Wattbot:*nt*

Technical Notes

Jim Herd

April 2021

Contents

[Wattbot:*nt* 1](#_Toc68553592)

[1 Introduction 3](#_Toc68553593)

[2 System 3](#_Toc68553594)

[2.1 General system overview 3](#_Toc68553595)

[2.2 PORT number allocation. 5](#_Toc68553596)

[3 FPGA 6](#_Toc68553597)

[3.1 General FPGA overview 6](#_Toc68553598)

[4 Low-level control microprocessor 7](#_Toc68553599)

[4.1 Data packets between microprocessor and FPGA. 7](#_Toc68553600)

[4.2 Process structure of µP software : GitHub Tag V1.0 8](#_Toc68553601)

[4.3 Process structure of µP software : GitHub Tag V2.0 9](#_Toc68553602)

[5 High-level processor 10](#_Toc68553603)

[5.1 Overview 10](#_Toc68553604)

[5.2 Communications protocol 10](#_Toc68553605)

[Appendices 13](#_Toc68553606)

[Appendix A :: System constants and constraints 13](#_Toc68553607)

[Appendix B :: FPGA register allocation. 14](#_Toc68553608)

[Appendix C :: State machine to parse incoming ASCII coded command 16](#_Toc68553609)

[Appendix D :: text 19](#_Toc68553610)

Figures

[Figure 1 General system structure 3](file:///C:\jth\HW_new_robot\Wattbot_nt_Documentation\Wattbot_nt_technical.docx#_Toc68546361)

[Figure 2 General FPGA structure 5](file:///C:\jth\HW_new_robot\Wattbot_nt_Documentation\Wattbot_nt_technical.docx#_Toc68546362)

# 1 Introduction

This document brings together technical details relating to the design of Wattbot:*nt*.

# 2 System

## 2.1 General system overview

The system is split into three subsystems

|  |  |
| --- | --- |
| Name | Responsibilities |
| FPGA | * high speed time dependent digital signals. |
| Low level controller  (LLcontrol) | * management of interaction with the FPGA * Real-time clock * Complex bus peripherals   + I2C   + CANbus |
| High level Controller  (HLcontrol) | * Implementation of high level system functionality   + Vision   + Planning   + Plan execution   + High level world interaction |

Commands from HLcontrol are managed by LLcontrol for execution on the FPGA on within the LLcontrol system.

Figure 1 shows the general structure of the system.

This three level structure aims to hide as much of the low level detail from designers interested in high level functionality. However, if necessary, users have access to the code of the lower two levels (SystemVerilog and C/C++) and may make changes and additions if appropriate.

|  |  |
| --- | --- |
| **Warning** | Modification of the code of the lower two levels will require significant investment in time to understand the intricate details of software, and the software tools required to implement any changes. |

CANbus bus

FPGA

H-bridge

(opt)

H-bridge

(opt)

H-bridge

(opt)

H-bridge

(opt)

Encoders

RC servo

RC servo

RC servo

RC servo

RC servo

RC servo

RC servo

RC servo

Low level control (LLC)

Microprocessor

High speed bus

Serial comms channel

High level

Control (HLC) computer

e.g. Raspi

Real-time

clock

Dynamixel

servo bus

I2C

sensors

Motion

Control

System

H-bridge

(opt)

H-bridge

(opt)

H-bridge

(opt)

H-bridge

(opt)

External

H-bridge

Internal

H-bridge

DC motors

Figure 1 General system structure

## 2.2 PORT number allocation.

When a command is sent to the FPGA it has an allocated PORT number, rather like the TCP/IP port number. This is the number that ensures that the reply is sent to the correct queue. This allows commands to come from a variety of sources e.g. PID commands from within the uP and remote commands from the High Level controller.

PORT numbers are represented by an unsigned 8-bit value with a range from 0 to 255.

Allocation is as follows

|  |  |
| --- | --- |
| PORT | Description |
| 0 | Reserved. |
| 1 | Debug process in High Level controller   * Any process within the uP can send messages to this PORT. |
| 2 🡪 19 | Used by processes in the Low Level controller (uP) |
| 20 🡪 255 | Used by processes in the High Level controller (e.g. PC, Raspbery Pi, etc) |

# 3 FPGA

## 3.1 General FPGA overview

The FPGA is structured as a peripheral system that is accessed by reading and writing to registers. There are two busses

1. An external 8-bit bi-directional bus connects the uP to the FPGA. This is a handshaked buss that run at a rate of about 100,000 transactions per second.
2. An internal 32-bit bi-directional bus that links all the peripheral subsystems to the uP.

