Real-Time Programming

OBJECTIVES

- Learn about basic mechanisms in Real-Time Systems.
- Handle signals: Setup and detect
- Configure and test the Real-Time Clock (RTC).

USEFUL INFORMATION

- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.
- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides for the Terasic DE2i-150 Board.
- Board Setup: Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - ✓ Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.

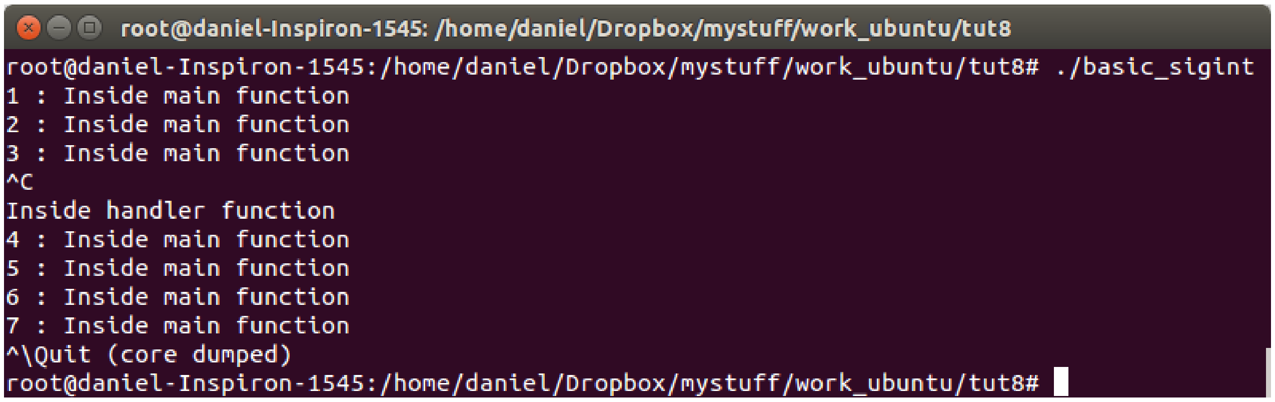
ACTIVITIES

FIRST ACTIVITY: HANDLING SIGNALS

- In this experiment, we will configure signals such that we can execute a ***handler function*** when a signal arrives. We will use both the `SIGINT` ("program interrupt") signal and the `SIGALRM` ("alarm interrupt").
  - ✓ **Setup, then catch a `SIGINT` signal**:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
// Defining handler function
void sig_handler(int signum){
   printf("\nInside handler function\n");
}

