

The GNAT Byte Code Interpreter Manual

Eric R. Dodson, Keith J. O'Hara, and Tucker Balch
Georgia Institute of Technology
<http://www.cc.gatech.edu/~borg/gnats/>

May 8, 2005

Abstract

The significant program memory constraints (4K) of the PIC 16F87 Microprocessor make it necessary to employ other means to implement large programs or a large number of smaller programs. In an effort to make this possible, a Byte Code Interpreter has been created to allow a user to program in a higher level language, such as Forth, on a general purpose computer, and compile this program into byte code which can be downloaded into the PIC onchip data EEPROM or into its *i²c* external memory. A byte code interpreter running on the PIC reads these byte codes and executes the desired functions on their behalf.

In this way, byte code programs can grow to the maximum size of the memory being used, or many individual byte code programs can be loaded from any source. This manual assumes that the byte code program will be loaded from the MPLAB IDE, but also mentions some other possibilities such as Mobile Code in a later section.

This manual is not intended to be a full Forth users guide, but it does point the unfamiliar reader to an excellent online Forth tutorial resource. There are 2 major sections in this manual: 1) a higher level user's programming guide explaining the Gnat environment modifications to Forth with a tutorial explaining several sample Gnat Forth programs, and 2) a more in-depth exploration of the Gnat Forth system with full descriptions of all source code files and process flow. Lastly, there is a discussion of Mobile Code, the Gnat Exerciser, and a full listing of the Forth words implemented in Gnat Forth.

Contents

1	Byte Code Interpreter	4
1.1	What is Forth?	4
1.1.1	Why Forth and not a C-like language?	4
2	GNAT Byte Code Application Programmer's Guide	5
2.1	Introduction to Forth	5
2.2	How to Compile a Forth Program, Download Hex Code to the Gnat, and Run Programs.	6
2.2.1	Runs on Linux or Mac OS X	7
2.2.2	gnatforthlib.gf	7
2.2.3	C-like #defines for constants	7
2.2.4	GnatForth Makefile	7
2.3	Event driven nature of Gnat Forth	8
2.4	Using the example programs	9
2.4.1	Sample.gf	9
2.4.2	VarTest.gf	11
2.4.3	RAMvarTest.gf	14
2.4.4	RAMlocalVar.gf	14
2.4.5	battery.gf	15
2.4.6	thermal-sensor.gf	18
2.4.7	SendReceive.gf	19
3	GNAT Byte Code System Programmer's Guide	23
3.1	Gnat Forth Environment Overview	23
3.2	Gnat Forth Compiler Overview	23
3.3	The Gnat Forth Communication Environment	25
3.4	How to add/modify/delete a Gnat Forth token	26
3.4.1	Deleting a Gnat Forth token	27
3.4.2	Modifying a Gnat Forth token	28
3.4.3	Adding a Gnat Forth token	28
3.5	gnatforth.c - the Gnat Byte Code Compiler	29
3.5.1	init()	31
3.5.2	get_token()	31
3.5.3	get_token_type()	31
3.5.4	process_token()	31
3.5.5	add_new_symbol()	33
3.5.6	get_symbol_attributes()	33
3.5.7	get_next_ram_location()	33
3.5.8	get_next_onchip_eeprom_location()	33
3.5.9	write_bytecode_to_dictionary()	33
3.5.10	write_value_to_dictionary()	33
3.6	gnatforth.h	34
3.7	tokens.h	34
3.8	interpret.c - the Gnat Byte Code Interpreter	34

3.8.1	Program Constants	34
3.8.2	Global variable definitions	35
3.8.3	The data and return stacks - global	35
3.8.4	execute_interpreted_program()	35
3.8.5	main()	37
3.8.6	get_8bit_bytecode()	37
3.8.7	get_16bit_bytecode()	38
3.8.8	push_data() and push_ret()	38
3.8.9	pop_data() and pop_ret()	38
3.9	Still Todo	38
4	Mobile Agent Code	39
4.1	Introduction to Mobile Code	39
4.2	Mobile Code Protocol	39
4.3	Context switching of mobile code agent	40
4.4	Mobile Code Future...	41
A	The GNAT Exerciser Manual	42
A.1	Introduction	42
A.2	Procedure	43
B	List of Forth Words	44
B.1	Forth keywords run directly by the Gnat Byte Code Interpreter	44
B.1.1	API keywords	44
B.1.2	Standard Forth keywords	44
B.1.3	Gnat specific Forth Non-API keywords	46
B.2	Forth Keywords Implemented in Gnat Forth	46

1 Byte Code Interpreter

1.1 What is Forth?

Forth is a stack-based language. Not stack-based in the C function call stack sense, but stack-based in the Push Down Automaton sense. Operands and function arguments are placed on the stack before the operator or function is called. The operator consumes the values off the stack, performs the operation, and places the results back onto the stack. This is what is commonly called “postfix” or “Reverse Polish Notation”, and is quite different than our usual way of “infix” thinking. See <http://c2.com/cgi/wiki?PostfixNotation> for a full discussion of postfix notation.

Forth actually uses two stacks:

- a data stack, and
- a return stack.

The data stack is used for most operands and data values, and the return stack is used for execution flow control statements, subroutine return addresses, loop counters, etc.

You will find that Forth is a simple language to learn, and one that lends itself very well to the embedded programming environment.

1.1.1 Why Forth and not a C-like language?

When searching for a source language, we knew it must be very Gnat specific (i.e., it should contain only the byte codes and program structures that were needed by the Gnats) and restricted (i.e., it should contain the smallest subset of byte codes and program structures that were needed for the Gnats). Our programs will only run on the Gnats, so there is no need for more power or generality than we really need. This gives us conciseness and restricts the space of possible programs, possibly letting us prove things about GNAT programs, such as termination or schedulability. Proving Gnat Forth program termination will be the subject of a future investigation.

One aspect of a Forth-based system that makes the process easier is small concise programs. Small programs allow us to separate our applications’ concerns easily. Applications can be single-minded as there is no need for an application to do more than one task.

Another aspect is that writing a Forth byte code compiler and the companion interpreter is not as difficult as modifying the code generation piece of a more general purpose compiler such as gcc. Forth is a simple language, which isn’t a problem since our application nuggets are simple as well. In this respect, C is too general and overkill for our small application.

NQC (Not Quite C) was investigated and could have been a good idea, but we think Forth is a better option for some of the generality reasons mentioned above. One reason we decided against NQC is that we might stop using NQC on the robots because of its limitations. The limitations are mostly due to its

reliance on the RCX firmware which has weak communication support (1-byte messages). So, using NQC on both the Gnats and the robots is unlikely. Also, NQC generates very RCX specific bytecodes, most of which don't make much sense for the Gnat byte code interpreter.

Dealing with operands is simpler in a stack-based language such as Forth. In C, a compiler must deal with global vs. local variables, allocation of local variables on the stack, formal vs. actual parameters in function calls, etc. Whereas in Forth, with the exception of globally declared variables, all values are kept on one of the two stacks. This makes discussions of Mobile Code much simpler as well since the majority of Mobile Code concerns can then deal with executable code delivery rather than allocation of variables on the remote host.

In addition to NQC, several other flavors of C and BASIC were investigated (including Interactive C and Structured BASIC). Most either did not have easily available source code or did not lend themselves to timely modification.

We looked at different flavors of Forth before deciding to invent our own, which we named "Gnat Forth". One of these, PBForth, <http://www.hempeldesigngroup.com/lego/pbForth/download.html> is designed for the Lego RCX.

2 GNAT Byte Code Application Programmer's Guide

2.1 Introduction to Forth

One of the best ways get get started programming in Forth on the Gnat platform is to pick up a copy of Leo Brodie's book, "Starting Forth". This book is also available online. Try: <http://home.vianetworks.nl/users/mhx/sf.html> or <http://home.iae.nl/users/mhx/sf.html>.

The rest of this manual will assume that you have access to this book. You do not necessarily have to have read it; just be able to go to it for specific Forth questions.

But for those who want to start off right, the chapters that pertain to using Forth in the Gnat Environment are:

- Starting Forth First Edition, Introductions
- Chapter 1 Fundamental Forth
- Chapter 2 How to Get Results
- Chapter 4 Decision, Decisions, ...
- Chapter 6 Throw it for a Loop
- Chapter 8 Variables, Constants, and Arrays

The book is extremely easy to read and all the examples are well explained. For a good command line version of Forth, which is where we recommend you begin, use Gforth. It is freely available from <http://www.gnu.org/software/gforth/> and compiles on Linux or Mac OS X.

2.2 How to Compile a Forth Program, Download Hex Code to the Gnat, and Run Programs.

The Gnat Forth Byte Code Interpreter was written with the Gnat API version 2.1. For a full discussion of the Gnat API, please refer to the GNATs Developer's Manual at http://borg.cc.gatech.edu/gnats/doc/gnat_manual.pdf

The process begins by writing a Gnat Forth program on a Linux or Mac OS X host. The Gnat Forth source code is passed through the C preprocessor, then through the Gnat Forth byte code "compiler" to produce the byte code hex file. There are no fancy optimizations here; just straight conversion of Forth keywords and numbers to byte code. The compiler outputs a hex file that can be imported into the Microchip IDE window for the EEPROM.

