

# **Senior Design**

CS400

**Design Report**

SubSim

I/O Group

**Submitted to:**

**Submitted by:**

**Submitted on:**

Steven L. Barnicki

Steve Nolte

Brian Rittner

Scott Wadell

Eric Wurtz

2/19/2003

# Table of Contents

Senior Design.....	1
Table of Contents.....	2
Overview of project.....	5
Major System Components and their Interrelationships.....	6
Engine Simulator.....	6
Master Control.....	6
Microcontroller001.....	6
Microcontroller002.....	6
Hardware Documentation.....	7
Circuit Schematics.....	7
Entire Network Overview.....	7
MicroController002.....	7
Shaft Encoder.....	7
Stepper Motor.....	11
Torpedo Firing Controls.....	12
Parts List.....	13
Design Steps Taken for Easy Maintenance of the Hardware.....	15
Software Documentation.....	16
Master Control.....	16
XML/RPC.....	16
Serial interface.....	18
Serial Interface(Controller 001).....	18
Serial Interface(Controller 002).....	20
User interface Specifications.....	22
Steps Taken for Easy Maintenance.....	22
ABET Concerns.....	23
Economic.....	23
Environmental.....	23
Sustainability.....	23
Manufacturability.....	23
Ethical.....	23
Health and Safety.....	23
Social.....	24
Project Management information.....	25
Time spent by Eric Wurtz on the design phase of project.....	25
Time spent by Brian Rittner on the design phase of project.....	33
Time spent by Scott Wadell on the design phase of project.....	33
Time spent by Steve Nolte on the design phase of project.....	33
Project Schedule.....	34
Miscellaneous Design Documentation.....	35
Circuit Layout.....	35
Channel Capabilities.....	35
Pin Usage.....	36

Design documentation.....	38
Arrays .....	38
Checksum .....	41
Copy.....	41
Non-volatile Copy.....	41
Element.....	41
Length .....	42
Pointer.....	42
Print.....	43
ASYNCHRONOUS SERIAL .....	43
Summary of messages .....	43
Creation .....	44
InputBuffer .....	46
Look .....	46
Put.....	47
Queue .....	47
Timeout.....	47
Valid .....	48
Accepting Print.....	48
DIGITAL.....	48
Creation .....	49
Asserted.....	50
High.....	50
Low .....	50
NotAsserted .....	51
Off .....	51
On .....	51
Output .....	52
Pulse .....	52
Toggle .....	52
Value.....	52
Printing.....	53
FILE.....	53
Put.....	55
Get .....	56
Queue .....	56
Reset.....	56
Empty.....	56
Name .....	56
Length .....	57
Readpoint.....	57
Close.....	57
Element.....	57
Find .....	57
Lock .....	58
Unlock .....	58

Help.....	58
PRINT TO .....	58
PRINT .....	58
OPERATING SYSTEM .....	59
Checksum .....	59
Copy.....	60
Debug .....	61
ErrorAction .....	61
Free.....	61
Output .....	62
Protect.....	62
Reset.....	63
Run .....	63
RunMode .....	63
UserSwitch .....	63
PRINT .....	64
PULSE COUNTER.....	64
Reset.....	66
Printing.....	66
PULSE WIDTH IN .....	66
Creation .....	66
Go .....	67
Done .....	67
Period.....	67
PRINT .....	68
PULSE WIDTH OUT .....	68
Creation .....	69
Asserted.....	70
Count .....	70
NotAsserted .....	70
Format.....	71
On, Go .....	72
Off .....	72
Period.....	72
Queue .....	72
Width.....	72
Printing.....	73
Die.....	73
SERIALIO.....	73
Creation .....	74
On .....	74
Put.....	74
Off .....	74
SHAFT .....	75
Creation .....	75

## Overview of project

The I/O group is a subset of a larger team that is developing a Cobia SS-245 submarine simulator for the Wisconsin Maritime Museum located in Manitowoc, WI. Our task is to design and implement the I/O for all of the peripheral devices of the submarine simulator. This is inclusive of the analog dials and gauges that are being simulated from the conning tower of the submarine.

The specifics of our group's responsibilities are based around a PC which will be called the Master Control. This controller will be interfaced to another PC, the control for the main simulator engine, via Ethernet and communicate by way of XML RPC. The Master Control will control each the device controllers, which in turn control the simulator's peripheral devices. The bus that the controllers are on, along with the Master Control's interface to the bus, are under the responsibilities of the I/O group as well.

The I/O team's work will focus on the I/O responsibilities for the following:

- Reading the rudder wheel position which will be taken using a bidirectional shaft encoder.
- Displaying the torpedo firing boxes which will be controlled using a microcontroller to light the LED's and read the switch positions.
- The team is also responsible for the submarine gauges that display the compass position, rudder motor speed, rudder angle, and submarine speed. These gauges will consist of a stepper motor for the compass and servos for the others.

# **Major System Components and their Interrelationships**

## ***Engine Simulator***

The engine simulator will be the computer that performs the actual simulation. It will be the center of all data communications between all the computers.

## ***Master Control***

The Master Control bridges the I/O microcontrollers to the Engine Simulator. The Master Control is linked to the Engine Simulator via an XML/RPC library which will control state variables including: rudder wheel position, left and right motor speed, gyrocompass position, torpedo status indicators, and the torpedo firing controls. While the Master Control will be running windows 98SE, the shell will not be set to the explorer.

The Master Control uses a RS232 serial interface to connect to the microcontrollers.

## ***Microcontroller001***

Microcontroller 001 will be VM-1 venom which will communicate through a serial interface using terminal software. The venom runs a proprietary high level language along with a real-time operating system. This allows the venom to respond to instructions received from the Master Control to perform its tasks. These tasks include: updating the rudder position gauge which is controlled by a servo using pulse width modulation, updating the left and right motor speed indicators which are also driven by servos, updating the gyrocompass which is controlled by a stepper motor, and taking readings from the shaft encoder to determine the position of the rudder wheel.

## ***Microcontroller002***

Microcontroller 002 will be a VM-1 board using the same language and OS as above. This board will again receive instructions through a serial terminal. The components of the system that will be controlled by this board include: input from two torpedo firing buttons, torpedo status lights and the torpedo ready switches.

## **Hardware Documentation**

The following documentation describes the hardware and its interconnectivity. This is accomplished by use of detailed overviews, control panel layouts and circuit schematics. A descriptive parts list will also be provided for further clarification. Relevant design steps that allow easy hardware maintenance will also be provided.

### ***Circuit Schematics***

This section contains schematics for the circuits used in this project.

### **Entire Network Overview**

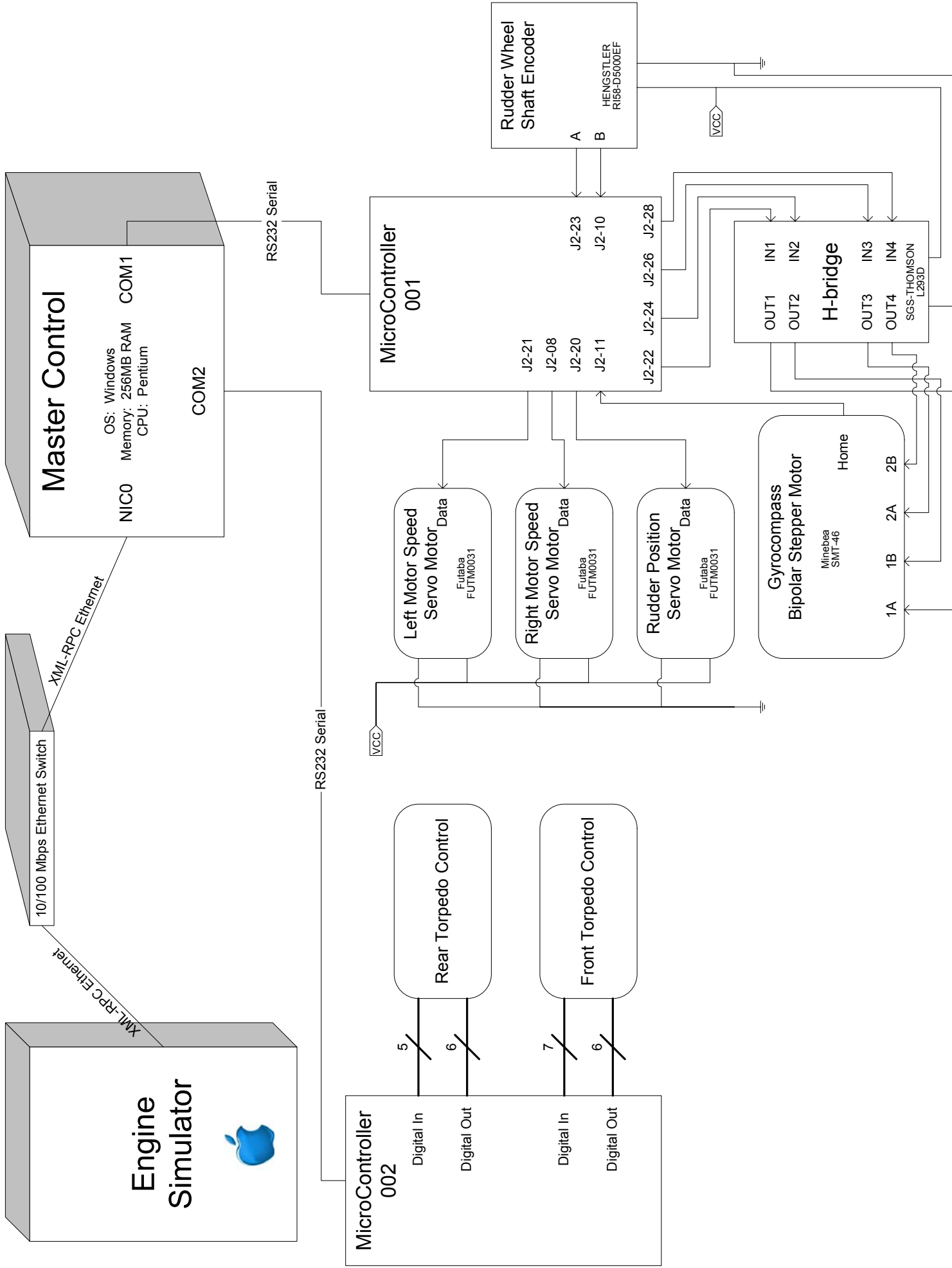
The next page has a schematic of the entire submarine simulator as related to the I/O.

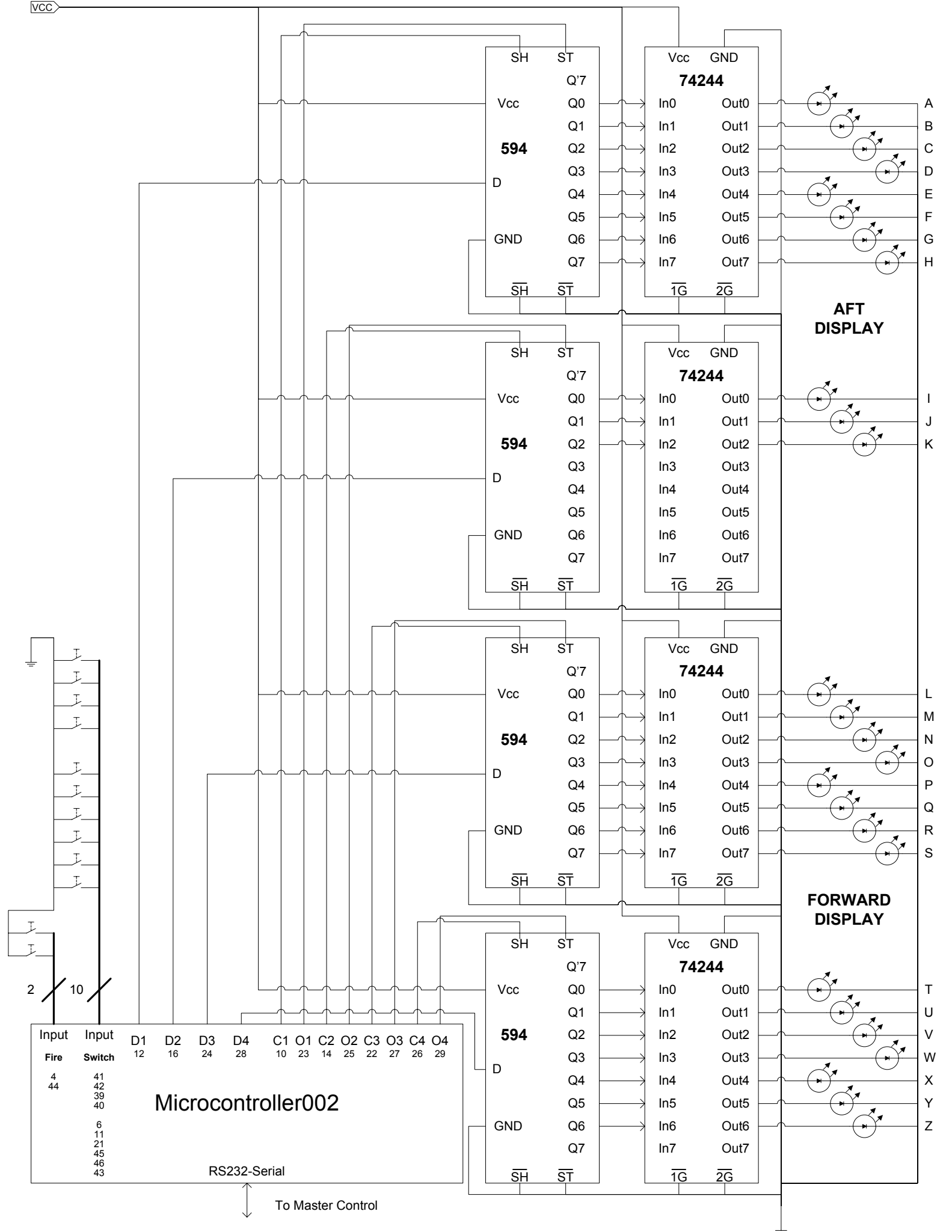
### **MicroController002**

Following the overview page is a more detailed schematic of MicroController002 and its peripherals.