int main(){
   int i;
   signal (SIGINT, sig_handler); // Register signal handler
   for(i=1;;i++){ // Infinite loop
      printf("%d : Inside main function\n",i);
      sleep(1); } // Delay for 1 second
   return 0;
}
```

- ▫ First, we set the ***handler function*** `sig_handler` to execute if the `SIGINT` signal arrives.
- ▫ Then, the program is in an infinite loop where the message "`Inside main function`" is printed every 1 second. The user can interrupt by entering ***Ctrl-c*** (this issues the signal `SIGINT`).
  - Usually, ***Ctrl-c*** exits the program. But here, we configured the signal `SIGINT` such that when it arrives it executed the ***function handler*** instead. This function prints the message "`Inside handler function`".
- ▫ To exit the program, you can use the `SIGQUIT` signal (the user enters ***Ctrl-***\).
- ▫ Application file: `basic_sigint.c`
- ▫ Compile this code: `gcc basic_sigint.c -o basic_sigint`
- ▫ Execute this application: `./basic_sigint` ↵
  - Enter ***Ctrl-c*** as many times as you prefer and verify that the function handler was executed. To exit the program, enter ***Ctrl-***\. Fig. 1 shows a sample execution.



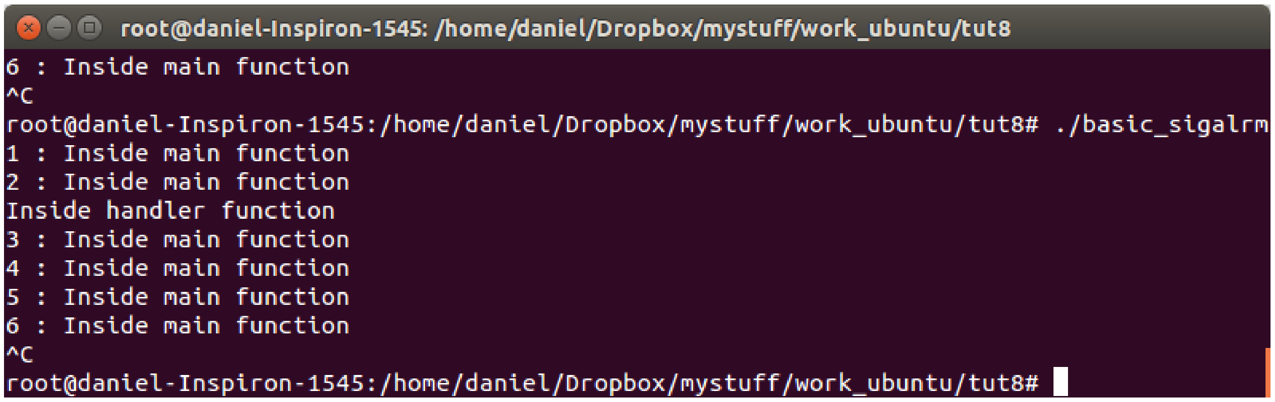
```
root@daniel-Inspiron-1545: /home/daniel/Dropbox/mystuff/work_ubuntu/tut8
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8# ./basic_sigint
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
5 : Inside main function
6 : Inside main function
7 : Inside main function
^\Quit (core dumped)
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8#
```

Figure 1. Setup and catch a SIGINT signal: sample execution.

- ✓ **Setup, then catch a `SIGALRM` signal**:

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

// Defining handler function
void sig_handler(int signum){
  printf("Inside handler function\n");
}

int main(){
   int i;
   signal (SIGALRM,sig_handler); // Register signal handler
   alarm(2);  // Scheduled alarm after 2 seconds
   for(i=1;;i++){  // infinite loop
      printf("%d : Inside main function\n",i);
      sleep(1);  // Delay for 1 second
  }
return 0;
}
```

- First, we set the ***handler function*** `sig_handler` to execute if the `SIGALRM` signal arrives.
- After that, we schedule an alarm to be issued after 2 seconds.
- Then, the program is in an infinite loop where the message "`Inside main function`" is printed every 1 second.
  - After 2 seconds (only one message got to be printed), the alarm is issued (signal `SIGALRM` arrives), causing the ***function_handler*** to be executed. This function prints the message "`Inside handler function`" once.
- To exit the program, you can use the `SIGINT` signal (***Ctrl-c***) or `SIGQUIT` signal (the user enters ***Ctrl-***\).
- Application file: `basic_sigalrm.c`
- Compile this code: `gcc basic_sigalrm.c -o basic_sigalrm`
- Execute this application: `./basic_sigalrm` ↵
  - Observe how the ***function handler*** executes after 2 seconds, and then the infinite loop continues. To exit the program, enter ***Ctrl-c***. Fig. 2 shows a sample execution.


root@daniel-Inspiron-1545: /home/daniel/Dropbox/mystuff/work_ubuntu/tut8

```
6 : Inside main function
^C
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8# ./basic_sigalrm
1 : Inside main function
2 : Inside main function
Inside handler function
3 : Inside main function
4 : Inside main function
5 : Inside main function
6 : Inside main function
^C
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8#
```

Figure 2. Setup and catch a SIGALRM signal: sample execution.

SECOND ACTIVITY: REAL-TIME CLOCK (RTC) CONFIGURATION

- In this experiment, we will deal with the RTC by illustrating how to perform the following steps:
  - ✓ Opening the device (RTC): `/dev/rtc` (in other systems, it is `/dev/rtc0`)
  - ✓ Set, monitor, and wait for Update Interrupts, Alarm Interrupts, and Periodic Interrupts.
  - ✓ Read data from the RTC device.

- To run this application, you are required to be logged as root. For the sake of better explanation, we show three portions of the code along with the respective explanation.

