University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Reprogramming Sensors for the Cyberjackets

Tom Parker

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering

May 2003 | CSBSC-03

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Tom Parker, May 2003

Abstract

PIC microcontrollers (as used for the sensors in the Cyberjacket project in the ongoing wearables research at Bristol University) are currently programmed mainly in PIC assembler. This project provides a higher-level C API to provide a way to program the PIC's without having to use assembler - which reduces the time taken to write new code, as well as improving maintainability of existing code. The API also adds a series of additional features not present in PIC assembler, including compile-time setting of minimum execution times for sections of code, loop structures and decision structures.

The API gives the end-user programmer a way to build a code tree from simple segments, and then to generate code for that tree. Because of the use of an API within an existing language, we can use that language's exisiting features to generate a wide variety of different pieces of PIC code from a single program, without the overheads of recompiling the program. This allows for greater specialisation in the PIC code, reducing the problems with the limited space of the PICs.

This project is designed for the ARM-based Bitsy computer, but would work equally well on other systems with minimal resources.

Contents

Chapter 1

Prologue

One of the major projects of the ongoing wearables research at Bristol University has been the "cyberjacket" project. We have been working on a new way to program the PIC microcontrollers used for the sensors in the "cyberjacket" project. The PICs are very simple microcontrollers, with limited functionality, and this project is designed to provide additional capabilities for the people programming the PICs, by exploiting what can be done by combining different parts of the existing capabilities, and providing a higher-level view of the program, with less knowledge about the underlying assembler code.

1.1 Aims

The main aims of this project are

- To provide a way for the sensors to be programmed with a variety of new programs by the Bitsy (the wearable computer that is the hub of the cyberjacket), allowing the new code for the sensors to be determined at run-time by the Bitsy, thus allowing flexibility without significantly increasing the time needed for the Bitsy to supply the code, as opposed to using pre-generated code.
- To provide a way to do this without having to directly use PIC assembler, which is the current standard method for programming the sensors.
- Additional features must have been added to what is possible to simply do with PIC programming, over what can be done with just PIC assembler. These will be features that were possible with PIC assembler before, but they will now be actual features in the system used for programming, thus allowing for greater and simpler use of these features. Examples of some features that will be added include timing specifications (i.e. specification of how long a section of code will take to run), while loops and if/then/else blocks.

- Time to alter/debug a program for the PIC must be significantly reduced i.e. for a given set of example projects, it must be shown that it will be faster to add additional features with the new system, than it would be to write the PIC assembler. It must also be possible to make many changes in succession without any additional complications.

1.2 Overview

- Chapter 2 talks about the background to the project - explaining more about the cyberjackets, PIC assembler, and existing attempts to make PIC programming easier.
- Chapter 3 looks at how we can implement the aims of this project, looking at a variety of design choices and seeing why we make the choices in our designs that we do
- Chapter 4 has a guide to the end project API, with more details about the underlying implementation, and how a number of tricky problems were solved
- Chapter 5 looks at an old implementation of some sensor code, and re-implement the same functionality using the new API
- Chapter 6 is the conclusion of this project, summarizing what we have achieved, and detailing future areas of research.
- There then follows the appendices
  - Appendix A is the full code for a sensor
  - Appendix B is the re-written code for the same sensor

Chapter 2

Background

2.1 Cyberjackets

The cyberjackets are part of the wearable computing projects at Bristol University, designed as a part of the focus of the project exploring the potential of computer devices that are as unconsciously portable and as personal as clothes or jewellery.

The jackets themselves consist of a heavy-duty jacket with a Bitsy (a small ARM-based portable computer with a wireless networking interface attached) in one of the pockets, a series of sensors (examples include GPS, accelerometers, compasses and ultrasonic sensing devices) attached to parts of the jacket, and a data bus lead connecting the Bitsy and sensors together.

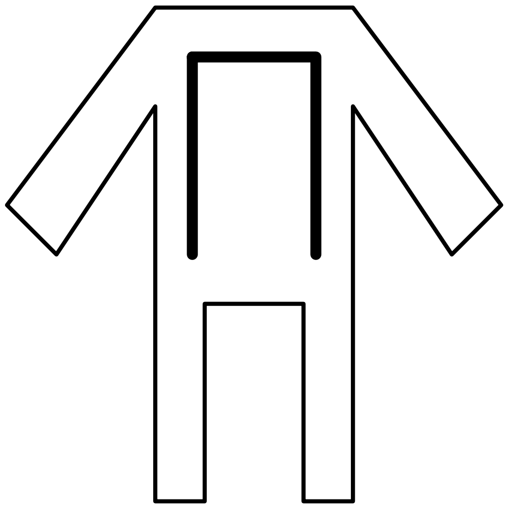
Figure 2.1: Abstract representation of a cyberjacket. This shows the data bus (thick line) as part of the main body of a person wearing the jacket (shown here as the thinner outline)

The main focus of this project is the sensors themselves, which despite their varied uses, are all based around the same microprocessor, a PIC16F84A, made by Microchip Technology Inc. This is a fairly simple 8-bit microprocessor, running at 20MHz, with only 1024 words of program memory, along with a further 132 bytes for data storage.

2.2 PIC Assembler

The PIC16F84A has a set of 35 different instructions to work with in its instruction set, but to do what would be considered most simple tasks in a higher level language (such as adding two numbers together), will take several instructions in PIC assembler. Additionally, the assembly language has no built-in capability to do various operations that are often needed in the sensors, such as ensuring that a given section of code will take a specific amount of time, as is needed for a variety of time-dependant applications. PIC assembler is also hard to write for - most programmers in this modern age are unused to assembler programming, and need a higher level language so that they may concentrate on what they want to get done, as opposed to having to deal with all of the smaller details about how repetitive parts of the program are done.

Various languages have already had tools written for them that enable writing code for the PIC chips without having to delve into the complexities of PIC assembler. These include various C compilers, several variants of BASIC and even a converter from Java bytecode.

Here are a few examples of languages/translators targeted for PIC assembler

2.2.1 [JAL]

JAL is an“Algol-style meekly typed block scoped language" - also similar to some forms of Pascal. The basic language is fairly clean and simple, but there is very little in the language that is specific to the real-time applications and capabilities of the PIC that are the major targets of this project. There is no scope for interrupt handling, nor is there anything to handle timing limits on sections of code.

2.2.2 [Aino]

Aino is a translator from Java bytecode to PIC assembler. You get (most) of the basic Java language features, but there are none of the standard Java libraries for example. Existing code for other platforms could possibly be reused, but most of it will rely on the normally very good standard libraries, all of which are far too large to fit onto the PICs. There is no need to learn a new language, so old skills can be re-used. Java is not designed for the PIC, so doing good optimization is hard and getting all of the capabilities of the PIC from this will be very hard.

2.2.3 [C2C]

C2C is a translator from C to PIC assembler. This has the advantage of using the C syntax - well known by most developers. But, there’s no standard C libraries - given that they're mostly bigger than the total PIC storage space, this isn't surprising. This is the closest to my proposed ideas, but it isn't as flexible. It does have interrupt handling, but no major additional features on top of the basic PIC command (like timing restraints).

2.2.4 [MicroSeeker PIC/Smalltalk]

Microseeker PIC/Smalltalk is a translator from a variant of Smalltalk to PIC assembler. The use of Smalltalk can either be counted as an advantage or a disadvantage, but as a majority of developers don't use Smalltalk, and the syntax appears to be somewhat non-standard compared to other languages, we currently classify this as a disadvantage. This product also came with no documentation whatsoever. Evaluating this product is difficult given this limit, but it doesn't appear to have anything to particularly recommend it. The automatic integration of comments into the output assembler is a nice feature, but not worth the learning curve of a new language, given the lack of anything else to recommend it.

2.3 Motivation for the project

A lot of the example systems mentioned above have fairly similar sets of problems, namely either they take an existing language with feature sets guided towards desktop or server applications and they try and add features to make it usable for PIC applications, or they try and create a new language with microprocessor specific features (timing for sections of code for example), which is then less flexible.

They are all also based around the standard model of designing code for microprocessor applications - compile a version of the code on a fast desktop machine once, which outputs assembler code which you write to the microprocessor using the standard methods for PIC assembler code.

The Bitsy is a portable computer, with all of the compromises about trading speed for lowered power consumption that is standard practice for this type of device. Attempting to use one of the existing language choices to re-compile the program every time we want different code, which may well be required every time we plug/unplug a sensor, is impractical. The cyberjacket system, in order to be usable as a device where booting/loading times don't need to be thought about, needs a much faster way to generate code for the sensors, while still retaining the flexibility of changing the code that re-compilation would allow.

But what if the flexibility to do several different problems with one large program can be left in at run time without reducing the space available for each individual program? Normally this isn't done because of the limited storage space of the PIC reduces the flexibility that can be placed in the code, but what if we can find an alternate way around the problem?

The Bitsy doesn't have a lot of storage space or processor capability, but if we can figure out how to create a system that can generate PIC code at runtime that is smaller and faster than recompilation, by reducing the number of steps to generate PIC assembler (or at least the steps that the Bitsy has to do) then we can have a flexible system that will work for the cyberjackets.

