View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

M68HC11 GRIPPER CONTROLLER SOFTWARE

NAGW-1333

by

Jodi Wei-Duk Tsai

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering
Troy, New York 12180-3590

May 1991

CIRSSE REPORT #90

© Copyright 1991
by
Jodi Wei-Duk Tsai
All Rights Reserved

CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGEMENT viii

ABSTRACT ix

1. Introduction 1

2. Overview 2
   - 2.1 Mechanical Aspects 2
   - 2.2 Software Interface 2
   - 2.3 Hardware Interface 2

3. VxWorks Gripper Library 7
   - 3.1 grRegWrite, grRegRead 7
     - 3.1.1 Synopsis 7
     - 3.1.2 Description 7
     - 3.1.3 Return Value 7
     - 3.1.4 Errors 9
     - 3.1.5 See Also 9
   - 3.2 grSrvGrab, grSrvForce. grSrvPos 10
     - 3.2.1 Synopsis 10
     - 3.2.2 Description 10
     - 3.2.3 Return Value 10
     - 3.2.4 Errors 10

4. Firmware Software Architecture 11
   - 4.1 Initialization 11
     - 4.1.1 Power Up Initialization (powerUpStart) 11
     - 4.1.2 ACIA Initialization (ACIAInit, ONACIA) 11
     - 4.1.3 RAM Initialization (initRAM) 13
     - 4.1.4 Hardware Initialization (hwInit) 14

4.1.5 Software Initialization (swInit) 18
4.2 Command Processing (processCmd) 18
4.2.1 Register Model 20
4.2.2 Servo Modes (grSrvGrab, grSrvPos, grSrvForce) 22
4.3 Interrupt Level (positionServo,forceServo,grabServo) 24
4.3.1 Grab Servo (grabServo) 24
4.4 Critical Data 26
4.4.1 Floating Point Control Parameters 26
4.4.2 Working Memory of Floating Point Routines 26
4.5 EPROM 27
4.6 Control Loop 27
4.7 Command Processing Real-Time Constraints 27
4.8 Debugging 28
4.8.1 EPROM Configuration 28
4.8.2 Debug Configuration 28
5. Tutorial 29
5.1 Initializing the Network Daemon 29
5.2 Creating the 68HC11 Buffalo Monitor Window 29
5.3 Constructing an Imakefile 31
5.4 Assembling and Downloading 31
6. Tools 32
6.1 rDevLogin - remote device login 32
6.1.1 Synopsis 32
6.1.2 Description 32
6.2 rDevd - remote device daemon 32
6.2.1 Synopsis 32
6.2.2 Description 33
6.3 rDevCmd 33
6.4 a611floatpp 35
6.5 moto2moto 35
6.6 motooffset 36

6.7 installMan 37
6.8 manc 37
6.9 xmanc 37
6.10 Public Domain Utilities 38
6.10.1 68HC11 Assembler 38
6.10.2 int2moto 38

7. Configuration Management 39
7.1 Motivation 39
7.1.1 Complicated Makefiles 39
7.1.2 Working Versions of Files 39
7.2 Imake Configuration Files 40
7.2.1 Files 40
7.2.2 Caveats 47
7.2.3 Bugs 47
7.2.4 See Also 47
7.3 Imakefile 47
7.3.1 VxWorks Library 48
7.3.2 VxWorks Binary 50
7.3.3 UNIX Library 51
7.3.4 UNIX Binary 52
7.3.5 Making subdirectories with no local targets 53
7.3.6 Making subdirectories with local targets 53
7.3.7 Making local targets only 56
7.3.8 Custom rules and dependencies 56

8. Conclusions 60

LITERATURE CITED 61

LIST OF TABLES

2.1	68HC11 Hardware Interface	3
3.1	Register Constants	8
3.2	Errors	9
4.1	Memory Map	14
4.2	Interrupt Vector Map with Buffalo monitor	16
4.3	Interrupt Vector Map with gripper controller	16
4.4	Command Character Table	19
4.5	Subroutine Table	19
4.6	RS-232 Serial Port Commands	20
4.7	Error Codes	20
4.8	Registers	21
4.9	State Transition Table	24
4.10	Bit Representation	25
4.11	State Transition Table in Bit Representation	25
6.1	Action Table	35

LIST OF FIGURES

2.1	CIRSSE Gripper	3
2.2	Configuration	4
2.3	Schematic of Pneumatic Piston Assembly	5
4.1	Software Architecture	12
4.2	Real-Time Clock Error Caused by Interrupt Overrun	18
4.3	Top Level State Transition Diagram	22
4.4	Grab Servo Mode State Transition Diagram	23
5.1	Hardware Configuration	30
6.1	Network Connections	34
6.2	Software Architecture	36

ACKNOWLEDGEMENT

I would like to thank my mother and father for their support. I would also like to thank Dr. Robert B. Kelley, my thesis adviser, for his guidance during the various stages of development.

ABSTRACT

This thesis discusses the development of firmware for the 68HC11 gripper controller. A general description of the software and hardware interfaces is given. The C library interface for the gripper is then described followed by a detailed discussion of the software architecture of the firmware. A procedure to assemble and download 68HC11 programs is presented in the form of a tutorial. The tools used to implement this environment are then described. Finally, the implementation of the configuration management scheme used to manage all CIRSSE software is presented.

CHAPTER 1
Introduction

The objective of this project was to produce a low-cost gripper controller for CIRSSE and attain the following design goals:

- provide an RS-232 command interface.
- implement position feedback control using potentiometers.
- implement force feedback control using strain gauges.
- allow the gripper to "grab" objects using the infrared sensors.
- minimize sampling period (maximize sampling rate) to allow maximum possible fluctuation in gripper control parameters.

In addition, software tools were created for SunOS and VxWorks to provide an easy to use edit-compile-download-debug environment for the development of programs using the 68HC11 EVB on a Sun workstation.

CHAPTER 2
Overview

2.1 Mechanical Aspects

The gripper is shown in Figure 2.1. The gripper has a crossfire infrared sensor to detect the presence of an object between the fingers. Potentiometers and strain gauges allow position and force feedback control to be implemented.

2.2 Software Interface

The top level interface through the gripper is through the VxWorks gripper library on the MV135 68020 processor boards. The actual servoing is accomplished with the 68HC11 EVB. Communication occurs between the MV135 68020 processor boards and the 68HC11 EVB via an RS232 cable. This configuration is shown in Figure 2.2.

2.3 Hardware Interface

The CIRSSE gripper is a pneumatically controlled system shown in Figure 2.3. The pressure is adjusted by writing an 8-bit value to port B of the 68HC11. This port is connected to a D/A converter outside of the 68HC11 evaluation board. The D/A converter generates a signal that is amplified to control a servo valve. This effectively controls the rate at which air enters the piston from both sides. Table 2.1 shows the port mapping.

Since the air pressure is fixed at all times, the 8-bit value at the 68HC11 port merely acts as an integral gain. A point exists where the servo valve delivers air at the same rate to both sides of the piston. The digital output value at this point is known as $z_{pressure}$. Although the digital output value may take on the full range of

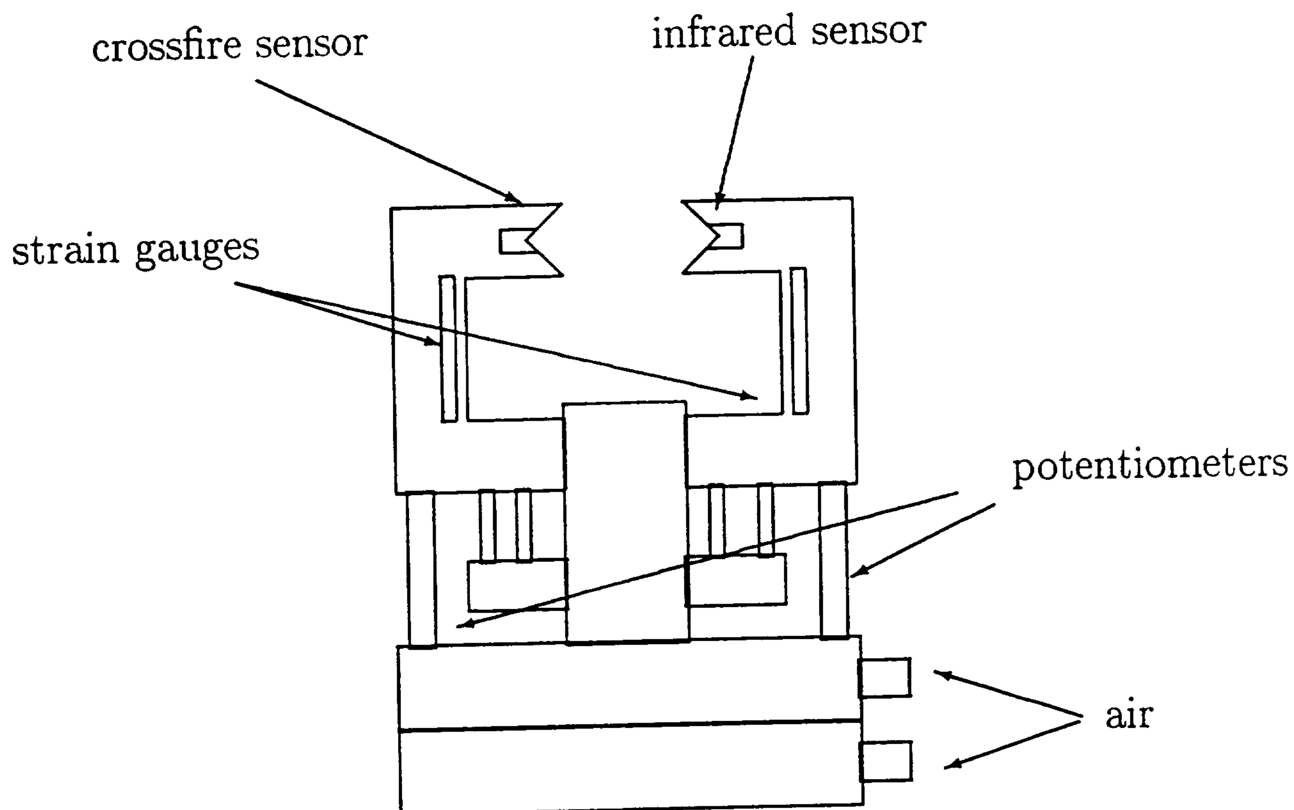

Figure 2.1: CIRSSE Gripper

function	*68HC11 port*
position	E0
force	E1
crossfire sensor	C bit 0
air valve	B bit 7-0