- First portion: Test ***Update Interrupts*** using blocking read as well as blocking select and non-blocking read.

```
// Real Time Clock Driver Test/Example Program. Copyright (C) 1996, Paul Gortmaker.
#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h> // for open()
#include <fcntl.h>
#include <unistd.h> // for read()
#include <stdlib.h>
#include <errno.h>
```

```
static const char default_rtc[] = "/dev/rtc";

int main(int argc, char **argv) {
     int i, fd, retval, irqcount = 0;
     unsigned long tmp, data;
     struct rtc_time rtc_tm;
     const char *rtc = default_rtc;

     switch (argc) {
       case 2: rtc = argv[1]; /* FALLTHROUGH */
       case 1: break;
       default: fprintf(stderr, "usage:  rtctest [rtcdev]\n"); return 1;
     }

   fd = open(rtc, O_RDONLY);  if (fd ==  -1) { perror(rtc); exit(errno); }

   fprintf(stderr, "\n\t\t\tRTC Driver Test Example.\n\n");

   /* Turn on Update Interrupts (one per second) */
   retval = ioctl(fd, RTC_UIE_ON, 0);
   if (retval == -1) {
      if (errno == ENOTTY) { fprintf(stderr, "\n...Update IRQs not supported.\n"); goto test_READ; }
      perror("RTC_UIE_ON ioctl"); exit(errno);
   }

   fprintf(stderr, "Counting 5 update (1/sec) interrupts from reading %s:", rtc); fflush(stderr);

   for (i=1; i<6; i++) {
      retval = read(fd, &data, sizeof(unsigned long)); // Blocking read
      if (retval == -1) { perror("read"); exit(errno); }
      fprintf(stderr, " %d",i); fflush(stderr);
      irqcount++;
   }

   fprintf(stderr, "\nAgain, from using select(2) on /dev/rtc:"); fflush(stderr);

   for (i=1; i<6; i++) {
      struct timeval tv = {5, 0};     /* 5 second timeout on select */
      fd_set readfds;

      FD_ZERO(&readfds);
      FD_SET(fd, &readfds);
      /* The select will wait until an RTC interrupt happens. */
      retval = select(fd+1, &readfds, NULL, NULL, &tv); // Blocking select. Waits till interrupt
      if (retval == -1) { perror("select"); exit(errno); }

      retval = read(fd, &data, sizeof(unsigned long)); // Non-blocking read ('cause of select())
      if (retval == -1) { perror("read"); exit(errno); }
      fprintf(stderr, " %d",i); fflush(stderr);
      irqcount++;
   }

   /* Turn off update interrupts */
   retval = ioctl(fd, RTC_UIE_OFF, 0); if (retval == -1) { perror("RTC_UIE_OFF ioctl"); exit(errno); }
```

✓ The steps here can be summarized as:
- ▫ Open the RTC (`/dev/rtc`) device: `fd = open("/dev/rtc", O_RDONLY)`
- ▫ Turn on Update interrupts (one per second): `retval = ioctl(fd, RTC_UIE_ON, 0);`
- ▫ Count 5 Update Interrupts: Execute 5 blocking `read()` on RTC. Each read waits until an interrupt is received.
- ▫ Count 5 Update Interrupts: Execute 5 blocking `select()` on RTC, each time perform a non-blocking `read()` on RTC.
- ▫ Turn off update interrupts: `ioctl(fd, RTC_UIE_OFF,0)`

- Second portion: Read data from RTC (this is different from using the `read()` system call) and test *Alarm Interrupts*.

```
test_READ:
   /* Read the RTC time/date */
   retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
   if (retval == -1) { perror("RTC_RD_TIME ioctl"); exit(errno); }

   fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d, %02d:%02d:%02d.\n",
            rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
            rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

   /* Set the alarm to 5 sec in the future, and check for rollover */
   rtc_tm.tm_sec += 5;
   if (rtc_tm.tm_sec >= 60) { rtc_tm.tm_sec %= 60; rtc_tm.tm_min++; }
   if (rtc_tm.tm_min == 60) { rtc_tm.tm_min = 0; rtc_tm.tm_hour++; }
   if (rtc_tm.tm_hour == 24) rtc_tm.tm_hour = 0;

   retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
   if (retval == -1) {
      if (errno == ENOTTY) { fprintf(stderr, "\n...Alarm IRQs not supported.\n"); goto test_PIE; }
      perror("RTC_ALM_SET ioctl"); exit(errno);
   }

   /* Read the current alarm settings */
   retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
   if (retval == -1) { perror("RTC_ALM_READ ioctl"); exit(errno); }

   fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n",
            rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

   /* Enable alarm interrupts */
   retval = ioctl(fd, RTC_AIE_ON, 0);
   if (retval == -1) { perror("RTC_AIE_ON ioctl"); exit(errno); }
   fprintf(stderr, "Waiting 5 seconds for alarm..."); fflush(stderr);

   /* This blocks until the alarm ring causes an interrupt */
   retval = read(fd, &data, sizeof(unsigned long)); // Blocking read
   if (retval == -1) { perror("read"); exit(errno); }
   irqcount++;
   fprintf(stderr, " okay. Alarm rang.\n");

   /* Disable alarm interrupts */
   retval = ioctl(fd, RTC_AIE_OFF, 0); if (retval == -1) { perror("RTC_AIE_OFF ioctl"); exit(errno); }
```

