

Getting Started with Catalina

Table of Contents

Basic Concepts.....	2
Opening a Catalina Command Line Window.....	2
Compiling your first Catalina Program.....	3
Compiling Multiple Files.....	6
Specifying Memory Models and Load Options.....	8
Specifying a Supported Propeller 1 Platform.....	12
Specifying a Supported Propeller 2 Platform.....	17
A note about Catalina Code Sizes.....	21
Specifying Options using Environment Variables.....	24
Loading Programs.....	25
Supporting Other Propeller 1 Platforms.....	26
Supporting Other Propeller 2 Platforms.....	28
Advanced Concepts.....	29
Using a different Target or Target Package.....	29
Using an SD Card with Catalina.....	31
Using the SD Card to load Programs.....	31
Using the SD Card as a file system.....	32
Multi-CPU Systems.....	34
Introduction.....	34
Specifying the CPU.....	34
Multi-CPU Utility Programs.....	35
Proxy Devices.....	36
TriBladeProp.....	38
TriBladeProp Overview.....	38
TriBladeProp Tutorial Hardware Setup.....	39
TriBladeProp Tutorial Software Setup.....	40
TriBladeProp Tutorial Part 1 – Loading and running programs.....	42
TriBladeProp Tutorial Part 2 – Using proxy devices.....	45
What next?.....	48

Basic Concepts

This document provides a brief tutorial to get you up and running and compiling C programs using the Catalina C compiler from the command line.

If you do mainly intend to use Catalina from within the **Catalina Geany IDE**, you may instead prefer to start with the tutorial given in the documents **Getting Started with the Catalina Geany IDE**. However, it is still recommended that you work through this tutorial once you have completed that, because this document covers some concepts not included in the **Geany** tutorial.

The tutorial starts out with some basic concepts, but also introduces some of the more advanced concepts, such as the different memory modes and addressing modes – however, it is not necessary to complete the whole tutorial to begin using Catalina.

If you have one of the supported platforms, you need only read up to the section **Loading Programs**. If you have a different platform you should also read the section **Supporting Other Propeller Platforms**. The advanced concepts sections can be tackled separately once you are familiar with the basic concepts.

This document assumes you are using Catalina under Windows. If you are instead using Catalina under Linux, then while the Catalina commands and options will be identical, some of the operating system commands and directory names will need to be replaced with their Unix equivalents.

This document also assumes you have installed Catalina in the default location (under Windows this is *C:\Program Files (x86)\Catalina*). It also assumes you have a Propeller 1. If instead you have a Propeller 2, most of this document is still relevant, except for the sections on the multi-CPU and proxy driver support. Note that for the Propeller 2 you have to explicitly add the **-p2** option to each catalina command to indicate that Catalina should produce code for the Propeller 2, not the Propeller 1 (which is the default).

If you have not installed Catalina yet, see the **Catalina Reference Manual** for instructions.

If you have already installed Catalina, but to a directory other than the default directory, then all the Catalina commands and options shown will be identical – but some of the directory names may need to be changed.

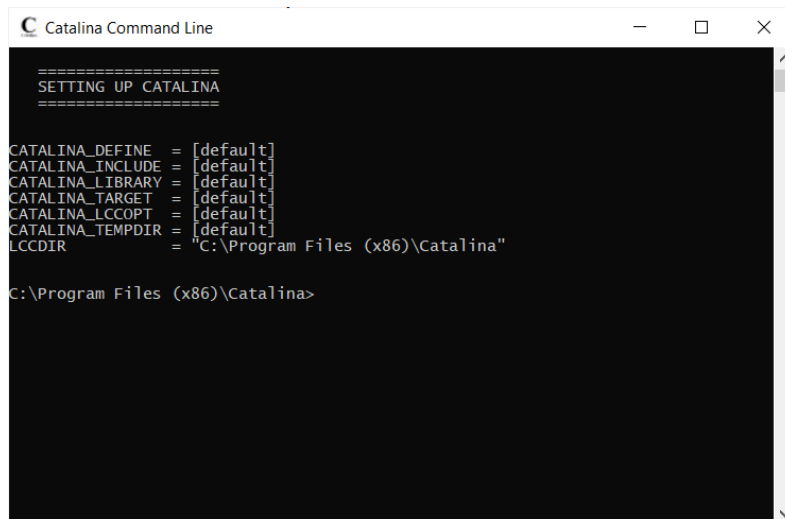
Note that this document is a *compiler* tutorial, not a *programming* tutorial – it assumes you are at least slightly familiar with the C language.

Opening a Catalina Command Line Window

Like most compilers, Catalina can be used from a command line window, such as the Windows command interpreter.

If you installed Catalina using the Windows Installer, a menu item called **Catalina Command Line** will have been created in the **Catalina** program group, and possibly also as a desktop icon – just select that menu entry or icon.

A command line window similar to the following will appear:



```

Catalina Command Line

=====
SETTING UP CATALINA
=====

CATALINA_DEFINE = [default]
CATALINA_INCLUDE = [default]
CATALINA_LIBRARY = [default]
CATALINA_TARGET = [default]
CATALINA_LCCOPT = [default]
CATALINA_TEMPDIR = [default]
LCCDIR          = "C:\Program Files (x86)\Catalina"

C:\Program Files (x86)\Catalina>

```

Launching Catalina this way means your Catalina environment has been set up for you.

If you want to start the command window another manually, or use Catalina in an existing command window, there is a batch file provided that you can use to do this¹ - just enter the following in an ordinary command window:

```
"%LCCDIR%" \use_catalina
```

Note that the quotation marks are required here because the path to the Catalina directory (which is what the LCCDIR environment variable contains) usually contains a space. The **use_catalina** command adds the appropriate Catalina directories to the Windows path, and then displays the values of various environment variables that Catalina uses. Initially, most of these variables will have no value set (indicated by the **[default]** value shown above) – this is quite normal. The use of the environment variables is optional. Examples of their use are given later in this document, but for the moment we can simply ignore them.

There is also a command you can use to display your current Catalina environment:

```
catalina_env
```

This is what actually produces the output shown above. If it does not then it probably means you have not yet set up your Catalina environment (e.g. via the **use_catalina** command).

Compiling your first Catalina Program

After opening a Catalina Command Line window, we will first copy the Catalina *demos* directory to your own user directory and then go to that directory (to build programs in the Catalina installation directories, we would need to have

¹ If you did not allow Catalina to set **LCCDIR** on installation, or you want to use a different version of Catalina than the default one, then you must first set the **LCCDIR** environment variable - e.g:

```
set LCCDIR=C:\Program Files (x86)\Catalina_X.Y\
"%LCCDIR%" \use_catalina
```

Note that Windows is not very consistent about using quotes - the **set** command must not have them, but the **use_catalina** command must have them.

administrator permissions).

To do this on Windows you can use the following commands:

```
xcopy /e /i "%LCCDIR%\demos" "%HOMEPATH%\demos\"  
cd "%HOMEPATH%\demos"
```

On Linux you would say something like:

```
cp -R /opt/catalina/demos ~/demos  
cd ~/demos
```

Now let's compile our first Catalina program. The traditional first C program is called *hello_world.c*. Initially, we'll get some errors when we compile it – this is deliberate. The tutorial will then go on to explain what the errors mean and how to fix them.

To invoke the Catalina compiler to compile *hello_world.c*, type:

```
catalina hello_world.c
```

If you see a message similar to the following, don't worry – it means everything is working correctly!

```
Undefined or Redefined symbols:  
printf undefined
```

However, if instead you see a message similar to this:

```
'catalina' is not recognized as an internal or external command,  
operable program or batch file
```

then check that Catalina is installed correctly, that your Catalina environment is set up correctly (use the command **catalina_env** to check).

If instead you see a message similar to this:

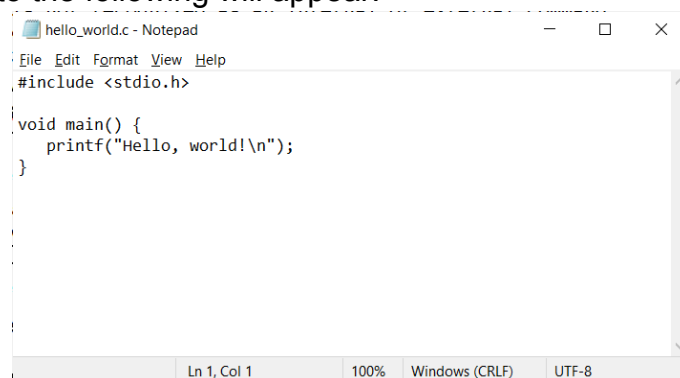
```
catalina: can't find "hello_world.c"
```

then check that you have typed the file name correctly, and that you are in the correct directory – use the **dir** command to check that the file *hello_world.c* actually exists in the current directory.

Let's assume everything worked as expected – i.e. you got the expected message about **printf** being undefined. To understand why we get this error message, let's examine the program we just tried to compile. To do this type:

```
notepad hello_world.c
```

A window similar to the following will appear:



This document is not intended to be a C programming tutorial, but note that the

program apparently calls a function called **printf** that does not appear to be defined anywhere within this program – this is because **printf** is a standard C library function. We need to tell Catalina to fetch this function from the appropriate library.

To tell Catalina to include library functions when compiling, we add a command-line option to our compile command. Traditionally, the standard C library is called **libc**, and including it is done by adding the command line option **-lc** to the compile command (here the **-l** means *library*, and the **c** means to look for a library called **libc**. Similarly, we will later use options like **-lcx**, which means look for a library called **libc_x**, **-lci**, which means look for a library called **libc_i**, or **-lmb** which means look for a library called **lib_mb**)².

So to tell Catalina where to find **printf** when we compile our program, we will add the **-lc** option. So now enter the following command:

```
catalina hello_world.c -lc
```

This time, instead of the error message you should see some information messages, about the program size, similar to those shown below:



```
Catalina Command Line
C:\Users\rosch\demos>catalina hello_world.c -lc
Catalina Compiler 4.9
code = 17452 bytes
cnst = 144 bytes
init = 216 bytes
data = 1068 bytes
file = 24016 bytes

C:\Users\rosch\demos>
```

The numbers tell you the sizes of the various C program segments, as well as the final program size:

```
code = 17452 bytes - the size of the code segment,
cnst = 144 bytes  - the size of the constant data segment,
init = 216 bytes  - the size of the initialized data segment
data = 1068 bytes - the size of the uninitialized data segment
file = 24016 bytes - the size of the final executable
```

These sizes may seem very large for such a simple program. Don't worry – we will address this issue in a later section.

The binary file we just produced could be executed by a suitably configured Propeller platform – but we won't try to execute it yet, because there is more we need to learn about Catalina before we can be sure the file will execute correctly on any *particular*

² Some C compilers will automatically find the standard C libraries, but when using Catalina it is necessary to explicitly specify when to include library functions, and which libraries to look in to find them. The main reason for this is that Catalina has multiple versions of the standard libraries. For example, **libc_x** is an extended version of the standard C library with additional file system support, and **libc_i** is an integer-only version of the standard C library without floating point support in stdio. Another example are the libraries **lib_m**, **lib_ma** and **lib_mb** – these are all variations of the same maths library which use different numbers of cogs at run-time.

Propeller platform.

So first we'll carry on with some more compiler usage basics.

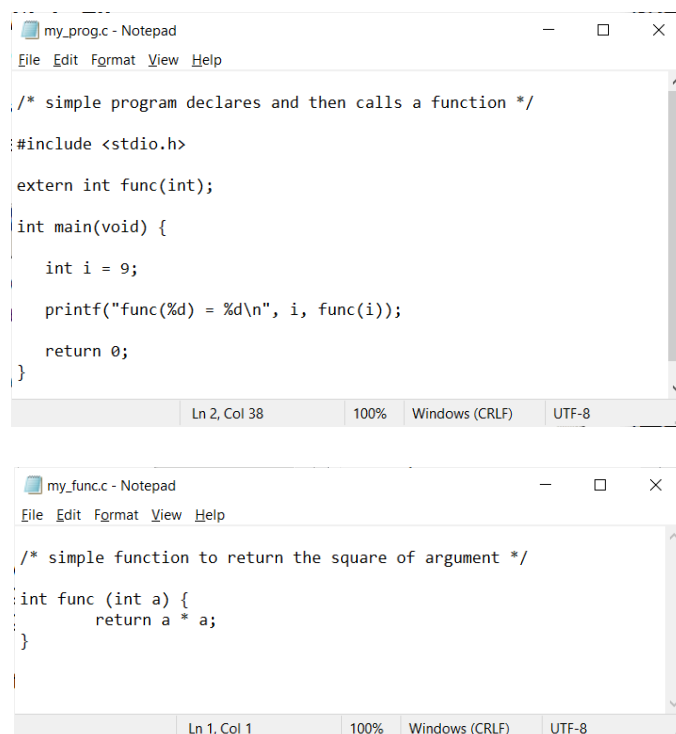
Compiling Multiple Files

So far, we have only compiled a program that consists of a single source file (i.e. the file *hello_world.c*). But C programs often consist of many source files. For example, there are two files called *my_prog.c* and *my_func.c* in the demo directory.

To see them, type the following commands:

```
notepad my_prog.c
notepad my_func.c
```

You will see windows similar to the following:



If you try compiling either one of these files separately, you will simply get errors (try it if you like – compiling *my_func.c* will generate a message about **main** being undefined, and compiling *my_prog.c* will generate a message about **func** being undefined – and also **printf** unless you include the **-lc** command line option).

To get this program to compile correctly, there are three different approaches that can be taken:

1. *Compile both the main program and function files using the same command.* To do this, type:

```
catalina my_prog.c my_func.c -lc
```

The program should compile successfully using this command.