To try this out:

1. Retrieve and expand the Gnat API version 2.1 source distribution from <http://borg.cc.gatech.edu/gnats/#software>. The header files and GnatLib.c are required for the ByteCodeInterpreter. (For those users internal to Georgia Tech, set your CVSROOT to /net/hzr1/users/borg/Software/gnats/CVS and check out the GnatBaseAPI.v2 module.)
2. Retrieve and expand the ByteCodeInterpreter source distribution from the same website. (For those users internal to Georgia Tech, set your CVSROOT to the path above and check out the ByteCodeInterpreter module.)
3. `make` will make `gnatforth`.

```
$ make
cc -o gnatforth gnatforth.c
$
```

4. `make run` will make a new MCH (MicroChip Hex) file.

```
$ make run
cc -x c -P -E Sample.gf > Sample.gf2
./gnatforth Sample.gf2 Sample
GnatForth (16 bit version)
Producing MCH hex output file: Sample.MCH
Dictionary is 107 bytes.
$
```

5. Create a new MPLAB project with `interpret.c` as the C file. See the "The GNATs Developer's Manual" at <http://borg.cc.gatech.edu/gnats/#resources> for instructions on creating new projects and compiling them from within the MPLAB IDE.
6. Compile `interpret.c` within MPLAB.

7. Import the newly compiled MCH file into the EEPROM view window. From the MPLAB IDE View pull down, open the EEPROM window. Right click on the left side of the window and select Import From File. Choose the filename, such as Sample.MCH.
8. Download the program to the Gnat. Again, see the “The GNATs Developer’s Manual” for instructions.
9. Turn on the Gnat and watch it execute the byte code.

2.2.1 Runs on Linux or Mac OS X

The Gnat Forth compiler runs on either Linux or Mac OS X. The included Makefile allows for easy compilation. Executing `make` will compile the included C source code into Gnat Forth byte code compiler called simply `gnatforth`.

2.2.2 gnatforthlib.gf

`gnatforthlib.gf` is an application level library containing Forth definitions for some standard Forth macros and Gnat API Forth implementations. It may also be used as a source for cutting and pasting the needed functions if the entire `gnatforthlib.gf` file is not needed. See the appendix for a full listing of the Forth words contained in `gnatforthlib.gf`.

2.2.3 C-like #defines for constants

Forth allows for constant values to be defined by using the `CONSTANT` keyword. In the Gnat Forth implementation, however, the C preprocessor is used instead. The two preprocessor keywords that have been tested with `gnatforth` are `#define` and `#include`.

The GNU C compiler command for invoking only the C preprocessor on a file called `program.gf` and saving the results into a file called `program.gf2` is:

```
cc -x c -P -E program.gf > program.gf2
```

This is the same command used by the Makefile described below.

2.2.4 GnatForth Makefile

Included in the `ByteCodeInterpreter` distribution is a Makefile:

```
all: gnatforth

debug: gnatforth.c gnatforth.h tokens.h GnatBaseAPI.h GnatBaseAPIConfig.h
      cc -DDEBUG -o gnatforth gnatforth.c

gnatforth: gnatforth.c gnatforth.h tokens.h GnatBaseAPI.h GnatBaseAPIConfig.h
          cc -o gnatforth gnatforth.c
```

```

test: gnatforth gnatforth.c test.gf
      ./gnatforth test.gf test

run:  gnatforth gnatforth.c program.gf
      cc -x c -P -E program.gf > program.gf2
      ./gnatforth program.gf2 program

clean:
      rm gnatforth *.gf2

```

The important lines are the ones for `gnatforth` and `run`. Executing “`make`” will compile the source code into the `gnatforth` executable, which is the Gnat Forth byte code compiler. Executing “`make run`” will run `gnatforth` against the Gnat Forth byte code source file called `program.gf`. You may edit this Makefile and use it to compile your own Gnat Forth byte code source file, or use it as an example of how to invoke the C preprocessor in conjunction with the `gnatforth` compiler.

2.3 Event driven nature of Gnat Forth

One design goal is to be able to program different tasks independently. For instance, the task that monitors the voltage should be independent of the task that propagates path planning messages. They should be able to be programmed by different people with different goals.

All the application tasks we have been able to indentify so far are event based. Possible events might be a button push, a timer event, a reprogramming message, or a path planning message. The way we design programs/tasks should give us some type of event handling abstraction.

We believe the event handler execution should be controlled by some type of guard condition that describes when they should be fired. The task dispatcher/scheduler would be responsible for making sure the event handlers get called when they should. It could enforce some type of prioritization if there is a resource (e.g. time) conflict. For example, if all tasks aren’t schedulable.

So, in a forth dialect:

```

: <HANDLER-NAME> ( <GUARD> )

  <HANDLER LOGIC>
;

```

An example that blinks the red led every hour to let us know the device is alive:

```

: heart-beat-blink ( timer_secs % 3600)
  2 blink_red
;

```

The scheduler basically mimics the interrupt handler on the PIC. There is a list of event handlers that gets polled. If the criteria for an event are TRUE, then the event fires and the Forth word that handles the event is executed.

2.4 Using the example programs

In this section, the user will be introduced to programming in Gnat Forth by using example programs as a tutorial. Sample.gf is the first of the example programs. We will look at it line by line. In subsequent examples we will only look at the newly introduced concepts.

2.4.1 Sample.gf

The first example program we will look at is called Sample.gf. It blinks the red LED 4 times, the green LED 2 times, and repeats. Notice that the C preprocessor keyword `#define` is used for constants. You may also `#include` other .gf (GnatForth) files.

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580

: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;

: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;

: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
  loop
  drop drop drop
;

: <agent0> [ TRUE ] ( -- )

begin
  100 RED_POWER RED_LED 4 display_led
```

```

    100 GREEN_POWER GREEN_LED 2 display_led
false until
;

: <init> ( -- )
  100 GREEN_POWER GREEN_LED 3 display_led
;

END_PROGRAM

```

The LED numbers in the `#defines` are the pin constants used by the CCS C compiler. The power constants are numbers derived by our team and are average brightness values.

New Forth words are treated as subroutines. `: name` signals the beginning of a word definition with `name` being the name of the subroutine. `A ;` ends the word.

The first Forth word definitions we see are `ROT`, short for `ROTATE`, and `OVER`, and are Gnat Forth implementations of the keywords of the same name. Briefly, `ROT` takes the third number down in the stack and *moves* it to the top. The previous top and second number down on the stack are *moved* down to to become the second and third numbers down on the stack, respectively. `OVER` *copies* the second number down on the stack to the top of the stack. See the previously mentioned Forth reference for a full description of all Forth keywords.

`DISPLAY_LED` is a Forth implementation of the Gnat API call `display_led()`. Notice that the arguments for `DISPLAY_LED` are listed in the comment following

```

: DISPLAY_LED

```

and are the delay time, the LED power, the constant for the LED, and the number of times to blink. Remember that the first item listed is at the bottom of the stack. So, the number of times to blink is the value on the top of the stack when `DISPLAY_LED` is called, and, thus, is the first argument for the `DO` loop of the function.

The `DO` loop takes 2 arguments, which are values for the loop limit and the starting index. So,

```

  5 0 DO
    (your code here)
  LOOP

```

executes the code within the `DO` loop 5 times with the loop index taking on values of 0 through 4.

`TURN_ON_LED`, `DELAY_MS`, and `TURN_OFF_LED` are all Gnat API function calls. See the index for the description of their arguments. Their functions are obvious. As an exercise for the reader to practice stack operations, try manually working out the `DISPLAY_LED` function assuming that it was called with this line:

```

100 GREEN_POWER GREEN_LED 2 display_led

```

The next line we see is:

```
: <agent0> [ TRUE ] ( -- )
```

<agent0> is the name of this word. The < and > around the word name signifies to the compiler that this is not a regular word definition but an agent definition. Think of it as a C main() function (where execution begins). Whereas, all the other regular word definitions are like functions that main() calls.

[TRUE] is the activation guard, which is basically a conditional test. The interpreter runs the activation guard piece of code first, and upon completion returns the top of the stack to the main interpreter loop. If the test is TRUE, then the activation guard has passed and the agent code may be activated (executed). If the test returns FALSE, the agent code is not run. In this case, TRUE pushes a TRUE value (1) on the stack. Since this is the only statement in the guard, it will always be TRUE and the agent code will always run. We will see examples below where the agent may not always run.

The stack usage comment is the last thing on this line and shows that the word expects nothing to be on the stack prior to it being called and will leave nothing on the stack after it is finished.

The control loop in <agent0> is the indefinite loop: BEGIN...UNTIL. Code following the BEGIN is executed until the UNTIL word is reached. UNTIL pops the top of the stack and examines it. If the value is FALSE, control passes back to the top of the loop at BEGIN. If the value is TRUE, the loop is exited. Therefore, the code immediately preceding UNTIL is a conditional test. In this case, the keyword FALSE before the UNTIL pushes a FALSE (0) on the stack, so we will never exit the loop.

We can now see that <agent0> is merely an endless loop that blinks the red LED 4 times and then the green LED 2 times.

The last word definition is:

```
: <init> ( -- )
```

<init> is a special agent/function that is only executed when the Gnat is first turned on. In this case it only blinks the green LED 3 times to show that the Gnat is functioning properly. <init> can also be a function to initialize variables as we shall see below.

The final keyword is END_PROGRAM and tells the compiler that the end of the source file has been successfully reached.

Now, compile this program, download it to the Gnat, and try it out!

2.4.2 VarTest.gf

In this program we are going to introduce the VARIABLE keyword and show how to store and retrieve values from the storage reserved by VARIABLE.

Briefly, this example stores the value 10 in a variable and the value 2 in another. It then blinks the red LED the number of times in the first variable and the green LED the number of times in the second variable. The next

iteration increments the first variable by 1 and the second variable by 2 before repeating.

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580

VARIABLE counter-green
VARIABLE counter-red

: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;

: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;

: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
  loop
  drop drop drop
;

: <agent0> [ TRUE ] ( -- )

begin
  200 RED_POWER RED_LED counter-red @ display_led
  200 GREEN_POWER GREEN_LED counter-green @ display_led
  counter-green @ 2 + counter-green !
  counter-red @ 1 + counter-red !
false until

;

: <init> ( -- )
  100 GREEN_POWER GREEN_LED 3 display_led
  2 counter-green !
  10 counter-red !
;

;
```

END_PROGRAM

The word `VARIABLE` reserves a 16 bit storage area that is referred to by the name following `VARIABLE`. In this case, we have 2 such areas: one called `counter-green` and the other called `counter-red`. The storage is reserved from the top (higher addressed) portion of the onchip data EEPROM. Each agent is allowed five 16 bit variables.