In order to reduce the number of steps needed for generating PIC assembler on the Bitsy, the easiest way is to simply start further down the standard path used to compile source code - get rid of the processor intensive text processing stages, and make the building of the parse tree a much simpler job, by working out as much as possible beforehand. We can do this by writing an API to give the end user programmer direct access to the code generation steps of a language with basic standard procedural features (e.g. loops, variables, arithmetic and boolean expressions) with some additions for the PIC (I/O handling for example). With this, the end user can write a program that tells our API what they want to do, as well as any other code that they want that the Bitsy has libraries for, and we don't have to mess around with text processing, we can write the API so that most (as many as possible) invalid programs simply will not compile, and we have a workable method for producing a program that can generate PIC assembler that can be flexibly generated (i.e. change depending on the current state of the Bitsy) without taking a long time to run on the Bitsy.

Chapter 3

Analysis of Design Choices

There are a variety of different approaches that could be taken in order to manage to achieve the stated goals of this project. In this chapter we will analyse a series of these choices, illustrating the differnet options and explaining why we picked the choices that we decided to use.

3.1 Object-orientated vs. Functional approaches

Our choosing between the two approaches to the design of the API decides how we will be able to put together code

3.1.1 Object-orientated

With an object orientated approach, the program is built up by first creating a “Code” object, and then adding the list of statements that you want to be executed to this. The “Code” object would also have methods to create variables and return references to them that can be used with the various methods used to add statements.

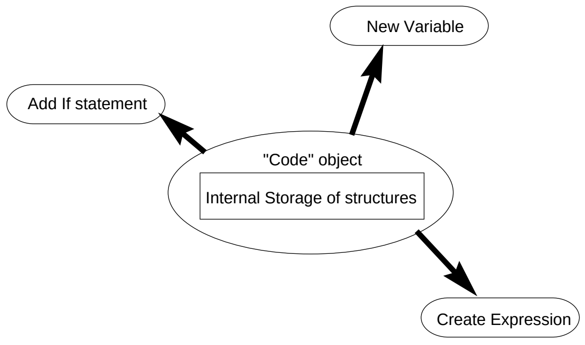

Figure 3.1: Object orientated approach

3.1.2 Functional

A functional approach is centred around a series of structures, and functions to manipulate structures. There are different functions to create different kinds of nodes - for example the "+" node in the diagram is an example of an node representing part of a mathematical expression. The "2" and "3" nodes are also mathematical expression nodes, albeit very simple examples, and this allows a "+" node (or any other binary mathematical operator) to be used to combine any two other expression nodes to form a new expression subtree. Making nodes generic like this allows expressions of unlimited complexity to be built up by combining various types of node.

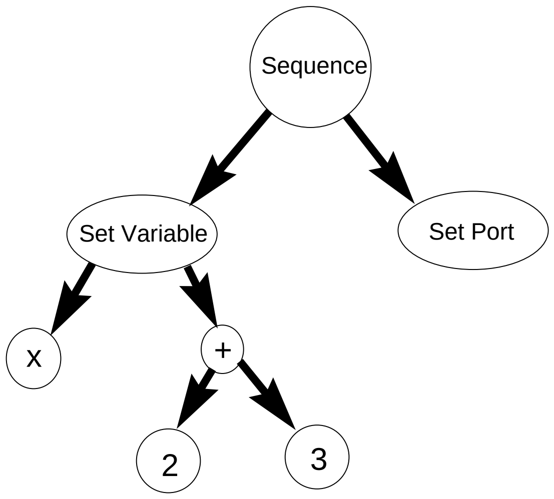

Figure 3.2: Functional approach

3.1.3 Which one?

Both the functional and object orientated approaches have both their good and bad points, and the choice between the two is explored later on in this chapter

3.2 Timing Control

One of the stated goals of the project is providing control over how long a section of code takes to run. Given the lack of processor support for this, this has to be built up from the exisiting commands to create the behaviour that we want. Timing support can be done in three different ways, either by giving a minimum time limit for a section of code, a maximum limit or an absolute i.e. making a section always take a particular amount of time, no matter what. What will determine which options we choose is the problems we will have in enforcing these conditions in code. Absolute limits can be thought of as a combination of both maximum and minimum limits, so we will just consider the maximum and minimum cases, as absolute limts are possible if both of the other two can be achieved. There are two different cases that we have to consider when working out what can be done here - code with a static execution time (i.e. a fixed sequence of instructions) and code with a dynamic execution time (looping code with a variety of constraint conditions).

3.2.1 Maximum limits

Maximum limits can be enforced on static execution time code - simply by removing any instructions on the end of the code, but this is often destructive, unpredictable and generally not very useful. Maximum limits also have a number of problems without processor support in providing limits for dynamic execution code - namely that any limits must be limited by the amount of time between checks. If, for example, we are checking whether we have hit the upper limits every 20 processor cycles, then our maximum limit must therefore be a multiple of 20, as all other values can't be enforced, as we will either check too early or too late. We can increase the rate of checking, but then we start to spend more time checking how long we've taken, and not enough time actually doing any useful work.

Because maximum limits appear to be mostly beyond our reach for this project, absolute limits also cannot be easily enforced.

3.2.2 Minimum limits

Minimum limits are much more plausible that maximum limits. For the static case, we just add an appropriate number of NOPs (No OPeration commands - telling the processor to do nothing for a cycle) to the end of the code and we know that it will take the minimum amount of time, as the time for the static code plus the time for the NOPs is our minimum time. For the dynamic case, we can keep track of how many iterations of the loop have been done, and with a knowledge of how long each iteration takes we know what the minimum number of iterations necessary to exceed the minimum execution time. At the end of the loop, if this minimum number of iterations has not been achieved, we can use NOPs again (with a loop bounded by the minimum number of iterations) to wait until the minimum amount of time has passed. This is explained in more detail in section 4.6 on page 22 - "Timing".

3.3 I/O Port handling

One of the major features of the PIC chips is a pair of output ports, totalling 13 bits of general purpose I/O that can be used for a wide variety of applications. An important design choice regarded how to present the ports to the end users - do we have special functions to read and write to/from the ports, ot do we simply have them as special cases of variables?

If the ports are to be treated as special cases of variables, then every time we do an assignment to the variable then the port is set accordingly, and reading data from the variable results in a read of the current data from a port. But then there are a couple of problems with this approach. Firstly that assigning an arbitrary, run-time derived value, to a variable stored as a register is a simple procedure in PIC assembler, using the W register as a temporary storage for the derived value.

But to set the output value of a port to a runtime derived value is a much harder matter - because the standard instructions to set a port's output value to high or low don't look at any of the other registers at all. The value that the port is about to be set to is encoded into the instruction (a BCF or BSF - Bit Set/Clear F), and so to set a port to an arbitrary value requires an if/then/else structure executing either BSF or BCF depending on our input value. This is much more complicated than the standard variable assignment case, and most of the complexity is unnecessary for the majority of applications

The other problem with ports as variables is that if we read data from a variable, and then read it again a few moments later, then unless we've changed it in the meanwhile, we would expect the value to be the same. This is not necessarily true with a port, as the input data can be continuously changing. If however we create functions that read/write from/to the ports then we force the user to deal with ports as a separate and different type of input as opposed to standard variables. This removes the complexity mentioned above, and also allows us to have stable references i.e. internal registers rather than changing ports, to values from outside the system.

3.4 Conclusions

Having gone through all of the main design issues, there are a number of other considerations that must be thought about here. Given that

- the end program has to be able to run as quickly as possible on the Bitsy (which is a relatively slow system)
- the Bitsy has very limited storage space
- it should be as easy as possible for programmers used to the cyberjacket systems to convert to using this new program

The logical choices for languages to implement this program in (and to provide the end API for) therefore are C (for functional code) and C++ (for object-orientated code) as

- Both are already being used to write programs intended for the Bitsy
- They are low enough level languages to be fast, while being of a high enough level to allow the programmer to spend more time thinking about what they want to do as opposed to dealing with all the small details about how they want to do repetitive tasks.

- Optimization of the end code can already be done, as there has been a port of the gcc code to compile code for the ARM processor used by the Bitsy.
- All the necessary basic runtime libraries are already present on the Bitsy - thus reducing additional drains on the Bitsy's limited storage space
- They are already well known to most programmers - certainly to any who would have been otherwise programming code for the PIC's in assembler. This is less the case for C++, but C code can then be used to interface to the C++, so the end programmer using the API wouldn't necessarily need to know C++, just C.

