KI7NNP

KI7NNP SDR Transmitter Software

Version 2.4

Jed Marti

November 9, 2022

Contents

List of Figures

List of Tables

Chapter 1

The System

The KI7NNP low frequency transmitter project, LFXMIT, was to build an amateur transmitter from parts in the modern junk box. There's no longer vacuum tubes, 375 volt electrolytic capacitors and 20 pound power transformers. Now there's last generation micro-controllers, VLSI support chips, surface mount discrete components and small connectors of various sorts. Add to the junk box software libraries, compilers, linkers and flash programming; things not envisioned in the 1960's when I first began.

This manual describes the KI7NNP SDR transmitter software including includes interfaces to the important hardware components in Figure 1.1, a scripting language, a file system, and a Morse code generator.

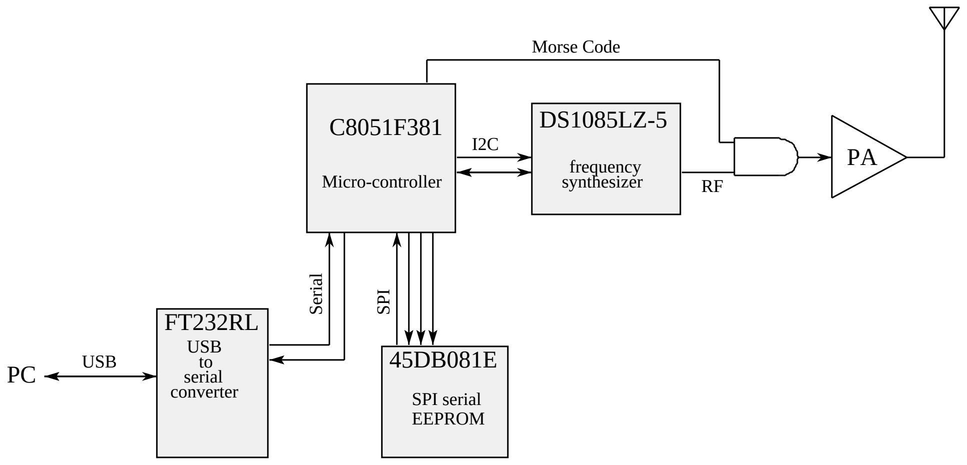


Figure 1.1: Transmitter organization

I selected the C8051F381 micro-controller[1] (MCU) from Silicon Labs because I had several on hand. It is a low power Harvard architecture device[1] with the venerable 8051 8 bit instruction set.

- 8 bit arithmetic operations including hardware multiply and divide, single bit operations and memory mapped I/O operations,
- Multi-level interrupts,
- 48 megahertz operation,
- 65k flash program storage,
- 4K + 256 bytes of RAM storage,
- I^2C bus interface,
- SPI serial interface,

[1]Program code is in re-programmable flash memory, dynamic data in a separate RAM address space.

- Serial port with variable baud rates,
- Timers,
- A/D converters,
- Numerous general purpose I/O bits.

The SDR software is optimized for this processor and requires a few tricks to shoehorn the code into the available space. The small amount of available RAM requires some dangerous programming practices.

Communication and control is through a USB serial port connected to tablet, laptop or workstation running the CTERM system or some other serial port driver[2]. This permits scripts to be downloaded, run, debugged, and maintaining the file system.

A 1 megabyte SPI serial flash memory stores scripts and data files to control transmission, data collection, and tune the frequency synthesizer for the current temperature.

- Initial flash formatting,
- File download/upload,
- Delete file,
- Display file contents,
- Execute file,
- Two files can be opened for read and write.

The scripting language is a mishmash of Basic, Python and other syntaxes. It's not designed for performing large scale computations but rather, simple activities with some level of decision making. The transmitter output is Morse code though other mechanisms can be programmed.

The MCU controls the Analog Device's DS1085 frequency synthesizer over the I^2C bus [3, 4]. A table stored in the MCU converts the requested frequency into the parameters necessary. These tables are generated by

[2]Terra Term and PUTTY are two possible programs but they do not implement an acceptable file upload/download protocol

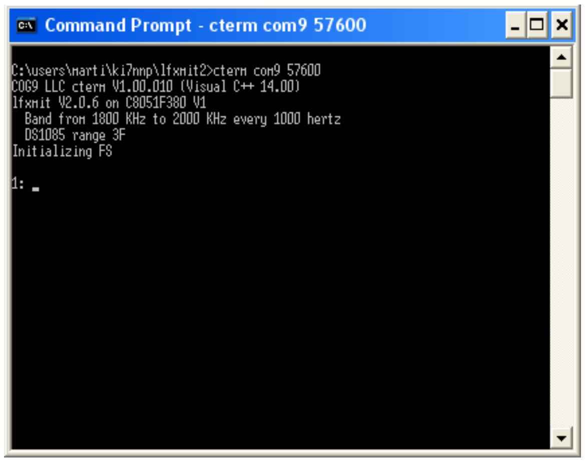
brute force off-line programs. Frequency shift keying can be accomplished by switching between two frequencies with the I^2C at a reasonable rate. Alternatively, the output can be gated to the RF power amplifier by the MCU for On/Off keying.

The PC board has analog inputs, additional GPIO bits, and four indicator LEDs. The MCU and synthesizer derive their power from the USB port but the analog RF PA section must have its own power. It can be run as a stand-alone, telemetry device or as part of a full up SDR transceiver with user control.