Note that you can specify many files (not just two) in the same command, and they can appear in any order – but one (and only one!) of the files must contain a **main** function.

2. *Compile the programs without linking, and then link them separately.*

To do this, type:

```
catalina -c my_prog.c
catalina -c my_func.c
catalina -o my_prog my_prog.obj my_func.obj -lc
```

The program should compile successfully using these commands. The first two commands compile their respective C programs but do not attempt to link them together (this is what the `-c` option means). There is no checking done at this point for any undefined symbols. The outputs of these two commands are “obj” files – which are in fact not binary files, but are assembly language files (Catalina is different to traditional compilers in this respect) that require further processing.

The final command does no compilation – instead it just links the two previously generated object files together, and also links the result with the standard library and generates the final program binary. The object files can be deleted once the program has been built.

3. *Create a library containing the function, and then tell Catalina to use this library when compiling the main program.*

The commands to create libraries are a little more complex, and are usually not worthwhile for a program that consists of only a small number of files – but if you have many files it is often worth going to the extra effort to create a library.

Each library requires its own subdirectory, which must all be in a subdirectory called **lib** - i.e. **lib\name** – in this case let’s assume we want to refer to the library using the name **x**. To do this we must create a subdirectory called **lib\x**, as follows:

```
mkdir lib\x
```

Now enter the subdirectory. While not strictly necessary, this makes the next few commands much simpler to type (and also simpler to explain). To do this, enter:

```
cd lib\x
```

Now compile the file *my_func.c* from the original directory, adding the `-S` command-line option. The `-S` option (note that this is a *capital* S) tells Catalina to compile the program, but not to try to link it or generate an executable output file – it will just leave the result as an assembly language file (with a `.s` extension).

The command to use is therefore³:

```
catalina -S ../..\my_func.c
```

This produces a file called *my_func.s*. If you had other functions to add to the library, you would use a similar command for each function.

The last step in creating the library is to create a library index. The compiler expects each library to have an index called *catalina.idx* – this allows symbols within the library to be located and resolved by the Catalina Binder without having

³ Note that you can also use `-c` in place of `-S`, which produces `.obj` files instead of a `.s` file. This is because both are just assembly language versions of the C file - for historical reasons (i.e. there being no object format defined for the original Propeller) Catalina does not have a separate binary object format.

to laboriously re-process each file in the library every time.

The Catalina Binder (which is normally invoked silently for you by Catalina) is also the tool you use to do library management. To create an index of all symbols imported or exported by all the assembly language files in the current directory, enter the following command:

```
catbind -i -e *.s -o catalina.idx
```

Now your library is complete. Go back to the original directory:

```
cd ../../
```

We are now ready to compile the main program, specifying that Catalina should look in the library we have just created to find a definition for any unresolved symbols – which we do by adding the option **-lx** to the Catalina command. So enter the command:

```
catalina my_prog.c -lc -lx
```

The program should now compile without errors.

One final note – if the symbol **func** was one that was defined *both* in **libc** and in **libx**, Catalina would report that the symbol was being **redefined**. Catalina makes no assumptions about which definition of the symbol you actually want – the compilation simply fails in all such cases, and you have to decide yourself which library you want to include.

Specifying Memory Models and Load Options

So far, every program we have compiled has resulted in a Catalina program that could be loaded and executed on just about any Propeller 1 with a 5 Mhz clock. In each case the C program, the libraries, and all the necessary start-up and run-time code are combined into a single executable that can either be loaded into the Propeller using the normal tools, or programmed into a 32 kb external EEPROM connected to the Propeller (to be loaded into the Propeller automatically by the Propeller on boot – just like a SPIN program).

However, Catalina supports several different types of memory model, and several different methods of loading/executing programs.

The memory models supported are:

TINY This model is supported on both the Propeller 1 and Propeller 2. This is Catalina's default model on the Propeller 1. **TINY** programs can be up to 32 kb in size on the Propeller 1, or 512 kb on the Propeller 2. This model corresponds to the normal Parallax mode used for SPIN or PASM programs.

TINY programs can be programmed into EEPROM, or loaded using *any* program loader, including the Parallax **Propeller Tool**, the Parallax **Propellant** loader, Catalina's **Catalyst** SD card loader or Catalina's **Payload** serial loader.

All the programs we have compiled to this point have resulted in TINY programs.

COMPACT This model is supported on both the Propeller 1 and Propeller 2.

COMPACT programs can be up to 32 kb in size on the Propeller 1, or 512 kb on the Propeller 2. This model generates very compact C programs that use a hybrid of LMM and interpreted code.

TINY programs can be loaded using the Parallax **Propeller Tool**, the Parallax **Propellant** loader, or Catalina's **Payload** serial loader. They can be programmed into EEPROM as EMM programs or loaded onto SD CARD as SMM programs.

SMALL This model is supported on Propeller 1 and Propeller 2 platforms that have external memory, or a Propeller 1 with an EEPROM larger than 32 kb available. SMALL programs can have code segments up to 16 Mb in XMM RAM, but all data segments, the stack and the heap space must fit into the normal Propeller 32 kb of Hub RAM.

SMALL programs can be loaded into EEPROM, or they can be loaded with Catalyst or Payload.

LARGE This model is supported on Propeller 1 and Propeller 2 platforms that have external memory available. **LARGE** programs can have code and data segments and heap space up to a total of 16 Mb in XMM RAM – the Hub RAM is used only for stack space.

LARGE programs can be loaded into EEPROM, or they can be loaded with Catalyst or Payload.

NATIVE This model is supported only on the Propeller 2, and is the default model on that platform. **NATIVE** programs use the new Propeller 2 Hub execution mode. Such programs can be up to 512 kb in size. The NATIVE mode can only be used in conjunction with the **-p2** option, which tells Catalina to compile programs for the Propeller 2.

Currently, the Propeller 1 platforms with supported XMM RAM include the DracBlade, RamBlade, TriBladeProp (CPU #1 or CPU #2), the Hydra and Hybrid (using the HX512 Xtreme SRAM card), the C3, and Propellers using the FlashPoint SuperQuad and RamPage modules. On other Propeller 1 platforms, Catalina can only be used to compile TINY programs (or SMALL programs on Propeller 1 platforms with an EEPROM larger than 32 kb).

COMPACT, **TINY** and **NATIVE** programs do not need any special load processing to be loaded into Hub RAM. They are ordinary Parallax-compatible binaries and can be loaded using any Parallax-compatible program loader.

On the Propeller 1, there are special load options that can be specified during the program compilation. These load options are:

EEPROM COMPACT and TINY programs do not *need* any special processing to be executed from EEPROM, but there is a special loader that allows them to be loaded in two phases, allowing such programs to make more effective use of the Propeller Hub RAM. For SMALL or LARGE programs to be executed from EEPROM, they *must* be compiled with a special Catalina target that knows how to load the XMM portion of the program from EEPROM to XMM RAM. The “special” Catalina targets are loaded into the *first* 32K of the EEPROM. The program is then loaded into the EEPROM starting at 32kb. The target loads all the

plugins and drivers, and then loads the Catalina C program for execution –the advantage of using the special target (even for TINY programs) is that provided you have an EEPROM of at least 64Kb connected to your propeller, the C program code does not need to share the same 32kb space as the target code – allowing for larger C programs to be loaded and executed.

SDCARD The Catalyst program is Catalina's SD Card Loader. COMPACT, TINY, SMALL and LARGE programs do not *need* any special processing to be able to be loaded by Catalyst – but COMPACT and TINY programs can be loaded using a special loader that that allows them to be loaded in two phases, allowing such program to make more effective use of the limited Hub RAM. Like the special EEPROM loader, the SDCARD loader divides programs into several sections – the first 32kb contains the target, which sets up the execution environment for the program (by loading all the plugins and drivers etc). Then the *second* 32kb of the file contains the application program. Finally, the actual kernel is loaded from the last 2kb of the file. This means all program files compiled with the SDCARD loader option are exactly 66 kb in size.

FLASH SMALL and LARGE programs that are to be executed out of SPI Flash RAM require a special load process, and so they must also use a special Catalina target. Both the Catalyst and the Payload loaders know how to load and run FLASH programs.

You select whether you want your program compiled as a **COMPACT**, **TINY**, **SMALL**, **LARGE** or **NATIVE** program, and also (on the Propeller 1) whether to use the **EEPROM**, **SDCARD** or **FLASH** loader by defining these Catalina symbols on the command line. Defining Catalina symbols is done using the command line option **-C**. Note that Catalina symbols are different from C symbols, which can also be defined on the command line using the **-D** option⁴. Some examples are given below.

The default when no symbols are defined (which is what we have done so far) is for programs to be compiled using the **TINY** memory model (Propeller 1) or **NATIVE** memory model (Propeller 2), and to use a target compatible with the standard Parallax loader (as well as Catalina's own loaders).

Below are examples of all the possible combinations of memory model and loader option:

TINY

To compile a TINY program that can be loaded with Catalyst, Payload (or any other serial load program) or which can be programmed into EEPROM:

```
catalina hello_world.c -lc
```

To compile a TINY program that uses the special EEPROM loader:

```
catalina hello_world.c -lc -C EEPROM
```

To compile a TINY program that uses the special SDCARD loader, to be loaded

⁴ It may seem more logical to use the **-C** command line option for defining C symbols and some other letter for defining Catalina symbols, but **-D** is the de-facto standard that C compilers use for defining C symbols, so Catalina does the same, and instead uses **-C** for Catalina symbols. See the **Propeller Reference Manual** for more details on C and Catalina symbols.

using Catalyst:

```
catalina hello_world.c -lc -C SDCCARD
```

Note that you can also include the **-C TINY** option – but it is not necessary on the Propeller 1 as this is the default. It can be specified on the Propeller 2 to force **TINY** mode, but **NATIVE** mode is a much better option on the Propeller 2.

SMALL

To compile a **SMALL** program that uses a loader compatible with Catalyst and Payload:

```
catalina hello_world.c -lc -C SMALL
```

To compile a **SMALL** program that can be programmed into EEPROM (Propeller 1 only ⁵):

```
catalina hello_world.c -lc -C SMALL -C EEPROM
```

To compile a **SMALL** program that can be loaded and executed from FLASH (Propeller 1 only ⁶):

```
catalina hello_world.c -lc -C SMALL -C FLASH
```

LARGE

To compile a **LARGE** program that uses a loader compatible with Catalyst and Payload:

```
catalina hello_world.c -lc -C LARGE
```

To compile a **LARGE** program that can be programmed into EEPROM (Propeller 1 only ⁷):

```
catalina hello_world.c -lc -C LARGE -C EEPROM
```

To compile a **LARGE** program that can be loaded and executed from FLASH (Propeller 1 only ⁸):

```
catalina hello_world.c -lc -C LARGE -C FLASH
```

COMPACT

To compile a **COMPACT** program that can be loaded with Payload (or any other

⁵ The Propeller 2 has no EEPROM.

⁶ The Propeller 2 also has FLASH RAM, but using it does not require the FLASH option to be specified. Instead, it uses a special load script. For example, to compile and load the program *hello.bin* into FLASH, you would use commands like:

```
catalina -p2 hello.c
flash_payload hello.bin -o2
```

Note the use of **-p2** in the **catalina** command, and **-o2** in the **payload** command. The **-p2** option means to compile for the Propeller 2, and the **-o2** option means to look for a Propeller 2 “.bin” file and use the default baud rate for a Propeller 2 (230400 baud). Note that on the Propeller 2 you must have the microswitches set correctly to program and execute from FLASH. See the **Catalina Reference Manual (Propeller 2)** for more details.

⁷ See previous footnotes.

⁸ See previous footnotes.

serial load program):

```
catalina hello_world.c -lc -C COMPACT
```

To compile a **COMPACT** program that uses the special EEPROM loader:

```
catalina hello_world.c -lc -C COMPACT -C EEPROM
```

To compile a **COMPACT** program that uses the special SDCARD loader, to be loaded using Catalyst:

```
catalina hello_world.c -lc -C COMPACT -C SDCCARD
```

NATIVE

NATIVE mode is supported only on the Propeller 2. To compile a **NATIVE** program that can be loaded with Catalyst, Payload (or any other serial load program) on the Propeller 2:

```
catalina -p2 hello_world.c -lc
```

Note that you can include the **-C NATIVE** option – but it is not necessary on the Propeller 2 as this is the default.

You will find a complete summary of all Catalina command line options, including all the symbols that affect the compilation process, in the document **Catalina Command Summary**. It may be worth printing this out and keeping it handy!

Specifying a Supported Propeller 1 Platform

So far, we have just been compiling programs but not specifying anything about the particular Propeller hardware we have. How do we do that?

Behind the scenes on each compile, Catalina has been adding in the run-time support infrastructure required to make each compiled program a self-contained executable. But so far, all programs have been compiled only for a default Propeller platform (which is the **CUSTOM** platform, suitable for just about any Propeller 1 with a 5 Mhz clock).

If you have a Hydra, Hybrid, or another platform that does not have a 5 Mhz clock, then a bit more work needs to be done to make sure the programs are correctly configured for *your particular* propeller platform.

Recall that when programming in SPIN (or PASM) you also need to specify things such as clock frequency, and the I/O base pins (or pin groups) for device drivers to use. You need to do much the same in C – but with Catalina it is done by selecting the appropriate *target* – and then further configuring that target by using various command line options to specify the *platform*.

Catalina *targets* achieve the same result as setting the various constants such as `_clkfreq` and `_xinfreq` in SPIN – plus a lot more. A Catalina target not only sets these basic values, it also loads and initializes all the device drivers and other support infrastructure (such as floating point libraries or a real-time clock) that the C program will need – so that when the program starts executing it has a well-defined environment in which to do so. In SPIN, you have to set all this up for yourself each time – but not with Catalina!

