

The University of Michigan

## **Micro-Controller Interface Board (MCIB)**

With relevance to BAA 98-08: Tactical Mobile Robots

**Last Revised July 8, 1998**

by

**Johann Borenstein**

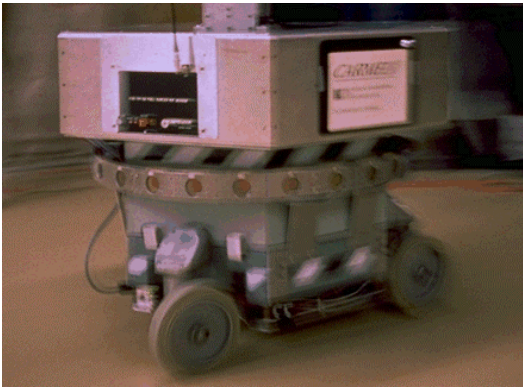
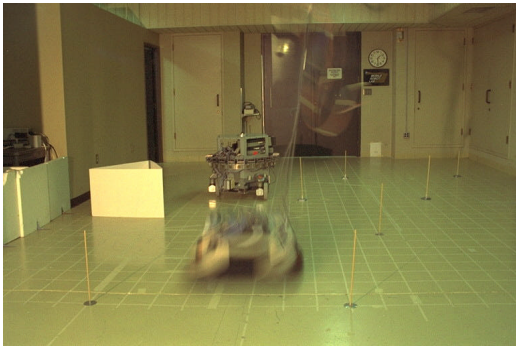



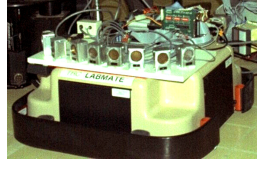
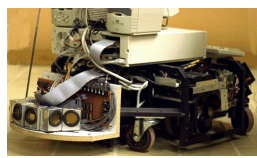



# TABLE OF CONTENTS

<b>INTRODUCTION .....</b>	<b>1</b>
<b>1. THE ELECTRONIC HARDWARE .....</b>	<b>3</b>
<b>1.1 System Overview .....</b>	<b>3</b>
<b>1.2 The Electronic Interface.....</b>	<b>4</b>
1.2.1 The MC68HC11 Micro-controller .....	4
1.2.2 The HC11 Master .....	6
1.2.3 The HC11 Slaves.....	8
1.2.4 The PC - HC11 Communication .....	9
1.2.5 The SPI Communication .....	12
1.2.6 The FIFO .....	13
1.2.7 The HCTL Quadrature Decoders .....	15
<b>2. THE MCIB SOFTWARE .....</b>	<b>16</b>
<b>2.1 The Master HC11 Software .....</b>	<b>16</b>
2.1.1 Task #1 - Communication and Program Execution.....	16
2.1.2 Task #2 - Generation of the PWM Signals.....	17
2.1.3 Task #3 - Continuous A/D Conversions.....	18
<b>2.2 The HC11 Slave EERUF Implementation .....</b>	<b>19</b>
2.2.1 The Multitasking Architecture.....	19
2.2.2 Task #1 - Communications and Treatment of Buffer.....	20
2.2.3 Task #2 - The Generation of the Fire Signals.....	20
2.2.4 Task #3 - Checking for Echoes .....	21
2.2.5 Task #4 - The Generation of the BINH Signals.....	21
2.2.6 The Fire Signal Table.....	21

# INTRODUCTION

The University of Michigan's (UM) Mobile Robotics Lab has specialized for over 11 years in the area of mobile robot obstacle avoidance. UM's obstacle avoidance systems have been implemented successfully on a large variety of mobile robots, on a power wheelchair, on a portable system for the blind, and on a novel, wheeled guidance device for the blind, called GuideCane. Common to all of these systems is the need for *fast* obstacle avoidance, on the order of 1 m/s or faster. UM has demonstrated the ability of its systems to function at such speeds, and we believe that UM's reflexive obstacle avoidance systems are the fastest of their kind (i.e., performing near-range, ultrasonic sensor-based, reflexive obstacle avoidance).

 <p>CARMEL II: 24 sonars, 0.8 m/s.</p>	 <p>Labmate (foreground) zaps through an obstacle course at 1 m/s, while CARMEL (background) "paints" the environment with ultrasonic noise.</p>
 <p>NavBelt: portable device for the blind, 8 sonars.</p>	 <p>GuideCane: 10 sonars, 0.7- 1.0 m/s</p>
 <p>NavChair: 12 sonars, 1.2 m/s.</p>	 <p>LabMate: 8 – 12 sonars, 1.0 m/s.</p>  <p>SWAMI Jr.: 8 sonars, narrow-isle navigation.</p>
 <p>OmniMate: 32 sonars, 0.5 m/s omnidirectional motion.</p>	

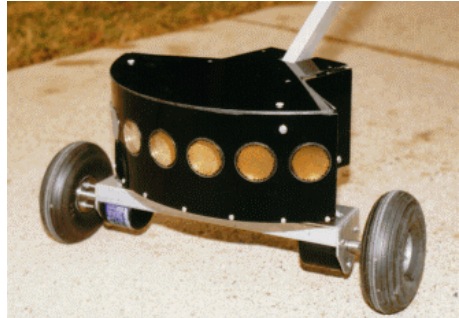

One key component of UM’s obstacle avoidance systems is a very sophisticated sonar firing algorithm, called *Error Eliminating Rapid Ultrasonic Firing* (EERUF). With EERUF individual sonars can detect and reject false readings caused by stray echoes from other sonars onboard (i.e., crosstalk) or off-board (i.e., from other mobile robots), or even from ultrasonic noise in the environment (i.e., gun fire in an urban combat scenario).

To avoid crosstalk, conventional methods mandate lengthy waiting periods in-between firing individual sonars, to allow the echo of each sonar to abate before the next sensor is fired. Conventional systems are also susceptible to ultrasonic noise from other sources (such as other mobile robots). With EERUF, on the other hand, about 97% of crosstalk and other ultrasonic noise is detected and rejected, allowing for substantially faster firing, and hence, faster travel.

For a recent development of UM’s Mobile Robotics Lab, the GuideCane<sup>1</sup> guidance device for the blind, a new “hardware-only” implementation of EERUF was developed. This work is of immediate applicability to the current DARPA BAA 98-08 because the basic functionality requirement – reflexive obstacle avoidance in indoor and outdoor environments at speeds above 1 m/s – is the same for both applications. The size-, weight-, and power-limitations are also essentially the same for both applications.

This document provides technical details on UM’s “hardware-only” implementation of the EERUF method. However, the EERUF implementation is only one part of UM’s so-called Micro-Controller Interface Board (MCIB), which also provides numerous other relevant low-level control functions.

**Table I:** Similarity between UM's GuideCane and the Pioneer AT platform. The scale is roughly the same for both pictures.

		
Size: 50-cm class Speed: > 1m/s Obstacle Avoidance: 10+ sonars Environment: Indoor/outdoor		Size: 50-cm class Speed: > 1m/s Obstacle Avoidance: 7+ sonars Environment: Indoor/outdoor

---

<sup>1</sup> The GuideCane recently won the 1998 Discover Magazine Award for Technological Innovation, in the robotics category.