- ✓ The steps here can be summarized as:
  - Read the RTC time/date (`rtc_tm` is a `rtc_time` structure): `ioctl(fd, RTC_RD_TIME,&rtc_tm)`
    - The RTC's time is organized in a structure (`rtc_tm`) of type `rtc_time`.

```
struct rtc_time {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday; /* unused */
   int tm_yday; /* unused */
   int tm_isdst;/* unused */
```

  - Set the alarm to 5 seconds in the future:
    - First, the structure `rtc_tm` elements are updated to 5 seconds.
    - Then, the alarm is set: `ioctl(fd, RTC_ALM_SET, &rtc_tm)`
  - Read current alarm settings: `retval = ioctl(fd, RTC_ALM_READ, &rtc_tm)`
  - Enable Alarm Interrupt: `retval = ioctl(fd, RTC_AIE_ON, 0)`
  - Wait until Alarm Interrupt comes by executing a blocking `read()` on RTC.
  - Disable alarm interrupts: `retval = ioctl(fd, RTC_AIE_OFF, 0)`

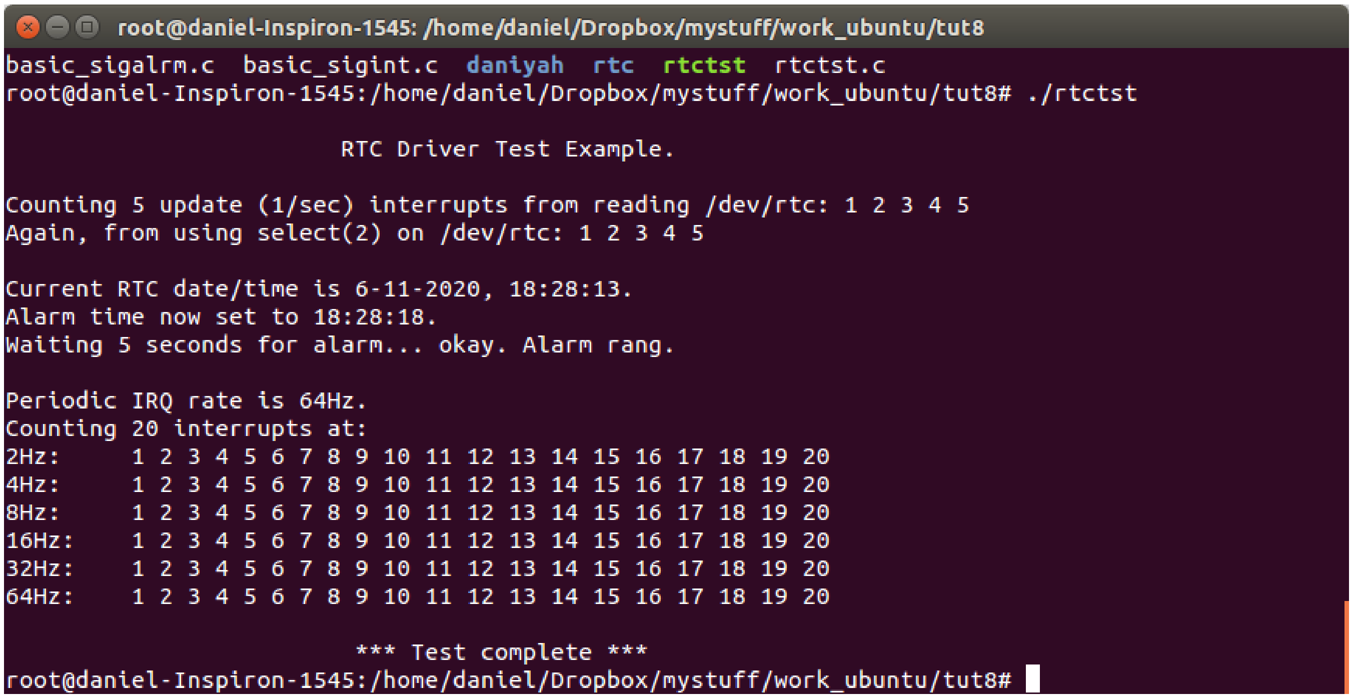
- Third portion: Configure and test ***Periodic Interrupts***.

```
test_PIE:
   /* Read periodic IRQ rate */
   retval = ioctl(fd, RTC_IRQP_READ, &tmp);
   if (retval == -1) {
      /* not all RTCs support periodic IRQs */
      if (errno == ENOTTY) { fprintf(stderr, "\nNo periodic IRQ support\n"); goto done; }
      perror("RTC_IRQP_READ ioctl"); exit(errno);
   }

   fprintf(stderr, "\nPeriodic IRQ rate is %ldHz.\n", tmp);
   fprintf(stderr, "Counting 20 interrupts at:"); fflush(stderr);

   /* The frequencies 128Hz, 256Hz, ... 8192Hz are only allowed for root. */
   for (tmp=2; tmp<=64; tmp*=2) {
      retval = ioctl(fd, RTC_IRQP_SET, tmp);
      if (retval == -1) {
         /* not all RTCs can change their periodic IRQ rate */
         if (errno == ENOTTY) { fprintf(stderr, "\n...Periodic IRQ rate is fixed\n"); goto done; }
         perror("RTC_IRQP_SET ioctl"); exit(errno);
      }
      fprintf(stderr, "\n%ldHz:\t", tmp); fflush(stderr);
      /* Enable periodic interrupts */
      retval = ioctl(fd, RTC_PIE_ON, 0); if (retval == -1) { perror("RTC_PIE_ON ioctl"); exit(errno);}

      for (i=1; i < 21; i++) {
         retval = read(fd, &data, sizeof(unsigned long)); // Blocking Read
         if (retval == -1) { perror("read"); exit(errno); }
         fprintf(stderr, " %d",i); fflush(stderr);
         irqcount++;
      }

      /* Disable periodic interrupts */
      retval = ioctl(fd, RTC_PIE_OFF, 0); if (retval==-1) { perror("RTC_PIE_OFF ioctl"); exit(errno);}
   }

done:
     fprintf(stderr, "\n\n\t\t\t *** Test complete ***\n");
     close(fd);
     return 0;
}
```

