Technical Report

IRI-TR-10-05

Slave architecture for the Robonova MR-C3024 using the HMI protocol IRI Technical Report

codename *Hydrozoa*

Edgar Simó Serra
Jordi Pegueroles Queralt

Institut de Robòtica i Informàtica Industrial

Abstract

The goal of the project is to develop a new firmware for the servo control board **Hitec MR-C3024** [8] to improve its specifications. This board will be used to control the movement of a humanoid robot, driven by digital servos using the communications protocol **HMI**, developed by Hitech.

Institut de Robòtica i Informàtica Industrial (IRI)
Consejo Superior de Investigaciones Científicas (CSIC)
Universitat Politècnica de Catalunya (UPC)
Llorens i Artigas 4-6, 08028, Barcelona, Spain

Tel (fax): +34 93 401 5750 (5751)
`http://www.iri.upc.edu`

Corresponding author:

E. Simó
tel: +34 93 405 4490
`esimo@iri.upc.edu`
`http://www.iri.upc.edu/staff/esimo`

Contents

List of Figures

1 Introduction

For a robot to move well, it must have a highly synchronized software and hardware system. In the case of humanoid robots, this need is even more accentuated, because for the robot to appear and move like a humanoid, it must be very fluid.

The Humanoid Lab Project currently (year 2010) uses a board developed by Hitech in order to control the motors of the robot. While this board is effective when used as an autonomous board, it is lacking when used as a slave. When the Humanoid Lab Project added an on-board computer to deal with inverse kinematics and other high level processes major flaws were found for this functionality in the slave board. To overcome these problems it was proposed to design a minimal electronics and software system to overcome these problems.

The most important problems with the MR-C3024 [8] lay in the fact that it's meant to work as a host and not a slave. When tried to use in slave configuration it generally displays really slow communication and low reliability. It can not give motor feedback nor stop the motors once they have been made to move. This makes it very limited as a slave. As a host it has similar limitations. It must be programmed in Robobasic [6] which is a generally inflexible language with many limitations. This makes things like inverse kinematics nearly impossible to implement on the MR-C3024 [8] and the impossibility to take into account movement dynamics.

The name Hydrozoa comes from the animal which generally consist of many polyps around a central cavity that are related to jellyfish. If we think of the polyps as servos and the central cavity as the MR-C3024 [8] it conveys the message of a group of servos working together like a Hydrozoa.

The Hydrozoa Projects aims to replace the firmware of the Hitec MR-C3024 board [8] to augment it's capability as a slave device. It will make the most of the hardware available and require the minimum amount of support electronics for the final implementation. A major feature is instead of using the traditional PWM interface of the servos, it uses the HMI interface of the HITEC Digital Servos like the HSR-8498 Servo [7]. This allows servo feedback when doing inverse kinematics. This will all be done following the philosophy of open source. The openness is fundamental since allows other users to learn how it is implemented while at the same time allow them to solve possible issues and add new features in the future.

This technical document is split into three parts: hardware design [section 2], software design [section 3] and testing [section 4] for easier reading. The section 2 goes into detail of the support electronics needed to be able to use the Hydrozoa firmware. The technical details of the software implementation is explained in section 3. The explanation on the entire set up and the results of the Hydrozoa firmware are explained in section 4.

2 Hardware Design

The HMI interface used by digital servios requires a small interface circuit. For this a small electronic support board was designed that implements the serial TTL to HMI interface. This design allows servos to be connected by daisy chaining which also requires more headers to connect the servos.

2.1 HMI interface

The work carried out is a continuation of a reverse engineering work started by Richard Ibbotson [3]. The HMI communication protocol is based on a mono-channel bidirectional serial communication. To achieve this, a logical interface must be developed. See figure 1 for an implementation example.

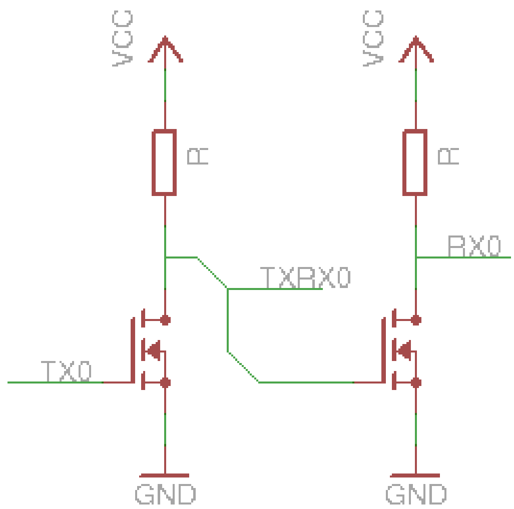


Figure 1: Bare HMI interface.

When none of the devices is sending data, the bus is pulled up by a resistor. Only one device can send data at the same time. To start the communication, a NUL command (7 null characters) must be send, in order to sync host and slave devices.

Please note that the resistor value in figure 1 is not defined. The choosing of the resistor value depends on the number of servos connected. The reason is that each servo has an internal resistance. This makes it so that the "servo resistance" decreases as you add more servos (equation (1), causing the voltage to lower. Once the voltage goes below the threshold of detection of the microcontrollers communication will no longer work. This can be explained by equation (2), where R_{pullup} is the value of the pullup in figure 2.

$$R^T_{servo} = \frac{R_{servo}}{N_{servos}} \tag{1}$$

$$V_{servo} = V_{cc}\frac{R^T_{servo}}{R^T_{servo} + R_{pullup}} \tag{2}$$

The threshold of correct operation is about 2.5 V. This means that below 2.5 V the servos will not communicate properly. With an $R_{pullup} = 10\,\text{kOhm}$ we get the results of table 1. We can use this table and the equation (2) to calculate the resistance of the servo which is $R_{servo} \approx 30\,\text{kOhm}$. If we wish to use 10 servos on bus, using equation (2) again we get that the

Table 1: Input voltage to the servos depending on the number of servos connected.

Input Voltage	Nmuber of servos connected
3.75 V	1
3.00 V	2
2.50 V	3
2.25 V	4

pullup should be $R_{pullup} < 3\,\text{kOhm}$. This value depends on how many servos will be connected, however a value of 1 kOhm is a safe value for the pullup.

2.2 Expansion Board

To connect all servos in bus mode, a simple board was developed to allow easy and fast connections. It has to independent buses, powered directly from the main battery. Each bus has it's own HMI interface to connect the servos data line to the microcontrollers serial ports.

In figure 2 the schematic of the simplest HMI interface implementation for the MR-C3024. Interface's logic is powered by the robot's main battery through a linear regulator. It has an header for connecting the serial ports of the MR-C3024 board and headers for connecting all the servos. It is using a 1 kOhm pull up for

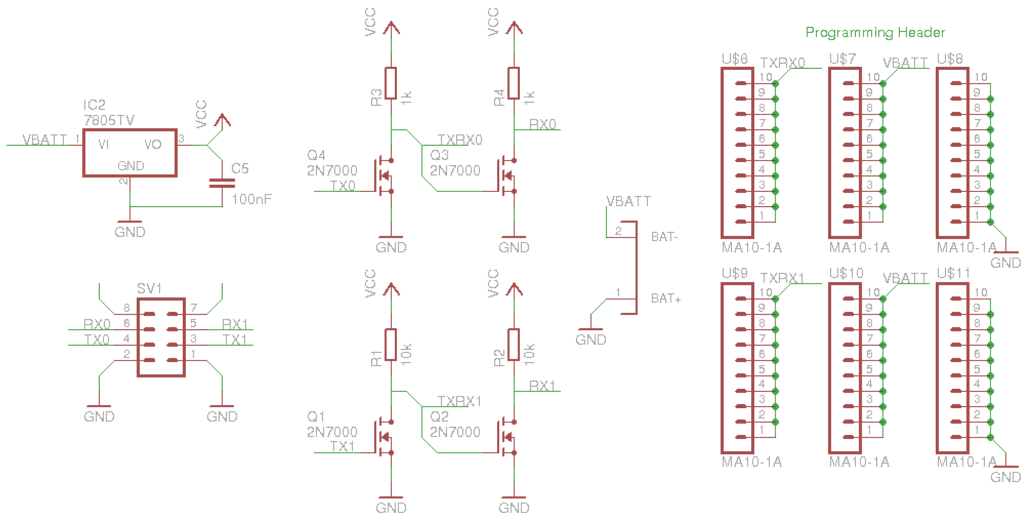


Figure 2: MR-C3024 expansion board schematic with 1 kOhm pullups.