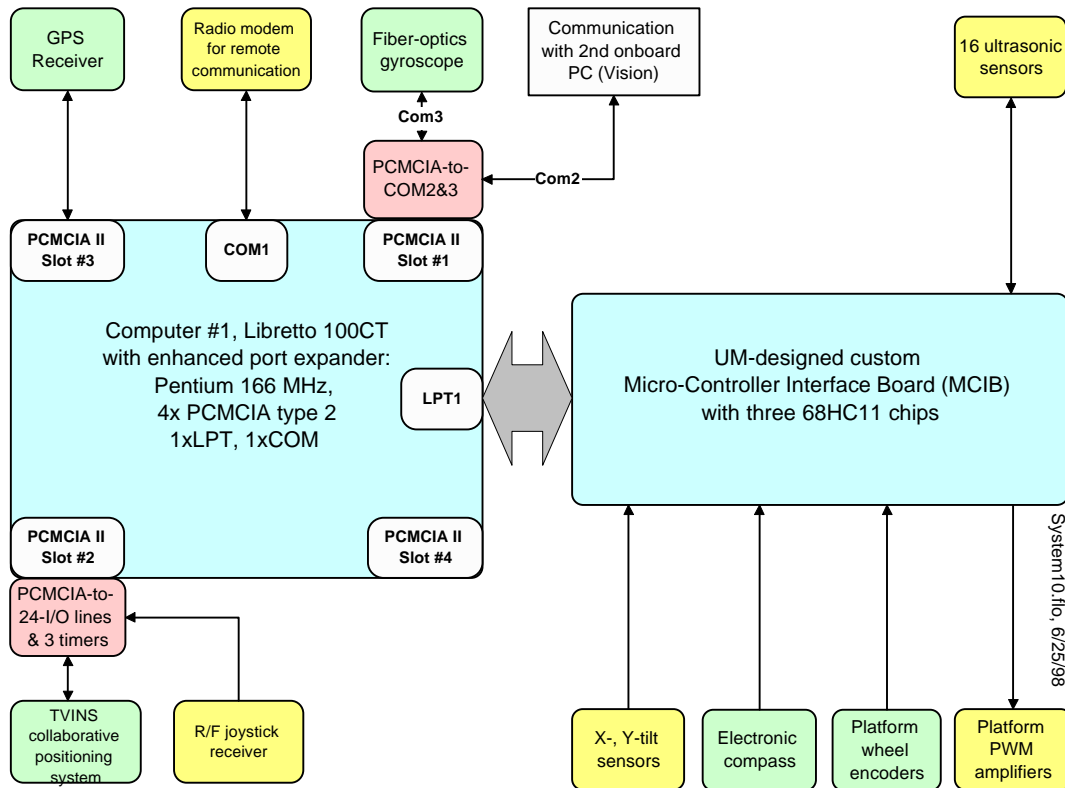
# THE ELECTRONIC HARDWARE

This section explains the hardware components of The University of Michigan’s (UM) *Micro-Controller Interface Board* (MCIB). To better understand the function of the device, we first present an overview which shows how the board is going to be used in UM’s Part A development effort under BAA 98-08. UM’s system, which is based on the RWI Pioneer AT mobile platform, is called “MOVERS.”

## 1.1 System Overview

MOVERS’ System architecture is shown in Figure 1. The main onboard computer is a Toshiba Libretto 100 CT mini-notebook. For simplicity we will refer to it as the “PC” in the remainder of this document. The PC, shown in Figure 2, is a full featured Windows 95/98 enabled laptop of diminutive size. It features a 166 MHz Pentium processor, 32 MB RAM, and a 2.1 GB harddisk. The foremost advantage of this type of PC over widely used embedded controllers of the equally diminutive PC 104 format family is the built-in 7.4” LCD active matrix color display and a keyboard. During development the availability of an onboard monitor and keyboard are invaluable.

The PC is connected to a University of Michigan (UM) custom-built *Micro-Controller*



**Figure 1:** System overview. The Micro-Controller Interface Board (MCIB) uses three 68HC11 chips to read-and pre-process most of the onboard navigation sensors.

*Interface Board* (MCIB) through its bi-directional parallel port. The MCIB serves as the interface between the PC and most of the sensors and actuators. The MCIB consists of three MC68HC11E2 micro-controllers, four counters, and some logic devices.

The MCIB executes many time-critical tasks, which would take too many resources if executed by the main PC. The MCIB receives commands from the PC and then takes care of the sensors and actuators without any further involvement of the PC. Most of the sensor data is preprocessed by the MCIB before being communicated to the PC. This approach minimizes the bandwidth requirements for the communication between the PC and the MCIB, and it reduces the computational power required by the PC to control the sensors and actuators.



**Figure 2:** The Toshiba Libretto 100 CT is a 2.2-pound miniature notebook computer that serves as the main onboard PC. It features a 166 MHz Pentium processor, 32 MB RAM, and a 2.1 GB harddisk.

## 1.2 The Electronic Interface

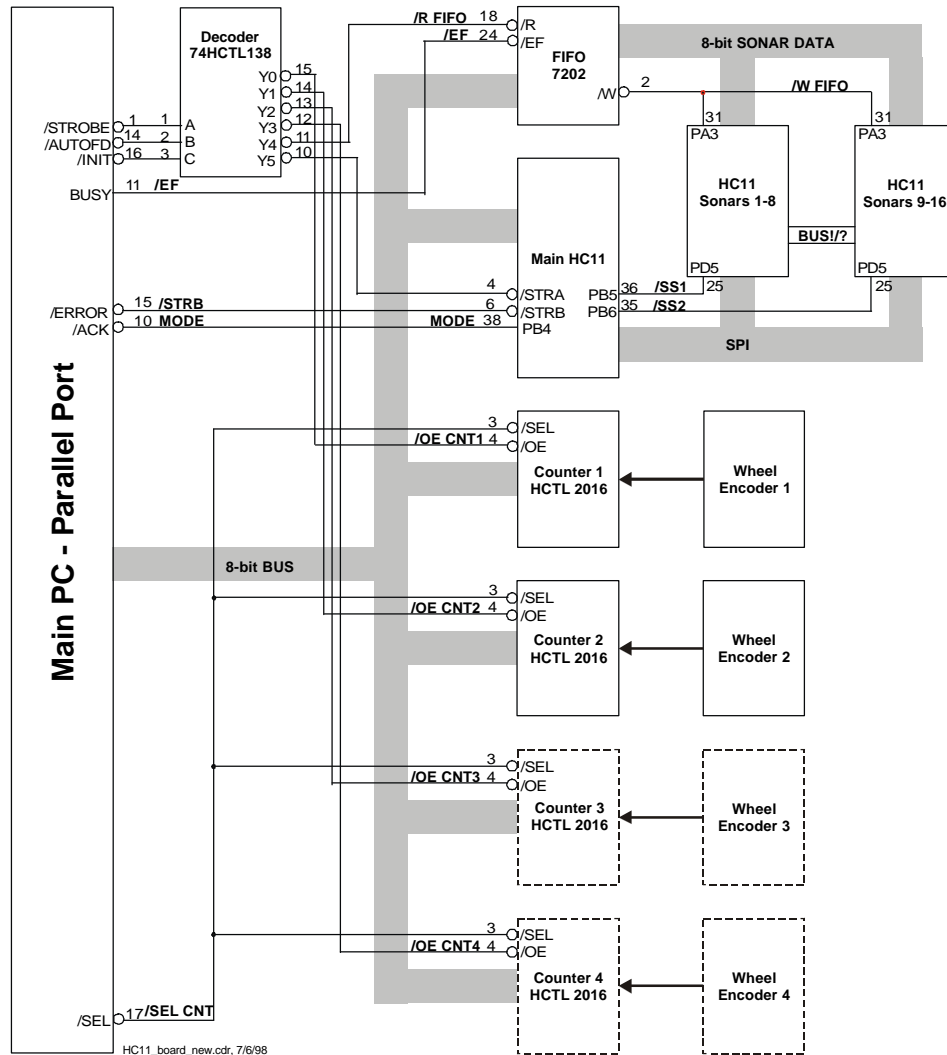
The architecture of the MCIB is shown in Figure 3. The communication between the PC and the MCIB is based on the PC's standard parallel port. The parallel port consists of an 8-bit bi-directional data bus, four digital outputs and five digital inputs. In the present design all but two digital inputs are in use. The 8-bit bi-directional data bus together with the four digital outputs and an additional 3-8 decoder allows the PC to communicate with the master micro-controller, the *First-In-First-Out* (FIFO) register, and the HCTL quadrature decoders. Two of the digital inputs are used for the handshaking with the HC11. Another digital input is used to supervise a FIFO flag. This bus architecture permits parallel communication at high speed. Furthermore, most communications are buffered to minimize delays. The (currently wire-wrapped) prototype of the MCIB is shown in Figure 4 and the new printed circuit board design in Figure 5.

