Embedded and Real-Time Operating Systems

Luis Schubert, Chiara Piccolroaz

January 29, 2025

1 Installing QNX 7.0 on the BeagleBone

This section provides a detailed description of installing QNX 7.0 on the BeagleBone. The goal is to configure the BeagleBone as a development environment that provides both power and network connectivity using a single USB cable. The steps include connecting to BeagleBone via USB and Setting Up Serial Communication, preparing and Booting from a Bootable SD Card, configuring the QNX Image in Momentics, establishing an NCM Network Connection, creating and running new projects on QNX.

1.1 Establishing a Physical Connection to the BeagleBone

To establish a stable serial communication between the BeagleBone and the host system:

1. Connect the USB cable to the UART port J1 of the BeagleBone.

2. Wire the USB connections to the BeagleBone as follows:
   - Connect the GND pin from the USB to the 1st pin (GND) on J1;
   - Connect the TX pin from the USB to the 4th pin (RX) on J1;
   - Connect the TX pin from the USB to the 4th pin (RX) on J1.

1.2 Configuring a Serial Connection Using PuTTY

To establish a serial connection using PuTTY:

1. Start PuTTY with root privileges to gain access to serial interfaces.

2. Configure the serial connection as follows:
   - Port: /dev/ttyUSB0;
   - Baud rate: 115200 (default baud rate for the BeagleBone);
   - Data bits: 8;
   - Parity: None;
   - Stop bits: 1;
   - Flow control: None.

3. Save the configuration to make it easy to load for this and subsequent sessions.

1.3 Preparing from a Bootable SD Card

Before booting the BeagleBone with QNX, an SD card must be prepared accordingly:

1. Acquire the necessary files:
   - Download *MLO* and *u-boot.img* from the provided course platform.
   - Generate the *ifs-ti-am335x-beaglebone.bin* image. This is achieved by compiling the QNX BSP project (*BSP-ti-am335x-beaglebone.zip*) in the Momentics IDE. This compiled binary will be found in the images directory.

2. (If necessary) Format the SD card:
   If the SD card is corrupted or contains unwanted partitions, it should be formated. First, use a partitioning tool such as *GParted* to create a new partition table in msdos format. Then, format the entire card to the FAT32 file system to ensure compatibility with the BeagleBone.

3. (If necessary) Mount the SD card:
   If the SD card is not automatically recognized, it should be mounted manually. The following code lines enable the creation of a directory, the identification of the device, and the mounting of the SD-card.

```
    # enter the following code
    sudo mkdir /mnt/<dir_name>                        # creation
    lsblk                                             # identification
    sudo mount /dev/<device_name> /mnt/<dir_name> # mounting

```

4. Copy the files to the SD card:
   In the final step, copy the required files to the SD card using the commands:

```
    # enter the following code
    sudo cp <path_to_MLO> /mnt/<dir_name>         # MLO
    sudo cp <path_to_u-boot.img> /mnt/<dir_name> # u-boot.img
    sudo cp <path_to_ifs> /mnt/<dir_name>         # ifs-ti-am335x-beaglebone.bin

```

   The files *MLO*, *u-boot.img*, and *ifs-ti-am335x-beaglebone.bin* must be copied exactly in this order.

1.4 Booting from a Bootable SD Card

To boote the BeagleBone with QNX from the prepared SD card:

1. Connect the USB cable and open PuTTY with the saved serial configuration.

2. Power on the BeagleBone.

3. Interrupt the boot process by pressing any key in the terminal to access the U-Boot prompt. Here the symbol => should be displayed.

4. Enter the following commands to load and start the QNX image. Here the kernel shell prompt (#) should be displayed, indicating a successful boot.

```
    # enter the following code
    mmcinfo
    fatload mmc 0 81000000 ifs-ti-am335x-beaglebone.bin
    go 81000000

```

1.5 Configuring the QNX Image in Momentics

The kernel configuration includes necessary drivers and startup settings which run at startup of the BeagleBone. This is specified in the image file *ifs-ti-am335x-beaglebone.bin* which is generated upon the compilation of the beaglebone.build file within the QNX BSP project *BSP-ti-am335x-beaglebone.zip*. In order to modify the kernel configuration the build file must be modified accordingly.