An FPGA state machine (name…) performs all the necessary conversions between the two busses.

FSM = Finite State Machine

µP

FPGA

other functions

8-bit bus

32-bit bus

Low level

microprocessor

Digital I/O

Figure 2 General FPGA structure

The FPGA looks like a set of 256 32-bit I/O registers. For a fixed SysyemVerilog compilation, the registers will be defined. However, if the compilation parameters that define the number of subsystems are changed (eg the number of PWM units) then the register addresses will change. This is not a problem as the number of subsystems is defined in register 0 and should be the first register to be read to allow internal addresses to be pre-calculated.

Register addressing is shown in Appendix I.

# 4 Low-level control microprocessor

## 4.1 Data packets between microprocessor and FPGA.

The data interface between the low level microcontroller is an 8-bit bidirectional bus. A state machine in the FPGA converts between this 8-bit bit and the internal 32-bit bus.

The microprocessor accesses the FPGA as a set of registers. The current system has 256 available registers (0-<255). At this time, only bits 0 and 7 of the command byte are used. Bit 0 is a READ/WRITE bit and bit 7 is a RESET bit. The 6-byte microprocessor to FPGA packet is as follows (shown in C struct format for documentation purposes)

**typedef** **struct** {

**cmd\_t** command;

**uint8\_t** register\_number;

**uint32\_t** data;

} uP\_to\_FPGA\_packet\_t;

**typedef** **struct** {

**int** R\_W :1;

**int** spare :6;

**int** reset :1;

} cmd\_t;

The data returned from the FPGA can be set at compile time to be one or two 32-bit words. The first word is data and the second word is status.

However, there is very little status data that can be returned therefore this 32-bit value predominately zero, resulting in a waste of time and bandwidth on the uP/FPGA bus. The current system defaults to just sending the data. If you wish to use the status word that the FPFA and uP code must be recompiled with the appropriate compile time definitions.

**#ifdef INCLUDE\_32\_BIT\_STATUS\_RETURN**

**typedef** **struct** {

**uint32\_t** data;

**uint32\_t** status;

} FPGA\_to\_uP\_packet\_t;

**#else**

**typedef** **struct** {

**uint32\_t** data;

} FPGA\_to\_uP\_packet\_t;

**#endif**

The current code implementation allocates space for the status word but does not load it with any data.

How can status get to the uP from the FPGA when no status word is sent?

1. Each FPGA subsystem has a status register which can be read. Refer to register bit definitions.
2. There is a single digital status line connecting the FPGA to the uP. It is asserted low and can be used by a subsystem to indicate a problem.

## 4.2 Process structure of µP software : GitHub Tag V1.0

The uP runs the MBED RTOS software. The initial test structure uses the following structure with three processes (threads) and two mailbox queues.

FPGA

Com

Port

High level

controller

P3

P2

Low Level controller (µP)

P1

Queue1

Queue2

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Program name** | **Notes** |
| **P1** | Process | read\_from\_HLcontrol\_task | * Read ASCII command string from controlling computer * Send FPGA related commands to FPGA command queue * Implement non-FPGA commands |
| **P2** | Process | write\_to\_HLcontrol\_task | * Take ASCII reply strings from reply queue and send to controlling computer |
| **P3** | Process | FPGA\_IO\_task | * Take FPGA command from FPGA command queue and Implement on the FPGA through the 8-bit bi-directional bus. |
| **Queue 1** | Mailbox | FPGA\_cmd\_queue | * FIFO queue of LLcontrol (uP) commands coded as ASCII strings |
| **Queue 2** | Mailbox | HLcontrol\_reply\_queue | * FIFO queue of binary coded FPGA register commands |

This structure provides the basic system to test and exercise the FPGA hardware. The ASCII command format allows easy access from a high level control computer e.g. PC, Raspberry Pi, etc.

Testing uses a windows laptop with a C# program (March 2021).

## 4.3 Process structure of µP software : GitHub Tag V2.0

(30/3/21 Design only)

Added process (P4) to V1.0 design that can initiate and control a sequence of FPGA commands, e.g., execute timed reading of an encoder channel as part of testing. Works by injecting FPGA commands into “Queue 1” and receiving results through “Queue 4”. A specific PORT number would be assigned to this activity.

