An Extensible Microcontroller and Programming Environment

by

Alexandra Sara Theres Andersson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by.....
David P. Cavallo
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Extensible Microcontroller and Programming Environment

by

Alexandra Sara Theres Andersson

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we designed and implemented an extensible microcontroller and a Scheme compiler. We hope that by providing students a with portable, extensible computational device capable of real world interaction through sensors and actuators they will conceive of, and implement, projects that teaches more than traditional, book based learning can provide. Programming and hardware development is often written off as too difficult for both students and teachers. But when people are provided with open ended devices and tools to develop them into artifacts that are personally meaningful to them they often exceed all expectations for what someone with no formal technical background can do. By writing technology off as too difficult to understand we rob students of the chance they might otherwise have of developing creativity in the area of technical design.

Thesis Supervisor: David P. Cavallo

Acknowledgments

thank you

Contents

Chapter 1

Introduction

Microcontrollers provide an interesting oppertunity in enhancing learning environments by offering students small, portable, low cost, programmable devices capable of real-world interaction. This thesis investigates how one might design hardware and software that students without technical background can learn to use. The device should in particular not just support one project, or even one type of projects, but be extensionable enough on both the hardware and software side to handle the desired task.

1.1 Motivations

The importance of technology for the education of middle- and high-school students has certainly been embraced by the educational and political establishment. The availability of computational facilities in public schools have proliferated, but to what end? The availability of technology alone does not guarantee that the student will take away any useful knowledge or technical skills. In many cases computers are simply used to provide students drill in subjects they would otherwise have learned on paper. The Future of Learning group have run many workshops in schools around the world trying to do something more interesting with technology. Microcontrollers have been a central component in many of these projects. This thesis attempts to design hardware and software around a microcontroller to allow for arbitrary future

extensions. Both software and hardware was designed with extendibility foremost in mind.

1.2 Discussion of the suitability of Scheme

We would like students to learn programming at earlier ages. But more than learning how to program we would like for the to develop computational thinking skills. As a beginner to programming it is easy to get lost in the details of the language, and attempt to write early, and therefore easy, programs through a trial and error process that teaches how to write particular pieces of code without quit understanding exactly why anything works. An understandable computational model is of great importance in learning the skill of computational thinking. Instead there is a tendency to dumb a system down for use by students on the assumption that they will otherwise be unable to use it. One example of this is the evulation of the programming language LOGO. At its conception LOGO was a real programming language with plenty of power. Fearing that it might prove to difficult to use, power was sacrificed in favor of simplicity. The language is now useless except to write trivial, short programs. A more useful language would have a low threshold to allow beginners to learn while doing, as well as a high ceiling, providing advanced students not only with plenty of computational power, but also with the ability to extend the language as they see fit. Scheme was chosen as the language to adapt for use in this project for these reasons.

Scheme was designed as a teaching language and has been used in MIT's introductory programming course for many decades. It has a well developed computational model. Its syntax is easy and the number of primitives to learn is few. At the same time it is a very powerful language. It is also relatively easy to understand how the language compiles to PIC assembly, at least if one ignores the details of implementations of primitives and memory management. The compiler developed here is derived from the compiler in Chapter 5 of Structure and Interpretations of Computer Programs. It can be understood separately from the specifics of primitive implementation and memory management issues. With some knowledge of PIC assembly, software

support of new hardware functionality can easily be added to the compiler.

1.3 Description of hardware extensibility

Whenever one designs a board one might implicitly exclude certain use because of inherent hardware limitations. We made extensionability of the hardware a primary concern in the design of this device. The base board comes with 19 controller I/O pins that can be used to extend the hardware. We have also designed some template extension boards to demonstarte extensionability and describe two different multiplexing protocols to further extend the use of these pins. It is hoped that this type of extensionability will promote use of the board beyond the intent of the author.

Chapter 2

Hardware Design

2.1 Microcontroller selection

Peripheral Interface Controller (PIC) microcontrollers are cheap and incorporate many useful peripheral features on the chip, such as a 10 bit analog to digital converter, a Universal Synchronous Asynchronous Receiver Transmitter (USART) module, and a Inter Integrated Circuit (I^2C) module. The PIC18F452 from Microchip[3] was selected over other PICs primarily for it 8bit hardware multily module which will significantly speed up multiplication and division algorithms. Useful peripheral features are the USART module used for serial communication with PCs, and the I^2C module for communication with peripheral chips, such as the EEprom and Real Time Clock (RTC).

2.1.1 Programming

The board is set up for In Circuit Serial Programming (ICSP). We can use standard PIC programming tools, i.e. MPLAB, to program the device in assembly or C. The board connects to the PIC programmer via 5 pins on the chip, made available through a connector on the board. Two of these pins, Vdd and MCLR, need to be isolated from the circuit for the programming phase becausae of excessive capacitance on these pins in regular operation. This isolation achieved by a Double Pole Double Throw

(DPDT) switch.

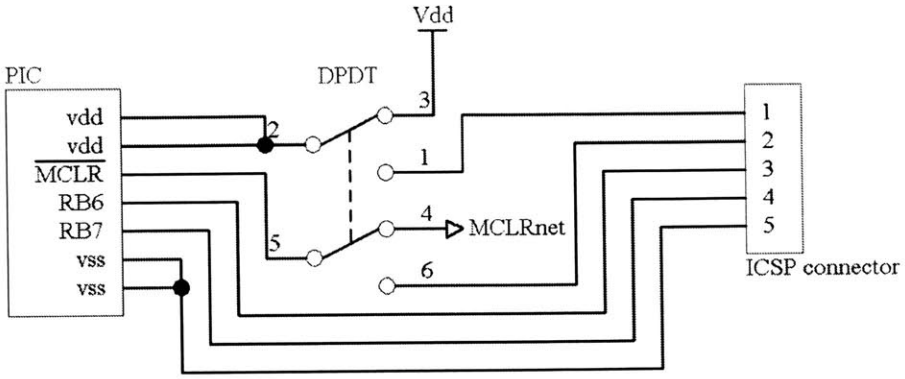

Figure 2-1: In Circuit Serial Programming circuit

2.2 Other hardware selection

2.2.1 Real Time Clock

A battery backed up RTC[6] is provided for convenient real time readout. When the board is not powered it is powered off its backup battery to accuratly keep time. It is interfaced to the PIC through the I^2C bus.

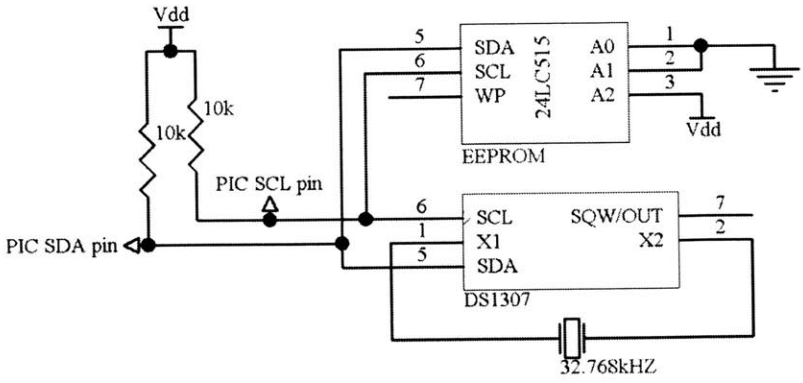

Figure 2-2: I^2C bus

2.2.2 Power Management

LT1579[2], a 5V dual source regulator was selected for power managment. The primary power source is 9V DC adaptor input, secondary source is 4x1.5V battery pack. The chip has several nice features: low dropout voltage, if both sources are present it draws power only from primary source. It has an output pin that gets pulled low when voltage starts to drop out of regulation. This pin can sink enough current to allow for directly connecting an LED to this pin, providing convenient low battery indication. The board is intended to operate on standard alkali batteries.