A quick scan of the source code also shows 2 more new words:

- `!` is pronounced *store* and is used to store a value into a variable. Note how close to assembly language this is. The syntax for using `!` is

```
value variable-name !
```

- `@` is pronounced *fetch* and is used to retrieve the value stored in a variable and place it on the top of the stack. The syntax for using `@` is

```
variable-name @
```

Before we look at the `<agent0>` code, let's look at `<init>`. `<init>` still blinks the green LED 3 times to let us know the initialization has been successful, but now it also initializes the global variables `counter-green` to 2 and `counter-red` to 10. This follows the syntax for `!` as shown above.

`<agent0>` is also basically the same as in the previous example program. But now, instead of blinking the LEDs for a fixed number of times it fetches the appropriate variable's value for each LED. Note how the line

```
200 RED_POWER RED_LED counter-red @ display_led
```

uses `@` to put the value stored in `counter-red` onto the stack, which is the number of times to blink. Other than this and the delay time being slightly longer, this is the same call as in `Sample.gf`. The same code is repeated for the green LED.

It should not be very hard for the reader to figure out what the next line does.

```
counter-green @ 2 + counter-green !
```

Did you say that the value in `counter-green` was fetched to the stack, 2 was placed on the stack, `+` consumed both values and replaced them with their sum, and this sum was stored into `counter-green`? If so, then great! If not, then the reader should work this out on paper and possibly re-read the section in "Starting Forth" on variables.

When running this program, notice that the green counter is incremented by 2 each iteration and the red counter by only 1.

2.4.3 RAMvarTest.gf

This is the same program as VarTest.gf, but this time using RAM variables. The only lines to have changed are

```
VARIABLE counter-green
VARIABLE counter-red

to

RAM_VARIABLE counter-green
RAM_VARIABLE counter-red
```

The difference is that `RAM_VARIABLE` storage is allocated in the PIC RAM area. This allows for slightly faster access. There are five 16 bit RAM locations allocated for each agent.

2.4.4 RAMlocalVar.gf

Gnat Forth also allows for variables to be declared locally. Each agent is allotted 5 RAM variables and 5 EEPROM variables. Declaring a variable locally within an agent word definition reserves space for the variable in that agent's allotment. This method of reserving space per agent allows that agent's code to be mobile since there is a fixed amount of space pre-allocated in a fixed location. However, it is still global in its "scope". Any Forth word may use it as long as the variable is declared before NOTE: variables declared outside of an agent definition are tied to the first agent.

In the example below, `<agentRed>` declares 3 local RAM variables and `<agentGreen>` declares 2.

This is our first example of a multi-agent source file. It is still quite simple. Notice that all we did was to split the functionality of each LED into its own agent. The `<init>` function stays the same.

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580

: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;

: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;

: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
```

```

    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
loop
drop drop drop
;

: <agentRed> [ TRUE ] ( -- )
RAM_VARIABLE counter-red
RAM_VARIABLE stuff2
RAM_VARIABLE stuff3
200 RED_POWER RED_LED counter-red @ display_led
counter-red @ 1 + counter-red !
;

: <agentGreen> [ TRUE ] ( -- )
RAM_VARIABLE counter-green
RAM_VARIABLE stuff1
200 GREEN_POWER GREEN_LED counter-green @ display_led
counter-green @ 2 + counter-green !
;

: <init> ( -- )
100 GREEN_POWER GREEN_LED 3 display_led
2 counter-green !
10 counter-red !
;

END_PROGRAM

```

LocalVar.gf and MixedVar.gf are both example programs in the same vein as the previous one. LocalVar.gf is exactly the same as RAMlocalVar.gf except the variables are declared as VARIABLE instead of RAM_VARIABLE. MixedVar.gf simply mixes the use of VARIABLE and RAM_VARIABLE. Testing these two programs is left as an exercise for the reader.

2.4.5 battery.gf

This example program demonstrates the usage of the GET_BATTERY_DACVALUE Gnat Forth word and blinks the battery voltage value. GET_BATTERY_DACVALUE is simply a Forth wrapper for the Gnat API function of the same name, and pushes the return value onto the data stack. Only the least significant 15 bits are displayed since the data stack holds signed values. The battery voltage value can be converted into an actual voltage by using the formula in the “GNATS

Developer's Manual".

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580

: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;

: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;

: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
  loop
  drop drop drop
;

: 2DROP ( n1 n2 -- )
  DROP DROP
;

: 2DUP ( n1 n2 -- n1 n2 n1 n2 )
  OVER OVER
;

: DISPLAY_DATA_BYTE ( data_byte -- )
  0x4000
  15 0 do
    2dup and 0 =

    if 1000 RED_POWER RED_LED 1 display_led
      else 1000 GREEN_POWER GREEN_LED 1 display_led
    then

    2 /
  loop
  2drop
```

```

;
: <agent0> [ TRUE ] ( -- )

1000 delay_ms
GET_BATTERY_DACVALUE
display_data_byte

2000 delay_ms
;

: <init> ( -- )
100 GREEN_POWER GREEN_LED 3 display_led
;

END_PROGRAM

```

In addition to the usual 2 stack operation words ROT and OVER, we have 2 new ones: 2DROP and 2DUP. 2DROP drops the top 2 elements off the data stack, as opposed to DROP which only drops the top element. 2DUP duplicates the top 2 elements of the data stack, as opposed to DUP which only duplicates the top element.

The version of DISPLAY_DATA_BYTE here will blink green (for a 1 value) or red (for a 0 value) for the least significant 15 bits of the value passed in. The value 0x4000 is the initial value of the bit mask that is used to mask out all but the desired bit. After each iteration it is divided by 2, and thus is equivalent to a right bit shift. Since the DO loop executes 15 times we see all the least significant 15 bits.

Another new word we need to look at is the IF statement. The line:

```
2dup and 0 =
```

is the conditional statement the result of which the IF word consumes. The format of an IF statement is:

```

condition IF true condition statements
           ELSE false condition statements
           THEN

```

So, in this case if the bit tested is 0 then we have a TRUE condition, and will execute the code between the IF and the ELSE. If the bit is a 1, we will execute the code between the ELSE and the THEN. Here is the relevant code from the example:

```

if 1000 RED_POWER RED_LED 1 display_led
else 1000 GREEN_POWER GREEN_LED 1 display_led
then

```

DISPLAY_DATA_BYTE leaves the data stack clean by dropping both values it has been using: the data value passed in and the bit mask value pushed on the stack at the top of the function.

The <agent0> code should be very obvious in function. It delays for 1000 ms, runs GET_BATTERY_DACVALUE, displays the data received, and delays again.

The <init> function is the same as the previous ones.

2.4.6 thermal-sensor.gf

Reads the thermal sensor and blinks its value. This code is nearly identical to battery.gf. The difference is that now we are running the READ_THERMAL_SENSOR function, and DISPLAY_DATA_BYTE is not a separate function. It is contained within the agent code. Also, the bit mask is contained in a RAM_VARIABLE called initial_value, not on the stack.

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580

RAM_VARIABLE initial_value

: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;

: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;

: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
  loop
  drop drop drop
;

: <agent0> [ TRUE ] ( -- )

1000 delay_ms
read_thermal_sensor
```

```

8 0 do
  dup initial_value @ and 0 =

  if 1000 RED_POWER RED_LED 1 display_led
    else 1000 GREEN_POWER GREEN_LED 1 display_led
  then

  initial_value @ 2 / initial_value !
loop

drop
0x80 initial_value !