Chapter 2

Command Line Interface

A keyboard is more efficient than a clumsy graphics panel so control is by a command line interface (CLI). You connect the SDR to your PC's USB port, start the communications program and tell it what to do. Only command line warriors need apply.



```
C:\users\marti\ki7nnp\lfxmit2>cterm com9 57600
COG9 LLC cterm V1.00.010 (Visual C++ 14.00)
lfxmit V2.0.6 on C8051F380 V1
  Band from 1800 KHz to 2000 KHz every 1000 hertz
  DS1085 range 3F
Initializing FS

1:
```

Figure 2.1: Startup on COM9 at 57.6k baud

The **1:** indicates that the SDR accepts commands through typed input. Backspace works, ↑U and escape clear the line and enter terminates it. The serial port LED will be lit whilst waiting for characters. The maximum input line size is 120 characters.

2.1 Start up

At start up the system looks for a file named *autoexec.lf*. If present, this is a script to be run automatically. The system can run in stand-alone mode without being connected to a PC. It simply takes power from the USB port, you don't need a data connection.

2.2 File System Commands

The file system can store up 24 files with a combined size of 1 megabyte. Only one top level directory is allowed. File names are case sensitive, commands are not.

2.2.1 *dir* or *ls*

This command lists the directory contents - file names and sizes in bytes. The actual space used will be expanded to a multiple of 1k bytes. For Linux users, the abbreviation **ls** works as well. The are no arguments accepted for this command.

```
1: dir
56        pb.lf
128       pc.lf
16040     m630.dat
```

2.2.2 del *file-name* or rm *file-name*

Delete the file name that follows the command if it exists. The file and its contents are freed, the directory rewritten, and the blocks released from the FAT. For Linux users, the abbreviation **rm** is provided. The filename must be in the same case as that stored. Wild cards are not implemented.

```
1: del badprog.lf
2:
```

2.2.3 download *file-name*

If you're interfacing through CTERM![2], the **download** command will copy a file from the PC and store it on the flash file system. The file name can be up to 25 characters long. The file system activity LED will light for each 256 bytes read and written. The system will echo a period for each 64 bytes read during the transfer. The file name stored is case sensitive even if the PC isn't (such as when talking to Windows). You can download from a sub-directory but the slash becomes part of the 25 characters allowed for the file name.

```
1: download m630.bin
...........................................................................
...........................................................................
...............
2:
```

2.2.4 format

This command will initialize the file system by erasing its current contents and building an empty directory and file allocation table (FAT). The storage is divided into 1024 byte blocks of four 256 byte sectors each. The first block is reserved for the FAT and directory. Each directory entry is 32 bytes - 4 bytes for the file size, 2 bytes for the first block index, and 25 characters for the file name.

The process takes about 30 seconds and may depend upon the size of SPI flash.

2.2.5 reset

Resets the hardware, clears the symbol table, the program space and dynamic storage in preparation for a new program. This is especially useful if a script has fragmented the dynamic storage.

2.2.6 send *file-name*

Send a text file using Morse code at the current rate settings. Letters are case insensitive, digits and a few punctuation marks are supported: *at-sign* . , ? - = : ; ( ) / " \ *underscore* - +. One of the display LEDs will blink with the dots and dashes. A blank is the equivalent of the dead space between letters.

See also the **SEND** statement, section 3.6.16 on page 34, the **DITMS** variable in section 3.5.4 and the **SPACEDITS** variable in section 3.5.7 on page 27.

2.2.7 more *file-name*

Display the contents of the file on the terminal interface.

```
1: more pe.lf
// Test Frequency statement
frequency 1805000;
stop;
2:
```

2.2.8 run *file-name*

This command loads and "compiles" a script and starts executing it. This process removes comments, spaces, and new line characters. Keywords and glyphs such as <= are converted into single bytes. The symbol table is initialized with symbols that aren't keywords, their index is used instead of their characters. Numbers are converted to their binary forms. Depending upon the number of comments and blank spaces, this typically reduces the size by more than 2 to 1 and greatly speeds up execution.

The file is loaded and execution begins.

```
1: run pa.lf
Load 17 bytes
Hello World!
STOP at 0x0010
```

2.3 Other Commands

These commands are useful for tuning the RF hardware.

2.3.1 carrier on/off

Useful for tuning the output circuits, turn the carrier on or off. The blue cod LED will turn on (or off). If you have a high current power amplifier you may need to reduce its current flow to avoid heat build-up.

2.3.2 freq *integer*

Set the current frequency in hertz. The frequency must be in the current supported range.

2.3.3 gc

Execute the garbage collector.

Chapter 3

Scripting Language

Control scripts hide the most difficult 8051 programming from the user. The syntax borrows from BASIC with a mixture of whatever seemed convenient and wouldn't tax the limited storage available.

Key characteristics include:

- Automatic compacting garbage collector and dynamic storage allocation.
- File system interface,
- Program chaining,
- Fast "compilation" reduces source code size to a minimum,
- Integer and floating-point operations.

If you're expecting complicated data structures, procedures, coroutines and other features, look elsewhere.

3.1 Data Types

There are 5 available data types. A variable can have any one of these stored - its type is not fixed.

3.1.1 Integers

Integers are signed 32 bit values in the range -2147483648 to +2147483647. Arithmetic on these quantities may have wrap around overflow.

Input can be decimal, or if prefixed with 0x, hexadecimal.

3.1.2 Floats

These are single-precision, 32 bit floating-point values. Exponents can range from -38 to +38 and 5 or 6 digits in that range. Exponents are signaled by E and the base must have a decimal point. (i.e. 3E+6 is not allowed).

3.1.3 Strings

Strings are case sensitive characters enclosed in double quotes. They can be up to 118 characters long (just less than the maximum input line size). Special characters can be included by prefixing them with a back slash (\).