An example layout is shown in figure 3. This example uses all through-hole components and is easy make and solder. However in most applications a more complex version that mates with the MR-C3024 board would be advisable. The amount of headers to connect servos to is also adjustable depending on the amount of servos intended to connect to it.

The architecture of the final layout showing hardware is shown in figure 4. It can be seen how the MR-C3024 board is connected with RS-232 to the Gumstix which acts as a host. The MR-C3024 is also connected to the expansion board from figure 3 by use of TTL serial ports. The expansion board is connected to all the servos using the HMI protocol. An important note is that the power supply for logic and servos are seperate. The logic all runs on a single cell lithium battery (3.7 V) while the servos run off a more powerful 7.4 V battery. This is to avoid the servo power spikes on the logic power lines. It help keep the voltage constant and avoids possible issues caused by large power spikes.

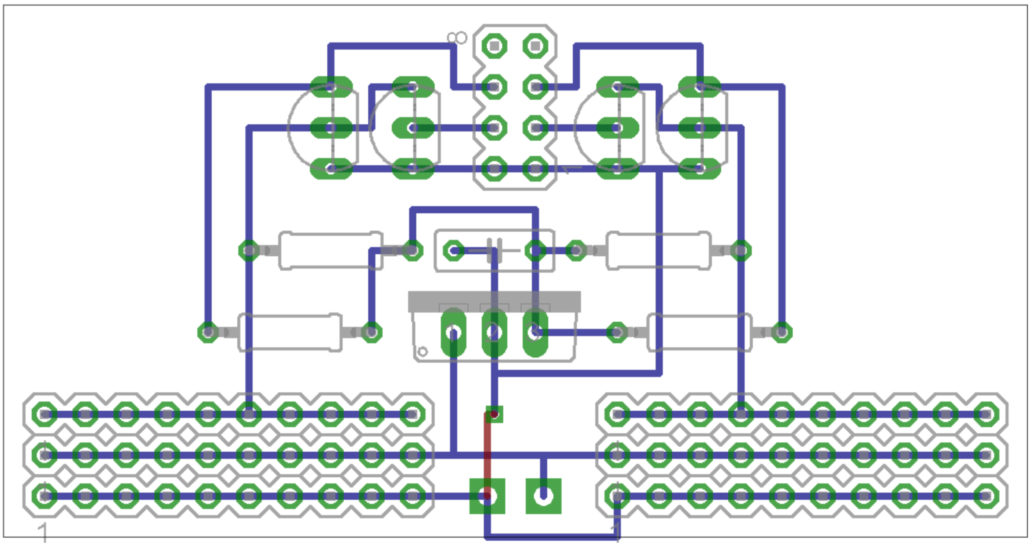


Figure 3: MR-C3024 expansion board layout

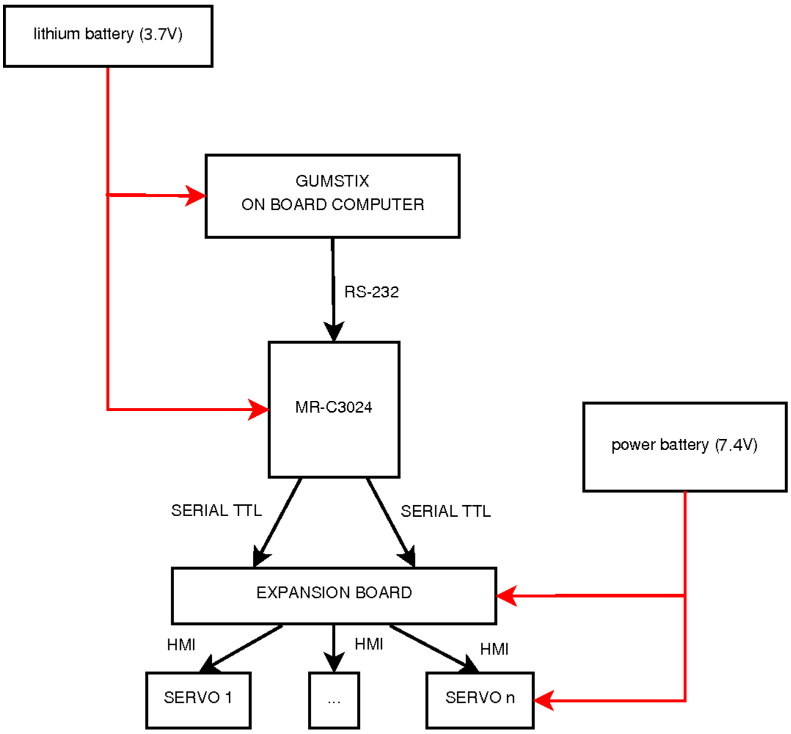


Figure 4: Robot's architecture

3 Software design

The objective of the software is to allow fast precise control of all the servos with position feedback. The servos would also have to be synchronized into movements to allow smooth fluid motions. The software should also expose as much of the HMI API available, however there are only few commands that work when controlling HITEC servos in bus mode. The available commands are:

1. Set target positions: Sets the target position for a specific servo to move to.
2. Set speed and read position: This hybrid command sets the speed of a specific servo and reads it's current position.
3. Set go/stop: This command affects all servo on the bus and can either stop them temporarily or tell them to resume their movement.
4. Release: This command releases the servo so that they can move freely.

The only ones used by the Hydrozoa firmware currently are the commands that affect individual servos. That is the set target positoin and set speed and read position. These commands allow the firmware to move groups of servo while doing feedback.

3.1 Software layout

The Hydrozoa software is divided into modular subsystems to allow maximum flexibility and simplicity. It is made up of the following subsystems:

- Communication subsystem [3.2]: This subsystem handles the communication with the host by processing commands.
- Servo subsystem [3.3]: This subsystem handles communication on a servo level. It also handles
- Group subsystem [3.4]: This subsystem handles creating groups.
- Scheduler subsystem [3.5]: This subsystem is handles all periodic tasks.

In figure 5 we can see the different interactions between the different subsystems. The interaction between the communication subsystem and the group or servo subsystem depends on what mode the controller is set. More details on the different modes in section 3.4.

3.2 Communication subsystem

The communication subsystem is in charge of communication with the host device. It handles the changes of modes and converts all the host commands into function calls to the different subsystems. This is all done at 57600 Baud which is above the MR-C3024's original communication speed and three times faster then the servo communication speed.

The communication is primarily stream based. It uses variable-length commands that it processes as it receives. The replies are asynchronous also so if you send two commands you might receive replies in a different order. Which is why you have to process each command individual and not make assumptions on the order of the packets.

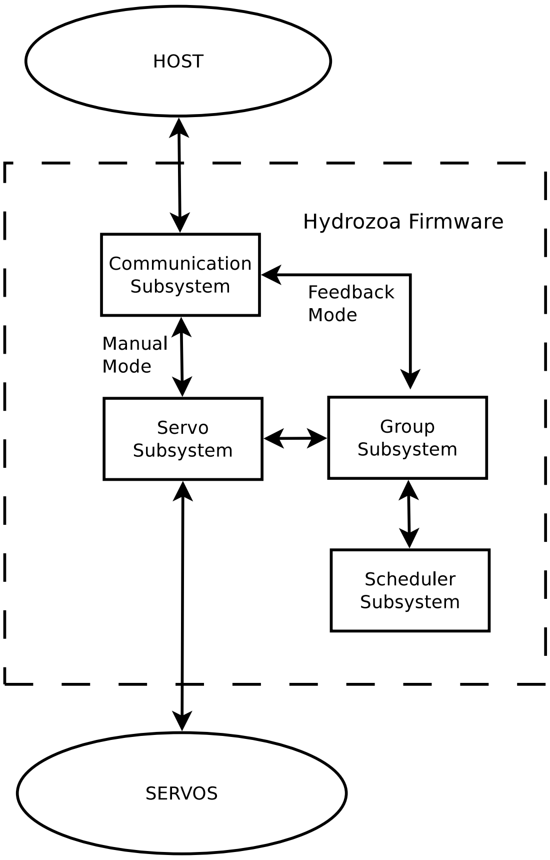


Figure 5: Overview of interactions between the different software subsystems.