### 1.2.1 The MC68HC11 Micro-controller

The master- and the two slave micro-controllers are identical Motorola MC68HC11s. The block-diagram of a typical MC68HC11<sup>2</sup> is shown in Figure 6.

---

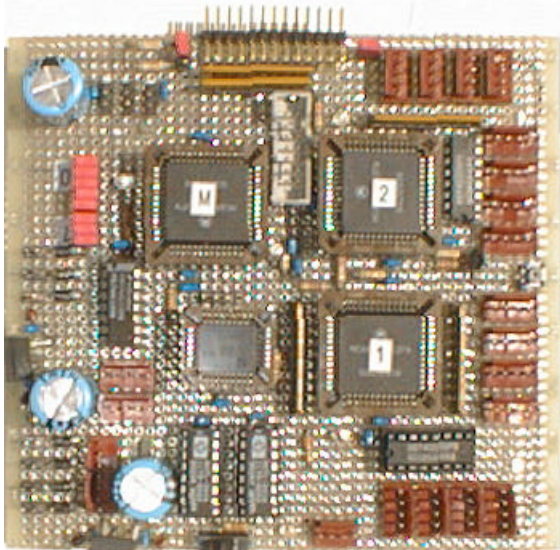
<sup>2</sup> Motorola, M68HC11 Reference Manual, 1991.



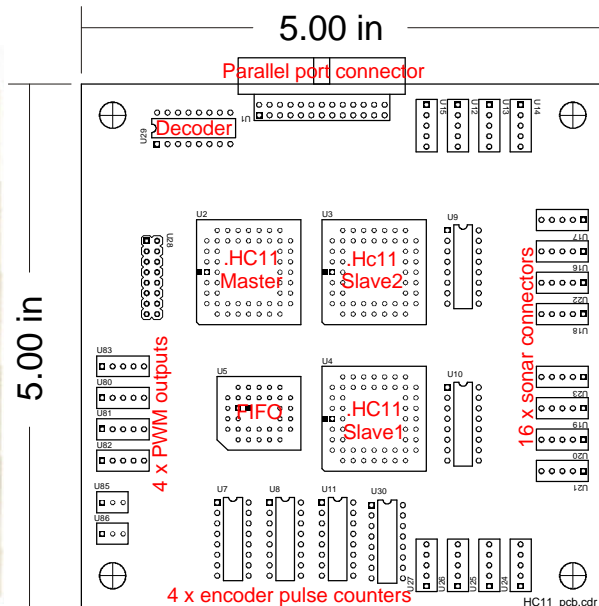
**Figure 3:** Schematic overview of UM's Micro-Controller Interface Board (MCIB). Components shown with solid lines were part of the earlier GuideCane prototype and are thus fully tested. Components shown with dotted lines will be added for slippage detection on the Pioneer AT.

The peripheral functions of the MC68HC11 include:

- Eight-channel A/D converter with an eight bit resolution.
- Asynchronous serial communications interface (SCI).
- Synchronous serial peripheral interface (SPI).
- 16-bit timer with three input-capture lines, five output-compare lines and a real-time interrupt function.
- An 8-bit pulse accumulator can count external events or measure external periods.
- Self-monitoring circuitry: watchdog, clock monitor system and illegal opcode detection.



**Figure 4:** The (currently wire-wrapped) prototype of the Micro-Controller Interface Board (MCIB) measures about 5"x5".



**Figure 5:** Printed circuit board layout of the Micro-Controller Interface Board (MCIB).

- Two software-controlled power-saving modes, WAIT and STOP, are available to conserve power.

The MC68HC11 name actually refers to a family of advanced 8-bit micro-controllers<sup>3</sup>. The difference between the MC68HC11 family members lies mainly in the amount and kind of memory. The version used here is the MC68HC11E2, which has 2 K of EEPROM (more than any other family member). For simplicity we will refer to this chip as the “HC11” in the remainder of this document.

Since the HC11 bus runs at only 2 MHz and EEPROM space is limited, it is preferable to program it in assembly instead of a higher level language. Fortunately, the HC11 has a quite comprehensive, orthogonal instruction set. As the software can be stored in the internal EEPROM, the micro-controller can be used in single-chip mode with no external memory.

### 1.2.2 The HC11 Master

A simplified diagram of the HC11 Master is shown in Figure 7. Port A is used to generate the PWM signals for up to four motor amplifiers. Three of the four unused pins on this port are input capture lines available for future extensions. The fourth currently not used pin is the pulse accumulator input.

Port B is used to independently reset the other logic devices on the MCIB, to select one of the two HC11 slaves, and for hand-shaking with the PC. Port C together with the *R/W* and *AS* lines is used to communicate with the PC over its bi-directional parallel port.

<sup>3</sup> Motorola, M68HC11 Reference Manual, 1991.