2000 delay_ms
;

: <init> ( -- )
  0x80 initial_value !
  100 GREEN_POWER GREEN_LED 3 display_led
;

END_PROGRAM

```

The reader should verify that this program works as it should. The 8 bit value displayed is the degrees Celsius in binary.

2.4.7 SendReceive.gf

This example program demonstrates passing a counter variable between Gnats. Each button press on a sending Gnat increments the counter and transmits it over the IR emitter. The receiver blinks the counter value.

Before we look at the example code, we need to know how the Gnat environment's communication buffers function. In most other Forth implementations, a program would push all data onto the data stack, and then call the communication code to consume that data and send it. This is where the Gnat environment must break with the standard. In the Gnat environment, we are drastically limited in memory. In order to prevent data stack overflow when using communication functions, separate communication buffers are used, and new Forth words are introduced to populate the buffers.

The Gnat API defines a single packet receive communication buffer. The interpreter defines a second buffer for sending packets. The new words introduced are:

- `!C` means "Store Communications" and is used for storing data in the Gnat send buffer. Example usage:

```
variable1 @ 0 !C      ( add generation to send buffer )
```

The above line puts the variable1 value on the stack and the communication send buffer offset of 0 on the stack. !C stores the variable1 value at offset 0 in the communication send buffer.

- @C means “Fetch Communications” and is used for retrieving data from the Gnat receive buffer. Example usage:

```
0 @C variable1 !      ( get value into generation var )
```

The above line begins by putting the offset value of 0 on the stack. @C consumes this value, and puts the receive communication buffer value at that offset onto the stack. The rest of the line is a normal store function (stores the value into variable1’s storage location).

The !C word consumes 2 values from the data stack: send_communication_buffer_offset and data_value with the send_communication_buffer_offset on the top of the stack. SEND_CAPSULE should be one of the next calls, and is the words that sends the packet.

The @C word consumes 1 value from the data stack: receive_communication_buffer_offset. GET_PACKET must have been called previously to make the data in the buffer available.

We’ll see these 2 new words used in the code below. Also notice that we’ve defined 3 new constants: IROUT_PWR_LOW, IR_OUT_ON_ALL, and APPLICATION_MESSAGE.

```
#define GREEN_LED      43
#define GREEN_POWER    0x0380
#define RED_LED        42
#define RED_POWER      0x0580
#define IROUT_PWR_LOW  0x0C80
#define IR_OUT_ON_ALL  0x0F
#define APPLICATION_MESSAGE 0x00
```

```
RAM_VARIABLE generation
```

```
: ROT ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP R> SWAP
;
```

```
: OVER ( n1 n2 -- n1 n2 n1 )
  >R DUP R> SWAP
;
```

```
: DISPLAY_LED ( delay_time led_power led number_times_to_blink -- )
  0 do
```

```

    over over TURN_ON_LED
    rot dup DELAY_MS
    rot rot dup TURN_OFF_LED
    rot dup DELAY_MS
    rot rot
loop
drop drop drop
;
: DISPLAY_COUNT ( count -- )
dup 125 swap / swap
0 do
    GREEN_POWER GREEN_LED turn_on_led
    dup
    0 do
        3 delay_ms
    loop

    GREEN_LED turn_off_led
    dup
    0 do
        3 delay_ms
    loop

loop
drop
;

: <IREvent> [ GET_RECV_STATUS APPLICATION_MESSAGE = ] ( -- )
1 get_capsule
0 @C generation !      ( get value into generation var )
generation @ display_count ( display the generation )
;

: <ButtonEvent> [ button_pressed? TRUE = ] ( -- )
generation @ 1 + generation !      ( increment the generation )
generation @ 0 !C      ( add generation to send buffer )
1 IROUT_PWR_LOW IR_OUT_ON_ALL send_capsule ( send the capsule )
75 RED_POWER RED_LED 3 display_led      ( red to show sent )
;

: <init> ( -- )
0 generation !
100 GREEN_POWER GREEN_LED 3 display_led
;

END_PROGRAM

```

In this example, we have 2 agents: `<IREvent>` which is responsible for processing packets if one has arrived, and `<ButtonEvent>` which waits for the button to be pressed to perform its action.

For `<IREvent>`, notice that the activation guard is now a real test. It calls `GET_RECV_STATUS` which places the current status of the receive queue on the stack. If there is a packet ready to be processed this value is `APPLICATION_MESSAGE` (0); if not, this value is -1. It then pushes `APPLICATION_MESSAGE` onto the stack, and checks for the equality of the two values. So, if the `GET_RECV_STATUS` value is `APPLICATION_MESSAGE`, then the activation guard has passed, and we can execute the agent code.

The first thing `<IREvent>` does is call `GET_CAPSULE` which consumes the value on the stack (in this case 1) and copies that number of received items to the receive buffer. It also resets the receive flags in the interpreter. Next, we want to get the element we just copied onto the stack and then into the variable called `generation`.

```
0 @C generation !      ( get value into generation var )
```

The discussion above on the communication buffers should help the reader decode the line. The last thing we want to do is to display the `generation` count. `DISPLAY_COUNT` will display a variable number in a fixed amount of time. See the “GNATs Developer’s Manual” for a description of the API call `display_count()`.

The activation guard for `<ButtonEvent>` checks to see if the button has been pressed. `BUTTON_PRESSED?` returns `TRUE` if the button has been pressed. When the button is pressed we want to increment the current `generation` value:

```
generation @ 1 + generation !      ( increment the generation )
```

Next, we want to put the `generation` value into the send buffer. `!C` wants the value to be directly below the buffer offset on the data stack:

```
generation @ 0 !C      ( add generation to send buffer )
```

`SEND_CAPSULE` is a Forth wrapper of a Gnat API function. The “1” in the line below is the number of values to send. The other values consumed by `SEND_CAPSULE` are obvious:

```
1 IROUT_PWR_LOW IR_OUT_ON_ALL send_capsule      ( send the capsule )
```

Lastly, blink 3 times to show the packet has been sent:

```
75 RED_POWER RED_LED 3 display_led      ( red to show sent )
```

`<init>` initializes `generation` to 0.

3 GNAT Byte Code System Programmer's Guide

This section assumes that the reader has a basic familiarity with Forth and the Gnat Forth environment. Please read the previous section if you are not comfortable with either of these.

3.1 Gnat Forth Environment Overview

The Gnat environment consists of two pieces: a Gnat Forth compiler that runs on the Linux or Mac host, and the Byte Code interpreter that runs on the Gnat hardware and its source code is manipulated from within the MPLAB IDE.

The Gnat Forth compiler uses 3 distribution source files: `tokens.h`, `gnatforth.h` and `gnatforth.c`. The Byte Code interpreter uses 2 source files from this distribution: `gnatforth.h` and `interpret.c`, and also the source files from the Gnat API version 2.1 distribution. Both distributions are available from the Gnat website.

There is also a Makefile in the Gnat Forth Byte Code Interpreter distribution. Its function was described in the previous section. It is used to build the compiler and run it against Gnat Forth (`.gf`) source files to produce MicroChip Hex (`.MCH`) files. The `.MCH` file is downloaded into the Gnat via the MPLAB IDE. This procedure is also discussed in the previous section.

One last piece of the Gnat Forth environment is the C preprocessor. In this case, the GNU C preprocessor. This is used so that the user has the ability to use `#define` and `#include` in the Gnat Forth source programs. This saves the system from having to implement the standard Forth word "CONSTANT". The command line for this is included in the Makefile, but for completeness it is also shown here:

```
cc -x c -P -E Sample.gf > Sample.gf2
```

The original source file is `Sample.gf` and the post processed file is `Sample.gf2`.

3.2 Gnat Forth Compiler Overview

One of the things that makes Forth easy to "compile" is that it has a very simple syntax, so building your own interpreter isn't too hard. Tokens are separated by white space, so no fancy state machines are needed. But, you do need a symbol table for words (just Forth-speak for subroutines) and variables.

New Forth words are treated as subroutines. `:` `name` signals the beginning of a word definition with `name` being the name of the subroutine. `A` `;` ends the word. The source program calls the word by name; the compiler outputs a `CALL_MACRO` bytecode with the dictionary address. The `;` is replaced by a `RETURN` bytecode.

The name of the word: `<IREvent>` is special to the compiler because it is an event handler/agent. The compiler writes the address of the word into the dictionary where the pointer to this event type is held.

The `CALL_MACRO` bytecode in the interpreter on the Gnat pushes the `program_counter + 2` (the return address) onto the return stack then jumps to the address of the word given after the `CALL_MACRO` bytecode. Obviously, the `RETURN` pops the return address of the stack and jumps to that address to continue.

The `BEGIN...UNTIL` is the Forth indefinite loop (like a C “do-while” loop). `BEGIN` starts the loop. `UNTIL` pops the stack, and on a `FALSE` condition returns to the `BEGIN` statement and on a `TRUE` condition lets control continue to the instruction following the `UNTIL`. The `BEGIN...UNTIL` loop in the Gnat Forth implementation is fully nestable.

The `DO...LOOP` is the Forth definite loop (like a C “for” loop). `DO` expects a limit and a start index (the loop counter) to be on the stack before it is called. `LOOP` increments the loop counter and checks to see if it is equal to the loop limit. If it is, then the loop is exited, otherwise control is passed back to the top of the loop at the `DO` statement. The `DO...LOOP` loop in the Gnat Forth implementation is fully nestable.

We use a registry at the beginning of the dictionary to keep track of event handlers/agents and meta-information about them. The registry is defined in `gnatforth.h`. The dictionary is laid out like this:

- number of handlers at byte 0
- agent0 registry at next 9 bytes
- agent1 registry at next 9 bytes
- agent2 registry at next 9 bytes
- agent3 registry at next 9 bytes
- init code pointer at byte 37
- first dictionary addr at byte 39

The individual registry entries are also define in `gnatforth.h`:

```
#define REGISTRY_ENTRY_LENGTH          9

#define REGISTRY_AGENT_ID_OFFSET      0
#define REGISTRY_AGENT_VER_OFFSET     1
#define REGISTRY_AGENT_AGUARD_ADDR_OFFSET 2
#define REGISTRY_AGENT_EGUARD_ADDR_OFFSET 4
#define REGISTRY_AGENT_LENGTH_OFFSET  6
#define REGISTRY_AGENT_ADDR_OFFSET    7
```

The `main()` function in the interpreter runs each of the agent’s activation guards in turn. These are just pieces of Forth code that evaluate the state of the Gnat. If the guard returns `TRUE`, then the agent code itself is run. Here’s an excerpt from `main()` on the interpreter:

```

while (TRUE) {
    // Execute the Activation Gate first before calling the application
    // level Forth handler/agent.
    for (current_handler_number = 0;
        current_handler_number < MAX_HANDLERS; current_handler_number++) {

        current_handler_address = get_16bit_bytecode(current_handler_number *
            REGISTRY_ENTRY_LENGTH + REGISTRY_AGENT_ADDR_OFFSET + 1);

        // if this is a valid handler/agent run the activation guard.
        if (current_handler_address != 0x0000) {

            // if the activation guard returns true run the agent itself.
            if (execute_interpreted_program(get_16bit_bytecode(
                current_handler_number * REGISTRY_ENTRY_LENGTH +
                REGISTRY_AGENT_AGUARD_ADDR_OFFSET + 1)) == TRUE) {
                execute_interpreted_program(current_handler_address);
            }
        }
    }
}

```

3.3 The Gnat Forth Communication Environment

In most other Forth implementations, a program would push all data onto the data stack, and then call the communication code to consume that data and send it. This is where the Gnat environment must break with the standard. In the Gnat environment, we are drastically limited in memory. In order to prevent data stack overflow when using communication functions, separate communication buffers are used, and new Forth words are introduced to populate the buffers.

The Gnat API defines a single packet receive communication buffer. The interpreter defines a second buffer for sending packets. The new words introduced are:

- **!C** means “Store Communications” and is used for storing data in the Gnat send buffer. Example usage:

```
variable1 @ 0 !C      ( add generation to send buffer )
```

The above line puts the variable1 value on the stack and the communication send buffer offset of 0 on the stack. !C stores the variable1 value at offset 0 in the communication send buffer.

- **@C** means “Fetch Communications” and is used for retrieving data from the Gnat receive buffer. Example usage:

```
0 @C variable1 !      ( get value into generation var )
```

The above line begins by putting the offset value of 0 on the stack. @C consumes this value, and puts the receive communication buffer value at that offset onto the stack. The rest of the line is a normal store function (stores the value into variable1's storage location).

The !C word consumes 2 values from the data stack: `send_communication_buffer_offset` and `data_value` with the `send_communication_buffer_offset` on the top of the stack. `SEND_CAPSULE` should be one of the next calls, and is the words that sends the packet.

The @C word consumes 1 value from the data stack: `receive_communication_buffer_offset`. `GET_PACKET` must have been called previously to make the data in the buffer available.

Here is the interpreter code for !C and @C. `scratch` and `scratch2` are 16 bit variables. `send_pkt_buf.data` is the 16 byte data buffer in the send packet structure. `recv_data` is the API recv buffer array.

```
// Special store instruction for storing a value into
// the send buffer.
else if (instruction == COMM_STORE) {
    scratch = pop_data(); // data buffer offset
    scratch2 = pop_data(); // data value
    send_pkt_buf.data[scratch*2] = scratch2_lowbyte;
    send_pkt_buf.data[(scratch*2)+1] = scratch2_highbyte;
}

// Special fetch instruction for fetching a value from
// the send buffer.
else if (instruction == COMM_FETCH) {
    scratch = pop_data(); // data buffer offset
    scratch2_lowbyte = recv_data[scratch*2];
    scratch2_highbyte = recv_data[(scratch*2)+1];
    push_data(scratch2);
}
```

3.4 How to add/modify/delete a Gnat Forth token

A good place to start when looking at the source code for the Gnat Forth environment is dealing with tokens. There are 4 source files that need to be modified to add, modify, or delete a token. Here is the list with a brief description of what each file is responsible for.

1. `tokens.h` pairs token strings to token mnemonics.
2. `gnatforth.h` assigns byte codes to token mnemonics.
3. `gnatforth.c` is a very simple “compiler” and runs on the Linux or Mac host.

- (a) Basically just converts token strings to byte codes.
 - (b) Allocates memory for declared variables.
 - (c) Inserts `call_macro`, `return`, `if/then/else`, and looping parameters and pointers.
 - (d) Assumes everything else is a byte code that the interpreter on the Gnat will decode and run.
4. `interpret.c` runs on the Gnat, decodes byte codes, and runs the appropriate API, control structure, arithmetic, or stack manipulation code.

The next subsections build upon each other, so please read them all. It would be a good exercise for the reader to have a copy of the source files available while reading.

3.4.1 Deleting a Gnat Forth token

1. Edit `tokens.h`:
 - (a) Add an entry to the revision history to let others know which token you are deleting.
 - (b) Find the token (probably by its string) in the list and either comment it out or delete it.
2. Edit `gnatforth.h`:
 - (a) Add an entry to the revision history to let others know which token you are deleting.
 - (b) Find the token by its mnemonic in the `#define` list and comment it out or delete it.
3. Edit `gnatforth.c`:
 - (a) In the `process_token()` function, find the token by its mnemonic. It will only have an entry if there is a special operation the compiler needs to perform when processing the token. If it is there, comment it out or delete it.
 - (b) If a change to the file was made add an entry to the revision history to let others know which token you are deleting.
4. Edit `interpret.c`:
 - (a) Add an entry to the revision history to let others know which token you are deleting.
 - (b) In the `execute_interpreted_program()` function, find the token by its mnemonic. Comment it out or delete it.

3.4.2 Modifying a Gnat Forth token

1. If you are modifying the token string or mnemonic, edit `tokens.h`:
 - (a) Add an entry to the revision history to let others know which token you are modifying.
 - (b) Find the token (probably by its string) in the list, and perform your modification.
2. If you are modifying the mnemonic or byte code, edit `gnatforth.h`:
 - (a) Add an entry to the revision history to let others know which token you are modifying.
 - (b) Find the token by its mnemonic in the `#define` list, and perform your modification.
3. If you need to modify the way the compiler processes this token, edit `gnatforth.c`:
 - (a) In the `process_token()` function, find the token by its mnemonic. It will only have an entry if there is a special operation the compiler needs to perform when processing the token. Perform your modification.
 - (b) If a change to the file was made add an entry to the revision history to let others know which token you are modifying.
4. If you need to modify the way the interpreter on the Gnat deals with the byte code, edit `interpret.c`:
 - (a) Add an entry to the revision history to let others know which token you are modifying.
 - (b) In the `execute_interpreted_program()` function, find the token by its mnemonic, and perform your modification.

3.4.3 Adding a Gnat Forth token

1. Edit `tokens.h`:
 - (a) Add an entry to the revision history to let others know which token you are adding.
 - (b) Add your token to the bottom of the list before the NULL terminators. The left column is the double-quote enclosed token string that the user will type into the source code. The right column is the token mnemonic.
2. Edit `gnatforth.h`:
 - (a) Add an entry to the revision history to let others know which token you are adding.

- (b) Add your `#define` statement to the end of the list before the

```
#define COMMENT    0xFA
```

line. Be sure the byte code you assign is unused.

3. Edit `gnatforth.c`:

- (a) If you need to have the compiler do special processing for this new token, add an “else if” to the `process_token()` function before these lines:

```
// Lastly, if the token doesn't match any of the above, just
// output its bytecode and let the interpreter deal with it.
else {
    write_bytecode_to_dictionary(current_dict_loc, token);
    current_dict_loc++;
}
```

Then add your code.

- (b) If a change to the file was made add an entry to the revision history to let others know which token you are adding.

4. Edit `interpret.c`:

- (a) Add an entry to the revision history to let others know which token you are adding.
- (b) In the `execute_interpreted_program()` function, add an “else if” before these lines:

```
// gotta get around the PUSH constant common code...
goto COMMON;
```

Then add your code.

3.5 `gnatforth.c` - the Gnat Byte Code Compiler

This is the source file for the GNAT specific Forth implementation. We will step through this file.

There is a stack used for storing control statement addresses. This makes it possible to nest control statements such as `if/then/else`, `do/loop`, etc. This `define` controls the size of that stack.

```
#define STACK_SIZE    32 // stack for nested control statements
```

The total possible agents is defined:

```
#define NUM_AGENTS    4 // number of mobile code agents
```

The symbol table stores more than just the string that represents the name of the symbol and its address. It also has space to store the guard addresses and the agent identifier and version. This information is used when writing the registry entry for the particular agent. The other fields are described in the comments below.

```
typedef struct symbol_table {
    char *name;
    int address;                // 16 bit absolute address for agents,
                                // offsets for variables
    int activation_guard_address; // 16 bit absolute address
    int existence_guard_address;  // 16 bit absolute address
    int length;                  // total length of agent code used for
                                // mobile code transmission

    int agent_identifier;
    int agent_version;
    symbol_type_t type;          // one of the enum'd values above
} symbol_table_t;
```

Each agent is allowed five 16 bit variables in RAM and five 16 bit variables in onchip EEPROM. The source program declares a variable to use RAM space with the `RAM_VARIABLE` keyword. It declares a variable to use onchip EEPROM with the `VARIABLE` keyword. See the previous section for an application level discussion on these keywords.

After opening the appropriate files the compiler prints the information banner.

