercise you need to show the following to a student assistant to get the exercise approved.

1. Run a busy-wait test with disturbance, and show results both with and without disturbance.
    - What happens when disturbance is introduced?
    - Are you able to get any results before using the “yield” function?
    - What does it do?
2. Run a reaction test with period of 1 ms, and show results both with and without disturbance.
3. Run a reaction test with period of 100 us or lower and show results.
    - What happens when the period is lower?
    - What are the implications if the periodic function is supposed to do some calculations during a period not just read and write io as we do here?

Store your results from the timer test and from the reaction test using busy-wait, periodic polling with period of 1 ms and 100 us. You will use these results to compare with other tests you will do later.

6. Appendix: Eclipse Configuration

The `_GNU_SOURCE` flag must also be added in the Eclipse configuration, as shown below. As with the libraries, you should not include the initial `-D`.


Properties for Linux BRTT test
type filter text
Resource
Builders
C/C++ Build
Build Variables
Discovery Options
Environment
Logging
Settings
Tool Chain Editor
C/C++ General
Project References
Run/Debug Settings
Task Repository
WikiText
Settings
Configuration:
[All configurations]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
GCC C Compiler
Preprocessor
Symbols
Includes
Optimization
Debugging
Warnings
Miscellaneous
GCC C Linker
General
Libraries
Miscellaneous
Shared Library Settings
GCC Assembler
General
Defined symbols (-D)
_GNU_SOURCE
Undefined symbols (-U)
Restore Defaults
Apply
OK
Cancel