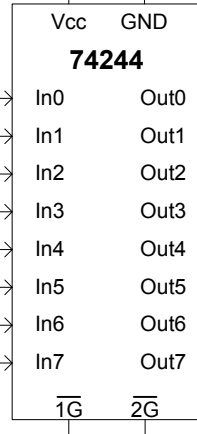
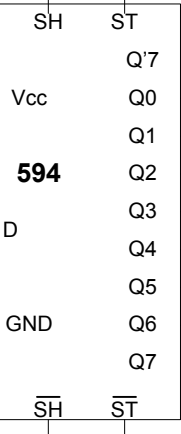
### **Shaft Encoder**

Then a description of the shaft encoder for the rudder will follow.

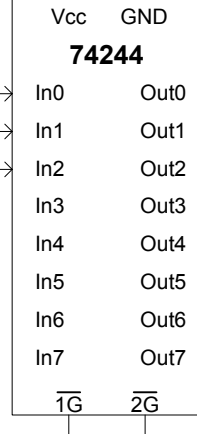
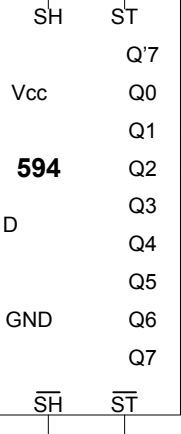




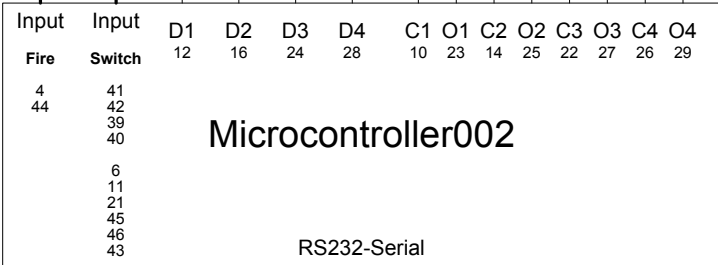
VCC



**AFT DISPLAY**

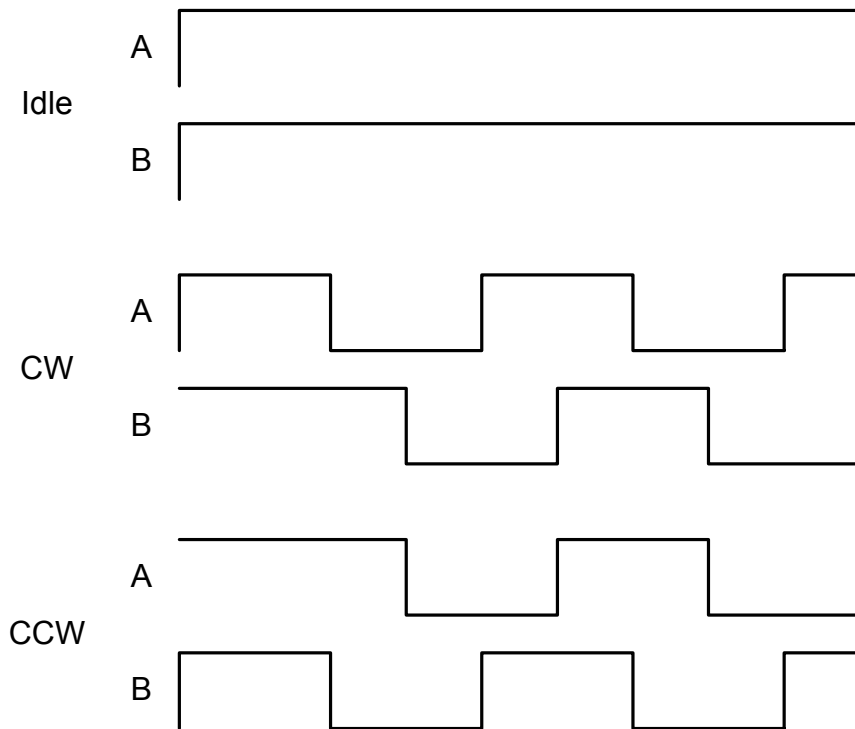
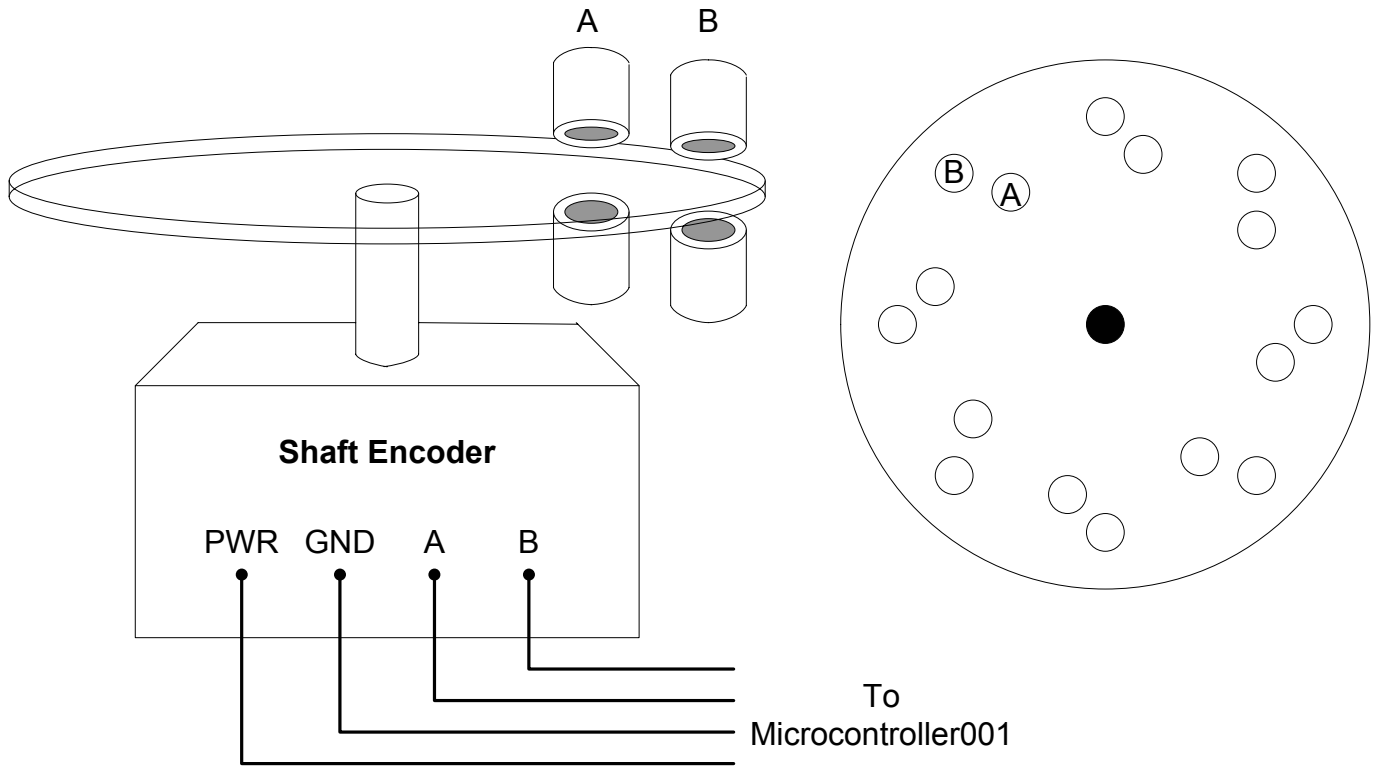


**FORWARD DISPLAY**



To Master Control

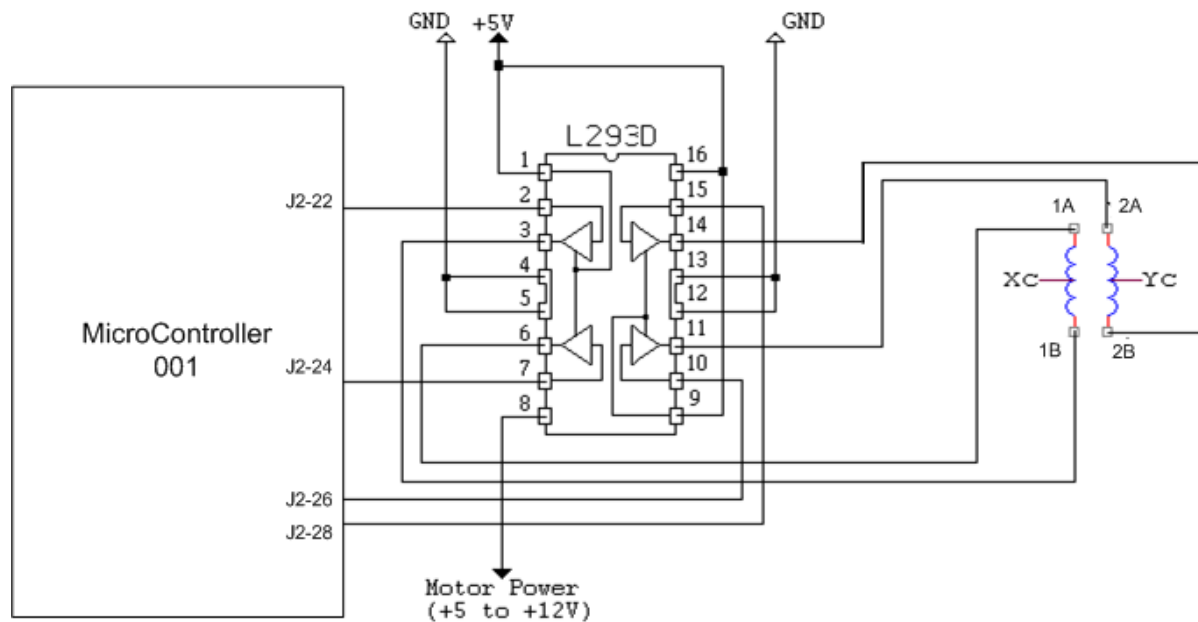
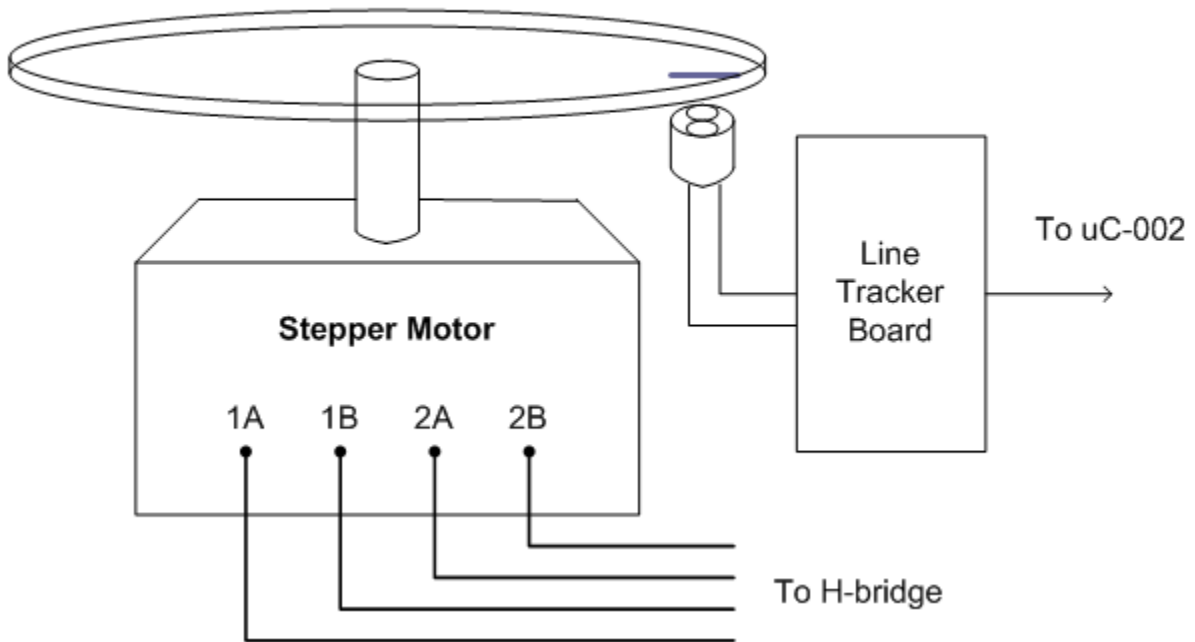
# Rudder Wheel Shaft Encoder



## Stepper Motor

The following two diagrams show the setup up for the gyrocompass. It will be controlled by MicroController002 that commands a stepper motor via an H-bridge.