1. Import the QNX BSP project:
   Right-click in the Project Explorer and select QNX Source Package and BSP as shown in the figure 1a. After assigning a name and selecting the correct path to the zip file, the project should be visible in the Project Explorer.

2. (Recommanded) Avoid overwriting the build file:
   We recommend creating a new target to build the image while preserving the custom build file in order to avoid unintentional overwriting of the build file during compilation. For this, right-click on the images folder in the Momentics IDE and select *Build Targets*, then *Create*. Name the target "all" as specified in the Makefile within the image folder.

3. Modify the build file accordingly:
   To establishing an NCM Network Connection, we uncomment a few relevant lines of code within the build file (see GitLab).

4. Reboot from the SD card:
   From now on, the image can be generated by double-clicking on the "all" target. It can then be transferred to the SD card to allow the BeagleBone to reboot from the SD card (see steps above). For a sanity check, append a message (e.g. "hello world" / version number / ...) to $display_msgr$ in the build file.

1.6 Establishing an NCM Network Connection

To create a NCM Network Connection via USB cable:

1. Verify the cnm recognition:
   To verify that the ncm device is recognized enter one of the following commands, which should return a similar output:

```
    # enter the following code
    ifconfig #list TCP/IP-devices

    # expected output
    enx020000040506: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::458:6b25:ace3:e0e7  prefixlen 64  scopeid 0x20<link>
        ether 02:00:00:04:05:06  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1936 (1.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
    # enter the following code
    ip a #list IP-devices

    # expected output
    3: enx020000040506: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
    link/ether 02:00:00:04:05:06 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.101/24 brd 192.168.10.255 scope global enx020000040506
    valid_lft forever preferred_lft forever
```

2. (If necessary) Fix possible problems:
   If the ncm device is not recognized, we can disable network-manager's handling of the QNX interface and make sure that the connection is passed through to the VM.

   To handle the first possible issue, edit the $NetworkManager.conf$ file by including the MAC-adress of the QNX interface under the option $unmanaged-devices$. This adress can be found by the command ipa

```
    # enter the following code
    ip a
    sudo vim etc/NetworkManager/NetworkManager.conf # opn the NetworkManager.conf
```

```
    # edit the following line with the correct mac-adress
    unmanaged-devices=mac:<mac-adress of QNX interface>
```

   To handel the second possible issue, we can make sure that the host user is in the group $vboxusers$, with the following code:

```
    # enter the following code
    sudo usermod -a -G vboxusers $USER
```

   Then activate "QNX Software Systems QNX NCM Network Device" in the Virtualbolx. To do add appropriate USB-Device-Filter to the Virtualmachine by clicking on the USB-Symbol in the bottom left corner of the Virtualbox. Then right-click on the bottom right of the running Virtualmachine on the USB-Symbol and check the appropriate box.

3. Assign a static IP-Adress:
   The best way is to configure QNX using a dhcp server to dynamically assign an IP-adress to the newly connected device. For simplicity, we manually assign a static IP-adress. Thereby *enx020000040506* is the device name, 192.168.10.101 is the IP-address assigned to the BeagleBone, and 192.168.10.100 is IP-address of the host device.

```
    # enter the following codes
    sudo ifconfig enx020000040506 192.168.10.101     # for assigning the IP-adress
    ping 192.168.10.101                              # for securing connection

```

1.7 Creating and running new projects on QNX

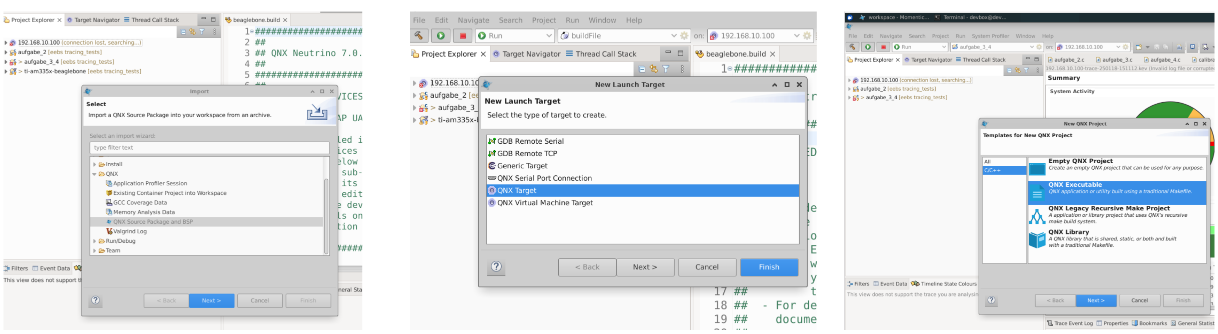
To run a new project in the Momentics IDE:

1. Select a new launch target:
   Create a new launch target by clicking on the Target options and then selecting $QNXTarget$, as shown in Figure 1b. Then enter the correct host IP-address, which in our case is 192.168.10.100.