Special	Code	Description
\r	0x0D	Carriage return.
\n	0x0A	New line.
\"	0x22	Include a double quote in a string.
\\	0x5C	Include a backslash.

Table 3.1: Special characters in strings

Strings can be concatenated using the + operation. Adding strings to numbers causes the numbers to be converted to strings. The results can then be transmitted in Morse code.

Strings can be compared for equality - the length and characters must match. The comparison is case sensitive. When a string is compared to anything else it is not equal.

3.1.4 Byte Vectors

A byte vector is like a string, but has a fixed size and must be associated with a variable. As the value of a variable it will appear as:

<BYTEVECTOR *n*>

where n is the number of bytes in the vector. You can assign values to vector elements in the integer range 0 - 255 or access their values which are converted to 32 bit integers.

Byte vectors can be read and written to files like strings but all elements will appear, even 0's. These are particularly useful for I^2C data transfers with sensors.

3.1.5 Float Vectors

A float vector contains multiple single-precision floating-point values and must be associated with a variable. As the value of a variable it will appear as:

<FLOATVECTOR *n*>

where n is the number of values in the vector. You can assign values to vector elements with any valid floating point values. Integers will be converted to their single-precision floating-point values.

3.1.6 Boolean Values

Conditional functions, relational tests, and logical operations work with zero = false, and anything else true.

3.2 Expressions

Diadic and monadic operators are parsed and evaluated. Multiplication and division have a higher precedence than addition and subtraction.

Symbol	Description
(...)	Alter precedence.
$*$, /, %	Multiplication, division, remainder (integers only).
+, -	Sum and difference, left associative. Integers converted to floats in mixed mode. Addition of a string causes the other value to be converted to a string.
&, |	Bitwise and, or of integers only. Left associative.
$\sim$	Ones complement of an integer.
$<, <=, >, >=, !=, ==$	Relations on numbers. In the case of mixed types, integers are converted to floats.
!	Logical negation.
&&, ||	Logical AND and OR. Both sides are evaluated. Left associative.

Table 3.2: Arithmetic operations in descending precedence.

3.3 Function Calls in Expressions

A few functions are implemented. The names are case insensitive. The syntax is:

function-name(*expression*)

These can be used anywhere an expression is allowed.

3.3.1 abs

Returns the absolute value of its argument both integer and floating-point. Other types cause an error to be signaled.

3.3.2 isInt

isInt returns 1 if the argument evaluates to an integer and 0 if not.

3.3.3 isFloat

isFloat returns 1 if the argument evaluates to a floating-point value and 0 if not.

3.3.4 isString

isString returns 1 if the argument evaluates to a string and 0 if not.

3.3.5 isByteVector

isByteVector returns 1 if the argument evaluates to a ByteVector and 0 if not.

3.3.6 isFloatVector

isFloatVector returns 1 if the argument evaluates to a FloatVector and 0 if not.

3.3.7 readch

Read the next character from the selected input file handle (does not work for I^2C handle 3). In the case of files, returns -1 if end of file reached. The serial port does not detect end of file.

3.3.8 round

Converts a floating-point value to an integer. Positive values are rounded up, negative values rounded down. An integer argument is passed through unchanged, anything else signals an error.

3.3.9 sqrt

Returns the square root of a positive number, integer or floating-point. A negative value will signal an error.

3.4 Vector Indices in Expressions

You can retrieve any byte from a ByteVector or FloatVector with a subscript expression.

variable[*integer-valued-expression*]

Here *variable* must have a ByteVector or FloatVector as its value or an error is signaled. The *integer-valued-expression* must have a range of $0 \rightarrow size - 1$. Errors are signaled if the expression isn't an integer or the variable does not have a ByteVector or FloatVector value.

You assign values to a vector using the **LET** statement where the same syntax is on the left hand side of the assignment.

3.5 Variables

You can have a restricted number of variables (default 16) to store values. A variable identifier is any number of characters the first of which must be A-Z or a-z. Subsequent letters can include digits and the underscore. Variable names use precious dynamic space so long ones will eat into that available.

There are a number of special variables, not in the symbol table, but with specific values.

3.5.1 A2DC0, A2DC1

These variables are assigned to two available 10 bit analog to digital converters. Reading them causes an ADC conversion and returns a value of $0 \rightarrow 1023$. You cannot assign values to these variables.

3.5.2 ANHCHAR

If there is a character available on the serial port its value is 1 otherwise 0. It does not block while waiting for a character.

3.5.3 DIG0, DIG1

These variables will have the value of the two unassigned GPIO bits - 0 or 1 depending on the voltage being low or high. Assigning a value to these variables will change the state of the pin - 0 for ground anything else is high.

3.5.4 DITMS

This contains the number of milliseconds a Morse code dit should be on. You can set this to any positive 32 bit number. The maximum value is approximately 24 days. A dash will be 3 times the length of a dit. See also **SPACEDITS** section 3.5.7 on page 27. The default value is 100 milliseconds.

3.5.5 RAWTEMP

Accessing this variable returns the raw value of the C8051F381's on board temperature sensor with a range of $0 \rightarrow 1023$. You cannot assign a value to this variable.

3.5.6 RED, YELLOW, GREEN, BLUE

These are GPIO bits normally connected to the LEDs used to indicate system activity. Reading them gets the current state and assigning a 0 turns off the LED, anything else turns it on.

3.5.7 SPACEDITS

The number of **DITMS** intervals to leave between characters. The value can be accessed and changed. The maximum value is 255, values larger will be truncated.

3.5.8 TEMP

This returns the calibrated MCU temperature in degrees Celsius. You cannot assign a value to this variable.

3.6 Statements

Statement syntax is a mashup of Basic, RLISP, and Python. There are no "go to" statements, control flows from the first statement to the last. However, blocks of statements are allowed.