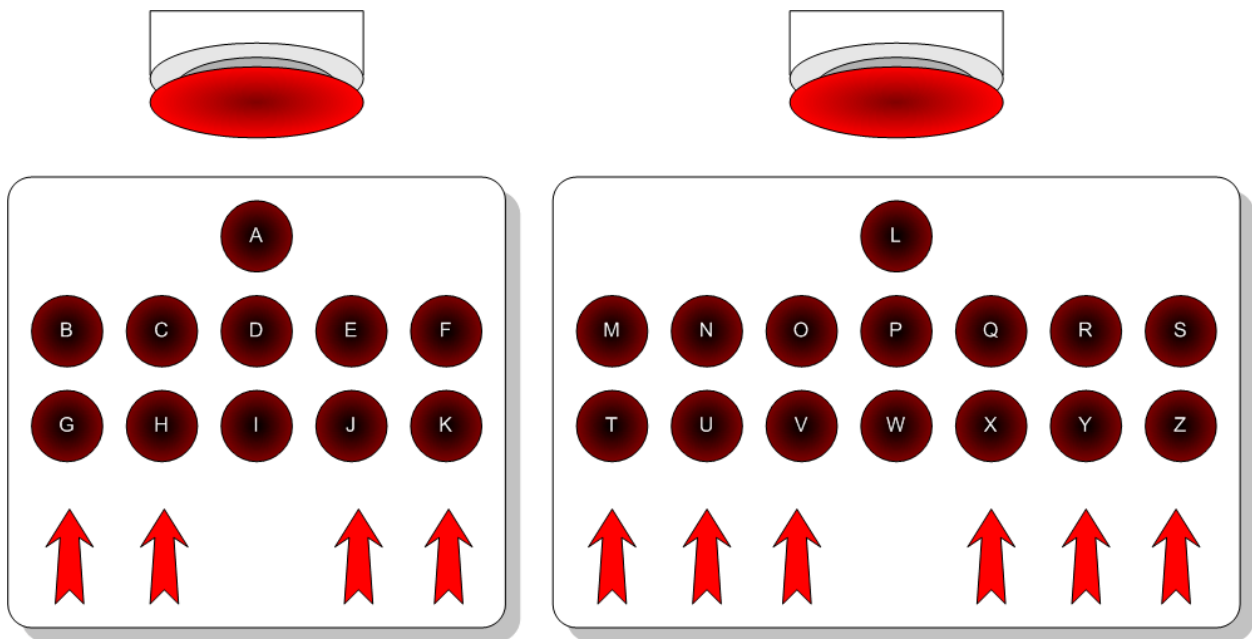
### Gyrocompass



## Torpedo Firing Controls

The following is a schematic of the original submarines fire controls along with a narrative of its workings that will be emulated by the simulator.

- 1) Load the tube with torpedo.
- 2) In torpedo room, the *MANUAL READY* switch will get set when torpedo is loaded.
- 3) The bottom ready light is illuminated in the conning tower for the corresponding tube indicating that a torpedo has been loaded.
- 4) A torpedo tube will be selected for calibration from the conning tower by setting the *STAND BY* switch.
- 5) The spindle calibrates the corresponding torpedo and the GYRO SPINDLE "In" light is illuminated.
- 6) The GYRO SPINDLE "In" light goes off and the ANGLE SET light and middle READY light illuminate.
- 7) Upon successful calibration, the corresponding top light for torpedo illuminates in conning tower.
- 8) Pressing fire button turns on buzzer and FIRE light, followed by a torpedo launch.



Aft Torpedo Status  
Output: 11  
Input: 5

Forward Torpedo Status  
Output: 15  
Input: 7

## Parts List

The following is a list of the parts that we will need to implement the given project.

Part	Description	Manufacturer	Part Number	Price [£]	Price	#	Distributor
VM-1 control computer	The VM-1 is the latest credit-card-sized control computer from Micro-Robotics.	Micro-Robotics Ltd.	5800	£72.50	\$118.23	2	Micro-Robotics Ltd.
VM-1 application board	The Application Boards provide plug-in access to RS232 and RS485 serial ports; various analogue, digital and pulse IO; keyboard and display interfaces, including both graphic and alphanumeric displays; two I2C busses (long I2C option); Microwire®/SPI® and an EEPROM. They require an unregulated DC supply.	Micro-Robotics Ltd.	5802	£90.00	\$146.76	2	Micro-Robotics Ltd.
VM-1 Breakout board	Doubles as a Flash programmer for duplicating your finished application, or for updating your copy of Venom-SC.	Micro-Robotics Ltd.	5805	£34.08	\$55.57	1	Micro-Robotics Ltd.
VM-1 Flash ROMS	The Venom-SC language is burned into a flash device, which is plugged into the VM-1 controller. This flash device also doubles as the application storage area. When you have finalized your application, your code is burned into the flash so it is secure from alteration. Of course you can re-program the flash, either to change your application, or to update your version of Venom-SC.	Micro-Robotics Ltd.	5524	£15.00	\$24.46	2	Micro-Robotics Ltd.
Futaba Servo Motor	S3003 Standard; Torq/Spd 4.8V: 44/.23; Torq/Spd 6.0V: 57/.19; Bearing: Top; Size: 1.6X.8X1.4; Weight: 1.30	Futaba	FUTM0031			3	Nathan Schlehlein

Stepper Motor	STEPPER MOTOR, 400 STEPS/REVOLUTION Minebea "Astrosyn" Type 17PY-Q202-03. Small precision bipolar stepper motor. 0.9 degrees/ step. 400 steps / revolution. 12.5 ohm coils. 1.66" x 1.66" x 0.92" body. 0.2" dia. x 0.4" long shaft. Four 2.25" leads w/ four pin socket connector, 0.1" spacing.	Minebea	SMT-46	\$6.75	1	ALL ELECTRONICS CORP.
LEDs		Unknown	L1-0-W5TH70- 1	\$1.95/10	26	LED Supply
Hbridge chip for stepper	PUSH-PULL FOUR CHANNEL DRIVER WITH DIODES	SGS-THOMSON	L293D	\$2.70	1	Digi-Key
Shaft Encoder	12mm Hollow Shaft 10...30VDC=. Has 4 position connector.	HENGSTLER	RI58- D5000EF	\$25.00	1	Independent
Master Control Driver chips for LED's	Octal Tri-State Buffer	Philips	74244	\$2.44	4	Philips
Shift Register	8-Bit shift register	Fairchild Semicinductor	MM74HCT164	\$0.25	4	Fairchild Semicinductor

## ***Design Steps Taken for Easy Maintenance of the Hardware***

There were many design steps taken to ensure easy maintenance of the hardware used on this project. The steps are as follows:

- The CPU used on the main board is under-clocked so that no fan is needed. This is done so that if there is a problem with the fan the CPU will not be affected, such as the problem of overheating.
- The Master Control will net boot to eliminate the need for a hard drive. This will add further stability to the Master Control and eliminate failure due to hard drive complications.
- The Master Control will receive its image from an image server so that if the Master Control image needs to be changed or updated, it only has to be updated on the image server. This allows for the image to be changed in only one place instead of having to change it on the Master Control.
- The microcontroller boards will use flash ROMs. so that if a software update is required to take place all that is needed is to replace or flash the ROMs. This will eliminate the task of having the Master Control load the program to the microcontroller at every boot up. This will also ensure that the program is always in memory at boot time and is ready to run.
- The torpedo firing display will use LEDs instead of tungsten bulbs. This will lessen the probability that a bulb may burn out and have to be changed.

# Software Documentation

The following is a description of the major functions that will be written to drive the hardware. A brief statement with regards to the user interface will also be discussed.

## ***Master Control***

### **XML/RPC**

#### ***Description and purpose***

This function will set the rudder position on the engine simulator via XML/RPC.

#### ***Function and signature***

Void SetRdrPos(double Position)

#### ***Pre-conditions***

None

#### ***Post-conditions***

The rudder position will be set to the new value defined by *Position*.

#### ***Exception conditions***

None

#### ***Special conditions***

None

#### ***Description and purpose***

This function will set the correct fire tube(s) on the engine simulator via XML/RPC.

These values will be Boolean. A “True” value will indicate an active tube while a “False” value will indicate an inactive tube.

#### ***Function and signature***

Void SetTorpFire(bool [10])

#### ***Pre-conditions***

None

#### ***Post-conditions***

The correct tubes will be set according to the Boolean array “bool [10]”.

#### ***Exception conditions***

None

#### ***Special conditions***

None

#### ***Description and purpose***

This function will get the correct left motor speed from the engine simulator via the XML/RPC. This value may then be passed to MicroController 001 to adjust the Left Motor Speed Servo Motor.

#### ***Function and signature***

double GetLSpeed()

#### ***Pre-conditions***

None

**Post-conditions**

The correct left speed value will be available to pass to MicroController 001.

**Exception conditions**

None

**Special conditions**

None

**Description and purpose**

This function will get the correct right motor speed from the engine simulator via the XML/RPC. This value may then be passed to MicroController 001 to adjust the Right Motor Speed Servo Motor.

**Function and signature**

double GetRSpeed()

**Pre-conditions**

None

**Post-conditions**

The correct right speed value will be available to pass to MicroController 001.

**Exception conditions**

None

**Special conditions**

None

**Description and purpose**

This function will get the correct position for the gyrocompass via the XML/RPC. This value may then be passed to MicroController 001 to adjust the Gyrocompass Stepper Motor.

**Function and signature**

double GetCompPos()

**Pre-conditions**

None

**Post-conditions**

The correct position value will be passed to the Gyrocompass Stepper Motor.

**Exception conditions**

None

**Special conditions**

None

## **Serial interface**

### **Serial Interface(Controller 001)**

#### ***Description and purpose***

This function will send the correct value (speed) to the right servo motor. The function will return a Boolean value; True is a successful set and False if otherwise.

#### ***Function and signature***

Bool SetRSpeed(double Speed)

#### ***Pre-conditions***

None

#### ***Post-conditions***

The correct speed value will be passed to the right servo motor. A Boolean value will be returned; True is successful and false if otherwise.

#### ***Exception conditions***

None

#### ***Special conditions***

None

#### ***Description and purpose***

This function will send the correct value (speed) to the left servo motor. The function will return a Boolean value; True is a successful set and False if otherwise.

#### ***Function and signature***

Bool SetLSpeed(double Speed)

#### ***Pre-conditions***

None

#### ***Post-conditions***

The correct speed value will be passed to the left servo motor. A Boolean value will be returned; True is successful and false if otherwise.

#### ***Exception conditions***

None

#### ***Special conditions***

None

#### ***Description and purpose***

This function will send the correct value (position) to the Gyrocompass stepper motor. The function will return a Boolean value; True is a successful set and False if otherwise.

#### ***Function and signature***

Bool SetComp(double Position)

#### ***Pre-conditions***

None

#### ***Post-conditions***

The correct position value will be passed to the Gyrocompass stepper motor. A Boolean value will be returned; True is successful and false if otherwise.

#### ***Exception conditions***

None

***Special conditions***

None

***Description and purpose***

This function will send the correct rudder value (position) to the Rudder Position Servo Motor. The function will return a Boolean value; True is a successful set and False if otherwise

***Function and signature***

double SetRdrPos()

***Pre-conditions***

None

***Post-conditions***

The correct position value will be passed to the Rudder Position Servo Motor. A Boolean value will be returned; True is successful and false if otherwise.

***Exception conditions***

None

***Special conditions***

None

## Serial Interface(Controller 002)

### **Description and purpose**

This function will get the selected tube(s) from the front and rear Torpedo Display. The function will return a vector containing Boolean values; True if tube is set and False if otherwise.

### **Function and signature**

Bool [10] GetTubeSel()

### **Pre-conditions**

None

### **Post-conditions**

The correct Boolean vector will be returned denoting the selected tube(s).

### **Exception conditions**

None

### **Special conditions**

None

### **Description and purpose**

This function will set the indicators on the Front Torpedo Display. A Boolean value will be returned; True is successful and false if otherwise.

**\*Note:** For front and rear displays vector positions[0-2] are reserved for the feedback indicators.

position[0] – Ready light  
position[1] – Gyro spindle “in”  
position[2] – Angle set  
postion[3-14] – Status lights

### **Function and signature**

Bool SetFTorpDisp(bool Front [15])

### **Pre-conditions**

None

### **Post-conditions**

The Front Torpedo Display will be set as indicated by the Boolean vector Front.

### **Exception conditions**

None

### **Special conditions**

None

### **Description and purpose**

This function will set the indicators on the Rear Torpedo Display. A Boolean value will be returned; True is successful and false if otherwise.

**\*Note:** For front and rear displays vector positions[0-2] are reserved for the feedback indicators.

position[0] – Ready light  
position[1] – Gyro spindle “in”  
position[2] – Angle set  
postion[3-10] – Status lights

***Function and signature***

Bool SetRTorpDisp(bool Rear [11])

***Pre-conditions***

None

***Post-conditions***

The Rear Torpedo Display will be set as indicated by the Boolean vector Rear.

***Exception conditions***

None

***Special conditions***

None

**\*Note:** For front and rear displays vector positions[0-2] are reserved for the feedback indicators.

### ***User interface Specifications***

There will be no user interface in the final implementation. However, we will be using the standard console for simulation and debugging purposes. Prior to completion of the project, this feature will be disabled.

### ***Steps Taken for Easy Maintenance***

The steps that were taken to ensure easy maintainability on the software side of the project are as follows.