3.3 Servo subsystem

The servo subsystem handles commands to individual servos. It is the lowest layer built around the HMI protocol. It is designed only to reflect the commands the servos accept.

The ATmega128 MCU on the MR-C3024 board only has 2 hardware UARTs available. Since the board has to communicate with the host device (Gumstix PXA270) by rs232 that only leaves 1 hardware UART available. To be able to work with position feedback, the communication speed would have to be optimized to allow as fast as possible feedback updates.

The HMI interface runs at 19200 BAUD with 2 stop bits. Each servo packet is 7 bytes in length. We have to control 24 servos. That means that the maximum rate would be:

$$f_{feedback} = \frac{f_{baud}}{N_{servo} * L_{packet} * L_{bits}} = 10.39\,\mathrm{Hz} \simeq 10\,\mathrm{Hz}$$

In order to increase the feedback speed to 20 Hz it was decided to split the 24 servos into two groups of 12. The more used group of 12 would use the hardware UART while the other group of 12 would use a software UART. In order to work with both hardware and software UART an abstraction layer was created to allow access to any type of UART independently of it's internal workings.

```
/**
 * @brief Represents an uart.
 */
typedef struct Uart_s {
   void (*putc)(uint8_t c); /**< Puts a character on the UART. */
   int (*status)(void); /**< Checks to see if outgoing buffer is
                              empty, returns 1 if empty. */

   /* Recieve buffer stuff. */
   int pos; /**< Current position in the recieving buffer. */
   uint8_t buf[6]; /**< Buffer containing recieving data. */
} Uart_t;
```

The abstraction only has functions for writing to the UART and checking it's status. It's important to be able to check the UART status, because if the line hasn't been communicating for a while the servos will desync with the MR-C3024. To avoid desynchronization they need to be sent 7 zero bytes before the actual package. Otherwise they may lose bytes causing the communication to desynchronize and lose a command. Before sending any command the software will check to see if the outgoing buffer is full, if it is not it will send the sync bytes before sending the actual command.

Another important thing to note is that the abstraction automatically handles the reply using a small buffer with just enough information to contain each reply. This allows communication to work transparently with many virtual UARTs.

3.4 Group subsystem

The group subsystem handles creating groups of servos. The Hydrozoa software allows two different control modes: either individual servo control or group-based control. The individual mode is a simple way of having control of the servos. It only exposes the servo commands. The group-based control mode is meant for synchronizing movements among groups of servos. It is the most useful for the control of humanoid type robots.

The main feature of groups is the synchronization, but they also have the added advantage of being able to get feedback really fast. When a group is moving it will automatically be polled for the servo positions which allows low-latency access to current servo positions at any given moment. It will also send an event that allows interrupt-based detection of groups finishing movements. The procedure of doing a synchronized group movement is shown in figure 6.

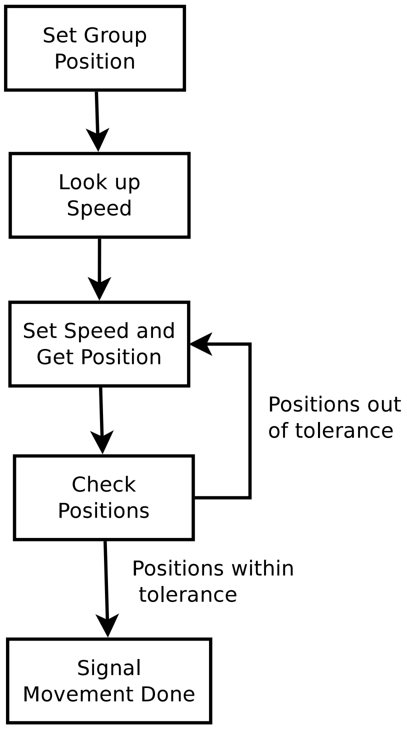


Figure 6: Overview of group synchronization.

Synchronization originally posed a problem as feedback is slow and the main processor of the MR-C3024 is too slow to be able to do complex calculations. After failed attempts to do speed-based position control a simple solution appeared. It was decided to use look-up tables to find the proper speed command for a target speed. When a group receives a move command it finds the longest distance a servo has to move and uses that as a reference time to finish movement in. It then calculates the speed at which all the other servos of the group have to move at to be able to finish the movement in that time. The speed value is then found from a look-up table which converts the degrees/second calculated to an 8-bit value representing the servo velocity to move at.

The look-up table is generated automatically by the firmware. It is done by a helper application called calcspeed (section C.3 which uses the individual servo control mode to move the servo back and forth at different speeds. It then calculates the real world value of those speeds and outputs them into a csv file. This file can then be used to generate a python function that will then create the C-code for the look up table. This C-code is used directly in the servo control subsystem of the software.

3.5 Scheduler subsystem

The scheduler subsystem is very simple in this case. The only situation in which periodic tasks are needed is when doing group-based feedback control. The servo feedback is handled by a periodic task which constantly polls all the servos of every moving group to see if they have reached their destination yet.

The other functionality of the scheduler subsystem is to minimize power consumption by the microcontroller. This is done by using sleep modes whenever the microcontroller is not running.

However power consumption of the microcontroller is usually negligible when compared to the power consumption of the servos.

3.6 Host High level API

The high level API is built around the CRobot [9]. It wraps around the lower level firmware commands in a more flexible and dynamic matter. This also allows error detection and correction. However the main advantage is being able to design and write applications using the firmware functionality much faster than otherwise.

The concept of the high level API is basically based around the entire group API. The proper way to use the API is to follow some simple steps:

1. Initialize the class.
2. Create the servo group.
3. Set the servo group settings (optional).
4. Issue group commands.

The API focuses on groups and using them. Some of the more important functions are:

- **get_maxGroup:** Gets the maximum available amount of groups.
- **grp_create:** Creates a group of servos.
- **grp_destroy:** Destroys a group of servos.
- **grp_move:** Moves a group of servos.
- **grp_stop:** Stops the movement of a group.
- **grp_waitDone:** Waits until a gorup finishes moving.
- **grp_setTol:** Sets the tolerance of a gorup.
- **grp_isDone:** Checks to see if a group is done moving.
- **grp_feedback:** Gets the position feedback for a gorup.
- **grp_setSpeed:** Sets the maximum group target speed.

3.7 Unit Tests

To be able to rapidly test the functionality of the firmware and guarantee that it is working perfectly some unit tests were designed. These are automated tests that allow quick testing of all the features of the firmware. If anything goes wrong or there is any issue with the firmware it will be detected and the developer will be informed to be able to correct this. This is very important to be able to have a robust and reliable firmware.

The unit tests consist of a single application that runs through the entire API testing all the different commands and verifies the execution of them. The single application has been called 'testfirmware' (see section C.2) and is available in the source code repository. The execution is straight forward, the electronics board has to be plugged in with some servos connected. Afterwards the testfirmware command is executed with the path to the rs232 port as a parameter and it will begin execution. If all is successfully it will run through all the commands and print a message indicating success.

4 Testing

To ensure the reliability and quality of the Hydrozoa firmware, a standard procedure and applications (see section C) for testing were developed.

4.1 Getting Started

The standard procedure for setting up servos to work with Hydrozoa firmware is the following:

1. Program servo ids with the setid command of testservo (see section C.1.3). Make sure the servo ids do not overlap.
2. Program the MR-C3024 board to use the Hydrozoa Firmware (see section B.4).
3. Connect the MR-C3024 board to the Expansion board.
4. Connect the servos to the expansion board. Try to distribute servos evenly on both UARTs. The most used servos should ideally go on the hardware UART.
5. Connect the MR-C3024 board to the Host.
6. Run tests (see section C.2) to ensure everything is working properly.
7. Start using the library (see section A).

