Getting Started with Catapult

Table of Contents

Basic Concepts....2
Catapult pragmas....3
#pragma catapult common [name] [attributes]....3
#pragma catapult secondary [name [(type)]] [attributes]....4
#pragma catapult primary [name] [attributes]....4
Catapult pragma attributes....4
address(value)....4
stack(value)....5
mode(name)....5
options(list)....5
binary(name)....5
overlay(name)....5
Catapult macros....6
CATAPULT_ERROR(name, address)....6
FIXED_START(name, arg, cog, result)....6
RESERVE_AND_START(name, arg, cog, result)....7
RESERVE_SPACE(name)....8
RESERVED_ADDRESS(name)....8
RESERVED_START(name, arg, cog, result)....8
OVERLAY_SPACE....9
OVERLAY_START(name, arg, cog, result)....9
STOP(cog)....10
Sharing Data between Catapult Segments....11
Compiling a Catapult C program monolithically....11
Using Catapult with XMM programs....12
Using Catapult with Overlay Files....13
Using Catapult with multi-threading....14
Using Catapult with the Parallelizer....15
Using Catapult with Lua....16
Using Catapult for Client/Server applications....17
Using Catapult with ordinary C programs....18
Using Catapult with Geany....19
Other Things to Note about Mult-Model Programs....19

Basic Concepts

This document provides a brief tutorial to get you up and running and compiling C programs using Catalina Catapult.

Catapult is a utility intended to simplify the process of developing, debugging and maintaining Catalina multi-model programs.

Like Catalina itself, Catapult is a command line utility, and this document will use the Catalina Command line to demonstrate it - so if you have not used Catalina before, or have not used it from the command line, it may be useful to read the document **Getting Started with Catalina** before this document. However, Catapult can also be used with the Catalina Geany IDE. See **Using Catalina with Geany** later in this document.

The directory *demos\catapult* contains the example programs to demonstrate Catalina Catapult that are discussed in this document.

A multi-model program typically consists of a primary program in which speed is not critical (and so is typically executed from XMM RAM) which then loads and executes subsidiary programs that execute from Hub RAM whenever speed, precise timing or functionality not supported by XMM programs - e.g. multi-threading or interrupts (on the Propeller 2) is required.

Alternatively, the primary program may consist of a Hub-based primary program which loads other Hub-based secondary programs as overlays from files as necessary.

In either case, the main advantage of multi-model programs is that secondary programs do not consume Hub RAM except when they are actually executing. Multi-model support therefore allows C programs larger than Hub RAM to execute while still allowing them to access all the functionality of the Propeller when needed.

The main disadvantage of multi-model programs is that they are more difficult to develop, build, debug and maintain than ordinary C programs. Catapult helps by allowing multi-model programs to be developed and kept in a single C source file that contains both the primary program and all the secondary programs.

Catapult also helps by splitting the source file into separate program files, compiling each one using the appropriate memory model and options, invoking the utility required to turn these into data files to be loaded when required by the primary program, and then compiling the primary program to produce a single binary ready for execution.

The best way to see how Catapult simplifies this process is to build and run a demo program. There are several versions of the same demo program in this folder - some specifically for the Propeller 1 (e.g. *demo_p1.c*) and others specifically for the Propeller 2 (e.g. *demo_p2.c*).

The reason there are different versions of the program for the Propeller 1 and Propeller 2 is that the Propeller 1 and 2 support different memory models and different address ranges - both of which are important when building multi-model programs.

The **catapult** command always accepts a single C source file name as its argument.

To compile the Propeller 1 version of the demo program:

```
catapult demo_p1.c
```

or, to compile the Propeller 2 version of the demo program:

```
catapult demo_p2.c
```

Both these commands will take the specified C source file, split it up (in this case) into one primary and two secondary programs, compile those programs separately, and then combine them back into a single output binary. A command file will be created called either *_catapult.bat* (Windows) or *_catapult.cmd* (Linux).

Doing this manually is quite possible, but it can be quite complex. For an example of this, see the Makefile in the *demos\multimodel* folder. Catapult eliminates the need to write such complex Makefiles. Examine the command file generated by building the demo program (*_catapult.bat* or *_catapult.cmd*) to see that Catapult generates the necessary commands for (in this case) the three **catalina** commands and two invocations of the **spinc** utility required to build the final binary.

The resulting binary program can be loaded and executed using payload.

For the Propeller 1:

```
payload -i demo_p1
```

or, for the Propeller 2:

```
payload -i demo_p2
```

Note that if you compile BOTH the Propeller 1 and Propeller 2 versions, you may need to add the extension in the payload command (i.e. **.binary** for the Propeller 1, or **.bin** for the Propeller 2) or else use the **-o1** or **-o2** payload option to specify whether you are loading a Propeller 1 or Propeller 2.

Now for the details ...