Table 2.1: 68HC11 Hardware Interface

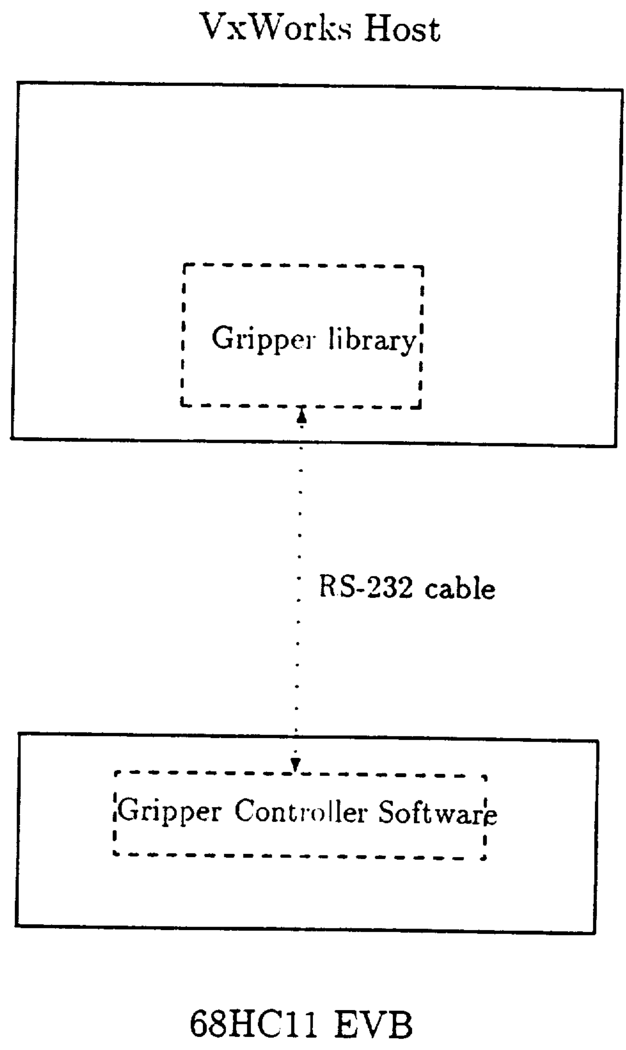

Figure 2.2: Configuration

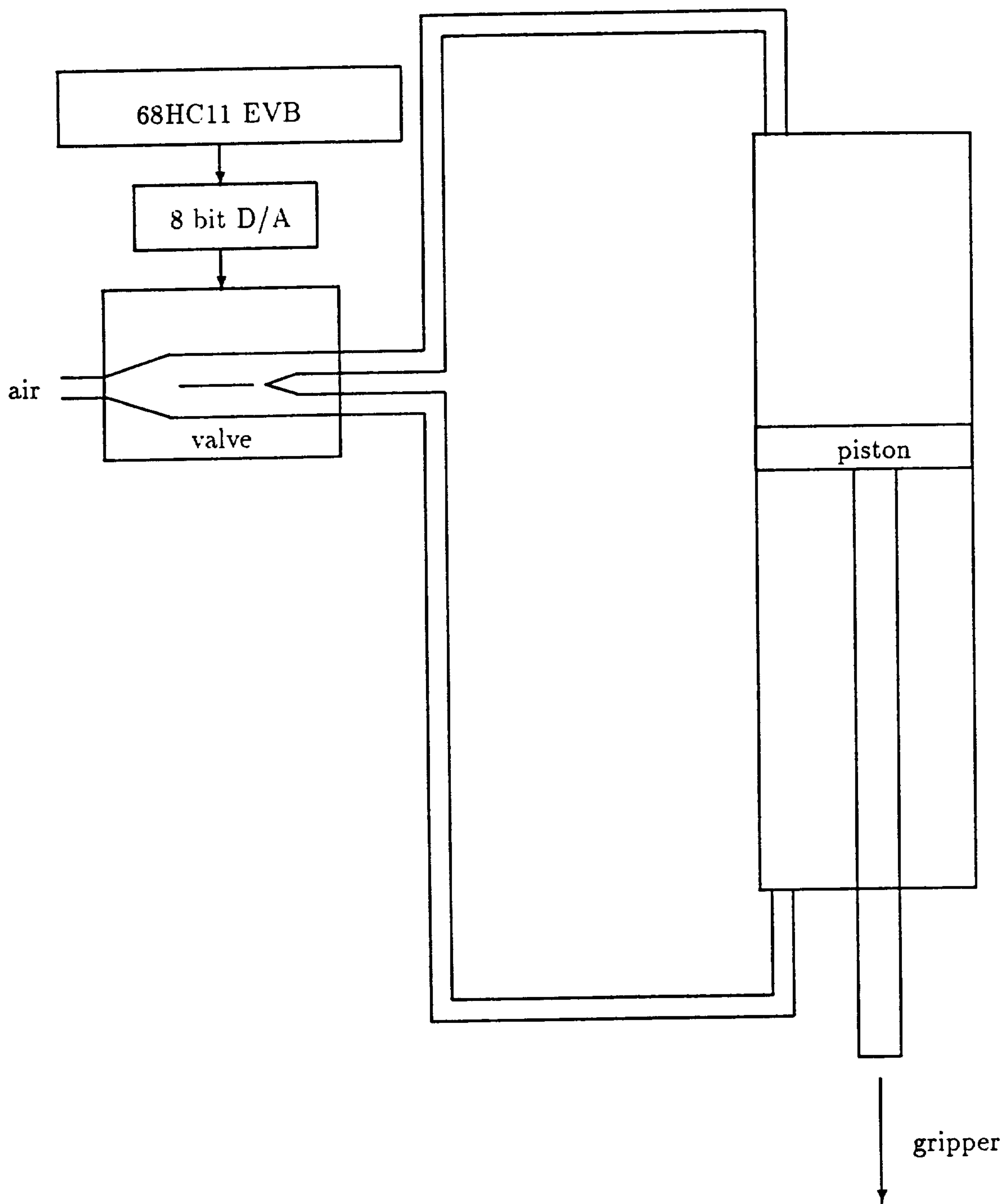

Figure 2.3: Schematic of Pneumatic Piston Assembly

8-bit values, in practice, only very slight perturbations are needed from $z_{pressure}$ in order to move the fingers of the gripper to any given position or force setpoint.

The current method used to adjust the pressure is a proportional-integral-derivative (PID) controller. All parameters may be dynamically adjusted while the gripper controller is servoing.

CHAPTER 3
VxWorks Gripper Library

The gripper library is a standardized C library interface to the gripper controller. This allows future changes to be made to the software in the 68HC11 without invalidating the code written at the application level.

The gripper library also isolates the application software from the real-time constraints of the gripper protocol. This is done by providing the proper intercharacter delay and the intercommand delay when sending commands to the gripper.

3.1 grRegWrite, grRegRead

3.1.1 Synopsis

```
#include <grLib.h>


int grRegWrite(int fd,GRREG regNo,double value)
int grRegRead(int fd,GRREG regNo,double *pValue)
```

3.1.2 Description

These functions read and write to software registers in the 68HC11. In all of these functions, the argument *fd* is a file descriptor returned by **open (2V)** [Sun 88].

The argument *regNo* refers to one of the constants in Table 3.1. The mathematical representations of these registers is given in Table 4.8.

3.1.3 Return Value

These functions return 0 if the call is successful otherwise 1 is returned.

constant	*remarks*
grReg_positionSetPoint	fully open = 0, fully closed = 255
grReg_positionPropGain	
grReg_positionDerivGain	
grReg_positionIntglGain	
grReg_positionIntglMin	
grReg_positionIntglMax	
grReg_forceSetPoint	minimum force = 0, maximum force = 255
grReg_forcePropGain	
grReg_forceDerivGain	
grReg_forceIntglGain	
grReg_forceIntglMin	lower bound on the integral of the force error
grReg_forceIntglMax	upper bound on the integral of the force error
grReg_xfireGrabThreshold	for grab mode, if $x_{sensor} \leq x_{threshold}$ then force servo, otherwise position servo
grReg_xfireLED	turns crossfire LED on or off. off = 0, on = 1
grReg_zeroPressure	digital output value to air valve to apply zero force
grReg_position	fully open = 0, fully closed = 255
grReg_positionError	
grReg_positionErrorPrev	
grReg_positionPropTerm	
grReg_positionDerivTerm	
grReg_positionIntglTerm	
grReg_positionIntgl	
grReg_positionIntglNext	
grReg_positionControl	
grReg_positionControlOutput	
grReg_force	min force = 0, max force = 255
grReg_forceError	
grReg_forceErrorPrev	
grReg_forcePropTerm	
grReg_forceDerivTerm	
grReg_forceIntglTerm	
grReg_forceIntgl	
grReg_forceIntglNext	
grReg_forceControl	
grReg_forceControlOutput	
grReg_xfireSensor	min intensity = 0 maximum intensity = 255
grReg_realTime	time in seconds, rolls over back to zero every $65536(8.19_{milliseconds}) = 536.739_{seconds}$

Table 3.1: Register Constants

constant	*remarks*
S_grLib_VxWorks_INVARG	An invalid argument was given when calling the function.
S_grLib_VxWorks_INVCMD	An invalid command was given. This indicates that garbage was sent over the RS232 serial line, probably a hardware fault.
S_grLib_VxWorks_BAD_RESPONSE	The response from the 68HC11 could not be recognized because the 68HC11 EVB was not initialized properly
S_grLib_VxWorks_TIMEOUT	There was no response from the 68HC11 EVB.
EBADF	**ioctl(2)** failed because the stream was not associated with a valid file descriptor.
ENOTTY	**ioctl(2) FIONREAD** request does not apply to kind of object associated with the file stream.
EINVAL	The request or arguments to **ioctl(2)** were invalid.
EFAULT	The Data transfer that occurred within **ioctl(2)** resulted in a read or write to an address outside of the process' address space.

Table 3.2: Errors

3.1.4 Errors

The error codes returned are shown in Table 3.2.

3.1.5 See Also

In the UNIX man pages, see **open (2V)**, **close (2)** [Sun 88].

3.2 grSrvGrab, grSrvForce, grSrvPos

3.2.1 Synopsis

```
#include <grLib.h>


int grSrvGrab(int fd)

int grSrvPos(int fd)

int grSrvForce(int fd)
```

3.2.2 Description

These functions enable various servo modes. The gripper controller software in the 68HC11 attempts to maintain the position and force set points set by the **grReg(2)** commands.

In all these functions, the argument *fd* is a file descriptor returned by **open(2V)**.

grSrvGrab() enables position mode or servo mode depending on whether there is an object detected between the fingers of the gripper.

grSrvPos() enables position servo mode and attempts to maintain the position.

grSrvForce() enables force servo mode and attempts to maintain the force set point.

3.2.3 Return Value

These functions return 0 if the call is successful otherwise 1 is returned.

3.2.4 Errors

See Table 3.2.

CHAPTER 4

Firmware Software Architecture

The software for the gripper controller is a simple real-time executive which operates in three different modes.

grab servo mode servos to the force set point if there is an object detected between the fingers, otherwise servos to the position set point.

position servo mode servos the gripper to maintain a position set point.

force servo mode servos the gripper to maintain a force set point.

For each section or subsection heading, the relevant subroutine in the gripper controller source code is indicated in parentheses.

4.1 Initialization

4.1.1 Power Up Initialization (powerUpStart)

Upon power up, the 68HC11 EVB [1] starts executing instructions at 0xe000. The power up startup code sets the OPTION register and TMSK2 register immediately since these must be set within 64 bus-cycles after reset [Motorola 89, p. 3-8]. After this period expires, some of the bits in these registers become read-only and may not be set.

4.1.2 ACIA Initialization (ACIAInit, ONACIA)

ACIAInit initializes the ACIA on power up to enable communication via the RS-232 serial port. **ONACIA** re-initializes the ACIA after an error.