These steps are all crucial to making sure everything is working properly.

4.2 Calibration

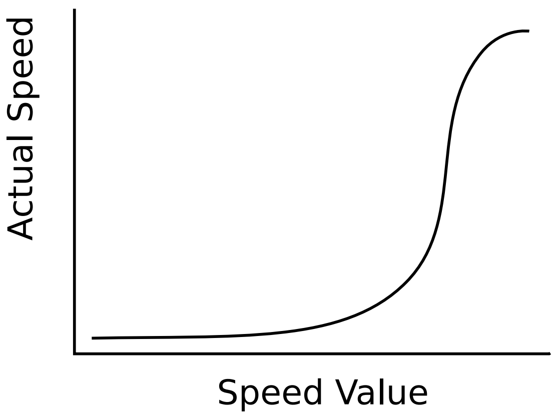


Figure 7: An approximation of the HSR-8498 servo speed curve

One of the important things is the calibration of servo speeds. Servos have different speed curves. A speed curve is the relation between the real world speed and the digital speed value the servo accepts. The original HSR-8498 [7] has a bad curve in the sense that it is strongly non-linear and the effective area is very small. Conceptually the bad speed curve can be seen in figure 7. However more expensive servos like the HSR-5498 have a much better speed curve as seen in figure 8. It has a large linear area that saturates near the maximum. This means there are many more useful values and a longer.

Usually the default values should be good enough for the synchronization of the servos in a group. However if problems are experienced with the synchronizations, the servos can be recalibrated using the calcspeed function which is explained in detail in section C.3. More details on the implementation of the group speed synchronization in section 3.4.

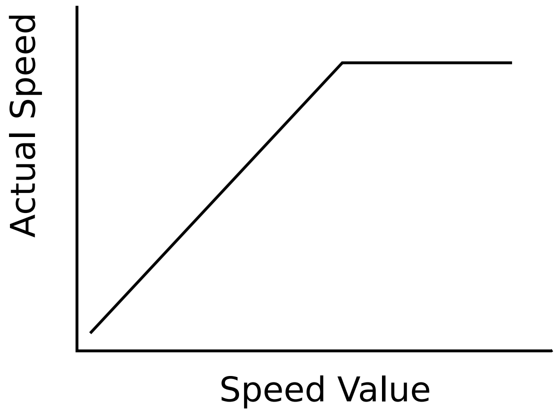


Figure 8: An approximation of the HSR-5498 servo speed curve

4.3 Experimental Setup

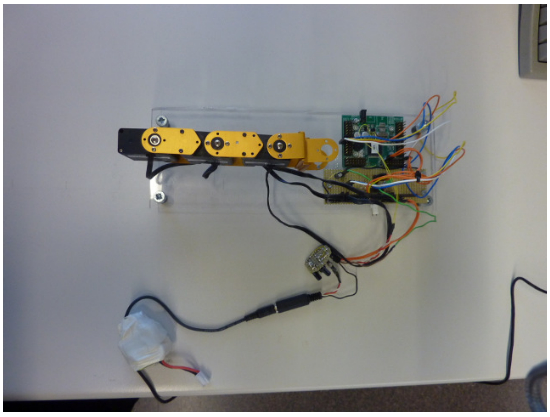


Figure 9: View of the Hydrozoa testing set up

For the testing and development of the Hydrozoa three servos were connected to both serial buses on a prototyped expansion board. The entire set up can be seen at figure 9. The three servos were set up so that the zeros align it fully straight.

A close up of the set up can be seen in figure 10. There is an regulator on the expansion board that supplies the MRC3024 board.

4.4 Results

A test application (see section C.5) was designed to be able to send movement commands while reading back the positions to make sure the groups were properly synchronized. This is done by sending a move group command followed by as many read group position commands as possible until a end movement command is received. The results can be seen at figure 11. The discrete steps of each request can be seen for each servo and they all converge to the target position at more or less the same time. This confirms that group synchronization and feedback with the Hydrozoa firmware works.

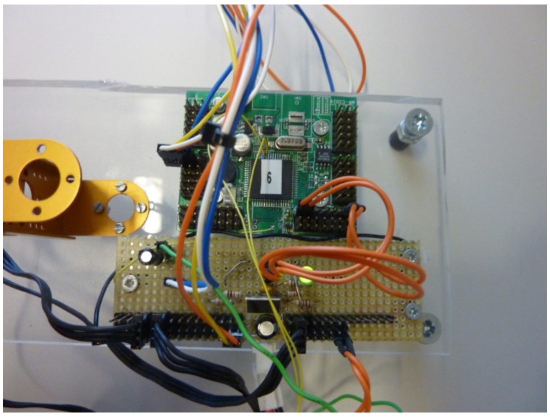


Figure 10: Close up view of the Hydrozoa testing set up

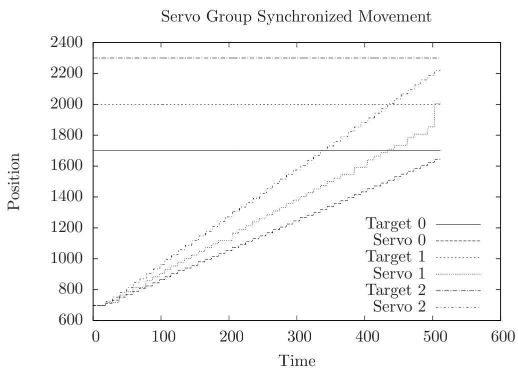


Figure 11: Results of the testplot application

5 Conclusions and Future Work

The implementation is working well. The replacement firmware is optimized for servo group synchronization and movement and is up to the task. Many features have been added to improve the handling of the servos. The resulting movement is smooth and fluid.

A movement subsystem has been also added to the CRobot library to allow easy usage of the firmware. In the repository there are examples in section A on how to use this subsystem to control some servos.

For the future it would be interesting to develop a new robust firmware uploader. The current one (section B) is a windows application that has some bugs that have not been addressed. For smoother movements, an internal queue could be added, so the microcontroller could chain synchronized movements. A flexible implementation of inverse kinematics for humanoids would also be interesting to integrate into the CRobot movement subsystem developed for this application. It would also be good to expose more functionality of the MR-C3024 board through the Hydrozoa firmware, like ADC ports, buzzer, multi-purpose i/o, etc... There is still room for improvement and evolution.

More information about this project can be found at the official project web page:
`http://apollo.upc.es/humanoide/`

References

[1] Various Authors. Finally cracked!
http://robosavvy.com/forum/viewtopic.php?t=271&postdays=0&postorder=asc&start=60, 2010.

[2] Generalitat de Catalunya. Generalitat de catalunya.
http://www.gencat.cat/, 2010.

[3] Richard Ibbotson. Hitec HSR-8498HB Digital Servo Operation and Interface.
http://robosavvy.com/Builders/i-Bot/HSR8498HB%20Servo.pdf, 2010.

[4] Institut de Robòtica i Informàtica Industrial. CSIC-IRI Website.
http://www.iri.upc.edu/, 2010.

[5] La Farga. Beques de programari lliure.
http://www.lafarga.cat/beques/, 2008.

[6] miniROBOT Corp. roboBASIC Home.
http://www.robobasic.com/, 2010.

[7] Hitec Robotics. HSR-8498HB HMI Standard Robot Servo.
http://www.hitecrcd.com/products/servos/robotics/hsr-8498hb.html, 2010.

[8] Hitec Robotics. Robot Controller - MR-C3024.
http://www.hitecrobotics.com/product/proB_02.html, 2010.

[9] The Humanoid Lab Team. CRobot Git Repository.
http://apollo.upc.es/gitweb/?p=crobot.git;a=summary, 2010.

[10] The Humanoid Lab Team. The Humanoid Lab Website.
http://apollo.upc.es/humanoide/, 2010.

A Using the CRobot Library

This is a guide on how to get started using the software here presented.

1. Get, compile and install CRobot library. In [10] is extensively explained.

   note: this library can be tested on any computer, laptop or PC embedded, that runs Linux.