2. Create a new project:
   Right-click in the Project Explorer and select New QNX Project as shown in the figure 1c. Then choose the options C/C++ and QNX Executable. After assigning a name and selecting the correct path, the project should be visible in the Project Explorer.

3. Select a new launch Configuration:
   Create a new launch configuration by clicking on the Configuration options and then selecting $C/C++QNXApplication$, similarly to the steps for select a new launch target. Then a name and the correct QNX Project among the one created previously in the Project Explorer.

   After selecting the correct launch Configuration and launch target, the object file for the selected project can be created and run on the Beagle Bone by clicking on the Build icon (hammer) and the Run icon (green arrow).



(a) (b) (c)

Figure 1: Creating and running new projects on QNX: Left: Import the QNX BSP project; Center: Select a new launch target; Right: Create a new project

2 Threads und Condition Variables

To simulate people entering and leaving a room, we implemented a program using POSIX threads and condition variables. The implementation can be found in *aufgabe_2.c*.

2.1 Context

Threads are independent execution units within a process that share memory but run concurrently. In this exercise, we utilize POSIX threads (pthread), a standardized threading API for Unix-like systems that provides functions for creating, synchronizing, and managing threads. Synchronization can be achieved using condition variables together with mutexes. A condition variable allows threads to wait for or signal changes in a shared state, while a mutex is a synchronization primitive that ensures exclusive access to shared resources.

2.2 Approach

To accomplish this task, we use three POSIX threads, a condition variable and a mutex. Two threads, *arrival_thread* and *departure_thread*, simulate the arrival and departure of people at random intervals. The third thread, *monitor_thread*, monitor and log changes in room occupancy whenever a person enters or leaves the room. These movements are stored in a shared variable that can be safely accessed by a mutex and monitored by the conditional variable.

2.3 Implementation

In the main function of the program, the threads are created using the *pthread_create*() function and cleanly terminated using *pthread_join*() after a specified period (e.g., 10 seconds). Similarly, synchronization mechanisms such as the mutex and condition variable are properly destroyed after execution. During this execution period, the threads execute concurrently the following functions.

```c
void* arrival_thread(void* arg) {
    while (keep_running) {
        // Simulate the arrival of a person
        pthread_mutex_lock(&mutex); // Lock the Mutex
        people_in_room++;
        printf("A person enters the room. Current occupancy: %d\n", people_in_room);
        pthread_cond_signal(&condvar); // Signal to the monitor_thread
        pthread_mutex_unlock(&mutex); // Unlock the Mutex
        // Wait for a random time before the next person arrives (between 1 and 3 seconds)
        sleep(rand() % 3 + 1);
    }
    printf("arrival_thread is finished.\n");
    return NULL;
}
```

Code Listing 1: *arrival_thread*

```c
void* departure_thread(void* arg) {
    while (keep_running) {
        pthread_mutex_lock(&mutex); // Lock the Mutex
        // Simulate the leaving of a person
        if (people_in_room > 0) {
            people_in_room--;
            printf("A person leaves the room. Current occupancy: %d\n", people_in_room);
            pthread_cond_signal(&condvar); // Signal to the monitor_thread
        }
        pthread_mutex_unlock(&mutex); // Unlock the Mutex
        // Wait for a random time before the next person leaves the room (between 1 and 3 seconds)
        sleep(rand() % 3 + 1);
    }
    printf("departure_thread is finished.\n");
    return NULL;
}
```