The interfaces between the Master Control and the microcontrollers are object oriented so that objects may be added if need be. Furthermore the Master Control is interfaced with the other parts of the simulator using XML/RPC. This provides for easy expansion of the simulator if another aspect of the submarine is to be simulated in the future.

## **ABET Concerns**

The following are the issues that we will identify in order to meet ABET criterion 4.

### ***Economic***

The Wisconsin Maritime Museum has generously offered to fund this project. Therefore, while there may not be a strict constraint on monetary issues, it will be our goal to make the best use of the equipment and devices that are purchased. In addition, we will require the time of other individuals in helping meet our goals. Hence we need to be clear in our requests so that time is not wasted.

### ***Environmental***

This project will not produce any waste when it is completed. In construction of the project, consideration should be put into maximizing the use of materials so that little is left unused.

### ***Sustainability***

Thorough and clear documentation needs to be made so that future enhancements and add-ons to the submarine simulator may be made. The maintenance of the simulator after we finish our product will not be done by our team, which makes it critical that others be able to understand what we did and how it works if problems arise.

### ***Manufacturability***

In constructing our project the technology, both hardware and software, will need to be researched thoroughly so that conflicts do not occur. Hardware constraints are especially important to consider because they will need to interface with the parts of the simulator that other teams are building.

### ***Ethical***

One of the goals for any simulator is to make it as close to the original as possible. Therefore, staying true to the way things worked in the time the real submarine was in commission is something to strive for. Most visitors to the museum that will run the simulator have never been in a real submarine, and so our simulator should not give them the wrong impressions.

### ***Health and Safety***

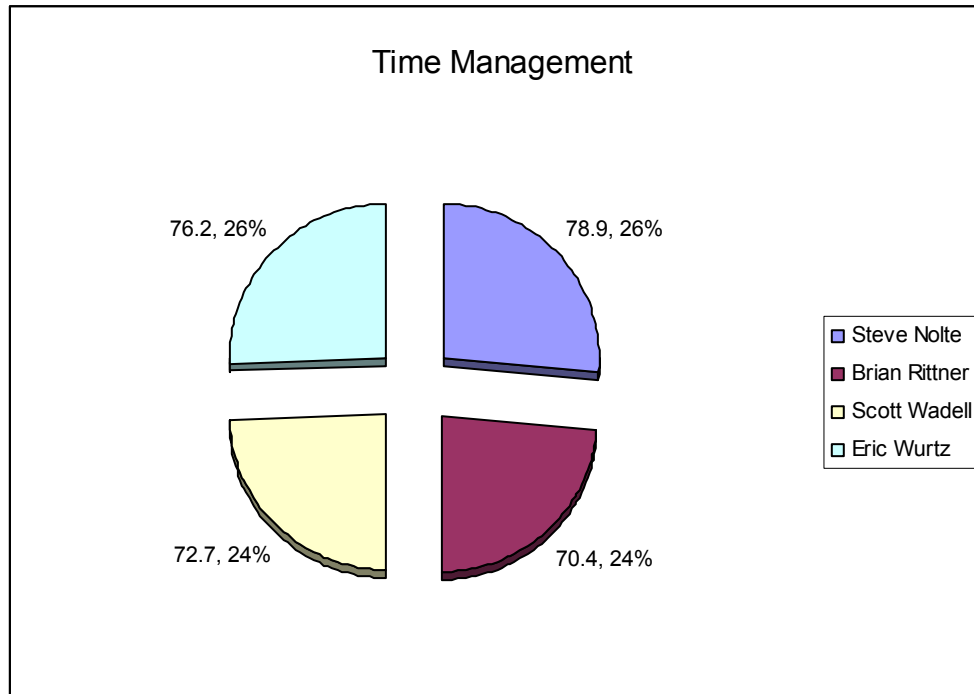
Safety is a very important aspect because this will be a commercial product for public visitors to the museum to use. As our part of the project is designed and implemented, concern for the safety of others that will use the simulator is important to keep in mind. This project will make use of electrical power that carries potential harm to humans. The project also needs to be safely accessible by the handicapped.

## ***Social***

Everyone in our group lives off-campus, which will be a challenge to find times to meet together. Also, there will be other people such as the curator of the museum and other individuals of different ages and backgrounds that will be helping with this product. The project as a whole includes a rather diverse group of people.

## Project Management information

The following is a summary of the minutes spent by each group member on the design portion of the project:



### ***Time spent by Eric Wurtz on the design phase of project***

**12-06-2002-11:00 to 12:00**

Met for an hour with the entire sub team to get an overview of what we are actually simulating. Also I discussed with Nathan some specifics of the I/O sub Sub Team's part. The I/O consists of an I/O computer connected to the "main engine computer" by way of either net. They will communicate using XML RPC. The I/O computer will have to control a large number of microcontrollers that control a large number of dials, switches and lights, along with interactive spotlighting to direct the users attention to the specific gages during the simulating.

Among the items to be designed include: The interface between the I/O computer and the microcontrollers such as the media and the protocol; The software that relays the XML RPC to the controller bus; what to use to control the spot lights.

-60

**12-13-2002-12:00 to 14:00**

I was given a tour by Nathan of the WWII submarine that we will be simulating. We looked at the actual location that the simulator will be built in. We also looked at some of the gages that we taken from other submarines that we could use in the simulator.  
-120

**01-03-2003-10:36 to 14:16**

Our group began the number 1 requirement of CS400, the proposal. Our group completed the proposal as well.  
-220

**01-03-2003-14:22 to 15:56**

We started the number 2 requirement of CS400, the project initiation and preliminary schedule.

The positions were filled as follows

Steve Nolte	The Design Manager
Brian Rittner	The Technology Manager
Scott Wadell	Documentation Manager
Eric Wurtz	Project Manager

Brian Rittner has knowledge of the I<sup>2</sup>C bus which might be implemented so he was made technology manager. The rest of the jobs were filled arbitrarily.

Milestones and their deadlines were set to give sufficient time for the work to be done before the respective report was required.

We completed the number 2 requirement, the project initiation and preliminary schedule.  
-94

**01-06-2003-13:00 to 13:56**

Started work on criteria for technology research. Scheduled meeting times  
-56

**01-08-2003-11:00 to 16:00**

Researched chips with the I<sup>2</sup>C BUS implemented in hardware along with specs on the I<sup>2</sup>C bus.

Initial research on the I<sup>2</sup>C shows that it will probably work for our controller bus. We will use a USB to I<sup>2</sup>C interface chip to connect the Master Control to the I<sup>2</sup>C bus.

A candidate chip was selected for the USB to I<sup>2</sup>C interface.

We need to find out more of the specifics of the helmsmen's display, particularly the odometer and the rudder indicator.

On the software side, the library of calls for the I<sup>2</sup>C interface on the PC is written in C while the XML RPC library that will be provided for us might be in JAVA. We will have to find a way for the Master Control master software, probably written in C++, to be able to interact with JAVA software.

-240

### **01-09-2003-13:15 to 18:38**

Steve, Rittner, and I discussed the entire sub simulator layout with Schlehlein. All the PCs used in the simulator should use a boot ROM to boot off an image server. This is so that there is no drive failure within a PC that is built in to part of the simulator requiring it to get taken apart. This also requires us to use a small OS. Schlehlein said that he is preparing his own distribution of Linux and that we could probably use that as well. All that we require is a solid networking driver and USB driver. We would also prefer a good environment to write the C++ code.

The XML RPC library that we will be using is not completed but we should have access to that fairly soon. It should work in C++.

The final list of the simulator peripherals that we are going to implement include: the helmsmen's display, rudder wheel, and the torpedo fire control. We will not be able to get extra fire control boxes like we did the helmsmen's display and rudder wheel so we will have to make that our selves.

The fire control team has done some preliminary work on the boot ROMs. Schlehlein said that he would he would share that information with the I/O team and we could possibly continue work on that together.

The layout for the I/O section of the simulator includes:

- 1) A Master Control PC
- 2) A USB to I<sup>2</sup>C interface board
- 3) An I<sup>2</sup>C hub
- 4) A servo controller for the helmsmen's display
- 5) A shaft encoder and controller to get reading from the rudder wheel
- 6) A controller or two that control the fire control boxes

We have to take care of the following soon:

- 1) Look for a shaft encoder to work with the rudder wheel
- 2) Find a good USB to I<sup>2</sup>C board
- 3) Find a chip to handle all the servers in the helmsmen's display.
- 4) Find two chips to handle the fire control boxes
- 5) Find a chip to handle the shaft encoder output from the rudder wheel

We did a large amount of searching for a way to interface the Master Control to the controller bus. We found a number of interfaces but they only had drivers for windows.

-323

### **01-10-2003-13:00 to 16:00**

Our group continues to look for a way to interface the Mater Control to the I<sup>2</sup>C controller bus. We found a few implementations that would work exceptionally well; however, drivers only exist for windows. What we decided we will probably do is connect a standard uC to the PC through the serial port and interface that to a parallel to I<sup>2</sup>C converter.

-180

### **01-13-2003-14:00 to 19:02**

We finalized our criteria for are parts selection and the peripherals that we will be controlling. We had a discussion of the use of servos vs. stepper motors for control of most of the analog peripherals. We started to decided on the final amount of digital and analog I/O ports we will be needing. One issue continues to be how to mount the rudder wheel and get data off of it. The rudder wheel requires an amount of resistance as well.

We need to do:

- 1) Exactly what peripherals we will be controlling
- 2) The type and amount of ports required to control the peripherals
- 3) The controllers and boards used to control the peripherals
- 4) The type of rotary display control used behind the peripherals

The list of peripherals includes:

- 1) Motor order telegraph (left and right motor speed indicator)
- 2) Rudder angle indicator
- 3) Rudder wheel
- 4) Compass
- 5) Torpedo status indicator and firing boards
  - a. Status LED's

- b. Fire buttons
- c. Torpedo control

Left Motor Speed– Servo  
Right Motor Speed– Servo  
Rudder angle indicator-Servo  
Compass – Stepper motor  
Torpedo status indicator and firing boards  
    26 LED's-5 digital out though a MUX for address and 1 data.  
    10 Binary switches-10 Digital inputs  
    2 Fire buttons -2 Digital inputs  
Rudder Wheel input (shaft encoder)

We ended up shit canning I<sup>2</sup>C all together. Schlehlein was concurred about the cost of all the hardware to implement the bus as the controller bus used by everything. Instead we are going to use parallel, PS2, and serial ports for connections to all the peripherals. There will now be two PC's used to control the peripherals, one for the torpedo control and the other for the rudder wheel, and various other analog displays.

-302  
--1595

#### **01-14-2003-16:45 to 21:00**

The entire team began the technology report. The first draft was finished.

-255  
--1850

#### **01-15-2003-15:23 to 18:05**

I adding the finishing touches on the technology report after Scott looked it over and made corrections to it. Every group member got on MSN to continue work on the paper. A messy draft was sent to Steve for him to review and format correctly. When he gets done every group member will look it over then I'll ship it.

-162  
--2012

#### **01-16-2003-12:00 to 15:00**

Did our presentation

-180

#### **01-27-2003-13:16 to 16:56**

We began to put together a list of items that we need to order or obtain

- 1) Controller 001(VM-1 with the application board)
  - a. 4 digital output with pulse width modulation for servo and stepper motor
  - b. 1 digital input from stepper motor
  - c. ~6 digital inputs for shaft encoder
  - d. RS232 serial for interface to Master Control
- 2) Controller 002(VM-1 with application board)
  - a. 10 digital inputs for the switches for torpedo fire selection
  - b. 2 digital inputs for the torpedo fire buttons
  - c. up to 26 digital outputs for the torpedo status lights
- 3) 2 Servos for left and right motor speed indicators
- 4) Servo for rudder position indicator
- 5) Stepper motor for gyrocompass
- 6) Shaft Encoder for rudder wheel(Dynapar HS35)
  - a. <http://www.dynapar-encoders.com/frameaset.asp?Page=seriesHS35.htm>
- 7) 2 Torpedo fire buttons for Forward and Aft
- 8) Forward torpedo firing control box
  - a. 6 torpedo selection switches
  - b. 15 Status lights
- 9) Aft torpedo firing control box
  - a. 4 torpedo selection switches
  - b. 11 status lights
- 10)The master control PC
  - a. 2 RS232
  - b. NIC with boot ROM for network boot

We have to come up with a design for the rudder wheel so we know what kind of encoder to get and how to put it on.

We need to get a hold of the crappy embedded Linux that we will be using and get it loaded on the Master Control so we can start to mess around with the software libraries we will be using. Nathan said something about getting a boot server in the SDL so we can perform network boots in it for development and testing. This should be done soon. We need to know about the limitations of our Master Controls software environment under the constraints.

After further investigation in to the VM-1 uC, it will fulfill all our I/O needs for both controller 001 and 002. A shaft encoder was found as well. The encoder that was decided upon was the Dynapar HS35. This encoder is a bidirectional hollow encoder that supports steps from 1-1024 increments per reading.