2.2.3 PCB layout

The PIC and peripheral chips were selected in surface mount packages for smaller board size. A smaller board means a cheaper board. Some components (resistors and battery backup for RTC) are on the bottom of the PCB to shrink size further.

2.3 Extensions

The board was designed to be easily extendible. 19 Input/Output (I/O) pins are provided as extension pins. These pins are all of ports B and D, and three pins from port C. Several protocols for multiplexing these pins have been developed.

2.3.1 Parallel Addressing and Data Transfer

If we need no more than 8 extension board and no more than 16 pins per board we would choose 3 pins for addressing the boards, and the remaining 16 for data transfer. Addressing is done by comparing voltages on address pins with board address using the CD74HC85 [4] 4-bit comparator. In its simplest version we just pass the 16 data lines through two 8-bit bi-directional bus switchs, SN74CBT3245A.[5] With this design we can only drive outputs on the extension board while it is addressed, when we address a different board all outputs will become high impedance on the old one. If we need to drive some output pins while the board is not addressed we designate

some pins as outputs in hardware and put a latch on them. The latch is transparent when the board is addressed, sending all signals through to the output side. When the board is de-addressed the voltages are latched and will hold their values until the board becomes addressed again.

Extension board example 1: Parallel Addressing, no latches

This template extension board uses the three port C pins for addressing and the port A and B pins for data transfer. Two bi-directional bus switches provides a data path when the board is addressed, and high impedance on both sides of the swith when it is not.

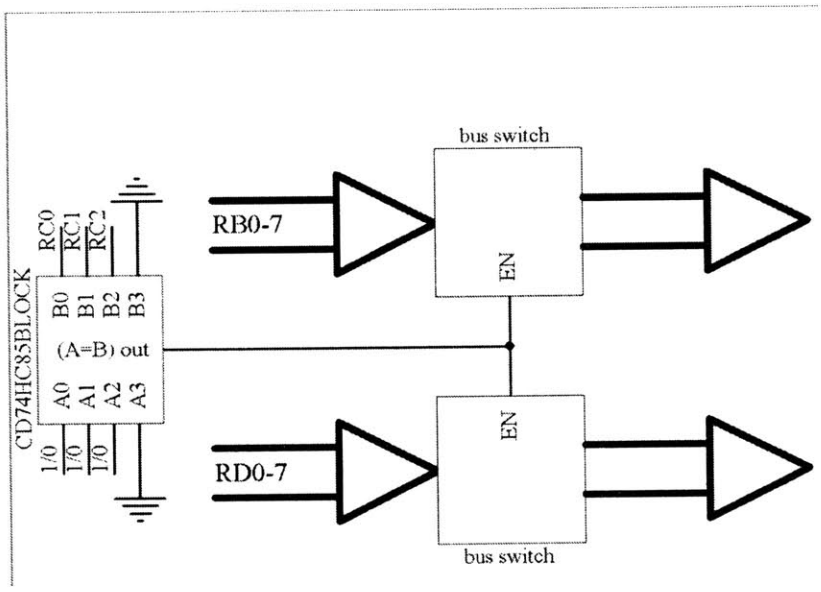

Figure 2-3: Parallell addressed extension board

Extension board example 2: Parallel Addressing, latch on one data byte

Here we demonstrate the use of output latches to drive devices on an extension board while it is not addressed.