Code Listing 2: *departure_thread*

The *arrival_thread* and *departure_thread* functions simulate people entering and leaving a room by incrementing and decrementing a shared integer variable, *people_in_room*, at random intervals (e.g. every 1 to 3 seconds). The mutex *mutex* is held during this operation to ensure safe access to this shared variable between the three threads, while the conditional variable *condvar* is used to notify the monitoring thread of changes.

```c
void* monitor_thread(void* arg) {
    while (keep_running) {
        pthread_mutex_lock(&mutex); // Lock the Mutex
        pthread_cond_wait(&condvar, &mutex); // (unlock mutex) Wait for signal (lock mutex)
        printf("Monitor: Current occupancy: %d person(s) in the room.\n", people_in_room);
        pthread_mutex_unlock(&mutex); // Unlock the Mutex
    }
    printf("monitor_thread is finished.\n");
    return NULL;
}
```

Code Listing 3: *monitor_thread*

The *monitor_thread* function continuously monitors the room and logs its changes after receiving the necessary signal from the other two threads. Again, a mutex is needed to prevent it from changing during logging.

3 Zeit verbraten

3.1 Approach

In order to allow for basic simulation of a realtime system in assignment 4, a function consuming CPU time (`waste_time`) is required. To properly assess the realtime properties of the simulated system, this function must be calibrated properly. Since the execution of `waste_time` should not require any external libraries and thereby has no direct feedback regarding the actual elapsed time, it receives a parameter which controls the extent of the function's CPU usage. To statically calibrate this parameter, an external function (`calibrate`) is used. It calls `waste_time` repeatedly with different values, each time measuring the elapsed time and adjusting the parameter accordingly until a specified accuracy threshold is met.

3.2 Implementation

`waste_time` is implemented by iterating a loop, which solely increments a `volatile` variable. The variable has to be volatile in order to avoid the loop not being executed as expected due to compiler optimization. The number of iterations is specified by the function's parameter mentioned above. An implementation of `waste_time` can be seen in Listing 4.

```c
void waste_time(size_t count) {
    volatile size_t i = 0;
    while (i < count) {
        ++i;
    }
}
```

Code Listing 4: waste_time

`calibrate` initially calls `waste_time` wich a rough estimate of the necessary iterations to acchieve the desired time. It then uses `time.h`'s `clock` functionality to measure the actually elapsed time in microseconds and recomputes the number of iterations accordingly for the next call to `waste_time`. This is done until the deviation of the actually elapsed time from the desired time is below 0.01%. For a calibration time of $1ms$, this acchieves an accuracy of $10\mu s$. A shortened implementation of `calibrate` can be seen in Listing 5.

```c
size_t calibrate(size_t desired_ms) {
    //...
    size_t count = desired_ms * 1700000;
    while(true) {
        start = clock();
        waste_time(count);
        end = clock();
        double elapsed_ms = clocks_to_ms(end - start);
        double factor = (double)desired_ms / (double)elapsed_ms;
        if (fabs(factor - 1.0) < 0.0001) {
            break;
        }
        count *= factor;
    }
    return count;
```

```
}
```

Code Listing 5: calibrate

4 Kooperierende Tasks

To implement a clock generator task that sets timing semaphores at fixed intervals, we developed a program using POSIX threads and semaphores. The implementation is available in $aufgabe_4.c$, but it also makes use of components presented in other exercises, such as the $waste_time$ function of exercise 3.

4.1 Context

POSIX threads (pthread) were introduced in exercise 2. Semaphores are primitives used to control access to shared resources, ensuring that tasks are executed in the desired order and within defined timing constraints. Both mechanisms can be used in real-time systems that require precise timing to ensure tasks execute within specified deadlines, as required in this exercise.

4.2 Approach

To accomplish the task, the program uses four POSIX threads and three semaphores to coordinate their execution. The *masterT* thread acts as a clock, generating timing signals at constant intervalls (e.g. 2 ms). The *slaveT* thread waits for the master signal before consuming CPU time for fraction of these intervals (e.g. 1 ms) and periodically signaling two additional threads, $followingT1$ and $followingT2$. These threads run concurrently, waiting for signals from the *slaveT* thread to consume CPU time (e.g 4 ms). The synchronization between *masterT* and *slaveT* threads, as well as between *slaveT* and the $followingT1$ and $followingT2$ threads, is managed by the corresponding semaphores.