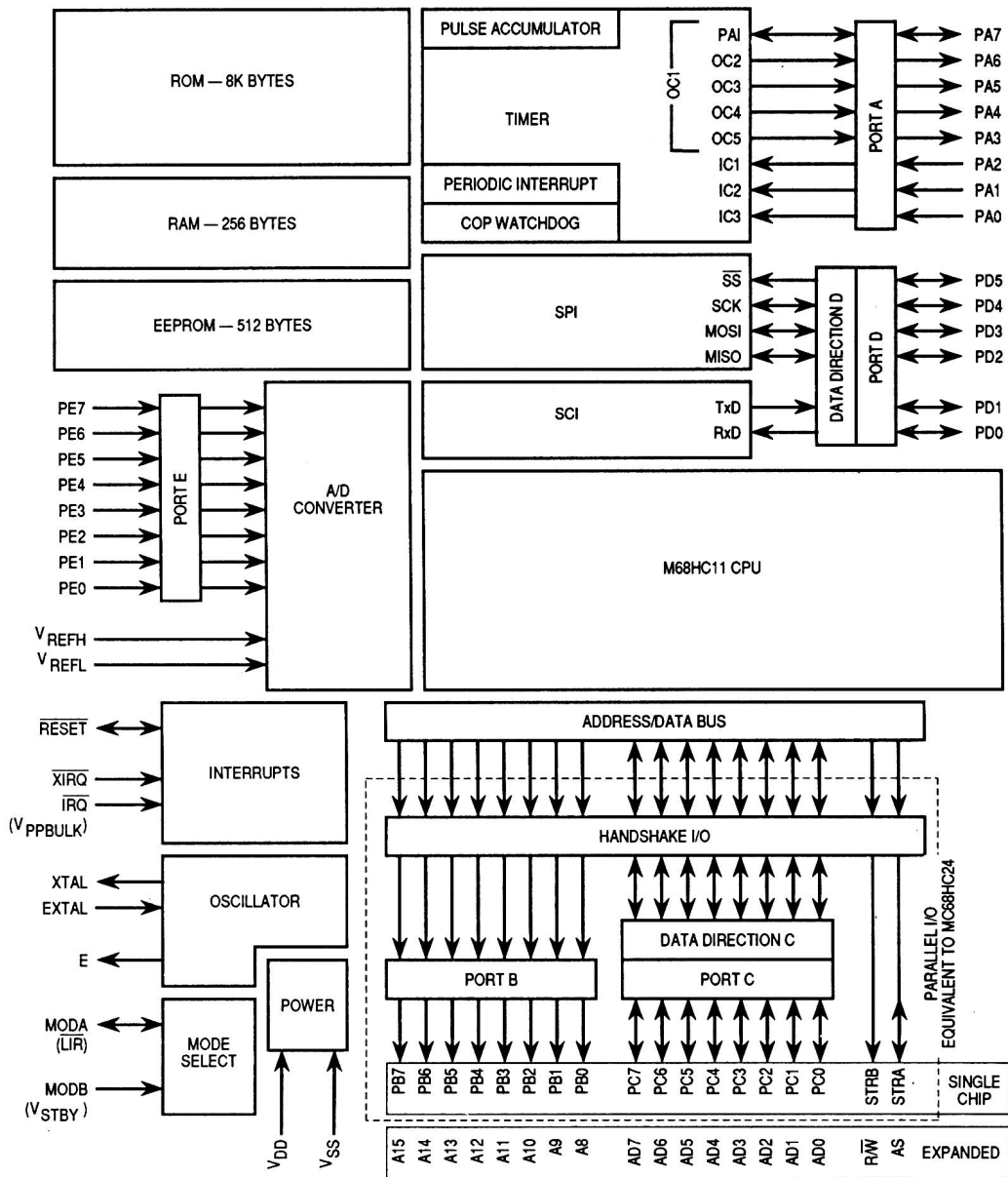


Figure 6: The MC68HC11A8 block diagram

The asynchronous serial communications interface (SCI) of Port D is used to download the program from the PC into the HC11's internal EEPROM. For future extensions, a standard serial port device can be connected to this port.

The synchronous *Serial Peripheral Interface* (SPI) is used to communicate with the two HC11 slaves (see Section 1.2.5, below). Additional SPI devices can also be added easily to the current architecture.

Port E is the eight-channel A/D converter with eight bits of resolution. In the present design one of the eight analog inputs will be used for the electronic compass and two analog inputs will be used for the tilt sensors.

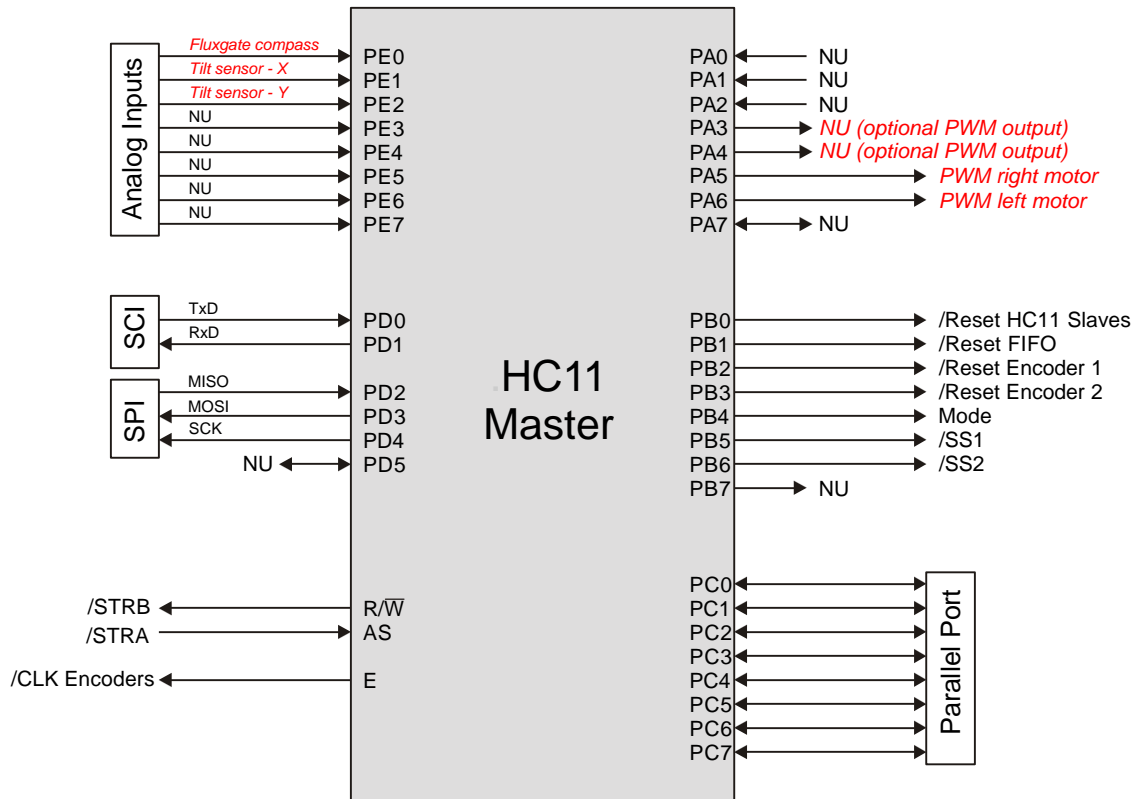


Figure 7: The HC11 Master

The *E* signal, a clock output running at 2 MHz, is used to drive the quadrature decoder devices (see Section 1.2.7, below).

### 1.2.3 The HC11 Slaves

A simplified diagram of an HC11 slave is shown in Figure 8. Both slaves are identically integrated into the hardware system. However, there are a few minor differences in their software. Because of the limited number of input and output lines and because of speed considerations, each HC11 slave operates only eight of the maximal possible 16 sonars.

Four outputs of Port A together with a double 2-4 decoder are used to generate the eight BINH signals for the sonars. Port B is used to generate the fire signals for the sonars. Port C together with *PA3* is used in an open-collector mode to write the sonar results into the FIFO. Port D is used the same way as for the master HC11. The *R/W* and *PA0* signals are used to synchronize the access to the FIFO bus.

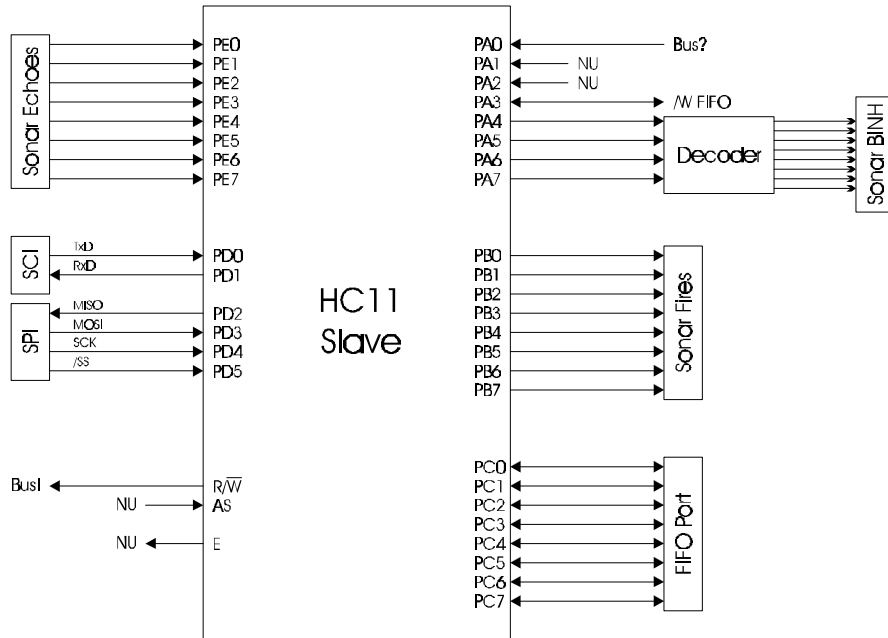


Figure 8: The HC11 Slave(s).

