Selecting the Correct Catalina Options for your Platform

On first glance, the ***Catalina Reference Manual***, and even the ***Catalina Command Summary*** document can be a bit daunting – especially for those unused to C, or who are coming to Catalina from Spin. Many newcomers to Catalina seem to find themselves asking the same questions:

Why does Catalina need so many options? Where do I start?

While many of the Catalina command-line options are self-explanatory, the options related to selecting the correct libraries, memory models and loader options seem to cause some confusion. With the advent of new “add-on” XMM boards such as the Propeller Memory Card, these questions are arising more and more often – hence this document, as an attempt to help address the issue. Note that most of this document is specific to the Propeller 1, which has very limited on-board RAM (32 kb) and therefore requires lots of options to make the most use of it. The Propeller 2 (which has 512 kb of on-board RAM) has less options because we don’t need so many options specifically intended to optimize the use of RAM or support add-on XMM boards.

One of the original guiding principles of Catalina was that the run-time should, as far as possible, be kept independent of the language. The idea being that you just write your program in "plain old C", and decide later how the program will best make use of the resources available on your particular Propeller platform. If you examine the many demo programs provided with Catalina, you will find few where the C code is dependent on, or has to be modified for, the particular Propeller platform on which it is executed (apart from programs that use hardware-dependent things like pin numbers which differ from platform to platform).

This is a different approach than that taken by many other language compilers – especially in the embedded software domain, where languages are sometimes so hardware dependent that they become very difficult to understand without a deep knowledge of the hardware on which they are designed to run. In other cases, a general purpose language is modified to suit - and end up littered with non-portable extensions that can obscure the purpose of the programs themselves.

Catalina takes neither approach. In fact, originally Catalina made *no concessions at all* to the Propeller hardware. Recent versions have added support for in-line PASM and direct support for the special registers as predefined C identifiers (I finally gave in on these requests because they came up so often) – but little else. But there is a price to pay for this - with Catalina, all these decisions are made *outside* the language – mostly in the form of command-line options, or in the form of **Catalina Geany** build options – which makes selecting the correct options very important!

Probably the most important choices to be made when compiling a Catalina program are to do with the libraries, the memory model, and the loader options. To demonstrate the usage of the Catalina options in these particular areas, we will use two programs:

sumeria.c	a game that can run on *any* propeller.
chimaera.c	a game that requires some kind of XMM RAM.

The sources for both are included in the Catalina demos/games folder. Or they can be downloaded from:

http://www.mipmip.org/C_games/

I chose them because they are both “plain old ANSI C” programs that I could use unmodified, and because they were the right size to demonstrate the various Catalina memory models. They are also quite fun to play!

You can examine the source code for these games using any text editor - but apart from the size of the resulting executables, the code itself is irrelevant to the purpose of this exercise.

Note that we will be using the command-line version of Catalina for this entire exercise, but that everything that is mentioned or done here can also be done from within the **Catalina Geany** integrated development environment.

Let's get started ...

LESSON 1. The Basics

For the first few Catalina compilation examples, we will use *sumeria.c* - a game in which you play the part of Hamurabe, managing the economy of ancient Sumeria.

To compile it, open up a Catalina Command Window, then go to the folder containing the games source files (in the Catalina source tree, this is *demos\games*, but it is recommended that you copy the entire demo folder to your own user directory), then try compiling s*umeria.c* using the following command[1]:

```
catalina sumeria.c
```

You should see something like:

```
Undefined or Redefined symbols:
 srand undefined
 exit undefined
 rand undefined
 log10 undefined
 time undefined
 puts undefined
 putc undefined
 fgets undefined
 printf undefined
 __iotab undefined
```

Here is the first thing you need to know – you must specifically tell Catalina what libraries to search for any functions that are not defined in the source files being compiled.

Catalina provides several variants of the C libraries to support different applications. To use the most commonly used C library, we would include the option **-lc** (read this as *use*

[1] Note that most of this document assumes you are using a Propeller 1. Much of it is also applicable to the Propeller 2, but if you are compiling for a Propeller 2 you will need to add -p2 to each Catalina command line to tell Catalina to compile for a Propeller 2 instead of a Propeller 1.