FPGA

Com

Port

High level

controller

P3

P2

Low Level controller (µP)

P1

Queue1

Queue2

P4

Queue3

Queue4

Additional objects to V1

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Program name** | **Notes** |
| **P4** | Process | sequence\_task | * Implements sequence of FPGA commands, specifically for test purposes. |
| **Queue 3** | Mailbox | sequence\_cmd\_queue | * FIFO queue of LLcontrol (uP) sequence commands |
| **Queue 4** | Mailbox | sequence\_reply\_queue | * FIFO queue of results from executed FPGA commands |

# 5 High-level processor

## 5.1 Overview

The FPGA and the uP organize the low level activity of the control of the robot. As specialized systems, the general user will not be able to change the chips used or the software burnt into these chips. Beyond this level, the user can choose whatever system is convenient, as long as they adhere to the communications protocol over a serial COM port. Example options include

* Raspberry Pi (any variant)
* Arduino or other embedded processor
* PC, MAC, etc

The hardware requirements are as follows

1. Real or virtual COM port
2. Any reasonable baud rate (default set to 256Kbaud)
   * Hardwired into current uP system.
3. 8-bit, no parity format

The serial COM port was chosen as it remains one of the most common low speed communication busses. Speed is not really an issue, as all the hard real-time activity is handled by the FPGA and the uP.

## 5.2 Communications protocol

Protocols come in many formats and complexity depending on the requirements of speed, functionality, and security. In this application, a simple command/reply format is sufficient.

* Opted to use ASCII strings to encode both the command and the reply
  + Surprisingly efficient given that the strings can be variable in length. Small numbers use few bytes.
  + Encoding/decoding is fast compared to transmission time
  + Human readable. Terminal programs can be used to test interactions.

A command has the following format

command\_letter port p\_0 p\_1 … p\_N **TERM**

|  |  |
| --- | --- |
| Element | Description |
| command\_letter | * Single command letter * Terminated with 1 or more spaces * Additional characters can be used but all accept the first letter is discarded. |
| port | * Port number to allow reply to be routed to the correct process. |
| p\_0 p\_1 … p\_N | * Set of number and text parameters terminated by one or more spaces. * Parameters can be   + Strings   + Integers (base 10)   + Reals (format x.y only) * Incorrect formats will be flagged as errors * Final parameter does not need a trailing space, but if suppled will be ignored. * The parser will flag |
| **TERM** | * String terminator can be any of the following   + Carriage return   + Linefeed   + Carriage return + linefeed |

1. Refer to Appendix A for current constraints

The reply has the format

port d\_0 d\_1 … d\_N **TERM**

|  |  |
| --- | --- |
| Element | Description |
| port | * Port number to allow reply to be routed to the correct process. |
| d\_0 d\_1 … d\_N | * Data items returned from execution of a command * Set of number and text parameters terminated by one or more spaces. * Parameters can be   + Strings   + Integers (base 10)   + Reals (format x.y only) |
| **TERM** | * String terminator can be any of the following   + Carriage return   + Linefeed   + Carriage return + linefeed |

# Appendices

## Appendix A :: System constants and constraints

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Name | Value | Date |
| Maximum characters in HLcontrol to LLcontrol command | MAX\_COMMAND\_LENGTH | 100 | 2/4/21 |
| Maximum parameters in a HLcontrol to LLcontrol command | MAX\_COMMAND\_PARAMETERS | 16 | 2/4/21 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Appendix B :: FPGA register allocation.

The internal register address is represented by an unsigned 8-bit integer (range 0 to 255).