3.5 Object-orientated implementation

The object-orientated implementation has a PICCoder object which represents a block of assembler code. This has a series of methods that can be used to generate statements, expressions and variables. The code that was generated by the method calls was stored internally in the PICCoder object. Other PICCoder objects were used to represent subsections of code, as shown by the simple example below. In the example below, the PICCoder object pic2 is used as both the true and false blocks for the ifExpr (method for generating an "if/then/else" construct) method call.

```
#include <PICCoder.h>

int main() {

    PICCoder *pic = new PICCoder();
    // Our root PICCoder object
    PICCoder::Variable *x = pic->newVariable(1);
    // Create a variable, initialize it's value to 1
    PICCoder::Variable *y = pic->newVariable(4);
    // Create another variable, initialize it's value to 4
    PICCoder::Variable *z = pic->arithExpr(x,'+',y);
    // Apply the operator '+' to the variables x and y, and put the value in z
    PICCoder *pic2 = new PICCoder();
    // Make another PICCoder object
    PICCoder::Variable *n = pic2->newVariable(1);
```

// Make a new Variable with the 2nd PICCoder object. Variables in the system are in fact global in scope, and so which PICCoder we use to create the Variable doesn't matter.

pic->ifExpr(z,"=",y,pic2,pic2);

// Add an if/then/else block to the main code, with the expression "does z equal y?" and pic2 to be executed for both the "then" and "else" cases (note that reusing the same block for both is unlikely in production code, but could be optimized to just call the pic2 code automatically without doing the test of the expression)

pic->outputcode();

// Output the code from the PICCoder object.

return 0;

}

At this stage in development the functionality of the system is limited (for example there is a method for setting minimum running times for blocks of code, but it is only a stub method for storing the running time length, without any capability to try and actually make sure this happens in the end code), but there is enough there to evaluate the overall direction of the design.

There are a number of significant problems here. In the very common case where a new block containing sub-blocks (If/then/else for example, as it contains two sub-blocks for its "then" and "else") is added, it is possible that the sub-blocks haven't yet been defined fully (i.e. more commands will still be added to them) and from the design it is uncertain whether or not additional commands will be run or not.

e.g. assuming pic and pic2 are existing PICCoder objects, and that z and y are Variables (as in the above example)

```
pic->IfExpr(z,"=",y,pic2,pic2);

pic2->SetVariable(z,4);
```

What does this do? Does the IfExpr get created with the instructions stored in pic2 at the time that IfExpr is called, or does it get created with the SetVariable code added to the pic2's as well? It depends on how the instructions are stored internally, and this is something that the end user should know as little as possible about, as otherwise they could easily get confused, and also future versions of the PICCoder need to work in the same way - which should not be the case, because otherwise we can't ever change anything, as we don't know what the user is depending on. We need to make sure that everything the user knows about the system is the API that we export - that needs to stay stable (or at least backwards compatible), but our internal implementation should be changeable.

This problem is caused by the fact that we are using objects to represent blocks of code, and so when a user specifies an object, the system could store the link as to the object by value or by reference, and either way has the potential to cause confusion as the user has to know which one we're using.

Given that storing our code as C++ objects is going to cause these problems, and that if we're not going to use objects at all, then the small speed increases of C over C++ (especially on a low specification system like the Bitsy) make the language change worthwhile and so the C++ implementation was discarded.

3.6 Functional Implementation

The initial version of the functional implementation used code like displayed below.

This is remarkably similar to the code used for the C++ implementation, with CodeNode structures replacing the previous references to both Variable's and PIC-Coder objects. This version still has a lot of the problems of the C++ implementation, as the addToBlock() method for example alters the "block" variable in situ, indicating that a stable reference to the "block" variable exists, and therefore still allowing the problems with the C++ implementation. A good example of this is the set_minsize function - this actually sets the minimum runtime for everything that ever gets placed in the "block" variable, not just whatever has been stored in there at the current time.

```
#include <PICCoder.h>

int main() {

    pic_start(); { // initialisation of the PIC library

        CodeNode *x = newVariable(1); // Create a new variable with initial value
            1
        CodeNode *block = newBlock(); // Create a new code block
        addToBlock(block,ifExpr(buildBoolean(newConstant(1),"==",newConstant(4)),newStub(),newStu

        // build an if/then/else expression with the boolean expression "1 == 4"
            and a pair of stub blocks (blocks with no content) as the "then" and
            "else branches
        addToBlock(block,setVar(x,newConstant(5))); // add a “set the Variable
            x to 5” statement to the “block” code block
        set_minsize(block,30); // set the minimum runtime of the “block” code
            block to 30 cycles
        outputcode(block); // output the code from the code block “block”
    }
    pic_destroy();
    return 0;
}
```

Initial C implementation

This was then adapted to try and remove some of the ambiguities. Important changes include

- the pic_start() and pic_destroy() code has now been folded into outputcode, as none of the internal structures that are in pic_start/pic_destroy are used either before or after outputcode.

- addToBlock() has been renamed to makeBlock(), and now returns a reference to a new block consisting of everything in the first argument block along with everything in the second argument’s block, which allows for the use of nested function calls to create complex structures, without the need for variable declarations for each part.

- set_execcycles() (was set_minsize() in the previous implementation, renamed for clarification purposes) now also returns a reference to a new block - a version of the old block that has been surrounded with a “wrapper” structure that indicates that it’s internal block should take a particular minimum number of cycles to execute.

- Block types have now been altered completely. The old CodeNode (with it’s “one size fits all” mentality) has been discarded in favour of S_Node (Statement Node - both representing a statement and a sequence of statements chained together), V_Node (Variable reference), E_Node (Expression Node - not explicitly declared here, but the VtoE function is used to provide an E_Node reference to a variable, and the newConstant() function returns a E_Node) and B_Node (Boolean expression - outputted by buildBoolean in the example)

- The alteration of the block types, along with the changing of buildBoolean to take a enum specified value (BOOL_EQ) for the operator (this is also the case for the arithmetic operation creation method arithExpr, but this is not used in the example) has vastly reduced the number of programs that would compile but wouldn't run. Arguments to methods can now be limited to only the type that is a valid value of the argument (numeric expressions for each side of a boolean expression for example). This results in a lot of potential bugs in code being caught a lot earlier, and at compile time, rather than being caught after the end program has been loaded onto the Bitsy.

```
#include <PICCoder.h>

int main() {

    V_Node *x = newVariable(NULL);
    V_Node *y = newVariable(NULL);
    S_Node *_master = makeBlock(setVar(x,newConstant(8), setVar(y,newConstant(12)));
    S_Node *_t1 = set_execycles(setVar(x,newConstant(12))),20);
    S_Node *_t2 = setVar(x,newConstant(12));
    _master = makeBlock(_master, ifExpr(buildBoolean(VtoE(x), BOOL_EQ, VtoE(y)),_t2,
        _t1));
    outputcode(_master);
    return 0;

}
```

Second C implementation

3.7 Higher-level language

While working on the functional implementation, an attempt at designing a higher-level language to make it easier to write the C code for the functional implementation was experimented with. The attempt revolved around ideas about using Lex, Yacc and a tree-walker generator called Memphis to make a Pascal-like language that could be used to think of the code at an even simpler and more abstracted level than the increasingly complex C code. An example of the code is below. Most of it should be fairly self-explanatory, except the [number]'s after "}" - this indicates that the block ended with the “}” should take at least “number” cycles to execute. This was eventually rejected, because of the limited number of additional features that this added, and the additional time needed to keep this and a rapidly-developing functional API in sync with each other.

```
// greatest common divisor
{
    x = 8
    y = 12
    while (x != y)
    {
        if (x > y)
        {
            x = x-y
        } [30]
        else {
            y = y-x
        } [25]
    }
    if (x > 2) {
        set porta,1
    }
}
```

Chapter 4

API Description

In this chapter we look at the API that we have produced. The API provides a series of different functions and data for creating commands for the PIC, by allowing the end-user programmer to combine different structures together to form a complete code tree, and to generate PIC code from that tree.

4.1 Node Types

- V_Node - V(ariable) node - a reference to a variable stored in a register
- E_Node - E(xpression) node - this is anything that can have a value returned from it. It could be a constant, a V_Node, an arithmetic expression or it could even be a reference to one of the I/O ports of the PIC
- S_Node - S(tatement) node - these can be NULL, or they could be a single statement, or a set of statements. A NULL is treated as a series of statements of length 0. For multiple statements, these are chained together in a linked list formation.

4.2 Variables and Constants

V_Node* newVariable(E_Node *init)

newVariable() creates a new V(ariable) node, initialised to the result of the E(xpression) node init. init can be NULL, in which case the variable is uninitialised, and any reads from it before a write will result in an undefined result.

E_Node* VtoE(V_Node *init)

VtoE creates an E_Node reference to a V_Node - because all V_Node's can become valid E_Node's (because they can be read from) but not all E_Node's are valid V_Node's (as not all expressions can be written to, only the one's that are wrappers around a V_Node)

E_Node* newConstant(DATA init)

newConstant creates a reference to a constant value, of type DATA. DATA is defined as to limit the valid values for "init" to between -128 and +127 (which are the integer limits for simple calculations on the PICs)

S_Node* setVar(V_Node *a,E_Node *setme)

setVar sets the V_Node "a" to the output of the E_Node "setme". This allows for all types of assignment, from constants and incrementing of V_Node's, through to complex mathematics.