Catapult pragmas

Catapult is implemented by adding pragmas to the C source file. There are three catapult pragmas:

#pragma catapult common [name] [attributes]

> This pragma introduces a common segment, which should contain all the definitions and types that are common to the primary and all the secondary programs - the common segment will be converted into a C header file shared by all other segments. If no name is specified, the default name is "common". Note that all code not included in any other segment will be included in the common segment, even if that code precedes the actual pragma (or there is no common pragma). The common segment should include only C definitions and types, and not data, variables or functions - those should be specified in the appropriate primary or secondary segment.

#pragma catapult secondary [name [(type)]] [attributes]

This pragma introduces a secondary segment, which should contain all the file scope data and code required for a secondary program, but no C main function. Instead, the segment should include a C function with the name of the segment that accepts a pointer to the specified type. The secondary segment is converted into a C program file. If no name is specified, the default name is "secondary". If no type is specified, the default type is the name of the segment with "_t" appended. There can be none, one or more secondary segments. Each secondary segment must be given a unique name. Each secondary segment should include all the data, variables and functions required by the secondary function. See also the section below on Sharing Data between Catapult Segments.

#pragma catapult primary [name] [attributes]

This pragma introduces the primary segment, which should contain all the file scope data, variables and code required for the primary program, including a C main function. The primary segment is converted into a C program file that includes either the binaries of all the secondary segments as an array of longs, or else the name of the overlay files containing this data. If no name is specified, the default name is "primary". There must be exactly one primary segment. If there are multiple primary pragmas, they must all be identical. See also the section below on Sharing Data between Catapult Segments.

Each pragma can specify additional attributes, which are described below.

There can be multiple instances of each pragma, but only one segment of each type will be created with each specified segment name, so each instance of a pragma type (**common**, **secondary** or **primary**) with the same name must be identical. Catapult will warn you if this is not the case.

The primary and secondary segment names must be unique, but the common segment may share the same name as another segment since the common segment is generated as a C header file (**.h**), and the primary and secondary segments are generated as C programs files (**.c**).

All the code introduced by the same pragma type with the same name will be combined in the same output file for compilation. For the common segment this file will be given the name of the segment plus a **.h** extension. For a primary or secondary segment it will have the segment name plus a **.c** extension. Catapult will complain if this would result in the same file name as the source file, to prevent accidentally overwriting that file. However, the binary output can be given that name by using the **binary** attribute (see below).

Catapult pragma attributes

Each of the catapult pragmas can also include various attributes:

address(value)

specifies the address to use for a secondary segment. The value can be a decimal value (e.g. 16384) or a hex value (e.g. 0x4000). This attribute applies

only to secondary pragmas.

`stack(value)`

specifies the stack size to use for a secondary segment. The value should be a decimal value (e.g. 500). This attribute applies only to secondary pragmas.

`mode(name)`

specifies the memory model to be used for the segment. For secondary segments, the mode name can be any of:

COMPACT or **CMM**	use the COMPACT memory model
TINY or **LMM**	use the TINY memory model
NATIVE or **NMM**	use the NATIVE memory model (P2 only)

For the primary segment, this can also be:

SMALL or **XMM SMALL**	use the SMALL memory model
LARGE or **XMM LARGE** or **XMM**	use the LARGE memory model

`options(list)`

specifies Catalina command line options to be added to the catalina command when compiling the segment.

For example:

```
options(-lc -lm -C TTY -O5)
```

All Catalina command line options are valid, but some should not be used since catapult will generate them automatically, such as **-o** for the name of the output. To define Catalina symbols use the usual **-C** syntax, and to define C symbols, use the usual **-D** syntax.

Note that options can be specified on common, secondary or primary pragmas - any options specified on common pragmas will be combined with the options specified for the primary or secondary pragma when the relevant programs are compiled. For instance, if a platform is included in a common pragma - e.g. options(**-C C3**) - it will apply to the compilations of both primary and secondary programs.

`binary(name)`

specifies the name of the binary output to be generated. Can be used in primary and secondary pragmas. If not specified, then the name of the segment is used. Use this attribute to name the binary output instead of specifying -o in the options attribute. Do not include an extension - the appropriate extension (i.e. .binary or .bin) will be added automatically.

`overlay(name)`

specifies the name of an overlay file to be generated. Can be used in

secondary pragmas only. Use this attribute to generate a file containing the overlay instead of an array of longs in the primary programs data segment. The overlay files must be placed on an SD card, and will be loaded automatically at execution time whenever the secondary program is started. The overlay name can include an extension. Note that this attribute CAN - but does not HAVE TO - be used in conjunction with the OVERLAY macros (described later). See the section on using Catapult with Overlay Files for more details.

The names used in the pragmas cannot be the same as any of the possible attributes (so for example, **address** cannot be used as a segment name).