2. Get and compile *hydrozoa* library. You can get it freely from:

   ```
   $ git clone git://apollo.upc.es/hydrozoa.git
   ```

   Read COMPILE to check it's dependencies and also a quick how-to.

3. Upload new firmware to MR-C3024 board. You can use a windows app. Found in RoboFlash directory inside *hydrozoa* repository. Just connect the serial cable as you'd do for uploading a robobasic program.

4. Go back to CRobot repository.

   ```
   $ cd test/momvement
   $ ./testmoves.bin
   ```

5. You should read "successfully moved all servos".

B MR-C3024 bootloader

Generally, a bootloader is a small program which runs at boot time and is capable of loading a complete application program into a processor's memory so that it can be executed. Note that the bootloader runs on the same processor into which it is loading a new program.

B.1 Why we need a bootloader?

In the case of AVR processors, the bootloader program is usually 256-4096 assembly instructions long and resides in a special portion of the FLASH memory called the bootblock. At boot time (when the processor has just been reset) the bootloader starts and is capable of communicating with the outside world to retrieve a new program and program it into the processor's FLASH memory. Depending on the bootloader and the available hardware, new application code can be loaded from any source including the serial port, SPI or I2C interfaces, external memory, hard disks, flash cards, etc. Once the programming is done, the bootloader program exits or the processor is reset, begins running the newly loaded code. Only AVR processors with the self-programming memory feature (those that have an SPM assembly instruction) can run a bootloader.

This is the case of the ATmega128, the brain of the MR-C3024. It has a proprietary bootloader protected with a custom serial protocol.

B.2 Reverse engineering

Thanks to the effort carried out the by people at robosavvy.com [1], the bootloader sequence was finally decrypted.

B.3 Bootloader sequence and algorithm

This is the bootloader algorithm, computer side. Note that is written in C#.

```
try
{
 CommPort.ReadByte();
 if (CommPort.ReadByte() == 0x3E)
 {
  byte[] Wakeup = { 0x3c, 0xF0, 0xA5, 0x5A, 0x0F, 0x80, 0x53 };
  CommPort.Write(Wakeup, 0, 7);
  byte[] Response = new byte[5];
  CommPort.ReadByte();
  CommPort.ReadByte();
  CommPort.ReadByte();
  CommPort.ReadByte();
  CommPort.ReadByte();
  textBox3.Text = "Starting program cycle" + Environment.NewLine;
  if (((FileLength ) % 256) == 0) Blocks = (FileLength) / 256;
  else Blocks = ((FileLength) / 256) + 1;
  textBox3.Text += Blocks + " Blocks to write" + Environment.NewLine;
  textBox3.Update();
  progressBar1.Minimum = 0;
  progressBar1.Maximum = Blocks;
  progressBar1.Value = 0;
  for (blockcount = 0; blockcount < Blocks; blockcount++)
```

```
  {
   progressBar1.Value = blockcount;
   BytePoint[0] = (byte)blockcount;
   BytePoint[1] = 0;
   Checksum[0] = (byte)BytePoint[0]; // Add high BytePoint
   // to checksum

   for (wordcount = 0; wordcount < 128; wordcount++)
   {
    DownLoad[wordcount * 2]         = FlashBuffer[(blockcount * 256) \
                                       + (wordcount * 2)];
    Checksum[0]                    += DownLoad[wordcount * 2];
    DownLoad[wordcount * 2]        ^= BytePoint[0];
    DownLoad[(wordcount * 2) + 1] = FlashBuffer[(blockcount * 256) \
                                       +  (wordcount * 2) + 1];
    Checksum[0]                    += DownLoad[(wordcount * 2) + 1];
   }


   CommPort.Write(BytePoint, 0, 2); // write BytePoint
   CommPort.Write(DownLoad, 0, 256);
   CommPort.Write(Checksum, 0, 1);

   CommPort.ReadByte();
   // should check here if the correct 0x21 ack is received
   // or process the 0x40 error  by resend / abort
  }
  progressBar1.Value = 0;
 }
 else textBox3.Text += "Bad Response from Robo Nova";
}
catch
{
 textBox3.Text += "NoResponse from RoboNova";
 textBox3.Text += "Port Closed" + Environment.NewLine;
 CommPort.Close();
 return;
}
textBox3.Text += "Port Closed" + Environment.NewLine;
textBox3.Text += "Program complete." + Environment.NewLine;
CommPort.Close();
}
```

B.4 Software Usage

A working application to upload binary code to the MR-C3024 can be found in the hydrozoa repository, and in the humanoid lab wiki [10]. A guide about how to use it is also located at the humanoid lab wiki [10].

Currently only works on Windows platforms, requiring .NET environment.

C Software Documentation

For testing purposes that are various applications for interacting with the newly developed firmware. All these applications unless otherwise specified are designed to run on Linux and in the console. When showing example usage, the '$' symbol represents the console prompt.

C.1 testservo

This application is for testing with the standalone servo board. The application provides direct calls for controlling all of the HMI servo commands. It also provides higher level functions like one for setting the ID of the servo.

C.1.1 Compilation

To compile the application, from the root of the hydrozoa repository run the following command:

```
$ cd testservo
$ make
```

If there is any error, please read it carefully to see what you may be missing.

C.1.2 Usage

To use the application just run it with the path to the serial port and the command you want to execute. For example to stop all the servos on a device connected through a serial-usb converter you would generally use:

```
$ ./testservo /dev/ttyUSB0 stop
```

When testservo is run with invalid parameters it will display the usage. You can also make it display by calling it with no parameters like:

```
$ ./testservo
```

C.1.3 Setting the ID

The main functionality of the testservo application is to set the ID of a servo. Some notes on setting the id:

1. Only one servo may be connected when setting the ID.
2. The valid ids for servos go from 0-127 inclusive. They are in decimal and not hexadecimal.
3. If there are any errors, make sure to run it again until it succeeds or the servo will not work anymore.

An example usage of the setid command is:

```
$ ./testservo /dev/ttyUSB0 setid 0
```

C.2 testfirmware

The testfirmware is an standalone application designed for testing all the API of the Hydrozoa firmware. It is capable of working with any number of servos but it is recommended to at least use two servos, one on each bus. The servos should also have consecutive IDs.

C.2.1 Compilation

To compile the application, from the root of the hydrozoa repository run the following command:

```
$ cd testfirmware
$ make
```

If there is any error, please read it carefully to see what you may be missing.

C.2.2 Usage

To use the application just run it from the console as such:

```
$ ./testfirmware
```

It should then proceed to display information of what it's doing while it moves the servos. It should also autodetect all the connected servos so it's a good way of seeing what servos are connected.

C.3 calcspeed

WARNING! THIS APPLICATION MAY BURN OUT YOUR SERVO, USE WITH CAUTION.

The calcspeed application is for calibrating the servo speed by creating a look up table that allows the Hydrozoa firmware to properly synchronize groups by relative the HMI speed command value with the servo real speed.

C.3.1 Compilation

See C.2.1.

C.3.2 Usage

The procedure to run calcspeed is simple. Just make sure you stop it before it burns the servo out, usually once it reaches around 200 speed. To do this type control+C and the running command should stop. To run it and generate the speed data use the following command:

```
$ ./calcspeed # Remember to kill it before it kills the servo
```

Now you should open the newly generated servo_tune.csv with your statistic program of choice and generate functions that approximate the results. To then generate the look up table you should then edit create_lookup.py and change the function "def f(x):" so that it returns the values of the function you found that approximates the result. The default function used is:

```
def f(x):
    if x >= 144:
        return -3e-5 * x**2 + 0.03544 * x + 245.7
    elif x < 144 and x >= 52:
        return 2e-5 * x**3 - 6.8e-3 * x**2 + 0.8463 * x + 210.5
    elif x < 52 and x >= 18:
        return 0.001 * x**3 - 0.1315 * x**2 + 6.1904 * x + 131.81
    else:
        return 0
```

Next you should run the create_lookup.py to generate the results with the following command:

```
$ ./create_lookup.py
```

You should now have a servo_tune.c file. This is the calibration file. Double check to make sure it was generated properly. If you are happy with the results, copy it over to src/servo_tune.c and recompile the Hydrozoa firmware. When you upload it, it should now be using the look up table you created.