S_Node* setVariableBit(V_Node *input, unsigned int bit)

S_Node* clearVariableBit(V_Node *input, unsigned int bit)

set/clearVariableBit() set/clear individual bits of a V_Node, allowing for precision alterations of a V_Node for a variety of uses.

V_Node* getTimer()

getTimer provides one of the special case variables. The V_Node returned by this function is in fact a reference to the TMR0 register allowing reading and writing to the PIC's internal timer module, which increments every instruction cycle.

S_Node* rotateVarLeft(V_Node *input)

S_Node* rotateVarRight(V_Node *input)

rotateVarLeft/Right performs a rotate shift on the specified V_Node, either to the Left or to the Right (depending on which method has been called). This rotate is performed in conjunction with the Carry bit of the Status register, so the use of noCarry may well be useful in combination with this instruction.

4.3 Expressions

E_Node* arithExpr(E_Node* x,arithop c,E_Node* y)

arithExpr creates an arithmetic expression, by combining a pair of E_Node's with a binary arithmetic operator. Valid values for arithop are

- ARITH_PLUS - addition
- ARITH_MINUS - subtraction
- ARITH_TIMES - multiplication
- ARITH_DIVIDE - division

This results in another E_Node which could then be given to another call of arithExpr to build up more complex operations.

B_Node* buildBoolean(E_Node *a,booleanop op, E_Node *b)

buildBoolean creates a boolean expression, by combining a pair of E_Node's with a binary comparison operator. Valid values for booleanop are

- BOOL_EQ - equality
- BOOL_NE - not equal
- BOOL_LT - less than
- BOOL_LE - less than or equal to
- BOOL_GT - greater than
- BOOL_GE - greater than or equal to

The resulting B(oolean)_Node is used for if and while expressions. Building more complicated boolean expressions is not possible with the API currently, mainly because this is something that is very rarely used.

B_Node* testVariableBit(V_Node *input, unsigned int bit)

testVariableBit creates a boolean expression that returns True if the nth bit of "input" (where n = "bit") is '1', and False if it is '0'.

B_Node* noCarry()

noCarry() is intended as an addition for easier conversion of existing assembler code. The B_Node that noCarry returns will return True if the Carry bit is not set, and False if it is. This is most commonly used in combination with the rotateVarLeft/Right methods.

4.4 Blocks

S_Node* makeBlock(S_Node* first, S_Node* second)

makeBlock returns an S_Node consisting of all of the contents of “first” followed by all of the contents of “second”. Note that both first and second could be NULL, or they could be a single statement, or a set of statements, because the API allows all of these. If either is NULL, then this can be thought of as a S_Node with no statements in at all, and so all the possibilities for the input S_Node’s can be dealt with appropriately.

4.5 Control Statements

S_Node* ifExpr(B_Node *a,S_Node *dotrue, S_Node *dofalse)

ifExpr creates an if/then/else expression. The B(oolean)_Node a is evaluated, and if it results in true, then the S_Node “dotrue” is executed, otherwise “dofalse” is executed. Note that either (or both) dotrue and dofalse can be NULL if you wish to not do anything on a particular branch.

S_Node* whileExpr(B_Node *a, S_Node *block)

whileExpr creates a while loop. The B_Node a is evaluated, and while it is true, the S_Node “block” is repeatedly executed. “block” can be NULL if you want to create a loop that waits for something.

4.6 Timing

S_Node* set_execcycles(S_Node *c, unsigned int s)

set_execcycles returns an S_Node that will execute the S_Node "c" with a guarantee that it will take at least "s" processor cycles to execute. In order to do this guarantee there are a number of problems that must be dealt with, but the one that may cause this command to fail is when it is impossible to determine how long a section of code will take to run. This happens with while loops, which will take an amount of time that is unknown at compile time, as their execution time depends on how many loops are done, and this isn't something that is generally known until runtime. Therefore, set_execcycles will work either with S_Node's that do not contain any while loops at all, or with S_Node's that are the return value from a whileExpr - these will have a guaranteed minimum runtime. How do we achieve a fixed minimum runtime for a section of code with a dynamic runtime?

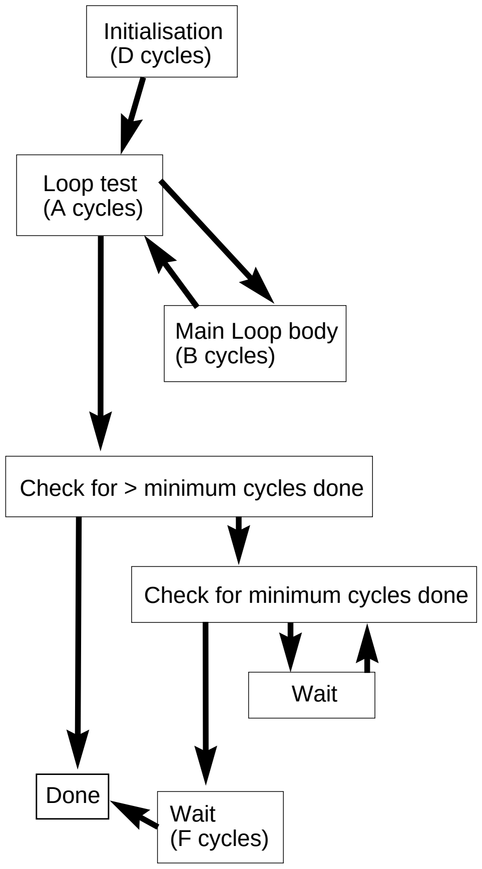

Figure 4.1: While loop diagram

4.6.1 While loop timing

A while loop with an overall guaranteed minimum runtime can be created by the following algorithm

- calculate the runtime of the boolean expression and a conditional jump that will be evaluated on each loop around, assuming that it doesn't jump - this is A
- calculate the runtime of each iteration of the while loop - not including the boolean expression check, but including the branch back to the beginning - this is B. This must be a constant value, so no while loops inside this code are allowed, and any if /then/else statements have the shorter of their two branches padded to make sure that both branches always take the same time.

- C is the minimum runtime
- If A < C then we add to the main code an increment of a counter and re-evaluate B. Otherwise, the while loop code is output now as just a standard while loop without any timing code.
- D is the length of code needed to initialise the counter at the beginning of the loop
- E is the minimum number of iterations needed to meet the minimum runtime - equal to (C - D)/(A+B), then rounded down to the nearest integer
- F is the remainder of (C - D)/(A+B) (i.e. [(C - D)/(A+B)] - E), subtract (A+1) - so the net result is [(C - D)/(A+B)] - E - (A+1). (We add one to the time for the last iteration of A, as the last check of the boolean expression must have failed in order for an exit from the while loop to have occurred and conditional jumps on the PIC take 2 cycles if taken, or 1 if not.)

The end code for the while loop is therefore this set of instructions in the order below

- Initialisation of the counter (should take D cycles)
- The boolean expression check, with a branch over the main loop if we get a false result, as per standard while loops (should take A cycles, except on the last check which gets the false result, which will take A+1)
- The main loop - including counter increment code (should take B cycles)
- If our counter is now greater than E, jump beyond all of the code listed below, as we've done more than the needed number of iterations.
- The following set of instructions is then executed until the test fails - this subloop must take A+B cycles per iteration
  - the counter variable is checked against the value E, and if the counter is equal to E, then we exit this loop
  - the counter is incremented
  - a series of NOPs - number of which will be altered to fit the size requirements of this loop - some of this may be looped e.g. write commands to loop around 5 times at 10 cycles/iteration rather than writing out 50 NOPs for example
  - a branch back to the start of this sub-loop

- Another series of NOPs (or loops containing NOPs) will be executed, except this one is F cycles long.

So therefore,

$time\ taken = D + [(number\ of\ iterations\ of\ main\ loop + iterations\ of\ padding\ loop) * (A+B)] + (A+1)$

(plus F instructions if we had exactly the minimum number of loops, to make up the minimum number of cycles)

which is greater than or equal to the minimum execution time, because

- the total number of loops (both the main and padding loops) is greater than or equal to E - call this G
- so what gets executed is
  - D cycles to initialise the counter
  - G iterations of A+B cycles i.e. either the main loop with a succeeded boolean check or a padding section of the same length
  - A+1 cycles for the failed boolean check, because the test must have failed once, otherwise we never would have exited the while loop
  - F instructions of end padding if needed

which gives us a while loop that will take at least the minimum number of instructions.

e.g. For a while loop that needs to take at least 110 cycles, with an A time (boolean expression + failed conditional jump) of 5, a B (main loop) time of 13 and assuming that D (counter initialisation) is 2 cycles.

- A < C so we need the timing code
- Adding the counter code to the main loop increases B from 13 to 15
- E (minimum iterations) = (C - D)/(A+B) = (110-2)/(5+15) = 5.4. Rounded down to 5
- F (remainder of the E calculation, minus (A+1)) = [(C - D)/(A+B)] - E - (A+1) = [(110-2)/(5+15)] - 5 - (5+1) = 8 - 6 = 2 cycles

4.7 Port Commands

S_Node* setPort(pic_port port, int pin)

S_Node* unsetPort(pic_port port, int pin)

These sets one of the I/O pins of the PIC to being an output pin, and to either outputting a “1” signal (for setPort) or a “0” signal (for unsetPort). The value of port is either PORT_A (for the A I/O port) or PORT_B (for the B I/O port). The pin is either in the range 0-7 for port A, or 0-3 for port B.