Statements are separated by semicolons and may extend over many lines.

3.6.1 Statement block

A statement block is a number of statements enclosed in curly brackets. These curly block statements can be nested as well though storage limits this to only a few levels.

$$\{ <statement_0>; \ldots <statement_n>; \}$$

3.6.2 BYTEVECTOR statement

Creates a byte vector with up to 255 bytes.

BYTEVECTOR <*variable*>**[** <*expression*> **]**;

Any value the variable had is removed and replaced with a ByteVector the size of the expression. The expression must be an integer value greater than 0 and less than or equal to 255.

3.6.3 CHAIN statement

Because only modest size scripts can be run, the **CHAIN** operation permits chunks to be strung together.

CHAIN <*string*>;

On encountering this command the following are performed:

1. Hard close all open files. If you have a file open for writing, its contents will be lost.
2. The file named is opened for input - an error signaled if it's not found. The input scanner is switched to this file.
3. The dynamic storage space is cleared and reinitialized.
4. The symbol table is cleared.
5. The fixed integers (0 and 1) are initialized.
6. The parser stack is initialized.

7. The FOR stack is initialized.
8. The program is loaded and execution begun.

To communicate between segments, write any variable values you need to a file and restore them at the start of the next script.

3.6.4 CLOSE statement

Close one of the open files finishing the final write on any file open for output. The file handle is released for further use.

CLOSE <*expression*>;

The expression must be an integer and have the value 0 or 1. Files not open or a bad value will cause an error halt.

See also the **CREATE** statement below and **WRITE** in section 3.6.20 on page 36.

3.6.5 CREATE statement

The system will create a new file if possible and assign its file handle to the variable given.

CREATE <*variable*> = <*expression*>;

Evaluate the expression which must be a string. Create the file if possible and assign the handle (0 or 1) to the variable. Errors occur if too many files are open or the file system is full. See also the **OPEN** statement, section 3.6.12 on page 32.

3.6.6 DIRECTION statement

This sets some of the parameters for the digital I/O lines.

DIRECTION [**DIG0** | **DIG1**] [**IN** | **OUT**] [**PP** | **OD**];

In any order, you must specify which pin **DIG0** or **DIG1** (not both, not neither). The pin can be either **PP** for push-pull (usual for low power output), or **OD** open drain typical for input or higher power. The pin can be either for input **IN** or output **OUT**. The default is **DIG0**, **IN**, and **OD**.

See the variables **DIG0** and **DIG1** in section 3.5.3 on page 26.

3.6.7 FLOATVECTOR statement

Creates a FloatVector with up to 255 entries

FLOATVECTOR <*variable*>**[** <*expression*> **]**;

Any value the variable had is removed and replaced with a FloatVector the size of the expression. The expression must be an integer value greater than 0 and less than or equal to 255.

Each single-precision floating-point value takes 4 bytes and RAM storage is very limited. Vector sizes greater than 90 will severely impact performance.

3.6.8 FOR statement

The **FOR** statement performs a loop iterating a variable value.

FOR <*variable*> = <$expr n_s$> **TO** <$expr_{to}$> [**BY** <$expr_{by}$>]
<*statement*>;

All expressions are evaluated when the **FOR** statement is encountered. If the optional **BY** clause is not present, the increment is assumed to be the integer 1. Both integer and floating point values are permitted.

FOR statements can be nested up to 3 deep. The variable is incremented from its initial value until it is greater than the **TO** value. If the increment is less than 0, then the decrement happens until the value is less than the **TO** value.

Multiple **FOR** and **WHILE** loops can be nested (see section 3.6.19 on page 35). The default depth is 3.

This example fills a FloatVector with values.

```
FloatVector a[10];
FOR i = 0 TO 10
{
    LET a[i] = sqrt(i + 3);
}
STOP;
```

3.6.9 FREQUENCY statement

Executing this statement sets the DS1085 frequency to the number of hertz specified or the nearest possible value.

FREQUENCY <*expression*>;

The expression must evaluate to an integer. to 2000000 hertz with a resolution of 1000 hertz.

3.6.10 IF statement

The usual meaning - if an expression is true or false (see section 3.1.6 on page 23) perform some action or not.

IF (<*expression*>) <$statement_t$>; [**ELSE** <$statement_f$>;]

If < *expression* > is true, then execute < $statement_t$ > otherwise skip to the next statement. If the **ELSE** statement is present, then it is executed if < *expression* > is false and < $statement_t$ > is skipped. Of course, the statements can be blocks for more complex operations.

3.6.11 LET statement

Executing this assigns values to a variable or element of a vector. The **LET** is optional.

LET <*variable*> = <*expression*>;

<*variable*> = <*expression*>;

Assign a value to the variable. The expression can have an integer, floating-point, string, or vector value. A ByteVector or FloatVector is not copied, just its address is assigned. Special variables that can be assigned are also permitted.

LET <*variable*>[<*subscript*>] = <*expression*>;

Here is an assignment to an element of a ByteVector. The < *subscript* > expression must evaluate to an integer in the range $0 \rightarrow size - 1$ or an error is signaled. For ByteVectors, floating-point values are truncated to $0 \rightarrow 255$ before assignment. Only the last 8 bits of an integer are assigned. Strings and other values cause an error. For FloatVectors, integers are converted to floating-point before assignment, other types cause errors.

3.6.12 OPEN statement

Open a file for reading and assign its handle to the variable.

OPEN <*variable*> = <*expression*>;

Evaluate the expression which must be a string. Open the file file if possible and assign the handle (0 or 1) to the variable. Errors occur if too many files are open or the file doesn't exist. Remember that file names are case sensitive. See also the **CREATE** statement, section 3.6.5 on page 29.