The most important characteristics of each Catalina target are the Propeller hardware configuration parameters it defines, the device drivers it loads, and the type of kernel it uses to execute the Catalina program.

Since we have so far not been specifying any particular target or platform to use, Catalina has been using a *default* target when generating executables, and configuring that target for the default CUSTOM platform. Specifically, we have so far mostly been using the *lmm_default* target – which (if no further options are specified) does the following:

- Sets the clock speed, and I/O pin configurations for the Hydra.
- Loads a serial driver for output.
- Loads a serial driver for input.
- Loads a HMI driver (to manage the text output and keyboard input)
- Loads the LMM Kernel

But there are some programs – such as the *test_time.c* demo program – that also need other drivers (in this case a real-time clock) which are not loaded by the default target.

To support such programs we can either explicitly request Catalina to use a target that loads necessary drivers, or to request Catalina to enable the real-time clock in the default target.

The Propeller 1 targets provided with Catalina are described in detail in the **Catalina Reference Manual (Propeller 1)** – in this particular case we will compile the program to use the predefined default target and simply enable the real-time clock support. To do this, we define the special symbol **CLOCK** on the Catalina command line. Defining symbols on the command line is done using the **-C** option, so we would enter a command like:

```
catalina test_time.c -lc -C CLOCK
```

This will generally work on any Propeller 1 with a 5Mh clock. If you intend to execute the program on a platform that uses a different clock speed (such as the **HYDRA** or **HYBRID**) you may also need to define other symbols. For example, to compile the program for the HYDRA we would define an additional Catalina symbol, using a command such as:

```
catalina test_time.c -lc -C CLOCK -C HYDRA
```

Catalina has build in support for many common Propeller 1 platforms, including the **HYDRA**, the **HYBRID**, the Parallax **DEMO** board, the Parallax **FLIP** module, and the **TRIBLADEPROP**, **RAMBLADE**, **RAMBLADE3**, **ASC**, **C3** or **QUICKSTART** boards. It

also supports Propeller 1 platforms that use the **SUPERQUAD**, **RAMPAGE**, **HX512**, **RP2** or **PMC** XMM Add-on boards.

Here is a list of Catalina symbols that can be defined on the command line which affect the configuration of the default Propeller 1 target package (note that some symbols only apply to specific targets, or memory modes):

HYDRA	use HYDRA pin definitions, drivers and XMM functions
HYBRID	use HYBRID pin definitions, drivers and XMM functions
DEMO	use DEMO board pin definitions and drivers (this platform has no XMM support)
DRACBLADE	use DRACBLADE pin definitions, drivers and XMM functions

RAMBLADE	use RAMBLADE pin definitions, drivers and XMM functions
RAMBLADE3	use RAMBLADE3 pin definitions, drivers and XMM functions
TRIBLADEPROP	use TRIBLADEPROP pin definitions, drivers and XMM functions
FLIP	use FLIP module pin definitions and drivers (this platform has no XMM support)
ASC	use ASC board pin definitions and drivers (this platform has no XMM support)
QUICKSTART	use QUICKSTART board pin definitions and drivers (this platform has no XMM support)
C3	use C3 pin definitions, drivers and XMM functions
SUPERQUAD	use SUPERQUAD pin definitions, drivers and XMM functions
RAMPAGE	use RAMPAGE pin definitions, drivers and XMM functions
RP2	use RAMPAGE 2 pin definitions, drivers and XMM functions
PMC	use PROPELLER MEMORY CARD pin definitions, drivers and XMM functions
PP	use PROPELLER PLATFORM pin definitions, drivers and XMM functions
CUSTOM	use a user-customized set of pin definitions, drivers and XMM functions (if applicable)
CPU_1	on the TRIBLADEPROP, this means to use CPU #1 pin definitions and XMM functions – if not specified, CPU #2 XMM functions are used by default
CPU_2	on the TRIBLADEPROP, this means to use CPU #2 pin definitions and XMM functions (this is also the default)
CPU_3	on the TRIBLADEPROP, this means to use CPU #3 pin definitions and devices.
ALTERNATE	use the alternate LMM Kernel (the alternate kernel is slightly smaller in size but does not include any floating point support).
TINY	compile an LMM program to use the TINY memory model
SMALL	compile an XMM program to use the SMALL memory model
LARGE	compile an XMM program to use the LARGE memory model
COMPACT	compile a CMM program to use the COMPACT memory

	model
SDCARD	use the SDCARD two-phase loader
EEPROM	use the EEPROM two-phase loader
FLASH	use an SPI FLASH loader
HIRES_VGA	load a High Resolution VGA driver (not supported on the HYBRID or RAMBLADE or RAMBLADE3)
LORES_VGA	load a Low Resolution VGA driver (not supported on the RAMBLADE or RAMBLADE3)
VGA	(same as LORES_VGA)
HIRES_TV	load a High Resolution TV driver (not supported on RAMBLADE or RAMBLADE3)
LORES_TV	load a Low Resolution TV driver (not supported on RAMBLADE or RAMBLADE3)
TV	(same as LORES_TV)
NTSC	use NTSC mode (TV drivers only)
NO_INTERLACE	use non-interlace mode (TV drivers only)
PC	load a PC terminal emulator HMI plugin with screen and keyboard support
PROPTERMINAL	load a PropTerminal HMI plugin with screen, keyboard and mouse support.
TTY	load a simple serial HMI plugin with screen and keyboard support (no proxy support).
VGA_640	load a VGA HMI plugin with resolution of 640 x 480.
VGA_800	load a VGA HMI plugin with resolution of 800 x 600.
VGA_1024	load a VGA HMI plugin with resolution of 1024 x 768
HIRES_VGA	same as VGA_1024
LORES_VGA	same as VGA_640
COLOR_1	use a color depth of 1 bit.
COLOR_4	use a color depth of 4 bits.
COLOR_8	use a color depth of 8 bits.
MONO	same as COLOR_1.
CR_ON_LF	Translate LF to CR LF on output
NO_CR_TO_LF	Disable translation of CR to LF on input
NON_ANSI_HMI	Disable ANSI compliance in HMI (revert to prior behavior)
CLOCK	load a Real-Time Clock plugin
SD	load the SD plugin (this is not usually required, since it is implied by the -lcx and -lcix options)

NO_FP	do not load any Floating Point plugins (even if implied by other options)
NO_FLOAT	same as NO_FP
NO_HMI	do not load any HMI plugin (even if implied by other options)
NO_MOUSE	do not start a mouse driver (even if one is loaded)
NO_KEYBOARD	do not start a keyboard driver (even if one is loaded)
NO_SCREEN	do not start a screen driver (even if one is loaded)
PROXY_SD	See the Proxy Devices section later in this document
PROXY_SCREEN	See the Proxy Devices section later in this document
PROXY_MOUSE	See the Proxy Devices section later in this document
PROXY_KEYBOARD	See the Proxy Devices section later in this document
CACHED_1K	Use a 1K cache for XMM access
CACHED_2K	Use a 2K cache for XMM access
CACHED_4K	Use a 4K cache for XMM access
CACHED_8K	Use a 8K cache for XMM access
CACHED	Same as CACHED_8K
GAMEPAD	Include the Gamepad driver
NO_ARGS	disable C command line argument processing (saves some Hub and Cog RAM in programs not intended to be run from a Catalyst command line).
NO_ENV	disable the processing of environment variables (saves some Hub and Cog RAM in programs that do not need to access Catalyst environment variables).
NO_EXIT	Disable the generation of code to handle the main() function exiting (if it never does)
NO_REBOOT	Disable the automatic reboot if the program exits from the main() function – useful if you are using the TV or VGA driver and want the screen output to remain when the program exits
VGA_640	Use 640 x 480 resolution (virtual graphics only)
VGA_800	Use 800 x 600 resolution (virtual graphics only)
VGA_1024	Use 1024 x 768 resolution (virtual graphics only)
VGA_1152	Use 1152 x 864 resolution (virtual graphics only)
VGA_2_COLOR	Use 2 color mode (1 bit color depth, virtual graphics)
VGA_4_COLOR	Use 4 color mode (2 bit color depth, virtual graphics)
DOUBLE_BUFFER	Smoother graphics (CGI or virtual graphics only)

Specifying a Supported Propeller 2 Platform

So far, we have just been compiling programs but not specifying anything about the particular Propeller hardware we have. How do we do that on the Propeller 2?

Behind the scenes on each compile, Catalina has been adding in the run-time support infrastructure required to make each compiled program a self-contained executable. But so far, all programs have been compiled only for a default Propeller platform (which is essentially the **P2_EDGE** platform).

If you have a Propeller 2 platform, then a bit more work needs to be done to make sure the programs are correctly configured for *your particular* propeller platform.

Recall that when programming in SPIN (or PASM) you also need to specify things such as clock frequency, and the I/O base pins (or pin groups) for device drivers to use. You need to do much the same in C – but with Catalina it is done by selecting the appropriate *target* – and then further configuring that target by using various command line options to specify the *platform*.

Catalina *targets* achieve the same result as setting the various constants such as clock frequency and mode – plus a lot more. A Catalina target not only sets these basic values, it also loads and initializes all the device drivers and other support infrastructure (such as floating point libraries or a real-time clock) that the C program will need – so that when the program starts executing it has a well-defined environment in which to do so. In SPIN, you have to set all this up for yourself each time – but not with Catalina!

The most important characteristics of each Catalina target are the Propeller hardware configuration parameters it defines, the device drivers it loads, and the type of kernel it uses to execute the Catalina program.

Since we have so far not been specifying any particular target or platform to use, Catalina has been using a *default* target when generating executables, and configuring that target for the default platform. Specifically, we have so far mostly been using the *nmm_default* target – which (if no further options are specified) does the following:

- Sets the clock speed, and I/O pin configurations (as for the P2_EDGE).
- Loads a serial driver for output.
- Loads a serial driver for input.
- Loads a HMI driver (to manage the text output and keyboard input)
- Loads the NMM Kernel

But there are some programs – such as the *test_time.c* demo program – that also need other drivers (in this case a real-time clock) which are not loaded by the default target.

To support such programs we can either explicitly request Catalina to use a target that loads necessary drivers, or to request Catalina to enable the real-time clock in the default target.

The Propeller 2 targets provided with Catalina are described in detail in the **Catalina Reference Manual (Propeller 2)** – in this particular case we will compile the program to use the predefined default target and simply enable the real-time clock support. To do this, we define the Catalina symbol **CLOCK** on the Catalina command line. Defining symbols on the command line is done using the **-C** option,

so we would enter a command like:

```
catalina -p2 test_time.c -lc -C CLOCK
```

This will use the **P2_CUSTOM** platform, which (unless modified) is essentially the same as the **P2_EDGE**. To compile the program for another platform such as the **P2_EVAL** we would define an additional symbol, using a command such as:

```
catalina -p2 test_time.c -lc -C CLOCK -C P2_EVAL
```

Catalina has built in support for several Propeller 2 Platforms, including the **P2_EVAL**, **P2D2** and the **P2_EDGE**. The default is to use the **P2_CUSTOM** platform.

Here is a list of symbols that can be defined on the command line which affect the configuration of the default Propeller 2 target package (note that some symbols only apply to specific targets, or memory modes):

P2	compile for the Propeller 2 (defining this symbol is the same as specifying -p2 on the catalina command line).
P2_EVAL	use P2_EVAL pin definitions, drivers and defaults.
P2_EDGE	use P2_EDGE pin definitions, drivers and defaults.
P2D2	use P2D2 pin definitions, drivers and defaults.
P2_CUSTOM	use a user-customized set of pin definitions, drivers and default values (if applicable).
TINY	compile an LMM program to use the TINY memory model.
COMPACT	compile a CMM program to use the COMPACT memory model.
NATIVE	compile an NMM program to use the NATIVE memory model.
TTY	load a serial HMI plugin with screen and keyboard support (no proxy support).
SIMPLE	load a simple serial HMI plugin with screen and keyboard support (no proxy support). May not support as high a serial baud rate as the TTY plugin.
CR_ON_LF	Translate LF to CR LF on output.
NO_CR_TO_LF	Disable translation of CR to LF on input.
CLOCK	load a software Real-Time Clock plugin.
RTC	load a plugin that knows how to use the Parallax Real-Time Clock add-on board.
SD	load the SD plugin (this is not usually required, since it is implied by the -lcx and -lcix options).
P2_REV_A	use instructions supported by the P2 Rev A chip (required for Rev A chips by some HMI drivers).
MHZ_175	use a clock frequency of 175 Mhz.
MHZ_200	use a clock frequency of 200 Mhz (recommended for

	building Catalyst).
MHZ_220	use a clock frequency of 220 Mhz (required for High resolution VGA).
MHZ_260	use a clock frequency of 260 Mhz (required for High resolution VGA using 4 bit or 8 bit color).
MHZ_300	use a clock frequency of 300 Mhz (may not work on some Propeller 2 boards).
NO_KEYBOARD	do not load the keyboard code and one USB driver
NO_MOUSE	do not load the mouse code and one USB driver
P2_REV_A	use instructions supported by the P2 Rev A chip (required for Rev A chips).
NO_FP	do not load any Floating Point plugins (even if implied by other options).
NO_FLOAT	same as NO_FP
NO_HMI	do not load any HMI plugin (even if implied by other options).
NO_ARGS	disable C command line argument processing (saves some Hub and Cog RAM in programs not intended to be run from a Catalyst command line).
NO_ENV	disable the processing of environment variables (saves some Hub and Cog RAM in programs that do not need to access Catalyst environment variables).
NO_EXIT	Disable the generation of code to handle the main() function exiting (if it never does).
NO_REBOOT	Disable the automatic reboot if the program exits from the main() function – useful if you are using the TV or VGA driver and want the screen output to remain when the program exits.
VGA_640	load a VGA HMI plugin with resolution of 640 x 480.
VGA_800	load a VGA HMI plugin with resolution of 800 x 600.
VGA_1024	load a VGA HMI plugin with resolution of 1024 x 768
HIRES_VGA	same as VGA_1024
LORES_VGA	same as VGA_640
VGA	same as VGA_640
COLOR_1	use a color depth of 1 bit.
COLOR_4	use a color depth of 4 bits.
COLOR_8	use a color depth of 8 bits.
COLOR_24	use a color depth of 24 bits.
MONO	same as COLOR_1

NO_KEYBOARD	do not load the keyboard code and one of the USB drivers.
NO_MOUSE	do not load the mouse code and one of the USB drivers.
NON_ANSI_HMI	Disable ANSI compliance in HMI (revert to prior behavior).
NO_FP	do not load any Floating Point plugins (even if implied by other options).
NO_FLOAT	same as NO_FP
NO_HMI	do not load any HMI plugin (even if implied by other options).
NO_MOUSE	do not start a mouse driver (even if one is loaded).
NO_KEYBOARD	do not start a keyboard driver (even if one is loaded).
NO_EXIT	Disable the generation of code to handle the main() function exiting (if it never does).
NO_REBOOT	Disable the automatic reboot if the program exits from the main() function – useful if you are using the TV or VGA driver and want the screen output to remain when the program exits.

