ECE 476: BASIC Interpreter

David Hodgdon

3rd May 2004

Abstract

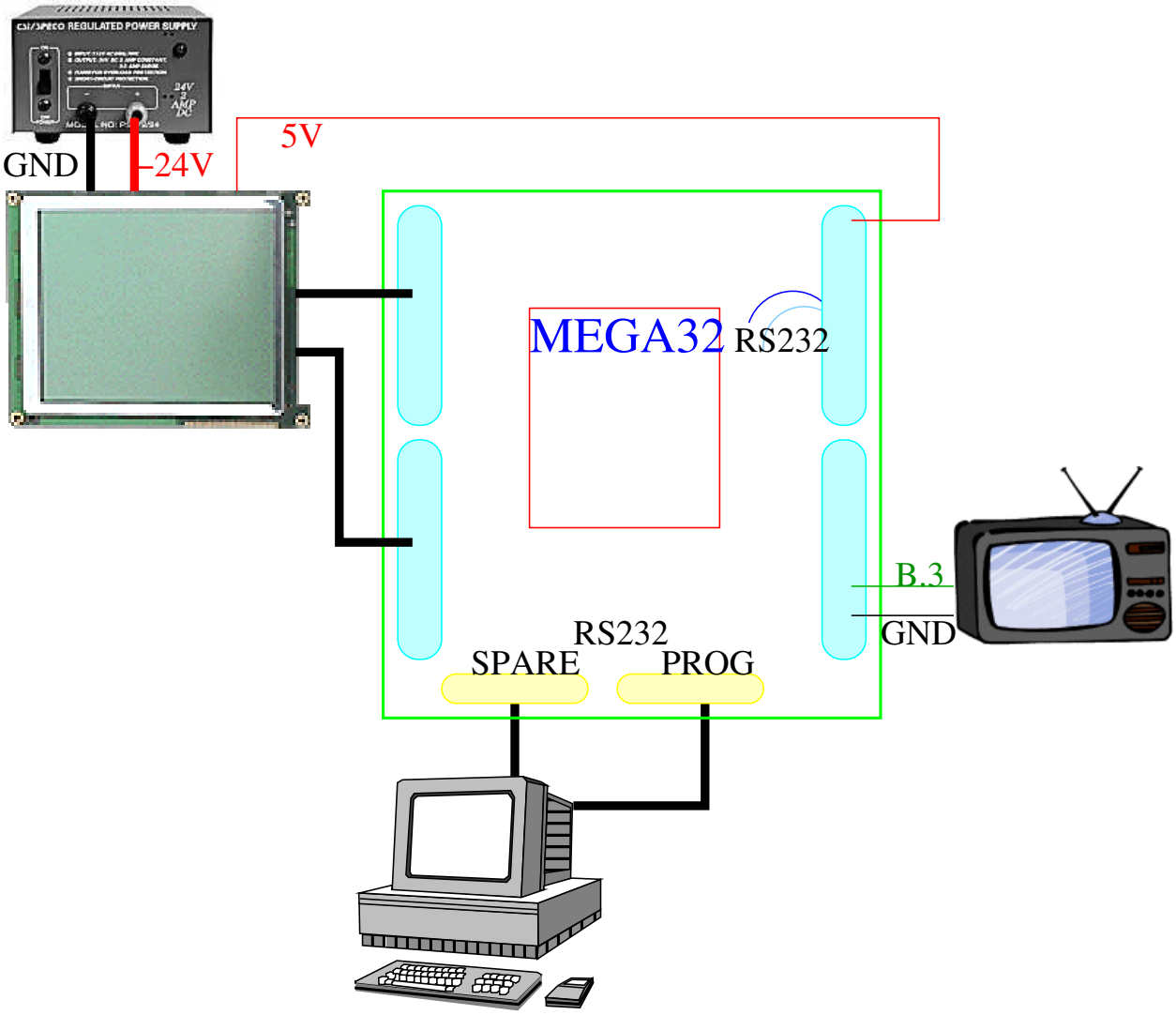
The BASIC Interpreter final project implements a very simple BASIC interpreter, written in embedded C for the Atmel Mega32 microcontroller. A Seiko G321D 320X200 LCD with a built-in SED1330F controller will act as a textual display. Variable width square waves are feed through the sound port of a television to produce "music." The lab PC acts as a filesystem, feeding the microcontroller the BASIC file one character at a time as requested, and as a user input device (keyboard).

Contents

List of Figures

List of Tables

Figure 1: High Level Design



1 Introduction

"The BASIC Interpreter final project implements a very simple BASIC interpreter, written in embedded C for the Atmel Mega32 microcontroller."

1.1 What I Did and Why

See Figure 1 for a high level concept of the project. The LCD was chosen as the textual display device because the SED1330F controller provides high-level commands that aid in a simple design. The LCD is better than TV video generation because

1. the characters don't have to be stored in the flash memory (they are stored on the LCD controller),
2. the pixels don't have to be stored in SRAM, they are stored on the LCD controller,
3. there are no timing constraints such as those involved in TV video generation,

4. and all the timers are free.

The computer acts as a file system to store the BASIC programs. This is better than storing the program on the microcontroller because

1. SRAM or flash isn't used,
2. it is easy to write and change the BASIC program on the PC because of the high availability of good text editors,
3. PCs are installed at each lab bench.

The hardware timers were used to produce square waves because

1. square waves sound *really* cool (authentic),
2. the generation of square waves requires blocking until musical note is finished,
3. square wave generation is simple, easy, and flexible.

The television's speakers were used for no special reason except that there is a television on every lab bench.

1.2 BASIC History

Early Years In 1964 the home computer was only a dream, and IBM still thought that nobody would ever want a computer in their home. Computers were primarily owned by the government, universities, and large businesses. Time sharing allowed a single computer to be shared among hundreds of users (each current user gets a slice of computing time). This advancement created a new class of "novice" computer users. John Kemeny and Thomas Kurtz (at Dartmouth) developed BASIC[1] as a simpler programming language for these new computer users. Because they choose to keep BASIC in public domain, the language was able to spread.[1]

Home Computing Revolution In the late 1970's, home computing became a reality, and therefore, the language developed for "novice" users suddenly became massively appealing. A little two person company called "Microsoft" started by developing a version of BASIC for the Altair. Because of Microsoft's success in this field, they were the choice of Apple and IBM for BASIC interpreters, which were build into the ROMs of Apple IIs and IBM PCs.[1]

Future Advancements The BASIC language has developed significantly over the years. There are now many incompatible versions available on the original person computers, graphing calculators, and modern PCs. The proposed implementation will follow the simpler versions (such as Applesoft BASIC[2]).

Present Day The BASIC language still lives in products such as Microsoft's Visual BASIC.[2] BASIC actually turned 40 years old this year, yielding a busy thread on slashdot.[3, 4]

2 Software

Figure 2 gives an idealistic overview of the software. Communication between blocks only occurs by higher blocks making calls on lower blocks — that is, the parse block calls on scan, input, var, and lcd, but the lcd block cannot call on anything, rather it interacts directly with hardware.