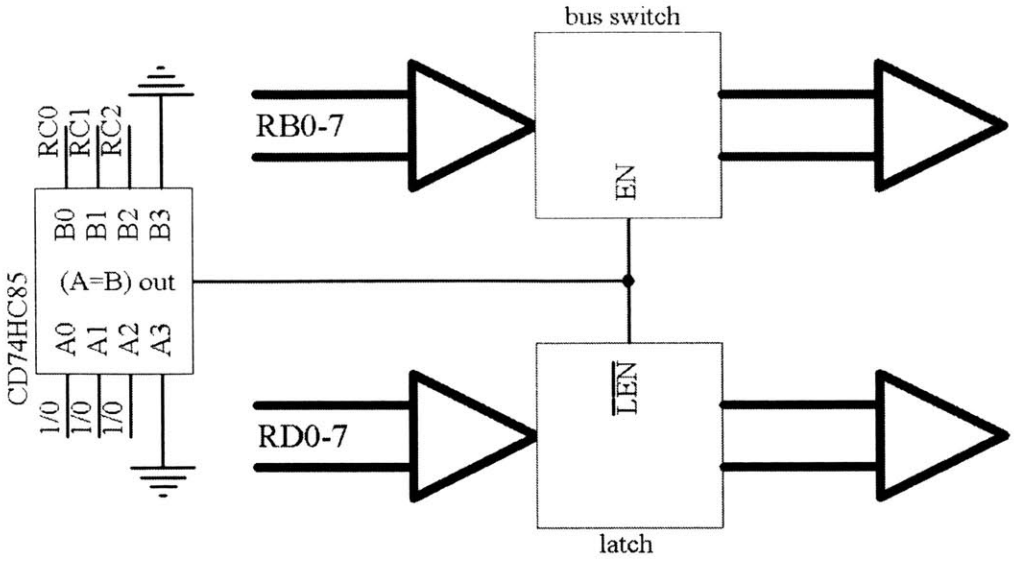

Figure 2-4: Extension board with latches

Extension board example 3: Serial Addressing, no latches

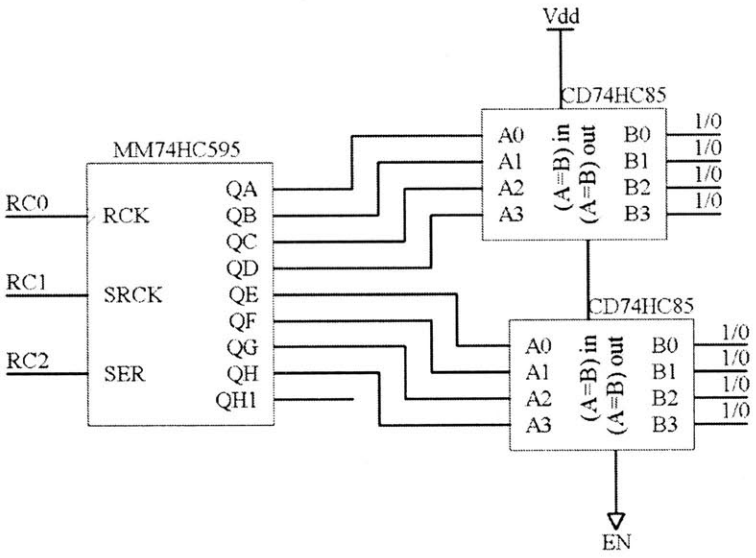

Figure 2-5: Serially addressed extension board

Chapter 3

Compiler Design

We would like to provide a Scheme compiler and interface to the controller board. Compiling a high level language such as Scheme to a small device such as the PIC has some challenges. The processor speed is only 40MHz, limiting our execution speed as compared to a PC. For the type of computation this board will support that will not be a significant problem. The user of the board needs to be aware of the processor speed, and the limitations it imposes, to avoid costly operations, such as floating point arithmetic whenever possible. A more severe limitation is the small size of data memory available on the PIC. We have 1kByte available in data memory after reserving memory for low level PIC opertions, such as primitive implementations and garbage collection. The PIC has much more program memory available, 32kByte, but moving data into program memory is an expensive operation. The processor will stall for about 2ms to write 8 bytes to program memory. The serial EEprom on the board has an internal write time of about 5ms for 64bytes, and a transfer time of 1.5ms. Not only is this faster than the on-chip program memory, execution can also continue during the internal write time, as well as during the data transfer using interrupts. There is still a limitation on EEprom access during this time makeing the solution non-ideal. A better approach would be to use off chip memory with faster write time. Those are inconvenient because they necessitate the implementation of a separate serialization scheme to interface to their parallel address/data lines, or we must give up our hardware extension lines. Future development of the board would

include further study of the memory implementation in an attemet to find a quicker scheme while maintaining extension lines on the PIC. This design will use a serial EEprom designed to run off the I^2C bus, as described in the hardware section.

Because of the small amount of memory available to us we have chosen to compile only a subset of the scheme implementation given in the Revised(4) Report on the Algorithmic Language Scheme, which is the language implementation our version is based on [1] A specification of the language of our compiler is given in Appendix A.

3.1 Memory managment

3.1.1 Addressing memory

We have a total of 64kByte available to us in the EEprom. The selected garbage collection scheme is of stop and copy nature, thus permitting us use of half the memory at any instant. Memory will be allocated to scheme code in 16bit words, leaving us with 14bits/address in the EEprom. Objects created at compile time will be located in program memory on the PIC. We thus extend our addresses to 15bits to allow for addressing both program and data memory. This size was chosen primarily to allow for 4-byte pairs. We anticipate that much of our memory allocation will be in pairs, as this is the nature of Scheme programs, allowing a relatively small pair size is thus of importance on a system with limited memory, such as this.

3.1.2 Paging

To optimize for speed in our slow memory transfer to off chip memory we have chosen to page the EEprom memory into twenty 64-byte pages. The page size was chosen based on teh EEprom page write size. We use sixty registers in datamemory for low level memory managmeny. Registers PSAkH, PSAkL, and PSAkme keep track of the k^{th} page in data memory. PSAkH/PSAkL hold the EEprom address corresponding to the k^{th} page. Since each page is only 64 bytes the 6 low order bits in PSAkL hold no information pertaining to the page in use. Rather, they will hold the value of the first

modified location in data memory on the page. PSAkme will hold the last modified location. When a page is read into data memory from the EEprom these registers will indicate that nothing has been modified. As the program modifies locations in datamemory the registers will be updated to reflect these changes. When a page is swapped out of data memory we write only the range of locations indicated by these two registers, which might be none.

Decoding an address

When we wish to access a value corresponding to a memory address we have to interface it through the paging scheme set up above. The address we start with will reference either a location in program memory or data memory, as determined by the high bit of the address. An assembly subroutine, ADDDEC, has been developed for this purpose. If it is a program memory address we can just returned whatever is at that location using operations native to the PIC, in this case we just return that address from ADDDEC. If we are specifying a location in data memory ADDDEC will check if the corresponding page is in program memory by chacking PSAkH and PSAkL. If the page is already in data memory we compute the data address from the k of the corresponding PSAkH. If the page is not in data memory we will have to fetch it from the EEprom before computation can proceed. ADDDEC will take care of this, and ultimatly return the PIC data memory address associated with that value. The caller at that point has an address that can be accessed by PIC primitives.

Reading a page from EEprom

A low level assembly routine for reading a page from EEprom into data memory has been developed and debugged. It utilizes the hardware I^2C module on the PIC to issue commands to the EEprom and read data values. Below is a pseudo code outline of this routine, the complete routine can be found in Appendix B.

```
check write-in-progress bit in GCstatus register
if set, loop until clear
```

```
generate start condition
send slave address specifying a write operation
send address to start reading at
generate a repeated start condition
send slave address specifying a read operation
generate an acknowledge following each recieved byte
after 64 bytes have been received, generate a stop condition
        rather than an acknowledge
```

This routine has been tested and timed. The read time for 64bytes is approximatly 2ms. We must check if a write is in progress before we start, or we will interfere with the interrupt driven write routine and cause unpredictable memory behaviour. The read routine does not use interrupts since continuing execution is pointless when we are waiting for a value we need to be moved to data memory.

Writing a page to EEprom

This complementary low level assembly subroutine implements the necessary instructions for writing a page to the EEprom from data memory. The write routine will be invoked after we have started using the last free page in data memory, in hope that the write will be done by the time we need another free page. We write the oldest page at this point, and then update our free page pointer to the newly freed data memory page. This routine does use interrupts since execution can continue while we are transfering data, as long as we don't need yet another free page. We set the write-in-progress bit in the GCstatus register at the begining of a write to keep other routines from interefering. Two registers; WRcontH, and WRcontL; keep track of where we should continue our write operation at an interrupt.

```
check write-in-progress bit in GCstatus register
if set, loop until clear      ;only allow one page page write
enable \ic interrupts
generate start condition
```

```
Program counter + 4 -> WRcont
return
send slave address specifying a write operation
Program counter + 4 -> WRcont
return from interrupt
send address to start writing at
Program counter + 4 -> WRcont
return from interrupt
send first byte
Program counter + 4 -> WRcont
return from interrupt
send second byte ...

...
after last byte, send stop condition
disable \ic interrupts
return from interrupt
```

3.1.3 Garbage Collection

We use a stop and copy algorithm. Since our memory is paged into PIC data memory from EEprom it is to our advantage that values occur as densely in pages as possiblem, with values of the same data structure in a small physical memory range. To this end we use stop and copy to allow for data compactification. We will first attempt to garbage collect all the pages currently in data memory, pushing addresses not in data memory onto a wait stack. Only when all our addresses to copy are not in data memory will we swap an old page for a new one. The stop and copy garbage collection algorithm used is standard for scheme. The root set is the current environment and the time garbage collection is initialized, everything reachable from the bindings in that environment, and its chain of ancestors, is live and will be copied to new locations. As of the writing of this thesis the garbage collector has not been fully debugged.

3.1.4 primitive data type encodings

We refere to data by its address in memory. In the case of pairs, symbols, and procedures this address is also the value of the data object. In the case of numbers and strings we need to dereference the address to get the value. We must always dereference the address to check the type of a primitive.

symbols

We have restricted our version of scheme to only allow symbols defined at compile time. Symblos will therefore be created only by the following special forms: quote, lambda, and define. Since symbols are guaranteed to be unique we store the printed representation of each symbols exactly once. Any place that symbol is used them simply references that loation in memory. Since all symblos are known at compile time we place their printed representations in program memory to save valuable data memory space. We further restrics symbols to occupy a restricted part of program memory, addressable by 12bits, to allow for smaller environments. In Table 3.1 we see the symbol encoding. The last byte is a byte of zeros, indicating the end of this symbol. This is the encoding for an even number of characters in the printed representation of the symbol. If the number of characters was odd we would have two bytes of zeros deliminating the symbol.