A note about Catalina Code Sizes

Given the limited amount of RAM, especially on the Propeller 1 (32 kilobytes), it is easy to generate programs where the binary may initially seem to end up too large to be useful. We saw this with our initial attempt to compile *hello_world.c*. We now have enough basic knowledge of Catalina to do something to address this.

Having the ability to run Catalina programs from external memory (XMM) is one option, but not all platforms support XMM RAM. On platforms with only the in-built Hub RAM Catalina offers many alternatives to reduce the final program size.

First, remember the traditional C “Hello, world” program:

```
#include <stdio.h>
void main() {
    printf("Hello, world!\n");
}
```

To compile this program, we used a simple command such as:

```
catalina hello_world.c -lc
```

This compiles the program, links it with the **libc** version of C89 library and uses the default target and drivers, and the LMM kernel. Here are the statistics actually produced by the above command²:

```
code = 21088 bytes
cnst = 144 bytes
init = 184 bytes
data = 352 bytes
file = 26968 bytes
```

At first sight, it appears that this trivial program consumes nearly **27 kilobytes** of precious Hub RAM! But this can be substantially reduced in various simple ways.

For example, by using the **libci** library (in place of **libc**) for programs that do not require support for i/o of floating point numbers (they can still use floating point internally – they just can’t scan or print them) the program size immediately reduces from **20 kb** to around **12 kb**. To see this, compile the same program using this command instead:

```
catalina hello_world.c -lci
```

Here are the statistics⁹:

```
code = 6248 bytes
cnst = 104 bytes
init = 164 bytes
data = 4 bytes
file = 11720 bytes
```

This big difference in code size is due to how much code is required in the standard C library to perform I/O of floating point numbers.

If you don’t need the full capabilities of the Standard C I/O libraries, Catalina also provides smaller alternatives. In this case, we can use the “tiny” library, which

⁹ The exact values will vary depending on your version of Catalina, or if you have set up a different default platform or target, or are using CATALINA_DEFINE environment variable, or are compiling for a Propeller 1 or a Propeller 2.

provides smaller (but more limited) versions of the standard C I/O functions. We specify those in addition to the standard library:

```
catalina hello_world.c -lci -ltiny
```

Here are the statistics:

```
code = 3000 bytes
cnst = 16 bytes
init = 180 bytes
data = 4 bytes
file = 8400 bytes
```

For this program, several other optimizations are also possible, and can be selected on the command line. For example, try the following command:

```
catalina hello_world.c -lci -ltiny -C NO_ARGS -C NO_FLOAT
```

Here are the statistics:

```
code = 2964 bytes
cnst = 16 bytes
init = 176 bytes
data = 4 bytes
file = 7440 bytes
```

Our program size is now about **7 kilobytes**. But that's not the end of the story. Notice that the sum of the segments (`code`, `cnst`, `init` & `data`) is actually only about **3 kilobytes**? The other **4 kilobytes** of the final file size is taken up by plugins and the kernel itself. But we don't need to waste even this space – Catalina provides EMM (**EEPROM**) and SMM (**SDCARD**) 2-phase loaders which can make this space available as code space. So in reality, the program requires only **3 kilobytes** of Hub RAM *even though it includes a substantial portion of the stdio library* (which of course only ever needs to be included once). This leaves us around **29 kilobytes** of Hub RAM for more C code (or for other purposes). And that code does not need to include the stdio library code again!

To make the significance of this last point more evident, consider the following program, very similar to the traditional C “hello, world” program:

```
#include <catalina_hmi.h>
void main () {
  t_string(1, "Hello, world\n");
  while (1);
}
```

Let's compile this program with the following command (this program is also in the `demos` folder, called `hello_world_3.c`):

```
catalina hello_world_3.c -lci -C NO_ARGS -C NO_EXIT -C NO_FLOAT
```

Here are the statistics:

```
code = 224 bytes
cnst = 16 bytes
init = 8 bytes
data = 4 bytes
file = 4532 bytes
```

Excluding the “one off” Catalina runtime overheads (i.e. the Catalina kernel, the HMI plugin and various support functions) this program actually compiles to only around

250 bytes, even though when compiled and bound with a suitable target, it occupies about **4.5 kilobytes** of RAM – this is because (in this case) Catalina must also include at least the following:

the Catalina Kernel (~2 kilobytes);

a Catalina HMI plugin and drivers (~2.5 kilobytes);

But if we were to load this program using an SMM or EMM 2-phase loader, it requires *only the 250 bytes* of Hub RAM – leaving over **31 kilobytes for more code!**

The difference between the two “hello world” programs is that the first version uses the **printf** function from the C89 library, whereas the second uses a **t_string** function which is built into the Catalina HMI plugin. This small change means that Catalina does not need to load a significant part of the C library (i.e. **printf** and its associated support functions). In reality, this small difference between the two programs amounts to over a thousand lines of library C code. But many programs do not require the full functionality provided by **libc**, and some do not need to load it at all.

EMM targets can be used on any Propeller with a 64 kb EEPROM attached, and SMM targets can be used on any Propeller with an SD Card attached. While these techniques do not reduce the program size, they do mean that more of the available Hub RAM can actually be used by the C program.

As if this is not enough, there is also the Catalina Compact Memory Model (CMM). This dramatically reduces the size of any Propeller programs – typically by over 50% - but at a cost in reduced speed. The resulting programs will still execute faster than Spin, but slower than the equivalent LMM program.

Let's compile the same program in COMPACT mode:

```
catalina hello_world_3.c -lci -C NO_ARGS -C NO_EXIT -C NO_FLOAT -C COMPACT
```

Here are the statistics:

```
code = 124 bytes
cnst = 16 bytes
init = 8 bytes
data = 4 bytes
file = 4508 bytes
```

Finally, there is also the Catalina Code Optimizer that can reduce code sizes even further – typically 15 – 20%. For example, here are the statistics when both CMM and the Optimizer are used on the above program:

```
catalina hello_world_3.c -lci -C NO_ARGS -C NO_EXIT -C NO_FLOAT -C COMPACT -O5
```

Here are the statistics:

```
code = 108 bytes
cnst = 16 bytes
init = 4 bytes
data = 4 bytes
file = 4488 bytes
```

Our final Catalina C program, which implements the same functionality as our original program, may now use as little as 132 bytes of Hub RAM to execute.

We can use a combination of these code size reduction techniques with nearly all C programs.

Specifying Options using Environment Variables

You can specify multiple symbols on the command line, but remember to specify **-C** before each one. For example, to compile a program for the Hybrid using a high-resolution TV driver in NTSC mode with clock support you would use a command like:

```
catalina test_time.c -lc -C HYBRID -C HIRES_TV -C NTSC -C CLOCK
```

Because using multiple symbols to specify the configuration of the target is so common, there is a better way to specify them if you intend using the same configuration for many compilations – set the **CATALINA_DEFINE** environment variable, as follows:

```
set CATALINA_DEFINE=HYBRID HIRES_TV NTSC CLOCK
```

Then a command such as:

```
catalina test_time.c -lc
```

has the same effect as specifying all the symbols using **-C** options on the command line (note that the **-C** option should not be specified for symbols defined using the environment variable).

The **CATALINA_DEFINE** environment variable can also be used to specify that you want to compile for the Propeller 2 by default, simply by including the Catalina symbol **P2**. For example:

```
set CATALINA_DEFINE=P2 TTY CLOCK
```

Then a command such as:

```
catalina test_time.c -lc
```

will compile the program for the Propeller 2 without needing to always specify **-p2** on the command line.

Knowing the current setting of the various Catalina environment variables can be very important. To display the current settings, use the command **catalina_env** – in the above case you might see output like:

```
CATALINA_DEFINE    = P2 TTY CLOCK
CATALINA_INCLUDE   = [default]
CATALINA_LIBRARY   = [default]
CATALINA_TARGET    = [default]
CATALINA_LCCOPT     = [default]
CATALINA_TMPDIR     = [default]
LCCDIR              = C:\Program Files (x86)\Catalina
```

To unset an environment variable, you can use the unset command:

```
unset CATALINA_DEFINE
```

More details on environment variables is given in the **Catalina Reference Manual**.

Note that some Catalina build scripts (e.g. the **build_all** script in the *utilities* directory) need to set up very specific configurations before compiling some of the utility programs, and these scripts check to see if the **CATALINA_DEFINE** variable is set – if so, they refuse to run, because the settings could interfere with the operation of the script. To run these scripts, first unset the **CATALINA_DEFINE** variable.

Loading Programs

For **LMM**, **CMM** or **NATIVE** programs, the simplest way to load a program (assuming your platform has a suitable serial or USB interface, and it is connected to your PC, and the Propeller is powered up) is to use the Payload serial loader. Just enter a command like:

```
payload hello_world
```

Payload will locate the Propeller automatically. On the Propeller 1 you can also load programs into EEPROM, by including the **-e** command line option:

```
payload hello_world -e
```

Before we can load XMM programs on the Propeller 1 or Propeller 2, we must first build some Catalina utilities for the platform we want to load. To do this, go to the Catalina *utilities* directory, and enter the following command:

```
build_utilities
```

This is an interactive batch program which will ask you questions about the Propeller platform you intend to run the programs on. Once this process is complete, you can then return to your program folder and use the Payload program, but this time also specify that Payload should use the XMM loader utilities you just built. For example:

```
payload SRAM hello_world
```

or

```
payload FLASH hello_world
```

Examples of the use of all the payload utilities are given in the *demos/utilities* folder. See the *README.TXT* file in that folder for more detailed examples, or the **Catalina Reference Manual for the Propeller 1** for more technical details.

The payload program can also use environment variables to simplify the entry of payload commands:

PAYLOAD_PORT set this to a port number to use a specific serial port.

PAYLOAD_BAUD set this to a baud rate to use.

For example:

```
set PAYLOAD_PORT=4
set PAYLOAD_BAUD=230400
payload hello_world
```

does the same as:

```
payload hello_world -p4 -b230400
```

Supporting Other Propeller 1 Platforms

For other Propeller 1 platforms (i.e. any propeller platform not mentioned in the previous sections), you can modify the definitions in the files used for the **CUSTOM** platform – these files include the definition of the clock frequency and the base pin/pin group definitions etc. The full set of related files are in *target\p1*:

Custom_DEF.inc

Custom_CFG.inc

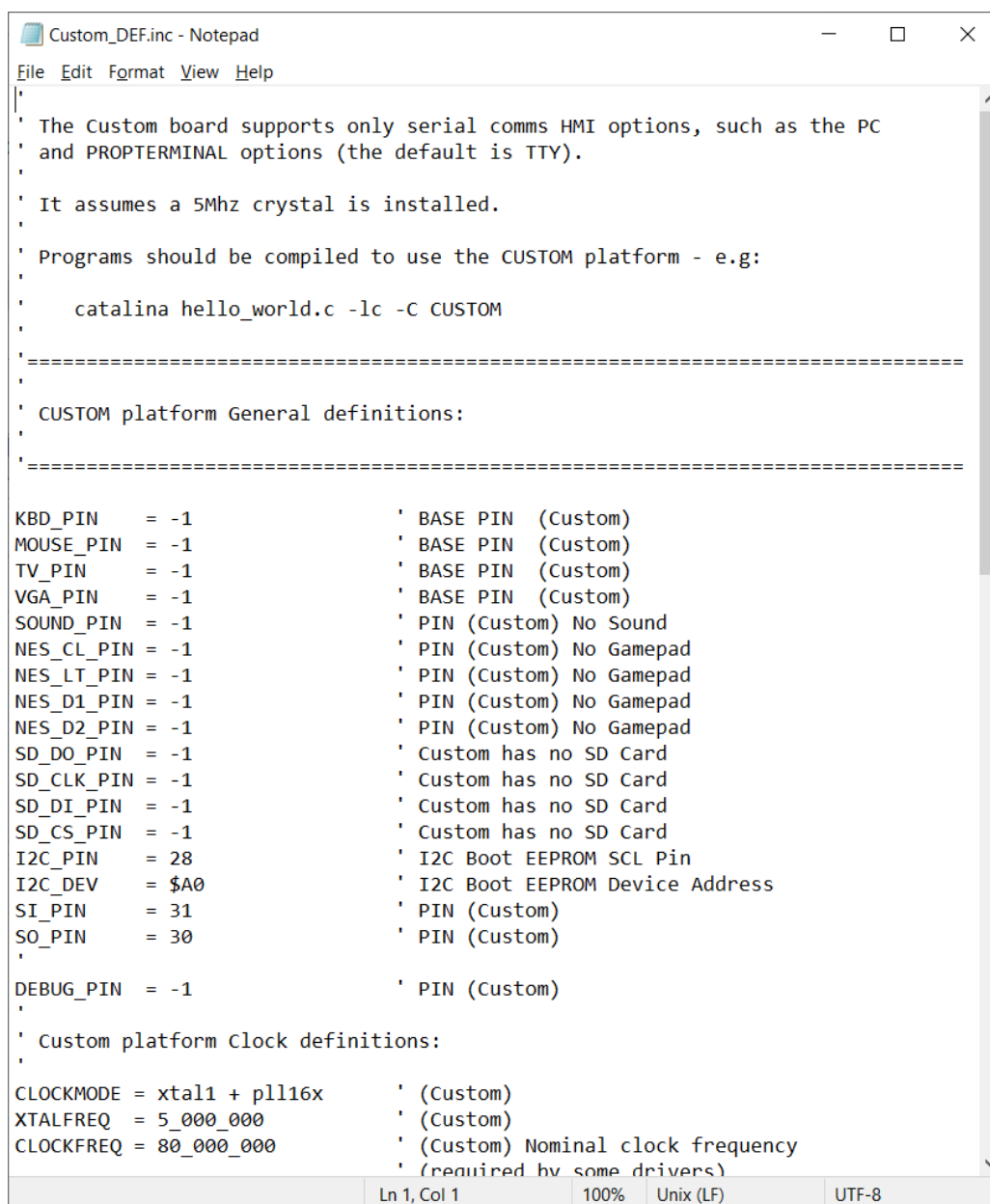
Custom_HMI.inc

Custom_XMM_DEF.inc.

Custom_XMM_CFG.inc

Custom_XMM.inc

For example, *Custom_DEF.inc* is a normal text file that can be edited using notepad. This file will look something like this:



```

Custom_DEF.inc - Notepad
File Edit Format View Help
'
' The Custom board supports only serial comms HMI options, such as the PC
' and PROPTERMINAL options (the default is TTY).
'
' It assumes a 5Mhz crystal is installed.
'
' Programs should be compiled to use the CUSTOM platform - e.g:
'
'     catalina hello_world.c -lc -C CUSTOM
'
' =====
'
' CUSTOM platform General definitions:
'
' =====

KBD_PIN      = -1          ' BASE PIN (Custom)
MOUSE_PIN    = -1          ' BASE PIN (Custom)
TV_PIN       = -1          ' BASE PIN (Custom)
VGA_PIN      = -1          ' BASE PIN (Custom)
SOUND_PIN    = -1          ' PIN (Custom) No Sound
NES_CL_PIN   = -1          ' PIN (Custom) No Gamepad
NES_LT_PIN   = -1          ' PIN (Custom) No Gamepad
NES_D1_PIN   = -1          ' PIN (Custom) No Gamepad
NES_D2_PIN   = -1          ' PIN (Custom) No Gamepad
SD_DO_PIN    = -1          ' Custom has no SD Card
SD_CLK_PIN   = -1          ' Custom has no SD Card
SD_DI_PIN    = -1          ' Custom has no SD Card
SD_CS_PIN    = -1          ' Custom has no SD Card
I2C_PIN      = 28          ' I2C Boot EEPROM SCL Pin
I2C_DEV      = $A0         ' I2C Boot EEPROM Device Address
SI_PIN       = 31          ' PIN (Custom)
SO_PIN       = 30          ' PIN (Custom)
'
DEBUG_PIN    = -1          ' PIN (Custom)
'
' Custom platform Clock definitions:
'
CLOCKMODE = xtall + pll16x  ' (Custom)
XTALFREQ  = 5_000_000       ' (Custom)
CLOCKFREQ = 80_000_000      ' (Custom) Nominal clock frequency
                                     ' (required by some drivers)

Ln 1, Col 1      100%  Unix (LF)  UTF-8

```

You will generally need to modify the various **PIN** numbers, and the **CLOCKMODE** and **XTALFREQ** to suit your platform. It is unusual to have to modify anything else in this file. You should comment out the **ERROR** line once you have configured the custom platform.

Once the custom platform has been configured, you use it by defining the **CUSTOM** symbol on the command line (e.g. by using **-C CUSTOM**).

By default, the **CUSTOM** platform is configured to support only one CPU, no SD card and no XMM RAM. If you are careful to always specify only combinations of options that are actually supported by your platform, then there is no need to edit any of the other target files. However, if your platform does not support all possible HMI options (e.g. it has no VGA output) then you may also choose to edit the *Custom_HMI.inc* file.

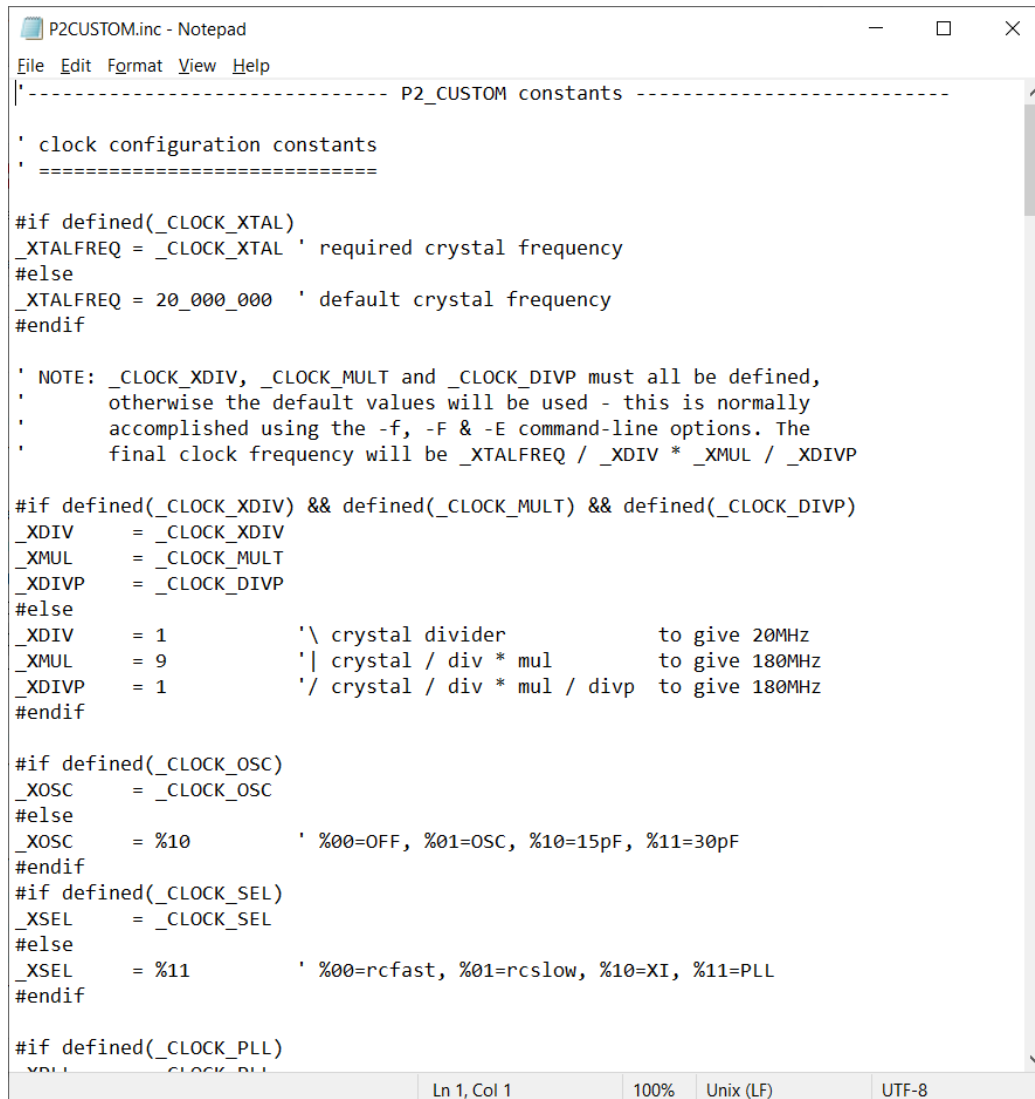
If your Propeller platform is unusual or unique, or has additional hardware that requires additional drivers, then you may need to develop your own dedicated targets. However, it is suggested that you base them on the default targets provided, using the symbols described above to select similar options (where appropriate).

Apart from the default targets, Catalina also provides various debug targets. Using the debug targets is beyond the scope of this tutorial – for details refer to the **Catalina Reference Manual**.

Supporting Other Propeller 2 Platforms

For other Propeller 2 platforms (i.e. any propeller platform not mentioned in the previous sections), you can modify the definitions used for the custom platform. This file is called *target\p2\P2CUSTOM.inc* and it includes the definition of the clock frequency and pin definitions etc (see also the file *target\p2\P2CUSTOM.TXT* for more information).

The file *P2CUSTOM.inc* is a normal text file that can be edited using notepad. This file will look something like this:



```

P2CUSTOM.inc - Notepad
File Edit Format View Help
|----- P2_CUSTOM constants -----|

' clock configuration constants
' =====

#if defined(_CLOCK_XTAL)
_XTALFREQ = _CLOCK_XTAL ' required crystal frequency
#else
_XTALFREQ = 20_000_000 ' default crystal frequency
#endif

' NOTE: _CLOCK_XDIV, _CLOCK_MULT and _CLOCK_DIVP must all be defined,
' otherwise the default values will be used - this is normally
' accomplished using the -f, -F & -E command-line options. The
' final clock frequency will be _XTALFREQ / _XDIV * _XMUL / _XDIVP

#if defined(_CLOCK_XDIV) && defined(_CLOCK_MULT) && defined(_CLOCK_DIVP)
_XDIV = _CLOCK_XDIV
_XMUL = _CLOCK_MULT
_XDIVP = _CLOCK_DIVP
#else
_XDIV = 1 '\ crystal divider to give 20MHz
_XMUL = 9 '| crystal / div * mul to give 180MHz
_XDIVP = 1 '/ crystal / div * mul / divp to give 180MHz
#endif

#if defined(_CLOCK_OSC)
_XOSC = _CLOCK_OSC
#else
_XOSC = %10 ' %00=OFF, %01=OSC, %10=15pF, %11=30pF
#endif
#if defined(_CLOCK_SEL)
_XSEL = _CLOCK_SEL
#else
_XSEL = %11 ' %00=rcfast, %01=rcslow, %10=XI, %11=PLL
#endif

#if defined(_CLOCK_PLL)
_XPLL = _CLOCK_PLL
#endif

```

Once the custom platform has been configured, you use it by defining either the **CUSTOM** or **P2_CUSTOM** symbol on the command line (e.g. by using **-C CUSTOM**).

If your Propeller platform is unusual or unique, or has additional hardware that requires additional drivers, then you may need to develop your own dedicated targets. However, it is suggested that you base them on the default targets provided, using the symbols described above to select similar options (where appropriate).

Apart from the default targets, Catalina also provides various debug targets. Using the debug targets is beyond the scope of this tutorial – for details refer to the **Catalina Reference Manual**.

Advanced Concepts

Using a different Target or Target Package

So far we have only used one target package, and one propeller target (the default ones in each case). Since we can specify so many propeller target configuration options on the command line, we may never need to use a different target package. But it can be useful in some circumstances – e.g. to develop dedicated targets for a specific platform or application.

By default, the Catalina compiler expects the target package to be in a directory called *target* (e.g. *C:\Program Files (x86) \Catalina\target*). However, there are two ways we can override this – by using command line options, or by using environment variables.

A target package can be any directory, and can be specified by using the **-T** option on the command line – note that this option requires a *capital-T* to be specified – the lower case option **-t** is used to choose individual targets *within* the target package, while the upper case option **-T** is used to choose a different target package.

Suppose we wanted to develop a new Catalina target – we might choose to do so by first copying the default target package to a new directory – say *C:\My_Package* – and we would then want to specify that Catalina should use the new target package in place of the default one. This can be one using the **-T** command line option:

```
catalina hello_world.c -lc -TC:\My_Package
```

Note that Catalina requires file and path names specified on the command line that contain spaces to be enclosed in double quotes – e.g.:

```
catalina hello_world.c -lc -T"C:\Program Files (x86)\My_Package"
```

The only problem with this approach is that the appropriate **-T** option has to be specified on each and every compilation. If you intend to do many such compilations, you can specify this using the **CATALINA_TARGET** environment variable:

```
set CATALINA_TARGET=C:\My_Package
```

Then a command such as:

```
catalina hello_world.c -lc
```

has the same effect as specifying the **-T** option on the command line. For this reason, knowing what the current setting of the various Catalina environment variables is can be very important. To display the current settings, use the command **catalina_env** – in the above case you might see output like:

```
CATALINA_DEFINE = [default]
CATALINA_INCLUDE = [default]
CATALINA_LIBRARY = [default]
CATALINA_TARGET = C:\My_Target
CATALINA_LCCOPT = [default]
CATALINA_TMPDIR = [default]
LCCDIR = [default]
```

More details on environment variables is given in the **Catalina Reference Manual**.