E_Node* readPort(pic_port port, int pin)

readPort sets the specified port (specification as for setPort/unsetPort), and then reads the current input signal to that port (either 0 or 1). The result of which is the value of the return E_Node.

4.8 Output

void outputcode(S_Node *top)

This outputs the code for the specified S_Node onto standard output.

Chapter 5

Comparison with existing code

The intention of this project has been to provide an easier way to write code for the PICs, and giving more features to the programmer. In this chapter we will look at a real-world existing example of an existing program for one of the sensors, and a version of the same program written using the API from this project.

5.1 Existing assembler code

The existing assembler program that we are comparing against is for one of the accelerometer sensors for the cyberjackets. The original program was written by Chris Djiallis, and parts of the code are also based on earlier work by Cliff Randell. The full text of the assembler program is in Appendix A.

5.2 New C code

The new program has been written by myself using the new API. Some parts of it have been translated directly across from the original code, and others have been re-written to better use various features of the API. A number of pieces of code that would never have been run in the original code have been removed entirely. The names of the macros have been mostly retained as function names for the various sections of code, in order to ease a comparison between the two different methods of coding. The full text of the API-using C code is in Appendix B.

5.3 Improvements

Various features of the new C code were particularly good examples of how the conversion helped the readability and maintainability of the program

- the XMIT function - in the assembler, the first part of XMIT is reliant on the byte to be sent being already loaded into W. In the C version, the fact that XMIT effectively takes an argument is much more explicitly specified as the XMIT function is defined as taking an E_Node as input.

  Assembler code for XMIT:

  ```
  XMIT MOVWF SER_TX
      MOVLW NUMBIT+1
      MOVWF BITCNT
      TXLOW
      GOTO XMITC
  XMITA RRF SER_TX,1
      SKPNC
      GOTO XMITB
      TXLOW
      GOTO XMITC
  XMITB TXHI
  XMITC WAITEM
      TSTF BITCNT
      SKPNZ
      RETURN
      DECFSZ BITCNT,1
      GOTO XMITA
      TXHI
      GOTO XMITC
  ```

C Code for XMIT:

```
  S_Node *XMIT(E_Node* tostore)
{
          S_Node *ret = makeBlock(
                 setVar(SER_TX,tostore),
                 setVar(BITCNT,newConstant(NUMBIT+1))
          );
          ret = makeBlock(ret,TXLOW());
          return makeBlock(ret,whileExpr(buildBoolean(VtoE(BITCNT),BOOL_EQ,newConstant(0)),XMITC()
}
  S_Node* XMITC()
{
          S_Node *doifNZ = makeBlock(
                 setVar(BITCNT,
                        arithExpr(VtoE(BITCNT),ARITH_MINUS,newConstant(1))
                 ),
                 ifExpr(buildBoolean(VtoE(BITCNT),BOOL_EQ,newConstant(0)),
                        makeBlock(
                            ifExpr(testVariableBit(SER_TX,0),TXLOW(),TXHI()),
                            rotateVarRight(SER_TX)
                        ),
                        TXHI()
                 )
          );
          return makeBlock(doifNZ,WAITEM());
}
```

The C Code is easier to read, easier to maintain, and is clearly specified - it is clear from just the first line of the function prototype what data is going into the function. With the assembler, you can read that the first line of the macro stores whatever is currently in W, and it is possible to deduce that therefore you should make sure that something useful is in W before you call this macro, but this is not something that the language makes clear.

- the MUL function - this has been removed entirely from the C version of the code, as the API provides a built in way to multiply two numbers together. Multiplying two numbers together is something that happens frequently in programs. It shouldn't be necessary to put the code to do this standard function in every program that uses it, it should be supplied by the language that the program is written in. The C version simply adds in the multiply code to its output whenever it is actually needed, without needing end-user intervention.

- the BITIN function - the C function BITIN is a combination of the code for the BITIN1 and BITIN2 macros in the assembler, as they only ever differed by one argument to a single instruction, but in assembler combining them would be have been more effort that it was worth.

  Assembler code for BITIN1:

```
BITIN1 MACRO
        MOVLW BAUDSET
        MOVWF TMR0
BITINA1 TSTRTC
        SKPZ
        GOTO BITINA1
        RRF SER_RX,1
        BCF SER_RX,7
        BTFSC PORTB, RXD1
        BSF SER_RX,7
        ENDM
```

C Code for BITIN (note that this replaces both BITIN1 and BITIN2):

```
 S_Node *BITIN(int pin)
{
        S_Node *ret = setVar(getTimer(),newConstant(BAUDSET));
        ret = makeBlock(ret,whileExpr(buildBoolean(VtoE(getTimer()),BOOL_EQ,newConstant(0)),NULL));
        ret = makeBlock(ret,rotateVarRight(SER_RX));
        ret = makeBlock(ret,ifExpr( buildBoolean(readPort(PORT_B,pin),BOOL_EQ,newConstant(0)),
              clearVariableBit(SER_RX,7),
              setVariableBit(SER_RX,7)
              ));
        return ret;
}
```

The C code, by replacing both the functions not only reduces overall code size, it decreases the amount of maintenance needed to alter the program as any changes to the BITIN routine only need to be done in one place, as opposed to 2 for the assembler version.

Even allowing for reasonable levels of commenting in both of the programs, the C version is almost 50% smaller (270 lines v.s. 523 lines for the the assembler version), because the C version can do more work for each method, rather than having to enter in 2 or 3 assembler instructions in order to do even simple tasks.

Chapter 6

Conclusions

In this chapter we will look at what we have managed to achieve. Did we achieve all our goals, and how could we improve on what we have done?

6.1 Goals

We achieved the main goals of the project. A faster method of writing PIC code has been written. The API is not only sufficiently broad to handle most of the standard features of the PIC microcontrollers, it also adds various features of its own. Timing control has been implemented, and works for code that has dynamic execution times, without the end-user having to deal with all the complexities that that causes. While loops and if/then/else structures, with a full set of basic boolean operatives have also been added. Complex mathematical expressions can be easily built up, and the multiplication and division operators (albeit for integer numbers only) have been added to the choices available.

6.2 Further development

The generated assembler from the API functions could be further optimised, but this would not require alterations to the end-user accessible functions as documented in Chapter 4. One important optimisation would be the use of CALL to implement repeated function calls. Various PIC features are still missing from the API (interrupt handing for example), but there is a sufficient subset of the functionality present to allow a wide variety of programming with the PIC.

One interesting additional feature would be non-integer numbers. These would have to be handled internally as integers, but would provide a greater degree of flexibility to the users.

6.3 Conclusions

One example of what this project's API can be used to implement is programs that change what code they output depending on the current state of the Bitsy, for example changing for different times of day or different code for the 1st sensor of a series as opposed to later ones - this could be used for auto-generating ID numbers for the sensors.

It is an enabling technology, allowing greater expression of the end-user's wishes by reducing the amount of work required to do simple operations, freeing them to spend more time thinking about what they want to do, as opposed to having to deal with the minor details of how standard, machine-generatable parts of the program will work. The options for what can be done using the API are endless - the only limit is the imagination of the end-user programmer.

Bibliography

[JAL]	http://www.voti.nl/jal/
[Aino]	http://personal.eunet.fi/pp/jokinen/
[C2C]	http://www.picant.com/c2c/c.html
[MicroSeeker PIC/Smalltalk]	http://www.huv.com/uSeeker/smalltalk/pic.html