Register 0 is a special read-only register that contains relevant system information. This is always read first by the uP to allow subsequent register addresses to be pre-calculated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Register** | **Address** | **Eg (1)** | **Notes** |
| **SYS\_INFO\_0** | 0 | 0 | Some basic information : Version, number of PWM/QE/RC subsystems |
|  |  |  |  |
| **PWM\_PERIOD** | PWM\_base + 0 | 1 | Period in units of 20 nanoseconds |
| **PWM\_ON\_TIME** | PWM\_base + 1 | 2 | PWM on-time in units of 20 nanoseconds |
| **PWM\_CONFIG** | PWM\_base + 2 | 3 |  |
| **PWM\_STATUS** | PWM\_base + 3 | 4 |  |
| **. . . . . . . . . . . .** |  |  |  |
| **PWM\_PERIOD** | PWM\_n + 0 | 5 | "n"th PWM subsystem |
| **PWM\_ON\_TIME** | PWM\_n + 1 | 6 |  |
| **PWM\_CONFIG** | PWM\_n + 2 | 7 |  |
| **PWM\_STATUS** | PWM\_n + 3 | 8 |  |
|  |  |  |  |
| **QE\_COUNT\_BUFFER** | QE\_BASE + 0 | 9 | Count of decoded encoder pulses |
| **QE\_TURN\_BUFFER** | QE\_BASE + 1 | 10 | Count of encoder index pulses |
| **QE\_SPEED\_BUFFER** | QE\_BASE + 2 | 11 | Number of 20 nanosecond counts during single encoder pulse |
| **QE\_SIM\_PHASE\_TIME** | QE\_BASE + 3 | 12 |  |
| **QE\_COUNTS\_PER\_REV** | QE\_BASE + 4 | 13 | Number of encoder counts in a 360 degree turn |
| **QE\_CONFIG** | QE\_BASE + 5 | 14 |  |
| **QE\_STATUS** | QE\_BASE + 6 | 15 |  |
| **. . . . . . . . . . . .** |  |  |  |
| **QE\_COUNT\_BUFFER** | QE\_m + 0 | 16 | "m"th quadrature encoder subsystem |
| **QE\_TURN\_BUFFER** | QE\_m + 1 | 17 |  |
| **QE\_SPEED\_BUFFER** | QE\_m + 2 | 18 |  |
| **QE\_SIM\_PHASE\_TIME** | QE\_m + 3 | 19 |  |
| **QE\_COUNTS\_PER\_REV** | QE\_m + 4 | 20 |  |
| **QE\_CONFIG** | QE\_m + 5 | 21 |  |
| **QE\_STATUS** | QE\_m + 6 | 22 |  |
|  |  |  |  |
| **RC\_SERVO\_PERIOD** | RC\_BASE+ 0 | 23 | Period in units of 20 nanoseconds |
| **RC\_SERVO\_CONFIG** | RC\_BASE+ 1 | 24 |  |
| **RC\_SERVO\_STATUS** | RC\_BASE+ 2 | 25 |  |
| **RC\_SERVO\_ON\_TIME + 0** | RC\_BASE+ 3 | 26 | Servo 0 pulse on-time in units of 20 nanoseconds |
| **RC\_SERVO\_ON\_TIME +1** | RC\_BASE+ 4 | 27 | Servo 1 pulse on-time in units of 20 nanoseconds |
| **. . . . . . . . . . . .** | . . . . |  | . . . . . . . . . . . |
| **RC\_SERVO\_ON\_TIME + 7** | RC\_BASE+ 10 | 33 | Servo 7 pulse on-time in units of 20 nanoseconds |

Notes.

1. The Eg column shows the register address allocationfor a system with TWO pwm units, TWO quadrature encoder units, and EIGHT servo units.

Given the information from register 0, base addresses of each type of subsystem can be calculated. The following gives the general idea. This is built into the uP software.

PWM\_base = 1;

QE\_base = ((NOS\_PWM\_REGISTERS \* NUMBER\_OF\_PWM\_CHANNELS) + PWM\_base;