F	E	D	C	B	A	9	8	7-0
0	0	0	1	x	x	x	x	$c_1 7$-$c_1 0$
				...				
$c_k 7$	$c_k 6$	$c_k 5$	$c_k 4$	$c_k 3$	$c_k 2$	$c_k 1$	$c_k 0$	0

Table 3.1: Symbol implementation

pairs

We want to allocate pairs as efficiently as possible since scheme expressions tend to produce a lot of them. A pair needs to be able to hold two addresses, which requires 30 bits with 15bits per address. We also need a tag to identify pairs, and a bit to hold

garbage collection information. We thus tag pairs with one bit, and reserve another bit for the garbage collector to use. The table below shows this pair implementation. a16-a1 is the address of the car, b16-b1 is the address of the cdr.

F	E	D	C	B	A	9	8	7-0
1	a16	a14	a13	a12	a11	a10	a9	a8-a1
GC	b16	b14	b13	b12	b11	b10	b9	b8-b1

Table 3.2: Pair implementation

environments

Environments is another one of scheme's datatypes that is heavily used. We would like a memory efficient implementation of environments as well. An environment is a list of *bindings* and an *enclosing environment.* Each binding has a reference to a *variable* and a *value* associated with that variable. Syntactically a variable is a *symbol.* We therefore know that it will be located in specific part of program memory, addressable with 12 bits. Further, we know that the enclosing environment is located in data memory, since no environments are created at compile time. We thus need only 14 bits to address the enclosing environment. Table 3.3 gives the environment encoding for an environment with k bindings.

F	E	D	C	B	A	9	8	7-0
0	0	1	GC	$v_1$12	$v_1$11	$v_1$10	$v_1$9	$v_1$8-$v_1$1
1	$b_1$16	$b_1$14	$b_1$13	$b_1$12	$b_1$11	$b_1$10	$b_1$9	$b_1$8-$b_1$1
				...				
1	x	x	x	v_k12	v_k11	v_k10	v_k9	v_k8-v_k1
1	b_k16	b_k14	b_k13	b_k12	b_k11	b_k10	b_k9	b_k8-b_k1
0	x	e14	e13	e12	e11	e10	e9	e8-e1

Table 3.3: Environment implementation with k bindings

numbers

We have four different number sizes: 1, 3, and 5, byte signed integers, as well as 5 byte floating point numbers. Tables 3.4, 3.5, 3.6, and 3.7 shows the encoding of those

numbers. The sign bit is the hight order bit of the first non-tag byte, denoted by: $b_1 7$. In the floating point number case e_k denotes the k^{th} bit of the exponent.

F	E	D	C	B	A	9	8	7-0
0	0	0	0	GC	x	0	0	$b_1 7$-$b_1 0$

Table 3.4: 1 byte signed integer

F	E	D	C	B	A	9	8	7-0
0	0	0	0	GC	x	0	1	$b_1 7$-$b_1 0$
$b_2 7$	$b_2 6$	$b_2 5$	$b_2 4$	$b_2 3$	$b_2 2$	$b_2 1$	$b_2 0$	$b_3 7$-$b_3 0$

Table 3.5: 3 byte signed integer

F	E	D	C	B	A	9	8	7-0
0	0	0	0	GC	x	1	0	$b_1 7$-$b_1 0$
$b_2 7$	$b_2 6$	$b_2 5$	$b_2 4$	$b_2 3$	$b_2 2$	$b_2 1$	$b_2 0$	$b_3 7$-$b_3 0$
$b_4 7$	$b_4 6$	$b_4 5$	$b_4 4$	$b_4 3$	$b_4 2$	$b_4 1$	$b_4 0$	$b_5 7$-$b_5 0$

Table 3.6: 5 byte signed integer

booleans

#t and #f are the two scheme booleans. They are unique values. We represent them in data structures as references to program memory locations forbidden to compiled Scheme code, i.e. 0x0000 and 0x0001 respectivly.

procedures

Procedure bodies are known at compile time and thus go in program memory. A procedure body is thus referenced as the location in program memory where execution should start. Procedures themself are located in data memory. They have two parts: a reference to the body in program memory, and a pointer to an environment captured by that procedure. We thus need 14 bits each for these addresses. The procedure encoding is given in Table 3.8.

F	E	D	C	B	A	9	8	7-0
0	0	0	0	GC	x	1	1	e7-e0
$b_1$7	$b_1$6	$b_1$5	$b_1$4	$b_1$3	$b_1$2	$b_1$1	$b_1$0	$b_2$7-$b_2$0
$b_3$7	$b_3$6	$b_3$5	$b_3$4	$b_3$3	$b_3$2	$b_3$1	$b_3$0	$b_4$7-$b_4$0

Table 3.7: 5 byte floating point number

F	E	D	C	B	A	9	8	7-0
0	1	0	GC	b12	b11	b10	b9	b8-b1
b14	b13	e14	e13	e12	e10	e9	e8-e1	

Table 3.8: Procedure encoding

3.2 Primitive procedures

scheme primitives such as arithmetic operations are implemented as assembly subroutines.

3.2.1 arithmetic operations

All arithmetic operations work on any combination of number sizes by dynamically selecting the appropriate operator, after converting all operands to the larges size of the operands. They produce an output that is the smallest size possible. They use registers NargA0, NargA1, NargA2, NargA3, NargA4, NargB0, NargB1, NargB2, NargB3, and NargB4. NargA and NargB are used for the two inputs, NargA is used for the output.

addition and subtraction

Addition and subtraction is straight forward after we have gotten the numbers into the correct registers as specified above. We add/subtract the low bytes first, then consecutively the higher order bytes with carry/borrow. Then, finaly we check for overflow in the carry/borrow bit and adjust number size accordingly. In the case of floating point numbers the add/subtract operation begins with adjusting the number with the smaller exponent to equal the larger exponent before proceeding with the

addition/subtraction.

multiplication

Multiplication is fairly straight forward as well because of the availability of a hardware 8-bit multiplication routine. In this case we also have to check for negative numbers as the harware multiplier only works with unsigned bytes. This is, again, straight forward.

division

We do not have hardware division. We use a naive division algorithm, computing the bits of the result one by one by comparing the sizes of the arguments, and subtracting, elementary school style.

3.2.2 pair operations

cons

Cons allocates a new pair. The car and cdr of the cons cell have already been allocated and have addresses at the point of cons allocation. These addresses will be in NargA, and NargB. Cons thus calls the low level memory allocation routine, which returns to it an address paged into PIC data memory. The addressed can then be put in their appropriate locations withing the 4-byte pair block, along with the tag bit, and GC bit.

car, cdr

Car and cdr takes the address of a pair and returns the addresses of the elements stored there. It is the responsanility of the caller to further dereference these addresses, and return the corresponding number since the value of a number is the actual number, not the location of that number.