Catapult macros

While the pragmas are all that is required to build a multi-model program, Catapult also provides a set of "convenience" macros, defined in the header file called *catapult.h*. These macros wrap the functions normally used to allocate the memory required by the secondary programs, and also to load, start and stop the secondary programs.

While not strictly required, using these macros rather than calling the functions directly is recommended because it allows the C file to also be compiled monolithically, and even executed on a single cog (although whether the program will execute CORRECTLY in this case depends on the program itself, and not on Catapult). See the **Compiling a Catapult C Program Monolithically** section below for examples of this.

Here is a description of the macros provided in *catapult.h*:

`CATAPULT_ERROR(name, address)`

name name of secondary

address correct address for the secondary

This macro prints an error message if it detects a secondary program has been compiled at the wrong address. It is defined automatically, but it can be redefined if (for example) your program has no HMI. In such a case it must signal the configuration error some other way. If you need to define your own version, first **#undefine** the existing definition.

`FIXED_START(name, arg, cog, result)`

name name of secondary function

arg name of variable used to pass arguments

cog cog to be used for secondary (which may be ANY_COG)

result int variable used to return cog actually used by secondary

This macro starts the named function as a secondary on the specified cog. The secondary will be loaded at the address specified in the secondary pragma, which must be unused. No checking is performed.

The actual cog used is returned in the int variable **result**.

The arg must be a structure of type name_t (or the type that was specified in the secondary pragma). It can be used to exchange data between the primary and the secondary. The arg structure must be in Hub RAM.

If multiple secondaries are started simultaneously using this macro, then it is up to the user to ensure the fixed Hub RAM space is both large enough for the secondary, and is not currently being used by either the primary or any other secondary.

```
RESERVE_AND_START(name, arg, cog, result)
```

name name of secondary function

arg name of variable used to pass arguments

cog cog to be used for secondary (which may be **ANY_COG**)

result int variable used to return cog actually used by secondary

This macro reserves local Hub RAM and then starts the named function as a secondary on the specified cog.

The secondary will be loaded at the address specified in the secondary pragma, which must be configured to match whatever address this macro allocates. This will be checked at run time, and the **CATAPULT_ERROR** macro will be used to report any mismatch.

The actual cog used is returned in the int variable **result**.

The arg must be a structure of type **name_t** (or the type that was specified in the secondary pragma). It can be used to exchange data between the primary and the secondary. The arg structure must be in Hub RAM.

If the secondary is to be started in a scope that may be exited while the secondary is still executing (e.g. in a function other than 'main') then this macro should not be used. Instead. use the **RESERVE_SPACE** macro to reserve Hub RAM in a suitable scope (e.g. **main**) and then use the **RESERVED_START** macro instead.

Since each secondary has its own space allocated, multiple secondaries can be started simultaneously using this macro.

When this macro is used, the address of the reserved space must appear in the **address** option of the catapult pragma for the secondary **name**. The **CATAPULT_ERROR** macro will be invoked automatically to report any mismatch. Initially, use a dummy address (e.g. **address(0x1000)** or similar) and then replace that address with whatever is reported by the **CATAPULT_ERROR** macro when the program is executed.

For example:

```
Error: secondary func_1 must specify address(0x00077ECC).
```

indicates that **address(0x00077ECC)** should be specified in the **secondary** pragma for **func_1**.

RESERVE_SPACE(name)

This macro reserves local Hub RAM and for the named secondary function. Use this macro to ensure that the Hub RAM is reserved in a suitable scope (e.g. **main**) and then the **RESERVED_START** macro can be used to actually start the secondary in any scope, such as a function other than **main**.

This macro must be placed where it is valid to have variable declarations - typically, at the beginning of a block - i.e. a section of code surrounded by **{** and **}**.

RESERVED_ADDRESS(name)

This macro returns the address of the local Hub RAM reserved for the named secondary program. It can be used (for example) to pass this address to a function outside the scope where the RESERVED_SPACE macro is declared.

RESERVED_START(name, arg, cog, result)

name name of secondary function

arg name of variable used to pass arguments

cog cog to be used for secondary (which may be **ANY_COG**)

result int variable used to return cog actually used by secondary

The **RESERVED_START** macro can be used to start the secondary in any scope, such as a function other than 'main'. The macro assumes that suitable Hub RAM has already been allocated using the **RESERVE_SPACE** macro, which must be visible (according to the usual C scope rules) from where this macro is used. This macro then starts the named function as a secondary on the specified cog.

The actual cog used is returned in the int variable **result**.

The secondary will be loaded at the address specified in the secondary pragma, which must be configured to match whatever address the **RESERVE_SPACE** macro allocates. This will be checked at run time, and the **CATAPULT_ERROR** macro will be used to report any mismatch.

The arg must be a structure of type **name_t** (or the type that was specified in the secondary pragma). It can be used to exchange data between the primary and the secondary. The arg structure must be in Hub RAM.

Since each secondary has its own space allocated, multiple secondaries can be started simultaneously using this macro.

