BTN415 Term Project, Winter 2017

Milestone #1

In this milestone you will create a **PktDef** class that defines and implements the robots application layer protocol. Your class implementation can be tested using the ***milestone1.cpp*** file provided.

PROCEDURES

Application Layer Protocol Definition

COMMANDS

The application layer protocol contains three main components. A header which contains an unsigned integer **PktCount,** the following command bit-field flags **Drive, Status, Sleep, Arm, Claw, and Ack. Make sure to add** two bits of padding after Ack complete 1-byte of space. Also, it should have a packet byte (unsigned char) **Length**, a a pointer of type char called **Data**, and a tail with a **1-Byte (unsigned char) CRC** validation. Definitions of these elements are as follows:

- **PktCount** – contains an integer number that is constantly incrementing each time a packet is transmitted between the client and the robot.
- Command Flags:
  - **Drive** – set to a value of 1 if the command is a **DRIVE** command
  - **Status** – set to a value of 1 if a response packet with sensor telemetry
  - **Sleep** – set to a value of 1 if the command is a **SLEEP** command
  - **Arm** – set to a value of 1 if the command is an **ARM** command
  - **Claw** – set to a value of 1 if the command is a **CLAW** command
  - **Ack** – set to a value of 1 if the command is an acknowledgement packet. I.e., after receiving a **Drive**, **Arm**, **Claw** or **Sleep** command, the robot will send a **Status** command with **Ack** set to 1, as well as the bit corresponding to the command also set to 1.

    NOTE: Drive, Arm, Claw and Sleep flags should never be set at the same time. The Ack flag should always be set with a corresponding command flag.
- **Length** – contains an unsigned char with the total number of bytes in the packet
- **MotorBody** – the drive command parameters are placed in the body of the command packet if the **Drive** flag is set to 1.
  - **Direction** – the drive command directive value
  - **Duration** – the number of seconds to execute the direction directive
- **CRC** – the packet validation value to ensure correct transmission

The following is a visual representation a size allocation of the application layer protocol for the mobile robot:

Motor Commands (Includes Drive, Arm and Claw operations):

Packet Header									Packet Body	Packet Trailer
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**MotorBody**	**CRC**
4-bytes	1-bit	1-bit	1-bit	1-bit	1-bit	1-bit	2-bits	1-byte	2-bytes	1-byte

MotorBody	
Direction	**Duration**
1-byte	1-byte

Sleep Commands (note that for this command, the Length will be 0. Hence, the body is empty):

Packet Header									Packet Trailer
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**CRC**
4-bytes	1-bit	1-bit	1-bit	1-bit	1-bit	1-bit	2-bits	1-byte	1-byte

Drive Command Parameter Definitions

The **Drive** command parameters (**MotorBody)** have a pre-defined value associated with the direction. These drive command directions are defined as follows:

- FORWARD 1
- BACKWARD 2
- RIGHT 3
- LEFT 4

The duration parameter of the **MotorBody** for Drive commands holds an *unsigned int* value representing the number of seconds to execute the command for.

Arm Command Parameter Definitions

The **Arm** command parameters (**MotorBody)** have a pre-defined value associated with the direction. These drive command directions are defined as follows:

- UP 5
- DOWN 6

The duration parameter of the **MotorBody** for Arm commands holds a value of zero (0).

Claw Command Parameter Definitions

The **Claw** command parameters (**MotorBody)** have a pre-defined value associated with the direction. These drive command directions are defined as follows:

- OPEN 7
- CLOSE 8

The duration parameter of the **MotorBody** for Claw commands holds a value of zero (0).

RESPONSES

Acknowledgement Response

The robot will validate and acknowledge every command transmitted to it. The following is a visual representation of an Acknowledgement packet from the robot. Note that, once again MotorBody has a length of 0.:

Packet Header									Packet Trailer
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**CRC**
4-bytes	1-bit	1-bit	1-bit	1-bit	1-bit	1-bit	2-bits	1-byte	1-byte
<value>	1	0	0	0	0	1	0	7	<value>

Negative Acknowledgement Response

If the robot rejects the commands packets CRC, it will transmit a Negative Acknowlegement Packet, commonly referred to as a NACK. The following is a visual representation of a Negative Acknowledgement packet from the robot. Note that, once again MotorBody has a length of 0.:

Packet Header									Packet Trailer
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**CRC**
4-bytes	1-bit	1-bit	1-bit	1-bit	1-bit	1-bit	2-bits	1-byte	1-byte
<value>	0	0	0	0	0	0	0	7	<value>

Where all Packet header flags are set to zero (), including the ACK bit and the PktCount is the same value as the commands PktCount.Telemetry Response

The response message from the mobile robot will use the same header and trailer definition as the command. The header will have the **Status** bit set to a value of 1 and the body of the message will be populated with the sensory information.