Appendix A - Accelerometer Assembler code

```
; acc.asm by Chris Djiallis based on gps.asm by Chris Djiallis and MultiPIC.asm by Cliff Randell
;
; DATE                  July 2001
; ITERATION             0.0
; FILE SAVED AS         acc.asm
; FOR                   PIC16F84A       PWRTE=off WDT=off CP=off
; CLOCK                 10 MHz Crystal
; INSTRUCTION CLOCK     2.5 MHz T= 400ns

        TITLE "acc.asm − PIC interface for Accelerometer device to PC (filtered) comms"

        LIST            P=16F84A

        __config        3FF9h

;---------------------------------------------------------------------------
;
;       This program is intended to provide an interface between
;       an Accelerometer and a processor with a serial RS232 port.
;
;       Requests from the processor are in the form of $SA... are responded to
;       with data also in the form $RA*******.
;
;       The format of the serial data is 4800 baud, 1 start, 1 stop and 8
;       data bits with no parity.
;
;       For use with DS275 RS232 Serial interface chip,
;       i.e uses normal polarity − not inverted.
;
;
;---------------------------------------------------------------------------
;       Generic Definitions
;---------------------------------------------------------------------------

INDF                    EQU     H'0000'
TMR0                    EQU     H'0001'
STATUS                  EQU     H'0003'
FSR                     EQU     H'0004'
PORTA                   EQU     H'0005'
TRISA                   EQU     H'0005'
PORTB                   EQU     H'0006'
TRISB                   EQU     H'0006'

RP0                     EQU     5

;-----------------------------------------------------------------------------
;
;                          Physical Port Assignment
;-----------------------------------------------------------------------------

COMPORT         EQU     PORTA           ; Serial comms ports

;-----------------------------------------------------------------------------
;
;                          Physical Bit Assignment
;-----------------------------------------------------------------------------

TXD     EQU     .1              ; Location of data on port A
RXD1    EQU     .0
X_IN    EQU     .1
Y_IN    EQU     .0


;-----------------------------------------------------------------------------
;
;                          Constant Assignment
;-----------------------------------------------------------------------------

MEMBAS  EQU     008H            ; Start of user ram registers

        ; Baud rate and software uart constant definitions

BAUDIV  EQU     b'00000010'     ; Rtcc setting for 3.2us tick with 10MHz clock
BAUD    EQU     .61             ; 4800baud−208us 2400baud−416us 1200baud−833us
                                ; 65 ticks of 3.2us = 208us
                                ; reduced to 61 on test (instruction delay)

BAUDSET EQU     100H−BAUD               ; Element time
BAUDHAF EQU     BAUDSET+BAUD/2          ; Half element time
BAUDSTR EQU     100H−BAUD−BAUD/2        ; (1 + 1/2) element time to drop
                                        ; start bit on data receive
NUMBIT  EQU     .8                      ; Number of transmit serial data bits

;-----------------------------------------------------------------------------
;
;                        Variable Assignment
;-----------------------------------------------------------------------------


SER_RX  EQU     MEMBAS          ; Serial shift register for data rx
SER_TX  EQU     SER_RX+1        ; Serial shift register for data tx
BITCNT  EQU     SER_TX+1        ; Counter of bits received
```

```
FLAG    EQU     BITCNT+1        ; 8 general purpose flag bits held here

ITERATOR   EQU  FLAG+1          ; |
COEFF      EQU  ITERATOR+1      ; |
COUNT      EQU  COEFF+1         ; > used by MUL (multipy) routine
RESULT     EQU  COUNT+1         ; |

ASC        EQU  RESULT+1        ; used by ASC2DEC routine

COUNTER    EQU  ASC+1
T1_X       EQU  COUNTER+1
T1_Y       EQU  T1_X+1
T2         EQU  T1_Y+1

;---------------------------------------------------------------------------
;                               Reset and start
;---------------------------------------------------------------------------

        ORG     00
        GOTO    SETUP

;---------------------------------------------------------------------------
;                               Macros
;---------------------------------------------------------------------------

;******
;       TSTRTC moves TMR0 to W reg and sets ZERO status

TSTRTC  MACRO
        MOVF    TMR0,0          ; Test for timeout
        ENDM

;******
;       TXLOW sets the transmit data RS232 line 1 low for a MARK

TXLOW   MACRO
        BCF     PORTB, TXD
        ENDM

;******
;       TXHI sets the transmit data RS232 line 1 high for a SPACE

TXHI    MACRO
        BSF     PORTB, TXD
        ENDM

;******
;       BITIN gets a bit from serial input and shifts into bit 7 of 'SER_RX'.
;       First the state of the serial input line is monitored for change
;       and if it changes the clock is re-synced.
;       Enter with clock sync'd TO LAST SAMPLE and rtc set for 1/2 element.
;       When the timer expires the data is read and on exit the carry is set
;       to the state of bit 0 of 'SER_RX' before the shift which is used to
;       monitor position of the bit count flag bit.

BITIN1  MACRO
        MOVLW   BAUDSET         ; Load 1 element time -
        MOVWF   TMR0            ; - and preset the baud rate divider

BITINA1 TSTRTC
        SKPZ                    ; Skip if rtc was zero
        GOTO    BITINA1         ; Loop until timer zeroes

        ; Timer has zeroed, shift SER_RX, sample data and exit

        RRF     SER_RX,1        ; Shift data to make way for next bit
        BCF     SER_RX,7        ; Clear Serial bit 7
        BTFSC   PORTB, RXD1     ; Test receive data
        BSF     SER_RX,7        ; Set Serial bit 7 - data was high i.e. a '1'
        ENDM                    ; Exit from BITIN1

;------------------BITIN2---------------

BITIN2  MACRO
        MOVLW   BAUDSET         ; Load 1 element time -
        MOVWF   TMR0            ; - and preset the baud rate divider

BITINA2 TSTRTC
        SKPZ                    ; Skip if rtc was zero
        GOTO    BITINA2         ; Loop until timer zeroes

        ; Timer has zeroed, shift SER_RX, sample data and exit

        RRF     SER_RX,1        ; Shift data to make way for next bit
        BCF     SER_RX,7        ; Clear Serial bit 7
        BTFSC   COMPORT,RXD2    ; Test receive data
        BSF     SER_RX,7        ; Set Serial bit 7 - data was high i.e. a '1'
        ENDM                    ; Exit from BITIN2

;******
;       WAITEM waits for one element time.

WAITEM  MACRO
        LOCAL   WAIT_1
        MOVLW   BAUDSET         ; Get the baud divider preset --
        MOVWF   TMR0            ; -- and preset the baud rate divider

WAIT_1  TSTRTC                  ; Read the RTC divider
        BNZ     WAIT_1          ; Loop until divider zeroes
        ENDM


;-----------------------------------------------------------------------------
;                               Subroutines
;-----------------------------------------------------------------------------

;******
;       XMIT sends the 8 bit (or less) byte in W reg to the transmit data
;       port as an async data byte with 1 start and 1 stop bit with the least
;       significant bit first.

XMIT    MOVWF   SER_TX          ; Data shifter
        MOVLW   NUMBIT+1        ; Get the bit count + 1 for start bit
        MOVWF   BITCNT          ; Preset data bit (down) counter

;       Send the start bit

        TXLOW                   ; Set start bit level
        GOTO    XMITC           ; Wait for start element to go

;       Set the transmit data level from the carry and wait for an element

XMITA   RRF     SER_TX,1        ; Clock shift register RIGHT through carry
        SKPNC                   ; If data (carry) is '0', skip
        GOTO    XMITB
        TXLOW                   ; Data is '0'
        GOTO    XMITC
XMITB   TXHI                    ; Data is '1'
XMITC   WAITEM                  ; Wait for the element to go

;       Count the elements as they are sent

