Anteros Labs Inc

Anteros Labs Micro-Car (ALMC) User Manual

[Type the document subtitle]

3/24/2010

Anteros Labs

Table of Contents

List of Tables

List of Figures

Revision History

Date	Revision No.	AL-SVN Rev. No.	Description
03/24/2010	1.1	58	Started tracking revision history.

About This Document

Abbreviation, Notation, and Documentation Conventions

Notation	Hardware or Software?	Description
ALMC	Both	Anteros Lab Micro-Car
ALMC-100	H	ALMC lower board, housing an LM3S818 microcontroller and other peripherals
ALMC-101	H	ALMC lower board programming board
ALMC-111	H	ALMC upper board, housing an XC4VFX12 FPGA
ALMC-112	H	ALMC camera base board
LBC	S	ALMC Lower Board Controller
UBC	S	ALMC Upper Board Controller

1. Introduction

1.1 Features

- **Processors:**
  - 350MHz-rated Virtex-4 FPGA for on-board image processing and user interface
  - 50MHz ARM Cortex-M3 microcontroller for motion control
- **Memory:**
  - Fx12 Mini Module features external 64MB DDR SDRAM and 4MB flash
  - Microcontroller features internal 8KB SRAM and 64KB flash
  - 1KB (16-bit) EEPROM for storing control parameters
- **Sensors:**
  - A 640x480 camera
  - 140Hz analog gyroscopes for on-board orientation sensing
  - 0.45° resolution optical encoder for on-board speed sensing
- **Communications:**
  - 2 serial radio modules dedicated to positioning and inter-vehicle communication
  - Each can transmit and receive at up to 115200 bps
- **Drive and steering system:**
  - Rear wheel drive with speed up to (TBA) m/s
  - Axle-articulated steering system with +/- 30° range
- **Power:**
  - 4 AAA batteries with at least 30 min runtime
  - Battery voltage-level warning system
- **Dimensions:**
  - 4in x 2in x 2in
  - 6.89 Oz

1.2 Overview

The ALMCs are palm-size wireless autonomous vehicles designed for the robotic testbed in UCLA Applied Math Lab (AML). They can be programmed to perform group maneuvers and carry out cooperative tasks, which are of high interest in today's cooperative control research.

The AML robotic testbed currently features a 2m x 1.5m arena and a camera-based overhead tracking system that can detect and identify tag-wearing micro-vehicles in real-time (30Hz). Through a dedicated wireless serial link, the ALMCs can receive broadcasted positioning information from the tracking system. Through a separate wireless link featuring Carrier Sensing Multiple Access (CSMA), the vehicles can achieve peer-to-peer communication. The vehicles are also built with a front-facing camera, two gyroscopes providing pitch, roll, and yaw information, and an optical encoder (attached to the drive motor) providing accurate speed sensing. All these real-time sensing capabilities are integrated into a fairly compact chassis, thus allowing users to achieve complex cooperative control in a relatively small area. This is immensely valuable to users who wish to verify abstract control theories in a real environment but are hindered by lack of physical space.

Electronic components of the vehicle are divided into two layers (two PCBs, see **Figure 1-2** through **Figure 1-4**), which are stacked vertically to form the main body of the chassis. The upper layer features the FX12 Mini Module with a Xilinx Virtex-4 FPGA, a 640x480 camera, and a serial wireless communication module. The lower board houses a 50MHz ARM Cortex-3 microcontroller, a separate serial wireless module, and two high performance gyroscopes. Batteries and the vehicle's drive and steering systems are also mounted on or attached to the lower board. While the lower board alone is fully capable of controlling the vehicle's motion, the upper board adds powerful processing, larger memories, enriched sensing, inter-vehicle communication, and simplified user interface. Such a design achieves a good balance in terms of minimizing vehicle footprint and center of gravity.

Four rechargeable AAA (4.8V 800mAH) batteries form the power source of the vehicle. The vehicle can continue to operate until the serial battery voltage drop below 4.2V. Under normal usage load, a single charge should last 30-60min (This number is only an estimate based on power consumptions of major components. It may be revised once field trials are conducted).

The lower board microcontroller would come loaded with driver software to perform basic drive and steering control, access on-board sensors, receive positioning information, and communicate with the upper board FPGA. To program the vehicles, the user would mainly operate within the FPGA. Simple examples that illustrate major capabilities plus detailed documentation of user-accessible components will be provided also.

1.3 System Diagram

Major electronic components of ALMC and their interfaces are illustrated as below:

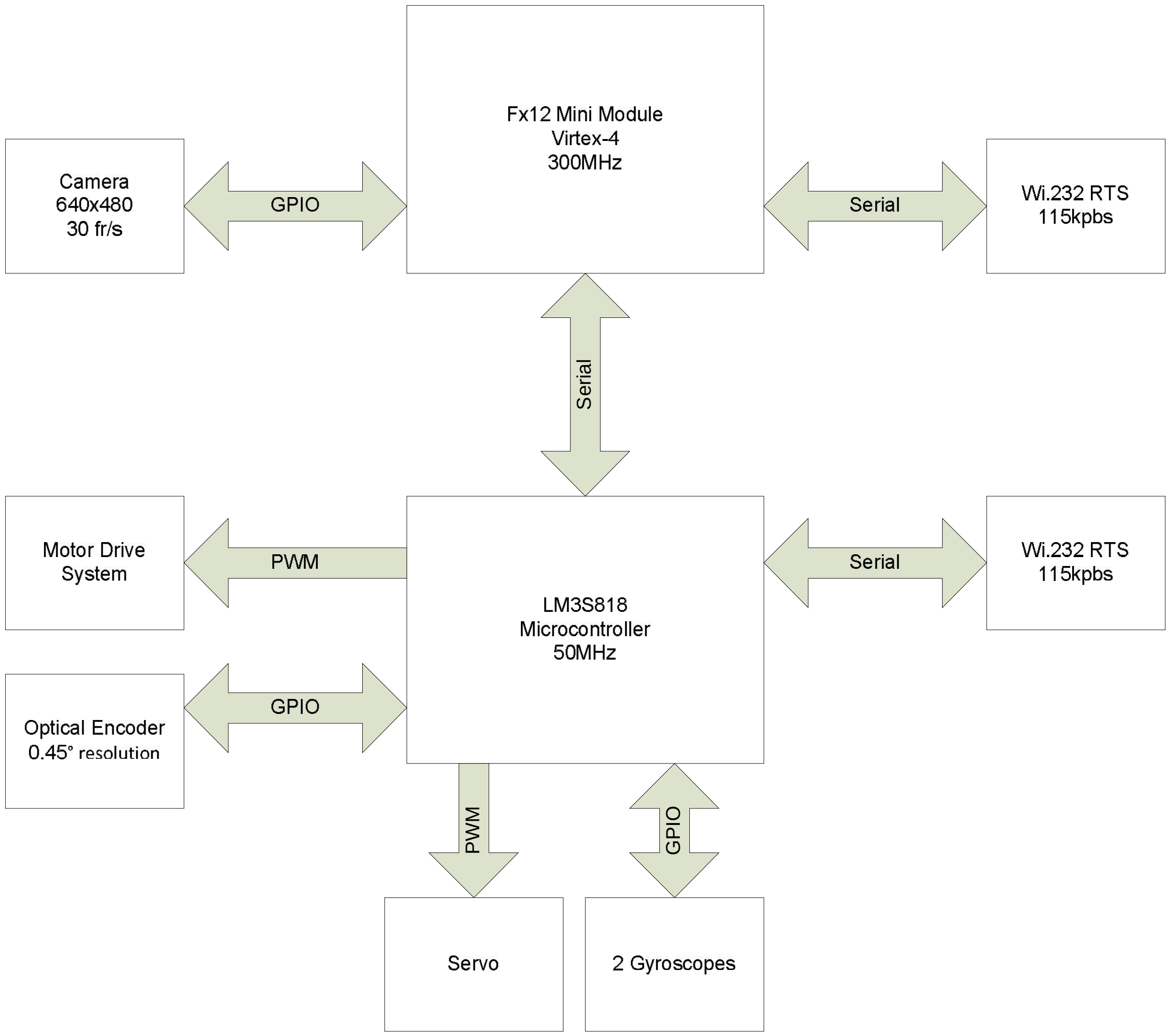


Figure 1-1: System Diagram

1.4 Mechanical Specifications

1.4.1 ALMC Physical Models and Component Locations

The following three figures show a 3D model of ALMC from various angles, each with dimension and component labels.