When this macro is used, the address of the reserved space must appear in the **address** option of the catapult pragma for the secondary **name**. The **CATAPULT_ERROR** macro will be invoked automatically to report any mismatch. Initially, use a dummy address (e.g. **address(0x1000)** or similar)

and then replace that address with whatever is reported by the **CATAPULT_ERROR** macro when the program is executed.

For example:

```
Error: secondary func_1 must specify address(0x00077ECC).
```

indicates that **address(0x00077ECC)** should be specified in the **secondary** pragma for **func_1**.

OVERLAY_SPACE

This macro reserves one block of Hub RAM, of sufficient size to accommodate any of the secondary functions defined in the program.

Use this macro to ensure that the Hub RAM is reserved in a suitable scope (e.g. 'main') and then use the **OVERLAY_START** macro to actually start a secondary. **OVERLAY_ADDRESS**

This macro returns the address of the local Hub RAM reserved for all secondary programs using the **OVERLAY_SPACE** macro. It can be used (for example) to pass this address to a function outside the scope where the **OVERLAY_SPACE** macro is declared.

This macro must be placed where it is valid to have variable declarations - typically, at the beginning of a block - i.e. a section of code surrounded by **{** and **}**.

OVERLAY_START(name, arg, cog, result)

name name of secondary function

arg name of variable used to pass arguments

cog cog to be used for secondary (which may be **ANY_COG**)

result int variable used to return cog actually used by secondary

The **OVERLAY_START** macro can be used to start a secondary in any scope, such as a function other than **main**. The macro assumes that suitable Hub RAM has already been allocated using the **OVERLAY_SPACE** macro, which must be visible (according to the usual C scope rules) from where this macro is used. This macro then starts the named function as a secondary on the specified cog.

The actual cog used is returned in the int variable **result**.

The secondary will be loaded at the address specified in the secondary pragma, which must be configured to match whatever address the **OVERLAY_SPACE** macro allocates. This will be checked at run time, and the **CATAPULT_ERROR** macro will be used to report any mismatch.

The arg must be a structure of type **name_t** (or the type that was specified in the secondary pragma). It can be used to exchange data between the primary and the secondary. The arg structure must be in Hub RAM.

Since all overlays use the same Hub RAM, it is up to the user to ensure the overlay space is only used by one secondary at a time (hence the term 'overlay').

When this macro is used, the address of the reserved space must appear in the **address** option of the catapult pragma for the secondary **name**. The **CATAPULT_ERROR** macro will be invoked automatically to report any mismatch. Initially, use a dummy address (e.g. **address(0x1000)** or similar) and then replace that address with whatever is reported by the **CATAPULT_ERROR** macro when the program is executed.

For example:

```
Error: secondary func_1 must specify address(0x00077ECC).
```

indicates that **address(0x00077ECC)** should be specified in the **secondary** pragma for **func_1**.

STOP(cog)

cog cog to be stopped

This macro stops and unregisters an executing secondary cog. The cog to be stopped should be the one returned in the result variable used in a **FIXED_START**, **RESERVE_AND_START**, **RESERVED_START** or **OVERLAY_START** macro.

To demonstrate the use of the **RESERVE_SPACE**, **RESERVED_ADDRESS**, **RESERVED_START**, **RESERVE_AND_START** and **CATAPULT_ERROR** macros, this folder contains two versions of the demo program with deliberate address configuration errors - one for the Propeller 1 (*error_p1.c*) and one for the Propeller 2 (*error_p2.c*).

To compile and execute the program for the Propeller 1:

```
catapult error_p1.c
payload -i error_p1
```

or, to compile and execute the program for the Propeller 2:

```
catapult error_p2.c
payload -i error_p2
```

Catapult will use the **CATAPULT_ERROR** macro to report any errors that need correcting using a customized message. Edit the C source files to correct the error and recompile the program, which should then execute correctly, and generate the same output as *demo_p1.c* or *demo_p2.c*.

The use of the **OVERLAY_SPACE**, **OVERLAY_ADDRESS** and **OVERLAY_START** macros is similar to the **RESERVE_SPACE**, **RESERVED_ADDRESS** and **RESERVED_START** macros in these examples, except that the Hub RAM allocated by **OVERLAY_SPACE** will be used by all secondary programs (so only one secondary can be executing at any one time).

Sharing Data between Catapult Segments

It is important to note that in a multi-model C program the primary and secondary programs may not even share the same type of RAM (e.g. in XMM LARGE programs the data segment is in XMM RAM, and in other memory models the data segment is in Hub RAM). Even if this is true when the program is first created, it may not remain true if (for instance) the primary and secondary programs are originally defined to be a NATIVE programs, but later the primary program is changed to be an XMM LARGE program.