        TSTF    BITCNT          ; Zero if just sent the stop bit
        SKPNZ                   ; Skip next if bit count is not zero
        RETURN                  ; Exit from XMIT

        DECFSZ  BITCNT,1        ; Dec. bit count, skip if zero
        GOTO    XMITA           ; Loop until all bits are sent

;       Bit count has zeroed, send the stop bit

        TXHI                    ; Set stop bit
        GOTO    XMITC           ; Wait for the stop bit to go

;******
;       RXBYTE receives and dumps the start bit. Receives 8 bits into
;       'SER_RX' with the most significant bit received first and then dumps
;       the stop bit.

;       First, test for the LOW start bit at the beginning of the byte.

RXBYTE1 BTFSC   PORTB, RXD1     ; Test receive data
        GOTO    RXBYTE1         ; Data is still high so loop until
                                ; start bit (low) is seen.

        MOVLW   BAUDHAF         ; Load half element time -
        MOVWF   TMR0            ; - and preset the timer

WAIHAF1 TSTRTC                  ; Read the RTC divider
        BNZ     WAIHAF1         ; Loop until divider zeroes

        BTFSC   PORTB, RXD1     ; Test receive data is still low
        GOTO    RXBYTE1         ; Data is high so loop until a real
                                ; start bit is seen.

;       Start edge seen (low level) - set timer for 1 elements which
;       causes BITIN to go to first data bit.

        MOVLW   BAUDSET         ; Load one element time -
        MOVWF   TMR0            ; - and preset the timer
        MOVLW   b'10000000'     ; Bit 7 is shifted with data and flags 8 rx'd.
        MOVWF   SER_RX          ; Initialise serial data register
RXBIT1  BITIN1                  ; Get a data bit
        BNC     RXBIT1          ; Loop until the 'carry' bit appears
```

```
;       Received character is in SERIAL, now dump the stop bit

        WAITEM                  ; Dump stop bit
        RETURN                  ; Exit from RXBYTE1


;------------- RX_MASK ---------------
;
;       >> receive mask data
;
;------------- RX_MASK ---------------

RX_MASK CALL    RXBYTE1         ; read first char of Lat data
        MOVF    SER_RX,0
        ;MOVWF  LATD1
        RETURN


;------------- MUL ---------------
;
;    i.e. result = coeff x iterator,
;         iterator set before call,
;         coeff set before call
;
;------------- MUL ---------------

MUL     CLRF    COUNT          ; count = 0
        CLRF    RESULT         ; result = 0
LOOP    MOVF    RESULT, 0      ; W = result
        ADDWF   COEFF, 0       ; RESULT = W + COEFF
        MOVWF   RESULT
        INCF    COUNT, 0       ; COUNT++
        MOVWF   COUNT
        XORWF   ITERATOR, 0    ; test if count = iterator
        BTFSS   STATUS, 2
        GOTO    LOOP           ; test is false
        RETURN                 ; test is true


;------------- ASC2DEC ---------------
;
;    >> ascii char in ASC before call
;    >> decimal value in W on return
;
;------------- ASC2DEC ---------------

ASC2DEC MOVLW   .48
        SUBWF   ASC, 0
        RETURN


;------------- PROC_MK ---------------
;
;    >> process mask data
;
;------------- PROC_MK ---------------

PROC_MK MOVLW   .10
        MOVWF   ITERATOR        ; iterator = 10
        ;MOVF    LATD1, 0
        MOVWF   ASC
        CALL    ASC2DEC         ; after call, decimal value of LATD1 is in W
        MOVWF   COEFF           ; COEFF = ASC2DEC(LATD1)
        CALL    MUL

        ;MOVF    LATD2, 0
        MOVWF   ASC
        CALL    ASC2DEC
        ADDWF   RESULT, 0
        ;MOVWF   M_DEG           ; M_DEG contains decimal value of mask degrees value

        MOVLW   .10
        MOVWF   ITERATOR        ; iterator = 10
        ;MOVF    LATM1, 0
        MOVWF   ASC
        CALL    ASC2DEC         ; after call, decimal value of LATM1 is in W
        MOVWF   COEFF           ; COEFF = ASC2DEC(LATM1)
        CALL    MUL

        ;MOVF    LATM2, 0
        MOVWF   ASC
        CALL    ASC2DEC
        ADDWF   RESULT, 0
        ;MOVWF   M_MIN           ; M_MIN contains decimal value of mask minutes value

        RETURN
```

```
;------------- TX_DATA ---------------
;
;    >> transmit accel to PC
;
;------------- TX_DATA ---------------

TX_DATA MOVLW   '$'
        CALL    XMIT

        MOVLW   'R'
        CALL    XMIT

        MOVLW   'A'
        CALL    XMIT

        MOVLW   '0'
        CALL    XMIT

        MOVF    T1_X, 0
        CALL    XMIT

        MOVF    T1_Y, 0
        CALL    XMIT

        MOVF    T2, 0
        CALL    XMIT

        MOVLW   ';'
        CALL    XMIT

        MOVLW   '\n'
        CALL    XMIT

        RETURN


;------------- READACC ---------------
;
;    >> read acceleration
;
;------------- READACC ---------------

READACC CLRF    COUNTER         ; set counter to zero
READ_X  BTFSC   PORTA, X_IN     ; wait for X_IN to be low (for synchronisation)
        GOTO    READ_X

WAIT_H  BTFSS   PORTA, X_IN     ; wait for first rising edge
        GOTO    WAIT_H

LOOP_X  NOP                     ; start counter / delay
        NOP
        ;NOP
        INCF    COUNTER, 1
        BTFSC   PORTA, X_IN     ; wait for first falling edge
        GOTO    LOOP_X
        MOVF    COUNTER, 0      ; T1_X = COUNTER
        MOVWF   T1_X

        CLRF    COUNTER         ; set counter to zero

READ_Y  BTFSC   PORTA, Y_IN     ; wait for Y_IN to be low (for synchronisation)
        GOTO    READ_Y

WAIT_H2 BTFSS   PORTA, Y_IN     ; wait for first rising edge
        GOTO    WAIT_H2

LOOP_Y  NOP                     ; start counter / delay
        NOP
        ;NOP
        INCF    COUNTER, 1
        BTFSC   PORTA, Y_IN     ; wait for first falling edge
        GOTO    LOOP_Y
        MOVF    COUNTER, 0      ; T1_Y = COUNTER
        MOVWF   T1_Y

LOOP_Y2 NOP
        NOP
        ;NOP
        INCF    COUNTER, 1
        BTFSS   PORTA, Y_IN     ; wait for second rising edge
        GOTO    LOOP_Y2
        MOVF    COUNTER, 0      ; T2 = COUNTER
        MOVWF   T2
        MOVF    T1_X, 0;
        ;SUBLW  .65             ; calibrate X
        MOVWF   T1_X

        MOVF    T1_Y, 0;
        ;SUBLW  .65             ; calibrate Y
        MOVWF   T1_Y

        RETURN


;---------------------------------------------------------------------------
;                       Cold start setup
;---------------------------------------------------------------------------

SETUP   CLRF    FSR
        TXHI                    ; Idle the transmit line as a mark

        BSF     STATUS,RP0      ; Select Bank 1
        MOVLW   b’00000011’     ; Set Port A
        MOVWF   TRISA
        MOVLW   b’00000001’     ; Set Port B
        MOVWF   TRISB
        MOVLW   BAUDIV          ; Set TMR0 for baud rate tick
        MOVWF   TMR0
        BCF     STATUS,RP0      ; Select Bank 0


;---------------------------------------------------------------------------
;                       Main program start
;---------------------------------------------------------------------------

        BSF     PORTB, TXD
MAIN    CALL    RXBYTE1
        MOVF    SER_RX,0
        XORLW   ’$’             ; test if $ symbol
        BTFSS   STATUS, 2
        GOTO    MAIN

        CALL    RXBYTE1
        MOVF    SER_RX,0
        XORLW   ’S’             ; test if S symbol
        BTFSS   STATUS, 2
        GOTO    MAIN

        CALL    RXBYTE1
        MOVF    SER_RX,0
        XORLW   ’A’             ; test if A symbol
        BTFSS   STATUS, 2
        GOTO    MAIN

        CALL    RXBYTE1
        MOVF    SER_RX,0
        XORLW   ’0’
        BTFSS   STATUS, 2
        GOTO    MAIN


        CALL    RXBYTE1
        MOVF    SER_RX,0
        XORLW   ’H’             ; test if P symbol
        BTFSS   STATUS, 2
        GOTO    JUMP
        GOTO    SETHIGH

JUMP    MOVF    SER_RX,0
        XORLW   ’L’             ; test if P symbol
        BTFSS   STATUS, 2
        GOTO    JUMP2
        GOTO    SETLOW

JUMP2   MOVF    SER_RX,0
        XORLW   ’P’             ; test if P symbol
        BTFSS   STATUS, 2
        GOTO    MAIN

        CALL    READACC         ; read ADXL Data if ’P’
        CALL    TX_DATA         ; transmit data to PC
        GOTO    MAIN

SETHIGH BSF     PORTB, TXD
        GOTO MAIN

SETLOW  BCF     PORTB, TXD
        GOTO MAIN

        END
```