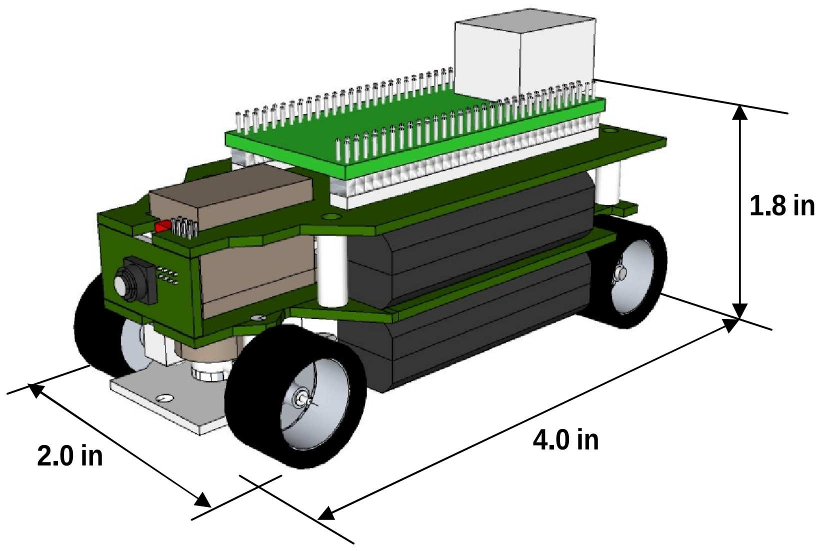
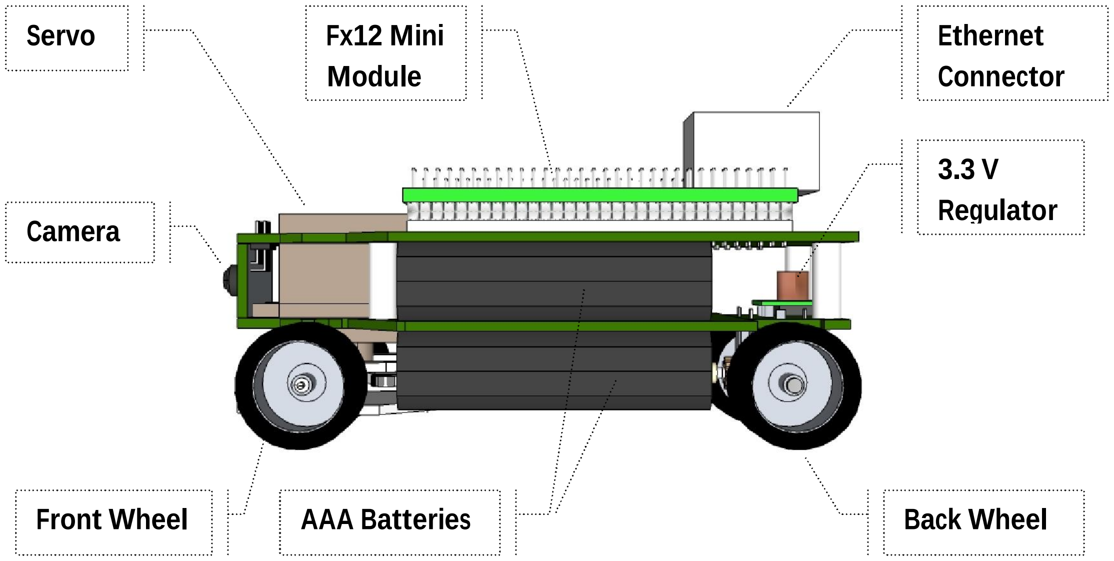


Figure 1-2: Isometric view of an ALMC and its physical dimensions

Figure 1-3: Side view of an ALMC plus visible component locations



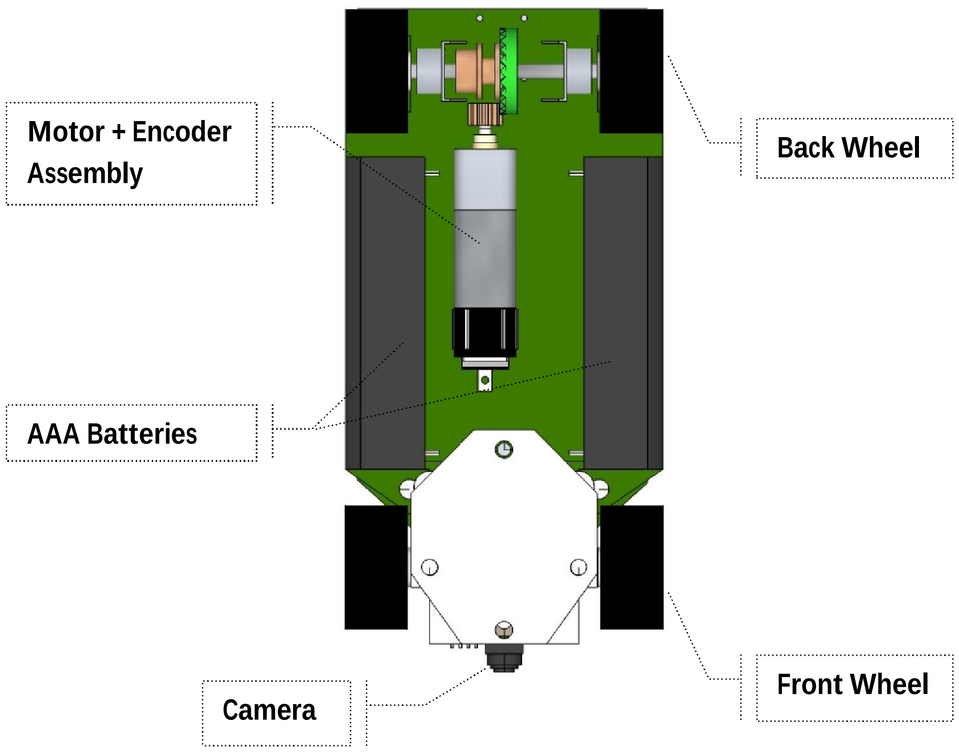


Figure 1-4: Bottom view of an ALMC plus visible component locations

1.4.2 Weight

195.4g or 6.89Oz (including long-range sensor, but no tracking tag)

1.4.3 Turning radius spec and speed profiles

To be specified

1.5 Communication Specifications

1.5.1 Overhead Tracking Broadcast

The ALMCs are designed to work with an existing overhead tracking system, which broadcasts positions and orientation information of all the vehicles present on the arena. In addition to its own information, each vehicle can also extract other vehicles' from the broadcast when required by the application. The broadcast is updated at the rate of the overhead camera (30 Hz). The radio modules can transmit and receive at up to 115200 bps. This would easily allow dozens of vehicles' information to be broadcasted, which well exceeds the current spatial capacity of the arena.

1.5.2 Inter-Vehicle or Vehicle-User Communication

The upper board features another radio module that's directed connected to the FPGA. This module is dedicated to inter-vehicle communication and vehicle-user communication. This module would be configured to operate at a different channel that the tracking radio to avoid interference. It can also transmit and receive at up to 115200 bps. This module also features CSMA medium access control, which would effectively minimize packet losses due to concurrent transmissions at the physical level. However to achieve efficient sharing of information among vehicles, the users would still need to device an application/transport layer protocol that's suitable to the application.

Anteros Labs

1.6 Electrical Characteristics

1.6.1 Absolute Maximum Ratings

Components	Characteristics	Min	Max	Unit
Lower Board	Supply Voltage	-0.6	12.1	V
LM3S818 Microcontroller	Operating Temperature Range	-40	85	°C
FX12 Mini Module	I/O Header Pin Voltage	-0.85	4.4	V

Table 1-1: Absolute Maximum Ratings

1.6.2 Recommended Operating Conditions

Components	Characteristics	Min	Typical	Max	Unit
Lower Board	Supply Voltage	4.4		6.0	V
LM3S818 Microcontroller	Operating Temperature Range			69	°C
FX12 Mini Module	I/O Header J1-19 ~J1-22		VCOO1		V
	I/O Header J1-49 ~J1-64		VCOO1		V
	I/O Header J2-49 ~J2-64		3.3		V

Table 1-2: Recommended Operating Conditions

1.7 Register Table

System control parameters that are unique to each vehicle while relatively stable over time are stored in a 1Kb EEPROM on the lower board. The EEPROM is divided into 64 16-bit registers. The following table shows the current designated usage of these registers. Registers that are left blank are not yet occupied.

Table 1-3: EEPROM Registers

Register Name	Address	Description
RegCID	0x00	Car ID
	0x01	Not in Use
	0x02	Not in Use
	0x03	Not in Use
RegMERR	0x04	Motor Encoder Count to Revolution Ratio
RegMGR	0x05	Motor Gear Ratio
RegMPG	0x06	Motor Proportional Gain
RegMIG	0x07	Motor Integral Gain
RegMDG	0x08	Motor Differential Gain
RegMDB	0x09	Motor Dead Band
RegSO	0x0A	Servo Offset
RegSG	0x0B	Servo Gain
RegSPG	0x0C	Servo Proportional Gain
RegSIG	0x0D	Servo Integral Gain
RegSDG	0x0E	Servo Differential Gain
RegSDB	0x0F	Servo Dead Band
RegGPO	0x10	Gyro Pitch ZRL (offset)
RegGPG	0x11	Gyro Pitch Sensitivity (gain)
RegGRO	0x12	Gyro Roll ZRL (offset)
RegGRG	0x13	Gyro Roll Sensitivity (gain)
RegGYO	0x14	Gyro Yaw ZRL (offset)
RegGYG	0x15	Gyro Yaw Sensitivity (gain)
RegRO	0x16	Range LP Offset
RegRG	0x17	Range LP Gain
	0x18	
	0x19	
	0x1A	
	0x1B	
	0x1C	
	0x1D	
	0x1E	
	0x1F	

	0x20	User Data
	0x21	User Data
	0x22	User Data
	0x23	User Data
	0x24	User Data
	0x25	User Data
	0x26	User Data
	0x27	User Data
	0x28	User Data
	0x29	User Data
	0x2A	User Data
	0x2B	User Data
	0x2C	User Data
	0x2D	User Data
	0x2E	User Data
	0x2F	User Data
	0x30	User Data
	0x31	User Data
	0x32	User Data
	0x33	User Data
	0x34	User Data
	0x35	User Data
	0x36	User Data
	0x37	User Data
	0x38	User Data
	0x39	User Data
	0x3A	User Data
	0x3B	User Data
	0x3C	User Data
	0x3D	User Data
	0x3E	User Data
	0x3F	User Data

2. Lower Board Controller

The ALMC Low Board Controller (LBC) bears the following two main responsibilities:

1. Control the vehicle's motion by providing PWM signals that drive the drive motor and the servo;
2. Provide sensory data and system information the user is interested in.