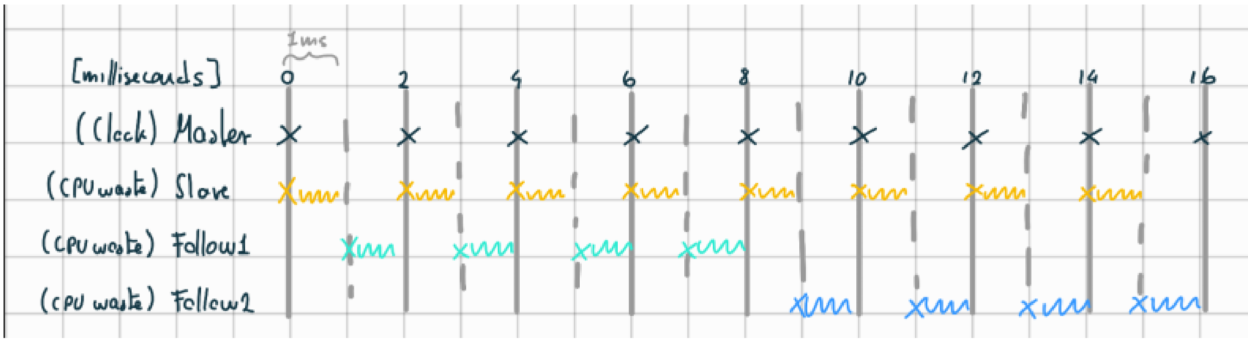
4.3 Implementation

The main function initializes semaphores using the *sem_init*() function and threads with the *pthread_create*() function. To meet real-time constraints, appropriate scheduling policies such as $FIFO$ are assigned, with priorities set as follows: the highest priority for the *masterT* thread, medium priority for the *slaveT* thread, and the lowest priority for the $followingT1$ and $followingT2$ threads. These threads call the functions shown below after being properly terminated with *pthread_join*().

```c
define MASTER_PERIOD_MS = 2;
void* masterT_task(void* arg) {
    printf("MasterT: masterT_task started\n");
    long tact_count_masterT = MASTER_PERIOD_MS; // (2ms)
    struct timespec next_time;
    clock_gettime(CLOCK_MONOTONIC, &next_time); // Get current time
    while (1) {
        printf("MasterT: Tact started\n");
        next_time.tv_nsec += tact_count_masterT * 1000000; // Incrementing the time
        if (next_time.tv_nsec >= 1000000000) { // Adjusting the time for overflow
            next_time.tv_sec += next_time.tv_nsec / 1000000000;
            next_time.tv_nsec %= 1000000000;
        }
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next_time, NULL); // Incrementing till the calculated time
        sem_post(&master_slave_sem); // Signals slaveT_task
        printf("MasterT: Semaphore signal sent\n");
    }
    return NULL;
}
```

Code Listing 6: *masterT_task*

The *masterT_task* function generates clock signals every 2 ms using *clock_nanosleep* with absolute time. First, it saves the current time in the *next_time* variable. Then it repetetly calculates the next wake-up time by incrementing the nanosecond field of variable by 2000000 ns and adjusting for overflow. After sleeping till the caluated time, it wakes the slave thread via the semaphore *master_slave_sem*.




```c
void* slaveT_task(void* arg) {
    printf("SlaveT: slaveT_task started\n");
    while (1) {
        sem_wait(&master_slave_sem); // Wait for signal from masterT_task
        waste_time(iterations_1000_ms / 1000 * SLAVE_TIME_MS); // Waste time (1 ms)
        if (n_counter % N == 0) {
            sem_post(&slave_following_sem1); // Signals followingT1_task
            printf("SlaveT: Semaphore signal sent\n");
            sem_post(&slave_following_sem2); // Signals followingT2_task
            printf("SlaveT: Semaphore signal sent\n");
        }
        n_counter++;
    }

    return NULL;
}
```