```
GnatForth (16 bit version)
```

If the `DEBUG` flag is set then

```
!!!DEBUG flag set!!!
```

also prints. The `DEBUG` flag is helpful in debugging the output hex file. It prints the hex address of the byte code next to the byte code itself. Lastly, the information banner prints:

```
Producing MCH hex output file: outfile_name
```

After processing, a message lets the user know the size of the resulting dictionary.

```
Dictionary is %d bytes.
```

Obviously, the compiler lets the user know if the dictionary is too large.

```
if (current_dict_loc > DICTIONARY_END_ADDRESS) {
    fprintf(stderr, "Error: dictionary too large\n");
    exit(1);
}
```

No matter the size of the dictionary, the compiler must fill up the remaining locations with `0xFF`.

3.5.1 `init()`

`init()` allocates memory for the dictionary and initializes it. It is better to malloc memory than to statically allocate it at compile time. The dictionary will be the output file in MicroChip hex format (2 hex digits per line for 256 lines).

3.5.2 `get_token()`

This function performs 2 things:

1. gets the next token into the `token_buffer` array, and
2. calls `get_token_type()` to get the type of token. This is the return value.

The nice thing about Forth (or at least the Gnat Forth implementation) is that all tokens are separated by whitespace. `#` is a comment character, so the compiler ignores this text to the end of the line.

`token_buffer` holds the token string, and the compiler converts all token strings to uppercase. The last step of this function is to get the token type and return it.

3.5.3 `get_token_type()`

Tokens can be one of these types:

1. within a comment (set or reset the `in_comment` flag),
2. a decimal or hexadecimal digit,
3. a keyword (get the specific token mnemonic for this keyword from the `token_table` in `tokens.h`), or
4. an identifier (variable or forth function name).

3.5.4 `process_token()`

For each token type there are different actions or outputs.

For a comment, do nothing.

For a decimal or hexadecimal digit, output:

- a push bytecode, and
- the hex version of the digit.

If it is an EEPROM variable definition, create a symbol table entry and assign it an address offset based on the agent number using the `get_next_onchip_eeprom_location()` function.

If it is a RAM variable definition, create a symbol table entry and assign it an address offset based on the agent number using the `get_next_ram_location()` function.

If the token is the beginning of a guard definition, set the appropriate flag (an activation guard is required and an existence guard is optional), and set the guard starting address.

When ending a guard definition, we want to update where the actual agent code begins. The agent code begins right after the last guard.

For beginning the definition of a new word, if this is the initialization word, write its address in the proper place in the dictionary. Otherwise, if it is an agent we need to write the proper fields into the registry at this point:

- write the agent ID to the registry for this agent
- write the agent version to the registry for this agent
- write the a_guard address to the registry for this agent
- write the e_guard address to the registry for this agent
- write the agent length to the registry for this agent
- write the agent address to the registry for this agent

NOTE: “agents” and “event handlers” are the same thing. If it’s just a regular Forth macro, we just want to output a RETURN bytecode.

For the special Forth word, FALSE, output:

- PUSH bytecode and
- 0 value.

For the special Forth word, TRUE, output:

- PUSH bytecode and
- 1 value.

There are several outputs for an IF statement:

- the IF bytecode,
- a blank 8 bit ELSE offset, and
- a blank 8 bit THEN offset

Offsets are used instead of absolute address to make mobile code possible. This way an agent can live in any location in memory. Since the offsets are only 8 bits, be sure to have the entire IF statement within 256 locations. The offsets blanks are filled in by the ELSE and THEN statements below. NOTE: a Forth IF looks like one of:

```
condition-statements IF true-statements THEN
condition-statements IF true-statements ELSE false-statements THEN
```

The THEN offset is used as a jump location if the condition is false and there is no ELSE. ELSE is used as a jump location if the condition is false so that those statements can be executed.

The ELSE word is optional. But if it is used, populate the “else” address in the “if” location. The nesting stack is used because IFs can be nested.

The THEN word doesn’t produce a bytecode. It is just a “jump to” location. It performs the following:

- populate the “then” offset in the “if” location,
- populate the “then” offset in the “else” location, if there is one.

When an unknown token string is encountered, we have to check to see if it is an identifier. Identifiers must have already been defined since this is a one pass “compiler”. If the identifier is a subroutine/word (i.e., we’re calling a subroutine/word), output a CALL_MACRO bytecode, and then output the address of the subroutine. If it is a variable, output a PUSH, and then output the offset of the variable. This has the effect of pushing the address of the variable onto the data stack.

Lastly, if the token doesn’t match any of the above, just output its bytecode and let the interpreter deal with it.

3.5.5 add_new_symbol()

Adds a new symbol table entry, fills in default values, and returns a pointer to the new entry. The calling function fills in the specifics.

3.5.6 get_symbol_attributes()

Returns the symbol table entry that matches the passed in symbol name.

3.5.7 get_next_ram_location()

Finds the next available RAM location for this agent and returns that offset. Each agent gets five 16 bit locations.

3.5.8 get_next_onchip_eeprom_location()

Finds the next available EEPROM location for this agent and returns that offset. Each agent gets five 16 bit locations.

3.5.9 write_bytecode_to_dictionary()

Writes an 8 bit value to the dictionary.

3.5.10 write_value_to_dictionary()

Writes a 16 bit value to the dictionary.

3.6 gnatforth.h

Header file for the GNAT specific Forth implementation. It contains:

- Defines for dictionary layout. The dictionary is laid out like this:
 - number of handlers at byte 0
 - agent0 registry at next 9 bytes
 - agent1 registry at next 9 bytes
 - agent2 registry at next 9 bytes
 - agent3 registry at next 9 bytes
 - init code pointer at byte 37
 - 1st dictionary addr at byte 39
- Defines for the registry. The registry is used when transmitting agent code by the mobile code transfer operation. The registry lives at the beginning of the dictionary. There is space available there for 4 agents, and the init code pointer follows the registry. Dictionary definitions follow the init pointer.
- Defines for the language in terms of byte codes. Token identifiers are assigned bytecodes here. These identifiers are associated with keyword strings in tokens.h.

3.7 tokens.h

In this table the strings that are used in the forth source programs are on the left. Their associated token identifiers are on the right. Token identifiers are assigned bytecodes in gnatforth.h. This list is null terminated and is `#include'd` by gnatforth.c.

3.8 interpret.c - the Gnat Byte Code Interpreter

This program expects as input byte code that has been compiled from Gnat Forth using the Gnat Forth compiler.

3.8.1 Program Constants

The data stack holds twelve 16 bit values and is signed. The return stack holds ten 16 bits values and is unsigned.

```
#define DATA_STACK_SIZE    12    // 12 signed 16 bit values
#define RETURN_STACK_SIZE   10    // 10 unsigned 16 bit values
```

In order to reduce the size of the generated assembly code in the PIC environment, it is very important to have large functions and their children reside in the same program memory segment. `execute_interpreted_program()` only fits

in the upper program memory segment, so we need to be sure its most frequently used child functions are in the upper program memory segment too. Its most frequently called children are the “stack operation” and “get bytecode” functions. The following addresses are where we want to have those children located.

```
#define BEGIN_INTERPRET_CODE 0x0F30
#define END_INTERPRET_CODE   0x0FFF
```

3.8.2 Global variable definitions

- `recv_data` shares allocated space with the receive buffer.
- The receive queue is not used for reasons of program code space.
- `current_handler_number` holds the id of the currently executing handler/agent.
- The agent RAM variable allocations = 40 bytes total. Each agent can allocate 5 16 bit variables with the keyword `RAM_VARIABLE`. Offsets are used when calculating the absolute address of the RAM variable.

3.8.3 The data and return stacks - global

The data and return stacks *cannot* reside in a single array space and grow together because the data stack is signed and the return stack is unsigned.

3.8.4 `execute_interpreted_program()`

This function is the main execution routine. It decodes the byte codes, performs the needed function, and returns the top of the data stack. This return value is needed because the main function must know the result of executing the guards. A guard is basically a conditional which if it evaluates to TRUE means the guard has been passed. Execution begins at the location passed in by `main()`, which is `program_counter`. And execution continues until the end of program bytecode: `0xFF`.

For aesthetic reasons it would be nicer to have had the decode operation in a switch statement, but a giant if-else-if structure has a smaller program memory footprint.

Not every byte code will be covered here. Only the ones where it might be unclear as to how the dictionary is used. For the Forth wrappers of Gnat API calls, please reference the “GNATs Developer’s Manual”.

Standard Forth “DO” loop. Pops the start and limit indexes off the data stack, and pushes the DO instruction address and the start and limit indexes onto the return stack. This allows nested DO loops.

End of the “DO” loop. “ENDDO_PLUS” is the LOOP+ instruction. Check to see if the start index has reached the limit. If not, then push both indexes back onto the data stack and set the `program_counter` to the DO address. If the

limit has been reached, pop the DO address from the return stack and continue to the next instruction.

“J” traditionally grabs the 3rd item down in the return stack, but since the interpreter also pushes the address of the DO onto the return stack, “J” must return the 4th item down.

“I” just copies the DO loop index onto the data stack.

Standard Forth “IF”. The offset for jumping to the THEN location is stored in the 8 bit location just after the IF bytecode. The offset for jumping to the ELSE location is stored in the 8 bit location just after the THEN offset. Evaluate the top of the data stack and do the appropriate jump based on the TRUE or FALSE value.