To accomplish these two goals, a multitasking system is necessary to access sensors, generate output signals, and interface users (via the Upper Board Controller) in a parallel fashion. We decided to use FreeRTOS as the operating system for the LBC, where operations can be carried out by parallel tasks. In FreeRTOS, a task is basically a thread that has periodical chances of execution. Different tasks can run at different frequencies (so long it doesn't exceed the maximum frequency FreeRTOS allows). Each task is also assigned a priority at the point of creation. Thus when multiple tasks need to run at the same time, the order to run is sorted according to their priorities (and possibly their order of creation for tasks with the same priority).

Since the LBC interacts with multiple software and hardware components at different rates, we need to divide the above two main tasks into smaller ones in the interest of efficiency and effectiveness. Thus we've identified a series of tasks necessary to accomplish the motion control goal. We'll first use signal flow diagrams to illustrate their interdependence and relations to hardware peripherals through a signal flow diagram (**Figure 2-1** and **Figure 2-2**). In the next chapter, we'll provide in-depth description of each task and discuss how we determine their frequencies and priorities.

For tasks related to motion control:

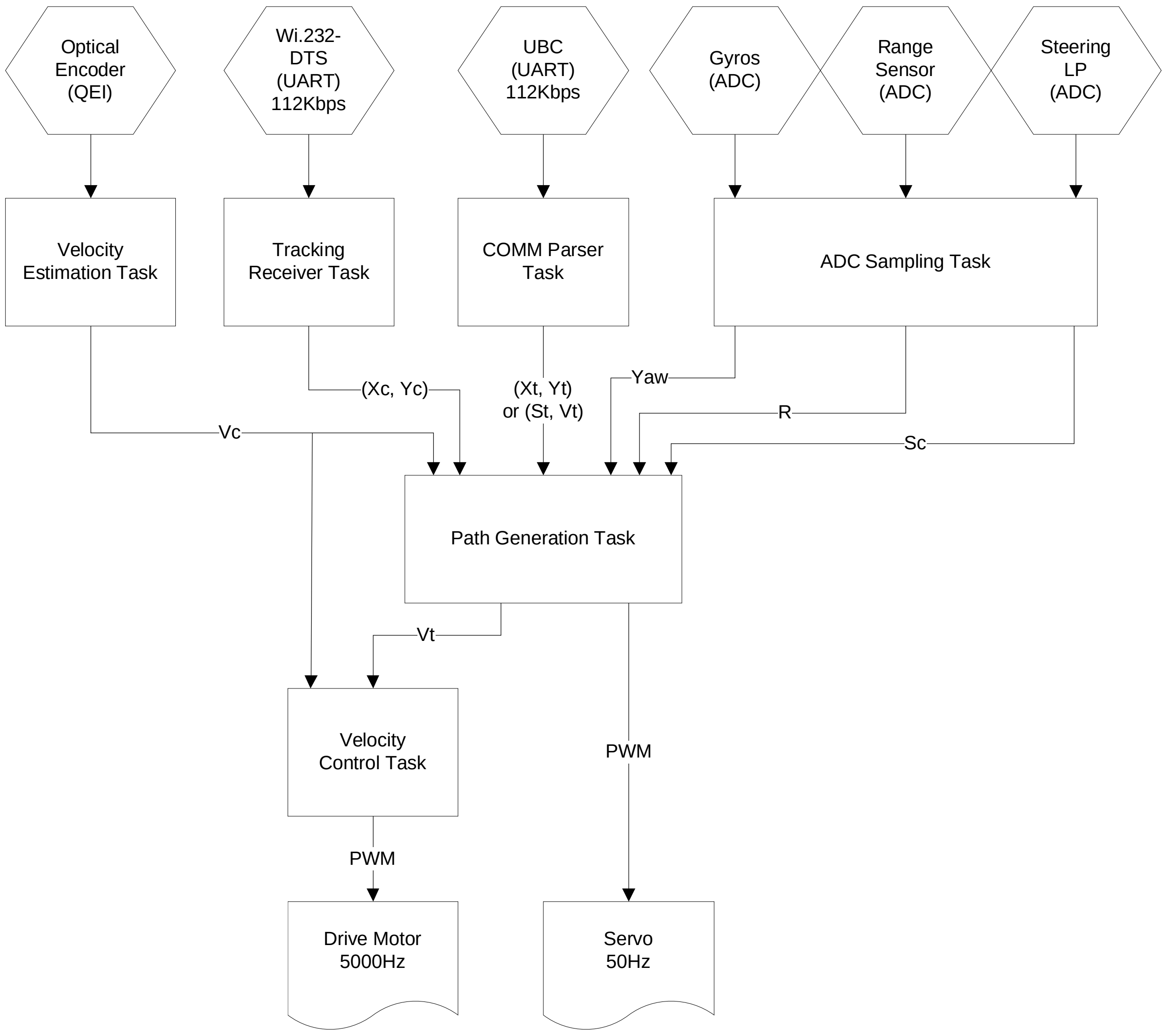


Figure 2-1: Task interdependence and signal flows related to LBC motion control

In **Figure 2-1**, a series of symbols are used to represent some inter-task signals. **Table 2-1** details what each of them represents.

Symbol	Description
Xt, Yt	X and Y coordinates of the target location, respectively
Xc, Yc	X and Y coordinates of the current location, respectively
St, Vt	Target steering angle and velocity, respectively
Sc, Vc	Current steering angle and velocity, respectively
Yaw	Yaw angle estimate of the vehicle
R	Range sensor reading, the distance to the nearest obstacle in front of the vehicle

Table 2-1: Inter-task signals related to motion control

For tasks related to outputting user data:

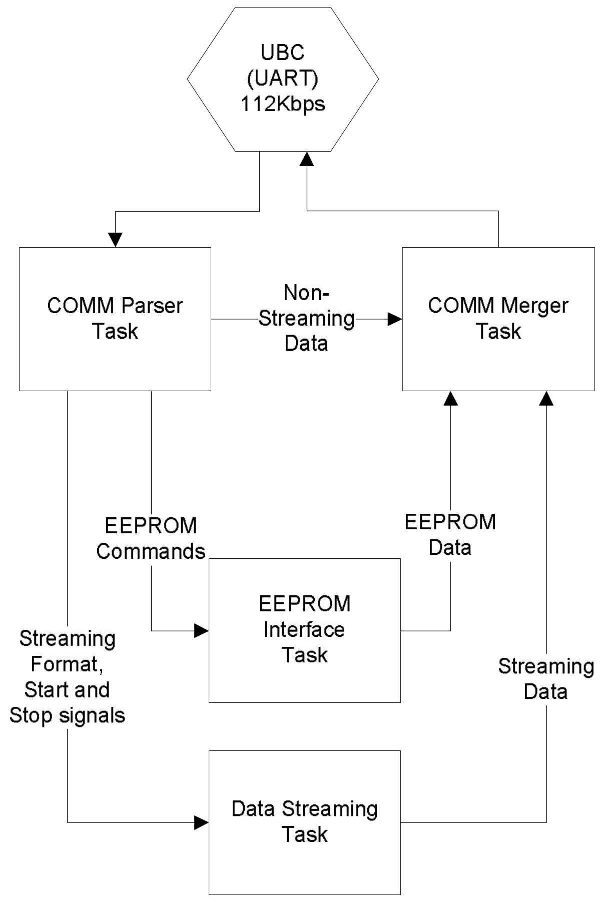


Figure 2-2: Tasks related to feeding data to the user

2.1 Task Descriptions

The frequency of a task can be determined by the frequency of the input and outputs of the task, plus how critical the task content is the motion and performance of the vehicle. This chapter details what each task is responsible for and how their respective frequencies and priorities are decided. Note that the maximum task frequency FreeRTOS allows is 1000Hz.

2.1.1 Velocity Control

This task controls the velocity of the vehicle by altering a PWM signal, which drives the drive motor via an H-bridge.

The task receives a target velocity from the Path Generation task and the current velocity from the Velocity Estimation task. It takes the difference between the two and use it to decide how much to accelerate or decelerate the motor by. A simple PID controller is used to achieve optimal control behavior.

To provide the PID controller fast enough update, we will run the task at the highest rate possible (1000Hz). As intermittent updates (happens when the task yield to other tasks due to low priority) will disrupt the control loop and cause instability, we will grant this task the highest priority by itself.

2.1.2 Velocity Estimation

This task estimates the current velocity of the vehicle based on changes in the optical encoder count. The resulted velocity estimates will be used by the Velocity Control task and the Path Generation task.

Since the Velocity Control task requires high update rate and priority, the Velocity Estimation task need to run at the same rate and given priority that's only below the VC task. Thus this task will run at 1000Hz and have the second highest priority.

2.1.3 Tracking Receiver

This task receives tracking information from the lower-board Wi-232 DTS module, which receives tracking information broadcasts from the overhead tracking system. This task will buffer tracking data bytes as soon as they come in from the UART and this task gets to run. Once a full frame of data is received (30Hz), it would decode the packet and extract relevant information. The decoded x and y coordinates would be used by the Path Generation task to determine motion.