Code Listing 7: *slaveT_task*

The *slaveT_task* function waits for the master thread's semaphore signal before wasting 1ms of CPU time using the *waste_time* function from exercise 3 and signalising the two following threads to proceed via their respective semaphores, *slave_following_sem*1 and *slave_following_sem*2. Considering the interval and the wasting times suggested in the description of this exercise, the *slaveT* thread must signal the *followingT*1 and *followingT*2 threads every 8 times to meet real-time conditions. This number of times corresponds to 16 ms, which is numerically enough time for the following threads to waste 4 ms each after having wasted 1 ms themselve, as shown in Figure . However, as we will demonstrate in the next exercise, 16 ms is not enough time to meet these condictions.

```c
void* followingT1_task(void* arg) {
    printf("followingT1_task started\n");
    while (1) {
        sem_wait(&slave_following_sem1); // Wait for signal from slaveT_task
        waste_time(iterations_1000_ms / 1000 * FOLLOWING_TIME_MS); //Waste time (4 ms)
        printf("followingT1: Semaphore signal received\n");
    }
    return NULL;
}

void* followingT2_task(void* arg) {
    printf("followingT2_task started\n");
    while (1) {
      sem_wait(&slave_following_sem2); // Wait for signal from slaveT_task
        waste_time(iterations_1000_ms / 1000 * FOLLOWING_TIME_MS); //Waste time (4 ms)
        printf("followingT2: Semaphore signal received\n");
    }
    return NULL;
}
```

Code Listing 8: *followingT1_task* and *followingT2_task*

The functions *followingT1_task* and *followingT2_task* wait for a semaphore from the slave thread, before watsing 4 ms of CPU time with the help of the *waste_time* function of exercise 3.

5 Kernel Events Tracing

5.1 Tracing with QNX Momentics

The Momentics IDE supports tracing functionality out of the box. To enable it for a specific launch configuration, open the launch configuration editing window, select "Tools" and check the box "System Profiler". This will automatically capture events during the program execution[1]. Once the tracing is complete, the resulting file should be opened automatically. If it is not, search for a file with the extension ".kev" in the project explorer. To analyze the previous exercise, we use the "Timeline" display type.

5.2 Tracing results

While tracing the previous exercise, we examined three different scenarios, each with different parameters for

- the time spent in `waste_time` by the slave thread, s
- the time spent in `waste_time` by the following threads, f
- the number of times, the slave thread wakes up between triggering the following threads k and
- the time of one whole cycle, c, which is $c = 2 * k$ since the master thread triggers the slave thread every $2ms$.

Scenario 1 $s = 1ms$, $f = 4ms$, $k = 8 \quad \rightarrow \quad c = 16ms$
These parameters lead to a theoretical minimal execution time of $k * s + 2 * f = 16$, in theory resulting in a load of ${}^{16}/_{c} = 1$. In this scenario, the realtime requirements can not be met, as can be seen in Figure **??**. Even though, in theory, there is enough time to accomodate all threads' CPU requirements, the actual execution time exceeds c. This is due to the fact, that context switches as well as other overhead in the thread functions themselves also consumes CPU time, resulting in a lack of time to complete all threads' tasks before a new cycle begins. The tracing result for this scenario is shown in Figure 3a.

Scenario 2 $s = 1ms$, $f = 3ms$, $k = 8 \quad \rightarrow \quad c = 16ms$
These parameters lead to a theoretical minimal execution time of $k * s + 2 * f = 14$, in theory resulting in a load of ${}^{14}/_{c} = 0.875$. In this scenario, the realtime requirements are met since the reduced load is enough to accomodate the time required for context switches and other overhead. However, not only the theoretical but also the actual load is well below 100%. The tracing result for this scenario is shown in Figure 3b.

To accomplish a higher load while still meeting the realtime requirements, we examine the following scenario.

Scenario 3 $s = 1ms$, $f = 5ms$, $k = 11 \quad \rightarrow \quad c = 22ms$
These parameters lead to a theoretical minimal execution time of $k * s + 2 * f = 21$, in theory resulting in a load of ${}^{21}/_{c} \approx 0.955$. The tracing result for this scenario is shown in Figure 3c.

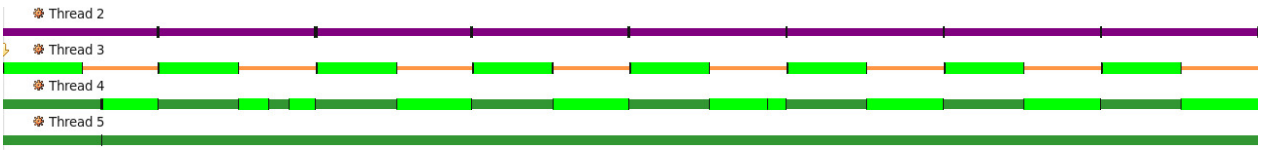
6 Anpassung der Stacksize der verwendeten Threads

In order to minimize memory usage, the stacksizes in assignments 2 and 4 shall be minimized. To achieve this, the maximal required stacksize of each individual thread has to be determined and the available stacksizes have to be adjusted accordingly.