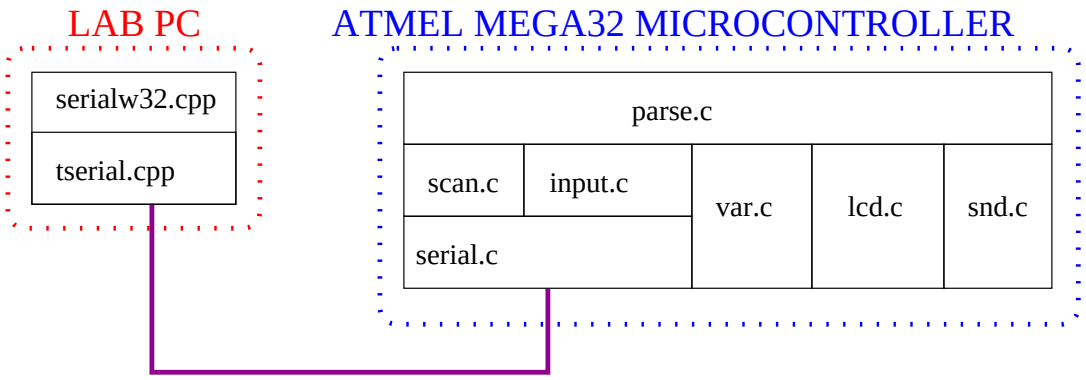
For debugging purposes and to reduce the time spend in lab, the BASIC interpreter runs both on the microcontroller (attached to the PC) and separately on the PC. Compiling for the microcontroller requires the `ATMEL` to be defined.

For clarity, I kept all functions static if possible and named functions with a prefix indicating the module in which the function occurred (e.g., `ParseExecuteStatement` in the parse module).

[1] **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

[2] Applesoft BASIC was distributed on the ROM of Apple II computers in the 1970's and 1980's.

Figure 2: Software Overview



2.1 Scanner

The scanner consists of `scan.h` and `scan.c`. The purpose of a scanner is to find symbols mostly independent of the grammar of the language. Scan functions access the BASIC source through the `GETC` macro, which returns one character of the source at a time. From these single characters, the scanner builds a symbol in function `ScanGetSymbol`. Symbols are defined by the following enumeration:

```
enum ScanSymbol{ScanIdent, ScanNum, ScanStr, ScanEOF, ScanEOL,
  ScanPlus, ScanMinus, ScanTimes, ScanDiv, ScanExpon, ScanNewStatement,
  ScanIntegerVar, ScanRealVar, ScanStringVar,ScanComma, ScanLeftParen,
  ScanRightParen,ScanSemicolon, ScanEqual, ScanLessThanEqual,
  ScanGreaterThanEqual, ScanGreaterThan, ScanLessThan, ScanNotEqual,
  ScanAND, ScanOR, ScanNOT,  ScanFOR, ScanGOSUB, ScanNEXT, ScanSTEP,
  ScanGOTO, ScanIF, ScanTHEN, ScanTO, ScanDATA, ScanREAD, ScanLET,
  ScanPRINT, ScanREM, ScanINPUT, ScanINT, ScanDIM, ScanLEN, ScanRAND, ScanREAL};
```

Given these symbols, "`1020 IF A% >= B% THEN GOTO 1000`" would be interpreted as the following symbols `ScanNum`, `ScanIF`, `ScanIntegerVar`, `ScanGreaterThanEqual`, `ScanIntegerVar`, `ScanTHEN`, `ScanGOTO`, and `ScanNum`. However, "`>< A DIM - *`", would be interpreted as `ScanNotEqual`, `ScanRealVar`, `ScanDIM`, `ScanMinus`, and `ScanTimes`. The scanner is very happy to scan such code — it is the parser's job to decide if the symbols make any sense.

2.2 Variables and Variable Allocator

The variable allocator consists of `var.c` and `var.h`. The purpose of the variable allocator is to allocate and free memory for variables.

2.2.1 Variables

All variables types are stored in a single struct. Embedded C is not the most elegant language for this sort of programming. Optimally, I would prefer an object orient language with inheritance and possibly interfaces so that I didn't need to create a single huge struct that fits all data types. However, given the C programming language, I think this type of design is a good one for both simplicity and readability.

Types BASIC has six types. The type is specified by adding a "type" character after the identifier. The type character "`%`" is for integers, "`$`" for strings, and nothing for reals. Array indices are placed in parentheses after the type character (there are up to three indices). For example `STR$` is a string, `CNT%` is an integer, `TEMP` is a real (no "type" character), `STRS$(I%)` is element `I%` of a one dimensional array of strings,

`CNTRS%(I%,J%)` is an element of a two dimensional array of integers, and finally `A(I%,J%,K%)` is an element of a three dimensional array of reals.

```
enum VarType{Integer=0, Real=1, String=2, IntegerArray=4, RealArray=5, StringArray=6};
```

The single all-encompassing struct is shown below.

```
struct VarDesc{
  enum VarType type;
  char ident[3];          // variable name
  char i1, i2, i3;        // array index 1
  char* str;              // string if string type
  unsigned char len;      // length if string type
  int intVal;             // integer value
  float realVal;          // real value
  struct VarDesc* next;   // pointer to next
  struct VarDesc* prev;   // pointer to previous
  char isTemp;            // 1=temp, 0=variable
};
```

Variable Names Notice that while variables can have long names, internally their name is only represented by the first two letters of their name. In this sense the variables names `COUNT$` and `COM$` refer to the same variable. In practice, one should never "utilize" this fact, as it is tremendously confusing; however, one should keep it in mind in order to avoid aliasing variables. In an interesting twist, one can have variables with the same two character identifiers as long as they are of different type. For example, `A`, `A$`, `A%`, `A(I%)`, `A$(I%)`, and `A%(I%)` are all different variables. I implement this by having separate linked lists to store each type's variables. This way, given `A%`, only the integer linked list is searched.

Arrays When implementing arrays their were two options that I considered

1. have a single `VarDesc` for an array and have it point to some block of memory that held the array
2. give each element of the array its own `VarDesc`.

I choose option 2 after much debate (with myself). All my parser functions work with the a `VarDesc`, so I hoped to avoid writing lots of special case code for arrays. I hoped to avoid wasting space because if arrays are allocated in a block, some of the indices may not be used — also there is the issue of zero vs one base indexing (and in my solution no additional space is waisted in either case). It could be argued that this implementation uses too much memory; however, it is far more consistent with my other engineering decisions.

Notice that there are three array index variables. I find it interesting that normal BASIC implementations support 3D arrays, where C / Java only support pseudo-multidimensional arrays through "arrays of arrays."

2.2.2 Variable Allocator

Variable Struct Allocation and Deallocation As previously mentioned each variable type has its own linked list to store variable structs — these lists have statically defined head / tail pointers. There is also a table (static array) of free variable structs, which are placed in a "free list" (another linked list) on initialization. New variables are "allocated" by taking a variable off the "free list" and placing it on the appropriate variable linked list (or if it is a temp variable, not placing it anywhere). This new variable "allocation" is performed by the following function `Var VarNewVar(enum VarType type, char * ident, char i1, char i2, char i3, char isTemp)`. On deallocation the variable is placed back in the free list; however, only temporary variables are deallocated because there is no scope in BASIC that allows a variables scope to end.

String Allocation When I used the CodeVision AVR compiler at home, I discovered that CodeVision AVR supported `malloc/free`; however an older version existed at the lab. This is important to consider when looking at my implementation. Even when a string variable is allocated with the `VarNewVar` call, the actual string remains unallocated. This allows more efficient use of memory. When the string is actually needed, the parser calls `void VarNewStr(Var v, unsigned char size)`. The string allocated using `xmalloc` and freed using `xfree`. An array of structs are maintained to describe the current string allocations (an address base / size pair), and a large character array from which they allocate. Writing good malloc emulation was not part of my project! My implementation is very simple, but internal fragmentation is very possible, which could become an issue given a small memory space.

2.3 Serial Communication

Serial communication provides two services in my project: user input and program code transferal.