The UART will run at 57600bps, while its FIFO is only 16 characters long. If the task doesn't retrieve data in the FIFO, the FIFO will be filled in approximately 1.4ms. To prevent data loss, we will run this task at 1000Hz. To insure against data loss, we also have to grant this task high enough priority over other long duty-cycle tasks to make sure this task gets to run every 1ms.

2.1.4 ADC Sampling

This task samples all six ADC channels plus the microcontroller chip temperature. Simple filtering (or other processing) may be applied to the sampled data to extract useful information for other tasks and the user.

There are six channels in the ALMC ADC. They are used to receive the following signals:

1. ADC0: Gyro Yaw
2. ADC1: Steering Linear Potentiometer
3. ADC2: Range Sensor
4. ADC3: 1.225V Reference
5. ADC4: Gyro Roll
6. ADC5: Gyro Pitch

Among the above channels, the three gyroscope channels plus the reference channel should be sampled at the same rate, while the other two channels can be sampled at different rates. There are two approaches to perform ADC sampling through periodical polling tasks: 1) Setup separate tasks for sampling independent channels (thus at different rates); or 2) Use a single task to sample all the channels at the highest rate necessary. Before we make a decision, let's first look at what is the sampling rate necessary for each (or each group of) channel.

2.1.5 Gyro Channels

The gyros outputs instantaneous angular velocities (in degree/second), which we integrate over time to obtain angular displacement (in degrees) from the initial orientation. For the integration, we adopt a piecewise-linear approximation for the velocity dynamic. Specifically, we assume that the angular velocity varies linearly over time within each sampling period. Thus, we can approximate the angular displacement over a sampling period as:

$$= - \cdot (\quad + \quad) \cdot \Delta$$

Where is the ith velocity sample, Δ is the sampling period, and is the angular displacement over the ith sampling interval (between the (i-1)th and ith velocity samples). Thus the current orientation can be calculated as:

$$= \quad + \sum$$

According to the manufacturer's datasheet, the gyroscope output signal is band-limited at 140 Hz. Thus by the Nyquist-Shannon Sampling Theorem, a sampling frequency higher than 280 Hz would be sufficient.

2.1.6 Other Channels

For Steering Linear Potentiometer reading, the sampling rate has to be as fast as the steering update rate, which is 50Hz.

For Range Sensor reading, the sampling rate can be application dependant. But if the range information is to be used for obstacle avoidance, the sampling rate should not be lower than the Path Generation update rate at 50Hz.

Thus to accommodate sampling requirements for the gyro outputs, we will use a single task to sample all the ADC channels (including microcontroller chip temperature) at 500Hz [Note: theoretically it should also be sufficient to sample the gyros at 300 or 400Hz. However due to constraints in FreeRTOS, it is far easier to run a task at 500Hz.

2.1.7 Path Generation

This task mainly performs higher level motion controls for the vehicle. It generates target velocities for the VC task, which drives the drive motor directly. It generates target steering angles as well. Since the target steering angles translate to PWM signals linearly, this task directly drives the servo.

There is an array of inputs that feeds this task's calculation of target velocities and steering angles. They include current location coordinates from the Tracking Receiver task, current velocity from the Velocity Estimation task, current steering angle, current yaw angle, and current range sensor output from the ADC Sampling task. This task also receives motion commands from the user via the COMM Parser task. There are two formats of commands, one in terms of target velocity and target steering and another in terms of x and y coordinates of a target location. When the command is received in the first format, the target velocity and steering angle can be directly used to drive lower level controls. When it's in the second format, this task is responsible for calculating a sequence of target velocities and steering angles that would lead the vehicle to the target location.

This task could run at the servo update rate (50Hz), the tracking update rate (30Hz), or somewhere in between. We've decided to use the servo update rate to minimize latency for command execution.

2.1.8 COMM Parser

This task is responsible for receiving messages from the UBC via a UART interface (a separate one than the tracking UART). Similar to the Tracking Receiver task, as soon as this task gets to run, it would buffer all the data that is either just received by the UART or already in the UART FIFO. Once an entire message is accumulated, it would decode it and pass along the information to its intended tasks.

The COMM Parser also executes all the commands that would incur little delay while leave long-delay tasks like EEPROM interfacing or data streaming to other tasks. This approach is adopted because it'd require significant redundant processing to have a separate task doing all the command handling, and it'd risk missing command data to let the COMM Parser handle all the commands.

Similar to the tracking UART, the COMM UART also has an 115200bps baud rate and a 16-character deep FIFO, thus the task needs to be run at 1000Hz to prevent data loss. As loss of command data from the UBC is not likely to jeopardize the stability of the motion control, we'd give this task a lower priority than tasks above.

2.1.9 COMM Merger

This task collects messages from other tasks (the COMM Parser task and the Data Streaming task) and sends them through the COMM UART in the order they're received. Since the Data Streaming task is controlled by the COMM Parser task, there will not be conflicts between the two tasks.

We'll run this task at the same rate as the COMM Parser task (1000Hz). Given the same priority as the COMM Parser task, this task will get to run following the COMM Parser task since we will create it right after creating the COMM Parser task.

2.1.10 EEPROM Interface

This task writes to and reads from the on-board EEPROM. It receives EEPROM commands from the COMM Parser task and performs the actual interfacing with the EEPROM. Since writing to or reading from the EEPROM are likely to impose some non-negligible latency, this would free the COMM Parser up to receive more commands from the UART.

The EEPROM has 64 16-bit wide registers. We divide it into two halves. The first 32 registers (Non-User Registers) will be reserved to store control parameter values, and the remaining 32 registers (User Registers) will be accessible to the users allowing them to store anything they wish to. We've assigned 21 of the Non-User Registers to specific control parameters (see **Table 1-3** for details).

This interface currently has the following four functionalities:

1) Commit the current values of the control parameters to their corresponding Non-User Registers to the EEPROM.
2) Restore the value of the control parameters from their corresponding Non-User Registers in the EEPROM.
3) Write a value to a User-Register in the EEPROM by an address.
4) Read the value of a User-Register in the EEPROM.

2.1.11 Data Streaming

This task packages data that are of interest to the user and sends the packages out to the user at a constant frequency. The packages will first be picked up by the COMM Merger task before it gets sent to the COMM UART. The start, stop, and format of streaming are set through the COMM Parser task.

We will run this task at the same rate as the tracking data (30Hz), which is probably the most important to the user in most applications.

2.1.12 Battery Level and Temperature Monitoring

This task is responsible for monitoring the battery voltage level and the microcontroller chip temperature. The battery level is compared to an internal reference voltage (4.54V). If the battery voltage falls below this level, a warning will be issued to signal the UBC (and hence the user) that the battery needs recharge. The temperature is already read by the ADC Sampling task at 100Hz, this task is only responsible for checking whether it has risen above 65°C. If that happens, a temperature warning will be issued to tip the user taking a break and letting the microcontroller cool off.

Since neither the battery level nor the temperature should change rapidly, we'll monitor them at a very low frequency (1Hz) and priority.

2.1.13 User LED Control

This task controls the on off pattern of the two user LEDs (one green and one red). Specifically it controls the frequencies and duty-cycles of the flashing according to a pair of system variables, which can be changed by the user via the WRIT_LED command.

To provide the widest frequency range possible for LED flashing, we will run this task at the highest rate (1000Hz) and the lowest priority.

2.1.14 User GPIO Control

This task controls the high-low pattern of the two user GPIO pins (GPIO0 and GPIO1). In fact, only outputs can be controlled at this point. Specifically it controls the frequencies and duty-cycles of the pin outputs according to a pair of system variables, which can be changed by the user via the WRIT_GPO command. We use the same mechanism here as in user LED control.

To provide the widest frequency range possible for the pin outputs, we will run this task at the highest rate (1000Hz) and the lowest priority.

2.1.15 User-Defined Task 0 and User Defined Task 1

These two tasks will be left blank for the user to program. The user would also need to recompile the entire LBC code and load it into the microcontroller.

There is a range of frequencies the user can select for these two tasks. The priorities will be the lowest among all the tasks; hence it is advisable that the user minimize processing cycles of the tasks.

2.2 Summery

Table 2-2 summarizes all the tasks that will be running concurrently in the LBC. Their priorities and frequencies are also listed. The tasks with a “Var.” frequency can run at user-selected rates (within a certain range).