3.2.3 predicates

type predicats

Our primitive datatype predicates are pair?, number?, string?, symbol?, and procedure?. The implementation of these is straignt forward, performing a tag check of the location to determine type, and returning wither of the booleans #t, or #f.

predicates for numbers

Our number comparison predicates are: =, >, <, <=, and >=. These are implemented using the PIC native comparison operations on bytes, consecutively, starting at the hight byte, and moving down. They have to persorm the appropriate number conversions as well to allow for comparison of different size numbers.

other predicates

Our only other predicate implemented at this time is eq?. eq? simply checks whether or not two object have the same location in memory. We can therefore use the PIC native equivalence comparison operation.

3.2.4 hardware specific operations

time

A subroutine is avaiable to get the current time off the RTC chip. This routine simply invokes the correct I^2C commands.

run-timer1, run-timer2, set-timer1! set-timer2!

We provide access to two of the hardware timers on the PIC, with interrupts. This is an attempt to start looking at the possibility to do interrupt driven programming using scheme on the PIC. set-timerx! takes an unsigned 16bit integer and sets the timer to this value. run-timerx takes a procedure as argument. It will enable interrupts for that timer, and then enable it. While execution is occuring that timer will

count up, incrementing once per instruction. When a rollover occurs the PIC will have an interrupt, vectoring to a location that then will run the procedure provided as an argument to run-timerx. When that procedure is done the PIC will return from interrupt to the point it was at whenthe interrupt occurred. The value returned by that procedure will be discarded since there is no place to return it it.

3.2.5 extensibility of primitives

All primitives are implemented as PIC subroutines. If we would like to extend the compiler with some different hardware routine we need simply write it in PIC assembly and bind that routine in the initial environment at compile time.

Appendix A

Language Specification

In this appendix we will give a specification of the subset of Scheme used for this project. It does not include all the features marked as essential in the formal language it is based on, and should therefore perhaps be renamed. This is, however, only intended as a starting point for compiling Scheme for small processors. Future work would therefore include extending the compiler with all essential features.

A.1 Syntax

The syntax for our language is as specified in the Revised(4) Report on the Algorithmic Language Scheme. We will summerize the important points here.

A.1.1 Whitespace, comments and parentheses

Any type of whitespace (space, tab, and newline to name a few), as well as parentheses, seperate character sequences on either side into *tokens*. Parentheses also have an important semantic role. A semicolon, ;, begins a comment, which extends until the next newline character. Comments will be ignored by the compiler and is used exclusivly to improve readability of programs to humans.

A.1.2 Identifiers

Any sequence of characters that begin with a character other than a digit, +, or -, is an identifier. Our set of characters include all alpha-numeric characters, as well as the following extended alphabetic characters: `+ - . * / < = > ! ? : $ % _ & ~ ^`

A.1.3 Syntactic keywords

Certain identifiers are syntactic keywords. Any identifier that is not a syntactic keyword can be used as a *variable*. The identifiers in Table A.1 are reserved as syntactic keywords, although some of them are not implemented in the compiler, beyond being disallowed as variable names.

=>	do	or
and	else	quasiquote
begin	if	quote
case	lambda	set!
cond	let	unquote
define	let*	unquote-splicing
delay	letrec	

Table A.1: Syntactic keywords

A.2 Semantics

We will now define the meaning of various constructs in our language. The syntax: ($< exp_1 > < exp_2 > \ldots < exp_n >$) in general means *evaluate* $< exp_1 >$ through $< exp_n >$ in any order, then *apply* the value of $< exp_1 >$ to $< exp_2 >$ through $< exp_n >$. Our full expression above is known as a *combination*. The sub-expressions, $< exp_1 >$ through $< exp_n >$, can be primitives, or combinations themself. If $< exp_1 >$ is a *syntactic keyword* we do not follow the rule above, but rather look at the rule associated with the *special form* associated with that keyword. The special forms implemented here described below.

A.2.1 special forms

define

Syntax: `(define name` $< exp >$`)`

Define does not evaluate its first argument, but does evaluate the second. It interprets the first argument as a name, and bind it to the value of its second argument in the current environmert.

lambda

Syntax: `(lambda (`v_1 ... v_n`)` $< bexp_1 >$ ... $< bexp_n >$`)`

A lambda creates a *procedure* that can be *applied.* None of the arguments in the lambda gets evaluated. Rather, we create a procedural object. If that procedure gets applied we *substitute* the values of the arguments for the *formals*, (v_1 ... v_n), in the *body*, $< bexp_1 >$ ... $< bexp_n >$.

set!

Syntax: `(set! name` $< exp >$`)`

Works like define, except `name` must be bound in the current environment, or one of its ancestors, and that binding is changed to the value of $< exp >$.

if

Syntax: `(if` $< test - exp >$ $< then - exp >$ $< else - exp >$`)`

If starts by evaluating $< test - exp >$. If it evaluates to #f $< else - exp >$ is evaluated next, and its value is returned. If it does not evaluate to #f $< then - exp >$ is evaluated next, and its value is returned. In particular, any value other than #f is interpreted as true by if, including the number 0.

quote

Syntax:`(quote <exp>)`

Quote keeps $< exp >$ from being evaluated. Rather it parses it as a syntactic expres-

sion and return its internal representation.