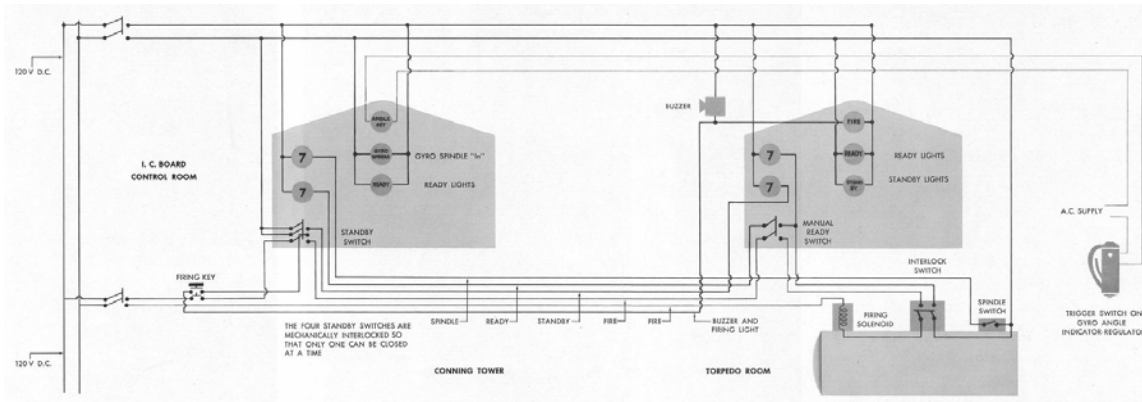
I started a detailed schematic of the topology.

Nathan just informed us that the museum will not provide any materials at all for our project. We will have to make or purchase all our dials and gages. We are briefly looking in to getting entire gages that take a digital input instead of making our own with servos.  
-220

**01-29-2003-12:48 to 17:07**

We started to plan out how we will get the design report done. Because the museum fucked us we need to get dials that look “authentic” but I am guessing that this will probably not happen now. We probably have to completely make the gages now.

We went through a schematic of the torpedo status lights and figured out how they worked.



We need to talk to the software team to find out what variables they want to deal with.  
-259

**02-03-2003-14:00 to 19:08**

We started to work on the design report due in week nine. We completed and major system components and their interrelationships. We generated a project schedule for the spring quarter. We put together a parts list but it is not yet complete.  
-308  
-2979

**02-04-2003-14:00 to 16:00**

Continued working on the design report. I started work on the complete circuit schematics. Also, I helped Wadell and Rittner with the function prototypes.  
-120

**02-07-2003-13:00 to 16:00**

Using the VM-1 documentation, I began laying out a complete circuit schematic. The documentation is not very good. You need to look at multiple PDF files to see what data port relates to what pins.

-180  
--3279

#### **02-10-2003-10:25 to 20:00 less 1 hour**

I began to wrap up the schematic. We found that, contrary to the board's high level documentation, the VM-1 boards do not have enough digital I/O. I designed a circuit that allows a few outputs on the board to drive all the LED using multiplexers, latches, and resistors. We decided that we would use a shift register to drive all the LED's using only 8 pins.

The schematics are finished. I am gathering all our information and pasting it in to the report.

I am finishing up the report to submit.

I am getting all my time I did design work from my log to include in the report then I am going to submit it

-505  
--3784

#### **02-14-2003-15:30 to 16:00**

Began working on the final presentation and report.

-30  
--3814

#### **02-16-2003-19:30 to 02-17-2003-02:55**

Continued work on final presentation and final report.  
Completed the peer evaluation.

-445  
--4259

#### **02-18-2003-15:00 to**

Met with group to finish the final report and presentation.

-315  
--4574

***Time spent by Brian Rittner on the design phase of project***

Complete and most recent listing can be found at: <http://subsim.hn.org/plantrack/>

Username: rittnerb

Password: Available on request

***Time spent by Scott Wadell on the design phase of project***

Complete and most recent listing can be found at: <http://subsim.hn.org/plantrack/>

Username: wadells

Password: Available on request

***Time spent by Steve Nolte on the design phase of project***

Complete and most recent listing can be found at: <http://subsim.hn.org/plantrack/>

Username: noltes

Password: Available on request

## Project Schedule

<b>Milestone</b>	<b>Date</b>	<b>Week</b>
Have the Master Control network booting and usable		1 – start
Serial Interface on Master Control		1 – mid
Get debounced input from fire buttons displayed on terminal		1 – end
Get Readings from switches on fire boxes		2 – start
Complete H-bridge circuit with the stepper motor		2 – mid
<b>Press Release for Seed Show</b>	<b>March 21</b>	<b>2 – end</b>
<b>Implementation Schedule</b>	<b>March 24</b>	<b>3 – start</b>
Control a servo using a controller from a terminal on the PC		3 – start
Control the stepper motor using the controller from a terminal on the PC		3 – end
Control status LED's from a terminal on the PC		4 – end
Get readings from the shaft encoder from a terminal on the PC		5 – start
<b>Mid-term Report</b>	<b>April 11</b>	<b>5 – end</b>
Calibrate and align gauges		6 – start
Master Control SW handles the terminal interface		6 – mid
XML-RPC interface on Master Control		6 – end
<b>Logs submitted for review</b>	<b>April 18</b>	<b>6 – end</b>
Bridging software on the Master Control		7 – start
Control the system using the XML-RPC		7 – end
<b>Logs submitted for review</b>	<b>April 28</b>	<b>8 – start</b>
Install the servos in the gauges (perhaps done for us)		8 – mid
Install the stepper motor in the compass (perhaps done for us)		8 – mid
Install the LED's on the fireboxes (perhaps done for us)		8 – mid
Install the switches on the fireboxes (perhaps done for us)		8 – mid
<b>Projects Finished!</b>	<b>May 2</b>	<b>8 – end</b>
<b>Implementation Report</b>	<b>May 5</b>	<b>9 – start</b>
<b>Logs submitted for review</b>	<b>May 12</b>	<b>10 – start</b>
<b>Final Project Report</b>	<b>May 16</b>	<b>10 – end</b>

# Miscellaneous Design Documentation

## *Circuit Layout*

The following was derived from the documentation for the VM-1 board. The paper work details the use of channels on how they can each be used for multiple purposes but these channels are not pins. You must look at more documentation to determine the pin and port that a channel uses.

## **Channel Capabilities**

The following is a list of available channels and their raw digital I/O capabilities.

<b>Channel Capabilities for uC002</b>				
<b>Inputs</b>	<b>Outputs</b>	<b>Both</b>	<b>Serial</b>	<b>Misc</b>
1				
2				
		3		
		4		
		5		
		6		
		7		
		8		
		9		
		10		
			11	
			12	
			13	
			14	
				15
		16		
		17		
		18		
19				
				20
21				
		22		
		23		
		24		
		25		
		26		
		27		
		28		
		30		
		31		

32  
33  
34  
35  
36  
37

40  
41  
42  
43  
44  
45  
46  
47

## Pin Usage

The following is a chart specifying the channel used, how it is used, and the channels associated port and pin.

Microcontroller001				
Usage	Port	Pin	Channel	Connect
PWM-1	J2	21	3	Left Motor Servo
PWM-2	J2	8	4	Right Motor Servo
PWM-3	J2	20	5	Rudder Position Servo
Digital Out	J2	22	7	Gyrocompass stepper motor 1a
Digital Out	J2	24	8	Gyrocompass stepper motor 1b
Digital Out	J2	26	9	Gyrocompass stepper motor 2a
Digital Out	J2	28	10	Gyrocompass stepper motor 2b
Digital In-1	J2	11	19	Input from stepper motor for zero position Quadrature shaft encoder for Rudder
QSE-1a	J2	23	1	wheel(a) Quadrature shaft encoder for Rudder
QSE-1b	J2	10	2	wheel(b)

Microcontroller002				
Usage	Port	Pin	Channel	Connect
	J2	4	1	Front Torpedo Fire button
	J2	6	2	Torpedo selection switch for Front 1
	J2	11	19	Torpedo selection switch for Front 2
	J2	21	21	Torpedo selection switch for Front 3

J2	45	40	Torpedo selection switch for Front 4
J2	46	41	Torpedo selection switch for Front 5
J2	43	42	Torpedo selection switch for Front 6
J2	44	43	Aft Torpedo Fire button
J2	41	44	Torpedo selection switch for Aft 1
J2	42	45	Torpedo selection switch for Aft 2
J2	39	46	Torpedo selection switch for Aft 3
J2	40	47	Torpedo selection switch for Aft 4
J2	10	3	ShiftReg1 Clock
J2	12	4	ShiftReg1 Data
J2	14	5	ShiftReg2 Clock
J2	16	6	ShiftReg2 Data
J2	22	7	ShiftReg3 Clock
J2	24	8	ShiftReg3 Data
J2	26	9	ShiftReg4 Clock
J2	28	10	ShiftReg4 Data

## Design documentation

The following data was taken from the VM-1 Object Reference Manual.

### Arrays

Arrays may be created in the NV-ROM (non-volatile RAM area) to store variable parameters that should be retained over power cycles. Arrays can hold data of the following types, one per element:

- 8, 16 or 32-bit integer
- Floating point number
- Pointer (to global variable)
- String constant

### Constant or Variable data

An Array may contain *constant* data (i.e. *read-only*), or *variable* data. These two major types of array are created in very different ways. Arrays of constant data are defined at the command line, in a similar way to procedures. This is because they 'live' in ROM, and so have a similar life cycle. They are not created with MAKE or NEW. They are not defined within a procedure. Arrays of variable data are defined either with MAKE or NEW, just like other objects, or by taking a copy of a constant array.

### Summary of messages

Checksum

Copy

Element

Length

Pointer

Reset

Valid

PRINT

## Creating a Constant Array

Constant arrays are created using the following syntax at the command line (i.e not within a procedure's TO...END):

```
Array obj (Any Constant prototype , Int Constant n)
Any constant_data ,
Array
Any constant_data ,
...,
END
```

*Prototype* is a value that typifies the type of data elements that the array will hold:

### Prototype Element Type

8	8-bit integer
16	16-bit integer
32	32-bit integer
*	Floating point
"Some text"	String constant
@dummy_name	Pointer to a global

n defines the size of the array – the number of elements it holds.

Each line of constant data defines the value of an element. If there are fewer values defined than the size of the array, then the rest of the array is filled with the last defined value.

### Examples:

The following defines an array of five 8-bit integers called lookup\_data with the values 2, 3, 5, 7 & 6.:

```
Array lookup_data (8 , 5)
2 , 3, 5, 7 , 6
END
```

The next example defines an array of six string constants called *phrases* with the values “Hello” “Goodbye” “Chow” “Chow” “Chow” and “Chow”.

```
Array phrases (“” , 6)
“Hello”
“Goodbye”
“Chow”
END
```

The example below defines an array of three procedure-pointers called *handler*.

```
Array handler (@dummy , 3)
@first_proc
@second_proc
@third_proc
END
```

### Creating a Variable Array

Variable arrays are defined in the normal way with MAKE (or NEW), or by taking a copy of a constant array. See the Copy message below.

```
MAKE obj Array (Any prototype , Int n , ...)
```

*\*Prototype and n are very similar to constant arrays.*

The ... indicate that you can put some initializing data in the parameter list. If there are fewer initializers than elements, then the value of the last initializer is used to fill the array. If no initializers are present, then the array is *not initialized*: the data may be any random set.

Take care not to put too many initializers in as you may run out of space on the parameter stack. A sensible limit is 10 or so. If you need more initializers than this use a constant Array and take a copy.

## Checksum

**obj . Checksum (Int num) ⇔ Int**

Checksum will take a 16-bit checksum (not specified currently) of all the bytes in the array and put its value into the array header. It will also return the value of the checksum so that it may be used for other purposes.

## Copy

**obj . Copy ⇔ Array**

Copy returns a copy of the array in RAM – optionally in the NV-RAM.

```
-->variable_data := lookup_data . Copy
```

This creates a new array initialized to the contents of the original. You can modify the elements as the copy will be in RAM.

## Non-volatile Copy

**obj . Copy (1) ⇔ Array**

This version of the copy message will create an ‘overlay’ copy of the array in the NVRAM (non-volatile RAM area). The data of the original array is not copied, only some of the header information. This means that any data in the ‘new’ array will be persistent over power cycles.

Note that in order for the new non-volatile Array to be located at the same address each time the controller resets, the Array-Copy messages and any other functions that use the NV-RAM (like the RAM filing system) should occur *once only, and in the same order, each Reset*. The best way to achieve this is to put them in the *init* routine, and then always run your code by typing ‘Run’ at the command line, or by powering-on in RUN MODE.

```
TO init
Free (2) := 10000 ; enough NV-RAM for all my n.v. Arrays
nv_array1 := array1 . Copy(1)
nv_array2 := array2 . Copy(1)
nv_array3 := array3 . Copy(1)
...
END
```

If there is not enough NV\_RAM to create the copy, then a ‘Resource Error’ is given.

## Element

**obj . Element (Int num) ⇔ Any**

Each element of the array can be read or set individually. *n* specifies the element number. Elements are accessed using the Element message:

```
-->Print phrases . Element (1)
Goodbye-->
```

Venom abbreviation allows `.` to replace `.Element()`, so you could also use:

```
-->Print phrases . (0)
Hello-->
```

Elements will be checked for the correct type when they are written. For integer arrays, the written values will be masked to the correct size.

## Length

**obj . Length** ⇔ Int

Length returns the number of elements in the array.

## Pointer

**obj . Pointer** ⇔ Int

Pointer returns an integer which points to the start of data in the array. Data is arranged contiguously: elements are in order with no gaps. String constants are arranged as a list of pointers followed by the string data. The data in an array of pointers has no defined format.

*[The pointer will always be aligned to the number of bytes required by the largest object used by the host processor. In the case of the VM-1 it will be aligned to an even address.]*

## Reset

Reset *operates only on Arrays that have been created using the Copy message on another Array*. It will copy the data from the original into the current array. This may be useful to reset to default values into non-volatile arrays.