Catalina provides three target packages for the Propeller 1 – *target\p1*, *embedded\p1* and *minimal\p1*:

target\p1 This is the standard Catalina Propeller 1 target package, which supports all platforms, all memory models, all loaders and plugins.

embedded\p1 This is a smaller Propeller 1 target package, intended to support only a single platform, and only the basic plugins required for ANSI compliance. However, it supports all memory models and loaders. This target package is intended as the place to start when developing a target for an application that need only run on a single specific Propeller platform – particularly one that requires unusual configuration, or unusual driver support. The basic target platform does not include support for any HMI drivers, so it is also suitable where either no such drivers are required (e.g. in a deeply embedded application) or where very specific or highly customized HMI drivers need to be developed.

minimal\p1 This is a very trivial Propeller 1 target package. It supports only a single platform, a single memory model (LMM), a single plugin and the serial loader. This target package is intended to be used to demonstrate adding plugins to Catalina. See the document **Getting Started with Plugins** for more details.

Catalina provides one target packages for the Propeller 2 – *target\p2*:

target\p2 This is the standard Catalina Propeller 2 target package, which supports all Propeller 2 platforms and plugins.

Note that Catalina automatically adds the *\p1* or *\p2* suffix to the target name based on whether or not the **-p2** command line option is used. This means that you still specify the target as just *target*, not as *target\p1* or *target\p2* on the command line.

Using an SD Card with Catalina

If you have a Propeller platform that has an SD Card, you can use it in two ways:

1. As a way of loading programs into the Propeller.
2. As a file system for Catalina programs to use.

The two uses are completely independent – a program may be loaded from an SD Card but not thereafter access the SD Card at all, or a program may be loaded from EEPROM but then access the SD Card as a file system. Of course, a single program may also do both.

Using the SD Card to load Programs

The *catalyst* subdirectory contains the **Catalyst** SD Loader program that can be used to load both normal SPIN programs and Catalina programs from an SD Card.

Propeller 1 programs that fit into the normal 32k of the Hub RAM (such as Catalina TINY or COMPACT programs) and Propeller 2 programs that fit into the normal 512k of Hub RAM (such as Catalina TINY, COMPACT or NATIVE programs) require no special handling¹⁰.

To compile Catalyst, use the batch provided in the *catalyst* subdirectory (there is a *build_all* script provided). See the *README.TXT* and *CATALYST.TXT* files in this folder for more details. For example, to build Catalyst for the Hybrid you might use a command like:

```
build_all HYBRID
```

or, for the TriBladeProp

```
build_all TRIBLADEPROP CPU_2 PC VT100
```

or, for the P2_EVAL

```
build_all P2_EVAL TTY VT100
```

The compilation process puts all the output files into the *catalyst/image* subdirectory. Copy the entire contents of this directory to a suitably formatted SD card (it should be formatted as a FAT32 volume).

On the Propeller 1, you would normally have Catalyst permanently loaded into EEPROM. For example:

```
payload -e image\catalyst.bin
```

On the Propeller 2, Catalyst makes a copy of *catalyst.bin* as *_BOOT_P2.BIX*, so that it will load automatically when the Propeller 2 is rebooted with the SD card inserted (provided you have set the configuration switches accordingly - see the Parallax Propeller 2 documentation).

On startup, Catalyst prompts you to enter a command, which can be the name of a

¹⁰ Actually, Propeller 1 program must fit into 31kb – Catalyst needs 1kb to perform the load from SD Card. However, Catalina also supports 2-phase loads, which allows cog code to be loaded into cog RAM and then that Hub RAM is available for more program code without requiring any special programming - so Catalina can be used to easily build larger programs than is possible with other compilers even on a Propeller 1 without XMM RAM.

file to load, plus any parameters the program expects at run time. Refer to the **Catalyst Reference Manual** for more details.

To compile a Propeller 1 XMM program to be loaded by Catalyst is relatively straightforward – for example, a command similar to the following can be used:

```
catalina hello_world.c -lc -C SMALL
```

The output of this will be a file called *hello_world.binary* – but note that Catalyst can only use DOS 8.3 type file names, so when this file is copied to the root directory of the SD Card it must be renamed to something like *HELLO.BIN*. Then the SD Card can be inserted onto the Propeller platform and the program can be loaded using command **HELLO**.

Finally, note that if you remove and re-insert the SD Card at any time, you will need to restart the Catalyst Loader program.

Using the SD Card as a file system

Catalina programs can also use the SD Card as a file system, using all the standard C functions defined in *stdio* (e.g. **fopen**, **fclose**, **fseek**, **fprintf**, **fscanf**, **fputc**, **fgetc** etc).

Catalina supports read/write access to SD cards formatted as FAT16 or FAT32 volumes¹¹. Files can be read or written into the root directory or into subdirectories. Up to 20 files can be opened simultaneously (actually 17, since **stdin**, **stdout** and **stderr** each consume a file handle whether explicitly opened or not).

Because the ANSI C library functions required to support file system access are quite large, Catalina supports several different versions¹² of the standard C library:

libc : this version only provides support for **stdin**, **stdout** & **stderr**

libci : this version only provides integer-only support for **stdin**, **stdout** & **stderr**

libcx : this version provides SD Card file system support.

libcix : this version provides integer-only SD Card file system support.

The library to use is specified using either the **-lc**, **-lci**, **-lcx** or **-lcix** command line option when compiling the C program.

To support SD Card access, Catalina needs to load an SD card support driver – but this is done automatically by Catalina if the program is linked with the **-lcx** or **-lcix** version of the C library.

For an example on using the SD Card as a stdio file system, see the *demos\file_systems\test_stdio_fs.c* demo program. To compile this program, use a command such as:

```
catalina test_stdio_fs.c -lcx -C SMALL
```

Catalina will automatically load the SD card driver in this case because the **-lcx**

¹¹ Only a sector size of 512 bytes is supported.

¹² Actually, there are more than this – there are special versions of each library required to support the large addressing mode. The small and tiny addressing modes both use the same library since the difference is implemented in the Kernel. However, the choice between small and large library versions is made automatically by the Catalina binder.

library was used (in place of the normal **-lc** library).

The output of this will be a file called *test_stdio_fs.binary* – but the Catalyst Loader can only use DOS 8.3 type file names, so this file must be renamed when it is copied to the root directory of the SD Card (e.g. to *TEST_FS.BIN*). Then the SD Card can be inserted onto the Propeller platform and the program can be loaded using the Catalyst Loader. The *test_stdio_fs.c* program prompts for a file name and then displays the chosen file on the screen (so choose a text file!). It then prompts for another file name and writes 1000 lines of text to that file. Then it loops – so you can enter the name of the file just written and it will be displayed.

Note that it is typically only practical to use the stdio file system from an XMM program on the Propeller 1 – this is because the stdio file system support is so complex that programs which use it cannot usually fit within 32kb of Hub RAM.

However, Catalina also provides a smaller, simpler but fully functional file system, based on DOSFS that can. See the *demos\file_systems\test_catalina_fs.c* demo program. To compile this program for the Propeller 1, use a command such as:

```
catalina test_catalina_fs.c -lcx -C COMPACT
```

The resulting binary (*test_catalina_fs.binary*) is a standard Propeller binary which can be loaded as normal, either using Payload or Catalyst.

On the Propeller 2, which has 512kb of Hub RAM, any TINY, COMPACT, NATIVE, SMALL or LARGE mode program can access the SD Card using either the Catalina file system or the stdio file system.

Finally, note that if you remove and re-insert the SD Card at any time, you will need to restart any program that accesses the SD card.

Multi-CPU Systems

Introduction

Catalina supports multi-CPU Propeller 1 systems (such as the **TRIBLADEPROP**) in two main ways:

1. By providing a set of “proxy” drivers. These drivers allow programs running on one CPU to have access to a device (e.g. screen, keyboard, mouse or SD card) physically connected to another CPU. The client CPU runs the proxy drivers and the server CPU (i.e. the one with the actual physical devices) runs a special proxy server program which services device requests from the client. The communication between the client and the server uses serial communications between the two CPUs. The proxy drivers are as close as possible in functionality to the ordinary drivers, and can usually be used without any C code changes. The main difference is that their performance will typically be slower because the proxy driver has to communicate serially with the real driver running on the other CPU to perform each driver function.
2. By providing a set of utilities that allows one CPU to control another CPU – i.e. to reset or load programs into that CPU. Programs can be loaded into RAM or EEPROM (if the target CPU has a boot EEPROM attached).

For more details on Catalina’s support for Multi-CPU systems, refer to the Catalina Reference Manual. This section just contains a brief tutorial-style introduction.

Specifying the CPU

The first problem that arises when using Catalina on a multi-CPU system is how to specify the CPU on which the Catalina program is intended to run?

In a multi-CPU system, each CPU typically has a different set of capabilities – after all, one of the main reasons for having multiple CPUs is because it is not usually possible to support all the desired input and output devices on a single Propeller chip – you simply run out of pins!

This means that it is typical in multi-CPU systems that some CPUs have VGA or TV outputs, while others have keyboard or mouse inputs. Or one CPU may have an SD card while another has XMM memory.

Catalina identifies each CPU in a multi-CPU system with the symbols **CPU_1**, **CPU_2**, **CPU_3** etc¹³.

To avoid confusion, these symbols should **always** be specified on the command line for each compilation in a multi-CPU system – even when they are not strictly necessary (e.g. because the program being compiled makes no use of the devices that differ between the CPUs).

In these tutorials we will also sometimes adopt the convention that each resulting executable has an extension of **.1**, **.2** or **.3** to indicate the CPU on which it is intended to run when loaded on an SD card.

¹³ Currently, only the symbols for CPUs 1, 2 and 3 are ever used – but these are just symbols, so adding more is not difficult. The purpose of the symbols is to support the use of conditional compilation in the various Catalina target files and utility programs.

For example:

```
catalina hello_world.c -lc -o hello_world_1 -C TRIBLADEPROP -C CPU_1
catalina hello_world.c -lc -o hello_world_2 -C TRIBLADEPROP -C CPU_2 -C PC
```

The above commands will generate two binary output files – *hello_world_1.binary* and *hello_world_2.binary*. The *hello_world_1.binary* program will use the TV terminal device driver for its output, while *hello_world_2.binary* will use the serial port on CPU #2 for output.

However, when we copy these programs to an SD card (which requires 8.3 file names), we might elect to rename them as *hello.1* and *hello.2* so as more easily distinguish between them, since they will only run successfully on the CPU for which they were compiled.

Following these simple rules may save much head-scratching, cursing and confusion later – particularly when you start using Catalina’s proxy device capabilities, which allow programs running on one CPU to access the devices running on another.

Multi-CPU Utility Programs

The next problem is how to load the Catalina program into one the CPU on which it is meant to run?

In some multi-CPU systems, each CPU has its own serial connection to a PC – and SPIN, PASM and Catalina LMM programs can be downloaded from the PC using the normal Propeller tools. But this method doesn’t work for Catalina XMM programs.

On a single CPU system, Catalina XMM programs are usually loaded from an SD card, or from EEPROM – but in multi-CPU systems some of the CPUs may have neither SD Cards nor EEPROMs. However, in these systems there is nearly always a couple of pins on each of the CPUs that are tied together to allow for inter-CPU communications – and Catalina can use these pins to establish serial communications capability between the various CPUs, allowing once CPU to be used to load XMM programs into another CPU via a simple serial protocol. Catalina needs two shared pins between the CPUs to do this, and can also make use of another common feature of multi-CPU systems – a shared pin that allows one CPU to reset another.

Catalina provides four utility programs to allow one CPU to load/manage another CPU:

Catalyst In multi-CPU systems, the Catalyst SD Loader can be on the CPU that has an SD card attached, but as well as loading programs into the local CPU, it can also be used to load programs off the SD card and into another CPU.

Generic SIO Loader This program must be run on a CPU to receive programs loaded from Catalyst (but isn’t this a “chicken and egg” problem – i.e. how do we get this program loaded – especially if the CPU doesn’t have an EEPROM? Well, see the **CPU_n_Boot** program below).

CPU_n_Boot This program uses the built-in program load capabilities of the Propeller (the same ones that the Parallax Propeller tools use) to load a boot loader program into *another* CPU using serial communications. The program that this program loads

is the **Generic SIO Loader** described above – so once you execute *this* program, you can then use **Catalyst** to load *any other* program.

CPU_n_Reset

This program simply resets another CPU – this is handy if the CPU has stopped responding for some reason.

Catalyst is built using the scripts in the catalyst subdirectory.

For the TRIBLADEPROP multi-CPU platforms, Catalyst can be compiled by using the normal Catalyst script. Note that you have to specify the CPU For example:

```
cd catalyst
build_all TRIBLADEPROP CPU_2 PC VT100
```

To build utilities for all the Multi-Prop CPUs, use the **build_all** script in the utilities folder¹⁴:

```
cd utilities
build_all TRIBLADEPROP
```

There are also copy scripts for the multi-CPU systems, to copy the appropriate utilities to a specified SD Card. The use of these scripts is described later.

The binary programs that are generated are named for the CPU on which they are intended to be executed as well as the CPU they affect. For example, the program to load the boot loader into CPU #1, but which is intended to be run on CPU #2 will be named *CPU_1_Boot_2.binary*. If adopting the conventions described earlier, this program would be named something like *boot2.1* when copied to an SD Card.

More details on these utility programs are given in the **Catalina Reference Manual**.

Proxy Devices

What do you do if your multi-CPU system does not have all the devices you want on the same CPU?

In multi-Prop systems it is rare for each CPU to have a full complement of all the common I/O devices that you might like to use in a C program – like a TV or VGA outputs, a keyboard or mouse inputs, or an SD card interface. Often the very purpose of using multiple CPUs is to support special hardware (such as XMM memory) that requires so many pins that a single Propeller CPU would not be able to support both that hardware and the common I/O devices.