*the **lib**rary named **c** – you will find this in the Catalina lib folder as **libc***) on the command line:

```
catalina sumeria.c -lc
```

Instead of **-lc**, we could have used **-lci, -lcx** or **-lcix**. These variants are integer-only versions, or extended versions of the basic C library. But for now, stick with **-lc** (we will use **-lcx** later). This command should now produce output similar to the following:

```
Undefined or Redefined symbols:
 log10 undefined
```

It seems that *most* of the functions are now defined, but the C library does not contain a function called **log10**. This is deliberate – Catalina (like many C compilers – especially those for embedded systems) does not include the maths functions in the standard C library - to include them as well, we must also add the option **-lm** (read this as *use the **lib**rary named **m** – you will find this in the Catalina library folder tree as **libm***):

```
catalina sumeria.c -lc -lm
```

The main reason for not including the maths functions is that (again) there are several variants of this library – instead of **-lm**, we could have used **-lma** or **-lmb** (on the Propeller 1 or 2), or **-lmc** (on the Propeller 2 only). Each of these variants requires a different amount of code space, traded off against requiring a different number of cogs at run time (0, 1 or 2). If we instead combined each possible variant of the maths library with each variant of the basic C library, we would have to have 16 different library combinations!

Using any of these maths libraries will get rid of the undefined symbols, but for now stick to **-lm**. Executing the above command should now produce output similar to the following:

```
error : Object exceeds runtime memory limit by 826 longs.
```

This indicates that the resulting executable is too large to fit in the Propeller 1's 32 kb of RAM. Don't worry – this is because Catalina's default code generator and default C library favours *execution speed* and *C language compatibility* over *code size*. We can make this program fit by trading off either of these (for example, very few programs actually need all the capabilities provided by the standard C library functions) but for the purpose of this exercise we will choose to maintain C compatibility, and instead explore the code size option – i.e. trading off a slower execution speed for reduced code size.

We will do this by requesting that Catalina use a different code generator that generates much smaller code. Like most things, this is done by specifying a Catalina symbol on the command line.

Each Catalina symbol is specified separately on the command line using the **-C** option. We will request the compact memory model (CMM) by specifying the symbol **COMPACT** [2]. So we must add **-C COMPACT** to our command, as follows:

```
catalina sumeria.c -lc -lm -C COMPACT
```

This time the program should compile correctly, and will print some size statistics, which

[2] Users familiar with other C compilers may see similarities between **-C** and the **-D** command-line option common to many C compilers. Catalina also fully supports **-D**, but the difference is that b is used to defined C symbols, whereas **-C** is used to define Catalina symbols, which do no affect the C language itself – you might want to think of them as Configuration options.

should look something like this:

```
code = 13324 bytes
cnst = 3220 bytes
init = 248 bytes
data = 1292 bytes
file = 25304 bytes
```

The symbol **COMPACT** tells Catalina to select the CMM code generator in place of the default LMM code generator. CMM results in much smaller code sizes, but at the cost of reduced execution speed (but note that the resulting program will still execute faster than the equivalent Spin program).

You can now load and run this program on any Propeller that uses a 5 Mhz clock – and you can load the resulting binary file using any Propeller loader, since the binary produced by the CMM and LMM code generators are standard propeller binary. However, using Catalina's **payload** loader is recommended instead, since it can not only load normal binaries, but also Catalina's XMM binaries – and it also has a terminal emulator built-in that simplifies interacting with programs that use serial I/O. You can do this using the following command:

```
payload sumeria -i
```

The **-i** command-line option tells payload to start its interactive terminal emulator after loading the program.

Try it, and play the role of Hamurabe, managing the economy of ancient Sumeria! (press **CTRL+D** to exit the payload terminal emulator and return to the command line).