Commands The PC and microcontroller communicate by sending commands defined by an enum (below) and characters. The enum values are defined specially so that they cannot interferer with ASCII characters.

```
enum SerialCmd{SerialOpenFile=-20, SerialNextChar, SerialResetPointer,
  SerialNextCharDATA,SerialResetPointerDATA,SerialFailureOpeningFile,
  SerialSuccessOpeningFile,SerialRequestInput};
```

2.3.1 User Input

When the parser reaches an INPUT command, `InputGetString()` sends a `SerialRequestInput` command to the PC. Both the PC and the microcontroller wait until the user enters a string (terminated by a return).

2.3.2 Program Code

Program code is stored on the PC. The microcontroller requests code as needed.

Opening a file First the microcontroller sends a `SerialOpenFile` command, which is followed by a length and each character of the filename.

Requesting characters The scanner requests characters one at a time using the `SerialNextChar` command. This implementation is probably responsible for majority of the slowness of project. A faster implementation would involve a buffer (with fewer `SerialNextChar` requests) or possibly a code cache (which would be very useful inside loops); however, speed was never my primary objective — my implementation is simpler and more debuggable.

Branches and New Lines of Code Originally I had planned to have a very simple interface on the PC. I would only allow the microcontroller to reset the file pointer on the input file. It would then have to read each character up until the needed line of code. However, this made the project very, very slow! I employed two methods to solve this problem.

1. When the file is opened, the contents are read into a large array. Since there is plenty of memory on the lab PC, this is not an issue. This reduces file access latency, which is good, but probably was not the majority of the slowdown.
2. Instead of reseting the file pointer (no longer a file pointer), the microcontroller sends the `SerialResetPointer` command along with a two byte line number. The PC can then, give the microcontroller the exact line it is looking for (should that line exist).

These changes improved the speed by a factor of about two to four, depending on the size of the file of course.

READ / DATA The READ / DATA keywords require additional state, parallel to the previously mentioned program code retrieval. Therefore, there are equivalent commands: `SerialNextCharDATA` and `SerialResetPointerDATA`.

2.3.3 Communication Code on Microcontroller

The code consists of `serial.h`, `serial.c`, `input.h`, and `input.c`. The code is a very simple blocking implementation using `putchar(char c)` and `getchar()`.

2.3.4 Communication Code on Lab PC

The code consists of `serial.h` and `serialw32.cpp` (my own code) and `tserial.hpp` and `tserial.cpp`, which implement a threaded serial communication interface. The code on the PC is compiled using Visual Studio .NET 2003 (in PH318). The interface on the PC is very simple. There is an infinite while loop: first a character is received using the blocking `getChar()` function, then a switch statement switches on that character, directing control flow to the appropriate command handling code.

2.4 Sound

The code consists of `snd.h` and `snd.c`. Sound is produced by running timer zero connected directly two `PORTB.3`. I also run timer one to determine how long a given note should be played (basically, to keep track of time). The parser calls `SndPlayNote(int note, char time)`, where any `note` less than 0 or greater than the maximum defined note is a rest. Notes between zero and eleven are defined by their timer zero output compare register values:

```
flash char notes[12]={239,   225,   213,   201,   190,
        179,   169,   159,   150,   142,   134,   127};
```

which I generated in matlab from the frequencies of the notes using

```
notes=[261.64 277.20 293.68 311.12 329.64 349.24 370 392 415.32 440 466.16 493.92]
round(16e6/256./notes').
```

Notice, that note zero is a C. The second octave is produced by right shifting the output compare value by one bit (divide by two). Eventually, the output compare value would get small, so I used different clock prescalars to solve this. The time is saved into a global variable called noteDuration. At each clock tick of timer one, this value is decremented (except it is a saturating down counter, so it doesn't go past zero). When a note enters `SndPlayNote`, it must first wait for the current note's time to expire. Because a note can play immediately after the last note has ended, slurred music is produced. If this is not the desired effect, 1/16 note rests can be added between notes (as seen on Appendix C.7 in the Mary Had a Little Lamb example code). An interesting consequence of this design decision is that one can start a note in one statement and end it in another — essentially the programmer has another means of producing a variable length note.

2.5 Parser

The code consists of `parse.h` and `parse.c`. Since this is an interpreter, the parser is really the "main" module of the project (it contains the `main()`, etc).

2.5.1 Expression Parser

The expression parser is a stack off functions with each function handling a different level of precedence in the BASIC expression precedence hierarchy:

- `()` [parentheses] are handled by `ParseFactor`,

- `^` [exponentiation] by `ParseExponTerm`,
- `*` [multiplication], / [division] by `ParseTerm`,
- `+` [addition], `-` [subtraction] by `ParseSimpleExpression` (which also handles unary plus / minus),
- `=` [equal], <> or >< [not equal], < [less than], > [greater than], <= or =< [less than or equal], >= or => [greater than or equal], by `ParseEqualityExpr`,
- `NOT` [logical complement] by `ParseNOTExpr`,
- `AND` [logical AND] by `ParseANDExpr`,
- and finally, `OR` [logical OR] by `ParseExpression`.[5]

Each expression starts by calling ParseExpression, then the call is passed up the stack until the appropriate operator is handled, then the result is passed back down the stack. This method is simple and easily modifible; however, on the microcontrollers, it is relatively slow. Also, I made sure that I reduced the number of local variables, in order to keep the stack small. Each operation is "handled" (in the appropriate function in the stack) by calling either `ParseDoOpUnary(Var x, enum ScanSymbol op)` or `ParseDoOp(Var x, Var y, enum ScanSymbol op)`.

Once I wrote the expression parser (in its general form), I could call on it whenever I wanted to evaluate an expression.

2.5.2 Executing a Statement

The function `ParseExecuteStatement()` executes a single statement. A statement either starts after a line number at the beginning of a line, or starts after a colon. The general structure of this function is that it switches on the first symbol of a statement. Each case statement defines the behavior of a different keyword (or plain assignment of a variable). Here are some of the keywords defined in `ParseExecuteStatement()`:

- `REM`, `DATA`, and `DIM` are interpreted as NOPs,
- `SND(<expression>, <expression>)`,
- `PRINT <expression>[;<expression>]*`,
- `INPUT <variable>[,<variable>]*`,
- `READ <variable>[,<variable>]*`,
- `GOTO <expression>`,
- `IF <expression> THEN <line of code>`,
- `FOR <real variable> = <expression> TO <expression> [STEP <expression>]`,
- `NEXT [variable]`.

Not all keywords are defined in `ParseExecuteStatement`. Keywords that produce a value, like non-void functions in C, are defined in `ParseFactor`. These keywords include: the integer cast function `INT`, the random number function `RAND`, and the string length function `LEN`, all of which also take expressions as input.

`ParseExecuteLine` (which executes a single line) usually calls `ParseExecuteStatement` only once, but makes more calls if there are colons (the statement divider character).

2.5.3 Branching

Lines are executed in order until either a `GOTO` or a `NEXT` statement is encountered; then the interpreter needs to branch to either the `GOTO` target or the line after the matching `FOR` statement. Originally, the next "PC" was recorded and searched for at the end of each line. While I knew this was not optimal, it was a very simple solution that covered all cases. It turned out that this implementation was *extremely* slow. The interpreter had to receive on average half the file at the end of each line (and receive the file one character at a time). In order to make the interpreter usable, I had to decrease the branch latency; therefore, I moved some complexity off of the microcontroller onto the PC. Now, instead of requesting that the PC reset the file pointer and start reading from the beginning, I request a specific line, which the PC searches for and returns. In order to make this process simpler and reduce file IO, I have the PC interface program read the BASIC code into memory — then it only needs to search an array from the PC value.