[1]The 68HC11 EVB uses the MC68HC11A1 version of the M68HC11 family

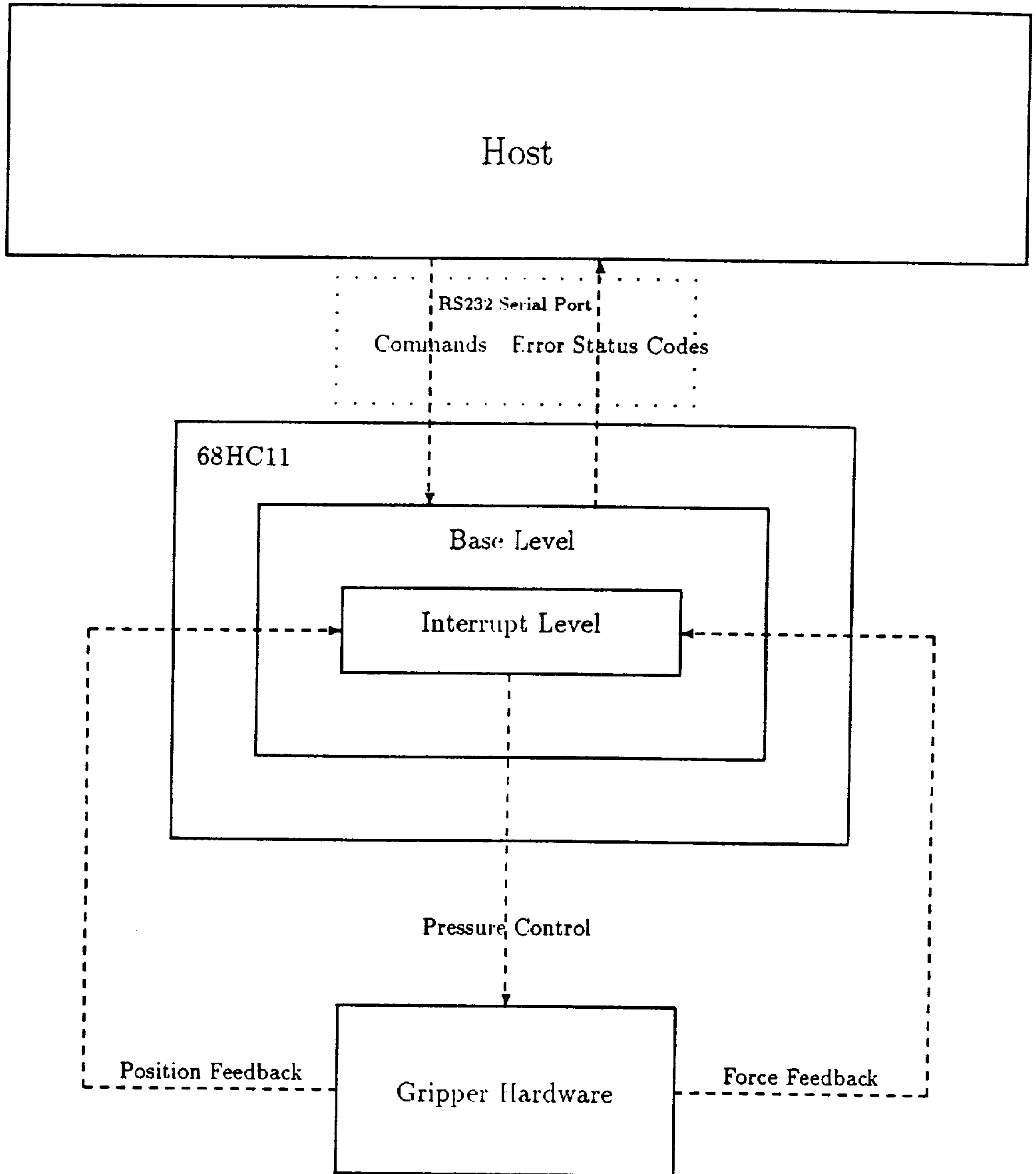

Figure 4.1: Software Architecture

Note that there are two RS-232 serial ports on the M68HC11 EVB. The *terminal* port is connected to the ACIA. The other ***host*** port is connected to the SCI on the M68HC11 and is **not** used [Motorola 86, Figure 6-2].

4.1.3 RAM Initialization (initRAM)

The on-chip RAM space is only 256 bytes. The on-chip EEPROM space is only 512 bytes [Motorola 89, Table 1-1]. Since the gripper controller software exceeds the capacity of both on-chip memories, an external RAM and an external EPROM are needed.

Two methods were considered in allocating program space to the external RAM and EPROM.

1. using the external RAM area for uninitialized data and initialized data; using the EPROM area for constant data, initialized data, and code.

2. using the RAM area for all code and data.

The first method uses EPROM and RAM space optimally. However, it introduces complexity by requiring a separate section of memory for constant data, initialized data, and uninitialized data.

The second method treats all code and data the same and merely copies the whole contents of the EPROM to the RAM area and then executes the code residing in RAM. The primary benefit of this method is that it is trivial to implement at the expense of wasting some space by needlessly duplicating constant data and code.

The second method is used for this application. The firmware copies itself to RAM after performing a few required initializations and jumps to the first routine **main** in RAM. Note that all jumps and branches prior to the **main** routine *must* use relative addressing since the program is assembled with the origin at the start of RAM (0xc000) not EPROM (0xe000). A memory map is shown in Table 4.1.

address	*description*
0x0000	direct RAM
0xc000	user RAM gripper controller code gets copied EPROM and is executed here
0xdf00	top of stack
0xe000	EPROM initialization copies code and data to RAM

Table 4.1: Memory Map

4.1.4 Hardware Initialization (hwInit)

During hardware initialization, the following are accomplished.

- crossfire port initialization
- A/D converter register initialization
- interrupt subroutine vector initialization
- real-time interrupt register initialization
- hardware diagnostics

4.1.4.1 Crossfire Port Initialization (hwInitXfire)

The crossfire LED is turned on by writing to bit 0 of the PORTC register. The port C databus is bi-directional. Each bit in the DDRC register determines whether the corresponding bit of the databus is either an input or output line [Motorola 89, p. 7-20]. Thus, proper configuration of the DDRC register requires that bit 0 be high.

4.1.4.2 A/D converter initialization (hwInitADC/ADCInit)

The A/D converter subsystem is initialized by the ADPU and CSEL bits in the OPTION control register. The A/D charge pump is initially disabled upon reset. The ADPU bit must be set to one to enable the A/D charge pump [Motorola 89, p. 12-12]. The CSEL bit is set to zero for this application since additional A/D error is caused by internal noise if the alternate clock source for the on-chip EEPROM charge pump is selected during A/D conversion [Motorola 89, p. 12-13].

4.1.4.3 Interrupt Vector Initialization (RTISetISR)

During each of the grab, position, and force servo modes mentioned in Section 4, a separate interrupt subroutine is used. These interrupt subroutines are set with the **RTISetISR** subroutine.

4.1.4.4 Real-Time Interrupt Initialization (RTIInit)

The real-time interrupt serves three purposes.

- provides periodic activation of the servo loop to ensure a constant sampling rate.
- allows a timer to be maintained so that the delay for grab mode can be implemented.
- maintains a real-time clock to determine if the servo loop is executing within the proper time.

RTIInit first sets the tick counter to zero. The counter is incremented on every real-time interrupt. This counter is used to calculate the real-time in seconds and is used to schedule events in grab mode.

Since the interrupt vector table is located from 0xffd6 to 0xffff, the Buffalo monitor automatically presets the interrupt vectors. However, these vectors merely

address	*description*
0x00c4	Start of Buffalo JMP vector table
0x00eb	JMP RTIInt
0x00ff	End of JMP vector table
0xffd6	Start of Buffalo monitor vector table
0xfff0	0x00eb
0xffff	End of Buffalo monitor vector table

Table 4.2: Interrupt Vector Map with Buffalo monitor

address	*description*
0x00c4	Start of unused JMP vector table
0x00eb	JMP RTIInt (still initialized but not used)
0x00ff	End of unused JMP vector table
0xffd6	Start of gripper controller vector table
0xfff0	RTIInt
0xffff	End of gripper controller vector table

Table 4.3: Interrupt Vector Map with gripper controller

point to JMP instructions in RAM which jump to the appropriate interrupt handler. Thus, a JMP instruction to the real-time interrupt handler must be present at the real-time interrupt address of the Buffalo monitor jump table. This configuration is shown in Table 4.2.

When the Buffalo monitor is removed and replaced with the gripper controller software, the jump table in RAM is no longer necessary since the interrupt vector table located from 0xffd6 to 0xffff can be set with the appropriate vectors by initializing the corresponding EPROM locations. The JMP instruction is still harmlessly initialized since the gripper controller is never aware of whether it was copied from EPROM or downloaded via the RS-232 port. This configuration is shown in Table 4.3.

The TFLG2 register is used to clear the RTIF status bit which is automatically set at the end of every interrupt [Motorola 89, p. 10-12]. The RTI rate is then set by initializing the lower two bits of the PACTL register.

When real-time interrupt initialization is complete, the interrupts are in a disabled state. They may be turned on and off with the **RTIStart** and **RTIStop** routines.

A real-time clock is maintained and is displayed as seconds. This parameter, t, is one of the gripper controller software registers shown in Table 4.8. If the interrupt handler routine does not execute in time for the next interrupt then the real-time clock here will be off from the actual elapsed time by some integer factor.

For example, if the interrupt routine takes 10ms to execute and an interrupt occurs every 8.19ms, the next interrupt will be missed. Therefore the real-time clock will be slowed by a factor of two. Thus, by merely observing the real-time clock and using a wristwatch, it is possible to verify that real-time interrupts are not being missed. An example of this type of error is given in Figure 4.2.

4.1.4.5 Hardware Diagnostics (hwInit, hwInitADC)

Hardware diagnostics are performed to ensure that the following features of the M68HC11 EVB are functional:

- A/D conversion
- real-time interrupts

To test the A/D subsystem, a conversion command is output to the ADCTL register and the CCF flag is then polled to ensure that a conversion complete is returned at approximately the correct time.

To test the real-time interrupt, a subroutine to set a byte in memory is installed as an interrupt handler. Proper generation of interrupts is verified by checking the

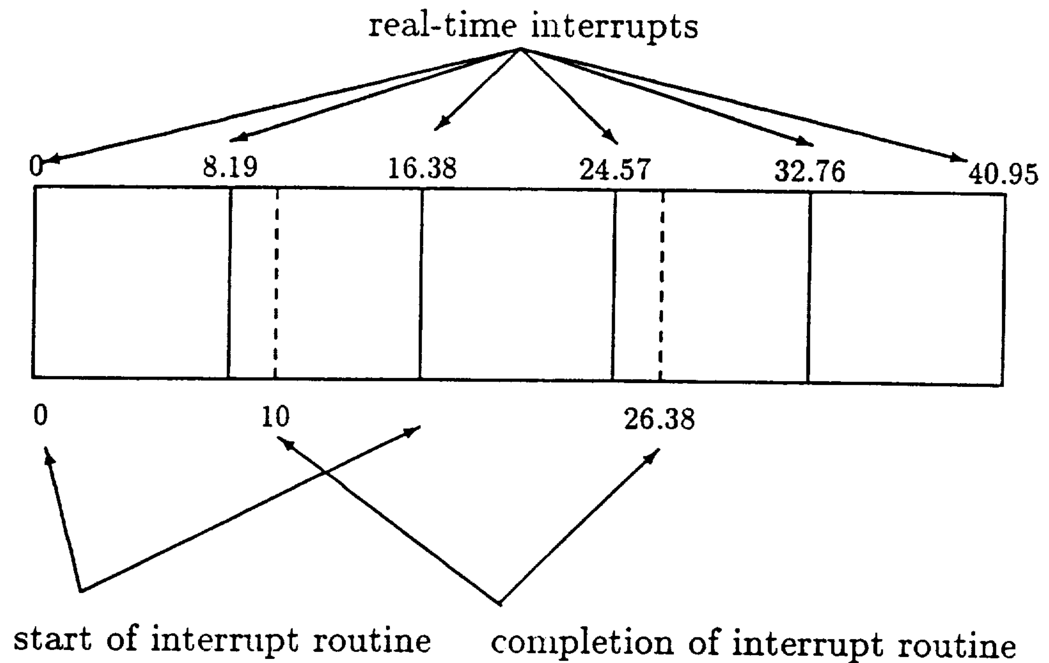

Figure 4.2: Real-Time Clock Error Caused by Interrupt Overrun

memory location after enabling and disabling interrupts.