Appendix B

Assembly Routines

B.1 addition subroutines

B.1.1 1-byte signed integer addition

```
;this is the memory managment stuff. the PICs data memory is divided
;into twenty 64 byte pages that are swapped in and out from EEPROM
;memory. all these pages start out empty, with corresponding addresses
;from the begining of EEPROM memory. They are filled as memory is
;used. when we start using the last page in data memory we will also
;start transfering the oldest page to the EEPROM. As we start
;accumulating more data in the EEPROM than would fit in the data
;memory we will have to start swapping pages in and out. when a page
;is read in to data memory page defined by PSAk (0<=k<=19) PSAkms will
;start at 0xFF, and PSAkme will start at 0x00. this is to indicate
;that nothing has been modified in this page yet. when a modification
;occurs we set those two registers to correspond to the modified
;value's location in EEPROM memory. PSAkms will thereafter be
;maintained to point to the first modified address in the block, and
;PSAkme to the last modified one. that way we can write a minimal
;amount when we have to swap that page out again.  page writes to the
```

```
;EEPROM are done based on interrupts. we start writing a page when we
;start using the last free page in data memory (the first time we get
;to PSA19). PSA0 will be written out first since that was the first
;created one. we initiate this write right after we have claimed
;PSA19.  we hope to be done by the next time we need to access the
;EEPROM. after that we initiate writing a page to EEPROM memory
;whenever we need a new data memory page. the page after the current
;free data memory page will be written out since that is the oldest
;one.  when the EEPROM memory is half full (PSAkH=0x7F, PSAkL=0xC0 for
;some k) garbage collection is initiated.  we use a stop and copy GC
;algorithm, moving data to the other half of the EEPROM.  data memory
;is cleared and the pages will be moved from the EEPROM to be copied
;and moved back.  only 16 pages will be available to hold data to be
;copied. of the remaining 4, one page will hold data being transfered
;to the EEPROM, another will accumulate values to be copied. the
;remaining two will hold locations yet to be copied. this is where we
;will place locations that are not currently in data memory in hope
;that we can first copy as much as possible from the pages already in
;memory.

;most of the PSA registers will be accessed indirectly only. they are
;named to indicate they are taken, and so that we can refere to them
;in comments.

PSA0H           equ             0x80
PSA1H           equ             0x81
PSA2H           equ             0x82
PSA3H           equ             0x83
PSA4H           equ             0x84
PSA5H           equ             0x85
```

```
PSA6H           equ             0x86
PSA7H           equ             0x87
PSA8H           equ             0x88
PSA9H           equ             0x89
PSA10H          equ             0x8A
PSA11H          equ             0x8B
PSA12H          equ             0x8C
PSA13H          equ             0x8D
PSA14H          equ             0x8E
PSA15H          equ             0x8F
PSA16H          equ             0x90
PSA17H          equ             0x91
PSA18H          equ             0x92
PSA19H          equ             0x93
PSA0L           equ             0x94
PSA1L           equ             0x95
PSA2L           equ             0x96
PSA3L           equ             0x97
PSA4L           equ             0x98
PSA5L           equ             0x99
PSA6L           equ             0x9A
PSA7L           equ             0x9B
PSA8L           equ             0x9C
PSA9L           equ             0x9D
PSA10L          equ             0x9E
PSA11L          equ             0x9F
PSA12L          equ             0xA0
PSA13L          equ             0xA1
PSA14L          equ             0xA2
PSA15L          equ             0xA3
```

```
PSA16L          equ             0xA4
PSA17L          equ             0xA5
PSA18L          equ             0xA6
PSA19L          equ             0xA7
PSA0me          equ             0xA8
PSA1me          equ             0xA9
PSA2me          equ             0xAA
PSA3me          equ             0xAB
PSA4me          equ             0xAC
PSA5me          equ             0xAD
PSA6me          equ             0xAE
PSA7me          equ             0xAF
PSA8me          equ             0xB0
PSA9me          equ             0xB1
PSA10me         equ             0xB2
PSA11me         equ             0xB3
PSA12me         equ             0xB4
PSA13me         equ             0xB5
PSA14me         equ             0xB6
PSA15me         equ             0xB7
PSA16me         equ             0xB8
PSA17me         equ             0xB9
PSA18me         equ             0xBA
PSA19me         equ             0xBB
EEfreeH         equ             0x7F            ;points to next free
                                                ;byte in EEprom
EEfreeL         equ             0x7E
PRptr           equ             0x7D            ;page register
                                                ;pointer, points to
                                                ;high byte register
```

```
                                        ;of next free data
                                        ;page
GCstatus        equ         0x7C        ;GC status word
BB              equ         0x7         ;block bit for eeprom
                                        ;address in GCstatus
DMF             equ         0x6         ;data memory full bit
                                        ;in GCstatus
MADDH           equ         0x7B        ;memory address high
                                        ;byte
MADDL           equ         0x7A        ;memory address low
                                        ;byte
TMP0            equ         0x79
EEaddh          equ         0x78        ;EEPROM high address
                                        ;byte to be read/
                                        ;written, start
                                        ;address
EEaddl          equ         0x77        ;EEPROM low address
                                        ;byte to be read/
                                        ;written, start
                                        ;address
DATAaddle       equ         0x76        ;EEPROM low address
                                        ;byte to be read/
                                        ;written, end address
DATAnumB        equ         0x75        ;number of data bytes
                                        ;requested


;we want to get something from memory, but all we have is an EEPROM
;address. this subroutine expects that address in the MADDH, MADDL
;registers. it will find out if the page this address is on is already
```

```
;in data memory. if it is, the corresponding datamemory address will
;be returned in FSR0. if not, the correct page will be loaded from
;EEPROM memory

memorycheck:                            ;DATAnumB contains number of
                                        ;words requested
                                        ;checks if data fits in
                                        ;memory, if not GC first
        MOVFF   EEfreeH, TMP0           ;move high EEprom free address
                                        ;byte to temporary register
        BCF     STATUS, C, 0
        RLCF    DATAnumB, w, 0          ;multiply by 2, move to WREG
        BTFSC   STATUS, C               ;carry is set if we rotate a
                                        ;1 out of DATAnumB
        INCF    TMP0, f, 0              ;if carry set, inc temporary
                                        ;version of EEfreeH
        ADDWF   EEfreeL, w, 0           ;add low EEfree to number of
                                        ;bytes requested
        BTFSC   STATUS, C               ;test for carry
        INCF    TMP0, f, 0              ;if carry, increment temporary
                                        ;version of EEfreeH
        MOVF    TMP0, w, 0
        XORWF   GCstatus, w, 0          ;xor with GCstatus, bit 7 is
                                        ;block bit for both
        BTFSC   WREG, 7, 0              ;check bit 7, if 0 okay, if
                                         ;1 GC
        GOTO    GC
        return                          ;otherwise memory okay, return
memoryput:
        MOVFF   DATAwH, POSTINC1        ;move to memory location
```

```
        MOVFF   DATAwL, POSTINC1
        MOVFF   EEfreeH, MADDH
        MOVFF   EEfreeL, MADDL
        INCF    EEfreeL, f, 0
        INCF    EEfreeL, f, 0
        BTFSC   STATUS, Z               ;did we roll over a 1 to
                                        ;EEfreeH?
        INCF    EEfreeH, f, 0           ;if so, add 1