Tasks	Priority	Freq (Hz)	Description
Velocity Control	ITP+6	1000	Controls the drive motor so that the vehicle run at a target velocity set by the Path Generation Task
Velocity Estimation	ITP+5	1000	Estimates current velocity based on encoder readings
Tracking Receiver	ITP+5	1000	Receives tracking data
ADC Sampling	ITP+4	500	Samples all the ADC channels and calculate the current steering angle, range, and yaw angle
Path Generation	ITP+3	50	Generates target velocity (or velocity profiles) for the VC task, generate and output PWM to the servo
COMM Parser (User Command Handler)	ITP+2	1000	Receives messages from the UBC and distribute them to the related tasks. It also handles many of the user commands directly.
COMM Merger	ITP+2	1000	Collects messages from various tasks and send them to the UBC in the order they’re received
EEPROM Interface	ITP+2	10	Write data to or read data from on-board EEPROM
Data Streaming	ITP+2	30	Streams user-requested data to the UBC via the COMM Merger task
Battery Level & Temperature Monitoring	ITP+1	1	Monitors the battery voltage level and the microcontroller chip temperature; Issue warnings if either goes across some preset safety levels.
User LED Control	ITP+1	1000	Controls the red and green LEDs based on user requests
User GPIO Control	ITP+1	1000	Controls two GPIO outputs based on user requests
User Defined Task0	ITP+1	Var.	Skeleton task, to be defined by the user
User Defined Task1	ITP+1	Var.	A second skeleton task, to be defined by the user

Table 2-2: List of all LBC tasks

We’ve also added to the frequency and priority information to **Figure 2-1** to provide a clearer hierarchy of tasks related to motion control:

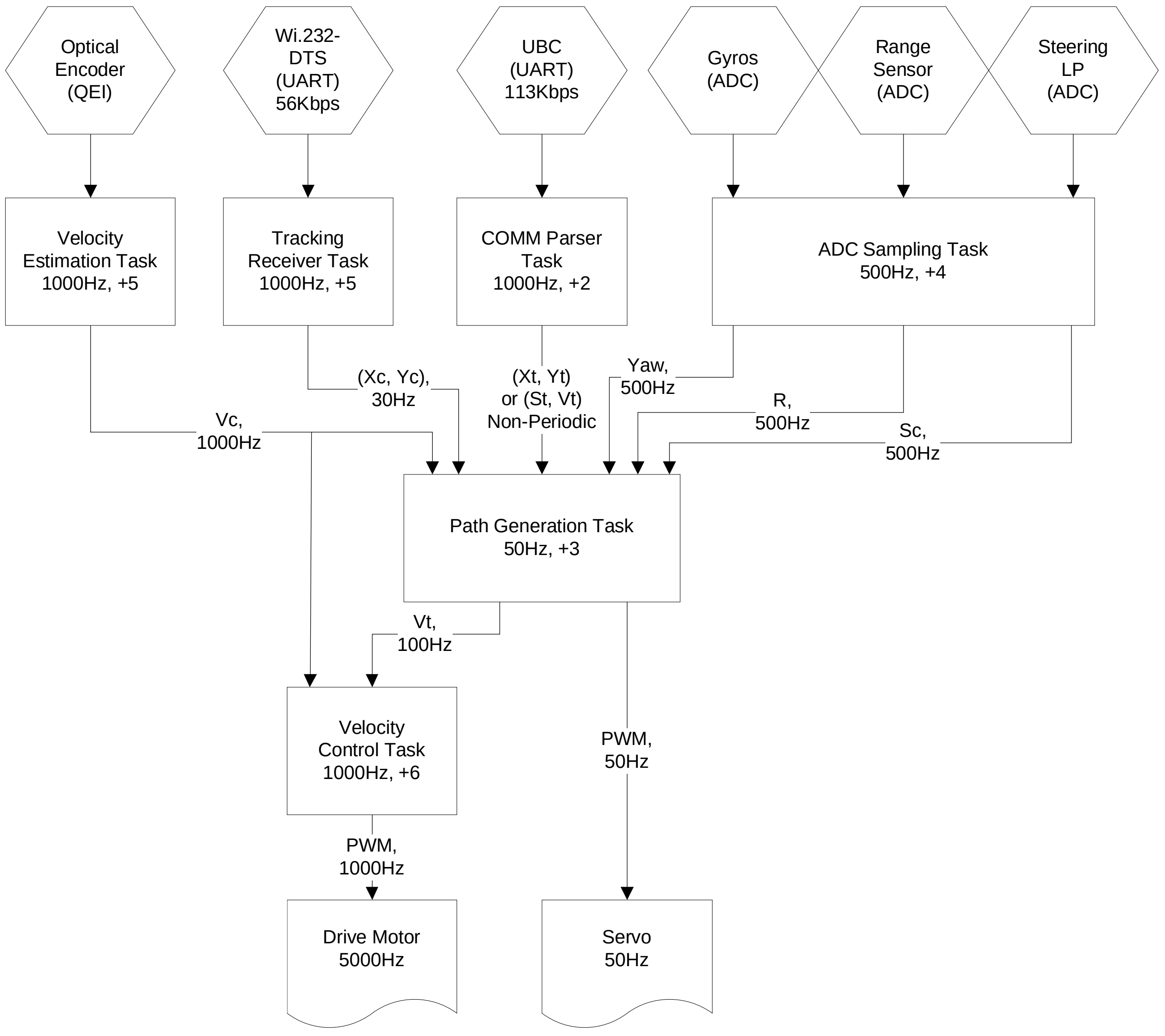


Figure 2-3: Motion-control related tasks plus their interdependence and signal flow

3. Communication Protocol

This chapter details the communication protocol between the Upper Board Controller (UBC, in the FPGA) and Lower Board Controller (LBC, in the microcontroller). The UBC interfaces with users and runs application-related operations, where the LBC has access to onboard sensors and controls the vehicle motions. Thus the main purpose of the communication is to deliver commands to the LBC and data plus status to the UBC.

3.1 Hardware Setup

The UBC communicates with the LBC through a UART interface with the following settings:

Baud Rate:	115200
Data Bits:	8
Stop Bits:	1
Parity Bit:	None
Flow Control:	None

3.2 Message Format

Communication between the LBC and the UBC are achieved through exchanging messages. All messages will assume the following format:

<Header><Data><Checksum><Terminator>

1 byte N bytes 1 byte 1 byte

Where the data field can vary in length (N >= 0) depending on the message type. We'll explain each individual field in the following section.

3.2.1 Header

The message header is a one-byte word whose most significant bit is "1" with the exception of 0xFF (reserved for the Terminator). Thus its value falls in the range of [0x80, 0xFE]. These values will be unique to the rest of the message. In addition to denoting the beginning of a message, it also indicates the message type thereof its purpose. We'll detail the mapping between header values and message types in the next chapter.

3.2.2 Data

Certain types of messages will encapsulate a sequence of data. Length of the whole data field, plus order and width of individual data are fixed for a specific message type, with the notable exception of streaming messages. For streaming messages, data formats are transmitted before streaming begins. For other messages, there will be a specific number of data bytes trailing the header, where each byte assumes the range of [0x00, 0x7F].

Since each message data byte can only be 7 bits long while widths of original data types vary, most of which are multiples of 8.To minimize bandwidth usage, we opt for the following data packing scheme:

1. Form a bit stream by packing the original data one after another (most significant bit first).
2. Divide the bit stream into 7-bit long words while maintaining the order of the bits. If the last word is shorter than 7 bits, prefix with zeros to make it 7 bits.
3. Prefix all the words with a zero to make them 8-bit words. Thus they will all assume the range of [0x00, 0x7F].

3.2.3 Checksum

An 8-bit Checksum will be generated for all messages by **XOR**ing all bytes from header to the last data byte, respectively. Then it will be **AND**ed with 0x7F, thus it assumes the range of [0x00, 0x7F]. Such Checksum can only be used to detect odd number of error occurrences.

For messages with no data, the Checksum will simply be the result of **AND**ing the header with 0x7F.

Since the first bit of the Checksum is always 0's, the Checksum can't be used detect any errors happening to the first bits. Such errors will likely result in unrecognizable messages and hence re-transmissions of the same messages.

3.2.4 Terminator

A byte with the value 0xFF will indicate the end of the message. This value is unique to the rest of the message.

Here is a table summarizing the lengths and value ranges of each message field:

Field	Length (bytes)	Value/Range	Description
Header	1	0x80 – 0xFE	start/type of message
Data	N (>= 0)	0x00 – 0x7F	data
Checksum	1	0x00 – 0x7F	checksum of previous bytes
Terminator	1	0xFF	end of message

Table 3-1: Message Structure

3.3 Byte Format

To ensure unambiguous encoding and decoding of individual message bytes, we design the following scheme to differentiate data/Checksums, the terminator, and various types of headers:

1) 0xFF is dedicated to be the terminator of a message.
2) With the exception of 0xFF, a byte with a '1' MSb (bit 7) is a header byte. A byte with a '0' MSb is either a data byte or a Checksum byte.
3) Among headers, bytes with their bit 4-2 being '111' are reserved to be general error headers. Messages with such headers should not generate replies from the receiver.
4) With the exceptions in 3), header bytes with '1' as bit 6 are for command messages and their corresponding positive acknowledgements, while the same headers that differ only on bit 6 are their corresponding error messages.
5) Bit 5 differentiates control messages from system messages.