3.6.13 PAUSE statement

Stop execution for a period of time.

PAUSE <*expression*> [**MS** | **MILLISECONDS** | **S** | **SECOND** | **SECONDS** | **M** | **MINUTE** | **MINUTES**];

The expression by default is the number of milliseconds. This can optionally be followed by a keyword indicating a multiplier.

Keyword	Description
MS	Milliseconds
MILLISECONDS	Milliseconds
S	Seconds, value multiplied by 1000
SECOND	...
SECONDS	...
M	Minutes, value multiplied by 60000
MINUTE	...
MINUTES	...

Table 3.3: PAUSE multipliers.

Floating point values are accepted and used with the multiplier but converted into an integer number of milliseconds. The smallest value that can be timed is one millisecond.

3.6.14 PRINT statement

Display values on the serial port.

PRINT <*expression*$_0$>[, :] ...<*expression*$_n$>;

Each expression is evaluated and displayed separated by a blank (for a comma) or no blank if a colon was present. After the last, the line is terminated.

Integers Displayed with minus sign.

Floats With 3 digits to the right of the decimal point[1].

Strings Just their characters.

ByteVectors Shows up as <**BYTEVECTOR** *n*> where *n* is the number of bytes reserved.

FloatVectors Shows up as <**FLOATVECTOR** *n*> where *n* is the number of elements reserved.

The following example demonstrates some of the functions and **PRINT**.

```
PRINT "pm.lf function test";
PRINT isInt(32):"->":isInt(32.4);
PRINT isFloat(32.5), isInt("foo");
PRINT isString("foo"), isString(34.5);
PRINT round(34.7), round(24), round(-3.7);
PRINT sqrt(4), sqrt(9.3);
PRINT "abs int ": abs(-3222): " float ":abs(34.5);
STOP;
```

See also the **WRITE** statement, section 3.6.20 on page 36.

[1]Needs to be fixed.

3.6.15 READ statement

The are four possible input files with the indices as shown in Table 3.4.

Handle	Description
0	File system handle. Assigned by **CREATE** or **OPEN**.
1	File system handle. Assigned by **CREATE** or **OPEN**.
2	Serial port.
3	I^2C followed by the bus address.

Table 3.4: **READ/WRITE file handles.**

The statement accepts one or more variables to fill with values from the input stream.

READ $<expression>$, $<var_0>$, $\ldots$, $<var_n>$;

When asked to read, the system reads a line from the selected file in the first expression up to and including the new line character. The items are then assigned one at a time to the variables listed. Accepted are strings, floats, and integers. Identifiers will be treated like strings.

In the case of I^2C, the first variable should have the device address - 8 bits but with the low order bit ignored.

See also the **WRITE** statement, section 3.6.20 on page 36 and the **CLOSE** statement, section 3.6.4 on page 29. See also the **readch** function on page 25.

3.6.16 SEND statement

Send a string to the RF system as Morse code.

SEND $<expression>$;

Evaluate the expression which must be a string and the send as Morse code. See also the **send** command in section 2.2.6 on page 17.

3.6.17 SLEEP statement

This is similar to **PAUSE** (see section 3.6.13 on page 32) except that the system is put in a low power mode.

SLEEP <*expression*>;

The expression must have an integer value of the number of seconds to sleep. During sleep:

- The MCU high frequency clock is turned off to save power,
- All LEDS are turned off to save power,
- The DS1085 frequency synthesizer is powered down,
- The 80 KHz low frequency clock is used to interrupt at approximately 1 second intervals until the second count expires,

3.6.18 STOP statement

Stop execution and return control to the command level.

STOP;

3.6.19 WHILE statement

Perform a statement while an expression has a non-zero value.

WHILE <*expression*> <*statement*>;

Execute the statement over and over until the expression has a value of 0. It must be an integer value. The statement also uses the FOR stack so nesting is coupled with it (see section 3.6.8 on page 30).

This example waits for a character from the serial port before exiting.

```
WHILE !ANYCHAR { PAUSE 500; };
PRINT "Done!";
STOP
```

3.6.20 WRITE statement

Write values to a selected file, one of those in Table 3.4 on page 34.

$$\textbf{WRITE} <expression>, <expr_0>, \ldots, <expr_n>;$$

The first expression value must be an integer and is one of the file handles. In the case of I^2C, $<expr_0>$ must be a 7 bit device address with the 8th bit (low order) ignored. Subsequent expressions are written as for the **PRINT** statement.

Any file file system file that is written to must be closed with **CLOSE** or the characters will not be saved.

Chapter 4

Source Code

The source code is covered by the GNU Public License. You're free to do most anything with it except make money or claim that you wrote it.

4.1 Compilation

The code was compiled using a Linux version of the Small Device C Compiler (SDCC) version 3.8.0. Changes to work with a GNU compiler usually involve only changing how 8051 registers bits are named.

The **Makefile** lists all the modules and their dependencies. The final lines convert the hex format to one used by the Silicon Labs programmer.

```
make
make load
```

To clean up the directory do the following. You'll have to change the value of **RM** to 'del' if you're running this on a Windows machine.

```
make clean
```

4.2 Leniency for Micro-controllers

Many concessions to the 8051 micro-controller complicate the code. Only some 114 or so bytes of directly addressable RAM are available and many of these are used by the support libraries, functions with more than one parameter, or functions with many local variables. Hence, **there are lots of shared global variables**!

The larger 4k RAM space requires special operations to access it - much additional code and frequent use of temporaries. Stored here are the instruction space and data space, stacks and various infrequently needed values.