        MOVLW   B'00111111'
        ANDWF   EEfreeL, w, 0
        BTFSS   STATUS, Z               ;did we roll over a page??
        return                          if not, return
memputrof:                              ;fixes page rollovers in
                                        ;EEfree pointer
        MOVFF   PRptr, FSR0L            ;this is the next free data
                                        ;page pointer (points to
                                        ;PSAkH)
        CLRF    FSR0H, 0
        MOVFF   EEfreeH, INDF0          ;put the EEprom high address
                                        ;in its place
        MOVLW   0x14
        ADDWF   FSR0L, f, 0
        MOVFF   EEfreeL, INDF0          ;put the EEprom low address
                                        ;in its place
        MOVF    PRptr, w, 0
        ADDLW   0x84
        MULLW   0x40                    ;get address of corresponding
                                        ;data page in PROD
        MOVFF   PRODL, FSR1L
        MOVFF   PRODH, FSR1H            ;free memory pointer updated
```

```
        CALL    incPRptr                ;increment PRptr
        BTFSS   GCstatus, DMF, 0        ;is data memory full after
                                        ;reading this page?
        RETURN                          ;otherwise, return
        GOTO    EEPROMwpINIT            ;initialize the page write if
                                        ;it is
        return


MemoryAdderssFetch:                     ;memory address to fetch is in
                                        ;MADD, will return data memory
                                        ;address
                                        ;of object in FSR0
        LFSR    0, PSA19H
HavePage?:
        MOVLW   0x13
        SUBWF   FSR0L, f, 0
        MOVLW   0x94
        CPFSLT  FSR0L, 0                ;check if our pointer is less
                                        ;than, or equal to, the last
                                        ;page address
        GOTO    PageFetch               ;if not, page is not in
                                        ;memory, fetch it
        MOVF    POSTINC0, w, 0          ;get page high address byte
        CPFSEQ  MADDH, 0                ;compare with the address we
                                        ;are looking for
        GOTO    HavePage?               ;if not, try next page
        MOVLW   0x13
        ADDWF   FSR0L, f, 0
```

```
        MOVF    INDF0, w, 0             ;get page low address byte
        XORWF   MADDL, w, 0             ;check if bits match
        ANDLW   B'11000000'             ;ignore low 6 bits
        BTFSS   STATUS, Z               ;check for zero result
                                        ;(address match)
        GOTO    HavePage?               ;else keep going
        MOVLW   0x13
        SUBWF   FSR0L, f, 0
GetPageAddress:
        MOVLW   0x7D                    ;subtract to get value
                                        ;relating to data address
                                        ;of object
        SUBWF   FSR0L, w, 0
        MULLW   0x40                    ;multiply with 64 for high
                                        ;and low data address values
        MOVFF   PRODH, FSR0H
        MOVFF   PRODL, FSR0L
        MOVLW   B'00111111'
        ANDWF   MADDL, w, 0             ;get low six bits of address
        IORWF   FSR0L, f, 0
        RETURN

PageFetch:                              ;move a page from EEPROM
                                        ;memory to data memory
        MOVFF   PRptr, FSR0L            ;get register for high EEPROM
                                        ;address placement
        CLRF    FSR0H, 0
        MOVF    MADDH, w, 0
        MOVWF   INDF0, 0                ;move high address to PSAkH
        MOVWF   EEaddh, 0
```

```
MOVLW   0x14
ADDWF   FSR0L, f, 0          ;corresponding low address
                             ;register
MOVLW   B'00111111'
IORWF   MADDL, w, 0          ;low address, six low order
                             ;bits are 1s to indicate
                             ;nothing
MOVWF   INDF0, 0             ;modified in page
ANDLW   B'11000000'
MOVWF   EEaddl, 0
MOVLW   0x14
ADDWF   FSR0L, f, 0          ;get modify end address
MOVLW   0x00
MOVWF   INDF0, 0             ;write modify end address to
                             ;its register
MOVF    PRptr, w, 0
ADDLW   0x84
MULLW   0x40
MOVFF   PRODH, FSR0H
MOVFF   PRODL, FSR0L         ;data memory corresponding to
                             ;current data page register
CALL    EEPROMrp             ;read page to data memory,
                             ;EEPROM address in
                             ;EEADDH/EEADDL
                             ;data memory address in FSR0
MOVF    MADDL, w, 0          ;after this call FSR0H is the
                             ;same, FSR0L has bits 7 and 6
ANDLW   B'00111111'          ;same as before, bits 0-5 are
                             ;1s
ANDWF   FSR0L, f, 0          ;now make them the value
```

```
                                        ;corresponding to the address
                                        ;we wanted
        CALL    incPRptr
        BTFSS   GCstatus, DMF, 0        ;is data memory full after
                                        ;reading this page?
        RETURN                          ;otherwise, return
        GOTO    EEPROMwpINIT            ;initialize the page write
                                        ;if it is

;set up addresses to start and end the writing at according to PSAkH,
;PSAkms, and PSAkme if PSAkme < PSAkms we don't need to write page to
;memory (it hasn't been modified since it was fetched.

;also, fix page write so that it is interrupt driven since we don't
;need to wait for that write to finish, need it done by the next time
;we need to read/write EEPROM memory use ISR vector registers to come
;back to right place in code on interrupt probably need to turn on
;buss collision interrupts as well as SSPIE



incPRptr:
        INCF    PRptr, f, 0
        MOVLW   0x93
        CPFSGT  PRptr, 0
        RETURN                          ;if less than 0x93, no page
                                        ;rollover
        BSF     GCstatus, DMF, 0        ;if we are at the last page,
                                        ;set the Data Memory Full bit
        MOVLW   0x80                    ;and restart the pointer at
```

```
                                        ;PSA0
        MOVWF   PRptr, 0
        RETURN


EEPROMwpINIT:
        MOVF    PRptr, w, 0
        ADDLW   0x84
        MULLW   0x40                    ;data memory corresponding to
                                        ;current data page register
        MOVF    PRODH, w, 0
        CPFSEQ  FSR1H, 0                ;are the high bytes of PRptr
                                        ;data page and the current
                                        ;free page the same? if not,
                                        ;okay to write this page
        GOTO    EEPROMwpTHIS
        MOVF    FSR1L, w, 0
        ANDLW   B'11000000'             ;ignore 6 low bits
        CPFSEQ  PRODL, 0                ;compare low bytes
        GOTO    EEPROMwpTHIS
        CALL    incPRptr                ;otherwise, next page is the
                                        ;page to write
EEPROMwpTHIS:
        MOVFF   PRptr, FSR0L            ;point to high EEprom address
        CLRF    FSR0H, 0
        MOVFF   INDF0, EEaddh           ;get high EEprom address of
                                        ;page
        MOVLW   0x14
        ADDWF   FSR0L, f, 0             ;point to low EEprom address
        MOVFF   INDF0, EEaddl           ;get low EEprom address to
```

```
                                ;start writing
MOVLW   0x14
ADDWF   FSR0L, f, 0             ;point to end write EEprom
                                ;address
MOVF    INDF0, w, 0
MOVWF   DATAaddle, 0            ;get end write EEprom address
CPFSLT  EEaddl, 0               ;compare start address with
                                ;end addres
return                          ;if start address is not less
                                ;than end, nothing to write
MOVF    PRptr, w, 0
ADDLW   0x84
MULLW   0x40
MOVF    DATAaddle, w, 0         ;get end wite EEPROM address
ANDLW   B'00111111'             ;preserve low 6 bits
IORWF   PRODL, w, 0             ;get high bits from PRODL
MOVWF   DATAaddle, 0            ;move to DATAaddle

MOVFF   PRODH, FSR0H            ;data memory high address
                                ;corresponding to current
                                ;data page register
MOVF    EEaddl, w, 0
ANDLW   B'00111111'             ;preserve 6 low bits
IORWF   PRODL, w, 0             ;get high bits from PRODL