The recommended way to share data between the primary program and one or more secondary programs is to define a C structure in the common segment, and then declare a local instance of that structure in the primary segment (e.g. as a local variable in the main program) and then pass a pointer to that instance to the secondary program when it is initiated. Other mechanisms are possible, but this is the way Catapult is designed to work, and is the mechanism used in the demo programs.

If the primary program is not an XMM LARGE program, then shared data can also be defined at file scope in the primary segment, but DO NOT define it at file scope in the common or secondary segment - the common segment will be DUPLICATED in each of the primary and secondary programs when they are compiled separately, which means definitions and types will be common, but each program will end up with a DIFFERENT file scope. The common segment should be used only to declare common definitions and types, not variables or data.

By default, Catapult assumes that the data type to be shared between the primary program and a specific secondary program is declared in the common segment, using the name of the secondary program with **_t** appended. However, this can be overridden by using an explicit type name in the secondary pragma. The same data type, and even the same instance of the data type, can therefore be used to share data between the primary program and multiple secondary programs.

See the demo programs for examples of defining and declaring shared data types.

If any access control or protection is required to access the shared instance of the data type, it must be added by the primary and secondary program - for example, a lock could be initialized by the primary program and passed in the data structure itself, and then used by both the primary and secondary programs.

Compiling a Catapult C program monolithically

Because Catapult uses pragmas, which are ignored by C compilers that do not recognize them, a program designed for Catapult can also be compiled monolithically - i.e. as a single C source file - but you have to specify all the required compile time options yourself.

To compile the Propeller 1 demo monolithically:

```
catalina demo_p1.c -lc -lma -C C3 -C TTY -O5
```


or, to compile the Propeller 2 demo monolithically

```
catalina demo_p2.c -p2 -lc -lma -C P2_EDGE
```

When compiling a multi-model program monolithically, rather than using Catapult to separate and compile each segment separately, the following message will be generated:

```
#error directive: THIS FILE IS INTENDED TO BE COMPILED USING CATAPULT
```

This is really a warning, not an error. It is intended to be a reminder that a multi-model program designed to be executed on multiple cogs may not execute correctly when compiled and executed on a single cog - but the demo programs in this folder have been specifically designed to be able to do so - so the binaries generated by the catalina commands above can be executed.

Being able to compile the program monolithically even if it will NOT execute correctly is useful during development because it means the program is at least guaranteed to be syntactically correct before it is split up into its various component programs for execution in multiple modes on multiple cogs.

Using Catapult with XMM programs

The demo programs described above all execute from Hub RAM by default, so that they can be run on any Propeller 1 or Propeller 2 platform. But Catalina's multi-model support is most often used when the primary program is executed from XMM RAM and the secondary programs are loaded and executed in Hub RAM when their functionality is required, when speed or timing is critical, or when multi-threading or interrupt support is required (which are not available to XMM programs).

On Propeller platforms that have XMM RAM installed (which both the C3 and the P2-EC32MB version of the P2_EDGE do), modifying a Catapult program to execute the primary from XMM instead of Hub RAM can be as trivial as a one word change.

For instance, in demo_p1.c just change **COMPACT** in this line:

```
#pragma catapult primary main mode(COMPACT) binary(demo_p1)
```

to **SMALL** or **LARGE**, and then re-execute the catapult command:

```
catapult p1_demo.c
```

Similarly, in demo_p2.c, just change **NATIVE** in this line:

```
#pragma catapult primary mode(NATIVE) binary(demo_p2)
```

to **SMALL** or **LARGE**, and then re-execute the catapult command:

```
catapult p1_demo.c
```

In both cases, since the result is now an XMM program, the build_utilities script will need to be used to build an appropriate XMM loader. And remember to specify the correct cache size - since no cache size is specified for the C3, no SRAM cache should be specified, but for the P2_EDGE the default cache size of 8K must be specified.

To use a differed cache size, add a appropriate options to the primary pragma, such as:

```
#pragma catapult primary main mode(SMALL) options(-C CACHED_1K)
```

Using Catapult with Overlay Files

Catapult can also be used to build multi-model programs where the secondary programs are stored in overlay files instead of residing in an array of longs in the primary program's data segment. Such secondary programs will be loaded automatically from the SD card whenever the secondary is started.

This technique is especially useful on platforms which have no XMM RAM, since such secondary programs do not occupy Hub RAM space until they are loaded.

To build a secondary to use an overlay file instead of an array of longs in the program's data segment, specify the attribute **overlay**. It is also possible to mix secondary programs loaded from overlay files with those loaded from an array of longs.

Note that the OVERLAY macros and the overlay attribute are often used together, this is not mandatory - secondary programs loaded from both overlay files and from an array of longs can use Hub RAM allocated with the **OVERLAY_SPACE** macro, and programs from overlay files can also use Hub RAM allocated with the **RESERVE_SPACE** or **RESERVE_AND_START** macros.