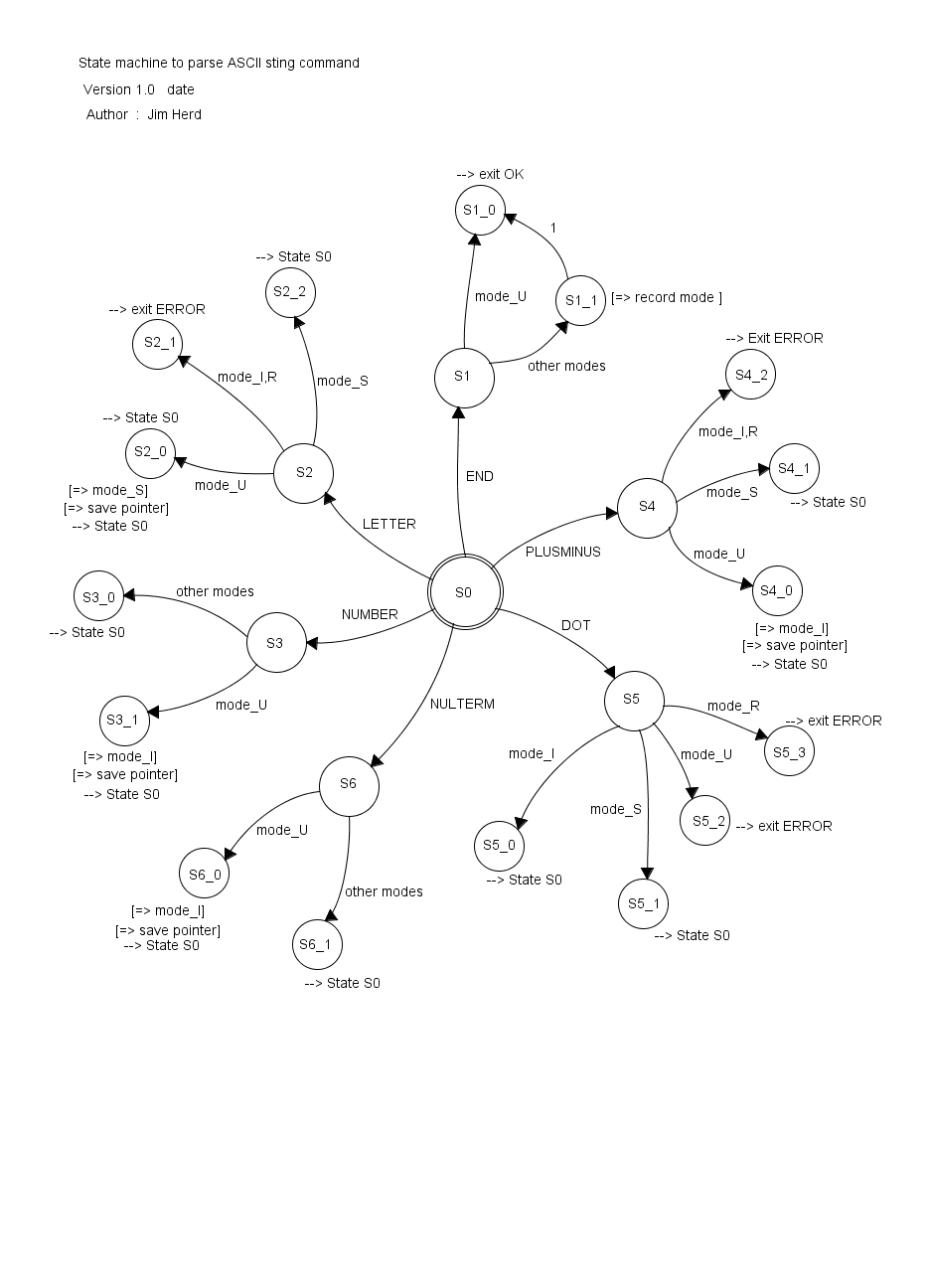
RC\_base = ((NOS\_QE\_REGISTERS \* NUMBER\_OF\_QE\_CHANNELS) + QE\_base;

## Appendix C :: State machine to parse incoming ASCII coded command

The command sent from HLcontrol to LLcontrol has the format

letter parameter\_1 parameter\_2 …. parameter\_n TERM

A parsing program that analyses this string is described with the following state machine.



The following is the code that implements the state machine

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// parse\_command : analyse command string and convert into labelled strings

//

// Breaks the command string into a set of token strings that are

// labelled REAL, INTEGER or STRING.

//

// Code uses a STATE MACHINE to walkthrough the command string. Refer

// to associated documentation.

// defines modes as scan progresses : U=undefined, I=integer, R=real, S=string

//

int32\_t parse\_command (void)

{

int32\_t count, mode, status;

uint8\_t character\_type;

argc = 0;

mode = MODE\_U;

status = NO\_ERROR;

for (count=0 ; count <= character\_count ; count++) {

character\_type = char\_type[command[count]];

switch (character\_type) {

case LETTER :

if ((mode == MODE\_I) || (mode == MODE\_R)) {

status = LETTER\_ERROR;

} else {

if (mode == MODE\_U) {

mode = MODE\_S;

arg\_pt[argc] = count;

// argc++;

}

}

break;

case NUMBER :

if (mode == MODE\_U) {

mode = MODE\_I;

arg\_pt[argc] = count;

// argc++;

}

break;

case DOT :

if (mode == MODE\_I) {

mode = MODE\_R;

} else {

if ((mode == MODE\_R) || (mode == MODE\_U)) {

status = DOT\_ERROR; // extra point in real value

}

}

break;

case PLUSMINUS :

if (mode == MODE\_U) {

mode = MODE\_I;

arg\_pt[argc] = count;

//argc++;

} else {

if ((mode == MODE\_I) || (mode == MODE\_R)) {

status = PLUSMINUS\_ERROR;

}

}

break;

case NULTERM :

if (mode != MODE\_U) {

arg\_type[argc++] = mode;

mode = MODE\_U;

}

break;

case END :

if (mode != MODE\_U) {

arg\_type[argc++] = mode;

mode = MODE\_U;

}

break;

} // end of SWITCH

} // end of FOR

return status;

}

## Appendix D :: text