Type	Bit Patterns			Byte Values	Header Symbols	Data Len.	Description
	Bit 7	Bit 6-5	Bit 4-0				
Data, Checksum	0	xx	xxxxx	0x00 – 0x7F	N/A	N/A	Data or Checksum bytes
Control Error Header	1	00	xxxxx	0x80 – 0x9B	ERR_CTRL	0	Specific error messages for the corresponding control messages (with the same Bit 4-0)
			111xx	0x9C – 0x9F	RESERVED		
System Error Header		01	xxxxx	0xA0 – 0xBB	ERR_SYS	0	Specific error messages for the corresponding system messages (with the same Bit 4-0)
			11100	0xBC	UNKCMD	0	Unknown command
			11101	0xBD	CSER	0	Checksum error
			1111x	0xBE – 0xBF	RESERVED		
Control Command Header		10	00000	0xC0	STOP	0, 0	Stop immediately
			00001	0xC1	MOVE_XY	28, 28	Move to coordinate (X, Y)
			00010	0xC2	MOVE_SV	28, 28	Steer to angle S, maintain velocity at V
			00xxx	0xC3 – 0xC7	NOT IN USE		
			01000	0xC8	CALB_ST	0, 28	Calibrate steering
			01001	0xC9	CALB_DM	0, 0	Calibrate drive motor
			01010	0xCA	CALB_GS	0, 28	Calibrate gyroscopes
			01xxx	0xCB – 0xCF	NOT IN USE		
			10000	0xD0	WRIT_EP	21, 21	Write to a EEPROM User register

			10001	0xD1	READ_EP	7, 21	Read from a EEPROM User register
			10010	0xD2	WRIT_LED	14, 14	Write to green and red LEDs
			10011	0xD3	WRIT_GPO	14, 14	Write to GPIO0 & GPIO1
			10xxx	0xD4 – 0xDB	NOT IN USE		
			111xx	0xDC – 0xDF	RESERVED		
System Command Header		11	00000	0xE0	INIT	0, 0	Initialization
			00001	0xE1	CMEP	0, 0	Commit system parameters to the EEPROM
			00010	0xE2	RSEP	0, 0	Restore system parameters from the EEPROM
			00011	0xE3	DRDS	0, 0	Disables both the servo and the drive motor
			00100	0xE4	DREN	0, 0	Enables both the servo and the drive motor
			00101	0xE5	STDS	0, 0	Disables data streaming
			00110	0xE6	STEN	0, 0	Enables data streaming
			00111	0xE7	NOT IN USE		
			01000	0xE8	READ_ID	0, 7	Read vehicle ID
			01001	0xE9	READ_XY	0, 28	Read coordinate (X, Y)
			01010	0xEA	READ_SV	0, 28	Read steering angle and velocity
			01011	0xEB	READ_AY	0, 14	Read adjusted yaw
			01100	0xEC	READ_PRY	0, 42	Read raw pitch, roll and yaw
			01101	0xED	READ_BT	0, 2	Read battery level and chip temperature
			01110	0xEE	READ_RS	0, 14	Read ranger sensor reading
			01111	0xEF	READ_HD	0,14	Read heading (provided by the tracking)
			10000	0xF0	READ_SF	0, 28	Read current streaming format
			10001	0xF1	UPD_SF	28, 28	Update streaming format
			10010	0xF2	READ_PAR	7, Var.	Read parameter
			10011	0xF3	UPD_PAR	Var., Var.	Update parameter
			10xxx	0xF4 – 0xFB	NOT IN USE		
			11100	0xFC	STREAM	Var.,	Streaming data from LBC

						N/A	to UBC
			111xx	0xFD – 0xFE	RESERVED		
Terminator			11111	0xFF	TERM	N/A	Terminator

Table 3-2: Byte Formats

A few notable things about Table 2:

1. Data and Checksum bytes are not distinguishable on their values, instead on their relative distances from the terminator. Namely, the immediate two bytes ahead of the terminator will be the Checksums. Other bytes in the range of [0x00, 0x7F] will be data.
2. We are assuming big-endian bit ordering, where bit 7 is the most significant bit and bit 0 is the least.
3. For control and system command headers, their corresponding positive acknowledgements will use the exact same headers.
4. The numbers in the “Data Len.” column indicate the total lengths of message data fields. These numbers are measured in bits. They are the sums of individual data widths. Note that these are just raw data lengths, while actual data lengths are longer since data bytes only occupy the lower 7 bits and the length may not be evenly divided by 7. When there are two numbers, the first number indicates the data length of the command message while the second one indicates the data length of the positive acknowledgement.
5. For each control and system command message header (with the exception of STREAM), there is a corresponding error message header for errors tied to the command. Each pair of such headers will share every bit except for the second most significant bit (Bit 6). We will detail each type of message in the next chapter.
6. The data fields of error messages are always empty.

3.4 Message Type

This chapter provides details about individual message types, differentiated by their headers. It will describe what each type of message does, the meaning and width of the data attached, plus the expected reply in cases of success and errors.

The widths of individual data fields in this chapter are quantified in bits. They refer to the widths of the original data, not the width they actually occupy in the messages. However, we'll commonly see widths of 1-bit, 7-bit, 14-bit, and larger multiples of 7 bits. This is because each byte in the data field can only convey up to 7 bits of information (see the last chapter).

As described in the last chapter, for each system or control message header (with the exception of STREAM), there exist a counterpart system or control error message header. We'll describe the error messages as replies (in case of errors) to their corresponding original messages. Thus their descriptions can be found in the section of their counter-messages.

3.4.1 Control Command Messages and Control Error Messages

Control command messages and their corresponding positive acknowledgements share the same headers, which are in the range of [0xC0, 0xDB]. The corresponding error messages are led by headers in the range of [0x80, 0x9B].

STOP

This message commands the LBC to stop moving as soon as it can. The byte value for the header is 0xC0 (1100 0000b). There is no data to be sent in this message, thus the message will look like:

```
0xC0, 0x40, 0xFF
```

Currently, there is no condition where the STOP command is not allowed to being execute, thus we will always reply with the same message acknowledging receipt of the command.

If there is a condition where STOP command is not allowed, the LBC will reply with an error message headed by 0x80 (1000 0000b). The error message for STOP error will look like:

```
0x80, 0x00, 0xFF
```

MOVE_XY

This message commands the LBC to move to position with coordinates (X, Y). The LBC checks the validity of the received coordinates and feeds them (if valid) to its motion-control task, which will be executing the movement. The x and y coordinates should fall in the range of [0, 640] and [0, 960], respectively, otherwise they will be rejected. The byte value for the header is 0xC1 (1100 0001b). The X and Y coordinates (14 bits each) will be included in the message, which will look like:

```
0xC1, <14-bit X>, <14-bit Y>, <Checksum>, 0xFF
```

If the coordinates are valid and successfully sent to the motion-control task, the LBC will reply with the exact same message. This just informs the UBC that the command is being executed. The LBC will not issue further messages regarding the development of the motion (as follow-ups to this command message). However the UBC can monitor the motion status by sending READ_XY commands.

If the coordinates are invalid, the LBC will reply with an error message headed by 0x81 (1000 0001b). The error message for MOVE_XY error will look like:

0x81, 0x01, 0xFF

MOVE_SV

This message commands the LBC to turn to a specific steering angle and maintain a specific velocity. The LBC first checks the validity of the target angle and velocity. If valid, it feeds them to its motion-control task, which will be executing the movement. Currently, the LBC command parser is not checking the validity of the steering angle and velocity provided by the user. The byte value for the header is 0xC2 (1100 0010b). The target steering angle and velocity (14 bits each) will be included in this message, which will look like:

0xC2, <14-bit Angle>, <14-bit Velocity>, <Checksum>, 0xFF

If the target angle and velocity are valid and successfully sent to the motion-control task, the LBC will reply with the exact same message. This just informs the UBC that the command is being executed. The LBC will not issue further messages regarding the development of the motion (as follow-ups to this command message). However the UBC can monitor the motion status by sending READ_SV commands.

If the angle and velocity are invalid, the LBC will reply with an error message headed by 0x82 (1000 0010b). The error message for MOVE_SV error will look like:

0x82, 0x02, 0xFF

CALB_ST

This message commands the LBC to perform steering calibration. The byte value for the header is 0xC8 (1100 1000b). There is no data to be sent in this message, thus the message will look like:

0xC8, 0x48, 0xFF

(TO BE REVISED) If the steering is successfully calibrated, the LBC will reply with a message led by the same header plus the 14-bit steering angle offset and the 14-bit steering gain. The reply message will look like:

0xC8, <14-bit Offset>, <14-bit Gain>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0x88 (1000 1000b). The error message for CALB_ST error will look like:

0x88, 0x08, 0xFF

CALB_DM

This message commands the LBC to perform drive motor calibration. The byte value for the header is 0xC9 (1100 1001b). There is no data to be sent in this message, thus the message will look like:

0xC9, 0x49, 0xFF

(TO BE REVISED) If the drive motor is successfully calibrated, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0x89 (1000 1001b). The error message for CALB_DM error will look like:

0x89, 0x09, 0xFF

CALB_GS

This message commands the LBC to perform gyroscope calibration. The byte value for the header is 0xCA (1100 1010b). There is no data to be sent in this message, thus the message will look like:

0xCA, 0x4A, 0xFF

(TO BE REVISED) If the steering is successfully calibrated, the LBC will reply with a message led by the same header plus three 14-bit gyroscope offsets. The reply message will look like:

0xCA, <14-bit Pitch Offset>, <14-bit Roll Offset>, <14-bit Yaw Offset>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0x8A (1000 1010b). The error message for CALB_GS error will look like:

0x8A, 0x0A, 0xFF

WRIT_EP

This message commands the LBC to write (or overwrite) the value of a 16-bit user register in the EEPROM. The byte value for the header is 0xD0 (1101 0000b). The 7-bit address of the register and the 16-bit new value will be included in this message, which will look like:

0xD0, <7-bit Address>, <16-bit Data>, <Checksum>, 0xFF

If the register is successfully written, the LBC will reply with the exact same message. To verify the success of the writing, try read the register value using the READ_EP command.

Otherwise, the LBC will reply with an error message headed by 0x90 (1001 0000b). The error message for WRIT_EP error will look like:

0x90, 0x10, 0xFF

READ_EP

This message commands the LBC to read and report back the value of a 16-bit user register in the EEPROM. The byte value for the header is 0xD1 (1101 0001b). The 7-bit register address will be included in this message, which will look like:

0xD1, <7-bit Address>, <Checksum>, 0xFF

The LBC will first reply with the exact same message acknowledging the receipt of the command.