Catalina has two functions that can be used to load overlays - one that uses the C stdio file system (**_load_overlay**), and one that uses the Catalina file system (**_load_overlay_unmanaged**). The code to load the overlay is automatically generated by the **spinc** utility, but the function to use can be selected by whether or not the Catalina symbol **FS_OVERLAY** is defined. If this symbol is defined (e.g. using the **options** attribute on the common or primary segment) then the Catalina file system overlay loader is used. This loader is smaller than the stdio file system loader, and is generally preferred on the Propeller 1, where using stdio consumes a significant amount of Hub RAM. However, note that when using the Catalina file system loader, the SD card must be manually mounted before any overlays can be loaded. This is typically done using the following code in the primary segment:

```
_mount(0, 0);
```

This is not required when using the C stdio file system overlay loader.

This folder contains two programs designed to demonstrate the use of overlay files - one for the Propeller 1 (*over_p1.c*) and another for the Propeller 2 (*over_p2.c*). The Propeller 1 version uses the Catalina file system overlay loader to minimize the program size, and the Propeller 2 version uses the C stdio file system version - but it could also use the Catalina file system loader if stdio support is not required.

To build the Propeller 1 program that uses overlays:

```
catapult over_p1.c
```

This will generate the following files:

over_p1.binary

func_p1.ovl
func_p2.ovl

Copy these to an SD card containing Catalyst. For example, if the SD card is mounted as drive **X**:

```
copy over_p1.binary X:over_p1.bin
copy func_1.ovl X:func_1.ovl
copy func_2.ovl X:func_2.ovl
```

Then insert the SD Card into the Propeller 1 and execute **over_p1**.

To build the Propeller 2 program that uses overlays:

```
catapult over_p2.c
```

This will generate the following files:

over_p2.bin
func_p2.ovl
func_p2.ovl

Copy these to an SD card containing Catalyst. For example, if the SD card is mounted as drive **X:**

```
copy over_p2.bin X:over_p2.bin
copy func_1.ovl X:func_1.ovl
copy func_2.ovl X:func_2.ovl
```

Then insert the SD Card into the Propeller 2 and execute **over_p2**.

Using Catapult with multi-threading

Catapult can also be used to build multi-model programs where the primary or secondary programs use multi-threading.

To specify multi-threading, simply specify the program should be compiled with the threads library by adding **-lthreads** to the appropriate pragma options. For example **options(-lthreads)**. This is particularly useful when the primary program is an XMM program, which would not normally support multi-threading.

To build a Propeller 1 XMM program that demonstrates a multi-threaded secondary:

```
catapult thread_p1.c
```

To build a Propeller 2 XMM program that demonstrates a multi-threaded secondary:

```
catapult thread_p2.c
```

Note that the program in both the above cases is an XMM program, so the **build_utilities** script will have to be executed to build a suitably configured XMM loader. In the Propeller 1 case above the utilities should be built for a C3 using FLASH and a 1K cache, and in the Propeller 2 case the utilities should be built for a P2_EDGE with PSRAM (i.e. a P2-EC32MB) and an 8K cache. In all cases, the catapult pragmas in the respective files can be modified to suit other platforms.

Note that calculating the stack required for a program that uses multi-threading can be difficult, because it depends on how many threads will be executing concurrently, as well as what they do (e.g. what local variables they allocate and what library functions they call). And remember that the stack space specified for each secondary in the Catapult pragmas has to be *at least as much* as the sum of stack space specified for each of the individual threads. This has to be done by trial and error. It is best to start by allocating too much stack space, get the multi-threading components working correctly (perhaps in isolation), and then reducing the stack space.

Using Catapult with the Parallelizer

Catapult can also be used to build programs where either the primary or secondary programs use the Catalina parallelizer. For more information on the Parallalizer, see the document **Parallel Processing with Catalina**.

To specify the primary or secondary program (or both) should be parallelized, simply specify the program should be compiled with the **-Z** option and also the threads library by adding **-lthreads** to the appropriate pragma options. For example **options(-Z -lthreads)**.

To build a Propeller 1 XMM program that demonstrates a multi-threaded primary:

```
catapult ll_p_p1.c
```

To build a Propeller 2 XMM program that demonstrates a multi-threaded primary:

```
catapult ll_p_p2.c
```

In both the above cases, the result is a normal program that can be simply loaded using payload. However, note that when you run these programs, the output will be garbled. This is intentional, and the reasons for it are described in more detail in the document **Parallel Processing with Catalina**.

To build a Propeller 1 XMM program that demonstrates a multi-threaded secondary:

```
catapult ll_s_p1.c
```

To build a Propeller 2 XMM program that demonstrates a multi-threaded secondary:

```
catapult ll_s_p2.c
```

Note that the program in both the above cases is an XMM program, so the **build_utilities** script will have to be executed to build a suitably configured XMM loader. In the Propeller 1 case above the utilities should be built for a C3 using FLASH and a 1K cache, and in the Propeller 2 case the utilities should be built for a P2_EDGE with PSRAM (i.e. a P2-EC32MB) and an 8K cache. In all cases, the catapult pragmas in the respective files can be modified to suit other platforms.