2.6 Software Conclusions

2.6.1 Difficult Parts

One of the most difficult parts of the software to write is the `xmalloc` / `xfree` code. I had naively thought that this would be easy and designed a really simple (incorrect) set of functions. I later fixed it so that it worked; however, I was still only able to allocate strings at the end of character memory and not between strings where another string had previously been allocated. So, I was running out of character memory all the time. Therefore, I changed the code a second time, making it far more robust. Of course, if malloc / free had been supported in the compiler (they were included in my demo version of CodeVision AVR), I wouldn't have had to implement them again.

2.6.2 Things I couldn't get to work

Keyboard I had originally intended to connect a keyboard directly to the STK/Mega32 for input. The hardware setup worked fine, I was able to read the signal on the oscilloscope; however, I would lose synchronization with the signal and then the characters would be all garbled. I decided that keyboard connectivity was of little importance because I was already using the UART for program code input and I could just add input functionality to that interface.

STK300 / ATMEL Mega103L I had originally intended to use a Mega103L because it had twice the SRAM; however, I immediately had difficulty getting the LCD to work. In all fairness, the data sheet does state that 5V input is required; however, I thought that the Mega103L's 3.3V would probably work... it didn't. I considered adding double CMOS inverters between the outputs of the Mega103L and the inputs of the LCD, but I decided to switch to the Mega32 because I wanted to get the project moving.

2.6.3 Further Work

My project could be extended by adding functionality to the BASIC interpreter.

Graphics It seems a waist to have this awesome LCD and not have graphics support. If some simple graphics functions were implemented, simple games could work.

Memory Someone could optimize my variable struct using unions to map the string length, integer intVal, and real number realVal to the same memory. These kind of fixes could increase the available variable space somewhat. A simpler solution would be to move to a microcontroller with more memory like the 128 or 103. Given more memory, games like star trek, eliza, etc. could be played.

Keyboard Someone could get the keyboard interface working. I know other groups have don this, so it's definitely possible.

Figure 3: LCD solder Points

3 Hardware

3.1 LCD

To connect the LCD to the whiteboard, I soldered a pin header (a row of pins) onto the the lcd connections (See Figure 3).

The LCD requires a 24 volt power supply. The highest voltage power supplies in the lab are 12 volt power supplies; however, these power supplies supply both +12V and -12V power (See Figure 4). Therefore, by connecting the +12V to the MCU ground and the -12V to the LCD, I was able to achieve -24V power. All I had to do was move a few solder points on the handy little power supply switch (See Figure 5). The LCD also requires a 5V supply, for which the Mega32 will suffice. I connect a short cable to the PORTD Vcc and GND connection to the white board. I could have used one of the signals that was already coming over to the board on the control or data port, but I wanted to be able to quickly power down my whiteboard in case of emergence (hence also the switch on the -24V supply.

Connecting the control port and data port to the LCD was straight forwardly documented in the data-sheet.[6] The data-sheet even provides an example initialization sequence to get the LCD working. Because previous groups had already implemented a simple LCD interface, I decided not to replicate work and used their code (See Section 5.3 for more information).

Figure 4: Power Supply Label

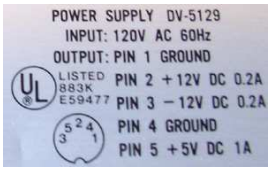
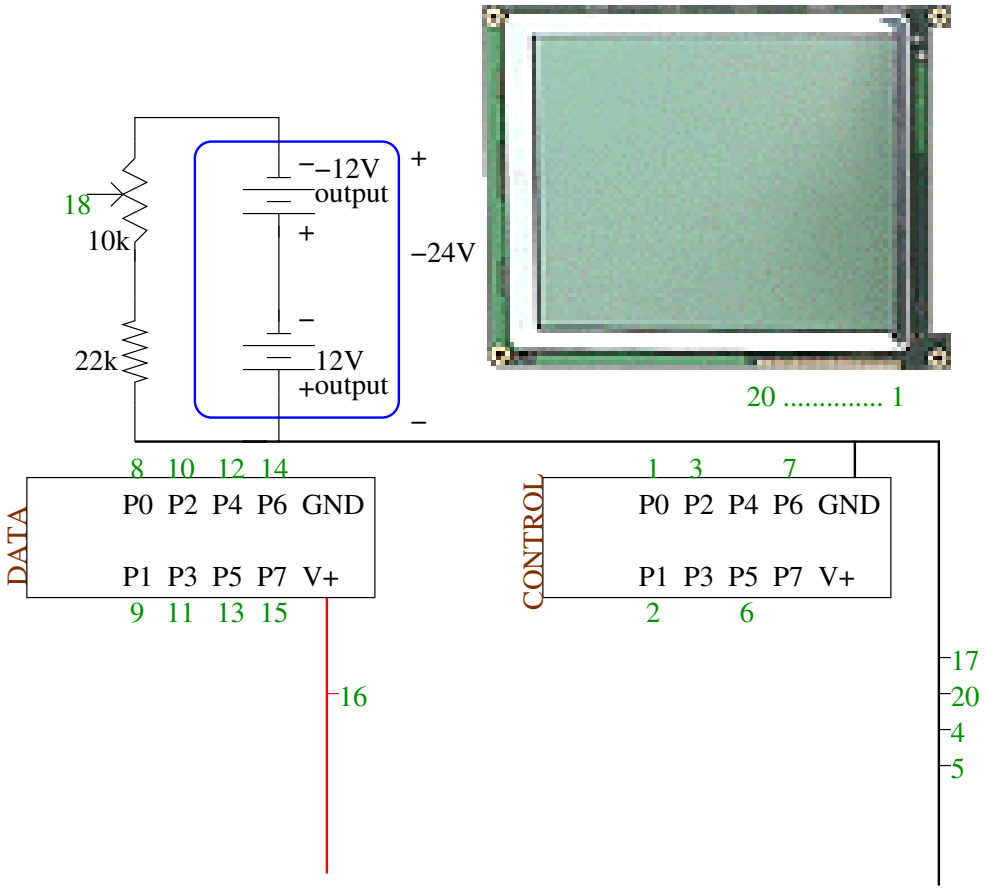


Figure 5: Power Supply Switch

Figure 6: LCD Setup



3.2 Sound

To setup the sound I follow Bruce Land's video generation code instructions.[7]

3.3 RS232

The RS232 standard was originally developed by the Electronic Industries Association in the 1960s as a standard for data communication (communication between a terminal and a mainframe often over a modem). The standard specifies signal timings, voltages, a protocol, and the mechanical connectors. I used the RS232 to program and communicate with the microcontroller.

The UART connection is the same as in lab3.[8] There must be a cable connecting RXD and TXD to PD0 and PD1 respectively (See Figure 7). Then a serial cable connects COM1 on the PC to RS232 Spare port on the microcontroller (See Figure 8).

Figure 7: UART Connection

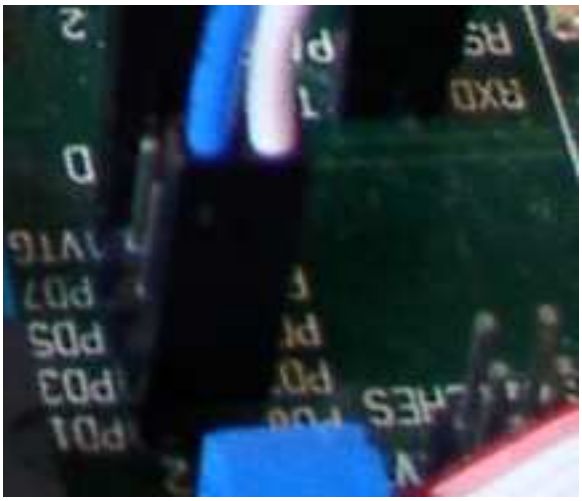


Figure 8: RS232 Cables Programming and Spare

3.4 Hardware Conclusions

3.4.1 I am an Electrical Engineer

The last time I soldered was freshman year in high school. Luckily, it all came back to me. My solder points look very healthy.