## 1.2.4 The PC - HC11 Communication

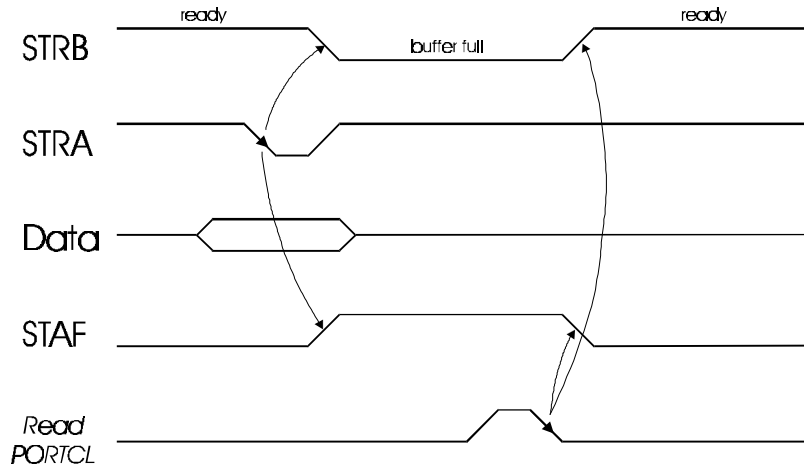
The communication between the PC and the HC11 micro-controllers is done through the master HC11 and the FIFO. The communication between the PC and the master HC11 is based on handshaking supported by the HC11 hardware. The procedure of writing a command (from the PC) to the master HC11 is explained in Section 1.2.4.1. The procedure of reading from the master HC11 is explained in Section 1.2.4.2.

The PC can not communicate directly with the two HC11 slaves. To send a command to the HC11 slaves, the PC sends the command to the master HC11 which will then send it to the HC11 slaves through the SPI as explained in Section 1.2.5. To read the sonar data from the HC11 slaves, the PC reads the buffered data from the FIFO as explained in Section 1.2.6.

### 1.2.4.1 The Write Command

The timing diagram for a write command is shown in Figure 9. The *STRB* signal is automatically generated by the HC11 hardware. The *STRA* signal and the data are controlled by the PC through its software. The *STAF* is an internal flag of the HC11. The *ReadPORTCL* represents the corresponding HC11 software command .

Normally, the master HC11 is in input mode waiting for a new command from the PC. The *STRB* signal indicates to the PC that the HC11 buffer is ready for a new command byte. It is important to note that even though the HC11 may be busy executing some task, the PC can latch a command byte into the HC11 buffer whenever the buffer is empty. This asynchronous communication allows the system to take full advantage of the multiprocessor architecture.



**Figure 9:** Timing diagram for the Write command

If the *STRB* signal is high, the PC can output the command data on the bus and latch it into the HC11 buffer by asserting the *STRA* signal (falling edge). This operation will automatically deassert the *STRB* signal indicating to the PC that the HC11 buffer is full. It will also assert the *STAF* flag, which tells the HC11 that there is new data in its buffer. When the HC11 detects that the *STAF* is asserted, it will read the data from the buffer. This action will deassert the *STAF* flag and automatically assert the *STRB* signal. The asserted *STRB* signal indicates to the PC that the HC11 is ready for another command. Meanwhile, the HC11 interprets the new command byte and executes the desired operation.

To minimize the communication delays, it is preferable to send only one byte for each command to the HC11. As long as only one byte is sent, the PC can simply latch it into the HC11 buffer without waiting for the HC11. If the command consisted of two bytes, two consecutive writings would be necessary, but these can not be handled by the HC11 hardware alone. In such a case, the PC could still latch the first byte into the HC11 buffer, but it would then have to wait for the HC11 to read this byte out of its buffer before sending the second byte.

In the current implementation, all commands are encoded in one byte as summarized in Table II.

#### 1.2.4.2 The Read Command

The timing diagram for a read command is shown in Figure 10. All signals have the same meaning as in the previous section. In addition, the *Mode* signal indicates to the PC if the master HC11 is in the input or output mode, and the *WritePORTCL* represents the corresponding HC11 software command.

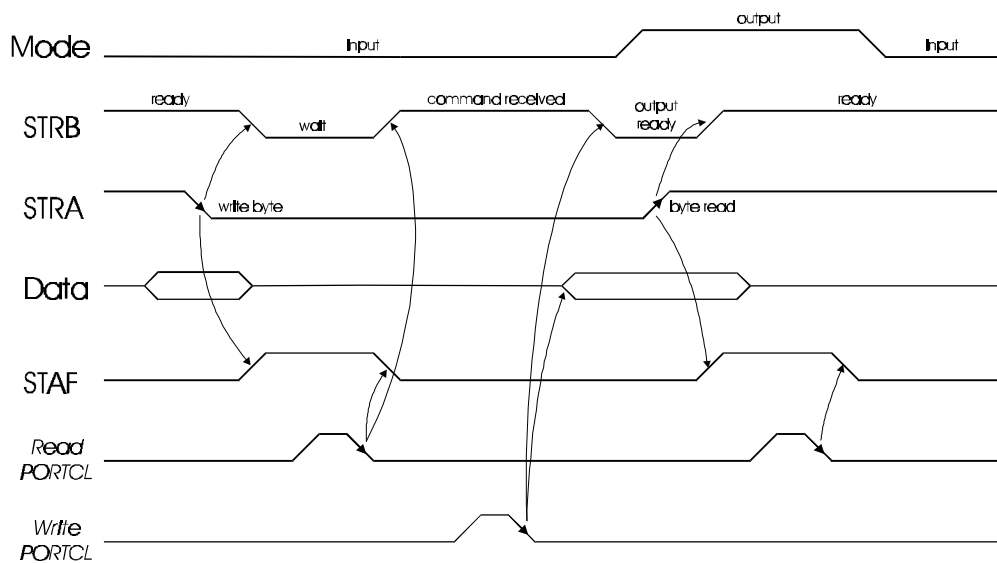
To read a byte from the HC11, the PC first has to send a command to the HC11 indicating the desired information. Therefore, the entire communication consists of a write operation followed by a read operation. The write operation is identical to the process described in the previous section. However, once the command byte is interpreted, the HC11 retrieves the desired information and puts it into an internal register. This automatically changes the HC11 port to a readable output mode and deasserts *STRB*. To

indicate to the PC that the data is ready, the *Mode* signal is set high. The PC then reads the data and asserts *STRA* indicating to the HC11 that it received the data. The *STRA* signal also automatically resets the HC11 output port into an input port to liberate the data bus for other communications. The HC11 then clears the *STAF* flag by reading its buffer, and deasserts the *Mode* signal to indicate to the PC that it is ready for a new command.

This mode is currently not used. For future extensions, this mode will be useful to read additional sensors connected to the master HC11 analog port. It will also be useful to read data from devices that can be connected to the master HC11's SPI or SCI port.

Command Byte	Command
1xxx'xxxx	Position of main servo (128 positions)
0000'xxxx	Activate resets on Port B (PB0-PB3)
0001'xxxx	EERUF modes (1-15) and stop sonars (0)
0010'xxxx	Fire single sonar (index 0-15)
0011'xxxx	Not used yet (16 values)
0100'xxxx	Right brake command (16 positions)
0101'xxxx	Left brake command (16 positions)
0110'xxxx	Main servo speed (16 values)
0111'0000	Read potentiometer of steering axis
0111'xxxx	Not used yet (15 values): xxxx ≠ 0