- ✓ The steps here can be summarized as:
  - Read periodic IRQ rate: `retval = ioctl(fd, RTC_IRQP_READ, &tmp)`
  - For a set of periodic interrupts (from 2 to 64 Hz, only powers of 2), do: set the frequency, enable period interrupts, detect a number of interrupts (20) of a given frequency, and disable periodic interrupts.
    - Set frequency (given by `tmp: 2, 4, 8, 16, 32, 64`): `reval = ioctl(fd, RTC_IRQP_SET, tmp)`
    - Enable Periodic Interrupts: `retval = ioctl(fd, RTC_PIE_ON, 0)`
    - Use a number (20) of blocking `read()` to wait until a number of periodic interrupts (20) is detected.
    - Disable periodic interrupts: `retval = ioctl(fd, RTC_PIE_OFF, 0)`

- Application file: `rtctst.c`
- Compile this code: `gcc rtctst.c -o rtctst`
- Execute this application: `sudo -i ↵`
  `./rtctst ↵`

- Fig. 3 shows an execution example.



```
root@daniel-Inspiron-1545: /home/daniel/Dropbox/mystuff/work_ubuntu/tut8
basic_sigalrm.c  basic_sigint.c  daniyah  rtc  rtctst  rtctst.c
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8# ./rtctst

                        RTC Driver Test Example.

Counting 5 update (1/sec) interrupts from reading /dev/rtc: 1 2 3 4 5
Again, from using select(2) on /dev/rtc: 1 2 3 4 5

Current RTC date/time is 6-11-2020, 18:28:13.
Alarm time now set to 18:28:18.
Waiting 5 seconds for alarm... okay. Alarm rang.

Periodic IRQ rate is 64Hz.
Counting 20 interrupts at:
2Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
8Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
16Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
64Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

                         *** Test complete ***
root@daniel-Inspiron-1545:/home/daniel/Dropbox/mystuff/work_ubuntu/tut8#
```

Figure 3. Sample execution: RTC Driver Test.