3.4.2 Further Work

I think the sound produced with the square wave sounds really authentic; however, one improvement would be to produce a sine wave. To make music flow though, I think another microcontroller would have to produce the sine wave.

4 Results of the design

4.1 Speed

The interpreter is slow. The major issue is definitely not the computation but the latency involved in receiving the code from the PC. This could be drastically improved by buffering multiple characters. If this were combined with switching to interrupt based receive, which could be done "in the background." Latency could be dramatically reduced. Another option would be to have keep a code cache of part or all of the BASIC program. Keeping part of the program would greatly increase the speed inside loops.

After improving the transfer speed, data structures and function calls are probably the biggest speed problem. The expression parser involves *many* function calls each allocating stack space for their parameters

and local variables. Reducing the number of functions could provide a small speed boost after the massive speed boost involved in improving transfer rates.

4.2 Usability

Backspace I drastically increased the usability by adding *backspace* functionality to the input module. Before if you miss typed a file name, you would have to restart the MCU.

Exiting with Errors The following will result in exiting with an error message:

1. errors in coding the BASIC programs,
2. using too many variables (including intermediate values),
3. using too many strings (including intermediate values),
4. using too many characters in strings (including intermediate values and internal fragmentation space),
5. changing variable parameters in `var.c` that result in too much static memory usage — this will cause strange errors in the expression parser (probably due to the stack running into variable space),
6. entering an invalid string as input such as a blank string (just hitting return),
7. trying to utilize keywords not supported by my implementation,
8. trying to utilize operators not defined on variable types (such as string-string multiplication),
9. mismatched variable types for binary operators (such as real-string division),
10. mismatched type-requested input/read types (requesting an integer for input/read and receiving/dataing a string).

In the case of the microcontroller exiting means going into an infinite loop, where on the PC, exiting means `exit(1)`. I think exiting with an error is very appropriate for any problem in the BASIC code, including type mismatches. I also think it is unavoidable to exit when one has run out of memory. So, in general, I think the extensive use of exiting with errors is a good thing.

Flexibility The project is very flexible; except for the limited memory issue, any BASIC program can be run on the microcontroller without reprogramming. Perhaps it would be simpler if the PC program didn't have to be restarted for each new program execution.

5 Conclusions:

5.1 Completion of Goals

I wanted to connect the keyboard directly to the microcontroller, but in hindsight, I think using the PC as an intermediate is better. I am, however, disappointed that my interpreter cannot play classic text-based games like eliza and star trek.

5.2 Next Time

If I were doing this all over again, I might use my dynamic memory allocation system to allocate both strings and variables — this way, modifying the character / variable memory parameters wouldn't be necessary.

5.3 Intellectual Property

LCD Code The Seiko G321D LCD has been a popular LCD in ECE476. I have found three groups that have utilized it since 2001; however, the oldest reports suggest previous work with this controller. I drew my lcd code from a combination of the three online projects: The Russian Block Game (Tetris)[9], Graphing Calculator[10], and Cornell Monopoly[11]. The major modification that I introduced was the terminal wrapping behavior.

Parser / Scanner The parser / scanner I used in my BASIC project was derived from a parser / scanner used in my FALL2003 ECE495: Optimizing Compilers project. Significant portions of the code were rewritten. ECE495 was concerned with CSubset, a simple subset of C, so the grammar differs greatly. The codes are mostly just similar in structure, as the grammars and principles of the language differ significantly. So, the code is a combination of old code that I wrote and code provided by the instructor, but mostly new code. The course number of ECE495 has since changed to ECE473.[12]

BASIC Language The BASIC language is public domain.

Windows Threaded Serial Interface Since the focus of ECE476 is not on programming for a PC, I built my PC interface on top of freely available code.[13] I also consulted Microsoft's documentation on serial communication.[14]

Sound Code I started with Professor Bruce Land's Video Generation code. I added support for many octaves. [8]

Mary Had A Little Lamb I don't believe this tune is copyrighted. I did, however, transcribe the music from a web site.[15]

5.4 Ethical considerations

The BASIC Interpreter project adheres fully to the IEEE Code of Ethics.[16]

- My interests in this course are in complete and utter harmony with each other (2): I want to make a cool project, I want to pass this class, and I want to graduate from Cornell.
- I have always been honest in my proposal, progress reports, final report, and in the lab (3). I started maintaining my web site very early to keep my proposal and up-to-date progress report in order to keep accurate information available to the course staff.
- I have rejected bribery in all its forms (4). While money is certainly not the only form of bribery, I have only been engaged in two money transfers in the scope of this course.

  1. I bought two G321D LCDs from David Li, a TA for the course (though his being a TA for the course was irrelevant to my buying them, I found out that he wanted to sell them from a friend).
  2. I sold one of the two LCDs to Bryan Galusha, another ECE476 student (at the same price that I had bought them from David Li, to be fair). However, I made sure that I did not collect the money from him until he was happy with the LCD and almost done with his project.

  Also, when I helped another group with their LCD, I did so out of the "kindness of my heart" and not expecting something in return.
- In my project I have made the highest effort to treat all people fairly regardless of race, religion, gender, disability, age, or national origin (8). A few issues here are
  - My project uses ASCII for text which is inherently focused on languages with latin character sets.

  - My project requires reasonably good vision to read the LCD because there is no backlight. This could be fixed by adding better lighting and including a magnifying glass.
  - The deaf will not be able to "enjoy" the square wave music. The music system can be easily modified to print out the notes instead of playing them.
  - In my "Knock, Knock Jokes Demo Code" I attempted to find jokes that were not offensive to any particular group, but were funny or cute.

- I have avoided injuring others, their property, reputation, or employment by false or malicious action (9). Never have my actions been false, and furthermore I have always considered the impact my work will have on others. For example, when testing my "music" I alway ensured that the sound was kept at a reasonable volume to avoid driving other students into "square wave insanity."

References

[1] Wikipedia. "Basic programming language," [online]. February 2004. Available from World Wide Web: `http://en.wikipedia.org/wiki/BASIC_programming_language`.

[2] Microsoft. "Microsoft visual basic .net 2003," [online]. 2003 [cited May 2, 2004]. Available from World Wide Web: `http://www.microsoft.com/PRODUCTS/info/product.aspx?view=22&pcid=3897bf96-7aa3-41e9-88d0-`

[3] CmdrTaco. "Basic computer language turns 40," [online]. April 2004 [cited May 2, 2004]. Available from World Wide Web: `http://slashdot.org/article.pl?sid=04/04/29/1932227&mode=thread&tid=126&tid=156`.

[4] J. M. Hirsch. "Basic computer language turns 40," [online]. April 2004 [cited May 2, 2004]. Available from World Wide Web: `http://apnews.excite.com/article/20040429/D82885100.html`.

[5] N. Mates. "Applesoft basic frequently asked questions," [online]. August 1997. Available from World Wide Web: `http://www.apple2.org/faq/FAQ.applesoft.html`.

[6] S. Instruments. "Model g321dx1r1ac," [online]. 2001. Available from World Wide Web: `http://www.seiko-usa-ecd.com/lcd/products/graphic_mods/g321dx1r1ac.html`.