C.4 testspeed

The testspeed application runs multiple speed commands to test the actual velocity of the servo. It is a good way to check if the lookup table for servo movement synchronization is good or if it needs to be adjusted.

C.4.1 Compilation

See C.2.1.

C.4.2 Usage

To use the application run it with:

```
$ ./testspeed
```

The console should then output the results.

C.5 testplot

The testplot application is a subset of the testfirmware application and it's only use is to generate a plot that verifies that the hydrozoa is indeed doing group synchronization. It must be used with three servos with ids of 0, 1 and 2 connected to it with free movement range.

C.5.1 Compilation

See C.2.1.

C.5.2 Usage

To generate the image run:

```
$ ./testplot
```

Hydrozoa
Version

Edgar Simo
Jordi Pegueroles

January 13, 2011

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Module Documentation

4.1 UART Library

Interrupt UART library using the built-in UART with transmit and receive circular buffers.

Defines

- #define UART_BAUD_SELECT(baudRate, xtalCpu) ((xtalCpu)/((baudRate)*16l)-1)

  UART Baudrate Expression.

- #define UART_BAUD_SELECT_DOUBLE_SPEED(baudRate, xtalCpu) (((xtalCpu)/((baudRate)*8l)-1)|0x8000)

  UART Baudrate Expression for ATmega double speed mode.

- #define UART_TX_BUFFER_SIZE 128
- #define **UART_FRAME_ERROR** 0x0800
- #define **UART_OVERRUN_ERROR** 0x0400
- #define **UART_BUFFER_OVERFLOW** 0x0200
- #define **UART_NO_DATA** 0x0100
- #define uart_puts_P(__s) uart_puts_p(PSTR(__s))

  Macro to automatically put a string constant into program memory.

- #define uart1_puts_P(__s) uart1_puts_p(PSTR(__s))

  Macro to automatically put a string constant into program memory.

Functions

- void uart_init (unsigned int baudrate)

  Initialize UART and set baudrate.

- void uart_setFunc (void(*func)(uint8_t))

  Sets function to call when byte is recieved.

- unsigned int uart_getc (void)
  Get received byte from ringbuffer.
- int uart_status (void)
  Check the status of the output buffer.
- void uart_putc (unsigned char data)
  Put byte to ringbuffer for transmitting via UART.
- void uart_puts (const char *s)
  Put string to ringbuffer for transmitting via UART.
- void uart_puts_p (const char *s)
  Put string from program memory to ringbuffer for transmitting via UART.
- void uart1_init (unsigned int baudrate)
  Initialize USART1 (only available on selected ATmegas).
- void uart1_setFunc (void(*func)(uint8_t))
  Sets the functiion to call when UART1 recieves data.
- unsigned int uart1_getc (void)
  Get received byte of USART1 from ringbuffer. (only available on selected ATmega).
- int uart1_status (void)
  Check the status of the output buffer.
- void uart1_putc (unsigned char data)
  Put byte to ringbuffer for transmitting via USART1 (only available on selected ATmega).
- void uart1_puts (const char *s)
  Put string to ringbuffer for transmitting via USART1 (only available on selected ATmega).
- void uart1_puts_p (const char *s)
  Put string from program memory to ringbuffer for transmitting via USART1 (only available on selected ATmega).

4.1.1 Detailed Description

```
#include <uart.h>
```

This library can be used to transmit and receive data through the built in UART.

An interrupt is generated when the UART has finished transmitting or receiving a byte. The interrupt handling routines use circular buffers for buffering received and transmitted data.

The UART_RX_BUFFER_SIZE and UART_TX_BUFFER_SIZE constants define the size of the circular buffers in bytes. Note that these constants must be a power of 2. You may need to adapt this constants to your target and your application by adding CDEFS += -DUART_RX_BUFFER_-SIZE=nn -DUART_RX_BUFFER_SIZE=nn to your Makefile.

Note

Based on Atmel Application Note AVR306

Author

Peter Fleury pfleury@gmx.ch http://jump.to/fleury

4.1.2 Define Documentation

4.1.2.1 #define UART_BAUD_SELECT(baudRate, xtalCpu) ((xtalCpu)/((baudRate)*16l)-1)

Parameters

xtalcpu system clock in Mhz, e.g. 4000000L for 4Mhz

baudrate baudrate in bps, e.g. 1200, 2400, 9600

4.1.2.2 #define UART_BAUD_SELECT_DOUBLE_SPEED(baudRate, xtalCpu) (((xtalCpu)/((baudRate)*8l)-1)|0x8000)

Parameters

xtalcpu system clock in Mhz, e.g. 4000000L for 4Mhz

baudrate baudrate in bps, e.g. 1200, 2400, 9600

4.1.2.3 #define UART_TX_BUFFER_SIZE 128

Size of the circular receive buffer, must be power of 2 Size of the circular transmit buffer, must be power of 2

4.1.3 Function Documentation

4.1.3.1 unsigned int uart1_getc (void)

See also

uart_getc

4.1.3.2 void uart1_init (unsigned int *baudrate*)

See also

uart_init

4.1.3.3 void uart1_putc (unsigned char *data*)

See also

uart_putc

4.1.3.4 void uart1_puts (const char * *s*)

See also

uart_puts

4.1.3.5 void uart1_puts_p (const char * *s*)

See also

uart_puts_p

4.1.3.6 unsigned int uart_getc (void)

Returns in the lower byte the received character and in the higher byte the last receive error. UART_NO_DATA is returned when no data is available.

Parameters

void

Returns

lower byte: received byte from ringbuffer
higher byte: last receive status

- **0** successfully received data from UART
- **UART_NO_DATA**
  no receive data available
- **UART_BUFFER_OVERFLOW**
  Receive ringbuffer overflow. We are not reading the receive buffer fast enough, one or more received character have been dropped
- **UART_OVERRUN_ERROR**
  Overrun condition by UART. A character already present in the UART UDR register was not read by the interrupt handler before the next character arrived, one or more received characters have been dropped.
- **UART_FRAME_ERROR**
  Framing Error by UART

4.1.3.7 void uart_init (unsigned int *baudrate*)

Parameters

baudrate Specify baudrate using macro UART_BAUD_SELECT()

Returns

none

4.1.3.8 void uart_putc (unsigned char *data*)

Parameters

data byte to be transmitted

Returns

none

4.1.3.9 void uart_puts (const char * *s*)

The string is buffered by the uart library in a circular buffer and one character at a time is transmitted to the UART using interrupts. Blocks if it can not write the whole string into the circular buffer.

Parameters

s string to be transmitted

Returns

none

4.1.3.10 void uart_puts_p (const char * *s*)

The string is buffered by the uart library in a circular buffer and one character at a time is transmitted to the UART using interrupts. Blocks if it can not write the whole string into the circular buffer.

Parameters

s program memory string to be transmitted

Returns

none

See also

uart_puts_P

4.1.3.11 void uart_setFunc (void(*)(uint8_t) *func*)

Parameters

Function to hook to byte recieve.

Returns

None

```
         : Sets the UART recieving function.
Purpose:  called when the UART has received a character
***************************************************************************/
{
   uart_recvFunc = func;
}
```

4.1.3.12 int uart_status (void)

Returns

1 if the output buffer is empty.

4.2 Software UART Library

Interrupt driven full duplex software UART library using the built-in Timer0, Timer2 and external int. 0, with transmit circular buffers.

Defines

- #define BR_19200

  Desired baudrate...choose one, comment the others.

- #define __AVR_ATmega128__

Enumerations

- enum AsynchronousStatesTX_t {
  TX_IDLE, TX_START, TX_TRANSMIT, TX_TRANSMIT_STOP_BIT1,
  TX_TRANSMIT_STOP_BIT2 }

  Type defined enumeration holding software UART's state.

- enum AsynchronousStatesRX_t { RX_IDLE, RX_RECEIVE, RX_DATA_PENDING }

  Type defined enumeration holding software UART's state.

Functions

- void uartsw_init (void)

  Initialize software UART.

- void uartsw_putc (unsigned char data)

  Put byte to ringbuffer for transmitting via software UART.