Variable	**Space**	**Description**
IACC, IACCB	xdata	Two 32 bit values that can be either float or integer and subdivided into 16 and 8 bit values.
DREG	RAM	A 16 bit value for accessing D space.
DREGB	xdata	A auxiliary 16 bit value for D space.
FACC, FACCB	xdata	Two 32 bit values that can be either float or integer and subdivided into 16 and 8 bit values. Intended for floating-point values.
STR[]	xdata	For string manipulation. Sets the maximum size on strings.

Table 4.1: Shared global variables.

Many of the routines in the dynamic storage allocator work with these variables rather than passing them as parameters.

4.2.1 SDCC 8051 specifics

The 8051 requires extra declarations to indicate where variables should be stored.

Declaration	Description
__bit	Indicates this is single bit value. Can be only 0 or 1. Very space and speed efficient.
__code	Indicates a constant value stored in code space. Cannot be modified and requires different internal code to access. Similar to C *const* declaration.
__xdata	Indicates code is stored in the extended RAM space. Requires extra internal code for access.

Table 4.2: 8051 SDCC declaration extensions.

4.3 Configuration

The *config.h* file controls the connection between chip pins and their functions. The file also sets the serial port baud rate, the I^2C transfer rate, the LED pins, DS1805 control pins, the Morse code pin and the timer rate.

The *lfxmit.h* file provides widely used function prototypes. It also contains the version, release and patch numbers that appear on the system logo.

The **VERSION** number changes in response to radical system reorganization (this has happened once already). The **RELEASE** number changes when I post changes that may make a new version incompatible with the previous. The **PATCH** number changes on a whim and usually indicates a bug fix.

4.4 Storage Configuration

The total size of the xdata space must not exceed 4096 bytes. You can check for overflow by examining *lfxmit.mem* and look for EXTERNAL RAM - the size should be less than or equal to 4096. The storage sizes are set in *storage.h.*

Variable	Default	Description
DSIZE	1023	The D space size - the maximum is 1023 bytes.
FORDEPTH	3	The number of FOR/WHILE loops that can be nested. Each entry takes 6 bytes.
ISIZE	1650	The I space size - the largest compiled program.
PSTACK	5	The size of the parse and evaluation stack. Each entry takes 2 bytes.
SSIZE	16	The number of symbol table entries. Each entry is 4 bytes.

Table 4.3: Storage configuration parameters in *storage.h*.

4.5 Interrupts and Timers

Some timers are run on interrupts, the other peripherals are operated by polling. You'll find the interrupt prototypes in *lfxmit.c*.

Timer	Int#	Description
0	-	Available
1	-	UART baud rate
2	-	I^2C 100 KHz timer, I^2C interrupt 7
3	14	Millisecond clock for **PAUSE** and Morse code
4	19	Low Frequency clock timer for **SLEEP**
5	-	Available

Table 4.4: Timers and Interrupts

4.6 Low RAM Use

The SDCC uses low RAM for passing more than one argument and temporaries in case stack space is too difficult. The source code uses the 4k external RAM extensively. Code reviews removed many of these by using globals (dangerous) and reorganizing expressions.

Code	RAM	Notes
arithmetic.c	0	
bits.c	0	
bytevector.c	0	
chain.c	0	
cmd.c	0	
digio.c	0	
dmalloc.c	4	**DREG** and a local variable.
download.c	0	
ds1085.c	10	Search the band table big offender.
files.c	15	Difficult arithmetic for output, read a pig.
floats.c	0	
floatvector.c	0	
for.c	4	Floating-point increment.
fs.c	13	Extra parameters, local variables.
functions.c	0	
i2c.c	11	Extra parameters.
if.c	0	
init51.c	0	
ints.c	0	
let.c	0	
lex.c	9	Local variable and arithmetic.
lfxmit.c	0	Main program.
logical.c	0	
more.c	0	
morse.c	0	
parse.c	1	**SPTR** stack pointer.
pause.c	0	
print.c	0	
progn.c	0	
program.c	0	
relations.c	0	
run.c	0	
serial.c	1	Second parameter.
sleep.c	0	
spi.c	4	Second parameters.
strings.c	9	Second parameter, temporary.
symtab.c	0	
vector.c	1	Subscript checking.
while.c	0	

Table 4.5: Low RAM usage by source code file, early version 2.

If you make significant code changes you need to verify that you haven't exceeded the available RAM or program storage. This is particularly true if you start using the transcendental library functions. Examine the **lfxmit.mem** file generated by the compiler.

The first part shows allocation of the 256 bytes of low RAM.

```
Internal RAM layout:
      0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00:|0|0|0|0|0|0|0|0|a|a|a|a|b|b|b|b|
0x10:|b|b|b|b|b|b|b|b|b|b|b|b|b|b|g|g|
0x20:|B|B|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
0x30:|c|d|d|d|d|e|e|e|e|e|e|e|e|e|e|e|
0x40:|e|e|f|f|f|h|h|h|h|h|h|h|h|h|i|j|
0x50:|k|l|l|l|l|m|m|m|m|m|m|m|m|n|o|p|
0x60:|p|q|q|q|q|Q|Q|Q|Q|Q|Q|Q|Q|Q|Q|Q|
0x70:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xc0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xd0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xe0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xf0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0-3:Reg Banks, T:Bit regs, a-z:Data, B:Bits, Q:Overlay, I:iData, S:Stack, A:Ab
lute
```

If the Q values (overlay) extend beyond 0x7F you're in trouble. Though 128 bytes of stack space aren't ever used, you can only put indirectly references vectors in that space. In the example above, there are 16 bytes left for general use.