If you have a supported Propeller platform that does *not* use a 5 Mhz clock, you can still run the program, but you will also need to specify the Catalina symbol for the platform on the command line when compiling it. You can also add other symbols - see the ***Catalina Reference Manual*** or the ***Catalina Command Summary*** for a complete list, but here is a simple example for the **HYDRA**, which has a 10 Mhz clock. (remember that you need to use **-C** in front of *each* symbol, and that all Catalina command-line options and symbol names are case-sensitive):

```
catalina sumeria.c -lc -lm -O5 -C COMPACT -C HYDRA -C HIRES_TV
-C NO_REBOOT -C NO_MOUSE
```

This command adds the following additional Catalina symbols:

HYDRA	use the clock and pin definitions for the Hydra.
HIRES_TV	use HiRes TV for I/O (instead of serial I/O).
NO_REBOOT	do not reboot the Propeller on program exit - this allows you to read the final game messages!
NO_MOUSE	do not load a mouse driver (this program does not use the mouse).

It also adds the use of the Catalina Optimizer via the **-O5** option. This can reduce the size of code generated. Refer to the **Catalina Optimizer Reference Manual** for more details.

LESSON 2: Big programs on platforms with SRAM

The next program we will use to demonstrate Catalina is *chimaera.c*. Chimaera is a text-based adventure game, similar to "Zork" and many other classic games.

Unlike Sumeria, Chimaera is far too large to run on a Propeller without XMM RAM. Try compiling it and see:

```
catalina -lcx -lm chimaera.c
```

The result will be something like:

```
error : Object exceeds runtime memory limit by 36699 longs.
```

This message indicates the program would exceed the 32 kb size of HUB RAM by *almost 40,000 longs* (i.e over 160,000 *bytes*) – i.e. this program requires around *200 kb* of XMM RAM to execute. No amount of code compression, optimization, or choice of stripped-down libraries is going to get around *that!*

The answer to this problem is to use XMM RAM. XMM RAM comes in various types – SRAM (Static RAM) and FLASH RAM are the main ones. SRAM can be written and read to a byte at a time. Flash RAM can generally be read a byte at a time, but only written to a *page* at a time.

In this lesson, we will learn how to compile such a program for Propeller platforms which use Static RAM (SRAM) as XMM RAM. If your platform has both Static RAM and FLASH RAM, you may be able to use just the SRAM if it has at least 256 kb of it or more. If not, you should still continue this exercise, but you will not be able to execute the programs we compile - that situation will be covered in the next lesson, which adds details on how to use FLASH RAM as XMM RAM.

But first, notice that I snuck in something new in the above command: **-lcx**. Chimaera requires the *extended* version of the standard C library. The extended version includes full SD card-based file system support. The file system support code is so large that it is not included in Catalina's “normal” C library – but Chimaera needs it because it can log game information, and also save and load games from disk.

But don't worry if your Propeller platform does not have an SD Card - you can continue to compile the programs with **-lc**, but to run them you also need to edit *chimaera.h* to set **LOGGING** to **0**. For the purposes of this exercise, we will assume your platform has both 256Kb or more of SRAM and an SD Card (and before executing the program, be sure and have an SD card inserted, or the program will not run!).

For the purposes of this exercise, we'll assume you have a RamBlade3 Propeller platform. If you have a different platform, you will need to adjust the commands accordingly.

Note that the RamBlade3 is a platform with built-in XMM RAM, so we only need to specify *one* symbol (i.e. **-C RAMBLADE3**). If your platform also has an XMM "add-on” board, you may need to specify *two* symbols (e.g. in the next lesson we will use a QuickStart platform with a Propeller Memory Card add-on board to provide the XMM RAM, and we will have to specify this combination as **-C QUICKSTART -C PMC**).

First, let's compile the program to use XMM RAM and see how big it really is. To do this, we simply specify either Catalina's **SMALL** or **LARGE** memory models. We do so by

adding the Catalina symbol **SMALL** or **LARGE** to our command line.

Remember that **SMALL** is a relative term - **SMALL** programs are still larger than Propeller programs that must fit wholly within Hub RAM (in fact, Catalina refers to those as **TINY** programs, but since this is the default memory model on the Propeller 1 you rarely need to specify this explicitly).

First, we will try the **SMALL** memory model. **SMALL** programs are generally both smaller and faster than **LARGE** programs, but the disadvantage of **SMALL** is that only the code segment is put into XMM RAM. All the non-code segments, plus the heap and the stack, must still fit into 32Kb of Hub RAM.