This means that it is common for the keyboard, mouse, TV/VGA and SD card devices to be spread over two or more CPUs. If you want to use them all from one program, you need some way of accessing the devices remotely.

To do this, Catalina provides “proxy” device drivers, and a “proxy” server. Proxy device drivers are included for the following devices:

- screen (TV or VGA)
- mouse
- keyboard
- SD card

These devices are used much like ordinary devices, but are activated when the

¹⁴ This script is deprecated for building utilities for single CPU systems – use the *build_utilities* script instead. But it is still supported for building the utilities for the multi-CPU systems

following symbols are defined on the Catalina command line:

```

PROXY_SCREEN
PROXY_MOUSE
PROXY_KEYBOARD
PROXY_SD

```

Here is a trivial example of compiling a program to use a proxy screen driver:

```
catalina hello_world.c -C CPU_1 -C PROXY_SCREEN
```

This command tells Catalina that this program is intended to run on CPU #1, but that it will use a proxy screen device that is physically connected to another CPU.

When we run this program on CPU #1, it will send any text output requests to a server program on another CPU. On that CPU, we must simultaneously run a server that knows how to process such proxy device requests (in this case, by accessing the local display device).

Catalina provides a special utility program, called the **Generic Proxy Server**, which knows how to act as a proxy for *any* or *all* proxied devices. We tell it what devices it should act as proxy for (and the appropriate real drivers to use) when we compile the server.

Note that proxy devices can usually be used on any CPU in a multi-CPU system. For example, in a multi-CPU system, CPU #1 might have an input for a keyboard, while CPU #2 might have a VGA display output. By using proxy devices, a Catalina C program that requires both keyboard and VGA can be run on *either* CPU.

Here are example commands to compile the same program to run on the two CPUs:

```

catalina test_suite.c -o client_2 -C CPU_2 -C PROXY_KEYBOARD
catalina test_suite.c -o client_1 -C CPU_1 -C PROXY_SCREEN

```

Each program will require its own **Generic Proxy Server**, which will have to be compiled slightly differently, and then executed on the other CPU to provide access to the proxy device. The Generic Proxy Server is a SPIN/PASM program, not a C program, so it must be compiled with **spinnaker**, not **catalina**. Here are the commands required to build the proxy servers to match the above programs:

```

spinnaker -p -a Generic_Proxy_Server -o server_1 -L ../target -b -C
CPU_1 -C NO_SCREEN

```

and

```

spinnaker -p -a Generic_Proxy_Server -o server_2 -L ../target -b -C
CPU_2 -C NO_KEYBOARD -C NO_MOUSE

```

Note that in a multi-CPU system, there can currently only be one client program and one proxy server – a proxy server cannot simultaneously serve multiple clients (this capability may be added in a future release).

Also note that the capabilities of the proxy devices does not necessarily include *all* the capabilities of the same device when used locally – in particular, the screen devices generally do not currently support cursor, scroll or color functions when used as proxy devices – these capabilities will be added in a future release. At the moment, if you need to use these functions, run your program on the CPU that has direct local access to the screen device, and proxy the other devices you need (such as keyboard, mouse or SD card).

TriBladeProp

TriBladeProp Overview

The standard Catalina Support Package supports the TriBladeProp multi-CPU platform. However, not all Catalina targets and options are applicable to all the CPUs. Details of the TriBladeProp CPU configurations are given in the file *TriBladeProp_README.TXT* in the Catalina *target* directory.

Here is a brief summary:

CPU #1 Supported by all LMM, EMM and XMM targets with all options. Note that for XMM programs to build correctly, both the **TRIBLADEPROP** option and the **CPU_1** option *must* be specified.

The SD Card on CPU #2 can be used indirectly by specifying the **PROXY_SD** option, and running a suitably configured proxy server on CPU #2. For more details on using proxy devices, see the Catalina Reference Manual.

CPU #2 Supported by all LMM, EMM and XMM targets. Note that for XMM programs to build correctly, both the **TRIBLADEPROP** option and the **CPU_2** option must be specified. The following options are supported directly:

ALTERNATE
EEPROM
NO_HMI
PC
PROPTERMINAL
CLOCK
NO_FP (or **NO_FLOAT**)

Note that no local screen, keyboard or mouse options are supported. However, the HMI devices on CPU #1 can be used indirectly by specifying the **PROXY_SCREEN**, **PROXY_KEYBOARD** or **PROXY_MOUSE** options, and running a suitable configured proxy server on CPU #1. For more details on using proxy devices, see the **Catalina Reference Manual**, or part 2 of the tutorial.

CPU #3 Supported by all the LMM and EMM targets. Even though it is not strictly required, both the **TRIBLADEPROP** option and the **CPU_3** option should be specified. Only the following options are supported:

ALTERNATE
NO_HMI
PC
PROPTERMINAL
CLOCK
NO_FP (or **NO_FLOAT**)

Note that no local screen, keyboard or mouse options are supported. However, the HMI devices on CPU #1 can be used indirectly by specifying

the **PROXY_SCREEN**, **PROXY_KEYBOARD** or **PROXY_MOUSE** options, and running a suitable configured proxy server on CPU #1. For more details on using proxy devices, see the **Catalina Reference Manual**, or part 2 of the tutorial.

The SD Card on CPU #2 can be used indirectly by specifying the **PROXY_SD** option, and running a suitably configured proxy server on CPU #2. For more details on using proxy devices, see the Catalina Reference Manual.

For more details on the meaning of the options, see the Catalina Reference Manual.

If you want to use the SD Card as a file system from a C program (which is possible directly from CPU #2, or using a proxy driver from CPU #1) remember to specify the **-lcb** option instead of the usual **-lc** option when compiling. The **libcb** library includes full file system support, whereas the standard **libc** library does not – if you forget to do this the program will compile and run, but will be unable to access any files on the SD Card.

TriBladeProp Tutorial Hardware Setup

This tutorial assumes the TriBladeProp is configured as follows. If your TriBladeProp is configured differently, some parts of the tutorial may not work correctly, or may have to be skipped.

- CPU #1** configured for TV output, with the mouse and keyboard on pins P8 – P12. TV, keyboard and mouse connected. Jumpers installed on LK11 and LK12 as described in the document *README.TriBladeProp* in the *utilities* directory. Configured for XMM with 512K SRAM installed. EEPROM installed. Jumpers installed on LK1 and LK2.
- CPU #2** configured for XMM with 1Mb SRAM installed. SD Card installed. The 4 pin Prop Plug jumper cable installed from J92 to J21 (i.e. from the USB Prop Plug to CPU #2).
- CPU #3** Links LK3, LK4, LK5 installed (A1 to B1 in each case). Pin P24 (available on J33) connected to the spare LED on Blade #3 ¹⁵.

With this configuration, we will mostly use a PC terminal package (such as payload in interactive mode) to communicate with CPU #2.

We will use the SD Card on CPU #2 to load programs to that CPU, and the serial communications loader to load programs from CPU #2 to CPU #1 and CPU #3.

We will see some program output on the TV, and use the keyboard and mouse connected to CPU #1. We will also use targets on CPU #2 that send outputs to a terminal emulator running on the PC.

We will boot CPU #3 from CPU #2, and load and execute a program on CPU #3 that flashes a single LED.

We will show programs running on all CPUs simultaneously.

This section assumes you are running Catalina under Windows – if you are running under Linux, some of the path names may differ.

¹⁵ Catalina supports either CPU #1 or CPU #2 being used as a proxy server. CPU #3 is not currently supported as a proxy server since it does not have a direct connection to any of the devices supported by a proxy server.

TriBladeProp Tutorial Software Setup

1. Make sure no Catalina environment variables are set (use **catalina_env** to check) and unset any that are not set to default (other than LCCDIR).
2. Prepare a LED Test Program for CPU #3. The following program (available in *demos/examples/ex_cpu_3.c*) will be used by default. It is a simple program designed to flash the LED of the TriBladeProp CPU #3 (which must be connected to pin 24, available on J23):

```
#include <catalina_cog.h>
void main() {

    unsigned count;
    unsigned mask    = 0x01000000; // bit 24
    unsigned on_off = 0x01000000;

    _dira(mask, mask);
    _outa(mask, on_off);
    count = cnt();

    while (1) {
        _outa(mask, on_off);
        count += 100000000;
        _waitcnt(count);
        on_off ^= mask;
    }
}
```

3. Enter the *utilities* directory (**not** *demos\utilities*) and build the TriBladeProp utilities:

```
cd utilities
build_triblade_utilities
```

If you get any errors during the build, make sure you have correctly executed the **use_catalina** script in the main Catalina directory.

4. Enter the *demos\catalyst* directory and build Catalyst for the TriBladeProp:

```
cd demos\catalyst
build_all TRIBLADEPROP CPU_2 PC
```

5. Enter the *demos\multicpu* directory and build the TriBladeProp demo programs:

```
cd demos\multicpu
build_triblade_demos
```

Note that just using the normal *build_all* scripts in some directories will not build programs correctly for all the various CPUs of the TriBladeProp – they are intended only to build programs for a single CPU. This is fine for Catalyst itself, since it is only the utilities and demos that run on other CPUs

6. Prepare a Micro SD Card (FAT format) and copy the following programs into the root directory. Since Catalina can only use short (8.3) file names, the following instructions will do some name mapping where required:

From the *utilities* directory, (**not** demos\utilities) enter the following command (assuming X: is the drive in which the SD card appears):

```
cd utilities
copy_triblade_utilities X:
```

This will copy (and rename) the following files:

```
CPU_1_Boot_2.binary      →   BOOT1.2
CPU_3_Boot_2.binary      →   BOOT3.2
CPU_1_Reset_2.binary     →
RESET1.2 CPU_3_Reset_2.binary →
RESET3.2
```

Copy all files in the *demos\catalyst\image* folder to the SD card – there is no script to do this – just copy all files to the SD card root directory using a command such as (assuming X: is the drive in which the SD card appears):

```
cd demos\catalyst\image
copy *.* X:
```

From the *demos\multicpu* directory, enter the following command (assuming X: is the drive in which the SD card appears):

```
cd demos\multicpu
copy_triblade_demos X:
```

This will copy (and rename) the following files:

```
othello_1.binary        →   OTHELLO.1
startrek_1.binary        →   STARTREK.1
othello_pc_2.binary      →   OTHELLO.2
test_suite_pc_2.binary   →   TEST.2
test_fs_2.binary         →   TEST_FS.2
startrek_2.binary        →   STARTREK.2
test_3.binary            →   TEST.3
```

In case the name mapping convention is not obvious, we are using the extension of the file to indicate the CPU on which the program is intended to be executed – so *RESET3.2* means the program will reset CPU #3, but it must be executed on CPU #2.

The Micro SD Card should then be inserted into the TriBladeProp.

7. Connect the TriBladeProp to the PC via the onboard PropPlug USB port.
8. Make sure you have a TV, keyboard and mouse connected to CPU #1, and a USB cable connecting your PC to CPU #2.

Finally, we're ready to go!

TriBladeProp Tutorial Part 1 – Loading and running programs

1. Power on the TriBladeProp. On the PC, we will use the payload loader in interactive mode to talk to the Propeller. We must also program the Catalyst program loader in the EEPROM of the TriBladeProp. We can do both of these with one payload command:¹⁶

```
cd demos\catalyst\image
payload -e catalyst.bin -i
```

If everything is working correctly, you should see a prompt like:

```
Catalyst v8.3
>
```

2. This is the main Catalyst prompt. At this prompt, you can do one of two things:

- (2.a) Enter a command – try entering **dir** and pressing return. The contents of the root directory SD Card will be displayed, showing the programs we have copied to the SD Card – e.g:

```
Directory ""
-----
TRIBLADE  SYSTEM~1  OTHELLO .1  STARTREK.1  OTHELLO .2  TEST .2
TEST_FS .2  STARTREK.2  LISP .2  TEST .3  T1_C_2 .BIN  T1_S_1 .BIN
T2_C_1 .BIN  T2_S_2 .BIN  BASICS .P5  BASICS .PAS  CAT .BIN  CATALYST.BIN
CATALYST.TXT  CP .BIN  DBASIC .BIN  ELIZA .BAS  EX_BASE .BAS  EX_CALL .BAS
EX_COLON.BAS  EX_CVI .BAS  EX_DEF .BAS  EX_DIM .BAS  EX_ERROR.BAS  EX_EXPR .BAS
EX_FILE1.BAS  EX_FILE2.BAS  EX_FILE3.BAS  EX_FILE4.BAS  EX_FILE5.BAS  EX_FILE6.BAS
EX_FILE7.BAS  EX_FN .BAS  EX_FOR .BAS  EX_GET .BAS  EX_GOSUB.BAS  EX_IF .BAS
EX_INKEY.BAS  EX_INPUT.BAS  EX_LET .BAS  EX_LINE .BAS  EX_LSET .BAS  EX_MID .BAS
EX_ON .BAS  EX_OPEN .BAS  EX_PEEK .BAS  EX_POS .BAS  EX_READ .BAS  EX_STR .BAS
EX_TIMER.BAS  EX_USING.BAS  EX_USR .BAS  EX_VAR~1.BAS  EX_WHILE.BAS  EX_WRITE.BAS
FACT .LUA  FIB .LUA  HELLO .LUA  HELLO .PAS  HELP .XVI  JZIP .BIN
LS .BIN  LUA .BIN  LUAC .BIN  MATCH .PAS  MKDIR .BIN  MV .BIN
PCOM .BIN  PINT .BIN  RM .BIN  RMDIR .BIN  ROMAN .PAS  SLOCPS .BAS
SORT .LUA  SST .BIN  SST .DOC  STARTREK.BAS  STARTREK.P5  STARTREK.PAS
TREK15 .BAS  UT-TREK .BAS  VI .BIN  ZORK1 .DAT  ZORK2 .DAT  ZORK3 .DAT
BOOT1 .2  BOOT3 .2  RESET1 .2  RESET3 .2
Catalyst 4.9
>
```

- (2.b) Select a CPU to load a program into. Press **SHIFT+n** for CPU #n – i.e:

SHIFT+1 (i.e. “!”) for CPU #1

SHIFT+2 (i.e. “@”) for CPU #2

SHIFT+3 (i.e. “#”) for CPU #3.