## Valid

Valid will return TRUE if the checksum in the array header matches the checksum of all the data in the array. Valid will be TRUE immediately after a *Checksum* message. It will be FALSE immediately after the value of any element in the Array is changed. Valid is useful for checking the integrity of non-volatile parameters.

```

TO init
Free (2) := 10000 ; enough NV-RAM for all my n.v. Arrays
nv_array1 := array1 . Copy(1)
IF NOT nv_array1 . Valid ;Check the integrity of the data.
[
nv_array1 . Reset ; corrupt? Set the defaults
nv_array1 . Checksum ; set the checksum.
PRINT "PARAMS CORRUPT – DEFAULTS LOADED."
] ...
END

```

## Print

**PRINT obj** *.fw*

Printing the array will print out each of the elements in order.

## **ASYNCHRONOUS SERIAL**

AsynchronousSerial objects interface to serial communication ports. The two hardware ports can operate at standard rates up to 38400 baud, or higher non-standard rates. Handshaking is optional: hardware ('RTS & CTS'), software (XON / XOFF) or none.

## Summary of messages

MAKE  
 Baud  
 Escape  
 Flush  
 Format  
 Free  
 Get  
 Handshake  
 InputBuffer  
 Look  
 OutputBuffer  
 Put  
 Queue  
 Timeout  
 Valid  
 PRINT TO

## Creation

### **MAKE obj AsynchronousSerial(Int baud\_rate, Int port, Int handshake)**

When making an AsynchronousSerial object, the baud rate, port and, optionally, handshaking are specified. Generally the VM-1 uses a group of four channels for each serial port: Transmit, Receive, Handshake input and Handshake output. If hardware handshaking is not enabled then the handshake channels are free to be used for other purposes. Here we create a serial communication object on port 1 talking at 38400 bd, with hardware handshaking:

```
MAKE serial AsynchronousSerial(38400,1,1)
```

*An object like this is created by the default startup routine at either 38400 or 9600 baud, depending on the state of the User Switch.*

The *port* values are port **1** and port **2**.

The *baud rates* available on the main serial port are from **31** to **500000**.

The *handshaking* takes values of **0** (none), **1** (hardware handshaking) or **2** (software handshaking). The input and output buffers are both 256 bytes long. This is not configurable.

You may need to connect RS232, RS485 or other line transceivers to your VM-1 in order to use this object to communicate with other equipment. Our development kits include these transceivers on the application board.

## Baud

**obj . Baud** ↔ Int

The Baud active variable controls the baud rate of the serial port.

To ensure a baud rate change happens smoothly, you should ensure that the serial output buffer is empty before changing baud rate. The baud rates available on the main serial port are from 31 to 500000. In general the exact baud rate is not possible. Printing *Baud* gives the exact rate used.

## Escape

**obj . Escape** ↔ Int

The Escape active variable turns *CTRL-C* (break) and *CTRL-T* (list all tasks) on and off. When escape is 0, CTRL-C & CTRL-T are just treated as normal characters.

## Flush

**obj . Flush**

This message will flush the serial input buffer, discarding any characters in it. It may be applied to the *AsynchronousSerial* object itself, or to an *InputBuffer* subobject.

## Format

**obj . Format** ↔ Int

The Format active variable allows the serial communication format to be changed. The integer value is a set of flags coded as bits in a binary number:

BIT	VALUE	MEANING...	...WHEN 0	...WHEN 1
0	1	Data length	8	7
1	2	No. stop bits	1	2
2	4	Parity type	Even	Odd
3	8	Parity used	No	Yes

The default format is '8-NONE-1' which has a format value of zero.

## Free

**obj . Free** ↔ Int

Free returns the amount of free space in the serial input or output buffer. If it is applied to the *AsynchronousSerial* object itself, it returns the input buffer value.

## Get

**obj . Get** ↔ Int

The Get message returns a character from the serial port, but waits if there isn't one available yet. CTRL-C characters are trapped for the *Escape* function. If these characters need to be received, Escape may be turned off with *serial.Escape*.

## Handshake

**obj . Handshake** ↔ Int

Handshake allows the handshaking function of the port to be set.

Value	handshaking
0	NONE
1	Hardware
2	Software

Software handshaking is performed with the XON and XOFF characters in the ASCII character set. If enabled, these characters assume their control function and may not be sent over the serial link.

## InputBuffer

**obj . InputBuffer** ↔ <InputBuffer>

This returns a sub-object to the AsynchronousSerial object that allows the input side of the serial stream to be treated distinctly. Further messages may be sent to the input buffer object:

```
sib := serial.InputBuffer  
WHILE sib.Queue [serial.Get] ; read chars while they are there.
```

The input buffer object may also be used with 'dot-chaining'. This is more readable but slightly less efficient.

```
WHILE serial.InputBuffer.Queue [serial.Get]
```

## Look

**obj . Look** ↔ Int

Look fetches the next character in the serial input buffer, or -1 if there was no character to fetch.

## OutputBuffer

This returns a sub-object to the AsynchronousSerial object that allows the output side of the serial stream to be treated distinctly. Further messages may be sent to the output buffer object:

```
sob := serial.OutputBuffer  
IF sob.Free [serial.Put(c)] ; send chars if there is room.
```

The output buffer object may also be used with 'dot-chaining'. This is more readable but slightly less efficient.

IF `serial.OutputBuffer.Free [serial.Put(c)]`

## Put

**obj . Put ( Int character )**

The Put message sends a single character to the serial output buffer, waiting if the buffer is full. The Put message may be used in preference to PRINT CHR to send binary data, because PRINT CHR cannot send character 0.

▼ *character* will be masked to the range 0 – 255

## Queue

**obj . Queue** ↔ Int

Free returns the number of characters waiting in the serial input or output buffers. If it is applied to the AsynchronousSerial object itself, it returns the input buffer value.

## Timeout

**obj . Timeout** ↔ Int

This active variable controls the timeout for software handshaking. If an XOFF character shuts off the transmitter and the subsequent XON is missed for some reason, the transmitter would stay off forever. However if timeout is set to a non-zero value, the transmitter will be turned on again after the timeout period. Timeout defaults to 10,000 (10 Seconds) currently. It may be turned off entirely by setting it to zero. Its value is set in milliseconds, up to a bit more than 1,000,000 (~16 minutes). Timeout is set with a resolution of 64mS. When reading the timeout value you may notice that the value you read is has been rounded down to the nearest 64.

## Valid

**obj . Valid** ↔ Int

*Valid* is used to detect errors in serial reception. It returns three error flags as bits in a binary number. The flags are all reset to zero once *Valid* has been read.

BIT	VALUE	MEANING	DESCRIPTION
0	1	Parity Error	Incorrect parity in the received character
1	2	Framing Error	The stop bit was 0 (should be a 1)
2	4	Overrun Error	One or more characters lost by serial Rx

You should be able to use *Valid* to detect *serial breaks*, i.e. when the serial line is put into the active state for a period (much) longer than the character frame. If *Valid* continuously indicates a framing error, and no new characters have been received then there is a serial break.

## Accepting Print

**PRINT TO obj , <print list>**

The serial port sends out the printed characters. The following print keywords are supported:

CR	ASCII 13 & 10
BEEP	ASCII 7
CHR <i>n</i>	Send character ASCII <i>n</i>

## DIGITAL

Digital controls both individual and groups of digital channels. The output state may be set and/or the input state read. Several different digital I/O devices are supported. Digital channels on the controller itself are numbered from 1 to 47

## Summary of messages

MAKE  
Asserted  
High  
Low  
NotAsserted  
Off  
On  
Output  
Pulse  
Toggle  
Value  
PRINT

## Creation

### **MAKE obj Digital(Int channel , Int channel2)**

Single channels or groups of channels (ports) may be created on the different devices available. If only the **channel** parameter is specified then a single channel digital I/O driver is created. If **channel2** is also specified, then a group of digital I/O channels from channel to channel2 inclusive, is created. These may be accessed as a single port. If a multi-bit digital is being specified, then the channels must be within the same 8-bit group (i.e. on the same IC), eg. 128 - 135 but not 129 - 136. Digital ports are not possible on channels 1 - 47 on the VM-1 itself. For example:

-->MAKE channel Digital(17) ;A digital I/O on VM-1 Channel 17  
-->MAKE port Digital(128,131) ;A 4 channel port on the 1st I2C bus

You may need to connect PCF8574 ICs to your VM-1 system in order to use parts of the channel range. Other objects may already be using some of the channels, precluding their use with Digital. See the channel table in the VM-1 Datasheet.

▼ There are a possible 128 digital I/O on each I2C bus. See the VM-1 Datasheet for numbers on the VM-1. Channels grouped into a port are guaranteed to reach a new state with only as much skewing as the hardware of the digital I/O device itself introduces (often only nanoseconds).

## Asserted

**obj . Asserted** ⇔ Flag

The Asserted active variable holds the state of the digital channel. The channel is turned on when Asserted is set TRUE and off when Asserted is set FALSE. When it is read, Asserted returns TRUE or FALSE depending on whether the channel is On or Off, respectively. Whether or not the channel is an input or an output, Asserted returns the state of the channel as if it were an input – i.e. it reads the actual voltage level rather than ‘what it ought to be. For groups of channels, setting Asserted will set the whole group to the requested state, and reading Asserted returns FALSE if any of the channels are in the Off state, TRUE if all the channels are in the On state.

```
-->TO thermostat  
MAKE d digital(32)  
FOREVER  
d.Asserted := temperature < 50  
END
```

## High

**obj . High**

High sets the digital channel to its high voltage state. If the channel is not already specified as an output channel, this message turns it into one. I2C channels are pseudo tri-state, so setting a channel to high is the same as setting it to be an input. For groups of channels (ports), High sets the whole group into the high voltage state.

## Low

**obj . Low**

Low sets the digital channel to its low voltage state. If the channel is not already specified as an output channel, this message turns it into one. For groups of channels (ports), Low sets the whole group into the low voltage state.

## NotAsserted

### **obj . NotAsserted** ⇔ Flag

The NotAsserted active variable holds the inverse state of the digital channel. When setting NotAsserted, the digital channel is turned on when NotAsserted is set FALSE and off when NotAsserted is set TRUE. When read, NotAsserted returns FALSE or TRUE depending on whether the channel is On or Off, respectively. Whether the channel is an input or an output, NotAsserted returns the inverse state of the channel as if it were an input. For groups of channels, setting NotAsserted will set the whole group to the requested state, and reading NotAsserted returns TRUE if any of the channels are in the Off state, FALSE if all the channels are in the On state.

```
-->TO thermostat  
MAKE d digital(32)  
FOREVER  
d.NotAsserted := temperature > 50  
END
```

## Off

### **obj . Off**

Off sets the digital channel to its inactive state, which is high for all the devices presently supported. If the channel is not already specified as an output channel, this message turns it into one. I2C channels are pseudo tri-state, and so setting a channel to Off is the same as setting it to be an input. For groups of channels (ports) Off sets the whole group into the inactive state.

## On

### **obj . On**

On sets the digital channel to its active state, which is low for all the devices presently supported. If the channel is not already specified as an output channel, this message turns it into one. For groups of channels (ports) On sets the whole group into the active state.

## Output

**obj . Output** ⇔ Flag

The Output active variable allows the *I/O direction* of a channel or port to be set or read. Setting Output TRUE makes the channel or port into an *output*, and setting Output FALSE makes the channel or port into an *input*. Output returns TRUE or FALSE depending upon whether the port or channel is an output or an input respectively. If there is a mixture of inputs and outputs in a port, Output will return TRUE. Digital channels on PCF8574 ICs on an I2C bus always return TRUE for Output because they are only pseudo tri-state devices. The I/O direction of channels in a port is guaranteed not to be skewed, up to the limit defined by the hardware.

## Pulse

**obj . Pulse**

Pulse momentarily pulses the channel or set of channels to their opposite state.

-->MAKE d digital(128)

-->d . Pulse

The minimum pulse width is 1 microsecond, but is likely to be considerably longer than this.

## Toggle

**obj . Toggle**

Toggle flips the state of a digital channel from On to Off and vice versa. For groups of channels, the states of all the channels are flipped.

## Value

**obj . Value** ⇔ Int

Value holds the numeric representation of the state of the channels in a port made up from a single channel or group of channels. A high state is a binary 1 and a low state is a binary 0. States output using Value are guaranteed not to be skewed up to the limit defined by the hardware.

The least significant bit of the number is given by the state of the lowest numbered channel in the port.

As with On, Off etc., if the channel is not already specified as an output channel, setting a value using this message turns it into one. Also, reading the value will return the numeric representation of the levels actually on the channels, whether the port is inputs or outputs.

▼ Uses values in the range 0 to  $2^n - 1$ , where  $n$  is the number of channels in the port.

It may be set to any value. Binary bits in the value that overflow the port size are masked off.

Output to multi-bit ports is guaranteed not to be skewed up to the limit defined by the hardware.

## Printing

### PRINT obj :f1

If no format specifier is used, the state of the channel or port (as defined in Asserted) as "ON "or "OFF", always with 3 characters is printed. If a format specifier greater than zero is supplied, then the *value* of the port is printed in square brackets.

```
-->MAKE d digital(128)
-->PRINT d
OFF-->d.On
-->PRINT d
ON -->PRINT d:3
[Digital: 0]-->
```

## FILE

A file object controls a file in a file system. A file is a sequence of data items of one of these types:

- 8 bit unsigned integer
- 16 bit signed integer
- 32 bit signed integer
- 32 bit floating point
- Text

Data can be written to the end of a file, and can be read in sequence from the beginning of the file or from any selected point.

