```
    }

    /* incrementNumPtr - increment the value of a variable.
     *  args: *pn - a pointer to an int
     *  return: none
     *  effect: the variable pointed to by pn is incremented
     */
    void incrementNumPtr(int *pn) {
      // pn is a pointer to the int, and *pn is the int itself
      *pn++; // * binds tightly, so parens (*pn)++ not needed.
    }
```

11.2. Programming the Propeller in C

In order to program the propeller with C code, we have to recognize a few constraints of the device. The first, and most critical, is that Hub memory is limited to 32KB and that Cog RAM is limited to 2KB. Next, the propeller has eight cogs and the C compiler and linker have to handle launching new cogs properly. In this book, I will discuss three cases (cases that I think cover most of the likely projects)

- If the size of the C program (after compiling) is less than approximately 30KB, then it will fit entirely in Hub memory. The compiler will place your code into Hub along with a *kernel* (approximate size 2KB, for a total size of less than 32KB). The kernel is a program that copies instructions from your code into a cog and executes them. This is known as the *Large Memory Model* or **LMM**. The main drawback of LMM is that every instruction resides in Hub and is copied to cog before execution, slowing down the program. In almost every way, though, this is a standard C program. Cogs can be launched and stopped; the counters and special registers like `ina` and `outa` can be read and set; the locks can be used, etc.

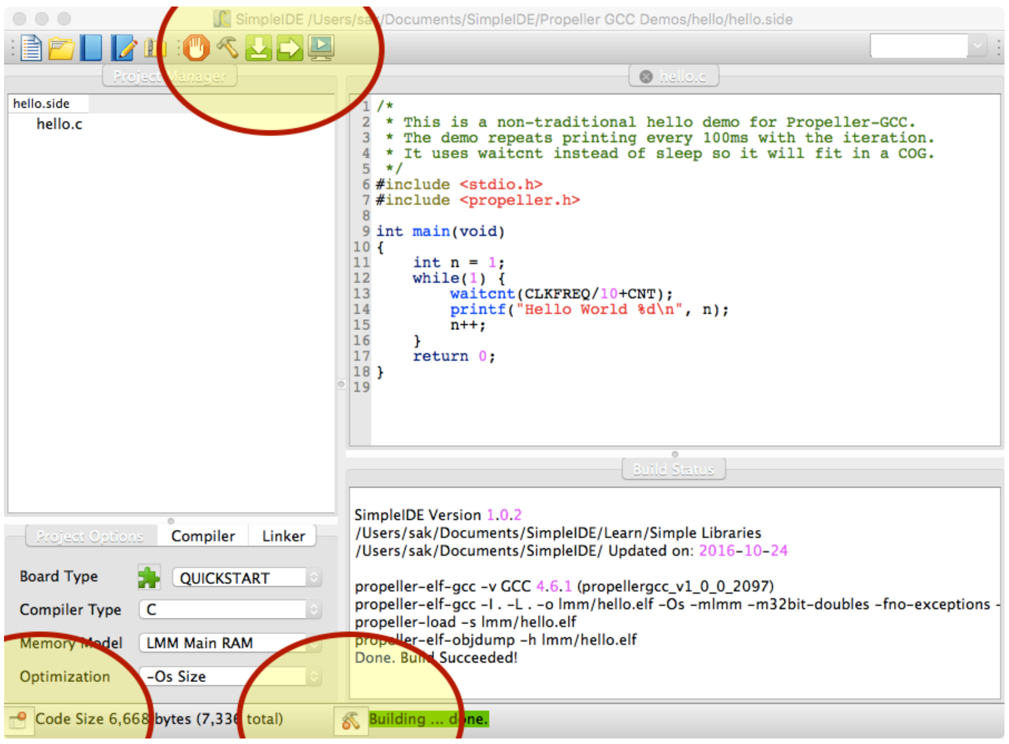
- If there is a need for higher speed from some part of the program, we can place that C code in a cog-c file (with extension `.cogc`). This part of the code must compile to Assembly code that is less than 2KB in size, and it will be placed into cog RAM and will run at full speed. The rest of the program will continue to operate under LMM mode. I call this *Mixed mode cog-c Model.* The advantage is that the program will run at full speed. The drawback is that the assembly code produced by the compiler may not be as efficient as code that you write.