MOVWF   FSR0L, 0                ;now we have the start EEprom
                                ;address in EEaddh/l, the end
                                ;data
                                ;low address in DATAaddle, the
                                ;corresponding start data
                                ;address
```

```
                                    ;is in FSR0. ready to let
                                    ;EEPROMwp write page

EEPROMwp:                           ;subroutine for writing a page
                                    ;of 64 bytes to EEPROM
                                    ;write can't cross 64 byte
                                    ;page boundries
        BCF     PIR1, SSPIF, 0      ;clear possible transmit done
                                    ;flag
        BCF     PIR2, BCLIF, 0      ;clear possible bus collision
                                    ;flag
        CALL    EEPROMadd

dataloopwp:
        CALL    I2Cidle             ;check if we are idle
        MOVF    POSTINC0, w, 0      ;move data value to W,
                                    ;increment FSR0
        MOVWF   SSPBUF, 0
        CALL    waitforSSPIF
        BTFSC   SSPCON2, ACKSTAT, 0 ;did the slave acknowledge?
        GOTO    EEPROMwp            ;otherwise, start over
        MOVF    FSR0L, w, 0         ;current low address
        CPFSLT  DATAaddle, 0        ;test if end address in less
                                    ;than current address
        GOTO    dataloopwp          ;if not, keep going
datadonewp:
        CALL    I2Cidle
        BSF     SSPCON2, PEN, 0     ;generate stop condition
```

```
        RETURN

EEPROMadd:              ;subroutine for specifying an address to be
                        ;read or written on the EEPROM
        BCF     PIR1, SSPIF, 0          ;clear possible transmit done
                                        ;flag
        BCF     PIR2, BCLIF, 0          ;clear possible bus collision
                                        ;flag

        CALL    I2Cidle
        BSF     SSPCON2, RSEN, 0        ;generate start condition
        CALL    waitforstart
        BTFSC   PIR2, BCLIF, 0          ;check for bus collision
        GOTO    EEPROMadd               ;if detected, try again
        CALL    I2Cidle
        MOVLW   B'10100000'             ;slave address specifying a
                                        ;write
        BTFSC   GCstatus, BB, 0         ;what is the high bit in the
                                        ;address??
        BSF     WREG, 3, 0              ;if 1, set the corresponding
                                        ;bit in the slave address
                                        ;see 24LC515 datasheet for
                                        ;detailed addressing info

        MOVWF   SSPBUF, 0               ;load shift buffer with slave
                                        ;address
        CALL    waitforSSPIF
        BTFSC   SSPCON2, ACKSTAT, 0     ;did the slave acknowledge?
        GOTO    EEPROMadd               ;otherwise, start over
        CALL    I2Cidle
```

```
        MOVF    EEaddh, w, 0
        MOVWF   SSPBUF, 0               ;load shift buffer with high
                                        ;EEPROM memory address
        CALL    waitforSSPIF
        BTFSC   SSPCON2, ACKSTAT, 0     ;did the slave acknowledge?
        GOTO    EEPROMadd               ;otherwise, start over
        CALL    I2Cidle
        MOVF    EEaddl, w, 0
        MOVWF   SSPBUF, 0               ;load shift buffer with high
                                        ;EEPROM memory address
        CALL    waitforSSPIF
        BTFSC   SSPCON2, ACKSTAT, 0     ;did the slave acknowledge?
        GOTO    EEPROMadd               ;otherwise, start over
        RETURN

EEPROMrp:
        BCF     PIR1, SSPIF, 0          ;clear possible transmit done
                                        ;flag
        BCF     PIR2, BCLIF, 0          ;clear possible bus collision
                                        ;flag
        CALL    EEPROMadd

EEPROMrpstart:
        CALL    I2Cidle
        BSF     SSPCON2, RSEN, 0        ;generate start condition
        CALL    waitforstart
        BTFSC   PIR2, BCLIF, 0          ;check for bus collision
        GOTO    EEPROMrpstart           ;if detected, try again
        CALL    I2Cidle
        MOVLW   B'10100001'             ;slave address specifying a
```

```
                                        ;read
        BTFSC   GCstatus, BB, 0         ;what is the high bit in the
                                        ;address??
        BSF     WREG, 3, 0              ;if 1, set the corresponding
                                        ;bit in the slave address
                                        ;see 24LC515 datasheet for
                                        ;detailed addressing info
        MOVWF   SSPBUF, 0               ;load shift buffer with slave
                                        ;address
        CALL    waitforSSPIF
        BTFSC   SSPCON2, ACKSTAT, 0     ;did the slave acknowledge?
        GOTO    EEPROMrp                ;otherwise, start over

dataloorprp:
        CALL    I2Cidle
        BSF     SSPCON2, RCEN, 0        ;enable reading
        CALL    waitforRCEN
        MOVF    FSR0L, w, 0
        XORLW   0xFF
        ANDLW   B'00111111'             ;we are done when last 6 bits
                                        ;are 1s
        BTFSC   STATUS, Z, 0
        GOTO    datadonerp
        MOVF    SSPBUF, w, 0            ;move read value to w
        MOVWF   POSTINC0, 0             ;move data value to file
                                        ;register, increment FSR0
        BCF     SSPCON2, ACKDT, 0
        BSF     SSPCON2, ACKEN, 0       ;acknowledge
        GOTO    dataloorprp
datadonerp:
```

```
        MOVF    SSPBUF, w, 0            ;move read value to w
        MOVWF   INDF0, 0                ;move data value to file
                                        ;register
        BCF     SSPCON2, RCEN, 0
        CALL    I2Cidle
        BSF     SSPCON2, PEN, 0         ;generate stop condition
        RETURN

I2Cconfig:
        MOVLW   0x19                    ;400kHz operation
        MOVWF   SSPADD, 0
        BSF     SSPSTAT, SMP, 0         ;enable slew rate control for
                                        ;high speed
        BCF     SSPSTAT, CKE, 0         ;disable SMBus specific inputs
        BCF     TRISC, SCL, 0           ;clock line output
        BCF     TRISC, SDA, 0           ;data line output
        BSF     SSPCON1, SSPEN, 0       ;enable the serial port on
                                        ;SCL and SDA
        BSF     SSPCON1, SSPM3, 0
        BCF     SSPCON1, SSPM2, 0
        BCF     SSPCON1, SSPM1, 0
        BCF     SSPCON1, SSPM0, 0       ;I2C master mode
        return

I2Cidle:
        MOVLW   B'00011111'
        ANDWF   SSPCON2, w, 0
        BTFSC   SSPSTAT, 2, 0
        BSF     WREG, 5, 0
        ADDLW   0x00
```

```
        BTFSS   STATUS, Z, 0            ;check if we are idle
        GOTO    I2Cidle                 ;wait until we are
        return

waitforstart:
        BTFSC   PIR2, BCLIF, 0          ;check for bus collision
        GOTO    buscoll
        BTFSS   PIR1, SSPIF, 0          ;check if we are done with
                                        ;the start condition
        GOTO    waitforstart
        BCF     PIR1, SSPIF, 0
        return

waitforRCEN:
        BTFSC   SSPCON2, RCEN, 0
        GOTO    waitforRCEN
        BCF     PIR1, SSPIF, 0
        return

waitforSSPIF:
        BTFSS   PIR1, SSPIF, 0          ;wait for transmit/recieve to
                                        ;finish
        GOTO    waitforSSPIF
        BCF     PIR1, SSPIF, 0          ;clear transmit done flag
        return

buscoll:
        BCF     PIR1, SSPIF, 0
        BCF     PIR2, BCLIF, 0
        return
```

```
END
```

Bibliography

[1] William Clinger and Jonathan Rees Editors. Revised(4) report on the algorithmic language scheme. *ACM Lisp Pointers IV*, July-September 1991.

[2] Linear Technology Corporation. *LT1579 Data Sheet.* http://rocky.digikey.com/WebLib/Linear%20Tech%20Web%20Data/LT1579.pdf, 1996.

[3] Microchip Technology Inc. *PIC18FXX2 Data Sheet.* http://www.microchip.com/download/lit/pline/picmicro/families/18fxx2/39564b.pdf, 2002.

[4] Texas Instruments. *CD54/74HC85, CD54/74HCT85 Data Sheet.* http://rocky.digikey.com/WebLib/Texas%20Instruments/Web%20data/CD74HC(T)85.pdf, 2002.

[5] Texas Instruments. *SN74CBT3245A Octal FET Bus Switch.* http://rocky.digikey.com/WebLib/Texas%20Instruments/Web%20data/SN74CBT3245A.pdf, 2002.

[6] Dallas Semiconductor. *DS1307 64x8 Serial Real Time Clock Data Sheet.* "http://rocky.digikey.com/WebLib/Dallas/Dallas%20Web%20Data/DS1307.pdf.