Any element of a file can be accessed by its numerical position in the file and read or changed. Any file can be printed, in a format that depends on the data type, and a text file can also accept a print job.

The interface is designed to be as similar as possible to that of the buffer object type.

## Summary of Messages

Put  
Get  
Queue  
Reset  
Empty  
Length  
Readpoint  
Close  
Element  
Find  
Lock  
Name  
Unlock  
Help  
PRINT TO  
PRINT

### Creation

The only way to create a file variable is by sending the Open message to a FileSystem object. This is also documented in the FileSystem section for the Open message.

**fs.Open(String name, Any Prototype [,Int maxlength])**

=> file object

name - is the name of the file. Upper and lower case letters and numerals are allowed in the file name, as are the following punctuation characters:

**! \$ % - \_ + ~ . / #**

The maximum file name length is 15 characters.

Note that '/' and '.' are not interpreted in any special way, and that this file system does not support subdirectories.

Prototype denotes the data type of the file contents and is one of the following:

8	8 bit unsigned integers
16	16 bit signed integers
32	32 bit signed integers
0.0 or any floating point value	Floating point
"" or any string value	Text

If a file with the same name exists, it is opened with the read pointer set to the beginning of the file and the write pointer set to the end of the file. If no file existed, an empty file is created with read and writes pointers at 0. If a third parameter maxlength is given, it sets a maximum length for the file. If created this way, any write to the file that results in its length exceeding that target will cause the first 512 bytes to be removed from the file. This is useful for log files where only recent data is of interest, and in such cases can remove the need for extra housekeeping code to prevent the filesystem from running out of space. Note that the maxlength parameter is in bytes and intended for use with text files. This parameter, if specified, must be at least 1024.

### Example

```
a := fs.open("aaa.txt", "")  
b := fs.open("temp", 1.0)
```

### Messages

#### Put

##### f.Put(Any value)

The data type of the parameter depends on the file type.

For integer file type, the value must be an integer. It is truncated if necessary to the size of the file data type. For the floating point file type, the value must be a floating point type. For a text file, the data can be a single character or a fixed string. The data is written to the end of the file.

## Get

**f.get** => Any

Returns a value from the current read point in the file and advances the read pointer to the next element. It is an error to attempt to read past the end of the file. (See Queue Message for how to avoid this)

The type of data returned is:

Integer for all type of integer file

Float for a float file type

Integer value of a single character for a text file

Note that a text file created with PRINT TO and containing line breaks will return the sequence 13, 10 at the line breaks (ASCII CR and LF).

## Queue

**f.Queue** => Int

Returns the number of elements remaining to be read from the file. For a text file, an element is a single character.

## Reset

**f.Reset**

Resets the read pointer to the beginning of the file.

This is equivalent to `f.Readpoint := 0`.

## Empty

**f.Empty**

Removes all the data in the file, leaving read and write pointers at zero. The file is then in the same state as if it had been newly created.

## Name

**f.Name** ⇔ string

Gets or sets the file name. Assigning to `f.name` renames the file.

The returned value is a fixed string whose physical address is the file name in the directory entry. As with any message returning a fixed string, great care should be taken if assigning it to a variable, as the string represented by that variable will only be valid as long as the file is open. Completely safe and useful operations include:

Printing a file name: `PRINT f.name`

Appending a file name to a text buffer: `buf.Put(f.Name)`

## Length

**f.Length** =>Int

Returns the length of the file in elements of the file's specified type.

For text files, this is the number of characters, counting each line separator as the two characters CR, LF.

## Readpoint

**f.Readpoint** ⇔ Int

Gets or sets the point at which the next Get message will read data from the file. The file starts at position 0.

## Close

**f.Close**

The file remains in the file system but is no longer associated with any variables that referred to it and no messages can be sent to it.

## Element

**f.Element(Int Elementnumber)** ⇔ Any

Sets or returns a single data element of the file.

For a text file the value is treated as an integer.

The file starts at Element 0. It is relevant to know that in a text file created by PRINT TO, a line break takes up two character elements (CR=13 and LF=10).

## Find

**f.Find(String pattern [, Int startpos])** => Int

Searches a text file for a string supplied either as a fixed string or as the contents of a text buffer.

Find returns an integer showing the element position of the beginning of the first instance of the search pattern if found, or -1 if not found.

The search start at the character position startpos in the file if specified, or else at the beginning of the file which is equivalent to a startpos value of 0. The search pattern has a maximum size limit of 255 characters. The Boyer-Moore search algorithm is used, which is very fast, especially with a long search pattern.

## Lock

**f.lock =>Int**

Prevents other tasks from accessing the file until it has been unlocked by an Unlock message. Returns the current lock level which will have been incremented by applying this Lock message.

**f.Lock(Int Level)**  
**f.lock := Level**

Either of these forms sets the locking level to an explicit number. A level of 0 unlocks the object.

## Unlock

**f.unlock**

Unlocks the file allowing other tasks to access it.

## Help

It is worth noting that the standard help message, which shows the type of a variable, additionally shows the file name for a file variable.

## Example

```
-->HELP a  
It is a text file named "abc.txt"  
-->
```

## PRINT TO

**PRINT TO f, list**

A text file can be the destination of a print operation. The print output is appended to the end of the file in exactly the same way as by a series of **Put** messages.

## PRINT

**PRINT f[: Int n1[: Int n2]]**

Lists the contents of the file. Text Files are displayed in their normal text format other file types are displayed one item per line.

If the colon(s) and format code(s) are present they are interpreted in the same way as when printing Buffers of various types:

File Type	1 <sup>st</sup> format number	2 <sup>nd</sup> format number
Text	If positive, print first n1 chars of file	Not present

	If negative, print last (-n1) chars of file	
Text	First char position in file to print	Number of chars to print.
All Integer Types	Minimum field width (number of characters to print per number)	Not used
Float	Minimum field width	Number of decimal places

## OPERATING SYSTEM

This is the class that provides access to system-wide features of the hardware, operating system and compiler. Note that Venom-SC allows all system messages to be called by just typing the message name alone, e.g.

`Reset`

Is the same as

`system.Reset`

### Summary of messages

MAKE  
Checksum  
Debug  
ErrorAction  
Free  
Output  
Protect  
Reset  
Run  
RunMode  
UserSwitch  
Valid  
PRINT  
Creation

### MAKE obj OperatingSystem

An object called system of type OperatingSystem is made by the default startup routine:

`MAKE system OperatingSystem`

▼ It is only useful to make one OperatingSystem object.

### Checksum

This is used to find the checksum of all the bytes in the Venom-SC code.

## Copy

*Copy* is used to either copy the current Venom-SC flash device in its entirety (Venom system and any application code – or – to download a new version of Venom-SC from the Micro-Robotics website [www.microrobotics.co.uk](http://www.microrobotics.co.uk) The syntax of the flash programming commands is:

*Copy*(0,flags) ; to copy the current flash  
*Copy*(1,flags) ; to download a new version

The instructions and progress reports for each operation will appear at the terminal. The *flags* parameter is a binary pattern that tells the copy message which optional operations to perform in sequence. In general use a value of *TRUE* (-1) for the most comprehensive operation. See the table for the meaning of individual flags.

*Copy*(0,TRUE) ; ID, erase, program, verify.

FLAG	VALUE	ACTION
ID	1	Print 'Silicon signature' ID
ERASE	2	Erase the device
PROGRAM	4	Program the device
VERIFY	8	Verify the program

In order to use the *Copy* message you will need the correct hardware. This is currently the VM-1 Control Computer and the 5805 Application Board. Code that has been downloaded from the website can't be verified by *Copy*, but you can verify the new code by calling the *Valid* message when you first use it.

*Important note: If the device has not been erased before programming, then the copy or download operation will fail ungracefully. The controller will reset on the watchdog. You will probably have to reset the controller and you may have to exit from the terminal emulator.*

## Debug

**Debug** ( Int n, ... ) ⇔ Flag

The debug command has many options allowing the internal state of the Venom-SC compiler & OS to be viewed and modified. These have not settled enough to document as yet. Typing Debug alone will list the available options.

## ErrorAction

Setting this to the value '1' will force Venom to reset the controller hardware if a runtime error occurs.

**ErrorAction := 1**

Note: ErrorAction defaults to 1 when the controller is reset, but the default startup procedure turns this safety feature off if the program mode switch is on. *Meaningful error action values may be expanded beyond 0 and 1 in future releases.*

## Free

**Free** ⇔ Int

Returns the amount of free RAM in the controller (in the system heap, actually). It will also report on other areas of the controller's memory.

Free (0)⇒ Heap memory free

Free (1)⇒ Largest free block in the heap

Free (2)⇒ Total NV RAM8 size

Free (3)⇒ NV RAM free

The size of the NV RAM area may be set using

**Free (2) := SIZE**

If you change this size Venom has to reset itself, as it is a major upset internally. For the security of the heap memory, all applications should include a line early on in the *init* procedure to set the NV RAM size to zero, or another constant value large enough to supply all the non-volatile RAM needed by your application.

## Output

**Output** ⇔ Any

*Output* is the current 'default output stream'.

Setting *Output* redirects the text from PRINT, LIST and HELP.

`Output := new_output_device`

If you redirect the output to *NIL* then the output is simply discarded:

`Output := NIL`

Reading *Output* returns the current default output stream object.

*Later, the redirection of system output – e.g. the prompt and error reports – will be possible too.*

## Protect

**Protect ( Int flags )**

The protect controls the 'ROMing' of Venom application code. It is used to create a protected application in the Flash memory. It is also used to clear out an application from flash memory. The table details the function of the Protect message.

`Protect (0)` ⇒ Erases the Application area of the Flash

`Protect (1)` ⇒ Programs the application code into the Flash

`Protect (2)` ⇒ Report the application area statistics

`Protect (4)` ⇒ Returns the device ID code, or 'Silicon Signature' of the flash device.

The Protect message returns the 'Silicon Signature' of the flash if a valid, write enabled device was found, zero otherwise. This device ID code is best printed out as a 4-digit hex number:

```
-->PRINT ~protect(4):-4  
01A4-->
```

You may continue to compile procedures into RAM after putting code into flash – the system will take care of using the latest version of each procedure. You may call `Protect(1)` repeatedly without erasing procedures in between, until the application area is full.

*Currently, any newly created procedures in RAM will be removed at startup if a ROMed application is present.*

The full ROMing functionality is still under development, but will be at least as flexible as described above. There will be at least 128K, probably 192K of application area available. Venom-SC is very efficient with application memory, so this should be enough for very large applications.

## **Reset**

### **Reset**

Resets the controller. If it is in run mode then the application will run, else you will get the startup banner.

## **Run**

### **Run**

Runs the application as *if* in Run Mode, even if the program mode switch is on.

↔

## **RunMode**

### **RunMode**

Returns TRUE if the system is in Run Mode, i.e. if the program mode switch is off, or if you typed 'Run' at the command line.

## **UserSwitch**

### **UserSwitch (int switch) ↔Int**

Reports the state of the two switches on the controller. It takes a parameter that specifies which switch: 1 for the program mode switch (labelled RUN - PRG), and 2 for the user switch (labelled 9K6 - 38K).

## **Valid**

### **Valid ↔Int**

When a new version of Venom-SC is released, a checksum is taken over all the bytes of code in the Language, Operating system and Objects. This checksum value is written into the code, enabling you to validate your copy of Venom-SC. Valid will return TRUE (-1) if the checksum is good, FALSE (0) otherwise. Of course, if your Venom-SC code is highly damaged then you won't be able to call Valid.

PRINT Valid  
-1-->

## PRINT

### PRINT system

Printing the system object gives the size of the symbol table and global area, and the amount of the heap memory free. Other general system information may be added from time to time.

```
-->print system  
Symbol table 55 bytes  
8 Global variables  
99814 Heap bytes free (biggest block 99700)  
NV RAM area 0 bytes (0 unused)  
-->
```

## PULSE COUNTER

PulseCounter is used to count pulses on one of the 'clock input' channels.

### Summary of messages

MAKE  
Count  
Reset  
PRINT

### Creation

#### MAKE obj PulseCounter (Int chan, Int edge, Int chanx)

A new PulseCounter object is created with a zero pulse count. **chan** is one of the clock input channels, and **edge** is one, two or three. The optional parameter **chanx** is explained below.

### Relating Clock and Pulse I/O

Each *PulseCounter* object uses a counter/timer register from the processor's internal *TPU* module to hold the count. This means that a pulse I/O object such as *PulseWidthOut* or *PulseWidthIn* can't use the register.

The optional parameter **chanx** allows you to *choose* which of the pulse I/O channels you wish to become unavailable. If you don't include **chanx**, a default channel will be used:

Pulse counter input channel	Default channel used
1	18
2	6
16	3
17	4

### Examples

```
MAKE p_in1 PulseCounter (1,1) ;channel 1. Rising edge.  
MAKE p_in2 PulseCounter (1,2,19) ;chan 1 Falling edge.
```

In the first example the pulse count input is on channel 1, and the associated default channel, 18, becomes unavailable. In the second example we chose to make channel 19 unavailable instead.