4.1.5 Software Initialization (swInit)

Software initialization consists of all the run-time initialization of various constants used in the controller algorithms. The two goals are to minimize the amount of information that needs to be updated if parameters are changed and to optimize performance by computing common subexpressions beforehand.

4.2 Command Processing (processCmd)

Command processing is performed at the non-interrupt driven level. This level of execution performs all the command line processing through the RS-232 serial interface.

First, the software reads a command line from the RS-232 serial port. Each

offset	*command letter*
0	g
1	p
2	f
3	w
4	r
5	s

Table 4.4: Command Character Table

offset	*process command subroutine*
0	**grSrvGrab**
2	**grSrvPos**
4	**grSrvForce**
6	**grRegRead**
8	**grRegWrite**
10	**dispGripStat**

Table 4.5: Subroutine Table

command consists of a single letter. When a command is received, the command is parsed by searching the table of letters shown in Table 4.4.

Using the index into this table, a corresponding entry is found in the table of subroutine addresses shown in Table 4.5. The subroutine is then invoked to execute the command. The X register points to the current character being parsed. Upon entry into each of the subroutines invoked. the X register points to the arguments in the command buffer.

The commands for the gripper controller are shown in Table 4.6. The argument *register* in the **write register** and **read register** commands refers to one of the registers in Table 4.8. The *registerValue* parameter is a floating point number.

After processing each command, a line is returned to the host through the

function	*command to 68HC11*	*response from 68HC11*
grab servo mode	g	*errorCode*
position servo mode	p	*errorCode*
force servo mode	f	*errorCode*
read register	r *register*	*errorCode registerValue*
write register	w *register registerValue*	*errorCode*

Table 4.6: RS-232 Serial Port Commands

errorCode	*error*
0	no error
1	invalid argument
2	invalid command

Table 4.7: Error Codes

RS-232 serial port. The *errorCode* field contains the error status code. The meaning of the *errorCode* field is shown in Table 4.7. There is another *registerValue* field returned in response to the **read register** command. This is a floating point number.

4.2.1 Register Model

The register model is used to set all of the parameters for the gripper controller primarily because the control algorithm is expected to change often. Therefore, it is important that control parameters can be easily added or deleted from the system without requiring any major changes to the gripper protocol.

In addition to the PID gains, there are additional registers representing terms which must be computed at the end of every sample period. These terms are included in order to assist the user in determining the correct gains to be used with a gripper. The mathematical representations for all these registers are shown in Table 4.8.

register	*control parameter*	*writable*	*representation*
0	position set point	yes	p_{setpnt}
1	position proportional gain	yes	p_{pgain}
2	position derivative gain	yes	p_{dgain}
3	position integral gain	yes	p_{igain}
4	position integral minimum	yes	p_{imin}
5	position integral maximum	yes	p_{imax}
6	force set point	yes	f_{setpnt}
7	force proportional gain	yes	f_{pgain}
8	force derivative gain	yes	f_{dgain}
9	force integral gain	yes	f_{igain}
10	force integral minimum	yes	f_{imin}
11	force integral maximum	yes	f_{imax}
12	crossfire grab threshold	yes	$x_{threshold}$
13	crossfire grab delay	yes	x_{delay}
14	crossfire LED	yes	x_{led}
15	zero pressure point	yes	$z_{pressure}$
16	position	no	p
17	position error	no	$p_{err} = p - p_{setpnt}$
18	previous position error	no	$p_{errprev}$
19	position proportional term	no	$p_{pterm} = p_{pgain} p_{err}$
20	position derivative term	no	$p_{dterm} = \frac{p_{dgain}(p_{err} - p_{errprev})}{T}$
21	position integral term	no	$p_{iterm} = p_{igain} p_i$
22	position integral	no	p_i
23	next position integral	no	$p_{inext} = p_i + \frac{T}{2}(p_{errprev} + p_{err})$
24	position control	no	$p_{ctl} = p_{pterm} + p_{dterm} + p_{iterm}$
25	position control output	no	$p_{out} = \min(\max(p_{ctl} + z_{pressure}, 0), 255)$
26	force	no	f
27	force error	no	$f_{err} = f - f_{setpnt}$
28	previous force error	no	$f_{errprev}$
29	force proportional term	no	$f_{pterm} = f_{pgain} f_{err}$
30	force derivative term	no	$f_{dterm} = \frac{f_{dgain}(f_{err} - f_{errprev})}{T}$
31	force integral term	no	$f_{iterm} = f_{igain} f_i$
32	force integral	no	f_i
33	next force integral	no	$f_{inext} = f_i + \frac{T}{2}(f_{errprev} + f_{err})$
34	force control	no	$f_{ctl} = f_{pterm} + f_{dterm} + f_{iterm}$
35	force control output	no	$f_{out} = \min(\max(f_{ctl}, 0), 255)$
36	crossfire sensor	no	x_{sensor}
37	expiration time	no	t_{expire}
38	real time	no	t
39	grab state	no	s_{grab}
40	grab mask	no	s_{mask}

Table 4.8: Registers

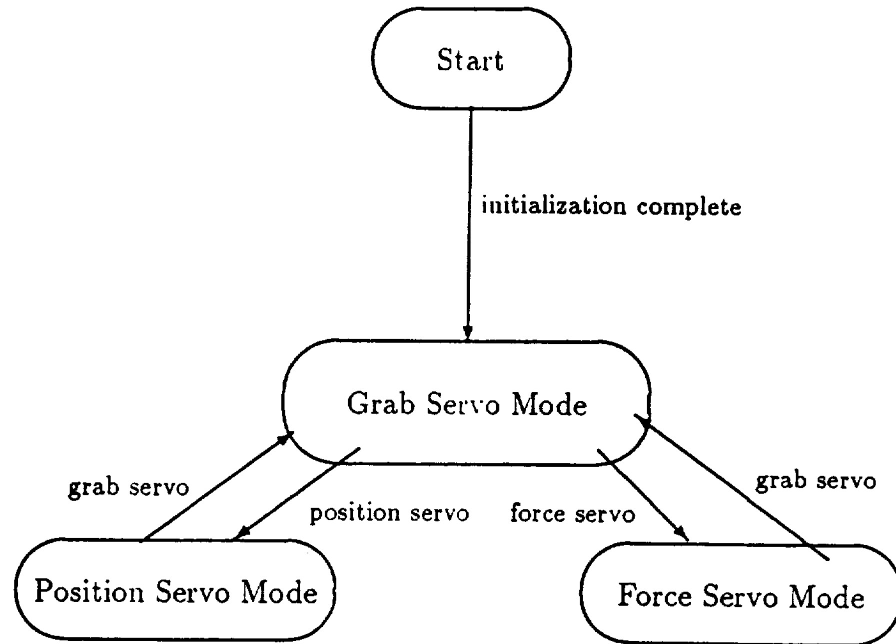

Figure 4.3: Top Level State Transition Diagram

4.2.2 Servo Modes (grSrvGrab, grSrvPos, grSrvForce)

The grab, position, and force servo commands initiate calls to **grSrvGrab**, **grSrvPos**, and **grSrvForce** respectively. All these routines initialize an interrupt subroutine vector using **RTISetISR**. This creates three top level states that are initiated by each command. The top level states for the gripper controller are shown in Figure 4.3. The gripper controller begins in grab servo mode upon initialization.

Both **positionServo** and **forceServo** are similar in that they contain no additional states. However, the **grabServo** mode contains substates.

When the gripper controller is in the grab servo state, the gripper will close whenever an object blocks the crossfire beam between the fingers. This closing can be delayed a certain period of time. Figure 4.4 shows the different states for grab servo mode. In both *Position Wait* and *Position Delay* states, position servoing is performed and in the *Force* state, force servoing is performed.

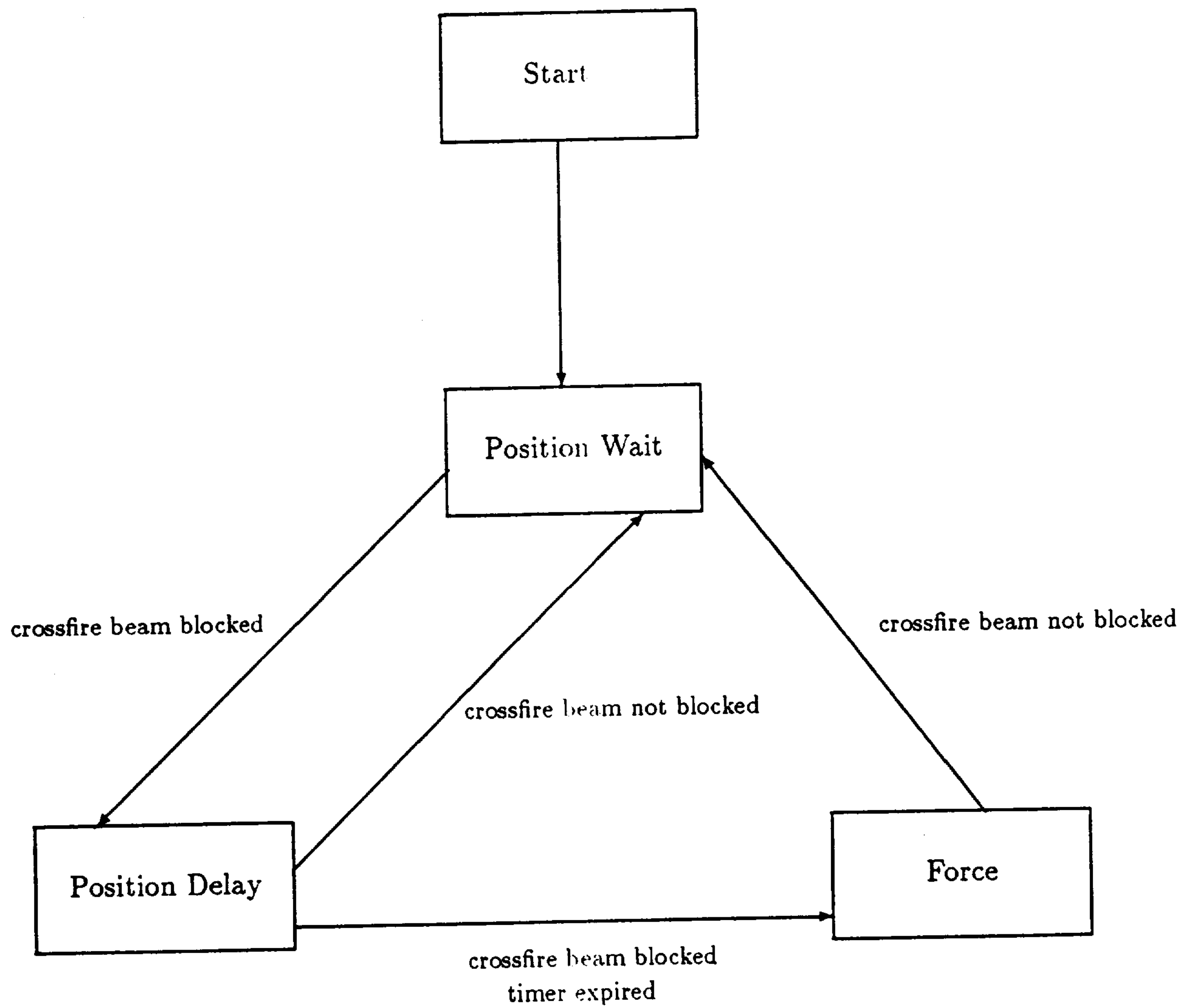

Figure 4.4: Grab Servo Mode State Transition Diagram

timer expired	*crossfire blocked*	*state* s_{grab}	*action*	*next state* s_{grab}
no	no	position wait	position servo	position wait
no	no	position delay	position servo	position wait
no	no	force	position servo	position wait
no	yes	position wait	start timer	position delay
no	yes	position delay	position servo	position delay
no	yes	force	force servo	force
yes	no	position wait	position servo	position wait
yes	no	position delay	position servo	position wait
yes	no	force	position servo	position wait
yes	yes	position wait	start timer	position delay
yes	yes	position delay	force servo	force
yes	yes	force	force servo	force

Table 4.9: State Transition Table

4.3 Interrupt Level (positionServo,forceServo,grabServo)

The interrupt level refers to the interrupt driven software in the controller, the interrupt handler code. This level of execution performs the servoing of the fingers. A periodic interrupt occurs every sampling period T. T is 8.19 milliseconds.