Appendix B - Accelerometer C code

```
#include <PICCoder.h>

/* Converted from acc.asm (by Chris Djiallis based on gps.asm by Chris Djiallis
and MultiPIC.asm by Cliff Randell) */
/* Converted to acc.c by Tom Parker */

/*TITLE "acc.c - PIC interface for Accelerometer device to PC (filtered) comms"*
/

const int TXD = 1;
const int RXD1 = 0;
const int RXD2 = 1;
const int X_IN = 1;
const int Y_IN = 0;
int const MEMBAS= 0x008;
int const BAUDIV = 2;   /* Rtcc setting for 3.2us tick with 10MHz clock */
int const BAUD = 61;            /* 4800baud-208us 2400baud-416us 1200baud-833us
*/
                                /* 65 ticks of 3.2us = 208us */
                                                                /* reduced to 61
 on test (instruction delay) */

int BAUDSET;            /* Element time */
int BAUDHAF;            /* Half element time */
int BAUDSTR;    /* (1 + 1/2) element time to drop */
int const NUMBIT=8;                     /* Number of transmit serial data bits *
/

V_Node* SER_RX;
V_Node* SER_TX;
V_Node* BITCNT;

V_Node* T1_X;
V_Node* T1_Y;
V_Node* T2;
V_Node* COUNTER;

S_Node *BITIN(int pin)
{
        S_Node *ret = setVar(getTimer(),newConstant(BAUDSET));
        ret = makeBlock(ret,whileExpr(buildBoolean(VtoE(getTimer()),BOOL_EQ,newC
onstant(0)),NULL));

        /* Timer has zeroed, shift SER_RX, sample data and exit */

        ret = makeBlock(ret,rotateVarRight(SER_RX));    /* Shift data to make wa
y for next bit */
        ret = makeBlock(ret,ifExpr( buildBoolean(readPort(PORT_B,pin),BOOL_EQ,ne
wConstant(0)),
                clearVariableBit(SER_RX,7),
                setVariableBit(SER_RX,7)
                ));

        return ret;
}

        /******* */
        /*      TXLOW sets the transmit data RS232 line 1 low for a MARK */

S_Node* TXLOW()
{
        return unsetPort(PORT_B,TXD);
}

        /******* */
        /*      TXHI sets the transmit data RS232 line 1 high for a SPACE */

S_Node* TXHI()
{
        return setPort(PORT_B,TXD);
```

```
}

S_Node* WAITEM()
{
        return makeBlock(
                setVar(getTimer(),newConstant(BAUDSET)),
                whileExpr(buildBoolean(VtoE(getTimer()),BOOL_EQ,newConstant(0)),
NULL)
        );
}

        /******* */
        /*      XMIT sends the 8 bit (or less) byte in W reg to the transmit dat
a */
        /*      port as an async data byte with 1 start and 1 stop bit with the
least */
        /*      significant bit first. */

S_Node* XMITC();

S_Node *XMIT(E_Node* tostore)
{
        S_Node *ret = makeBlock(
                setVar(SER_TX,tostore),
                setVar(BITCNT,newConstant(NUMBIT+1))
        );
        ret = makeBlock(ret,TXLOW());
        return makeBlock(ret,whileExpr(buildBoolean(VtoE(BITCNT),BOOL_EQ,newCons
tant(0)),XMITC()));
}

S_Node* XMITC()
{
        S_Node *doifNZ = makeBlock(
                setVar(BITCNT,arithExpr(VtoE(BITCNT),ARITH_MINUS,newConstant(1))
),
                ifExpr(buildBoolean(VtoE(BITCNT),BOOL_EQ,newConstant(0)),
                        makeBlock(
                                ifExpr(testVariableBit(SER_TX,0),TXLOW(),TXHI())
,
                                rotateVarRight(SER_TX)
                        ),
                        TXHI()
                )
        );
        return makeBlock(doifNZ,WAITEM());
}


        /******* */
        /*      RXBYTE receives and dumps the start bit. Receives 8 bits into */
        /*      'SER_RX' with the most significant bit received first and then d
umps */
        /*      the stop bit. */

        /*      First, test for the LOW start bit at the beginning of the byte.
*/

S_Node *RXBYTE1()
{
        S_Node *ret = whileExpr(buildBoolean(readPort(PORT_B,RXD1),BOOL_EQ,newCo
nstant(0)),NULL); /* Test receive data, loop until data low*/
        ret = makeBlock(ret,setVar(getTimer(),newConstant(BAUDHAF))); /* Load ha
lf element time and preset the timer */
        ret = makeBlock(ret,whileExpr(buildBoolean(VtoE(getTimer()),BOOL_EQ,newC
onstant(0)),NULL)); /* Read the RTC divider and Loop until divider zeroes */
        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_B,RXD1),BOOL_EQ
,newConstant(0)),NULL)); /* Test receive data, loop until data low (true start b
it) */
```

```
        /*      Start edge seen (low level) - set timer for 1 elements which */
        /*      causes BITIN to go to first data bit. */

        ret = makeBlock(ret,setVar(getTimer(),newConstant(BAUDSET))); /* Load on
e element time and preset the timer */
        ret = makeBlock(ret,setVar(SER_RX,newConstant(1<<7))); /* Initialise ser
ial data register */
        ret = makeBlock(ret,BITIN(RXD1)); /* Get a data bit */
        ret = makeBlock(ret,whileExpr(noCarry(),BITIN(RXD1))); /* Loop until the
 'carry' bit appears */

        /*      Received character is in SERIAL, now dump the stop bit */

        return makeBlock(ret,WAITEM());
}

        /*------------- TX_DATA --------------- */
        /* */
        /*    >> transmit accel to PC */
        /* */
        /*------------- TX_DATA --------------- */

S_Node* TX_DATA()
{
        S_Node *ret =       XMIT(newConstant('$'));
        ret = makeBlock(ret,XMIT(newConstant('R')));
        ret = makeBlock(ret,XMIT(newConstant('A')));
        ret = makeBlock(ret,XMIT(newConstant('O')));
        ret = makeBlock(ret,XMIT(VtoE(T1_X)));
        ret = makeBlock(ret,XMIT(VtoE(T1_Y)));
        ret = makeBlock(ret,XMIT(VtoE(T2)));
        ret = makeBlock(ret,XMIT(newConstant(';')));
        return makeBlock(ret,XMIT(newConstant('\n')));
}

        /*------------- READACC --------------- */
        /* */
        /*    >> read acceleration */
        /* */
        /*------------- READACC --------------- */

S_Node *READACC()
{
        S_Node *ret = setVar(COUNTER,newConstant(0)); /* set counter to zero */
        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,X_IN),BOOL_EQ
,newConstant(0)),NULL)); /* wait for X_IN to be low (for synchronisation) */
        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,X_IN),BOOL_EQ
,newConstant(1)),NULL)); /* wait for first rising edge */

        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,X_IN),BOOL_EQ
,newConstant(1)),
                        set_execcycles(setVar(COUNTER,arithExpr(VtoE(COUNTER),AR
ITH_PLUS,newConstant(1))),4)
                )); /* wait for first falling edge */
        ret = makeBlock(ret,setVar(T1_X,VtoE(COUNTER))); /* T1_X = COUNTER */

        ret = makeBlock(ret,setVar(COUNTER,newConstant(0)));
        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,Y_IN),BOOL_EQ
,newConstant(0)),NULL)); /* wait for X_IN to be low (for synchronisation) */
        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,Y_IN),BOOL_EQ
,newConstant(1)),NULL)); /* wait for first rising edge */

        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,Y_IN),BOOL_EQ
,newConstant(1)),
                        set_execcycles(setVar(COUNTER,arithExpr(VtoE(COUNTER),AR
ITH_PLUS,newConstant(1))),4)
                )); /* wait for first falling edge */
        ret = makeBlock(ret,setVar(T1_Y,VtoE(COUNTER))); /* T1_Y = COUNTER */

        ret = makeBlock(ret,whileExpr(buildBoolean(readPort(PORT_A,Y_IN),BOOL_EQ
,newConstant(0)), /* wait for second rising edge */
                                set_execcycles(setVar(COUNTER,arithExpr(VtoE(COUNTER),AR
ITH_PLUS,newConstant(1))),4)
                ));
        ret = makeBlock(ret,setVar(T2,VtoE(COUNTER))); /* T2 = COUNTER */

        return ret;
}


/*---------------------------------------------------------------------------- */
/*      This program is intended to provide an interface between */
/*      an Accelerometer and a processor with a serial RS232 port. */
/* */
/*      Requests from the processor are in the form of $SA... are responded to *
/
/*      with data also in the form $RA*******. */
/* */
/*      The format of the serial data is 4800 baud, 1 start, 1 stop and 8 */
/*      data bits with no parity. */
/* */
/*      For use with DS275 RS232 Serial interface chip, */
/*      i.e uses normal polarity - not inverted. */
/* */
/* */
        /* Baud rate and software uart constant definitions */

int main()
{
                /*--------------------------------------------------------------
-------------- */
                /*                      Variable Assignment */
                /*--------------------------------------------------------------
-------------- */


        /*V_Node* FLAG = newVariable(NULL);*/   /* 8 general purpose flag bits h
eld here */
        S_Node *_master;

        SER_RX = newVariable(NULL);             /* Serial shift register for dat
a rx */
        SER_TX  = newVariable(NULL);    /* Serial shift register for data tx */
        BITCNT = newVariable(NULL);     /* Counter of bits received */

        COUNTER = newVariable(NULL);
        T1_X = newVariable(NULL);
        T1_Y = newVariable(NULL);
        T2 = newVariable(NULL);

        BAUDSET=0x100-BAUD;
        BAUDHAF=BAUDSET+BAUD/2;
        BAUDSTR=0x100-BAUD-BAUD/2;


        _master = TXHI();
        _master = makeBlock(_master,setVar(getTimer(),newConstant(BAUDIV)));
        _master = makeBlock(_master,setPort(PORT_B, TXD));
        _master = makeBlock(_master,RXBYTE1());

        _master = makeBlock(_master,
                ifExpr(buildBoolean(VtoE(SER_RX),BOOL_EQ,newConstant('$')),
                        makeBlock(
                                RXBYTE1(),
                                ifExpr(buildBoolean(VtoE(SER_RX),BOOL_EQ,newCons
tant('S')),
                                        makeBlock(
                                                RXBYTE1(),
                                                ifExpr(buildBoolean(VtoE(SER_RX)
,BOOL_EQ,newConstant('A')),
                                                                   makeBlock(
                                                                            RXBYTE1(),
                                                                            ifExpr(buildBool
ean(VtoE(SER_RX),BOOL_EQ,newConstant('O')),
                                                                                     makeBloc
k(

RXBYTE1(),

ifExpr(buildBoolean(VtoE(SER_RX),BOOL_EQ,newConstant('H')),

setPort(PORT_B,TXD),

ifExpr(buildBoolean(VtoE(SER_RX),BOOL_EQ,newConstant('L')),

        unsetPort(PORT_B,TXD),

        ifExpr(buildBoolean(VtoE(SER_RX),BOOL_EQ,newConstant('P')),

                makeBlock(READACC(),TX_DATA()),

                NULL)

)

)
                                                                        ),
                                                                        NULL)
                                                                ),
                                                        NULL)
                                                ),
                                        NULL)
                                ),
                        NULL)
                );


        outputcode(_master);
        return 0;
}

/*--------------------------------------------------------------------------- */
```