[7] B. Land. "Electrical engineering 476 video generation with avr microcontrollers," [online]. May 2003 [cited May 2, 2004]. Available from World Wide Web: `http://instruct1.cit.cornell.edu/courses/ee476/video/`.

[8] B. Land. "Ee 476: Laboratory 3 security system," [online]. January 2004 [cited May 2, 2004]. Available from World Wide Web: `http://instruct1.cit.cornell.edu/courses/ee476/labs/s2004/lab3.html`.

[9] D. Li and C. Fifadra. "The russian bloc game," [online]. 2002 [cited April 18, 2004]. Available from World Wide Web: `http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2002/dkl22/`. The Russian Bloc Game = Tetris.

[10] Y. Lin and J. Lee. "Ece476 final project: Ll-01 graphing calculator," [online]. 2001 [cited April 18, 2004]. Available from World Wide Web: `http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2001/yl131/`.

[11] K. F. Lam and J.-M. Qian. "Cornell monopoly," [online]. 2002 [cited April 18, 2004]. Available from World Wide Web: `http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2002/jq13/webpage/476monopoly.html`.

[12] M. Burtscher. "Optimizing compilers," [online]. April 2004 [cited May 2, 2004]. Available from World Wide Web: `http://www.csl.cornell.edu/courses/ece473/`. Called ECE495 when I took it.

[13] T. Schneider. "Serial communication for win32," [online]. April 2001 [cited May 2, 2004]. Available from World Wide Web: `http://www.tetraedre.com/advanced/serial/`.

[14] A. Denver. "Serial communication in win32," [online]. December 1995 [cited May 2, 2004]. Available from World Wide Web: `http://www.tetraedre.com/advanced/serial/`.

[15] EnchantedLearning.com. "Mary had a little lamb," [online]. 2003 [cited May 2, 2004]. Available from World Wide Web: `http://www.enchantedlearning.com/music/sheetmusic/maryhadalittlelamb.shtml`.

[16] IEEE. "Code of ethics," [online]. August 1990. Available from World Wide Web: `http://www.ieee.org/about/whatis/code.html`.

A Parts and Costs

Table 1: Parts and Costs

Item	Cost	Source
Seiko LCD	$10	David Li
MEGA32	$8	Lab
TV Speakers	$0	Lab
24V Power Supply	$0	Lab
White board	$5	Lab
Total	$23.00	

B Specific Tasks in the Project Carried Out by Each Team Member.

I worked alone. I did all the work. I endured all the pain and experienced all the joy myself.

C BASIC Test Cases

Because my project alone does nothing, I needed test cases that were both fun and covered a wide range of the features of my microprocessor. While these are not part of my project *per se*, they are quite necessary and took some time to design.

C.1 Guess a Number

The BASIC program in `guess.bas` is a simple number guessing game. The user enters a range of possible random integers and number of guesses, then guesses. After each guess, the program prints out too high / low and reminds the user of the number of guesses remaining. If the user guesses right, the user is given a complement. This program is a test case for long output size, multiple value inputs, DATA/READ statements, random numbers, conditional statements, and GOTO statements. See Figure 9.

```
1000 PRINT "Please enter a number range";" in the form: A,B"
1010 INPUT A%,B%
1020 IF A% >= B% THEN GOTO 1000
1021 PRINT "Enter the number of guesses you want to have:"
1022 INPUT I%
1023 IF I% <= 0 THEN GOTO 1021
1030 II% = I%
1031 NUM% = INT(RAND(1)*(B%-A%)+A%)
1040 INPUT G%
1045 II% = II% - 1
1050 IF G% = NUM% THEN READ STR$: PRINT STR$ : PRINT "I've picked another number for you" : GOTO 10
1060 IF G% < NUM%  THEN PRINT "Higher";"   (";II%;" left)"
1070 IF G% > NUM%  THEN PRINT "Lower";"    (";II%;" left)"
1075 REM PRINT II%;" Guesses Left"
1080 IF II% > 0 GOTO 1040
1090 PRINT "You've lost!"
1095 GOTO 1000
2001 DATA "You're the man!", "You Rule!", "A truly smart Cornell student", "You guessed right", "Yo
```

Figure 9: Guess Demo Code

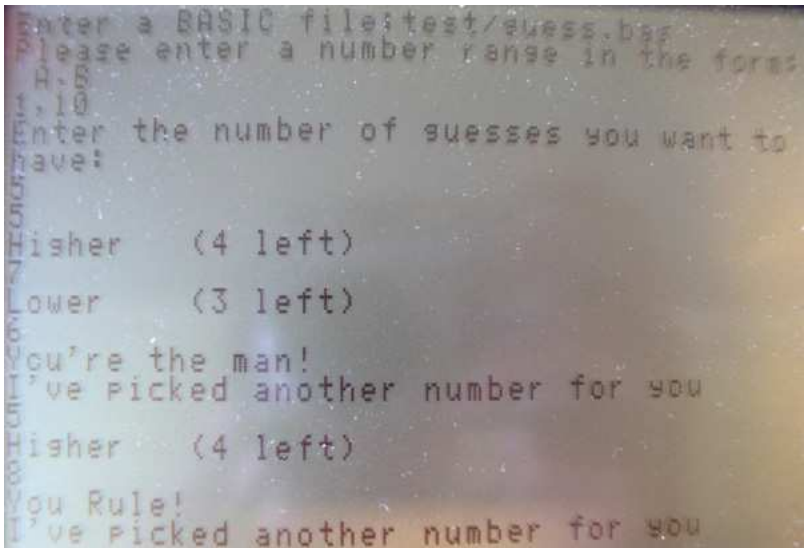
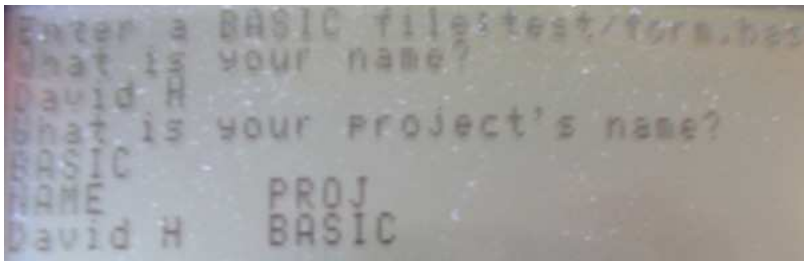


Figure 10: Form Demo Code (Example 1)



C.2 ECE476 Form

While most of the test cases were created out of a desire to make something useful, the test case ECE476 forms in `form.bas` was created to test specific features of the language. In particular, this code tests the string length function `LEN(STR$)` and string concatenation using the '+' operator. The program asks for two strings, first the user's name, second the project name. The code essentially implements a "tab" so that both the user's name and project appear under their respective headings. A list of number of spaces is searched backwards for the correct numbers of spaces to concatenate onto the name. If bigger strings are allocated first, when they are deallocated, there will be space for the smaller string in their place. See Figure 10 and 11.

```
1000 PRINT "What is your name?"
1050 INPUT STR$(0)
1100 PRINT "What is your project's name?"
1150 INPUT STR$(1)
1200 FOR I =  1 TO (LEN(STR$(0))+1) STEP 1
1300 READ SP$
1400 NEXT
1450 PRINT "NAME      PROJ"
1475 OUT$=STR$(0)+SP$+STR$(1)
1500 PRINT OUT$
```

Figure 11: Form Demo Code (Example 2)