To compile our program as a **SMALL** program, we would use a command like the following:

```
catalina chimaera.c -lcx -lm -C RAMBLADE3 -C SMALL
```

When we do this, we should see something like this:

```
code = 146268 bytes
cnst = 31772 bytes
init = 9812 bytes
data = 5912 bytes
file = 226612 bytes
```

This tells us the code size is about 150 kb. That's fine for a **SMALL** program, but we still have a problem - the total size of the *non*-code segments (cnst + init + data) is 32 + 10 + 6 = 48 kb! And this doesn't even include any heap or stack space – so this program is *way too big* to execute as a **SMALL** program!

So instead, let's compile it as a **LARGE** program. To do, we would use a command like the following:

```
catalina chimaera.c -lcx -lm -C RAMBLADE3 -C LARGE
```

We should see something like this:

```
code = 158684 bytes
cnst = 31772 bytes
init = 9812 bytes
data = 5912 bytes
file = 239468 bytes
```

LARGE programs only use the Hub RAM for stack space – everything else is in XMM RAM. The RamBlade3 has 512 kb of SRAM, which is enough to fit everything else, so this should be fine!

The next problem is to get this program loaded into the Propeller. Most Propeller loaders can only load the 32 kb Hub Ram (or the 32 kb EEPROM). To load programs into XMM RAM we need to use Catalina's **payload** loader – *and also compile the Catalina utilities for our platform*. The utilities include (amongst other things) **SRAM** and **FLASH** intermediate loaders that are first loaded into Hub RAM, and which know how to then load the actual program into the XMM on that platform. Since there is no common hardware standard for XMM RAM, these intermediate loaders have to be compiled specifically for the platform or XMM board they are to be used on.

To compile the utilities you execute the following command. This command is an interactive batch file, so you then follow the prompts, specifying information about your platform:

```
build_utilities
```

Note that it is important to build the utilities using options compatible with the compilation options you used (or intend to use) when compiling your programs. Most important in this case is to note that when we compiled the program, we did not specify the cache (using the cache will be covered in the next lesson) - so when we build the utilities, we must NOT specify a cache size, or the program will not load correctly.

Having built the utilities, we now load the program into SRAM via payload, by specifying the appropriate intermediate loader to use (**SRAM** in this case) as follows: [3]

```
payload SRAM chimaera -i
```

Try it, and enjoy adventuring in the land of Chimaera! (press **CTRL+D** to exit the payload terminal emulator and return to the command line).

[3] **HYDRA** users please note – you cannot use the serial I/O in conjunction with the Hydra XH512 add-on board - so you will need to choose another I/O option such as **HIRES_TV** or **HIRES_VGA**. Also, you will need to use the **MOUSE** intermediate XMM loader, not the **SRAM** one. Since we are using the mouse port as a serial port, you must also specify the **NO_MOUSE** option. Finally, since you cannot use the SD Card while the HX512 is plugged in, you must edit *chimaera.h* and set **LOGGING** to **0** and compile with **-lc**, not **-lcx**.

So on the HYDRA, the compile command will be something like:

```
catalina chimaera.c -lc -lm -C LARGE -C HYDRA -C HIRES_TV -C NO_MOUSE
```

and the load command (after running the **build_utilities** program) will be something like:

```
payload MOUSE chimaera
```


LESSON 3: Big programs on platforms with FLASH

Many Propeller 1 platforms use FLASH as XMM RAM, and some provide both FLASH and SRAM[4]. A board that provides only FLASH can be used to run **SMALL** programs, but not **LARGE** programs. A board that provides both SRAM and FLASH can be used for either **SMALL** or **LARGE** programs – and on such boards, programs can be run either out of SRAM only (**SMALL** or **LARGE** programs), FLASH only (**SMALL** programs only), or both (**LARGE** programs only).

We already know from the previous lesson that Chimaera requires the **LARGE** memory model, so let's assume that we have a QuickStart board, with a Human Interface Board and a Propeller Memory Card (**PMC**). The **PMC** has 4Mb of FLASH and 128Kb of SRAM, so it is fine for running Chimaera – but the size of the program necessitates that we use both the SRAM and the FLASH.