And again, note that when you run these programs, the output will be intentionally garbled.

There are some additional things to note when using the Parallelizer in a multi-model program:

- Catapult will issue a warning message about non-Catapult pragmas in the program being ignored. This is precisely what is required, since the Parallelizer pragmas must instead be processed by the Parallalizer (which will be invoked on the primary or secondary program if the **-Z** option is specified). Only the first such pragma will generate a warning.

- Calculating the required stack space can be difficult - it depends on the number of factory cogs and workers in the program, as well as what they do (e.g. what local variables they allocate and what library functions they call). And remember that the stack space specified for each secondary in the Catapult pragmas has to be *at least as much* as the sum of stack space specified for each of the individual workers and factories (or the default stack space, if no specific stack space is allocated - see the document **Parallel Processing with Catalina** for more information on default worker and factory stack space). This has to be done by trial and error. It is best to start by allocating too much stack space, getting the parallelized components working correctly (perhaps in isolation), and then reducing the stack space.

- The program must have enough free cogs to execute. Parallelized primary and secondary functions require at least two cogs each (rather than the usual one) - one for the main function, and one (or more) for the factory that executes the workers.

- Specifying the number of cogs that each factory in a parallelized function should use is recommended, or the first factory started will use all available cogs.

- Parallelized secondary functions should explicitly stop their factories before terminating - otherwise those cogs will never be released.

Using Catapult with Lua

Catapult can also be used to build programs that embed Lua scripts. It is particularly useful for building programs where the primary program is an XMM LARGE program that executes Lua scripts entirely from XMM, and the secondary programs execute from Hub RAM. In such cases, the slower speed of XMM execution is not really a problem, and the full capabilities of the Propeller and Catalina are available to the secondary programs.

To build a Propeller 1 XMM program that demonstrates a Lua primary and a multi-threaded secondary:

```
catapult lua_p1.c
```

To build a Propeller 2 XMM program that demonstrates a Lua primary and a multi-threaded secondary:

```
catapult lua_p2.c
```

In both the above cases, the result is an XMM LARGE program, so the **build_utilities** script must be executed to build an appropriate XMM loader. In the Propeller 1 case above the utilities should be built for a C3 using FLASH and a 1K cache, and in the Propeller 2 case the utilities should be built for a P2_EDGE with

PSRAM (i.e. a P2-EC32MB) and an 8K cache. In all cases, the catapult pragmas in the respective files can be modified to suit other platforms.

Using Catapult for Client/Server applications

Putting several of the above possibilities together, Catapult can also be used to build programs where one or more client programs execute in Hub RAM with full Propeller speed and functionality (including multi-threading) and use services provided by a server executing from XMM RAM to provide non-time critical support functions.

Catalina provides support for client programs calling server functions using the **Registry**, the same mechanism it provides for C programs calling functions provided by Cog-based **plugins**. The Registry provides a built-in mechanism for clients to locate and call services, and for protecting the server during concurrent service calls.

The client is implemented in C, but the server functions can be implemented either in C or Lua. C is smaller and also faster, but Lua is more functional and also more flexible - for example, the Lua server code can be loaded from SD Card at run-time, so the application can be easily extended or configured without re-compiling.

To build a Propeller 1 client/server program that demonstrates a C client executing from Hub RAM calling the server implemented as C functions executing from XMM RAM:

```
catapult srv_c_p1.c
```

To build the same C client but have the server implemented as Lua functions executing from XMM RAM:

```
catapult srv_l_p1.c
```

To build a Propeller 2 client/server program that demonstrates a C client executing from Hub RAM calling the server implemented as C functions executing from XMM RAM:

```
catapult srv_c_p2.c
```

To build the same C client but have the server implemented as Lua functions executing from XMM RAM:

```
catapult srv_l_p2.c
```

In all the above cases, the result is an XMM LARGE program, so the **build_utilities** script must be executed to build an appropriate XMM loader. In the Propeller 1 case above the utilities should be built for a C3 using FLASH and a 1K cache, and in the Propeller 2 case the utilities should be built for a P2_EDGE with PSRAM (i.e. a P2-EC32MB) and an 8K cache. In all cases, the catapult pragmas in the respective files can be modified to suit other platforms.

The above programs all demonstrate using the service profiles provided by default to service Catalina plugins, plus the addition of one profile that accepts a pointer to a Catapult shared data structure information for passing data to the service and returns an integer. They use the client/server functions implemented in the Catalina headers and libraries (see *include/service.h* for more details):

```
register_services(int lock, svc_list_t list);

void dispatch_C(svc_list_t list);

void dispatch_Lua(lua_State *L, svc_list_t list);
```

However, additional custom service profiles can be defined and used in conjunction with custom service dispatchers. This is illustrated in an example in the *demos/catapult/custom* directory, which has a demo program very similar to those above, but which adds an additional custom service profile which accepts two C **char *** parameters and returns an **int** - e.g:

```
int my_function(char *c1, char *c2);
```

See the customized C definitions and functions in the folder (described in more detail in *demos/catapult/custom/service.h*).