SHIFT+0 (i.e. “)”) to return to the default

When a CPU is selected the prompt will change to indicate the currently selected CPU.

3. Now let's load our first program. We will start by loading a program into CPU #2. If CPU #2 is not selected (as shown by the prompt), select it by entering the single character “@” at the Catalyst prompt, and verify that the prompt changes to indicate CPU #2. Then type in the file name *OTHELLO.2*. Press Enter and you should see the opening few lines of the program displayed in the terminal emulator:

```
REVERSI
You can go first on the first game, then we will take turns.
```

¹⁶ Note that if your TriBladeProp has no EEPROM on CPU #2 you can still do the tutorial – but you will need to omit the -e flag to payload, and download the catalyst.bin program again to CPU #2 after each step.

```

    You will be white - (O)
    I will be black   - (@).

Select a square for your move by typing a digit for the row
and a letter for the column with no spaces between.

Good luck! Press Enter to start.

```

Othello is compiled to use the PC HMI plugin, so the output of the program is sent to the terminal emulator, and your keystrokes in the payload terminal emulator are sent back to the program. You can therefore play Othello using the payload terminal emulator.

When you are finished playing – or you just want to proceed with the demo – you can reboot the TriBladeProp to return to the Catalyst prompt. If our TriBladeProp has an EEPROM then Catalyst will restart each time. Otherwise, also restart payload and reload Catalyst

- Now let's try loading a program into another CPU. We'll use CPU #1, and assume that the CPU has no EEPROM – so first we have to load the companion load program. There is a program we can load into CPU #2 that knows how to load the companion program into another CPU – for CPU #1, this is *BOOT1.2* To execute this, make sure CPU #2 is selected (not CPU #1) then type in the file name *BOOT1.2* and press enter. You will see output similar to that shown below:

[illegible]

If for some reason you don't see this, then go to step 8 to see what you can do about it. Then return here and try again.

The reason you may see the gobbledygook shown above between the Catalyst prompts is that CPU #2 is using the serial link to download the companion load program to CPU #1 – and this is the same serial link that the payload terminal emulator is using. Once the load is completed and the Catalyst prompt is shown, we can download a Catalina program to CPU #1.

First, we need to select CPU #1 in the Catalyst – to do this, type the single character “!” - the prompt should change to indicate CPU #1. Then type in the filename *STARTREK.1* and press ENTER. You will again see gobbledegook in payload while the program loads (for the same reason as described above), and then the program should start up, displaying the usual Super Star Trek opening lines on the TV connected to CPU 1.

The full sequence of what just happened is that the *STARTREK.1* program has been read from the SD Card by the Catalyst program on CPU #2, downloaded across the serial connection from CPU #2 to CPU #1, and the companion loader in CPU #1 has read the records and loaded them into XMM RAM as they arrived, then restarted itself. The target that the program was compiled with knows how to load the program back from XMM RAM and start it executing.

Payload should return to the normal Catalyst prompt when the load is completed. If for some reason it does not, then go to step 8 to see what you can do about it. Then return here and try again.

5. Next, let's try loading a program on CPU #3. Note while we do this that CPU #1 is still happily busy playing Super Star Trek. First, ensure CPU #2 is selected by entering the single character "@" at the Catalyst prompt, and verifying that CPU #2 is indicated – recall that to load the companion loader, we execute a program on CPU #2, not on CPU #3 (which doesn't yet know how to load such a program). The program we will execute on CPU #2 is *BOOT3.2* – type this file name in and press ENTER.

You should again see some gobbledygook similar to that shown above when we did this for CPU #1. When complete, the Catalyst prompt will return. If for some reason you don't see this, then go to step 8 before returning here and trying again.

Assuming the companion loader was loaded correctly, next we select that we want to download a program to CPU #3 by entering the single character "#" at the Catalyst prompt (and verifying the now indicates CPU #3).

Next, execute download the program to CPU #3. The program we will load is the LED test program *TEST.3* – type in this file name and press ENTER. You should see the LED on CPU #3 flash every couple of seconds. If you do not, check you correctly have the LED wired to P24 of the CPU #3 Propeller chip.

6. There are many things that can go wrong during the load process – the wrong CPU can be specified, the wrong file can be loaded, the file may have been compiled with the wrong options etc. Also, it is possible that the CPU is already executing some program that interferes somehow with the load process – for instance, anything that uses the serial I/O pins.

This may even be caused by a program executing on one of the other CPUs.

In such cases, it is useful to be able to reset the CPU and then try again (the only other alternative being to cycle the power to the whole TriBladeProp).

Two programs are provided for this purpose – *RESET1.2* and *RESET3.2* will reset the CPU #1 and CPU #3 propellers (respectively).

Recall that (by convention) the .2 suffix indicates that these programs should be executed on CPU #2. Let's try it – we will reset CPU #3.

First, ensure CPU #2 is selected by entering the single character "@" at the Catalyst prompt, and verifying that CPU #2 is indicated. Then type in the filename *RESET3.2* and press ENTER. If the LED test program was executing, it should now stop.

Then you can try loading the companion loader again (e.g. by typing in *BOOT3.2* and pressing ENTER.)

Finally, it is worth noting that the whole process is much easier if the SD Load program is permanently loaded into the EEPROM of CPU #2, and the companion serial load program is permanently loaded into the EEPROMS of CPU #1 and #3.

That's the end of part 1 of this tutorial. Try loading some of the other demo programs, recompiling them for different targets on different CPUs. When you are

ready to learn about using proxy devices, come back to part 2 of this tutorial.

TriBladeProp Tutorial Part 2 – Using proxy devices

Because all CPUs on the TriBladeProp share the same I/O pins – and these are the pins normally used to communicate with the PC, loading programs that use proxy drivers is a little complex on the TriBladeProp.

Before proceeding with this tutorial, please review again the hardware setup of the TriBladeProp – taking particular note of the following:

The 4 pin Prop Plug jumper cable must be installed from J92 to J21 (i.e. from the USB Prop Plug to CPU #2).

To enable CPU #2 to communicate with CPU #1, the two optional links LK1 and LK2 are installed on that Blade #1. In this tutorial we will need to remove and re-install these jumpers as part of the load process.

When you are sure you are familiar with the TriBladeProp configuration (in particular you have identified LK1 and LK2 correctly), then proceed with the tutorial:

1. To build the programs we will use in this tutorial, enter the *demos\multicpu* directory, and execute the following command:

```
cd demos\multicpu
build_triblade_proxy_demos
```

This will build two different pairs of proxy client and proxy server programs that demonstrate the use of various proxy devices. Examine the file if you like as examples of how each client and server is built – but this tutorial is mainly about how to execute programs that demonstrate the use of proxy devices.

2. Remove the Micro SD card from the TriBladeProp and insert it into your PC SD card reader/writer. Execute the following command in the *demos\multicpu* directory (assuming x: is the driver in which the SD card appears):

```
cd demos\multicpu
copy_triblade_proxy_demos x:
```

Then insert the SD card back into the TriBladeProp.

3. We will assume that **Catalyst** has been loaded into the EEPROM of CPU #2. If this is not the case, you will need to load it manually each time using Payload or the Parallax Propeller tool.

We will execute in turn each of the proxy client/server test program pairs.

4. The first test program runs a simple Catalina test suite on CPU #2, using the TV display and keyboard from CPU #1. Here are the steps required to run it:

- 4.a. Using payload in interactive mode to CPU #2, reset the TriBladeProp using the on/off switch. you should see the usual Catalyst prompt:

```
Catalyst v4.9
>
```

- 4.b. Now let's prepare to load our test program into CPU #1. First we have to load the companion load program. To do this, make sure CPU #2 is selected (not CPU #1) by entering a single character "@" at the prompt, and verifying that CPU #2 is indicated, Then type in the file name *BOOT1.2* and press enter. When that completes, the usual Catalyst prompt should return
- 4.c. We will first load the server program into CPU #1. Ensure CPU #1 is selected by entering a single character "!" at the prompt, and verifying that CPU #1 is indicated. Then type in the command *T1_S.1* and press ENTER (the name stands for *Test 1 Server for CPU #1*).
- 4.d. When that program has finished loading, you should see a blank screen on the TV display. Now that the server is running, it will be attempting to use the same pins for inter-CPU I/O that we need to use to load the client program – so before we can go any further we need to temporarily isolate CPU #1 by removing LK1 and LK2. Do this now.
- 4.e. When we have done this, the Catalyst program running on CPU #2 can be used again (you may need to press enter several times in the terminal emulator window to return to the program load prompt).
- 4.f. Ensure CPU #2 is selected by entering a single character "@" at the prompt, and verifying that CPU #2 is indicated. Then type in the command *T1_C.2* and press ENTER (the name stands for *Test 1 Client for CPU #2*).
- 4.g. Now both the client and server are running – but before they can communicate, we need to re-install the links on LK1 and LK2. Do this now.
- 4.h. When the programs can communicate, you should see the program display some text on the TV display and then pause, waiting for some input. Now press a key on the keyboard connected to CPU #1 – the program running on CPU #2 should receive the key stroke and continue.

Note that while the proxy client and server are communicating, you may see various rubbish characters appear in payload. This is normal, and is because the inter-CPU communications uses the same Propeller pins as the normal PC serial communications.

Now CPU #2 can be used to run all the usual Catalina programs that require user interaction – using the keyboard, mouse and screen on CPU #1!

- 5. The second test program uses the SD card. It is run using similar steps as in test 1, above – but this time we will first load the *client* on to CPU #1, and then the *server* onto CPU #2, since we want to be able to access the SD card on CPU #2 from CPU #1.

The steps to load the program are as follows:

- 5.a. Use payload in interactive mode to communicate with CPU #2, and reset the TriBladeProp by using the power switch. You should see the usual Catalyst prompt:

Catalyst v4.9

- >
- 5.b. Now let's prepare to load the second test program into CPU #1. First we have to load the companion load program. To do this, make sure CPU #2 is selected (not CPU #1) by entering a single character "@" at the prompt, and verifying that CPU #2 is indicated. Then type in the file name *BOOT1.2* and press enter. When that completes, then the Generic SD Card loader prompt should return:

Catalyst v4.9

- >
- 5.c. We will first load the client program into CPU #1. Ensure CPU #1 is selected by entering a single character "I" at the prompt, and verifying that CPU #1 is indicated. Then type in the command *T2_C.1* and press ENTER (the name stands for *Test 1 Client for CPU #1*).
- 5.d. When that program has finished loading, you may see some lines of text appear on the TV display, including a request for a file name to read. Do not enter a file name yet – we need to load the server program on CPU #2 first.
- 5.e. Now that the client is running, it will be attempting to use the same pins for inter-CPU I/O that we need to use to load the server program – so before we can go any further we need to temporarily isolate CPU #1 by removing LK1 and LK2. Do this now.
- 5.f. When we have done this, the Catalyst program running on CPU #2 can be used again (you may need to press enter several times in the terminal emulator window to return to the program load prompt).
- 5.g. Ensure CPU #2 is selected by entering a single character "@" at the prompt, and verifying that CPU #2 is indicated. Then type in the command *T2_S.2* and press ENTER (the name stands for *Test 2 Server for CPU #2*).
- 5.h. Now both the client and server are running – but before they can communicate, we need to re-install the links on LK1 and LK2. Do this now.
- 5.i. Now that the programs can communicate, you can enter a file name on the keyboard connected to CPU #1 – the program running on that CPU #1 should attempt read the file from the SD Card on CPU #2. Type in the name of a file to read – e.g. try *CATALYST.TXT*.

Note that while the proxy client and server are communicating, you may see various rubbish characters appear on the PC terminal emulators. This is normal, and is because the inter-CPU communications uses the same Propeller pins as the normal PC serial communications. But once the program is executing, you can close payload and it will keep running.

When the read is complete, the program will ask for a filename to write. Enter something like *TEST.TXT*. Catalina will write about a hundred lines of text to this file (Note that this can take a few seconds – it depends partly on the write speed of the SD card), and then ask you to enter a filename to read again. This time, enter *TEST.TXT* and the program will display the file it just created.

Now CPU #1 can run all the usual Catalina programs that require file system support – using the SD card installed on CPU #2!

That's the end of part 2 of this tutorial. Try compiling and loading some of the other demo programs using proxy devices. However, please be aware that using proxy devices changes the number of cogs required to run the program (it may increase it or decrease it) and therefore some programs will run out of cogs unless. To avoid using cogs unnecessarily, you should always use the **NO_SCREEN**, **NO_KEYBOARD** or **NO_MOUSE** and **NO_FLOAT** options to prevent cogs being wasted on devices or other plugins you do not need.

What next?

This is the conclusion of this tutorial. However, there are many programs in the *demos* directory and subdirectories that you can experiment with. You can compile all the programs in a given subdirectory by using the *build_all* scripts in each subdirectory, or you can try building each one individually – perhaps specifying different command line options to see what effect each one has.

Also, there is more information on Catalina, including more information on the different targets Catalina provides and additional command line options in the **Catalina Reference Manual**. Happy Landings!