```
// Standard Forth IF.
// The offset for jumping to the THEN location is stored in the
// 8 bit location just after the IF bytecode.
// The offset for jumping to the ELSE location is stored in the
// 8 bit location just after the THEN offset.
// Evaluate the top of the data stack and do the appropriate
// jump based on the TRUE or FALSE value.
else if (instruction == IF) {
    scratch = get_8bit_bytecode(program_counter + 1); // "then" offset
    scratch2 = get_8bit_bytecode(program_counter + 2); // "else" offset

    // If condition before the IF is true then fall through to
    // the next instruction. If the condition is false then
    // perform this logic.
    if (pop_data() == 0) {

        // If there isn't an else, jump just past "then" offset
        if (scratch2 == I_NOP) {
            program_counter += scratch;
        }

        // If there is an else, jump just past "else" offset
        else {
            program_counter += scratch2;
        }
        continue;
    }
    program_counter += 2;
}
```

The offsets are used in the manner described in the application level section above.

If we reach the ELSE instruction, we just finished executing the TRUE condition code (immediately after the IF) and we now need to jump to the

THEN location. The offset of THEN is in the 8 bit location just after the ELSE.

For storing a value into a variable we need to check to see whether it is a RAM or a onchip EEPROM variable. If the address is less than 512 it is a RAM variable. First, get the address, then get the value from the stack. The tricky part is remembering that the address in the onchip data EEPROM range is stored as the MSB is in the lower addressed position and the LSB is in the higher addressed position. For retrieving a value from a variable the process is much the same.

For communications, a special buffer is defined for storing values before they are send and as they are retrieved. The data stack could be used for this, but it would mean a larger stack. The Gnat-specific instructions “!C”: communication store and “@C”: communication fetch have been introduced. See SendReceive.gf for an example of how to use these instructions.

3.8.5 main()

The main function performs:

1. initialize the PIC,
2. copy the onchip EEPROM to *i²c* EEPROM (for now),
3. initialize the Forth dictionary,
4. run each agent’s activation guard, if the guard returns TRUE run the agent code, and
5. repeat 4).

The section of code that copies the onchip EEPROM to the *i²c* EEPROM can go away as soon as we have a method of programming the *i²c* EEPROM directly from either a bootloader or an application. These lines add 4% usage of program memory.

Since `execute_interpreted_program()` lives in the upper program memory block, so should the following functions. Use the following code (with the previously discussed `#defines`) to do this:

```
#ORG BEGIN_INTERPRET_CODE, END_INTERPRET_CODE default
```

3.8.6 get_8bit_bytecode()

Returns the 8 bit bytecode value at the location passed into this function. The return value will be a byte code. Included is code for fetching from both onchip EEPROM and offchip EEPROM.

3.8.7 `get_16bit_bytecode()`

Returns the 16 bit bytecode value at the location passed into this function. The code fixes the byte order for 16 bit values. The MSB is in the lower addressed position, and the LSB is in the higher addressed position. Included is code for fetching from both onchip EEPROM and offchip EEPROM.

3.8.8 `push_data()` and `push_ret()`

There are separate `push_data()` and `push_ret()` routines because this produces less code than a single function with a condition that determines which stack is to be used. These functions push element onto their respective stacks. No return value. Also no error checking to save program memory space.

3.8.9 `pop_data()` and `pop_ret()`

There are separate `pop_data()` and `pop_ret()` routines because this produces less code than a single function with a condition that determines which stack is to be used. These functions return the top element from their respective stacks. Also no error checking to save program memory space.

3.9 Still Todo

1. List of yet to be implemented Forth keywords:
 - CREATE and ALLOT arrays (and structures)
2. Other language specific todos
 - West Nile-like bytecode transfer - Mobile code
 - hooks for a command line interpreter
 - port hop count propagation code (GnatsSU) – TOO LARGE without offchip memory
 - Since we have full control over the interpreter's program counter we can load the interpreter on top of a scheduler that is able to be fully preemptive, i.e. the scheduler loads up the initial value of the interpreter's program counter and then runs the interpreter until the time slice is over.
3. List of API functions and their status with the interpreter
 - void `set_cpu_speed(CPU_SPEED speed)`; DONE - needs work - NOT A PRIORITY. Need a way to fix the `delay_ms()` and `delay_us()` built in functions.
 - bool `wait_for_channel(uint16 wait_cycles)`; TO DO. Interpreter does this, use: `#ifdef NON_BLOCKING_SEND` or `#ifdef BLOCKING_SEND`

4 Mobile Agent Code

4.1 Introduction to Mobile Code

In environments where we could possibly have thousands of Gnats deployed, possibly performing different tasks, a mobile code system makes sense. The code should probably propagate at the level of event handlers/agents. For instance, we could introduce some kind of heartbeat functionality to one Gnat and it propagates itself to other Gnats. In fact, we could attach a propagation policy with each handler/agent similar to the activation guard describing how the agent code should be propagated.

We add a new guard to follow the activation guard in agent code:

```
: <HANDLER-NAME> ( <ACTIVATION-GUARD>, <PROPAGATION-GUARD> )  
  