The ADCTL register is configured so that the position, force, and crossfire sensors are read simultaneously from ports E0, E1, and E3 respectively as shown in Figure 2.1. Conversions are performed on all three channels simultaneously by setting the SCAN bit to zero and setting the MULT bit to one [Motorola 89, p. 12-14].

4.3.1 Grab Servo (grabServo)

Table 4.9 shows the the grab servo state table. There are two inputs which cause all state changes. Table 4.11 is derived after substituting the bit representations in Table 4.10.

The actual implementation of the state transition table uses the first three

symbol	*bit representation*
no	0
yes	1
position wait	00
position delay	01
force	10
position servo	00
force servo	01
start timer	10

Table 4.10: Bit Representation

timer expired	*crossfire blocked*	*state* s_{grab}	*action*	*next state* s_{grab}
0	0	00	00	00
0	0	01	00	00
0	0	10	00	00
0	1	00	10	01
0	1	01	00	01
0	1	10	01	10
1	0	00	00	00
1	0	01	00	00
1	0	10	00	00
1	1	00	10	01
1	1	01	01	10
1	1	10	01	10

Table 4.11: State Transition Table in Bit Representation

columns of Table 4.11 as an offset into a table which contains the action routine to be performed and the next state.

4.4 Critical Data

There are two sets of critical data.

- the floating point control parameters written and read by the user via the **write register** and **read register** commands
- the working memory used by the floating point routines

4.4.1 Floating Point Control Parameters

Critical data is inevitably introduced because of the use of floating point control parameters. There is no atomic instruction which can read or write 3 bytes, the length of a floating point number. When a critical control parameter is set, the interrupts are masked while the floating point number is being transferred to the *register area* used by the interrupt handlers. This prevents the interrupt handler from using a partially updated floating point parameter. Fortunately, the amount of critical code is limited to two places in the software. Thus, masking can be performed without losing any interrupts and without causing any discontinuities in the servoing.

4.4.2 Working Memory of Floating Point Routines

Critical data also exists in the working memory of the floating point routines, labeled **floatContext** in the source code. Since one set of floating point routines are used for both non-interrupt and interrupt driven software, potential problems arise if both levels of execution are using the same data.

Since the working memory only involves 6 bytes of memory which can be copied quickly, the real-time interrupt handler saves the non-interrupt level contents

of this memory area before calling the interrupt handler. After the interrupt handler completes, the non-interrupt level contents is restored and execution resumes.

4.5 EPROM

The gripper controller firmware is designed to replace the Buffalo monitor firmware at 0x6000. Testing has been simplified by avoiding the use of any programming methods which would require the configuring of jumpers.

4.6 Control Loop

The control algorithm is essentially the same for position and force servoing. Thus, it would have been possible to have one generic control loop subroutine. However, this would require one of the following time consuming activities:

- indexing to each of the two control structures for each position and force parameter.
- copying of the control structure to and from the memory area used by the shared servo loop.

Both indexing and copying memory are time consuming activities for a large number of registers. Indexing also complicates the code significantly. It was decided to keep the code and data as simple and as fast as possible by keeping the position and force servo loops separate at the expense of using slightly more memory for code.

4.7 Command Processing Real-Time Constraints

There is a fundamental limit imposed on the number of characters which may be sent to the gripper controller in a certain period of time. Because the servo loop must always be executed every 8.19 ms, the non-interrupt level software only

executes between the time the servo loop code has completed *and* the next real-time interrupt. Since all reads from the RS-232 serial port occur at the non-interrupt level, there is a possibility that characters may not be read if characters are sent down too quickly. Thus, in order to guarantee that each character is read by the gripper controller software, it is necessary to wait at least 20 ms between characters. This will ensure that the servo loop has completely executed and that the non-interrupt level software has read the character from the serial port.

In addition to waiting between characters, it takes a minimum of 40ms to process a command. Thus, there must be at least a 40ms delay before any characters are sent down to the 68HC11 following any the input of any command.

4.8 Debugging

The gripper controller can be configured for EPROM or downloading via the RS-232 port. The latter configuration is useful for debugging the software.

4.8.1 EPROM Configuration

The software originally is located all in EPROM. When initialization begins the **initRAM** subroutine copies EPROM to RAM. The source file *ctl.asm* contains this code.

4.8.2 Debug Configuration

The software is downloaded into RAM at 0xc000. In this configuration, **initRAM** **must not** be called since the software is already relocated and the EPROM contains the Buffalo monitor. This source file is derived from *ctl.asm* by deleting the **bsr initRAM** instruction. This is done automatically by the *Imakefile*.

CHAPTER 5
Tutorial

A set of software development tools allows the programmer to perform all editing and debugging on a Sun workstation. These software development tools work within emacs and X windows to create an integrated edit-compile-debug environment. The procedure for editing, compiling and debugging is presented in this chapter.

The hardware configuration and the tools needed are shown in Figure 5.1.

This section assumes you are already familiar with UNIX and X windows.

5.1 Initializing the Network Daemon

Before any communication from the Sun workstation to the 68HC11 EVB can begin, a *network daemon* must first be present on one of the 68020 MV135 processor cards in the VME cage. This network daemon must be present on the same 68020 MV135 processor card that is connected to the 68HC11 EVB via an RS232 serial cable. In the following example, the 68HC11 EVB must be connected to the lower serial port (/tyCo/1).

The network daemon is invoked with the commands

```
rlogin vx2.ral.rpi.edu
ld < /usr2/testbed/CIRSSE/installed/VxWorks/bin/rDevd.o
rDevInit "/tyCo/1"
```

at the VxWorks prompt.

5.2 Creating the 68HC11 Buffalo Monitor Window

A terminal connection to the 68HC11 Buffalo monitor is started with

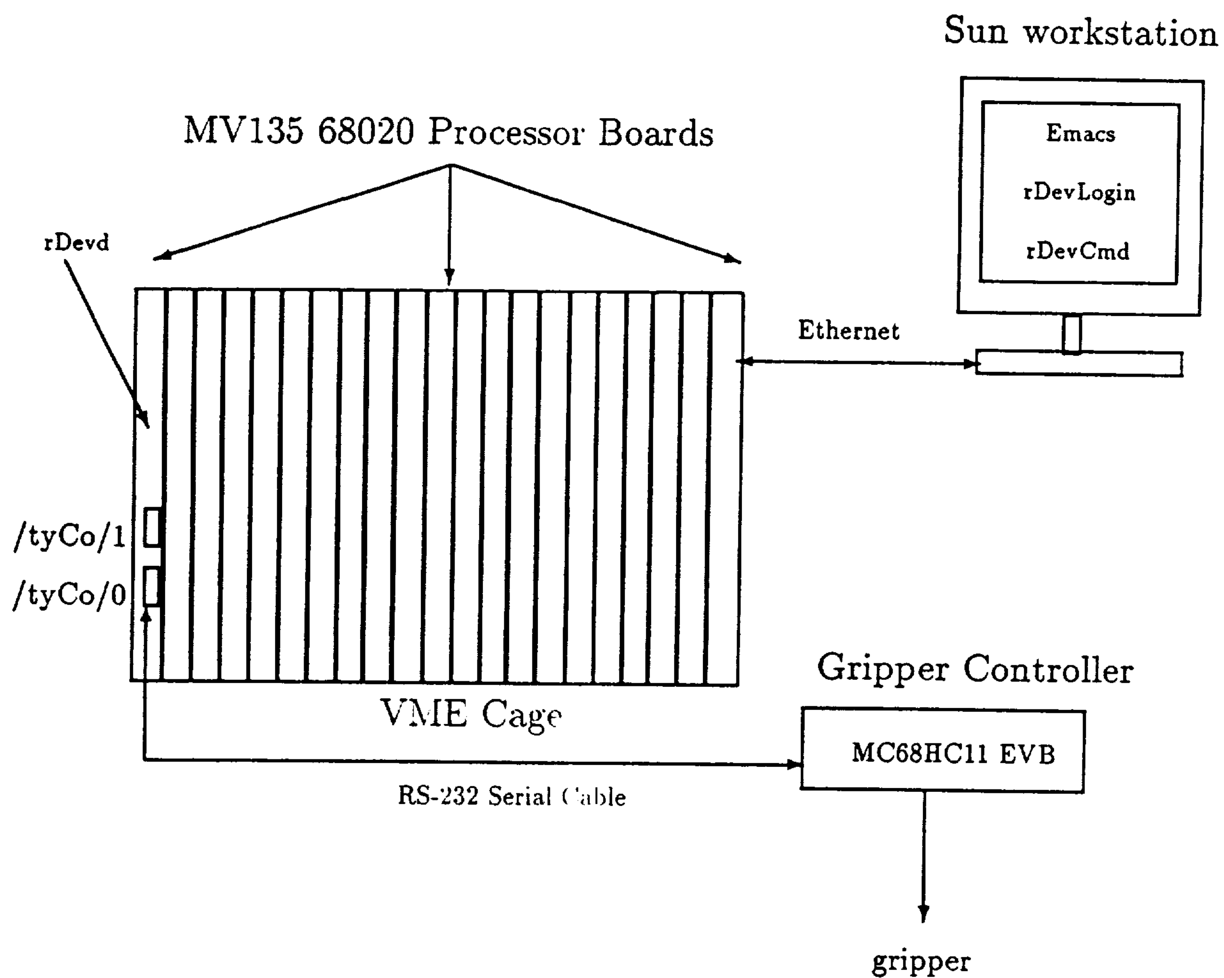

Figure 5.1: Hardware Configuration

```
rDevLogin hostname
```

in any *xterm* window.

5.3 Constructing an Imakefile

An *Imakefile* merely lists all the targets and dependencies which need to be generated. In this case, the *Imakefile* is trivial and only consists of one line.

```
AllTarget(prog.Y)
```

5.4 Assembling and Downloading

The resulting program may be compiled and downloaded by issuing the following command.

```
cmkmf prog.Y
```

A listing file named prog.lst is created. Lines with errors are marked with a letter from A-Z in the first column. Here is an example.

```
U 0000  7e 00 00          main    jmp     main2
  0003                            end


0000  main
```

The U stands for *undefined.* A complete listing of all error codes is given in the manual page [Colley 87].

CHAPTER 6
Tools

A variety of tools have been constructed to aid in developing software for the M68HC11 EVB. These tools provide simple ways to assemble, debug, and download programs.