Using Catapult with ordinary C programs

While Catapult is primarily intended to simplify the process of compiling multi-model programs, it can be used with any Catalina C program.

To do this, simply include a catapult **primary** pragma in the main C program. The pragma does not need to be the first line, it can be anywhere in the C program - but any lines preceding the pragma will be automatically added to a header file called 'common.h' by default. While the resulting program will still generally compile ok, this is probably not what was intended.

The catapult primary pragma can then be used to specify any necessary command-line options, including any other C programs that need to be compiled along with main C program (for example, to always include the **-lthreads** library on a multi-threaded program, or **-lma** on a program that needs floating point).

Then simply execute catapult on the main C program.

Note that additional command line parameters CANNOT be specified on the Catapult command line, but the **CATALINA_DEFINE** environment variable can be used to do so.

For example, here are two trivial C files that illustrate this:

First, this is *print.c*:

```
#include <stdio.h>
void print(char *s) {
   printf(s);
}
```

And this is *main.c*:

```
#pragma catapult primary binary(main) options(-C NO_REBOOT -lci print.c)
extern void print(char *s);
void main() {
   print("Hello, World!\n");
}
```

Because of the catapult pragma in *main.c*, we can compile both the *main.c* and *print.c* C files with the one catapult command:

```
catapult main.c
```

Note that Catapult has no command line options to do things like specify the propeller platform, or whether it is a Propeller 1 or 2. Catapult determines if a program is to be compiled for a propeller 1 or 2 by checking whether the Catalina symbol **P2** is defined in **CATALINA_DEFINE**, and also by examining the options specified to see if they contain the option **-p2**.

However, if the catapult options do not define any specific propeller platform or HMI options, the normal **CATALINA_DEFINE** mechanism can be used to specify this. For instance, specify a Propeller 1 C3 platform using commands such as:

```
set CATALINA_DEFINE=C3
catapult main.c
```

Since it can affect the compilation process, Catapult will print the current value of **CATALINA_DEFINE** if it is set.

Using Catapult with Geany

Catapult does not have to be used from the command-line. It can also be used with Catalina's version of the Geany IDE. The recommended way to do this is to modify the project's **Build File** command.

Simply open an existing project file (or create a new one) and select the **Project -> Properties** menu item, then select the **Build** tab and modify the **Build File** command to:

```
catapult "%d\%f"
```

Note that Catapult cannot use the project's **Catalina Options** field. Instead, leave this field blank, and instead add a suitable catapult pragma to the program itself. For example, a simple "Hello, world!" program might look as follows:

```
#pragma catapult primary options (-p2 -lci) binary(hello_world)

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

Other Things to Note about Mult-Model Programs

Secondary programs do not load any plugins. This means that any plugins required by any secondary programs must be specified for both primary and secondary programs (so that they compile correctly) but must actually be loaded by the primary

program.

So, for example, if a secondary program is a **COMPACT** program but needs floating point support, then a floating point plugin MUST BE SPECIFIED for both the primary and secondary program, even if the primary program does not need it. Also, note that to be shared between the primary and secondary, this must be a floating point plugin, and cannot be the software only floating point library - i.e. the primary program must specify **-lma** or **-lmb** or (on the Propeller 2) **-lmc** instead of just **-lm**.

Similarly, if the secondary program needs file system support, the SD card plugin must be loaded by the primary program, either by linking it with the extended versions of the library - i.e. **-lcx** or **-lcix** instead of **-lc** or **-lci**, or by manually forcing the load of the SD card plugin via a **-C SD** option.

Similarly, if the secondary program needs CLOCK or RTC support, the primary program must specify **-C CLOCK** or **-C RTC**.

Similarly, the secondary programs must use the same HMI options as that used by the primary program, so the appropriate HMI options must be specified for the primary program even if it does not use it.

With Catapult, this is most easily done by specifying all such options in the common pragma, rather than in the primary and secondary pragmas.

Changing the way a program is started - e.g. using **payload** vs using **Catalyst** - can change the addresses allocated for local variables, which can impact the address allocated by the catapult macros that reserve Hub RAM, such as the **RESERVE_SPACE**, **OVERLAY_SPACE** and **RESERVE_AND_START** macros. This is because payload does not accept command line arguments, whereas Catalyst does, so when loading a program using Catalyst, stack space is allocated to hold the command line arguments (even if no arguments are specified). To eliminate this difference in programs that do not use command-line arguments, add **-C NO_ARGS** to the options specified in the **common** or **primary** pragma (it is added automatically to each **secondary** pragma).