- The final model is where the speed-critical code is written in PASM and is saved on a cog. This cog is now fully under your control. You can optimize it for your needs. (of course, as with the previous case, the code has to be less than 2KB in size). The rest of the code continues to run under LMM. This is *Mixed mode PASM model.*

11.2.1. SimpleIDE

To write the code, compile and link it, download it to the propeller, and view the output, we will use the SimpleIDE application (an Integrated Development Environment). This is a cross-platform IDE (Windows, Linux, and Mac OS) that is aware of all three of the models above and does the detailed work of compiling the programs correctly and linking them in the right way.

The place to start with SimpleIDE is at `learn.propeller.com` where you can download the program and step through a series of excellent tutorials on using the program and developing LMM projects.



```c
/*
 * This is a non-traditional hello demo for Propeller-GCC.
 * The demo repeats printing every 100ms with the iteration.
 * It uses waitcnt instead of sleep so it will fit in a COG.
 */
#include <stdio.h>
#include <propeller.h>

int main(void)
{
    int n = 1;
    while(1) {
        waitcnt(CLKFREQ/10+CNT);
        printf("Hello World %d\n", n);
        n++;
    }
    return 0;
}

```

Board Type: QUICKSTART
Compiler Type: C
Memory Model: LMM Main RAM
Optimization: -Os Size

```
SimpleIDE Version 1.0.2
/Users/sak/Documents/SimpleIDE/Learn/Simple Libraries
/Users/sak/Documents/SimpleIDE/ Updated on: 2016-10-24

propeller-elf-gcc -v GCC 4.6.1 (propellergcc_v1_0_0_2097)
propeller-elf-gcc -I . -L . -o lmm/hello.elf -Os -mlmm -m32bit-doubles -fno-exceptions -
propeller-load -s lmm/hello.elf
propeller-elf-objdump -h lmm/hello.elf
Done. Build Succeeded!
```

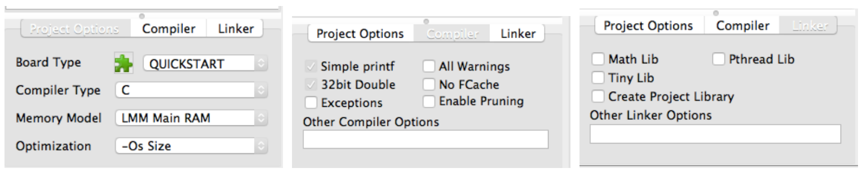
Code Size 6,668 bytes (7,336 total)   Building ... done.

Figure 11.2.: SimpleIDE window with Project Manager button, Build Window button (at bottom of picture) and Build button (the hammer at the top of the picture) highlighted.

There are three tabs in the Project Manager (at bottom left of Figure 11.2): the Project Options tab, the Compiler tab, and the Linker tab. The settings shown for each in Figure 11.3 are good for the examples in this book, so make sure you set them properly.

11.2.2. Hello, World

After installing SimpleIDE, you will have a number of examples in the `Propeller GCC Demos` folder (in the SimpleIDE workspace). One of those is called `hello`. Open the



Board Type: QUICKSTART
Compiler Type: C
Memory Model: LMM Main RAM
Optimization: -Os Size

✓ Simple printf
✓ 32bit Double
☐ Exceptions
☐ All Warnings
☐ No FCache
☐ Enable Pruning
Other Compiler Options

☐ Math Lib
☐ Pthread Lib
☐ Tiny Lib
☐ Create Project Library
Other Linker Options

Figure 11.3.: Settings for the Project Options, Compiler, and Linker tabs.

project file `hello.side` and build it; you should see messages in the Build Window ending with "Build Succeeded!". At the bottom of the window a message shows the size of the program (in this case, about 7KB). If the program successfully builds, you can download and run it on the propeller and have the output displayed on a terminal by pressing the icon at the top that shows a screen with an arrow.

The code itself is very straightforward:

Listing 11.3: Hello World program in C

```
1  #include <stdio.h>
2  #include <propeller.h>
3
4  int main(void) {
5      int n = 1;
6      while(1) {
7          waitcnt(CLKFREQ/10+CNT);
8          printf("Hello␣World␣%d\n", n);
9          n++;
10     }
11     return 0;
12 }
```

Lines 1–2 The `stdio` library has the `printf` function. However, because it is a complex (and large) function, the Compiler tab includes the option to use a "Simple printf" that reduces the size somewhat. The `propeller` library has the propeller-specific functions such as `waitcnt`, `waitpeq`, and the special registers such as `CNT`, `INA`, etc.

Lines 4–13 The main program is similar to the `PUB MAIN` method in Spin. This function shouldn't exit–it should initialize some variables and then enter an inifinite loop.