- int uartsw_status (void)
- void uartsw_setFunc (void(∗func)(uint8_t))

  Sets the recieve function callback for the sw uart.

Variables

- volatile AsynchronousStatesRX_t stateRX

  Holds the state of the UART.

- volatile unsigned char SwUartRXData

  Storage for received bits.

4.2.1 Detailed Description

```
#include <swuart.h>
```

This library can be used to transmit and receive data through a software UART.

Timer0 is set to overflow at desired baud rate to recieve a byte. Timer0 is triguered by external interrupt 0, thus RX pin must be tied to an external interrupt. Each byte is stored in a globally accessible variable, and only last received byte is available. Timer is idle while not receiveing.

Timer2 is used to transmit a byte. Timer2 is set to overflow at desired baud rate. ISR is active while there is data in the circular buffer. Timer2 is idle if there is no data to be send.

The SWUART_TX_BUFFER_SIZE constant define the size of the circular buffer in bytes. Note that this constant must be a power of 2. You may need to adapt this constant to your target and your application by adding CDEFS += -DUART_RX_BUFFER_SIZE=nn to your Makefile.

Footprint: 750 bytes of ROM 40 bytes of RAM

Note

Based on Atmel Application Note AVR304

Author

Jordi Pegueroles jpegueroles@iri.upc.edu http://80.32.95.100

4.2.2 Define Documentation

4.2.2.1 #define __AVR_ATmega128__

Size of the circular transmit buffer, must be power of 2

4.2.3 Enumeration Type Documentation

4.2.3.1 enum AsynchronousStatesRX_t

Enumerator:

RX_IDLE Idle state.

RX_RECEIVE Receiveing a byte.

RX_DATA_PENDING Data pending.

```
{
    RX_IDLE,
    RX_RECEIVE,
    RX_DATA_PENDING

}AsynchronousStatesRX_t;
```

4.2.3.2 enum AsynchronousStatesTX_t

Enumerator:

TX_IDLE Idle state.

TX_START Transmitting start bit.

TX_TRANSMIT Transmitting byte.

TX_TRANSMIT_STOP_BIT1 Transmitting stop bit.

TX_TRANSMIT_STOP_BIT2 Transmitting stop bit.

```
{
    TX_IDLE,
    TX_START,
    TX_TRANSMIT,
    TX_TRANSMIT_STOP_BIT1,
    TX_TRANSMIT_STOP_BIT2,

}AsynchronousStatesTX_t;
```

4.2.4 Function Documentation

4.2.4.1 void uartsw_init (void)

Initialize software UART.

This function will set up pins to transmit and receive on. Control of Timer0, Timer2 and External interrupt 0.

Parameters

void

Return values

void

References RX_IDLE, stateRX, stateTX, and TX_IDLE.

```
{
    SWUART_TxHead = 0;
    SWUART_TxTail = 0;

    //PORT
    TRXPORT |= ( 1 << RX_PIN );    // RX_PIN is input, tri-stated.
    TRXDDR |= ( 1 << TX_PIN );     // TX_PIN is output.
    SET_TX_PIN( );                 // Set the TX line to idle state.

    // Timer0
    DISABLE_TIMER0_INTERRUPT( );
    TCCR0    = 0x00;               // Init.
    TCCR0_P  = 0x00;               // Init.
    TCCR0   |= (1 << WGM01);         // Timer in CTC mode.
    TCCR0_P |= ( 1 << CS01 );      // Divide by 8 prescaler.

    // Timer2
    DISABLE_TIMER2_INTERRUPT( );
    OCR2     = TICKS2WAITONE;      // Count one period.
    TCCR2    = 0x00;               // Init.
    TCCR2_P  = 0x00;               // Init.
    TCCR2   |= ( 1 << WGM21 );    // Timer in CTC mode.
    TCCR2_P |= ( 1 << CS21 );      // Divide by 8 prescaler.

    //External interrupt
    EXT_ICR = 0x00;                // Init.
    EXT_ICR |= ( 1 << ISC01 );     // Interrupt sense control: falling edge.
```

```
    ENABLE_EXTERNAL0_INTERRUPT( ); // Turn external interrupt on.

    //Internal State Variable
    stateRX = RX_IDLE;
    stateTX = TX_IDLE;
}
```

4.2.4.2 void uartsw_putc (const unsigned char *c*)

Put byte to ringbuffer for transmitting via software UART.

This function sends a unsigned char on the TX_PIN using the timer2 isr.

Note

uartsw_init(void) must be called in advance.

Parameters

c unsigned char to transmit.

Return values

void

References stateTX, TX_IDLE, and TX_START.

```
{
    unsigned char tmphead;

    tmphead  = (SWUART_TxHead + 1) & SWUART_TX_BUFFER_MASK;

    while ( tmphead == SWUART_TxTail ){
       ;/* wait for free space in buffer */
    }

    SWUART_TxBuf[tmphead] = c;
    SWUART_TxHead = tmphead;

    if ( stateTX == TX_IDLE ) {
       stateTX = TX_START;
       TCCR2_P &= ~( 1 << CS01 );        // Reset prescaler counter.
       TCNT2    = 0;                     // Clear counter register.
       TCCR2_P |= ( 1 << CS01 );         // CTC mode. Start prescaler clock.
       ENABLE_TIMER2_INTERRUPT( );        // Enable interrupt
    }

}/* uart_putc */
```

4.2.4.3 int uartsw_status (void)

Checks to see if the send buffer is empty.

```
{
    if ( SWUART_TxHead == SWUART_TxTail )
       return 1;
    return 0;

}
```

Chapter 5

Data Structure Documentation

5.1 Group_s Struct Reference

Represents a group of servos.

Data Fields

- GroupState_t state
- uint8_t tol
- uint8_t size
- uint8_t servos [SERVO_MAX]
- uint8_t speed

5.1.1 Field Documentation

5.1.1.1 uint8_t Group_s::servos[SERVO_MAX]

IDs of the servos.

5.1.1.2 uint8_t Group_s::size

Size of the group.

5.1.1.3 uint8_t Group_s::speed

Speed to go at in grad/s.

5.1.1.4 GroupState_t Group_s::state

State of the group.

5.1.1.5 uint8_t Group_s::tol

Group tolerance.

The documentation for this struct was generated from the following file:

- servo.c

5.2 Servo_s Struct Reference

Represents a servo.

Data Fields

- uint8_t uart
- uint8_t speed
- uint16_t target
- uint16_t pos

5.2.1 Field Documentation

5.2.1.1 uint16_t Servo_s::pos

Servo's current position.

5.2.1.2 uint8_t Servo_s::speed

Servo's current speed.

5.2.1.3 uint16_t Servo_s::target

Servo's target position.

5.2.1.4 uint8_t Servo_s::uart

Which uart the servo is on.

The documentation for this struct was generated from the following file:

- servo.c

5.3 Uart_s Struct Reference

Represents an uart.

Data Fields

- void(∗ putc)(uint8_t c)
- int(∗ status)(void)
- int pos
- uint8_t buf [6]

5.3.1 Field Documentation

5.3.1.1 uint8_t Uart_s::buf[6]

Buffer containing recieving data.

5.3.1.2 int Uart_s::pos

Current position in the recieving buffer.

5.3.1.3 void(∗ Uart_s::putc)(uint8_t c)

Puts a character on the UART.

5.3.1.4 int(∗ Uart_s::status)(void)

Checks to see if outgoing buffer is empty, returns 1 if empty.