The following is a visual representation of a response packet:

Packet Header									Packet Body	Packet Trailer
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**RAW Data**	**CRC**
4-bytes	1-bit	1-bit	1-bit	1-bit	1-bit	1-bit	2-bits	1-byte	N-bytes	1-byte

Parity Algorithm & Example

The 1–byte CRC parity check performed by the robot is a simple count on the number of **BITS** set to ‘1’. For example:

Packet Header									Packet Body	Packet Trailer	
PktCount	**Drive**	**Status**	**Sleep**	**Arm**	**Claw**	**Ack**	**Padding**	**Length**	**MotorBody**	**CRC**	
00000000 00000000 00000000 00000001	1	0	0	0	0	1	00	00001001	00000001 00001010	00001000	Binary
1	1	0	0	0	0	1	0	9	1,10	8	Decimal

Class PktDef Requirements

For your Pkt_Def class, you should have the following defined:

- A structure **Header** which contains the header information based on the description above
- A structure **MotorBody** which contains the drive parameter information based on the description above
- An enumerated **CmdType** to define the command types {DRIVE, SLEEP, ARM, CLAW, ACK}
- The following constant integer definitions, matching the values previously presented:
  - FORWARD
  - BACKWARD
  - LEFT
  - RIGHT
  - UP
  - DOWN
  - OPEN
  - CLOSE
  - HEADERSIZE ← represents the size of the **Header** in bytes (must be calculated by hand)

Your class PktDef should contain, as a minimum, the following:

- A private structure to define a **CmdPacket**

- **Header**
- **char * Data**
- **char CRC**

- **A char *RawBuffer that will store all data in PktDef in a serialized form that can be used to transmit it over TCP/IP**
- The following member functions:
  - **PktDef()** – A default constructor that places the **PktDef** object in a safe state, defined as follows:
    - All Header information set to zero
    - Data pointer set to nullptr
    - CRC set to zero
  - **PktDef(char *)** – An overloaded constructor that takes a RAW data buffer, parses the data and populates the Header, Body, and CRC contents of the **PktDef** object.
  - **void SetCmd(CmdType)** – A set function that sets the packets command flag based on the **CmdType**
  - **void SetBodyData(char *, int)** – a set function that takes a pointer to a RAW data buffer and the size of the buffer in bytes. This function will allocate the packets **Body** field and copies the provided data into the objects buffer
  - **void SetPktCount(int)** – a set function that sets the objects **PktCount** header variable
  - **CmdType GetCmd()** – a query function that returns the **CmdType** based on the set command flag bit
  - **bool GetAck()** – a query function that returns True/False based on the **Ack** flag in the header
  - **int GetLength()** – a query function that returns the packet **Length** in bytes
  - **char *GetBodyData()** – a query function that returns a pointer to the objects **Body** field
  - **int GetPktCount()** – a query function that returns the **PktCount** value
  - **bool CheckCRC(char *, int)** – a function that takes a pointer to a RAW data buffer, the size of the buffer in bytes, and calculates the CRC. If the calculated CRC matches the CRC of the packet in the buffer the function returns TRUE, otherwise FALSE.
  - **void CalcCRC()** – a function that calculates the CRC and sets the objects packet CRC parameter.
  - **char *GenPacket()** – a function that allocates the private **RawBuffer** and transfers the contents from the objects member variables into a RAW data packet (**RawBuffer**) for transmission. The address of the allocated **RawBuffer** is returned.

You can download the Milestone1.cpp file from the course blackboard (or instructor’s website).

```
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include "Pkt_Def.h"
using namespace std;

int main()
{
	MotorBody DriveCmd;
	DriveCmd.Direction = FORWARD;
	DriveCmd.Duration = 20;

	PktDef TestPkt;
	char *ptr;

	//Testing the PktDef creation interface
	TestPkt.SetCmd(DRIVE);
	TestPkt.SetBodyData((char *)&DriveCmd, 2);
	TestPkt.SetPktCount(1);
	TestPkt.CalcCRC();
	ptr = TestPkt.GenPacket();

	cout << showbase
		<< internal
		<< setfill('0');

	for (int x = 0; x < (int)TestPkt.GetLength(); x++)
		cout << hex << setw(4) << (unsigned int)*(ptr++) << ", ";

	cout << endl;
	TestPkt.SetCmd(ACK);
	TestPkt.CalcCRC();
	ptr = TestPkt.GenPacket();

	for (int x = 0; x < (int)TestPkt.GetLength(); x++)
		cout << hex << setw(4) << (unsigned int)*(ptr++) << ", ";

	cout << endl << noshowbase << dec;
	//Testing Rx Buffer interface
	//You should create RAW data packets (like below) to test your overloaded constructor
	char buffer[9] = { 0x02, 0x00, 0x00, 0x00, 0x02, 0x09, 0x11, 0x24, 0x08};
	PktDef RxPkt(buffer);
	cout << "CommandID:  " << RxPkt.GetCmd() << endl;
	cout << "PktCount:   " << RxPkt.GetPktCount() << endl;
	cout << "Pkt Length: " << RxPkt.GetLength() << endl;
	cout << "Body Data: " << endl;

	ptr = RxPkt.GetBodyData();
	cout << showbase << hex;
	cout << "Byte 1 " << (int)*ptr++ << endl;
	cout << "Byte 2 " << (int)*ptr << endl;

	return 1;
}
```