Line 5 Define the variable `n` and initialize it to one.

Line 6 Enter an infinite loop.

Line 7 `waitcnt` is similar to the `waitcnt` in Spin, but it only has one argument. The processor will pause at this line until the counter value is equal to the argument of `waitcnt`: in this case, the current count value + one-tenth of a second. The variable CLKFREQ contains the number of counts in one second (generally 80 million at top speed, but it depends on the external crystal and the phase locked loop value).

Line 8 The `printf` function prints a formatted string to the terminal. Look at the manual page for `printf` for how to format numbers. In short, `%d` prints a decimal number, `%x` prints the number in hexadecimal format, `%f` prints a floating point value.

Line 9 Increment the value of `n`.

Running the program will result in the following in the terminal window:

```
Hello World 1
Hello World 2
Hello World 3
...
```

with a new message every tenth of a second.

11.2.3. Launching a new Cog

In order to launch a new cog in LMM mode, we must define a function and then pass that function to the `cogstart` function.

Start a new C project (Open→New) named `compr_cog0`. Set the Project Options, Compiler, and Linker as before.

For the purposes of display and discussion, I have split the file `compr_cog0.c` into three separate parts below, but really all three parts are in one file. Every multi-cog program will have these three parts.

Part 1 is the front matter where the libraries are included, the shared memory for the *stack* and the shared variables is set aside, and the constants are defined.

Listing 11.4: Part 1: Front matter for file compr_cog0.c

```c
/*
  compr-cog0.c - start a new cog to perform compression.
*/

/* libraries */
#include <stdio.h>
#include <propeller.h>

/* defines */

// size of stack in bytes
#define STACK_SIZE_BYTES 200
// compression constants
#define NSAMPS_MAX 128
#define CODE08 0b01
#define CODE16 0b10
```

```
17 #define CODE24 0b11
18 #define TWO_BYTES 0x7F // any diff values greater than this are 2
       ↪ bytes
19 #define THREE_BYTES 0x7FF // diff valus greater than this are 3
       ↪ bytes
20
21
22 /* global variables */
23 // reserved space to be passed to cogstart
24 static unsigned int comprCogStack[STACK_SIZE_BYTES >> 2];
25
26 // shared vars
27 volatile int nsamps;
28 volatile int ncompr;
29 volatile int sampsBuf[NSAMPS_MAX];
30 volatile char packBuf[NSAMPS_MAX<<2]; // 128 * 4
31 volatile int comprCodesBuf[NSAMPS_MAX>>4]; //128 / 16
```

LIne 12 The stack is region of memory used by the kernel to store internal variables and state. It should be at least 150 bytes plus four bytes per function call in the cog.

Lines 14–19 Constants used by all cogs.

Line 24 Declare and reserve space for the stack here.

Lines 27–31 Shared variables have a `volatile` qualifier to signal the compiler not to remove them during optimization. If the compiler thinks a variable is unused it won't reserve space for it. However, it is possible that a variable is used by a Spin or PASM cog unknown to the compiler.

Part 2 is the code for the cog. Define a function that is called by the main cog. This function (and any functions that it calls) will run in a separate cog from the main cog. However, this cog will have access to the variables declared above. Those are global variables and available to all functions in the file.

Listing 11.5: Part 2: Compression cog code in file compr_cog0.c.

```
/* cog code - comprCog
   use nsamps and ncompr to signal with main cog
     start compression when nsamps != 0
     signal completion with ncompr > 0
     signal error with ncompr = 0
   compress sampsBuf to packBuf - NOT DONE YET
   populate comprCodesBuf - NOT DONE YET
   - args: pointer to memory space PAR - UNUSED
   - return: none
 */
void comprCog(void *p) {
```

```
12    int i, nc, nbytes, codenum, codeshift, code;
13    int diff, adiff;
14
15    while(1) {
16      if (nsamps == 0) {
17        continue; // loop continuously while nsamps is 0
18      } else {
19        // perform the compression here
20        if (nsamps > NSAMPS_MAX || nsamps < -NSAMPS_MAX) {
21          ncompr = 0; // signal error
22          nsamps = 0;
23          continue;
24        }
25        ncompr = 3; // signal completion
26        nsamps = 0;  // prevent another cycle from starting
27      }
28    }
29  }
```

Line 11 Cog function definition: `void comprCog()` means that this doesn't return any value. The argument (`void *p`) means that an address is passed in—this is the equivalent of `PAR`. However, because we are using the global variables to pass information between cogs, we won't use `PAR`.