    <HANDLER LOGIC>  
;
```

An agent can be injected into the Gnat network and can infect the entire network. This discussion is concerned with transmitting individual agent code only; not the entire contents of the data EEPROM dictionary.

Another exciting possibility is that if a Gnat decides it doesn't have the resources (e.g.'s, the time, space, particular sensor, or nearby robots) to run an agent, it can instead forward it to nearby Gnats that might be able to run it. For instance, if the Gnats were localized, the code could live in every square 4 meters of the building, or only on the third floor, etc.

Moving code is not difficult since the state is completely determined by the code (dictionary) and the stack. No memory locations, such as global variables, are required to migrate. Any variables that are used by an agent must be declared as local variables. This ensures that the memory on the receiving Gnat has been allocated. Each agent has a pre-allocated memory heap, so if there is a slot available on a Gnat for another agent then it uses the pre-allocated memory space for that slot.

References for this section include papers on: Mate [1], Agilla [2], and Active Networks [3].

4.2 Mobile Code Protocol

We have investigated a WestNile-like protocol for mobile agent propagation. We use the agent ID and version to make mobile code propagation manageable on the Gnat platform. For example, when an incoming capsule arrives announcing the availability of some version of an agent, there will be a convenient place to look up whether this advertised code is needed.

A description of the mobile agent protocol (like WestNile) follows: There's a trade off here:

1. send all agent code together in a single packet, or

2. send activation guard (a_guard) code, existence guard (e_guard) code, and agent code as separate packets.

We believe the 2 approaches will need the same amount of logic to keep track of the 3 pieces of code. If a Forth bootloader were devised the entire Mobile Code process could be implemented in Forth. The following example is for approach 2.

- Source: type = NEW_AGENT_CODE_AVAILABLE, agent_ID, agent_version, num a_guards blocks, num e_guard blocks, num agent code blocks
- Sink: type = SEND_A_GUARD_CODE, agent_ID, agent_version
- Source: type = NEW_A_GUARD_CODE, agent_ID, agent_version, block number, 16 bytes of code for this block number
- (repeat until all A_GUARD blocks sent)
- Sink: type = SEND_E_GUARD_CODE, agent_ID, agent_version
- Source: type = NEW_E_GUARD_CODE, agent_ID, agent_version, block number, 16 bytes of code for this block number
- (repeat until all E_GUARD blocks sent)
- Sink: type = SEND_AGENT_CODE, agent_ID, agent_version
- Source: type = NEW_AGENT_CODE, agent_ID, agent_version, block number, 16 bytes of code for this block number
- (repeat until all AGENT_CODE blocks sent)

The Sink then calculates missing blocks and requests them by block number. Any Gnat can reply.

4.3 Context switching of mobile code agent

As in Mate and Agilla, the Gnats using the Byte Code Interpreter can have separate execution contexts. This is made possible because we have software control over the interpreted code program counter, stack pointers, and stacks. A context switch would be as simple as saving the previous context pointers, and loading in the new ones. In Mate and Agilla there are 4 contexts and they context switch between *each* instruction. On the Gnats we could easily have 2 or 3 and maybe 4 concurrently running agents. We are probably too resource constrained to have a variable number of contexts as in general purpose OSes unless we could keep the saved state of each execution context in the *i²c* memory. If save and restore functions using this memory is quick enough, we could have quite a few contexts.

4.4 Mobile Code Future...

An interesting demonstration may be to have the Gnats run bytecode *directly* from the robot. If the transfer was quick enough this might be possible; or the Gnat could cache several (16 or so) byte codes in RAM. This would resemble a little applet where the robot uses the Gnat (or a nearby grouping of Gnats) to perform some calculation for it. The Gnat would consider it a “run one time” agent.

A The GNAT Exerciser Manual

A.1 Introduction

The Gnat Exerciser will work with either version of Gnat hardware, and uses version 2 or 2.1 of the Gnat API. A known good Gnat is used as a helper for the communications testing, and is called the Companion Gnat. Right now the companion code only works on the older Gnat hardware. The companion code is called GE-Companion (for Gnat Exerciser Companion). It blinks green when an appropriate packet is received and blinks red when the button is pressed to transmit a packet.

The exerciser program performs the following:

1. The usual 3 green LED blinks upon startup to show the init phase has successfully completed.
2. A 1 second red LED display.
3. A 1 second green LED display.
4. Sends a 2 byte message (0xA5F0) over each (4) IR emitter separately and flashes its red LED after each transmission. The companion Gnat flashes its green LED to show a successful transmission.
5. Waits to receive a 2 byte IR transmission on each of its 4 receivers separately, and flashes its green LED show successful reception. The companion Gnat is used to transmit. Just press the companion Gnat button, and its red LED will light to show transmission. A blinking red LED on the Gnat being tested means there was some transmission error. No LED flash on the Gnat being tested means no IR was detected on that receiver pin (you should debug that pin).
6. Tests the battery level. Blinks red/green/red quickly to show the beginning of the process, logs value to locations 0x11-0x12 (and is surrounded by values 0xFA and 0xCE to make it easy to find). The Gnat then blinks red/green/red quickly to show completion. The battery level test takes 4 to 5 seconds. Read the value from the EEPROM via the MCP IDE, and to decode, use this formula:

$$(((\text{logged_value} * -1.32/4096) + 2.2) * 2.4).$$

7. Tests the temp sensor - ON NEW GNATS ONLY. This test is followed by a quick red/green/red LED flash. 10 readings will be made and will be logged into onchip EEPROM beginning at location 0x15 (and are surrounded by values 0xEE and 0xEE to make it easy to find). If 0xDD is at location 0x14 instead then there was a problem in reading the temperature sensor.

8. Tests the offchip memory: write/read check - ON NEW GNATS ONLY 16 bytes are written to 0x0000-0x0015 to test the lower address byte; these values are also written to the onchip EEPROM. These values are then read back and written to the onchip EEPROM. All 32 bytes are surrounded by values 0xBB and 0xBB to make it easy to find. The same test is then run on addresses 0x0100-0x0115. This tests the upper address byte. This test is followed by a quick red/green/red LED flash.
9. A 2 second delay before starting the process over.

A.2 Procedure

1. Check out of CVS the GnatExerciser module. This contains 2 MCP projects: one for the companion and one for the exerciser.
2. NOTE: Use the version 2 API headers that are included in the module. They are modified to work with an older version of the communications code.
3. Program a known good Gnat with GE-Companion.c.
4. Program the Gnat to be tested with GnatExerciser.c.
5. Turn on both Gnats.
6. The Gnat being tested will begin the testing procedure as defined above. Be sure to have the companion close by for the communications testing.
7. Remove power from the Gnat being tested and read in its onchip EEPROM. You should see:
 - (a) FA (battery voltage in 2 bytes) CE
 - (b) EE (10 temperature readings) EE
 - (c) BB (16 bytes that were written to off chip memory) (16 bytes that were read from the same locations) BB
 - (d) BB (16 bytes that were written to off chip memory) (16 bytes that were read from the same locations) BB
8. The first and second 16 bytes in item 7c should match. The same for the second group of 32 bytes. The first tests the low address byte and the second tests the high address byte.

B List of Forth Words

B.1 Forth keywords run directly by the Gnat Byte Code Interpreter

B.1.1 API keywords

These API keywords are basically Forth “wrappers” for selected Gnat API function calls. See the `GNATsDeveloper’sManual` for full details of these calls.

TURN_ON_LED	turn_on_led()
TURN_OFF_LED	turn_off_led()
GET_RECV_QUEUE_STATUS	get_recv_queue_status()
GET_RECV_STATUS	get_recv_status()
GET_CAPSULE	get_capsule()
SEND_CAPSULE	send_capsule()
DELAY_MS	delay_ms()
GET_TICKS	get_ticks()
BUTTON_PRESSED?	button_down() and wait_for_button_release()
SET_CPU_SPEED	set_cpu_speed()
RECV_SAFE_DELAY	recv_safe_delay()
GET_BATTERY_DACVALUE	get_battery_dacvalue()
READ_THERMAL_SENSOR	read_thermal_sensor()

And their stack usage comments...

TURN_ON_LED	(power led_number --)
TURN_OFF_LED	(led_number --)
GET_RECV_QUEUE_STATUS	(-- queue_status_value)
GET_RECV_STATUS	(-- status_value)
GET_CAPSULE	(number_items_to_move --)
SEND_CAPSULE	(number_of_items_to_send IR_power IR_emitters --)
DELAY_MS	(number_of_milliseconds --)
GET_TICKS	(-- number_of_ticks)
BUTTON_PRESSED?	(-- button_pressed_status)
SET_CPU_SPEED	(new_speed --)
RECV_SAFE_DELAY	(number_of_delay_cycles --)
GET_BATTERY_DACVALUE	(-- battery_dacvalue)
READ_THERMAL_SENSOR	(-- thermal_sensor_value)

B.1.2 Standard Forth keywords

These keywords are Gnat Forth implementations of standard Forth. Their functions are fully compliant with standard Forth. See Leo Brodie’s book, “Starting Forth” for full descriptions and tutorials. This book is available online from: <http://home.vianetworks.nl/users/mhx/sf.html> or <http://home.iae.nl/users/mhx/sf.html>.

DO Begins definite loop
 LOOP Ends definite loop. Increment loop counter by 1.
 +LOOP Ends definite loop. Increment loop counter by TOS.

I Push the definite loop counter on the data stack.
 J Push the nested definite loop counter on the data stack.

BEGIN Begins indefinite loop.
 UNTIL Ends indefinite loop.L

IF Conditional test. Marks beginning of conditional true code.
 THEN Marks end of conditional code.
 ELSE Marks beginning of conditional not true code.

: Begins a Forth word definition.
 ; Ends a Forth word definition.

FALSE Push 0 on the data stack.
 TRUE Push 1 on the data stack.

+ Arithmetic addition operator.
 - Arithmetic subtraction operator.
 * Arithmetic multiplication operator.
 / Arithmetic division operator.
 % Arithmetic modulo operator.

= Conditional equal operator.
 != Conditional not equal operator.
 < Conditional less than operator.
 > Conditional greater than operator.

SWAP Swap top 2 data stack elements.
 DUP Duplicate the top data stack element.
 DROP Remove the top data stack element.

VARIABLE Create a variable (in PIC onchip data EEPROM).
 ! Store value on top of data stack in variable.
 @ Fetch value from variable to top of data stack.

OR Bitwise OR. Can also be used for conditional.
 AND Bitwise AND. Can also be used for conditional.

>R Move from data stack to return stack.
 R> Move from return stack to data stack.
 R@ Copy from return stack to data stack.

B.1.3 Gnat specific Forth Non-API keywords

These are Gnat specific additions to the standard Forth implementation on the Gnats. See Section 2 for a tutorial that contains examples of how to use these keywords.

```
END_PROGRAM   Marks the end of the forth program.

[             Begins an agent guard definition.
]             Ends an agent guard definition.

!C            Store for Gnat send buffer.
Usage:
    value send_buffer_offset !C

@C            Fetch for Gnat receive buffer.
Usage:
    recv_buffer_offset @C

RAM_VARIABLE  Create a variable in PIC RAM heap.
Usage:
    RAM_VARIABLE  variable_name
```

B.2 Forth Keywords Implemented in Gnat Forth

These are miscellaneous Forth and Gnat API keywords that have been implemented in Forth. They are contained within gnatforthlib.gf, and the user may either #include this file or copy the desired keyword and any dependencies that keyword may have from gnatforthlib.gf to their own source file.

```
: 1+ ( n1 -- n1 + 1 )
: 1- ( n1 -- n1 - 1 )
: 2+ ( n1 -- n1 + 2 )
: 2- ( n1 -- n1 - 2 )
: 2* ( n1 -- n1 * 2 )
: 2/ ( n1 -- n1 / 2 )

: MIN    ( n1 n2 -- min )
: MAX    ( n1 n2 -- max )
: NEGATE ( n  -- -n )
: ABS    ( n  -- |n| )

: 0= ( n1 -- n2 )
    ( where n1 is a number to compare to 0 )
    ( n2 = true if n1=0 else n2 is false )

: 0< ( n1 -- n2 )
```

```

        ( where n1 is a number to compare to 0 )
        ( n2 = true if n1>0 else n2 is false )

: 0> ( n1 -- n2 )
      ( where n1 is a number to compare to 0 )
      ( n2 = true if n1<0 else n2 is false )

: >= ( n1 n2 -- n3 )
      ( n3 = true if n1 >= n2 else n3 is false )

: <= ( n1 n2 -- n3 )
      ( n3 = true if n1 <= n2 else n3 is false )

: NOT ( n1 -- n2 )
      ( n2 is logical inverse of n1 )

: INVERT ( n1 -- n2 )
        ( n2 is logical inverse of n1 )

: ROT   ( n1 n2 n3 -- n2 n3 n1 )
: OVER  ( n1 n2 -- n1 n2 n1 )
: 2DROP ( n1 n2 -- )
: 2DUP  ( n1 n2 -- n1 n2 n1 n2 )
: 2SWAP ( n1 n2 n3 n4 -- n3 n4 n1 n2 )
: 2OVER ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 )

: DISPLAY_LED      ( delay_time led_power led number_times_to_blink -- )
: DISPLAY_COUNT    ( time count -- )
: DISPLAY_DATA_BYTE ( data_byte -- )
: FLASH_RED_QUICK  ( -- )
: FLASH_GREEN_QUICK ( -- )

```

References

- [1] Philip Lewis and David Culler “Maté: A Tiny Virtual Machine for Sensor Networks”.
- [2] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu, “Mobile Agent Middleware for Sensor Networks: An Application Case Study”
- [3] David Wetherall, “Active Networks: Vision and Reality: Lessons from a Capsule-based System”, in *17th ACM Symposium on Operating System Principles*, OS Review, Volume 33, Number 5, Dec. 1999.