**Table II:** Command bytes that can be sent from the PC to the master HC11.

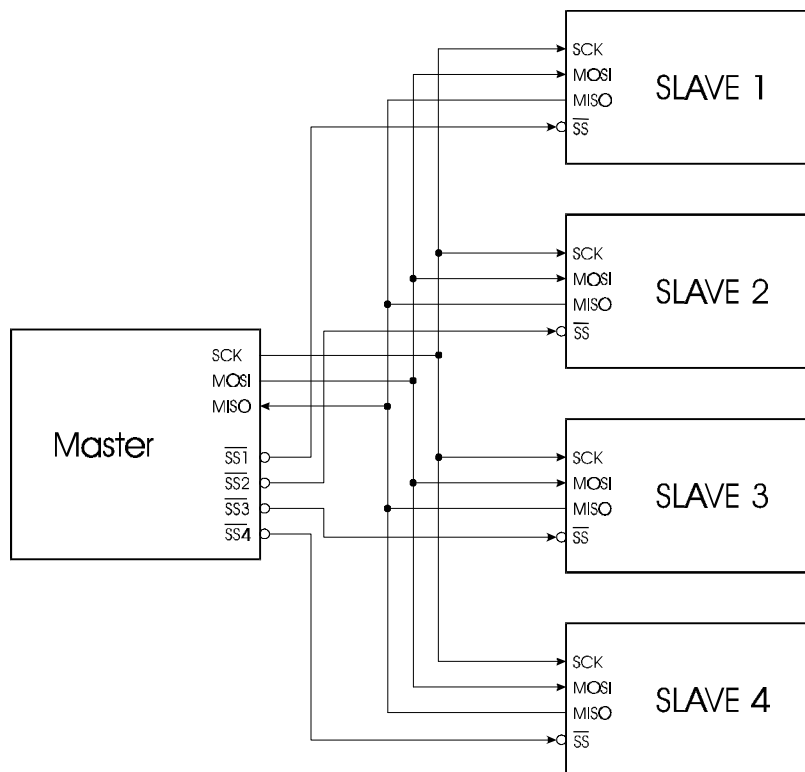


**Figure 10:** Timing diagram for the Read command

## 1.2.5 The SPI Communication

Both HC11 slaves are connected to the master HC11 through the synchronous *Serial Peripheral Interface* (SPI). The SPI interface is used primarily to allow the micro-controller to communicate with peripheral devices. Peripheral devices range from simple shift registers to complete subsystems, such as an A/D converter or another HC11. The SPI system is flexible enough to interface with numerous standard peripherals from several manufacturers. Data rates as high as 1 Mbit/sec are accommodated with one of the HC11 as the SPI master.

An example with four SPI devices connected to the master HC11 is shown in Figure 11. In the current interface architecture, there are only two SPI slaves. The master HC11 acts as the SPI master while the two HC11 slaves act as SPI slaves. This architecture allows one to easily add more SPI devices for future extensions.



**Figure 11:** Implementation example for the synchronous *Serial Peripheral Interface* (SPI) architecture, here shown with four slaves (the MCIB uses only two slaves.)

During an SPI transfer, an 8-bit character is shifted out one data pin while an 8-bit character is simultaneously shifted in a second data pin. So, one byte is simultaneously transmitted and received. The serial clock line (*SCK*) synchronizes shifting and sampling of the information on the two serial data lines (*MOSI* and *MISO*). The slave select line (*SS*) allows individual selection of a SPI slave.

The use of the SPI is extremely simple as it is fully supported by the HC11 hardware. All SPI transfers are started and controlled by a master SPI device, in this case the master HC11. To transfer a byte to an HC11 slave, the master HC11 simply asserts the

corresponding slave select signal and writes the byte into its SPDR register. The hardware of the two involved HC11s then takes care of the communication. At the end of the communication, the bytes that were originally stored in the two SPDR registers are swapped. To indicate the end of the SPI communication, the *SPIF* (SPI Transfer Complete Flag) is asserted.

## 1.2.6 The FIFO

The interface is equipped with a FIFO whose purpose is to buffer the outputs of the two HC11 slaves. The currently used FIFO is a high-density first-in first-out buffer with a depth of 1024 bytes. This buffer allows the HC11 slaves to write their data into the FIFO whenever they have a sonar reading as explained in Section 1.2.6.1. It also allows the PC to read the sonar data from the FIFO whenever the PC wants to, as explained in Section 1.2.6.2. Hence, due to the FIFO, the PC can asynchronously read the data from the HC11 slaves. This buffered communication allows the architecture to take full advantage of its distributed computing power.

### 1.2.6.1 The HC11 - FIFO Communication

The two HC11 slaves share an output bus and the */W FIFO* output to write the sonar data into the FIFO. To avoid access conflicts, the HC11 slaves use their *Bus?* and *Bus!* lines. The *Bus!* output of each slave is connected to the *Bus?* input of the other slave. Asserting the *Bus!* output means that the slave is in control of the bus or that it desires to be in control. A problem only occurs when both slaves try to take control of the bus simultaneously. To resolve this conflict, the first slave is given priority. The algorithms for the two slaves are shown in Figure 12. The precedence of the first slave over the second one is realized by the different reaction to the case when the result of the second *Bus?* inquiry is positive.

If this method failed for whatever reason, one or both slaves could become damaged by short-circuiting each other. To eliminate this risk, however unlikely it is, the nine shared output lines are used in an open-collector mode with pull-up resistors. Therefore, if both slaves put their data on the shared bus no damage would

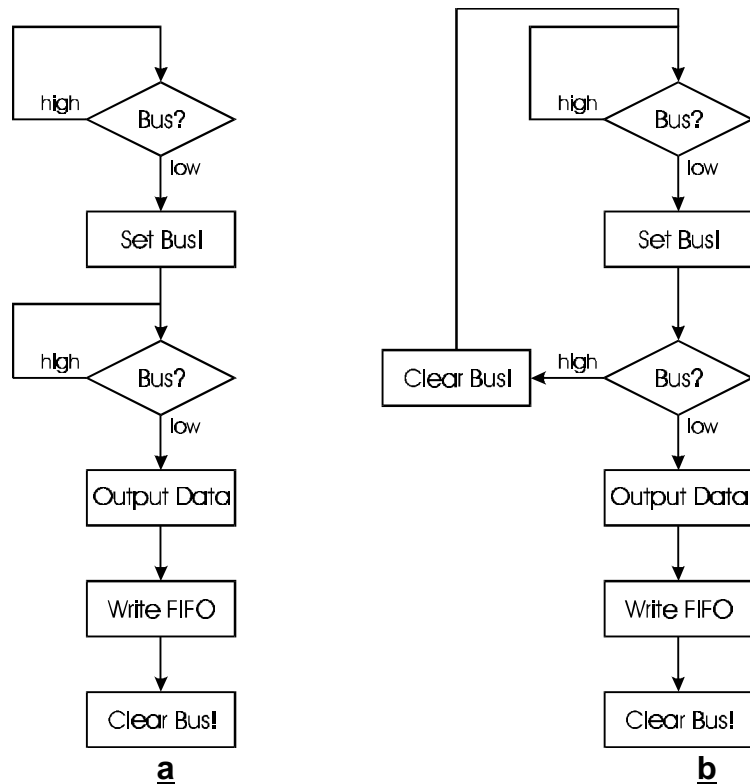


Figure 12: FIFO bus control for: a) slave 1 and b) slave 2

occur. The only effect would be that wrong data would be written into the FIFO. However, in most cases, this would be detected by the PC reading the wrong data from the FIFO.

### 1.2.6.2 The PC - FIFO Communication

The PC can easily determine if there is any data in the FIFO by checking the FIFO's "empty flag" (*EF*). If the *EF* flag is deasserted, the PC knows that there is new sonar data in the FIFO. The PC can then read out all sonar data by continuing reading until the *EF* flag asserts again. This method is very efficient, as the PC only has to check one input line to determine if new sonar data is available.

The PC could also be connected to the "half full" (*HF*) and "full full" (*FF*) flags of the FIFO. These flags are currently not used to save some of the parallel port input lines for future extensions. The *HF* could be used to tell the PC that it should read out the FIFO data. If the *FF* flag was asserted, it would indicate that probably a data overflow occurred because the PC waited too long before reading the FIFO. The PC would then have to send a reset command to the master HC11 to reset the two slaves and the FIFO. Next, the PC would have to send the command to start the sonars again. Therefore, if for whatever reason the FIFO filled up, the PC would instantly be aware of the problem and could go on without any interaction by the user.

A potential problem of the parallel port is "ringing." Misreadings due to ringing occur if the FIFO lines are directly connected to the PC parallel port. To eliminate ringing, each of the eight data lines is terminated by a resistor/capacitor network.