Thye last values are also important. For this processor, you may not have more than 4096 bytes of EXTERNAL RAM. We have 25 bytes left, add them to the symbol table, deeper FOR loops, or use XDATA for more variables.

```
Other memory:
   Name             Start    End      Size     Max
   ---------------- -------- -------- -------- --------
```

```
PAGED EXT. RAM                        0      256
EXTERNAL RAM     0x0001   0x0fe7   4071    65536
ROM/EPROM/FLASH  0x0000   0xd8ad  55470    65536
```

Chapter 5

DS1085 Programming

There are 54,525,952 possible frequency settings for the DS1085 frequency synthesizer including multiple possible settings for some output frequencies. I decided to use a brute force algorithm to build setting tables for each amateur band. Depending upon the band range and frequencies these can have resolutions as fine as ten hertz or as low as one kilohertz.

Two different versions of the source code are implemented:

1. **ds1085.c** Generates a .h file with the internal settings for the band selected.
2. **ds1085err.c** Generates error .csv files for a range of frequencies. Used for analysis purposes.

You would use these to generate your own lists of frequencies or extend the range outside if you want to use the system as a variable frequency generator but without a radiating element.

5.1 ds1085 program

Modifications can be made to this program. The following table is taken from the DS1085 documentation [5]. It shouldn't be modified unless the documentation changes.

```
// The offset table taken from the DS1085 documentation - the master
// oscillator base frequencies.
typedef struct {
    int8_t offset;                    // Offset from base RANGE value.
    int32_t lowf;                     // The lowest frequency (DAC == 0).
    int32_t highf;                    // The highest frequency.
} ftable;

// The table of range values for the master oscillator.
ftable OffSets[] =
{{-6, 30700000, 35800000},
    {-5, 33300000, 38400000},
    {-4, 35800000, 41000000},
    {-3, 38400000, 43500000},
    {-2, 41000000, 46100000},
    {-1, 43500000, 48600000},
    {+0, 46100000, 51200000},
    {+1, 48600000, 53800000},
    {+2, 51200000, 56300000},
    {+3, 53800000, 58900000},
    {+4, 56300000, 61400000},
    {+5, 58900000, 64000000},
    {+6, 61400000, 66600000}};
```

The command line needs 2 arguments, the band in meters and the file name to use for output. You can change this if additional arguments are needed.

```
    // Accepts a band (meters, and the .h file. Error if wrong number of
    // command line arguments.
    if (argc != 3)
    {
        printf("Usage: ds1085 band file\n!");
        printf(" band=2200,630,160,80\n");
        printf(" file - .h file\n");
        exit(-1);
    }
```

The following code converts the band in meters to the start and end frequencies and the increment between selectable frequencies. You might want to extend the upper frequency as the master oscillator slows with cold temperatures. You can adjust the increment `finc` as well though making it too small may overflow available code space. At the higher frequencies you won't get as good a resolution anyway.

```
// Get the band. Error if not a viable one. Set the frequency ranges
// and the increment all in hertz.
band = atoi(argv[1]);
if (band == 2200)
    { sfreq = 135700; efreq = 137800; finc = 10; }
else if (band == 630)
    { sfreq = 472000; efreq = 479000; finc = 10; }
else if (band == 160)
    { sfreq = 1800000; efreq = 2000000; finc = 1000; }
else if (band == 80)
    { sfreq = 3500000; efreq = 4000000; finc = 1000; }
else if (band == 40)
    { sfreq = 7000000; efreq = 7300000; finc = 1000; }
else
{
    printf("Band %d meters not supported\n", band);
    exit(-1);
}
```

The rest of the code should not be modified unless you want to pretty up the output format.

5.2 ds1085err program

This is a program used to examine frequency errors for a QST paper.

Chapter 6

Some Examples

Some example programs to demonstrate the facilities.

6.1 Simple Beacon

Send a call sign, wait 10 seconds - run until a keyboard input.

```
// Beacon program.
FREQUENCY 1810000;
WHILE (!ANYCHAR)
{
   SEND "vvv KI7NNP test K";
   SLEEP 10;
};
STOP;
```

6.2 Linear Least Squares

You can do limited numerical analysis to prepare raw data for sending. The following simple code reads a file of x,y numbers and does a linear least squares regression on it.

```
// KI7NNP test program.
// Read a file of 10 x,y values and compute the linear least
// squares values to fit the data points.
FLOATVECTOR x[10];
FLOATVECTOR y[10];

// Read file with random numbers.
OPEN fh = "rnd.dat";
FOR i = 0 TO 9 READ fh, x[i], y[i];
CLOSE fh;

// Dump data.
FOR i = 0 TO 9 PRINT x[i], y[i];

LET avgx = 0.0;
LET avgy = 0.0;
FOR i = 0 TO 9 {
    LET avgx = avgx + x[i];
    LET avgy = avgy + y[i];
};

LET avgx = avgx / 10.0;
LET avgy = avgy / 10.0;

LET num = 0.0;
LET den = 0.0;
FOR i = 0 TO 9 {
    LET num = num + (x[i] - avgx) * (y[i] - avgy);
    LET den = den + (x[i] - avgx) * (x[i] - avgx);
};

IF (den == 0.0) PRINT("Vertical Line!");
ELSE
    PRINT "y = ": num / den: "*X + ": avgy - (num / den) * avgx;
STOP
```

A sample data file that works with **READ**.

```
0.0717052 -0.24952
1.05267 1.44667
1.87982 1.9049
2.93783 2.52271
4.00516 3.84467
4.94355 5.06103
5.91001 6.04386
7.00548 7.35708
7.99994 7.91937
9.06107 8.74917
```