▼ The maximum number of PulseCounter objects is 4, limited by the number of clock input channels available on the VM-1.

If the input clock pulse width is less than approximately 0.1  $\mu$ s for single-edge detection or 0.16  $\mu$ s for 2-edge detection, pulses may be missed.

### Count

**obj . Count**  $\Leftrightarrow$  Int

Returns the current pulse count.

```
-->PRINT p . Count  
3245-->
```

▼ The PulseCounter object can keep track of over 2 billion pulses.

## Reset

### obj . Reset

Resets the pulse count to zero. For example:

```
-->PRINT p . Count,CR
3245
-->p . Reset
-->PRINT p . Count,CR
0
-->
```

## Printing

### PRINT obj

Prints the current pulse count in square brackets:

```
-->PRINT p
[PulseCounter: 10]
```

## PULSE WIDTH IN

PulseWidthIn is used to measure the width or period of incoming pulses on one of VM-1 channels 3, 4, 5, 6, 18 or 19. Pulses or periods from 8 $\mu$ s to over 35 minutes can be measured with a resolution of 1 $\mu$ s.

## Summary of messages

```
MAKE
Go
Done
Period
PRINT
```

## Creation

### MAKE obj PulseWidthin (Int chan, Int mode )

A new PulseWidthin object is created. **chan** must be one of: 3, 4, 5, 6, 18 or 19. **mode** has the following meanings:

- 0 Measure low pulse width: falling to rising edge
- 1 Measure high pulse width: rising to falling edge
- 2 Measure period between falling edges
- 3 Measure period between rising edges

## Examples:

```
MAKE pw1 PulseWidthIn (3, 1) ;VM-1 channel 3 High pulse width  
MAKE pw2 PulseWidthIn (18, 3) ;chn 18 Period between falling edges
```

## Go

### obj . Go

Starts a measurement cycle. The time measurement will start at the next leading edge as specified by mode when the object was created, and the measurement is complete as soon as the correct type of trailing edge is encountered.

## Done

### obj . Done ⇔Flag

Returns True (-1) when the measurement cycle is complete, i.e. both the leading and trailing edge have been seen, or 0 before this condition is met.

## Period

### obj . Period ⇔Int

Returns the measured period in microseconds. Periods below 8 $\mu$ s are not measured correctly because of interrupt response limitations, and the maximum useful value that can be returned is 214783647 (max value for a signed 32 bit integer)

The Period message can be used in two ways:

1. By itself, the Period message will initiate a measurement cycle and wait for the result before returning. The task is suspended while waiting.
2. In conjunction with the Go and Done messages, the task can loop and execute other code while waiting. When the Done message returns true, the next period message will return immediately with the value just measured. If a period message is sent any time after a Go message and before the measurement cycle is complete, the task will wait.

```
; example of 1st usage
width := pw.Period ; task is suspended while waiting
; example of 2nd usage
pw.Go
while NOT pw.Done
[ ; other code in loop executed while waiting
]
width := pw.Period
```

## PRINT

### obj . Print

Prints, inside square brackets, the text "PulseWidthIn : " followed by the last measured value, followed by a new line. No measurement cycle is initiated. If no measurement has been made since the object was created the value printed is -1. The Print message is not recommended for general programming with PulseWidthIn objects.

Example:

```
-->print pw
[PulseWidthIn: 456]
-->
```

## PULSE WIDTH OUT

PulseWidthOut generates pulse trains with variable Mark/Space ratio and frequency. A PulseWidthOut object may be set to generate a continuous signal, or a pulse train containing a specific number of pulses.

Care should be taken to ensure that the creation of a PulseWidthOut object does not use the same internal counter register as a PulseCounter or Shaft object. Normally the PulseWidthOut Width and Period values are given in units of 1 $\mu$ S however this may be altered using the *Format* message. Timing accuracy is governed by the VM-1 crystal oscillator.

## Summary of messages

MAKE  
Asserted  
Count  
Format  
NotAsserted  
Off  
On, Go  
Period  
Queue  
Width  
PRINT

## Creation

**MAKE obj PulseWidthOut ( Int channel, Int period, Int width, Int polarity, Int count)**

A new PulseWidthOut Object is created. Note that the pulse train will not actually start until the Go or On message is sent.

The table below describes each of the parameters.

Parameter	Range of values	purpose
Channel	3,4,5,6,18,19	The VM-1 channel to use
Period	2-65536	The period of the output signal
Width	0-(Period-1)	The width of the output signal
Polarity	0 or 1	0: Active low; 1: Active high
Count	0-2,147,483,647	Optional. The number of pulses to generate; 0 means continuous signal.

In general channels 5 and 19 are the best to use for PulseWidthOut, especially if high frequency pulses are required.

## Example creation code

```
;Create a 1:10 mark:space PWM waveform  
MAKE pwm pulsewidthout(6,10000,1000,0)  
pwm . On ;turn it on
```

The VM-1 channels may be used by other objects.

▼ The maximum number of PulseWidthOuts is 6 (one per available channel). The maximum value of period is 65536. The minimum value of period is 2, but also see warning below.

*Warning: When generating a defined-length pulse train using **count**, PulseWidthOut uses interrupts to count the pulses. If the period becomes too small the VM-1 may start to behave erratically. The exact threshold for this depends on interrupt loading. Keep the period above 100uS when using **count** until this has been characterised.*

*A similar restriction applies to all channels other than 5 & 19. When generating high frequency continuous pulse trains, setting the period and width values may result in an output that misses a state-change. Again, keeping the period above 100uS until this has been fully characterised will be well into the safe region.*

## **Asserted**

**obj . Asserted** ⇔ *Flag*

Setting Asserted TRUE starts a pulse train (like On), and setting it FALSE stops the train (like Off). Asserted returns TRUE if the pulse train is currently active, FALSE otherwise.

## **Count**

**obj . Count** ⇔ *Int*

Count allows the pulse train length to be read or set.

Setting the Count takes effect the next time the train is triggered (using Go or On). It has no effect on a currently active pulse train.

Reading the Count returns the number last set during creation or using Count.

## **NotAsserted**

**obj . NotAsserted** ⇔ *Flag*

Setting NotAsserted TRUE stops a pulse train (like Off), and setting it FALSE starts the train (like On). NotAsserted returns FALSE if the pulse train is currently active, TRUE otherwise.

## Format

**obj . Format := Int**

Setting format allows the timing, or even the source, of the clock pulses to be changed in the PulseWidthOut object. The value passed to the format message consists of a bit field. The lower three bits

(bits 0 – 2) determine the source of the clock pulses – one of several divisions of the processor clock crystal or even a VM-1 channel. The table shows the clock rate or VM-1 channel selected depending on the channel and bits 0 – 2. The clock rate is shown as  $\emptyset/N$  where  $\emptyset$  is the VM-1 clock crystal frequency (normally 16Mhz) and N is a division ratio. The column in bold is the default setting.

VM-1 channel/ Bits 0-2	000	001	010	011	100	101	110	111
3	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	17	$\emptyset/256$	16
4	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	17	$\emptyset/1024$	-
5	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	$\emptyset/1024$	$\emptyset/256$	$\emptyset/4096$
6	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	1	$\emptyset/256$	-
18	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	1	17	$\emptyset/1024$
19	$\emptyset$	$\emptyset/4$	$\emptyset/16$	$\emptyset/64$	2	1	17	16

The next two bits (bits 3 & 4) determine which edges to count on.

Bits 4,3	count on edge
00	Rising
01	Falling
10	Both
11	Both

Examples:

p.Format := %00011 ; use units of 4uS.

p.Format := %01011 ; VM-1 channel 2 falling edge is the clock.

## On, Go

**obj . Go**

**obj . On**

On starts a pulse train. On is necessary to start the train after the PulseWidthOut has been created. Every time the message On is given, the number of pulses to be sent out is reset to the value of Count. If this was zero, then the pulse train keeps going indefinitely. Go has exactly the same effect as On.

## Off

**obj . Off**

Off turns off the pulse output at the next edge.

## Period

**obj . Period**  $\Leftrightarrow$ Int

An active variable that allows the *overall* period of the waveform to be read or set. The units are determined by the clock source, but the default is  $\mu$ S. The minimum Period value is 2, but see the warning notice in *Creation*. If Period is set to less than the current Width then the output will go to the 100% duty state. As soon as Period is greater than Width again, pulses will reappear. If you have Count set when Period is less than Width, counting of pulses will continue even if no real pulses are generated. If you call both Width and Period, it is possible under some circumstances for one of these messages to have to wait for a maximum time given by *Period*. However, if you stick to a constant Period and vary Width, or vice versa, this will never occur.

## Queue

**obj . Queue**  $\Leftrightarrow$ Int

no count has been set it returns 0.

## Width

**obj . Width**  $\Leftrightarrow$ Int

Allows the *mark* period of the waveform to be read or set. The units are determined by the clock source, but the default is  $\mu$ S.

If Width is set to zero or less, then the output will go to the 0% duty state, i.e. pulses will not be generated. If Width is set to more than the current Period then the output will go to the 100% duty state so the output is turned on continuously, no pulses will be generated. As soon as Width is less than Period again, pulses will reappear.

If you have Count set when Width is zero, or Width is greater than Period, the count will continue, even if you cannot see the pulses.

If you change both Width and Period, it is possible under some circumstances for one of these messages to have to wait for a maximum time given by *Period* (while swapping tasks). However if you stick to a constant Period and vary Width, or vice versa, this will never occur.

## Printing

### PRINT obj

Prints the current width and period in square brackets.

`PRINT pwm`

`[PulseWidthOut: 1000/10000]`

## Die

Die turns off the pulse train immediately and removes the object.

## SERIALIO

SerialIO allows communication with a host of I/O ICs that communicate on the 'Microwire' and 'Serial Peripheral Interface' (SPI) busses. Because there are so many different devices available, and each has a slightly different protocol, the SerialIO object provides a general interface. This is analogous to the low level I2C Net commands, e.g. `I2CBus.Put()`. Each SerialIO object controls four VM-1 channels. These are connected to the 'Chip select', 'Serial Clock', 'Data in' and 'Data out' lines of the device(s). It is possible to multiplex the Chip Select signal to allow more devices to be put on the bus. These devices would not be fully independent as they share the Serial-Clock, Data-in and Data-out signals. Thus if you want to use them in different tasks you will need to use the locking provided.

The connections from the VM-1 to a device are as follows.

<i>VM-1 FUNCTION</i>	<i>CONNECT TO</i>
<i>μwire Din</i>	<i>Device's data out*</i>
<i>μwire Dout</i>	<i>Device's data in</i>
<i>μwire clock</i>	<i>Device's serial clock</i>
<i>μwire Chip select</i>	<i>Device's chip select</i>

*\*Not necessary if no data comes back.*

## Summary of messages

MAKE  
On  
Put  
Off

### Creation

#### MAKE s SerialIO (Int bits)

Where bits is the number of bits to be clocked in or out with each access. The maximum number of bits is 32. If your application requires several different word lengths, then you can define an object to handle each length. As well as returning the object, this command will initialise the VM-1 channels to be used. For example we might want 12 bits sent.

MAKE s SerialIO(12)

### On

#### obj . On

This pulls the Chip Select signal low to start a communication.

### Put

#### obj . Put(Int n) ⇔ Int

Data is sent and received with the same command. This is because some devices receive input data at the same time as sending output data. The message to use is:

s.Put(n)

This will clock the value of n out as 12 bits. It will also clock in any data output by the device. This is returned by Put and may be used in any way, for example:

value := s.Put(27)

Data is clocked in and out MSB first.

### Off

#### obj . Off

This pulls the Chip Select signal high to end a communication.

## SHAFT

Shaft monitors quadrature-phase inputs to keep track of the position of rotary (shaft) encoders. Up to 2 shaft encoders producing edges with a phase difference and overlap of at least 0.1µS and a pulse width of at least 0.16µS may be monitored.

### Summary of messages

MAKE  
Count  
Reset  
PRINT

### Creation

#### MAKE obj Shaft (Int ch1, Int ch2 , Int chanx)

A new Shaft object is created, monitoring channels **ch1** and **ch2**. The initial count is zero. The channels ch1 & ch2 must be paired: 1 with 2 and 16 with 17.

*Note: if the count value counts the wrong way, then you will have to alter the hardware in order to swap which phase of the signal goes to which channel.*

### Relating Clock and Pulse I/O

Each *Shaft* object uses a counter/timer register from the processor's internal *TPU* module to hold the count. This means that a pulse I/O object such as *PulseWidthOut* or *PulseWidthIn* can't use the register.

```
MAKE encoder1 Shaft(16,17)
```

```
MAKE encoder2 Shaft(1,2,3)
```

The first example creates a Shaft object reading channels 16 and 17. Channel 18 becomes unavailable by default. The second example uses the optional parameter **chanx** to create a Shaft object reading channels 1 and 2. Channel 3 becomes unavailable.

### Count

**obj . Count** ⇔ Int

An active variable that holds the count of quadrature phase edges seen by the Shaft object. The count can be set or read.

▼ Count can hold values up to around  $\pm 2$  billion, but only values up to around 1 billion may be held by variables. If the phase difference and overlap between the two input clocks is less than approximately 0.1 ns, or the pulse width is less than 0.16 ns, the VM-1 will lose counts. If the phase is very near zero, then the count may go either up or down.

## **Reset**

### **obj . Reset**

Resets the count to zero.

## **Printing**

### **PRINT obj**

Prints the current value of the Count.