### 1.2.6.3 FIFO Data Encoding

Since both operations, writing to- and reading from the FIFO, are very fast, there is no need to encode the sonar data in as few bytes as possible. For each sonar reading, five bytes are transmitted:

1. Start byte, currently \$60<sup>4</sup>.
2. Sonar index between 1 and 16.
3. High byte of 16 bit time of flight in increments of 8  $\mu$ s.
4. Low byte of 16 bit time of flight in increments of 8  $\mu$ s.
5. Stop byte, currently \$80.

The start and stop bytes are not necessary, but they increase the probability of detecting a communication problem. Even without these two bytes, the PC could detect a problem by verifying that the sonar index is between 1 and 16. Since the extra time required by the start and stop bytes is negligible, these two bytes are kept for safety reasons.

---

<sup>4</sup> The '\$' symbol indicates that the following number is in hexadecimal format.

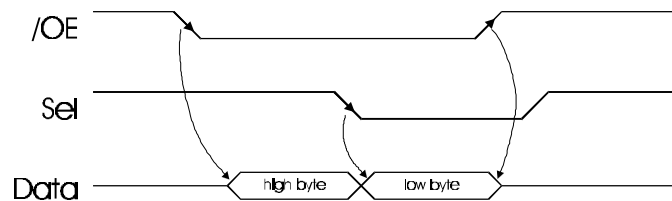


## 1.2.7 The HCTL Quadrature Decoders

The output of each encoder consists of two quadrature signals. By quadrature decoding these two signals, the encoder resolution is multiplied by a factor of four as each edge, rising and falling, is taken into account. However, quadrature decoding is a time-intensive task with restricting requirements.

A very effective and simple solution is the use of the *HCTL quadrature decoder interface* integrated circuits from Hewlett Packard. The *HCTL-2016* features full quadrature encoding with a 16-bit up/down counter, latched outputs, and high noise immunity due to Schmitt trigger inputs and digital noise filters. Moreover, due to the 8-bit tristate output, the two quadrature encoders can simply be connected to the 8-bit PC-interface bus.

The quadrature decoders require a clock signal. Rather than adding some clock circuitry, the *E* output of the master HC11 is used instead. The *E* output is a clock signal with a frequency of 2 MHz. To read the 16-bit content of the counter, the low and high bytes can be accessed independently by the use of the *SEL* signal as shown in Figure 13.



**Figure 13:** Timing of quadrature decoder reading

## 2. THE MCIB SOFTWARE

Although we describe the MCIB as a “hardware only” implementation (because it can be used as such by third party developers, the HC11s micro-controllers require, of course, some operating software. Software for the master and slave units is discussed in this section.

### 2.1 The Master HC11 Software

The master HC11 takes care of most of the communications, samples up to eight analog inputs, resets other devices, and generates the PWM signals for the motors. Since these tasks have different time constraints, the software is organized in a multi-tasking architecture. The tasks running in parallel are summarized in Table III.

Table III: Master HC11 tasks

#	Task	Type	Time constraints
1	Execute command as requested by PC	Main program	Low
2	Generation of PWM signals	Software interrupt: OC1 Hardware interrupts: OC2 - OC5	Every 20 ms
3	Continuous A/D conversions	Software interrupt: OC2	Every ~20 ms (when PA6 goes high)

In the current implementation, the main program is interrupted about every 20 ms by tasks #2 and #3. Because the interrupt service routines are small most of the computing power is reserved for the main program. Currently, only a small portion of the master HC11’s computing power is used, so that there is enough computing power and program space left for future extensions.

#### 2.1.1 Task #1 - Communication and Program Execution

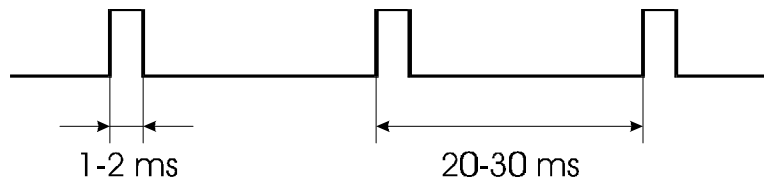
The main program spends most of its time waiting for a new command byte from the PC. When it receives a new command, it will decode it according to Table II (see page 11) and execute it. If the command is for the motors, the HC11 will put the corresponding value in an internal register, which will be read later by the PWM interrupt routine (see task #2).

If the PC needs to know the value of an analog input, the HC11 can simply read the value from the last A/D conversion and send it back to the PC through handshaking as explained in Section 1.2.4.

If the command is for one of the HC11 slaves, the master HC11 will send it to them through the SPI line as explained in Section 1.2.5.

### 2.1.2 Task #2 - Generation of the PWM Signals

Task #2 generates the PWM signals for up to four motors. Specifications for the PWM signals required to drive the Pioneer AT's power amplifiers were not available at the time of writing this document. For this reason we will assume here a typical PWM control signal that repeats every 20-30 ms with a variable pulse width ranging from 1 to 2 ms in duration:



**Figure 14:** Typical PWM control signal (may be different on actual Pioneer AT).

The generation of these signals can be implemented by two different methods. In the first method, the master HC11 generates the PWM signals with a 32.77 ms period. Even though this period is higher than the typical period, the PWM amplifier will still work properly. The reason for the choice of this particular period is that the internal timer of the HC11 overflows after 32.77 ms. Therefore, the PWM signals can be generated based on hardware interrupts only, without requiring any software interrupt servicing. This is very efficient, as it takes no computation power of the HC11. However, if a new command with a smaller pulse width is given to the HC11 in the time span between the falling edges of the new and old command, the PWM signal will stay high for an entire period. Since the critical time span is rather small, this problem occurs rarely. If it happens, the effect is small as the momentarily wrong command is filtered by the inertia of the motors and platform body.

In the second method, the PWM signals are generated with the typical period of 20 ms. This method eliminates the potential problem of the first method. It is also advantageous over the first method if the sampling rate of the UM software is smaller than 32.77 ms. As a disadvantage, this method requires software interrupt servicing, and therefore takes some computation power of the HC11. However, since the interrupt service routine is small and the master HC11 has more than enough computation power for its other tasks, the second method was implemented as it is safer and allows higher sampling rates.

This second method can be implemented with the HC11 *output compare functions*. These real-time interrupt functions allow one to efficiently implement up to four PWM signals. In the Pioneer AT configuration, two (or four<sup>5</sup>) PWM signals will be used. A basic and an enhanced method of implementation are shown in Figure 15.

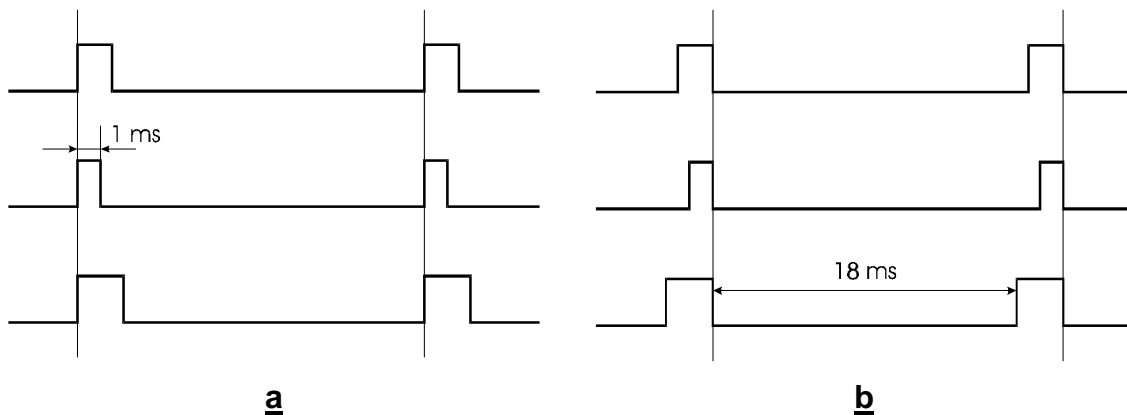
---

<sup>5</sup> UM will investigate the feasibility of its newly proposed method for detecting wheel slippage on the Pioneer AT: Taking advantage of the four built-in motors of the Pioneer AT, UM will decouple the currently mechanically linked wheel pairs on each side of the platform and drive each wheel independently.