Using FLASH RAM as XMM RAM brings an additional complication - the code required to access the FLASH also requires the use of the XMM Cache. Fortunately, using the cache is quite easy - just add one of the **CACHED_nK** symbols to the command (where **n**=1,2,4, or 8). In this case, we will use **CACHED_1K**. To use the FLASH RAM (instead of just the SRAM) we also have to add the symbol **FLASH**. Finally, we want to use serial I/O, so we will add the symbol **TTY** (since this is not the default on the QuickStart).

Here is a summary of all the options we need to add:

LARGE	the large memory model
QUICKSTART	the platform (which by default has no XMM)
PMC	the XMM add-on board
CACHED_1K	the program is to use a 1kb cache
FLASH	use FLASH for code segments.
TTY	use serial I/O

And so here is what the final command looks like:

```
catalina chimaera.c -lcx -lm -C LARGE -C QUICKSTART -C PMC -C TTY
-C CACHED_1K -C FLASH
```

When we execute this command, we see the following output:

```
code = 158684 bytes
cnst = 31772 bytes
init = 9812 bytes
data = 5912 bytes
file = 239616 bytes
```

Again, before we can load the program, we need to build the utilities, using the following command:

```
build_utilities
```

[4] This section only applies to the Propeller 1, which tends to have FLASH or SRAM on a separate expansion board. Propeller 2 boards tend to have the FLASH integrated on the main board, and payload supports FLASH using a different mechanism. Refer to the Catalina Reference Manual for the Propeller 2 and look for the section on FLASH support, and specifically the flash_payload command.

In this case, we not only have to specify the platform (**QUICKSTART**) and the XMM Add-on board (**PMC**) but we also have to specify the same cache size (i.e. **1**) that we used to compile the programs.

Then to load the program, we have to use the **FLASH** intermediate loader, as follows:

```
payload FLASH chimaera -i
```

Try it, and again enjoy adventuring in the land of Chimaera! (press **CTRL+D** to exit the payload terminal emulator and return to the command line).

LESSON 4: Other Possibilities

There are other possible ways to compile both Sumeria and Chimaera. For instance, another way to make Sumeria executable on the Propeller 1 would be to compile it to use XMM RAM, as either a **SMALL** or **LARGE** program (instead of as a **COMPACT** program).

Of course, to do this you need a platform with XMM RAM. For example, to use the RamBlade3, first rebuild the utilities, specifying **RAMBLADE3** as the platform, and specifying no cache size:

```
build_utiliites
```

Then we could compile and load Sumeria as follows:

```
catalina sumeria.c -lc -lm -C SMALL -C RAMBLADE3
payload SRAM sumeria -i
```

We can do much the same on the QuickStart with the Propeller Memory Card – but in this case we must also use the cache. Whether the cache is required or not depends on the platform - the cache is ALWAYS required when using FLASH, but when using SRAM only, it depends on how large the XMM support API is. The simplest way to determine this is to just try it - Catalina itself will tell you whether it needs the cache. For example, the command:

```
catalina sumeria.c -lc -lm -C SMALL -C QUICKSTART -C PMC -C TTY
```

will result in the following error:

```
#error: PLATFORM REQUIRES CACHE OPTION (CACHED_1K .. CACHED_8K or CACHED)
```

whereas the following command:

```
catalina sumeria.c -lc -lm -C SMALL -C QUICKSTART -C PMC -C TTY
  -C CACHED_1K
```

should compile successfully, and produce size statistics similar to the following:

```
code = 25804 bytes
cnst = 3220 bytes
init = 252 bytes
data = 1292 bytes
file = 63416 bytes
```

To load this program, we must rebuild the utilities – remember to specify the **QUICKSTART**, the **PMC**, and a cache size of **1** for the SRAM when prompted:

```
build_utilities
```

Then we can load the program using the **SRAM** intermediate loader, as follows:

```
payload SRAM sumeria -i
```

Of course, we may *choose* to use the cache even if it is NOT required. One reason for doing so is that some XMM hardware is quite slow, and the cache can speed up the program execution – at the cost of some additional Hub RAM (1 kb in this case) and the extra cog required to run the XMM Cache plugin.