OpenShoe runtime framework

John-Olof Nilsson and Isaac Skog

December 12, 2011

Abstract

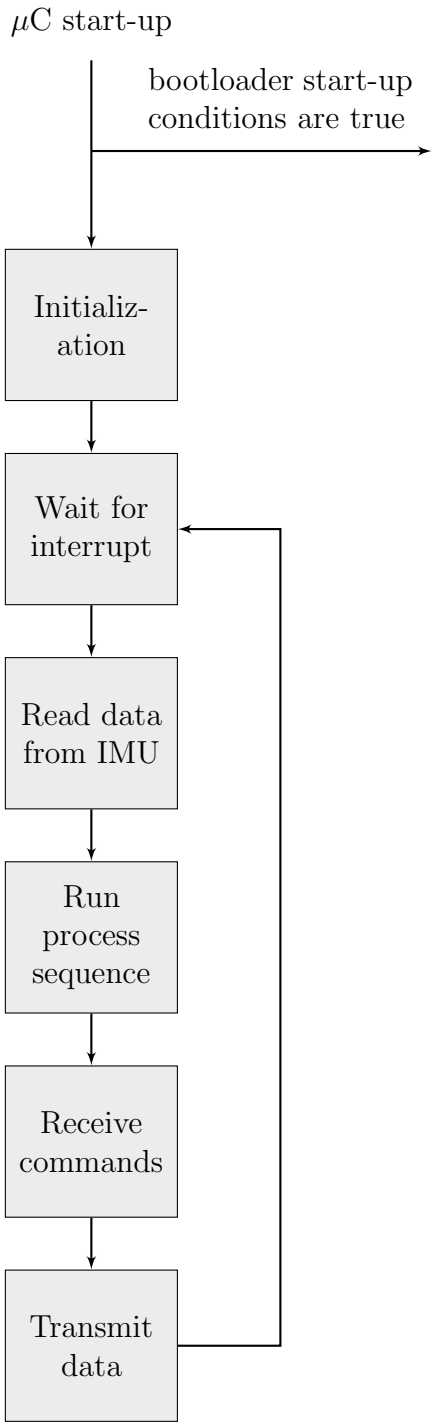
This documents describes the OpenShoe runtime framework and some underlaying functionalities. The framework contains the main execution loop of the OpenShoe microcontroller software and is responsible for receiving driving interrupts and triggered by this calling procedures for: 1) reading data from the IMU, 2) do all data processing, 3) receiving commands, and 4) transmitting replays and data.

1 Introduction

The OpenShoe runtime framework contains the main execution loop of the OpenShoe microcontroller software and is directly or indirectly responsible for calling all runtime routines of all other OpenShoe microcontroller software components. All other microcontroller software components are either directly or indirectly linked into the OpenShoe runtime framework. The execution is driven by interrupts which are periodically received from the IMU. Following an interrupt the OpenShoe runtime framework calls a static set of routines provided by other software components.

2 Flow chart

A flow chart of the execution of the OpenShoe runtime framework together with the corresponding main function c-code is found below. The functionality of the flow chart boxes is explained in the following sections.




```
int main (void) {

  system_init(void);



  while (true) {

    wait_for_interrupt();



    imu_burst_read();



    run_process_sequence();



    receive_command();



    transmit_data();
  }
}
```

3 Initialization

The OpenShoe runtime framework will call the initialization routines of the other software components. The framework itself sets up and contain functionality for receiving driving interrupts from the IMU. The setup is done based on macro definitions in configuration files containing the pin-out of the microcontroller and the routing of the OpenShoe PCB.

Initialization for the processing is most likely not executed directly in the main loop but rather in the *run process sequence*. This has to do with that processing will often be started based on user commands rather than statically in the program. Also, the processing might have to reinitialize which have to be done in the *run process sequence* since there is no way to return to the initialization code block.

4 Interrupts

Interrupts are periodically received from the IMU. The connection between the microcontroller hardware interrupt and the pin on which the IMU will send the interrupts are configured in the initialization section. The interrupt is of non-maskable type and cannot be ignored or blocked. The simple interrupt exception routine is shown below

```
void eic_nmi_handler( void )
{
// Save registers not saved upon NMI exception.
__asm__ __volatile__ ("pushm   r0-r12, lr\n\t");

eic_clear_interrupt_line(&AVR32_EIC, IMU_INTERUPT_LINE1);
imu_interrupt_ts = Get_system_register(AVR32_COUNT);
imu_interupt_flag = true;

// Restore the registers and leaving the exception handler.
__asm__ __volatile__ ("popm   r0-r12, lr\n\t" "rete");
}
```

The routine gets a clock register reading and sets the interrupt flag. No extensive processing should be done within the interrupt routine since USB communication will be blocked during the execution of the routine.

The `wait_for_interrupt` routine will wait in an infinite loop for the `imu_interrupt_flag` to be set. Once the flag is set it will toggle it back and return.

5 Read data from IMU

The `read_data_from_IMU` routine read off the appropriate IMU memory areas, parses and transform the data to SI-units, and update global variables used to communicate the data with the other functions.

6 Run process sequence

The `run_process_sequence` routine cycles through an array of processing function pointers which can configured via user commands. These functions will be responsible for doing all the data processing such as the processing for the ZUPT-aided INS, processing initialization routines, system calibration routines, etc. Whatever regular data processing that the user wants the system to do should be added as functions pointers in the process sequence array.

The process sequence array can be configured by the functions in it. This is often used by clean-up functions which are placed in the end of the sequence and which will halt some processing if some conditions are met.

7 Receive commands

The `receive_commands` routine parses data from the USB communication buffer and executes command response. Such command responses will typically add functions to the process sequence array or set some other parameters or flags in the system.

8 Transmit data

The `transmit_data` routine will output command responses when such are needed. The function will also output system data such as state estimates

and measurements data. This data will be output based on some data rate divider which divides the interrupt frequency.