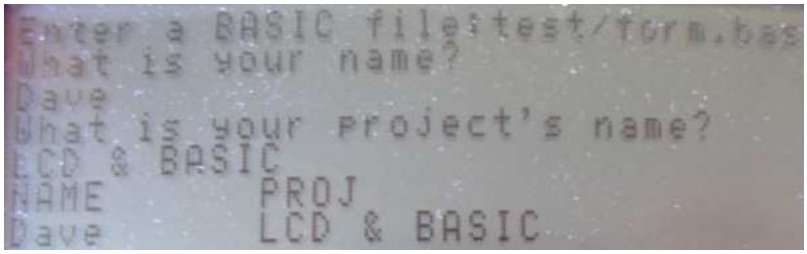
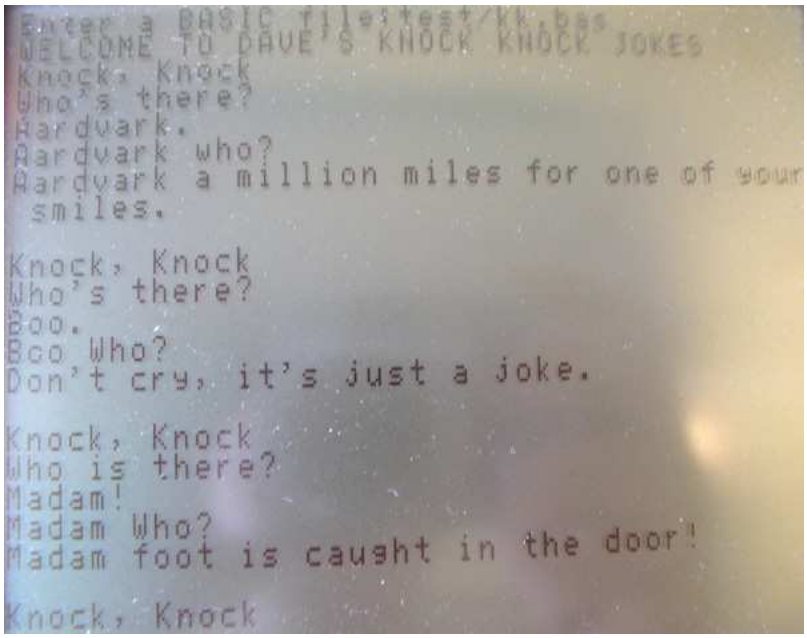


Figure 12: Knock, Knock Joke Demo Code




```
1550 GOTO 1550
1600 DATA "          ", "         ", "        ","       ", "      ", "     ","    ","   ","  ", " "
```

C.3 Knock Knock Jokes

The BASIC program `kk.bas` tells 19 different knock, knock jokes. This program tests READ/DATA statements, string allocation / deallocation, and large (multiple screen) outputs. See Figure 12.

```
1000 PRINT "WELCOME TO DAVE'S KNOCK KNOCK JOKES"
2000 R%=INT(RAND(1)*(10-1)+1)
2001 REM PRINT R%
2002 FOR I = 1 TO R%
2003 READ STR$:READ STR$
2004 NEXT
5000 PRINT "Knock, Knock"
5100 READ STR$
5200 INPUT ANS$
5300 PRINT STR$
```

```
5400 READ STR$
5500 INPUT ANS$
5600 PRINT STR$
7900 PRINT ""
8000 GOTO 5000
9000 DATA "Police.","Police let us in; it's cold out here."
9001 DATA "Doris.","Doris locked, that's why I had to knock!"
9003 DATA "Tank!","You're welcome!"
9004 DATA "Yo momma.", "Seriously, it's yo momma, open the damned door!"
9005 DATA "Aardvark.","Aardvark a million miles for one of your smiles."
9006 DATA "Boo.","Don't cry, it's just a joke."
9007 DATA "Madam!","Madam foot is caught in the door!"
9008 DATA "Old lady.","I didn't know you could yodel!"
9009 DATA "Olive.","Olive you!"
9010 DATA "Hawaii.","I'm fine, Hawaii you?"
9011 DATA "Anita!","Anita borrow a pencil!"
9012 DATA "Woo.", "Don't get so excited, it's just a joke."
9013 DATA "Pecan.","Pecan someone your own size!"
9014 DATA "Annie.","Annie thing you can do, I can do better."
9015 DATA "Police.","Police stop telling these awful knock, knock jokes!"
9016 DATA "Repeat.","Who Who Who!"
9017 DATA "Dwayne.","Dwayne the bathtub -- I'm dwowning!"
9018 DATA "Nobel.","Nobel, that's why I knocked!"
9019 DATA "Lettuce.","Lettuce in and you will find out!"
9020 DATA "Toby.", "Toby or not to be!"
```

C.4 Matrix Multiply

The BASIC program `mm.bas` performs matrix-matrix multiplication of two randomly generated N by N matrices; however, the printing of the matrices is specific to three by three matrices. It is also important to note, that this code requires at the very least nine variables per matrix (and their are three), three variables for loop counters, and one variable for the dimensionality of the three matrices — this totals thirty one variables. In actuality, more will be necessary to compute intermediate values; therefore, in this test case, I require that the parameters of the interpreter be changed. This will require a recompile and reprogram of the microcontroller. In my experience changing the following parameters

```
#define VARMEMSIZE 20
#define CHARMEMSIZE 80
#define NUMSTRINGS 6
```

to

```
#define VARMEMSIZE 34
#define CHARMEMSIZE 15
#define NUMSTRINGS 3
```

works quite well for these purposes. I would have liked to find parameters that would allow all my test cases to operate in perfect harmony. Unfortunately, there is simply not enough memory left. See Figure 13. The alignment here is not perfect, which is due to the simplicity of BASIC's print statement. Since my project is the interpreter and not the test case, I decided to settle for strange looking (but correct) output.

```
1000 N=3
1100 FOR I  = 1 TO N
1200 FOR J  = 1 TO N
1300 A%(I,J) = INT(RAND(1)*(10-1)+1)
```

Figure 13: Three by Three Matrix Multiply Demo Code

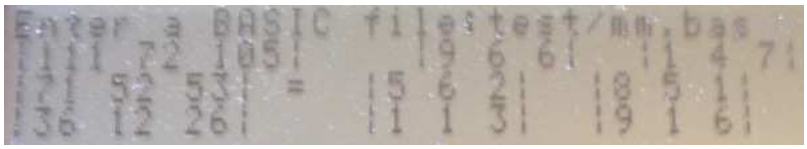


Figure 14: Prime Number Generator Demo Code

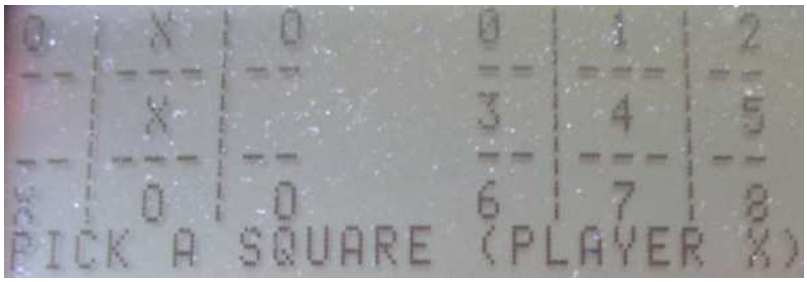

```
1350 B%(I,J) = INT(RAND(1)*(10-1)+1)
1400 NEXT J
1500 NEXT I
1600 FOR I  = 1 TO N
1700 FOR J  = 1 TO N
1800 FOR K  = 1 TO N
1900 REM PRINT "Generating C%(";INT(I);",";INT(J);")"
2000 C%(I,J) = C%(I,J) + A%(I,K) * B%(K,J)
2010 NEXT
2100 NEXT
2200 NEXT
2300 PRINT "|";C%(1,1);" ";C%(1,2);" ";C%(1,3);"|    |";A%(1,1);" ";A%(1,2);" ";A%(1,3);"|  |";B%(1
2400 PRINT "|";C%(2,1);" ";C%(2,2);" ";C%(2,3);"| =  |";A%(2,1);" ";A%(2,2);" ";A%(2,3);"|  |";B%(2
2500 PRINT "|";C%(3,1);" ";C%(3,2);" ";C%(3,3);"|    |";A%(3,1);" ";A%(3,2);" ";A%(3,3);"|  |";B%(3
2600 GOTO 2600
```