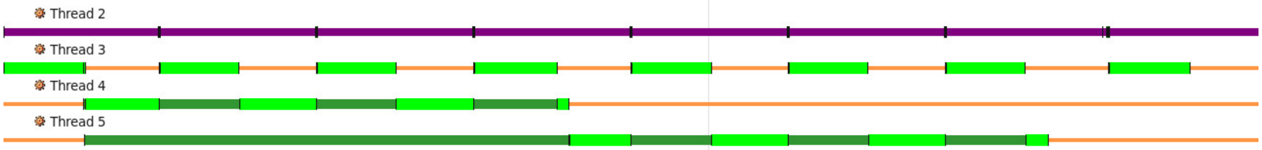
6.1 Reducing the stacksize for normal execution

According to the QNX documentation, when setting a custom stacksize, one should use the amount of memory the stack uses plus `PTHREAD_STACK_MIN` in order to accomodate any overhead.

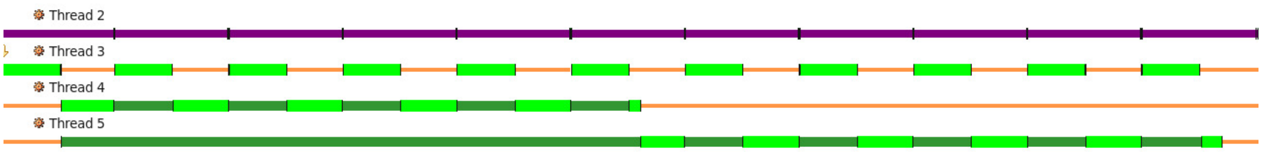
[1] The System Profiler will capture events during the entirety of the program's execution. To avoid latency issues while examining the trace, avoid capturing for long periods of time.



a: Trace for execution with $k = 8, f = 4ms, c = 16ms$, spare: $0ms$



b: Trace for execution with $k = 8, f = 3ms, c = 16ms$, spare: $3.7ms$



c: Trace for execution with $k = 11$, $f = 5ms$, $c = 21ms$, spare: $1.7ms$

Figure 3:
Tracing results of executions with the parameters of the different scenarios, showing the master thread ("Thread 2"), the slave thread ("Thread 3") as well as the two following threads ("Thread 4" and "Thread 5" respectively).
The colors of the graphs represent the following thread-states: Orange: "Waiting for semaphore", Dark Green: "Ready", Light Green: "Running" and Purple: "Sleeping". Each Figure shows a capture of exactly one cycle (c ms).
The times Thread 5 is waiting for a Semaphore at the ends of Figures 3b and 3c is the time that is left in the cycle after all threads have consumed their desired CPU time, given as "spare" in the figures' descriptions.

Actually setting a custom stack size N can be done by manually allocating N bytes and passing N and the pointer to the allocated memory to the thread via a `pthread_attr_t` during its creation. For this, one has to

1. initialize the `pthread_attr_t` using `pthread_attr_init` and
2. set the stack using `pthread_attr_setstack`, passing N and the pointer to the stack-memory.

6.2 Stacksize determination

To determine the amount of memory a specific thread's stack uses, a "watermark" can be used, which shows up to (/down to) what byte the thread's stack-memory was used. For this, the thread receives a custom piece of memory for it to use as its stack. This memory is allocated and filled with a recognizable pattern before the threads creation. The amount of allocated memory has to be sufficient for the thread to perform its regular functionality, therefore, using the systems default stacksize is a safe option. On QNX, this is 128 KB according to the documentation.

For the recognlizable pattern with which to fill the memory we chose the 32-bit value `0xAAAAAAAA` (repeating `1`s and `0`s in binary). After filling the memory, it is set as the thread's stack the same way as was described in Section 6.1. This happens in the function `paint_stack`. The thread is then ran for an amount of time that ensures realistic stack usage.