Lines 12–13 Local variables used by the cog.

Line 15 Inifinite loop that contains the actual code that does the work of the cog.

Lines 16–18 If `nsamps` is set to non-zero by main, enter the section that does the work.

Lines 20–26 Error checking and finally, the work of this cog. Set `ncompr` to 3 and `nsamps` to zero. (we will add code to do the actual compression later.)

Part 3 is the entry point for the program: the main function and the code that runs first. Again, this cog has access to the global variables. It also starts the new cog and interacts with it by setting and reading variables in those global variables.

Listing 11.6: Part 3: Main code in file compr_cog0.c.

```
/* main cog - initializes variables and starts new cogs.
 * don't exit - start infinite loop as the last thing.
 */
int main(void)
{
  int comprCogId = -1;
  int i;

  nsamps = 0;
  ncompr = -1;
11
12    printf("starting␣main\n");
13
14    /* start a new cog with
15     * (1) address of function to run in the new cog
16     * (2) address of the memory to pass to the function
17     * (3) address of the stack
18     * (4) size of the stack, in bytes
19     */
20    comprCogId = cogstart(&comprCog, NULL, comprCogStack,
         ↪ STACK_SIZE_BYTES);
21    if(comprCogId < 0) {
22      printf("error␣starting␣compr␣cog\n");
23      while(1) {;}
24    }
25
26    printf("started␣compression␣cog␣%d\n", comprCogId);
27
28    /* start the compression cog by setting nsamps to 1 */
29    sampsBuf[0] = 0xEFCDAB;
30    nsamps = 1;
31
32    /* wait until the compression cog sets ncompr to a non-neg
         ↪ number */
33    while(ncompr < 0) {
34      ;
35    }
36
37    printf("nsamps␣=␣%d,␣ncompr␣=␣%d\n", nsamps, ncompr);
38    printf("samp0␣=␣%x,␣packBuf␣=␣%x␣%x␣%x\n", sampsBuf[0], packBuf
         ↪ [0], packBuf[1], packBuf[2]);
39
40    while(1)
41    {
42      ;
43    }
44  }
```

Line 20 This is the key in `main`. The `cogstart` function takes 4 arguments. The first is the address of the function to place in the new cog: `&comprCog`. The ampersand symbol ()`&`) in C is to obtain the address of a variable or function. The next argument is the address of memory that will be passed to the cog in `PAR` (the “locker number” in my analogy in Chapter 6). In this case, because we are using global variables to exchange information, we won’t use `PAR` and can pass `NULL` (which is, as it states, the null pointer). The third and fourth arguments are the address of the reserved stack space and its length in bytes, respectively. The stack is a region of memory that the kernel needs to store variables and counters.

Line 30 Here we set `nsamps=1`, which signals the compression cog to begin its work.

Lines 33-35 The compression cog will set `ncompr` to a non-negative number when it completes its work. The main cog waits in this loop until it sees that the compression cog is finished.

Lines 37–38 Print out the results. `nsamps` should now be zero, and `ncompr` should now be non-negative.

Line 40 Enter an infinite loop, doing nothing.

The output from running this program is:

```
starting main
started compression cog 1
nsamps = 0, ncompr = 3
samp0 = EFCDAB, packBuf = 0 0 0
```

We have shown that we can communicate with the compression cog. How fast is it? In order to compare this to the Spin and PASM compression codes from the previous chapters, let's implement the compression code.

11.2.4. Compression code in C

We will edit the `comprCog` to include the packing. Replace the code with the following. It forms the difference between successive samples and checks the length of the difference. Depending on that length, it saves it either 1, 2, or 3 bytes of `packBuf`. (for simplicity I haven't included the part that populates the compression code `comprCodesBuf`).

```
  /* cog code - comprCog
   use nsamps and ncompr to signal with main cog
     start compression when nsamps != 0
     signal completion with ncmopr > 0
     signal error with ncompr = 0
   compress sampsBuf to packBuf
   populate comprCodesBuf - NOT YET DONE
   - args: pointer to memory space PAR - UNUSED
   - return: none
 */
void comprCog(void *p) {
  int i, nc, nbytes, codenum, codeshift, code;
  int diff, adiff;

  while(1) {
    if (nsamps == 0) {
      continue; // loop continuously while nsamps is 0
    } else {
      // perform the compression here
      if (nsamps > NSAMPS_MAX || nsamps < -NSAMPS_MAX) {
```