6.1 rDevLogin - remote device login

6.1.1 Synopsis

`rDevLogin hostname`

6.1.2 Description

The **rDevLogin** client redirects terminal input and output to a **rDevd** server at the specified *hostname*. The **rDevLogin** client attempts to open a connection at port 8200 of *hostname*. Using the **select(2), read(2), and write(2)** system calls, a full duplex connection is implemented [Sun 88].

6.2 rDevd - remote device daemon

The network server **rDevd** runs on VxWorks to create a full-duplex connection between a socket and a serial port on the 68020 MV135.

6.2.1 Synopsis

rDevInit deviceName

6.2.2 Description

The **rDevd** server listens on two public ports and waits for incoming connections. When it accepts a connection on either port, **rDevd** provides a byte stream between the client and the device specified by **deviceName**.

The two ports are used in order to establish two independent paths to send data to the tty device. One port, symbolically named **TELPORT**, provides a full-duplex connection to the tty device. Another port, symbolically named **CMDPORT**, provides only a send path to the tty device. This allows **TELPORT** to be used as an interactive terminal interface to the tty device while **CMDPORT** can be used to send commands to the tty device. Figure 6.1 shows this arrangement.

The **select(2)** call is used to look for incoming connections or data. Any input event received by the **rDevd** server results in two actions.

- an *accept* action where a connection is established.
- a *transfer* action where a block of data is transferred.

The inputs detected by **select(2)** are mapped into the actions in Table 6.1. Both the *socket index* and *destination socket index* columns refer to indices into an array of file descriptors. When data to be read appears on one of the file descriptors in the array, either an *accept* or *transfer* action is taken using the *action* column. The array element pointed to by *destination socket index* is used to store the file descriptor of the new socket connection accepted or store the file descriptor of the socket where data is to be written.

6.3 rDevCmd

The network client **rDevCmd** runs on SunOS to send a string to the **rDevd** server. Normally a carriage return always follows the string unless the -n parameter is specified.

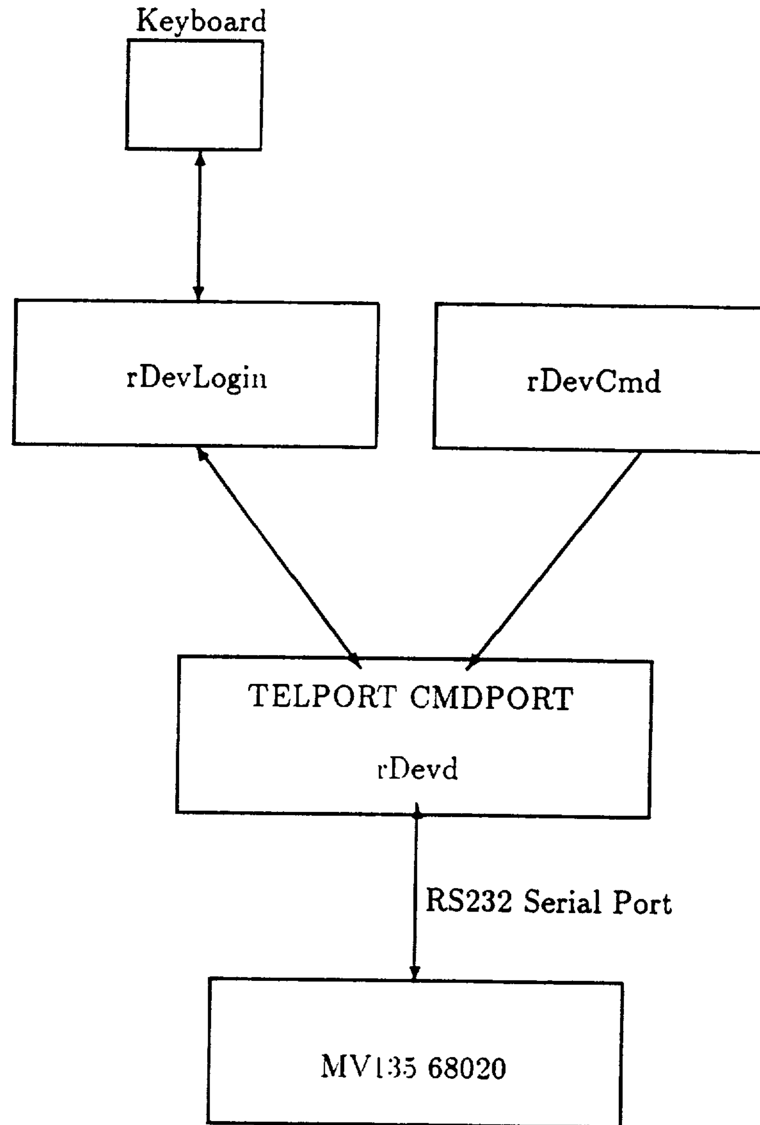

Figure 6.1: Network Connections

socket index	*action*	*destination socket index*
TELNETLISTEN	accept	TELNETMSG
TELNETMSG	transfer	DEV
CMDLISTEN	accept	CMDMSG
CMDMSG	transfer	DEV
DEV	transfer	TELNETMSG

Table 6.1: Action Table

The client **rDevCmd** uses the **connect(2)** system call to initiate a connection on a socket.

6.4 a611floatpp

The utility **a611floatpp** is a preprocessor which is designed to scan its standard input for occurrences of floating point numbers. If a floating point number is found and does not occur within a comment, it is expanded to its equivalent 3 byte hexadecimal representation in 68HC11 assembler. This preprocessor is used in conjunction with **a611** to allow the use of floating point numbers in 68HC11 source code using the floating point routines in the gripper controller software.

6.5 moto2moto

The utility **moto2moto** reformats Motorola S-records by reading in the original S-records and outputting S-records which are no longer than a specified length. This is needed because the S-records produced by the **int2moto** utility are too long in length to be read by the Buffalo monitor.

The program operates by reading in each S-record and storing each in a cell of a linked list. Each cell of the linked list is then scanned a second time. The object code for each cell is output until **maxBytes** are reached. A new S-record is started and more object code is output until no more object code remains in the cell. This

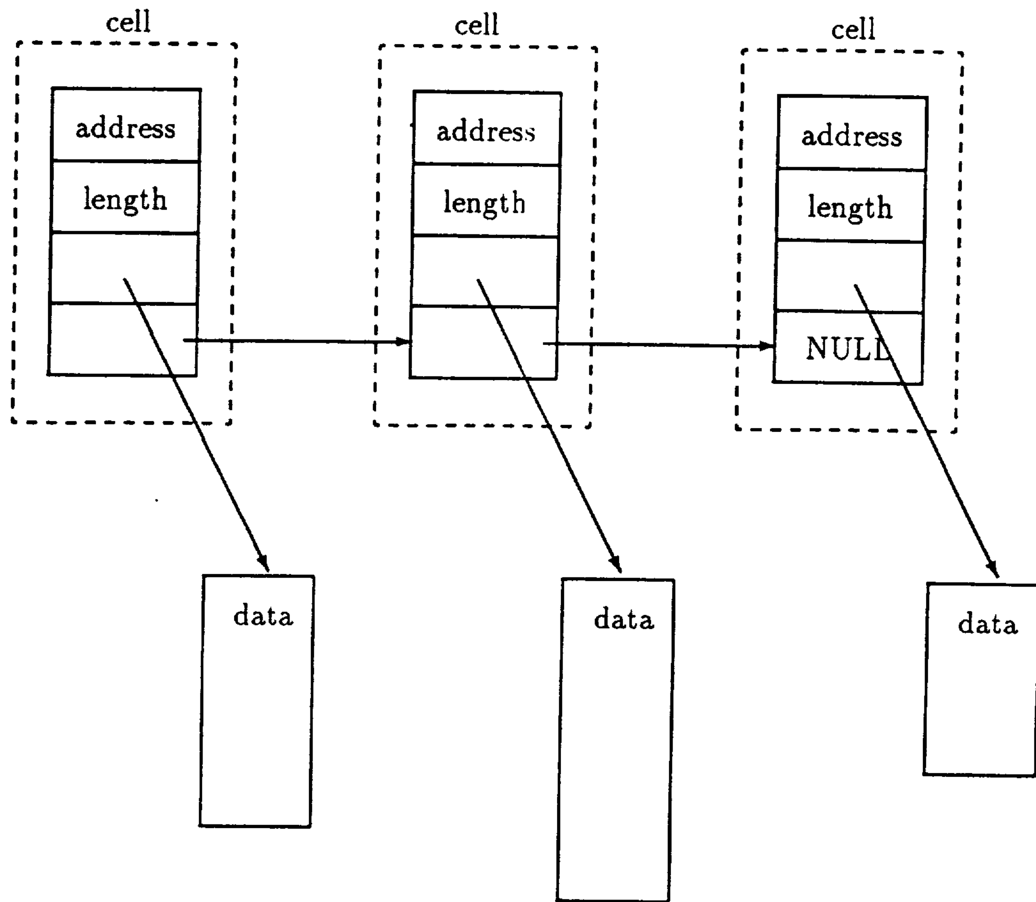

Figure 6.2: Software Architecture

linked list data structure is depicted in Figure 6.2.

Only S-records of type S0, S1, and S9 can be read by **moto2moto**.

6.6 motooffset

The utility **motooffset** offsets the base address of Motorola S-records. This is necessary for EPROM programmers which require a base address of 0x0000 in order to read in the S-records correctly.

For example, a program which is normally downloaded and tested in RAM may be located at 0xc000. The S-records would normally start at 0xc000. If this

program is to be downloaded from the EPROM, relocatable startup code must be added to copy the program from EPROM to RAM. The resulting object code is offset by -0xc000 [2] to make the first address 0x0000 so that the object code can be downloaded properly into the EPROM burner.

The operation of the **motooffset** utility is the same as the **moto2moto** utility except that the offset in the argument list is added to each address just before S-records are output from each cell.

6.7 installMan

The Bourne shell script **installMan** installs a manual page in a directory deriving the manual subdirectory and extension from the section header specified in the manual page itself. This is a convenient way to install manual pages.

6.8 manc

The Bourne shell script **manc** merely invokes the ordinary UNIX **man** command with the -M ./CIRSSE/installed/man option. This provides a short and easy way to obtain manual pages.

6.9 xmanc

The Bourne shell script **xmanc** sets the **MANPATH** environment variable to the ./CIRSSE/installed/man directory and then invokes the **xman** command. This provides a short and easy way to browse through manual pages in X windows.

[2] the two's complement of 0xc000

6.10 Public Domain Utilities

6.10.1 68HC11 Assembler

A public domain assembler was used to translate the 68HC11 assembler source to Intel formatted object code [Colley 87].

6.10.2 int2moto

Since the 68HC11 assembler outputs only Intel formatted object code, the assembler output must be passed through the **int2moto** utility to convert it to Motorola S-record format.

CHAPTER 7

Configuration Management

7.1 Motivation

There are two motivations for developing a configuration management scheme for software at CIRSSE.

- Makefiles are complicated since a diverse programming environment exists.

- Version control.

7.1.1 Complicated Makefiles

The development of software at CIRSSE presents a very diverse number of programming environments to the programmer.

For example, the gripper controller software resides on the 68HC11 EVB running the Buffalo monitor. It must also communicate to the outside world through a cable connected to a 68020 MV135 running VxWorks. The 68020 MV135 must also communicate to a SPARCstation host running SunOS. Software runs on all three of these platforms.

In addition, there are a number of compilers being considered to implement code at CIRSSE. Some of these are Sun's cc, GNU gcc, and GNU g++.

All this diversity complicates the construction of Makefiles. A much simpler method of constructing Makefiles has been developed.