In order to actually determine the amount of used memory, first, the "watermark" has to be found. The memory, the thread used as its stack can be retrieved using `pthread_attr_getstack`. Once this is done, determination of the amount of used memory happens by finding the first byte of the stack-memory, that is no longer equal to `0xAA`. This is done in `find_watermark_bottom_up`, which iterates over the stack-memory, starting at the pointer to the stack-memory until it finds the first 32-bit block that is not equal to `0xAAAAAAAA`. This block thereby contains the watermark, the position of which is determined by some bit-manipulation, which can be seen in the implementation of `find_watermark_bottom_up`, shown in Listing 9.

```
void* find_watermark_bottom_up(void* stackptr, size_t stacksize) {
    bool print_debug_info = false;

    uint64_t* sp = (uint64_t*) stackptr;
    size_t stacksize_64 = stacksize / 8;

    for (size_t i = 0; i < stacksize_64; ++i) {
        // as long as the current 8-Byte-Block is
        // equal to PATTERN, it was not used
        if (sp[i] == PATTERN) {
            continue;
        }
        // first used 8-Byte-Block found, test individual Bytes
        uint64_t mask = 0x00FFFFFFFFFFFFFF; // test lowest 7 Bytes
        for (size_t j = 0; j < 7; ++j) {
            if ((sp[i] & mask) == (PATTERN & mask)) {
                return ((char*)(sp + i)) + (8 - j - 1);
            }
            mask >>= 8;
        }
        return sp + i;
    }
    return stackptr;
}
```

Code Listing 9: find_watermark_bottom_up

The number of bytes n in the stack used by the thread can then be calculated using the pointer to the start of the stack-memory p, the watermark pointer w, the size of the allocated memory in bytes s using $n = s - (w - p)$.

6.3 Results

Using QNX Momentics' "System Summary" View[2], we identified the following improvements regarding stack usage for exercise 4.

By running with a reduced stack size according to the previous measurements, we achieved a reduction of stack-usage of the entire process from 24 KB out of 1044 KB (for an execution without a modification of the stacksize) to 8192 B out of 516 KB. This represents a reduction of about 50% with regards to available stack and a 65% reduction of used stack.

These results present a strong argument for using a custom stacksize in cases where it can be determined beforehand.

7 Minimizing the QNX Kernel

In order to minimize the QNX image, we took the approach of removing drivers that we deemed unnecessary for the functionality required in exercises 2 and 4. We then compared the differences in size of the resulting images.

Audio Drivers
By uncommenting the lines

```
io-audio
wave
mix_ctl
```

Code Listing 10: audio driver in beaglebone.build

[2] Opened by clicking: "Window" → "Show View" → "Other..." → "QNX System Information" → "System Summary"

which include audio drivers in the build image, we managed to reduce the size of the image from an initial 5573 KB to 5239 KB, yielding a decrease by 334 KB. With this configuration, exercises two and four still ran without any issues.

LED drivers

Uncommenting the lines

```
display_msg starting leds driver...
am335x-leds &
waitfor /dev/leds 4
am335x-leds
```

Code Listing 11: LED driver in beaglebone.build

the build image could only be reduced by further 3 KB down to 5236 KB.

SPI, I2C and RTC drivers

Uncommenting the lines

```
display_msg starting I2C driver...
i2c-omap35xx-j5 -i 70 -p0x44E0B000 --u0
waitfor /dev/i2c0
display_msg Setting OS Clock from on-board RTC
rtc -b 0x44e3e000 dm816x
date
display_msg Starting SPI driver...
spi-master -d dm816x base=0x481A0100,irq=125,edma=1,edmairq=555,edmachannel=43
spi-dm816x.so
i2c-omap35xx-j5
rtc
date
spi-master
```

Code Listing 12: SPI I2C and RTC drivers in beaglebone.build

to remove the SPI, I2c and RTC drivers (RTC needs I2C, so we removed it as well) results in a build image 5175 KB in size, meaning another 60 KB were saved. Exercises 2 and 4 still run successfully.

Thoughts

Removing the audio drivers gave us a significantly smaller image while not removing any core functionality, making the change useful for many potential QNX projects.
However, we would not recomment removing the LED, SPI, I2C and RTC drivers, as these offer functionality, that is used in many, if not most, projects related to microcontrollers, while only saving a small amount of memory.