The documentation for this struct was generated from the following file:

- servo.c

Chapter 6

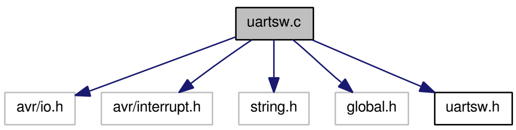
File Documentation

6.1 uartsw.c File Reference

Full duplex software and interrupt driven uart.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <string.h>
#include "global.h"
#include "uartsw.h"
```

Include dependency graph for uartsw.c:



Defines

- #define **SWUART_TX_BUFFER_MASK** (SWUART_TX_BUFFER_SIZE - 1)
- #define INTERRUPT_EXEC_CYCL 9

  Cycles to execute interrupt rutine from interrupt.

- #define **SET_TX_PIN**() (TRXPORT |= (1 << TX_PIN))
- #define **CLEAR_TX_PIN**() (TRXPORT &= ~(1 << TX_PIN))
- #define **GET_RX_PIN**() (TRXPIN & (1 << RX_PIN))

Functions

- ISR (INT0_vect)

  External interrupt 0 service routine.

- ISR (TIMER0_COMP_VECT)

  Timer0 interrupt service routine.

- ISR (TIMER2_COMP_VECT)

  Timer2 interrupt service routine.

- void uartsw_init (void)

  Function to initialize the software UART.

- void uartsw_putc (const unsigned char c)

  Send a unsigned char.

- int uartsw_status (void)
- void uartsw_setFunc (void(*func)(uint8_t))

  Sets the recieve function callback for the sw uart.

Variables

- static volatile unsigned char **SWUART_TxBuf** [SWUART_TX_BUFFER_SIZE]
- static volatile unsigned char **SWUART_TxHead**
- static volatile unsigned char **SWUART_TxTail**
- volatile AsynchronousStatesRX_t stateRX

  Holds the state of the UART.

- static volatile AsynchronousStatesTX_t stateTX

  Holds the state of the UART.

- static volatile unsigned char SwUartTXData

  Data to be transmitted.

- static volatile unsigned char SwUartTXBitCount

  TX bit counter.

- volatile unsigned char SwUartRXData

  Storage for received bits.

- static volatile unsigned char SwUartRXBitCount

  RX bit counter.

- static void(* **swuart_recvFunc**)(uint8_t ch)

6.1.1 Detailed Description

UART software implementation using Timer0, Timer2 and external interrupt 0.

Note that the RX_PIN must be the external interrupt 0 pin on your AVR of choice. The TX_PIN can be chosen to be any suitable pin.

6.1.2 Function Documentation

6.1.2.1 ISR (TIMER2_COMP_VECT)

Timer2 will ensure that bits are written and read at the correct instants in time. The state variable will ensure context switching between transmit and recieve. If state should be something else, the variable is set to IDLE. IDLE is regarded as a safe state/mode.

Note

uartsw_init(void) must be called in advance.

References stateTX, SwUartTXBitCount, SwUartTXData, TX_IDLE, TX_START, TX_TRANSMIT, TX_TRANSMIT_STOP_BIT1, and TX_TRANSMIT_STOP_BIT2.

```
{

  unsigned char tmptail;

  switch (stateTX) {

     // Check if there's a byte waiting in the buffer.
     case TX_START:
        if ( SWUART_TxHead != SWUART_TxTail) {

           /* calculate and store new buffer index */
           tmptail = (SWUART_TxTail + 1) & SWUART_TX_BUFFER_MASK;
           SWUART_TxTail = tmptail;
           /* get one byte from buffer and write it to UART */
           SwUartTXData = SWUART_TxBuf[tmptail];

           stateTX = TX_TRANSMIT;  /* start transmission */

           SwUartTXBitCount = 0;

           CLEAR_TX_PIN();                  // Clear TX line...start of preamble.

        }else{
           /* tx buffer empty, disable Timer2 interrupt */
           DISABLE_TIMER2_INTERRUPT( );     // Stop the timer interrupts.
           TCCR2_P &= ~( 1 << CS01 );       // Stop timer2
           stateTX = TX_IDLE;               // Error, should not occur. Going to a safe state.
        }
        break;

        // Transmit Byte.
     case TX_TRANSMIT:
        // Output the TX buffer.
        if( SwUartTXBitCount < 8 ) {
           if( SwUartTXData & 0x01 ) {           // If the LSB of the TX buffer is 1:
              SET_TX_PIN();                      // Send a logic 1 on the TX_PIN.
           }
           else {                                // Otherwise:
              CLEAR_TX_PIN();                    // Send a logic 0 on the TX_PIN.
           }
           SwUartTXData = SwUartTXData >> 1;     // Bitshift the TX buffer and
           SwUartTXBitCount++;                   // increment TX bit counter.
        }

```

```
         //Send stop bit.
         else {
            SET_TX_PIN();                         // Output a logic 1.
            stateTX = TX_TRANSMIT_STOP_BIT1;
         }
         break;

         // Go to idle after stop bit was sent.
      case TX_TRANSMIT_STOP_BIT1:
         stateTX = TX_TRANSMIT_STOP_BIT2;
         break;

      case TX_TRANSMIT_STOP_BIT2:
         stateTX = TX_START;
         break;

         // Unknown state.
      default:
         stateTX = TX_IDLE;                      // Error, should not occur. Going to a safe state.
   }
}
```

6.1.2.2 ISR (TIMER0_COMP_VECT)

Timer0 will ensure that bits are written and read at the correct instants in time. The state variable will ensure context switching between transmit and recieve. If state should be something else, the variable is set to IDLE. IDLE is regarded as a safe state/mode.

Note

uartsw_init(void) must be called in advance.

References RX_DATA_PENDING, stateRX, SwUartRXBitCount, and SwUartRXData.

```
{

   //Receive Byte.
   OCR0 = TICKS2WAITONE;                  // Count one period after the falling edge is trigged.
   //Receiving, LSB first.
   if( SwUartRXBitCount < 8 ) {
      SwUartRXBitCount++;
      SwUartRXData = (SwUartRXData>>1);   // Shift due to receiving LSB first.
      if( GET_RX_PIN( ) != 0 ) {
         SwUartRXData |= 0x80;           // If a logical 1 is read, let the data mirror this.
      }
   }

   //Done receiving
   else {
      stateRX = RX_DATA_PENDING;          // Enter DATA_PENDING when one byte is received.
      DISABLE_TIMER0_INTERRUPT( );        // Disable this interrupt.
      TCCR0_P &= ~( 1 << CS01 );          // Stop timer0
      EXT_IFR |= (1 << INTF0 );           // Reset flag not to enter the ISR one extra time.
      ENABLE_EXTERNAL0_INTERRUPT( );      // Enable interrupt to receive more bytes.

      /* Run the recieved byte function if applicable. */
```

```
        if (swuart_recvFunc != NULL)
           swuart_recvFunc( SwUartRXData );
     }
 }
```

6.1.2.3 ISR (INT0_vect)

The falling edge in the beginning of the start bit will trig this interrupt. The state will be changed to RX_RECEIVE, and the timer interrupt will be set to trig one and a half bit period from the falling edge. At that instant the code should sample the first data bit.

Note

uartsw_init(void) must be called in advance.

References INTERRUPT_EXEC_CYCL, RX_RECEIVE, stateRX, and SwUartRXBitCount.

```
{

   stateRX = RX_RECEIVE;                 // Change state
   DISABLE_EXTERNAL0_INTERRUPT( );       // Disable interrupt during the data bits.


   DISABLE_TIMER0_INTERRUPT( );          // Disable timer to change its registers.
   TCCR0_P &= ~( 1 << CS01 );            // Reset prescaler counter.

   TCNT0 = INTERRUPT_EXEC_CYCL;          // Clear counter register. Include time to
   run interrupt rutine.

   TCCR0_P |= ( 1 << CS01 );             // Start prescaler clock.

   OCR0 = TICKS2WAITONE_HALF;             // Count one and a half period into the f
  uture.

   SwUartRXBitCount = 0;                 // Clear received bit counter.
   CLEAR_TIMER0_INTERRUPT( );            // Clear interrupt bits
   ENABLE_TIMER0_INTERRUPT( );           // Enable timer0 interrupt on again

}
```

Index

Acknowledgements

Thanks to the Institut de Robòtica i Informàtica Industrial [4] and Humanoid Lab Team [10] for making this all possible. Specials thanks to Guillem Alenyà, Sergi Hernández and José Luis Rivero as mentors of the project. Finally a big thanks to the Generalitat of Catalunya [2] for financing the project as part of it's Beques de Programari Lliure project [5].

IRI reports

This report is in the series of IRI technical reports.
All IRI technical reports are available for download at the IRI website
`http://www.iri.upc.edu`.