7.1.2 Working Versions of Files

Version control is designed to make sure that working pieces of code are not lost while experimenting with new add ons to the code. This is accomplished by integrating SCCS into the configuration management scheme.

7.2 Imake Configuration Files

In order to make the task of system maintenance as easy as possible, **imake** has been used to minimize the amount of information necessary to perform all the necessary maintenance tasks. **Imake** also greatly simplifies the task of specifying rules and dependencies. Instead of designing and maintaining a separate *Makefile* in each directory which may require information such as common paths to be duplicated in each *Makefile*, a much shorter *Imakefile* is specified. This *Imakefile* consists of cpp macro calls and ordinary **make** rules, definitions, or dependencies. The ability of macro expansion allows *Imakefile's* to be several times shorter than *Makefile's* by isolating all configuration management details from the application programmer.

Configuration management schemes using **imake** also enable code to be developed either entirely within a single directory, distributed amongst several users or both without going through the hassle of updating duplicated configuration management related information in each *Makefile*.

In this configuration management scheme, there is only one *Imakefile* per directory. This *Imakefile* may be copied directly into the project directory and used as is without any modification. This greatly reduces the amount of effort required for maintenance of the *Makefile's* in an environment where software is not only being developed in separate directories owned by several different users but is also available in a project directory which is used for shared public access.

7.2.1 Files

There are several #include files. The #include files are separated so that different parts of the system can be added or deleted easily as needs change or if new software or hardware is added to the system.

- *Imake.rules* specifies all the generic rules.

- *Project.tmpl* specifies all the project dependent information such as directory specifics.
- *genclass.tmpl* specifies all the rules for producing container classes using the GNU libg++ class-generation facility.
- *LaTeX.tmpl* specifies all the LaTeX rules and definitions.
- *UNIX.tmpl* specifies all the UNIX related rules and definitions.
- *VxWorks.tmpl* specifies all the VxWorks related rules and definitions.
- *M68HC11.tmpl* specifies all the M68HC11 related rules and definitions.
- *Imake.tmpl* specifies all the rules and definitions common to every *Imakefile*.

7.2.1.1 Imake.rules

MakeSubdirs(subdirs) is a macro to descend the directories in the *subdirs* argument and extract an *Imakefile* and run **make** to create a *Makefile*. **Make** is then invoked using the same targets passed from the parent **make**.

addClean(targets) is a macro to add the list of files specified by *targets* to the list of files to be cleaned when performing **make** *clean*.

manInstall(manlist) is a macro to install man pages. The **cpp** C preprocessor is used on these man pages before actually installing them. This allows preprocessor directives such as **#include** to be used to included shared text between man pages to avoid unnecessary duplication. The argument *manlist* is a list of man pages, ie. manInstall(ls.man sed.man)

manSo(solinks,manpage) is a macro to create man pages which are sourced to a common manual page. An example of this is the **strings (3)** set of UNIX functions. The argument *solinks* is a list of man pages to be sourced to a common manual page. The argument *manpage* is the name of the sourced manual page, ie.

manSo(strcpy.man strdup.man,strings.man). The appropriate section is determined by the section field used in the .TH header of the file *manpage*.

installFile(flags,file,dir) is a generic installation macro to invoke the **install** utility with the given *flags* to install the file specified by *file* into the specified directory *dir*. Note that only one file may be specified, ie. *file* may not be a list of files. When using **installFile**, absolute paths should be avoided since these make the *Imakefile* dependent on where the source tree is located.

AllTarget(targets) is a macro to cause all *targets* to be built. Note that *targets* can be a list as well as an individual target.

LintTarget(incs,srcs) is a macro to run the **lint** processor on all C source files specified by *srcs*. *Incs* is a list of all #include files that are necessary for compilation. The *incs* argument is necessary in order for the rule to generate the appropriate dependencies.

7.2.1.2 Project.tmpl

Project.tmpl contains all the directory specific information. This information normally varies from project to project.

The **make** variable **PROJDIR** is the official project directory.

It is recommended that relative paths be used to reference any local configuration files so that programs can be moved to and from **PROJDIR** without changing any *Imakefiles*.

INSTALLDIR is the installation directory used to install all programs, scripts, and other target files.

CONFIGDIR is the configuration directory used to store all configuration management related files. These are all the files necessary to create *Imakefiles*.

7.2.1.3 UNIX.tmpl

UNIX.tmpl contains all the UNIX specific rules.

UNIXBinTarget(target,inclist.objlist) is a macro to link a group of cc object files specified by *objlist* into one executable. Explicit rules are not necessary to compile .c files into .o files. This is taken care of by the implicit suffix rules in */usr/include/make/default.mk*. *Inclist* is a list of #include files for the C sources. Their presence is necessary for a proper dependency. *Target* is the name of the executable file.

UNIXBinCCCTarget(target,inclist,objlist) is a macro which is the same as **UNIXBinTarget** except that *objlist* is the name of a group of g++ object files to be compiled into one executable.

UNIXLibTarget(target,inclist,objlist) is a macro which is the same as **UNIXBinTarget** except that *target* is the name of a library archive (.a) instead of an executable. The object files in *objlist* are combined to form the library archive specified by *target*.

UNIXBinInstall(binary) is a macro to install a UNIX binary executable.

UNIXLibInstall(library) is a macro to install a UNIX library archive.

UNIXHInstall(include) is a macro to install the #include file specified by *include*.

UNIXShInstall(script) is a macro to install a UNIX script.

7.2.1.4 VxWorks.tmpl

VxWorksBinTarget(target,inclist,objlist) is a macro to generate the rule to link the list of object files specified by *objlist* and to generate the proper #include dependencies from the #include files specified by *inclist*.

VxWorksLibTarget(target,inclist,objlist) is a macro which is exactly the same as **VxWorksBinTarget**. Its purpose is to logically separate object files

used as libraries from those object files which actually use those libraries.

Note that for either **VxWorksBinTarget** or **VxWorksLibTarget**, *target* must never be a file in *objlist*. This would create a circular dependency which would cause **make** to fail. To create a single object file from a single C file just leave the *objlist* argument empty.

For example, **VxWorksBinTarget(singleTarget.o, singleTargetInc.h,)**, will create *singleTarget.o* from *singleTarget.c*.

VxWorksBinInstall(binary) is a macro to install the object file specified by *binary*. Note that binary can only be a single file.

VxWorksLibInstall(library) is a macro to install the object file specified by *library*. Note that library can only be a single file.

VxWorksHInstall(include) is a macro to install the #include file specified by *include*. Note that *include* can only be a single file.

VxWorksShInstall(script) is a macro to install a VxWorks script.

7.2.1.5 M68HC11.tmpl

M68HC11.tmpl contains macros, rules and definitions to generate *Makefile's* for the 68HC11 attached to one of the serial ports on the MV135 running VxWorks. There are several macros in *M68HC11.tmpl* which are used only for constructing other macros within M68HC11.tmpl they should *NOT* be used any other way. These macros are **M68HC11asmToObj**, **M68HC11objToX**, and **M68HC11XToY**. These macros are used only to construct the implicit rules necessary to generate downloadable Motorola S-records for the 68HC11 Buffalo monitor.

M68HC11BinInstall(XFile) is a macro to install a Motorola S-record object file (.X). Note that there are implicit suffix rules for generating .X files.

M68HC11Clean(base) is a macro to add *base.obj, base.lst, base.X,* and *base.Y* to the list of targets to be cleaned. Note that *base* is a basename such as

"hello", not "hello.c". Note that there may be more than one base in the argument, ie. **M68HC11Clean(base1 base2)**.

M68HC11Window() is a macro to open an **xterm (1)** window which invokes **rDevLogin (1)** to connect **xterm (1)** to the 68HC11 EVB Buffalo monitor. The window may be created with the command **cmkmf Window**.

M68HC11Download(baselist) is a macro to download the .Y files specified in *baselist* to the 68HC11 EVB. Note that the names in *baselist* should not contain .Y. This extension is automatically added. The downloading can be accomplished by the command **cmkmf Download**.

7.2.1.6 LaTeX.tmpl

LaTeX.tmpl defines the implicit suffix rules to generate a .ps or .dvi file from a LaTeX .tex or xfig .fig file. In addition, *LaTeX.tmpl* defines macros.

LaTeXClean(base) is a macro to add *base.aux, base.log, and base.dvi* to the list of targets to be cleaned. Note that there may be more than one base in the argument, ie. **LaTeXClean(base1 base2)**.

7.2.1.7 Imake.tmpl

Imake.tmpl is a file that is the template for all *Makefile's* generated from *Imakefile*. All *Makefile's* generated from **imake** have the following targets:

- clean - clean all targets except installed targets
- lint - run the lint processor on all C source files
- all - generate all targets, but do not install anything
- install - generate all targets and install

The *clean* target is useful when the source code is working and installed targets are all that is needed. This reclaims disk space used by intermediate object files and other unnecessary files generated during the build.

The *lint* target is useful for checking all the C source code using the **lint** utility.

The *all* target is useful for testing if things compile correctly during editing and an installation is not desired because the code is not finished. Since the *all* rule does not perform any install, it may fail if some of the local targets depend on installed targets that have not yet been installed. In other words, it is not recommended that a **make** *all* be done on a whole directory unless it is certain that the required targets are installed.

The *install* target is useful for performing entire rebuilds or making sure that everything is up to date.

7.2.1.8 Directory Structure

- CIRSSE - Project directory
- CIRSSE/installed - installed files
- CIRSSE/installed/M68HC11 - installed 68HC11 files
- CIRSSE/installed/VxWorks - installed VxWorks files
- CIRSSE/installed/UNIX - installed UNIX files
- CIRSSE/installed/man - installed manual pages
- CIRSSE/src/config - configuration management info
- CIRSSE/src/apps - application (object code that calls "library functions"
- CIRSSE/src/lib - libraries
- CIRSSE/src/samples - sample Imakefiles

7.2.2 Caveats

The #include path is set using the -I option. -ICIRSSE/installed/UNIX/h is added to **CPPFLAGS** whenever any of the macros in *UNIX.tmpl* are used and -ICIRSSE/installed/VxWorks/h is added to **CPPFLAGS** whenever any of the VxWorks target macros in *VxWorks.tmpl* are used. Thus, if these macros are not used, the proper #include path is not added.

7.2.3 Bugs

You cannot mix macros from *UNIX.tmpl* and *VxWorks.tmpl*. You must maintain UNIX and VxWorks source code in different directories. This bug arises because there can only be one value of CPPFLAGS per *Makefile*.

7.2.4 See Also

In UNIX man pages **imake (1)**, **cpp (1)**, **man (1)**, **xman (1)** [Sun 88, manual pages] [X11 89, manual pages].

7.3 Imakefile

An *Imakefile* is merely a collection of **cpp** macros and **make** definitions and rules which are processed by **imake** to create a *Makefile* that will generate all the necessary commands in order to perform certain maintenance on software.

After constructing an *Imakefile*, it is possible to perform the following commands to do different types of maintenance tasks.

- cmkmf clean - clean all targets except installed targets
- cmkmf lint - perform lint checks on all code
- cmkmf all - build all targets but don't install anything

- cmkmf install - build and install all targets

Note that care must be taken when using *cmkmf all* because some targets may depend on other *installed* targets which will *not* be installed by *cmkmf all.* This kind of build is only useful if all installed targets needed are already in place.

The easiest way to learn what macro calls to put in *Imakefiles* to perform different tasks is to look at some examples. It is easiest to copy an existing *Imakefile* and edit it.