Once the register is successfully read, the LBC will send a second message led by the same header plus the 7-bit address and the 16-bit data. The reply message will look like:

0xD1, <7-bit Address>, <16-bit Data>, <Checksum>, 0xFF

If the reading failed, the LBC will reply with an error message headed by 0x91 (1001 0001b). The error message for READ_EP error will look like:

0x91, 0x11, 0xFF

WRIT_LED

This message commands the LBC to control the two user LEDs (one green and one red). The byte value for the header is 0xD2 (1101 0010b). Two 7-bit data will be included to indicate duty cycles or the LEDs. How duty cycles determine LED behaviors is defined in **Table 3-5**. Thus the message will look like:

0xD2, <7-bit Green LED Duty Cycle>, <7-bit Red LED Duty Cycle>, <Checksum>, 0xFF

If the LEDs are successfully switched, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0x92 (1001 0010b). The error message for WRIT_LED error will look like:

0x92, 0x12, 0xFF

WRIT_GPO

This message commands the LBC to control the two user GPIO pins (we actually only support General Purpose Outputs at this point). The byte value for the header is 0xD3 (1101 0011b). Two 7-bit data will be included to indicate duty cycles of the GPIO pins. How duty cycles determine GPIO pin behavior is defined in **Table 3-6**. Thus the message will look like:

0xD3, <7-bit GPIO0 Duty Cycle>, <7-bit GPIO1 Duty Cycle>, <Checksum>, 0xFF

If the GPIO pin duty cycles are successfully set, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0x923 (1001 0011b). The error message for WRIT_GPO error will look like:

0x93, 0x13, 0xFF

3.4.2 System Command Messages and System Error Messages

System command messages and their corresponding positive acknowledgements share the same headers, which are in the range of [0xE0, 0xFB]. The corresponding error messages are led by headers in the range of [0xA0, 0xBB].

INIT

This message commands the lower board controller (LBC) to initialize (or reset) the motion-control and sensing system. The byte value for the header is 0xE0 (1110 0000b). There is no data to be sent in this message, thus the message will always look like:

0xE0, 0x60, 0xFF

Once the initialization is successfully completed, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA0 (1010 0000b). The error message for INIT error will look like:

0xA0, 0x20, 0xFF

CMEP

This message commands the LBC to commit all the current system parameter values to their corresponding EEPROM registers. The byte value for the header is 0xE1 (1110 0001b). There is no data to be sent in this message, thus the message will look like:

0xE1, 0x61, 0xFF

Once committing to EEPROM is successfully completed, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA1 (1010 0001b). The error message for CMEP error will look like:

    0xA1, 0x21, 0xFF

RSEP

This message commands the LBC to restore all the current system parameter values from their corresponding EEPROM registers. The byte value for the header is 0xE2 (1110 0010b). There is no data to be sent in this message, thus the message will look like:

    0xE2, 0x62, 0xFF

Once restoration from EEPROM is successfully completed, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA2 (1010 0010b). The error message for RSEP error will look like:

    0xA2, 0x22, 0xFF

DRDS

This message commands the LBC to disable both the drive motor and the servo. The byte value for the header is 0xE3 (1110 0011b). There is no data to be sent in this message, thus the message will look like:

    0xE3, 0x63, 0xFF

If the drive motor and the servo are successfully disabled, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA3 (1010 0011b). The error message for DRDS error will look like:

    0xA3, 0x23, 0xFF

DREN

This message commands the LBC to enable both the drive motor and the servo. The byte value for the header is 0xE4 (1110 0100b). There is no data to be sent in this message, thus the message will look like:

    0xE4, 0x64, 0xFF

If the drive motor and the servo are successfully enabled, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA4 (1010 0100b). The error message for DREN error will look like:

0xA4, 0x24, 0xFF

STDS

This message commands the LBC to disable or stop data streaming. The byte value for the header is 0xE5 (1110 0101b). There is no data to be sent in this message, thus the message will look like:

0xE5, 0x65, 0xFF

If data streaming is successfully disabled, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA5 (1010 0101b). The error message for STDS error will look like:

0xA5, 0x25, 0xFF

STEN

This message commands the LBC to enable or start data streaming. The byte value for the header is 0xE6 (1110 0110b). There is no data to be sent in this message, thus the message will look like:

0xE6, 0x66, 0xFF

If data streaming is successfully enabled, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xA6 (1010 0110b). The error message for STEN error will look like:

0xA6, 0x26, 0xFF

READ_ID

This message commands the LBC to read and report back the ID number of the vehicle. The byte value for the header is 0xE8 (1110 1000b). There is no data to be sent in this message, thus the message will look like:

0xE8, 0x68, 0xFF

If the vehicle ID is successfully read, the LBC will reply with a message led by the same header plus 7 bits for the ID number. The reply message will look like:

0xE8, <7-bit ID>, < Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xA8 (1010 1000b). The error message for READ_ID error will look like:

0xA8, 0x28, 0xFF

READ_XY

This message commands the LBC to read and report back the current (X, Y) coordinates of the vehicle, which assume the ranges of [0, 640] and [0, 960], respectively. The byte value for the header is 0xE9 (1110 1001b). There is no data to be sent in this message, thus the message will look like:

0xE9, 0x69, 0xFF

If the (X, Y) coordinates are successfully read, the LBC will reply with a message led by the same header plus 28 bits for the coordinates (14 bits each). The reply message will look like:

0xE9, <14-bit X>, <14-bit Y>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xA9 (1010 1001b). The error message for READ_XY error will look like:

0xA9, 0x29, 0xFF

READ_SV

This message commands the LBC to read and report back the current steering angle and velocity of the vehicle. The byte value for the header is 0xEA (1110 1010b). There is no data to be sent in this message, thus the message will look like:

0xEA, 0x6A, 0xFF

If the steering angle and velocity are successfully read, the LBC will reply with a message led by the same header plus 28 bits for the data (14 bits each). The reply message will look like:

0xEA, <14-bit Angle>, <14-bit Velocity>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAA (1010 1010b). The error message for READ_SV error will look like:

0xAA, 0x2A, 0xFF

READ_AY

This message commands the LBC to read and report back the adjusted yaw of the vehicle, which is defined TBFI. The byte value for the header is 0xEB (1110 1011b). There is no data to be sent in this message, thus the message will look like:

0xEB, 0x6B, 0xFF

If the adjusted yaw is successfully read, the LBC will reply with a message led by the same header plus 14 bits of data. The reply message will look like:

0xEB, <14-bit Adj. Yaw>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAB (1010 1101b). The error message for READ_AY error will look like:

0xAB, 0x2B, 0xFF

READ_PRY

This message commands the LBC to read and report back the current raw pitch, roll, and yaw values of the vehicle, which are defined TBFI. The raw pitch value refers to the ADC reading of the pitch gyro minus the ADC reading of the reference voltage. Same thing goes for the raw roll and the raw yaw values. The byte value for the header is 0xEC (1110 1100b). There is no data to be sent in this message, thus the message will look like:

0xEC, 0x6C, 0xFF

If the raw pitch, roll, and yaw values are successfully read, the LBC will reply with a message led by the same header plus 42 bits of data (14 bits each). The reply message will look like:

0xEC, <14-bit Pitch>, <14-bit Roll>, <14-bit Yaw>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAC (1010 1100b). The error message for READ_PRY error will look like:

0xAC, 0x2C, 0xFF

READ_BT

This message commands the LBC to read and report back a binary battery level indicator and a binary temperature indicator. The battery level indicator shows whether the battery voltage has fallen below a certain level (4.5V), whereas the temperature indicator informs whether the microcontroller temperature has risen above 65°C. If either of these indicators is on, the LBC can become unstable. The byte value for the header is 0xED (1110 1101b). There is no data to be sent in this message, thus the message will look like:

0xED, 0x6D, 0xFF

If the two indicators are successfully read, the LBC will reply with a message led by the same header plus 2 bits of data (1 bit each). The reply message will look like:

0xED, <1-bit Battery Level>, <1-bit Temp>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAD (1010 1101b). The error message for READ_BT error will look like:

0xAD, 0x2D, 0xFF

READ_RS

This message commands the LBC to read and report back the ADC reading of the range sensor, which are defined TBFI. The byte value for the header is 0xEE (1110 1110b). There is no data to be sent in this message, thus the message will look like:

0xEE, 0x6E, 0xFF

If the range is successfully read, the LBC will reply with a message led by the same header plus 14 bits of data. The reply message will look like:

0xEE, <14-bit Range>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAE (1010 1110b). The error message for READ_RS error will look like:

0xAE, 0x2E, 0xFF

READ_HD

This message commands the LBC to read and report back the heading estimate provide by the overhead tracking, which assume the unit of milli-radian and fall in the range of [0, 6283]. The byte value for the header is 0xEF (1110 1111b). There is no data to be sent in this message, thus the message will look like:

0xEF, 0x6F, 0xFF

If the heading is successfully read, the LBC will reply with a message led by the same header plus 14 bits of data. The reply message will look like:

0xEF, <14-bit Heading>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xAF (1010 1111b). The error message for READ_HD error will look like:

0xAF, 0x2F, 0xFF

READ_SF

This message commands the LBC to read and report back the current data stream format. The data stream format is a variable that indicates which system variable will be included in the data stream and how they will be formatted. This variable can only be changed by UPD_SF commands from the user. The byte value for the header is 0xF0 (1111 0000b). There is no data to be sent in this message, thus the message will look like:

0xF0, 0x70, 0xFF

If the stream format is successfully read, the LBC will reply with a message led by the same header plus the stream format, which will be a 28-bit number. The reply message will look like:

0xF0, <28-bit Format>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xB0 (1011 0000b). The error message for READ_SF error will look like:

0xB0, 0x30, 0xFF

UPD_SF

This message commands the LBC to update the data stream format. The data stream format is a variable that indicates which system variable will be included in the data stream and how they will be formatted. This variable can only be changed by UPD_SF commands from the user. The byte value for the header is 0xF1 (1111 0001b). There is no data to be sent in this message, thus the message will look like:

0xF1, <28-bit Format>, <Checksum>, 0xFF

If the format doesn't have any of the NOT-IN-USE (see **Table 3-4**) bits high, then it is a valid format and the update should be successful. If the stream format is successfully updated, LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xB1 (1011 0001b). The error message for UPD_SF error will look like:

0xB1, 0x30, 0xFF

READ_PAR

This message commands the LBC to read and report back the value of a control parameter. See (to be filled in) for a list of control parameters and what each parameter controls. The byte value for the header is 0xF2 (1111 0010b). A 7-bit number indicating which parameter to read will be attached in this message. Only one parameter can be read each time. Thus the message will look like:

0xF2, <7-bit Param. Num.>, <Checksum>, 0xFF

If the control parameter is successfully read, the LBC will reply with a message led by the same header, the same 7-bit parameter number, plus the value of the requested parameter, the length of which may vary. The reply message will look like:

0xF2, <7-bit Param. Num.>, <Variable-Length Data>, <Checksum>, 0xFF

Otherwise, the LBC will reply with an error message headed by 0xB2 (1011 0010b). The error message for READ_PAR error will look like:

0xB2, 0x32, 0xFF

UPD_PAR

This message commands the LBC to update the value of a control parameter. See (to be filled in) for a list of control parameters, the range of each parameter, and what each parameter controls. The byte value for the header is 0xF3 (1111 0011b). A 7-bit number indicating which parameter to update plus the new parameter value will be attached in this message. Only one parameter can be updated each time. Thus the message will look like:

0xF3, <7-bit Param. Num.>, <Variable-Length Data>, <Checksum>, 0xFF

If the control parameter is successfully updated, the LBC will reply with the exact same message.

Otherwise, the LBC will reply with an error message headed by 0xB3 (1011 0011b). The error message for UPD_PAR error will look like:

0xB3, 0x33, 0xFF

STREAM

Data streaming from the LBC to the UBC consists of periodical transmission of packets of data. This is the data streaming message which contains system variables the user is interested in. The streaming frame rate will be set at 30Hz, the same as the tracking update rate. The start and stop of the periodical transmissions can be triggered by the STEN and STDS commands, respectively. The byte value for the header is 0xFC (1111 1100b). The data length varies depending on how the user set it up before the streaming starts, thus the message will look like:

0xFC, <Variable-Length Data>, <Checksum>, 0xFF

There is no reply to this message. If the UBC finds errors (including general errors) or simply wants to stop the streaming, it can send a STDS command to the LBC.

3.4.3 General Error Messages

Messages with headers in the range of [0x9C, 0x9F], [0xBC, 0xBF], [0xDC, 0xDF], and [0xFC, 0xFE] are general error messages. These messages denote errors that are not tied to specific command types. The sender of such messages expects no reply from the receiver.

UNKCMD

This message informs the UBC that the LBC cannot understand the last message it received from the UBC. The byte value for the header is 0xBC (1011 1100b). The header and data of the erroneous message will be included. Thus the UNKCMD message will look like:

0xBC, <Variable Length Data>, <Checksum>, 0xFF

There is no reply to the UNKCMD message.

CSER

This message informs the UBC that a Checksum error has occurred to the last message received by the LBC. The byte value for the header is 0xBD (1011 1101b). There is no data to be sent in the CSER message, which will look like:

0xBD, 0x3D, 0xFF

There is no reply to the CSER message.

3.5 Control Parameters and System Variables

In this chapter, we will detail how the READ_PARAM and UPD_PARAM commands address the control parameters and how the READ_SF and UPD_SF command address all the system variables.

3.5.1 Control Parameters

The READ_PARAM and UPD_PARAM commands both contain a 7-bit number indicating which control parameter the user wish to read or update. **Table 3-3** shows how this parameter number is used.

Parameter Number	Parameter Name	Signness	Width (bits)	Note
1	Motor Encoder Ratio	U	14	
2	Motor Gear Ratio	U	14	
3	Motor PGain	U	14	
4	Motor IGain	U	14	
5	Motor DGain	U	14	
6	Motor Dead Band	U	14	
7	Servo Offset	S	14	
8	Servo Gain	S	14	
9	Servo PGain	U	14	
10	Servo IGain	U	14	
11	Servo DGain	U	14	
12	Servo Dead Band	U	14	

Table 3-3: Control Parameter Numbering

3.5.2 System Variables and the Streaming Format

To read or update the streaming format, the user needs to send a READ_SF or UPD_SF command containing the new 28-bit format. The LBC will check the new format against a validity mask (0x002060E0) to make sure it's valid. The validity mask will mask out bits in the format that should never be set (as fields that's indicated as "NOT IN USE" in the table below).

Variable number N is associated with Bit N in the streaming format variable. In the 28-bit streaming format variable, Bit 0 is the least significant bit (lsb), whereas Bit 27 is the msb.

In the data stream, variable number 0 should comes first (right after the streaming header), whereas number 27 should come last (right before the Checksum bytes).

Variable Number	Variable Name	Width (bits)	Note
0	ID	14	
1	X and Y coordinate	28	14 bits each, X comes first.
2	Steering Angle and Velocity	28	14 bits each, Steering Angle comes first.
3	Adjusted Yaw	14	
4	heading (Orientation)	14	Heading estimate from tracking system
5-7	NOT IN USE		
8	ADC Steering LP	14	
9	ADC Range Sensor Reading	14	
10	ADC Gyro reference Voltage	14	
11	ADC Raw Gyro Pitch and Roll	28	14 bits each, Pitch comes first.
12	ADC Raw Gyro Yaw	14	
13-14	NOT IN USE		
15	Chip Temperature plus Battery Level and Temperature Warnings	14	Upper 7 bit will be the chip temperature lsb will be temperature warning (1: higher than threshold, 0: lower than threshold) 2nd lsb will be battery level warning (1: lower than threshold, 0: higher than threshold) The middle 5 bits should be zeros.
16	Target X and Y coordinate	28	14 bits each, X comes first.
17	Target Steering Angle and Velocity	28	14 bits each, Steering comes first.
18	Target Yaw (Target Orientation)	14	
19	Motor Control Loop Proportional and Differential Errors	28	14 bits each, Proportional Error comes first.
20	Motor Control Loop Integral Error	28	
21	NOT IN USE		
22	User Variable 1	14	To stream user variable 1, the user needs to first point user variable pointer 1 to the

			variable he/she would like to stream, which can be done in the use defined tasks. Only the lower 14 bits will be streamed.
23	User Variable 2	14	Ditto
24	User Variable 3	14	Ditto
25	User Variable 4	14	Ditto
26	User Variable 5	14	Ditto
27	User Variable 6	14	Ditto

Table 3-4: System Variable Numbering and Streaming Format Definition

3.5.3 LED Control

The upper 3 bits (Bit4-6) will be used to distinguish different modes and frequencies.

Bit Pattern (Bit6-Bit4)	Mode	Frequency (Hz)
000	BLINK	1
001	BLINK	2
010	BLINK	5
011	BLINK	10
100	BLINK	20
101	BLINK	30
110	DIM	50
111	DIM	100

Table 3-5: LED Control Variable Coding Scheme

The lower 4bits (Bit0-3) will be used to indicate the number of duty cycles. Out of the possible 16 values, only the lower 11 will be used. For instance, the lower 4 bits being 5 would mean 50% duty cycles, being 10 would be constant high. The value of 11 to 15 will have the same effect as 0, indicating constant low.

3.5.4 GPIO Control

Similar to the scheme used in LED control, the upper 3 bits (Bit4-6) will be used to distinguish different frequencies.

Bit Pattern (Bit6-Bit4)	Frequency (Hz)
000	1
001	2
010	5
011	10
100	20
101	30
110	50
111	100

Table 3-6: GPIO Control Variable Coding Scheme

The lower 4bits (Bit0-3) will be used to indicate the number of duty cycles. Out of the possible 16 values, only the lower 11 will be used. For instance, the lower 4 bits being 5 would mean 50% duty cycles, being 10 would be constant high. The value of 11 to 15 will have the same effect as 0, indicating constant low.

4. An Upper Board Controller Demo Program

We wrote a demo program for the upper board controller, which is capable of basic control and configuration of the lower board. The intention for the demo program is to demonstrate communication with the LBC through a set of commands. Hopefully, the user can either expand the functionalities of the program or write a new interface to accommodate the need their application.

More details to be provided.

5. Programming Guide

This chapter will mainly focus on programming the LBC, where we've provided an API library so that the user can perform fundamental maneuvers and controls. In addition, we've also created a demo interface program for the UBC, which demonstrates basic capabilities of interfacing with the LBC. This demo program is by no means comprehensive. The users can either expand its functionalities or build their own interface programs in a similar fashion.

5.1 The Lower Board Controller

A console program allowing programming multiple cars at once is in the works. At the mean time, the LM Flash Loader (from Luminary Micro) can be used to program one car at a time.

More details to be provided.

5.2 The Upper Board Controller Demo

The Xilinx Platform Studio (software) and the Xilinx Platform Cable USB (hardware) are needed to program the upper board.

More details to be provided.

5.3 Other Resources

A Wi232DTS evaluation program can be downloaded from Radiotronix for configuring the radio modules.

Contact Information

For more information, please contact info@anteroslab.com.