With the basic method, the OC1 (output compare 1) hardware interrupt is used to simultaneously generate the rising edges of the two (or four) PWM signals every 20 ms. The OC2-OC5 hardware interrupts are used for the falling edges of the four respective PWM signals.

In the enhanced method, the OC1 hardware interrupt is used to simultaneously generate the falling edges of the two (or four) PWM signals every 20 ms. The OC2-OC5 hardware interrupts are used for the rising edges of the respective PWM signals.

With both methods, only the OC1 interrupt is serviced. This interrupt service routine calculates the output compare timer values for the next set of events and writes them to the corresponding registers. For the basic method, as shown in Figure 15, these values must be updated no later than 1 ms after the OC1 interrupt occurred. In the current implementation, 1 ms is more than sufficient. However, if the master HC11 has to take care of more parallel tasks in future extensions, this time limit could be a problem. With the enhanced method, the time limit is 18 ms, so that there should be no problem even if several more tasks were added. For this reason, the enhanced method was implemented.



**Figure 15:** Generation of PWM signals: a) basic method, b) enhanced method

### 2.1.3 Task #3 - Continuous A/D Conversions

Analog-to-digital (A/D) conversion of signals will likely be necessary for some of the auxiliary sensors, such as UM's proposed 2-axis tilt sensor and, possibly, for a KVH fluxgate compass. The master HC11 can read the analog output values of these sensors. The conversion has to be triggered by an internal command or by the rising edge of an external signal. In our current implementation the signal is generated by the OC2 hardware interrupt, that is, the OC2 interrupt service routine is used to sample the signal at that specific time.

---

Then, using an additional set of wheel encoders (currently only the two front motors have encoders), conclusions about wheel slippage can be drawn by comparing the encoder readings from all four wheels.

In the current implementation analog signals are actually converted four consecutive times. Once the A/D conversion is started by the OC2 interrupt service routine, the HC11 hardware automatically makes these four conversions in a time interval of only 64  $\mu$ s.

The results of these conversions are stored in registers, which can be read by task #1. These four measurements are averaged before being sent to the PC.

## 2.2 The HC11 Slave EERUF Implementation

One of the foremost advantages of the MCIB is its built-in implementation of the EERUF algorithms for crosstalk and noise rejection with ultrasonic sensors. This section discusses some of the implementation details, but not the theoretical background for this approach.

### 2.2.1 The Multitasking Architecture

Because of the limited number of available input/output lines and because of concerns over the speed of program execution for the time-sensitive EERUF algorithm, one HC11 is dedicated for each group of eight ultrasonic sensors. In the current implementation there are two such slave HC11s, allowing a total of 16 ultrasonic sensors to be controlled accurately. Each slave HC11 has to accomplish several tasks at different rates and with different time constraints. To implement this efficiently, the slave HC11 software is also organized in a multi-tasking architecture. The four different tasks with their properties are summarized in Table IV:

**Table IV:** slave HC11 tasks

#	Task	Type	Time constraints
1	Compute TOF and write into FIFO	Main program	Low
2	Generate fire signals and set time for next BINH interrupt	Software interrupt: OC4	13 times in 200 ms according to EERUF schedule
3	Check echo signals	Software interrupt: OC5	Every 50 $\mu$ s
4	Generate BINH signals	Hardware interrupt: OC1	Exactly 0.6 ms after corresponding fire signal

The main program (task #1) is constantly interrupted by the two software interrupts (tasks #2 and #3) which generate the fire signals and check the echo signals. The fourth task (task #4) is supported by the HC11 hardware so that it does not require interrupt servicing. However, the times at which the BINH signals are changed are set during the

interrupt servicing of task #2. Tasks #2 and #3 communicate with task #1 through an internal FIFO buffer and several global variables.

### 2.2.2 Task #1 - Communications and Treatment of the Buffer

Task #1 reads the internal FIFO buffer containing the data from tasks #2 and #3. It analyzes this data and computes the time of flight (TOF). This task also takes care of all the other not time-critical tasks, e.g. the SPI communication with the master HC11.

The time constraints of task #1 are low. However, if it is not executed often enough, the internal buffer may overflow. As the timing of task #2 is fixed by the EERUF schedule and task #4 is only a hardware interrupt, only the timing of task #3 can be changed to influence the time allocation of task #1. Basically, the higher the execution rate of task #3, the better the sonar resolution, but also the less time is allocated to task #1, and hence the higher the risk of an internal buffer overflow. However, tests have shown that task #3 can even be executed every 25  $\mu$ s without problems. The algorithm of task #1 is summarized as following:

1. New request from master HC11? If yes execute, e.g. start sonars, stop sonars, change EERUF mode, fire single sonar.
2. Check internal buffer. If empty go back to step 1.
3. Read next byte from internal buffer.
4. If the byte is equal to *No\_Echo*, the corresponding sonar(s) has not received back an echo. Write index and TOF equal to zero into FIFO. Go back to step 1.
5. Otherwise, compute the TOF for corresponding sonar(s). Write index and TOF into FIFO.
6. Go back to step 1.

### 2.2.3 Task #2 - The Generation of the Fire Signals

Task #2 is responsible for generating the fire signals and setting the time for the next BINH interrupt. The fire signals must be generated as specified by the EERUF time schedule. This task is executed 13 times in an interval of 200 ms. Its algorithm is the following:

1. If a fire signal is deactivated, check if the corresponding echo was received. If not, write the *No\_Echo* byte and the index of the corresponding sonar(s) into the internal buffer.
2. Output new fire signals and save time into array `p_list_fire[index]`.
3. Set time and output for next BINH signal.
4. Set time for next fire signal event.
5. Increment EERUF table pointers.
6. Execute task #3.

### 2.2.4 Task #3 - Checking for Echoes

Task #3 is the most time intensive as it checks for new echoes every 50  $\mu$ s. The more often this task is executed, the better the sonar resolution. Hence, it is important to make this routine as short and fast as possible. For this reason, task #3 only checks for new echoes, but neither identifies the sonar(s) nor computes the TOF. It just saves the current time and a representative byte into the internal buffer. The representative byte contains bits set to one for sonars with new echoes. This data is then treated by task #1. The algorithm of the interrupt service routine is as follows:

1. Read echoes.
2. Check for new echoes with:  $\neg(\text{previous\_echoes}) \wedge (\text{current\_echoes})$ .
3. If the result is not zero, save it and the current time in internal buffer.
4. Save current echoes as previous\_echoes.
5. Set time for next interrupt.

### 2.2.5 Task #4 - The Generation of the BINH Signals

Task #4 is the most time constrained as the BINH signal must be activated exactly 0.6 ms after the fire signal. This can be done by using the hardware interrupt OC1 without any interrupt servicing. When the HC11 timer becomes identical to the time set in step 3 of task #2, the OC1 interrupt will automatically output the four bits defined in the same step. This assures exact activation of the BINH signals independent of other interrupts.

### 2.2.6 The Fire Signal Table

The original EERUF fire schedule<sup>6</sup> is tailored for a sonar system consisting of 12 sonars, and each sonar is fired once in every 100 ms interval. The difference between two consecutive sonar echoes must be smaller than 1 ms to be accepted as a valid reading. These fire signals are generated by task #2 according to the EERUF schedule. For an efficient software implementation of task #2, the EERUF time schedule is stored in a look-up table.

---

<sup>6</sup> Borenstein, J. and Koren, Y., "Error Eliminating Rapid Ultrasonic Firing for Mobile Robot Obstacle Avoidance", IEEE Transactions on Robotics and Automation, February 1995, Vol. 11, No. 1, pp. 132-138.