7.3.1 VxWorks Library

Here is an example of a simple *Imakefile* to create all the necessary rules and dependencies for a VxWorks library consisting of only one object module. No *.c* files appear anywhere except in the *LintTarget* macro in the *Imakefile* because **make** automatically checks for *.c* files automatically from the implicit suffix rules in */usr/include/make/default.mk.* This *Imakefile* can be found in
./src/samples/VxWorks/lib/single/Imakefile.

```
/**/# Comments prefixed by C comment delimiters like this line are
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */

AllTarget(sampleSLib.o sampleSLib.v)

VxWorksLibTarget(sampleSLib.o , sampleSLib.h, )
VxWorksLibInstall(sampleSLib.o)
```

```
VxWorksHInstall(sampleSLib.h)
VxWorksShInstall(sampleSLib.v)
LintTarget(sampleSLib.h , sampleSLib.c)
```

Note the comma at the end of the second argument. The third argument is blank.

If there is more than one object file in a VxWorks library then the *Imakefile* would look like this. This *Imakefile* can be found in *./src/samples/VxWorks/lib/multiple/Imakefile.*

```
/**/# Comments prefixed by C comment delimiters like this line are
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */

AllTarget(sampleMLib.o sampleMLib.v)

VxWorksLibTarget(sampleMLib.o,sampleMLib.h,functions.o subroutines.o)

VxWorksLibInstall(sampleMLib.o)
VxWorksHInstall(sampleMLib.h)
VxWorksShInstall(sampleMLib.v)
LintTarget(sampleMLib.h , functions.c subroutines.c)
```

Note that for **VxWorksLibTarget** the first argument, the library, must be unique from any of the object files in the second argument, ie. the names of the input and output files must all be unique. This is necessary to avoid circular dependencies.

7.3.2 VxWorks Binary

The *Imakefile* for VxWorks binaries is no different from the *Imakefile* for VxWorks libraries. The only functional difference is that the targets are put in a different directory during installation.

Here is an example of an *Imakefile* to create all the necessary rules and dependencies for a VxWorks binary when dealing with only one object file. No *.c* files appear anywhere except in the **LintTarget** macro in the *Imakefile* because **make** automatically checks for *.c* files automatically from the implicit suffix rules in */usr/include/make/default.mk.* This file can be found in
./src/samples/VxWorks/bin/single/Imakefile

```
/**/# Comments prefixed by C comment delimiters like this line are
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */

AllTarget(sampleSBin.o sampleSBin.v)

VxWorksBinTarget(sampleSBin.o , sampleSBinConst.h sampleSBinDecl.h, )
VxWorksBinInstall(sampleSBin.o)
VxWorksHInstall(sampleSBinConst.h)
VxWorksHInstall(sampleSBinDecl.h)
VxWorksShInstall(sampleSBin.v)
LintTarget(sampleSBinConst.h sampleSBinDecl.h , sampleSBin.c)
```

Note the comma at the end of the second argument. The third argument is

blank.

If there is more than one object file in the VxWorks binary then the *Imakefile* would look like this. This *Imakefile* can be found in *./src/samples/VxWorks/bin/multiple/Imakefile.*

```
/**/# Comments prefixed by C comment delimiters like this line are
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */

AllTarget(sampleMBin.o sampleMBin.v)

VxWorksBinTarget(sampleMBin.o,sampleMBinConst.h sampleMBinDecl.h,main.o io.o)
VxWorksBinInstall(sampleMBin.o)
VxWorksHInstall(sampleMBinConst.h)
VxWorksHInstall(sampleMBinDecl.h)
VxWorksShInstall(sampleMBin.v)
LintTarget(sampleMBinConst.h sampleMBinDecl.h , main.c io.c)
```

7.3.3 UNIX Library

Here is an example of a simple *Imakefile* to create all the necessary rules and dependencies for a UNIX library. consisting of only one object module. No *.c* files appear anywhere except in the *LintTarget* macro in the *Imakefile* because **make** automatically checks for *.c* files automatically from the implicit suffix rules in */usr/include/make/default.mk*. This file can be found in *./src/samples/UNIX/lib/Imakefile*

```
/**/# Comments prefixed by C comment delimiters like this line are
```

```
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */



AllTarget(sampleLib.a)


UNIXLibTarget(sampleLib.a , sampleLib.h , functions.o subroutines.o)
UNIXLibInstall(sampleLib.a)
UNIXHInstall(sampleLib.h)
LintTarget(sampleLib.h , functions.c subroutines.c)
```

7.3.4 UNIX Binary

Here is an example of an *Imakefile* to create all the necessary rules and dependencies for a UNIX binary. consisting of only one object module. No *.c* files appear anywhere except in the *LintTarget* macro in the *Imakefile* because **make** automatically checks for *.c* files automatically from the implicit suffix rules in */usr/include/make/default.mk*. This file can be found in *./src/samples/UNIX/bin/Imakefile*

```
/**/# Comments prefixed by C comment delimiters like this line are
/**/# copied into the Makefile generated by imake.

/*
 * Normal comments like this do not show up in the Makefile
 */
```

```
AllTarget(sample)

UNIXBinTarget(sampleBin , sampleConst.h sampleDecl.h , main.o input.o output.o)
UNIXBinInstall(sampleBin)
UNIXHInstall(sampleConst.h)
UNIXHInstall(sampleDecl.h)
LintTarget(sampleConst.h sampleDecl.h , main.c input.c output.c)
```

7.3.5 Making subdirectories with no local targets

There may be times when there are subdirectories which themselves contain *Imakefiles* which generate more targets.

Here is an example of an *Imakefile* that generates all targets in the subdirectories. This file can be found in *./src/samples/Imakefile.*

```
MakeSubdirs(UNIX VxWorks localTarget custom)
```

7.3.6 Making subdirectories with local targets

Sometimes local targets exist in a directory in addition to subdirectories. In this case all the targets in the subdirectories must be built before the targets in the local directory are built.

Consider the following directory hierarchy.

```
.:
Imakefile b/
a/ sampleLocalTarget.c

a:
```

```
Imakefile sampleSubdirATarget.c


b:

Imakefile sampleLocalTargetB.c
```

Targets in directories *a* and *b* must be built before the local target *sampleLocalTarget* is built.

The *Imakefile* to accomplish this is shown below. This file can be found in *./src/samples/localTarget/Imakefile*

```
MakeSubdirs(a b)


/*
 * This local target is built after
 * the subdirectories a, and b are built
 *
 * The standard  targets for non-leaf directories are
 * clean lint all install.  They cause all the
 * corresponding targets in subdirectories a and b
 * to be built.
 *
 * The targets clean.local neat.local lint.local
 * all.local and install.local are built after the
 * subdirectories are built.
 */


AllTarget(sampleLocalTarget)


UNIXBinTarget(sampleLocalTarget,,sampleLocalTarget.o)
```

```
UNIXBinInstall(sampleLocalTarget)
```

The subdirectories beneath the *localTarget* directory merely contain ordinary *Imakefiles* which build targets normally. They are shown below for completeness. This *Imakefile* for subdirectory *a* can be found in *./src/samples/localTarget/a/Imakefile.*

```
/*
 * This target is built before the directory above.
 *
 * The standard targets for leaf directories are
 * clean lint all and install
 *
 * Note there are no local include files so the second argument to
 * UNIXBinTarget and the first argument to LintTarget are empty.
 */

AllTarget(sampleSubdirATarget)

UNIXBinTarget(sampleSubdirATarget,,sampleSubdirATarget.o)
UNIXBinInstall(sampleSubdirATarget)
LintTarget(,sampleSubdirATarget.c)
```

This *Imakefile* for subdirectory *b* can be found in *./src/samples/localTarget/b/Imakefile.*

```
/*
 * This target is built before the directory above.
 *
 * The standard targets for leaf directories are
 * clean lint all and install
```

```
 *
 * Note there are no include files so the second argument to
 * UNIXBinTarget and the first argument to LintTarget are empty.
 */

AllTarget(sampleSubdirBTarget)

UNIXBinTarget(sampleSubdirBTarget,,sampleSubdirBTarget.o)
UNIXBinInstall(sampleSubdirBTarget)
LintTarget(,sampleSubdirBTarget.c)
```

7.3.7 Making local targets only

When code is being developed in very large source trees, sometimes only a local build is desired because traversing the entire tree is too time consuming. For this reason, there is a way to build only the local targets in a non-leaf directory. Instead of using the *clean. lint, all,* and *install* targets which compile all the subdirectories *and* the local targets, you can use the *clean.local, lint.local, all.local,* and *install.local* targets to build the targets in the local directory only.

Be very cautious when doing this since building things in this manner may cause the subdirectories to be out of date with respect to the local targets. For this reason, it is recommended that a *cmkmf clean* and *cmkmf install* be done after you are finished working on a particular directory to make sure all the targets from that node down are all up to date.

7.3.8 Custom rules and dependencies

There may be cases where none of the predefined macros are adequate to serve your needs. Since **imake** is merely a preprocessor to **make,** you are free to

incorporate whatever **make** rules and definitions you desire.

Since local targets may have a suffix of *.local* if the current directory is a leaf directory, the following **make** variables have been defined so that the standard targets may still be defined in a consistent manner. They are merely set to the values of the standard targets, but are suffixed by *.local* if *MakeSubdirs()* appears at the beginning of the *Imakefile*.

- CLEAN_TARGET - defined as clean or clean.local
- LINT_TARGET - defined as lint or lint.local
- ALL_TARGET - defined as all or all.local
- INSTALL_TARGET - defined as install or install.local

Use relative paths so that your *Imakefile* isn't dependent on where the source tree is installed. For example, use ../progdir/neatProg instead of /home/userdir/projdir/progdir/neatProg.

Here is an example of an *Imakefile* that contains custom rules and dependencies. This file can be found in *./src/samples/custom/Imakefile*.

```
/*
 * A customized Imakefile
 */



AllTarget(a b c)



a:
echo "Making target a"
```

```
touch a

b:
echo "Making target b"
touch b

c:
echo "Making target c"
touch c

/*
 * These make variables are defined as clean all install uninstall and
 * lint if the macro MakeSubdirs() is NOT used, ie. the current
 * directory is a leaf directory.  Otherwise these make variables are
 * defined as clean.local all.local install.local and uninstall.local.
 */

$(CLEAN_TARGET)::
echo "Cleaning targets"

$(ALL_TARGET)::
echo "Making special all targets"

$(INSTALL_TARGET)::
echo "Installing targets"

$(LINT_TARGET)::
```

```
echo "Making lint targets"

addClean(a b c)
```

CHAPTER 8

Conclusions

The VxWorks gripper library functions isolate the application layer from the details of the gripper protocol so that future changes in gripper controller parameters do not cause invalidate large portions of code.

The gripper controller software provides a simple command interface to the 68HC11 EVB and enables the gripper to either "grab" objects which come between the fingers of the gripper or unconditionally servo at a position or force set point.

The software development environment enables programs to be compiled and downloaded with one simple command. This environment can be used to maintain the gripper controller software and serve as a basis for developing any new software for the 68HC11 EVB.

LITERATURE CITED

[Colley 87] *68HC11 Cross-Assembler (Portable) Version 0.1*, William C. Colley III, 1987.

[Motorola 89] *M68HC11 Reference Manual*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Motorola 86] *M68HC11EVB Evaluation Board User's Manual*, Motorola Inc., 1986.

[Sun 88] *SunOS 4.0 Reference Manual*, Sun Microsystems Inc., Mountain View, California, 1988.

[WRS 90] *VxWorks Volume 2 version 5.00*, Wind River Systems Inc., Emeryville, California, 1990.

[X11 89] *X Window System Version 11 Release 4*, Massachusetts Institute of Technology, 1989.