And the result of running the script.

```
lfxmit V2.2.2 on C8051F380 V2
  160 meters, 1800 kHz to 2000 kHz by 1000 Hz
  D-space 1023 bytes
  I-space 1675 bytes
  Symtab 16 entries
  Loops 3 nested
  Initializing FS

1: run llsq.lf
Loaded 341 bytes
0 0.07170 -0.24952
1 1.05267 1.44667
2 1.87981 1.90490
3 2.93782 2.52271
4 4.00515 3.84467
5 4.94355 5.06102
6 5.91000 6.04385
7 7.00547 7.35707
8 7.99993 7.91937
9 9.06107 8.74917
y = 1.00169*X + -0.03434
STOP at 0x0154
2:
```

Chapter 7

Regression Testing

I tested the system with many small scripts. These run to completion or halt with an error. Testing error checking will have an indication of the expected error message before halting. An error halt message will be prefixed with 5 asterisks, other messages are just informative.

The source files are found in the `regr` directory.

7.1 Hello World: `hello.lf`

Test basic operation by displaying a message and stopping.

Failure Program does not load, error messages, control not returned to command line.

Success Displays message `KI7NNP LFXMIT Regression Test` followed by the STOP command index.

Relies on tests None.

7.2 Control Flow: `control.lf`

A simple test of the **if** and block statements. These must work or the remainder of the tests will not.

Failure Program does not load, error messages, control not returned to command line.

Success Some messages not prefixed with 5 asterisks and a valid stop statement.

Relies on tests `hello.lf`.

7.3 Arithmetic Relations: `relation.lf`

Test basic arithmetic relations: equal, not equal, less than, less than or equal to, greater than, greater than or equal to. Tests are between two integer or two floating-point values. The tests are numbered - if there's an 'f' following the number it means a float vs float comparison failed. The parsed code is predominated by error strings, hence the messages are shortened.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number.

Success Displays `"relation.lf succeeds"`.

Relies on tests `control.lf`.

7.4 Arithmetic Relations: `relation1.lf`

This tests relations between integer and floating-point values as above. It tests integer *relation* float, and float *relation* integer as these are separate components in the execution code.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number.

Success Displays `"relation1.lf succeeds"`.

Relies on tests `control.lf`.

7.5 String Relations: `relation2.lf`

Strings can be compared with == and !=. Test these with strings and numbers.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number.

Success Displays `"relation2.lf succeeds"`.

Relies on tests `control.lf`.

7.6 Arithmetic: `arithmetic.lf`

Test the arithmetic operations on fix and mixed modes. Test string concatenation.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter.

Success Displays `"arithmetic.lf succeeds"`.

Relies on tests `relation.lf`, `relation1.lf`, `relation2.lf`.

7.7 Functions: `functions.lf`

Test the basic functions **abs**, **round**, **sqrt** and the type testing functions **isInt**, **isFloat**, **isString**, **isByteVector**, and **isFloatVector**.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter.

Success Displays `"functions.lf succeeds"`.

Relies on tests `relation.lf`.

7.8 Loops: `loops.lf`

Test the **for** and **while** loops. The particularly messy for loop has mixed mode increment and testing.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter and extensions.

Success Displays `"loops.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`.

7.9 ByteVectors: `bytevector.lf`

Test creation, assignment, and referencing of byte vectors. Automatically chains to `floatvector.lf`.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter and extensions.

Success Displays `"bytevector.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`, `loops.lf`.

7.10 FloatVectors: `floatvector.lf`

Test creation, assignment, and reference of floating-point vectors.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter and extensions.

Success Displays `"floatvector.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`, `loops.lf`, `functions.lf`.

7.11 Logical Operations: `logical.lf`

Test the logical operations. This aren't implemented correctly but will have to do for now. The operator precedence appears to be OK.

Failure Program does not load, error messages, gc failure, control not returned to command line. Examine program source for test failure number or letter.

Success Displays `"logical.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`.

7.12 Bit Operations: `bitops.lf`

Test the integer bit operations - these only work on integers.

Failure Program does not load, error messages, control not returned to command line. Examine program source for test failure number..

Success Displays `"bitops.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`.

7.13 Symbol Table: `symtab.lf`

Test some operations on the symbol table. We fiddle the Morse code timing, turn the LEDs on and off in a patter, and flip the digital I/O lines. Test the symbols and the direction and pause statements.

Failure Program does not load, error messages, control not returned to command line. Examine program source for test failure number..

Success Blinks LEDs, flips digital I/O lines up and down, displays `"symtab.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`.

7.14 Files: `rw.lf`

Test reading and writing files. Test character I/O and having two files open. Creates a file with 3 values in it, reads the values back. Copies the file from the first steps to a second.

Failure Program does not load, error messages, control not returned to command line. Examine program source for test failure number. May fail if file system is full.

Success Some messages and displays `"rw.lf succeeds"`.

Relies on tests `relation.lf`, `control.lf`, `loops.lf`.

Index

Bibliography

[1] Silicon Labs. *C8051F380/1/2/3/4/5/6/7/C Full Speed USB Flash MCU Family*. **https://www.silabs.com/documents/public/data-sheets/C8051F38x.pdf**.

[2] Jed Marti. *CTERM Serial Communication*. **https:/www.cog9llc.com**.

[3] NXP. *I^2C-bus specification and user manual*, June 2007. UM10204, Rev. 03.

[4] Dominique Paret and Carl Fenger. *The I^2C Bus From Theory to Practice*. John Wiley & Sons, New York, 1997.

[5] Analog Devices (DALLAS Semiconductor). *DS1085L 3.3V EconOscillator Frequency Synthesizer*. **https://datasheets.maximintegrated.com/en/ds/DS1085L.pdf**.