C.5 Prime Number Generator

In prime.bas, I have implemented a *very* simple prime number generator. I have not added a square root function to my language, so the algorithm involves checking each number after two for primeness, by checking the remainder of that number divided by each number between two and one less than itself. Such an algorithm becomes very slow after the first few primes. See Figure 14.

```
1000 PRINT "Welcome to the BASIC prime number generator"
2000 PRIME% = 2
3000 PRINT PRIME%;", "
4000 PRIME% = PRIME% + 1
5000 CNT%=PRIME%
6000 CNT%=CNT%-1
7000 IF CNT% = 1 THEN GOTO 3000
```

Figure 15: Two Player Tic Tac Toe Demo Code




```
8000 IF INT(PRIME%/CNT%)*CNT%=PRIME% THEN GOTO 4000
9000 GOTO 6000
```

C.6 Two Player Tic Tac Toe Game

The BASIC program `tic.bas` implement a two player tic tac toe game with a three by three game board. After each play, it clears the screen and reprints the game board. A guide indicating the keyboard $\leftrightarrow$ game board mapping is printed alongside the game board (See Figure 15). This game tests the `HOME` (clear screen and move cursor to top) command, as well as heavy testing on conditions with the `AND` logical binary operator. Recall, <> is a BASIC not equals operator.

```
1000 PRINT "WELCOME TO DAVE'S TIC TAC TOE GAME"
1010 PL%=1
1050 CH$(0) = " "
1060 CH$(1) = "X"
1070 CH$(2) = "O"
2000 REM PRINT GAME SCREEN
2001 HOME
2010 PRINT CH$(B%(0));" | ";CH$(B%(1));" | ";CH$(B%(2));"     ";"0";" | ";"1";" | ";"2"
2015 PRINT "--|---|--";"     ";"--|---|--"
2020 PRINT CH$(B%(3));" | ";CH$(B%(4));" | ";CH$(B%(5));"     ";"3";" | ";"4";" | ";"5"
2025 PRINT "--|---|--";"     ";"--|---|--"
2030 PRINT CH$(B%(6));" | ";CH$(B%(7));" | ";CH$(B%(8));"     ";"6";" | ";"7";" | ";"8"
2500 IF((B%(0)<>0) AND (B%(0)=B%(1)) AND (B%(1)=B%(2))) THEN GOTO 5000
2510 IF((B%(3)<>0) AND (B%(3)=B%(4)) AND (B%(4)=B%(5))) THEN GOTO 5000
2520 IF((B%(6)<>0) AND (B%(6)=B%(7)) AND (B%(7)=B%(8))) THEN GOTO 5000
2530 IF((B%(0)<>0) AND (B%(0)=B%(3)) AND (B%(3)=B%(6))) THEN GOTO 5000
2540 IF((B%(1)<>0) AND (B%(1)=B%(4)) AND (B%(4)=B%(7))) THEN GOTO 5000
2550 IF((B%(2)<>0) AND (B%(2)=B%(5)) AND (B%(5)=B%(8))) THEN GOTO 5000
2560 IF((B%(0)<>0) AND (B%(0)=B%(4)) AND (B%(4)=B%(8))) THEN GOTO 5000
2570 IF((B%(6)<>0) AND (B%(6)=B%(4)) AND (B%(4)=B%(2))) THEN GOTO 5000
2749 SUM%=0
2750 FOR I = 0 TO 8
2752 SUM%=SUM%+(B%(I)=0)
2754 NEXT
2756 IF SUM%=0 THEN GOTO 6000
3000 PL%=(NOT (PL%-1))+1
3010 PRINT "PICK A SQUARE (PLAYER ";CH$(PL%);")"
3100 INPUT SQ%
3101 REM PRINT B%(SQ%)
3200 IF SQ%>=9 OR SQ%<0 THEN PRINT "MUST BE BETWEEN 0 and 8":GOTO 3010
```

```
3230 IF (B%(SQ%)<>0) THEN PRINT "PIECE MUST BE FREE":GOTO 3010
3300 REM PRINT "VALID"
3400 B%(SQ%)=PL%
4010 GOTO 2000
5000 PRINT "PLAYER ";CH$(PL%);" WON"
5400 GOTO 7000
6000 PRINT "STALE MATE"
6500 GOTO 7000
7000 PRINT "Let's play again"
7001 FOR I = 1 TO 10
7200 PRINT " "
7300 NEXT I
7400 FOR I = 0 TO 8
7500 B%(I)=0
7505 NEXT
7600 GOTO 1000
```

C.7 Mary Had a Little Lamb Music

The BASIC program `mary.bas` tests the music functionality of my interpreter. It asks which of 13 octaves to play marry has a little lamb. I would suggest 4 or 5. Many of the higher octaves are not audible (at least not given the lab TV speaker).

```
10   PRINT "Marry Had a little Lamb"
200  PRINT "Which octave(0-12)?"
300  INPUT OCT%
1000 CNT%=0
1010 B=11+OCT%*12
1011 A=9+OCT%*12
1012 G=7+OCT%*12
1013 D=2+(OCT%+1)*12
1014 Q=4
1015 R=1
2000 REM PRINT ""
2001 REM PRINT ""
2010 SND(B,Q)
2011 SND(-1,R)
2012 SND(A,Q)
2013 SND(-1,R)
2014 SND(G,Q)
2015 SND(-1,R)
2016 SND(A,Q)
2017 SND(-1,R)
2020 SND(B,Q)
2021 SND(-1,R)
2022 SND(B,Q)
2023 SND(-1,R)
2024 SND(B,2*Q)
2025 SND(-1,R)
2030 SND(A,Q)
2031 SND(-1,R)
2032 SND(A,Q)
2033 SND(-1,R)
2034 SND(A,2*Q)
2035 SND(-1,R)
```

```
2040 SND(B,Q)
2041 SND(-1,R)
2042 SND(D,Q)
2043 SND(-1,R)
2044 SND(D,2*Q)
2045 SND(-1,R)
2050 SND(B,Q)
2051 SND(-1,R)
2052 SND(A,Q)
2053 SND(-1,R)
2054 SND(G,Q)
2055 SND(-1,R)
2056 SND(A,Q)
2057 SND(-1,R)
2060 SND(B,Q)
2061 SND(-1,R)
2062 SND(B,Q)
2063 SND(-1,R)
2064 SND(B,Q)
2065 SND(-1,R)
2066 SND(B,Q)
2067 SND(-1,R)
2070 SND(A,Q)
2071 SND(-1,R)
2072 SND(A,Q)
2073 SND(-1,R)
2074 SND(B,Q)
2075 SND(-1,R)
2076 SND(A,Q)
2077 SND(-1,R)
2078 CNT%=CNT%+1
2080 SND(G,3*Q)
2081 IF CNT%=1 THEN SND(G,Q): SND(-1,R): GOTO 2000
2082 IF CNT%=4 THEN SND(G,Q): SND(-1,R): GOTO 200
3020 SND(-1,R):SND(D,Q)
3050 GOTO 2000
3100 GOTO 3100
```