A Tutorial on Catalina Targets, Plugins and the Registry

BASIC CONCEPTS **2**
INSTALLING AND CONFIGURING THE GENERIC PLUGIN 3
THE GENERIC PLUGIN **5**
THE REGISTRY **6**
THE TARGET **8**
ADVANCED CONCEPTS **10**
EMM & XMM TARGETS 10
SUPPORTING MULTIPLE PROPELLER PLATFORMS 10

Basic Concepts

One of the design goals of Catalina was to make the environment in which C programs execute as platform-independent as possible. Catalina does this partly by implementing an effective *Hardware Abstraction Layer* (HAL) so that most C programs do not need to know the details of the drivers that perform various platform-dependent I/O functions on their behalf.

On the Propeller, with its multi-core capabilities, drivers are usually implemented in a separate cog to the main program. But as well as device drivers, the multi-core capabilities of the Propeller encourage the use of cogs for performing other functions as well – such as floating point functions, or various library functions that would be inefficient to code in C.

Catalina uses the term ***Plugin*** to refer to such components – i.e. a program designed to run on a separate cog. This term was chosen to highlight the fact that plugins are usually developed independently of the programs that make use of them, that they may be used by many programs, and also that they may be dynamically loaded as needed and unloaded again when no longer required (although many device driver plugins are loaded once at initialization time and never unloaded).

Catalina defines a common technique for keeping track of all the loaded plugins, and also a common method for communicating with those plugins. This is done by using the ***Registry*** – which is a just section of Hub RAM reserved for plugins. However, the registry is not usually accessed directly by application programs – a set of low level registry functions to do so are provided, but these are usually "wrapped" in a set of user-friendly C functions that are plugin-specific. For example, to access a Real-Time Clock plugin, a function such as *get_time()* would typically be provided, which implements all the low-level work of interacting with the clock plugin via the registry.

Finally, Catalina introduces the concept of a ***Target***. A target defines the execution environment for a Catalina C program – including the hardware configuration (pin definitions, physical addresses, clock speeds) the software configuration (drivers and plugins) and the kernel itself (LMM or XMM kernel, or one of the special function kernels such as the multi-threading kernel). Catalina groups all these components together into a ***Target Directory*** for convenience.

Each target directory usually supports one or more Propeller platforms. For example, the standard target directory provided with Catalina supports the *Hydra*, *Hybrid*, *TriBladeProp*, *DracBlade*, *RamBlade*, *Demo* board and *Morpheus* platforms. Each target directory will also contain a specific set of plugins supported by those platforms (e.g. the standard Target Directory includes various *HMI*, *SD card*, *Real-Time Clock*, and *Floating Point* plugins).

Catalina allows multiple target directories, but only one can be specified when compiling the final program executable. Having multiple target directories can come in handy when developing new plugins, or when supporting many different propeller platforms with significantly different capabilities. This will become more evident later in this tutorial, since we will use a separate target directory to illustrate how to build a plugin – without affecting the standard Catalina Target Directory.

Within each target directory, there may be support for different individual targets. Each such target specifies the kernel to be loaded, and each target knows how to load and initialize all the plugins supported by that target. For example, in the standard target directory there are LMM targets, EMM targets, XMM targets, and debug targets.

This last point is quite important – this is because even though a plugin is essentially a “stand-alone” program, they can be complex to load and initialize correctly. Often plugins must share not only the registry, but sometimes other areas of Hub RAM, or other Propeller resources such as I/O pins or locks. It is the individual *targets* that know how to load and initialize the various plugins correctly - especially since the load process can differ depending on the type of kernel used. For example, LMM programs are loaded quite differently to EMM programs or XMM programs.

In summary, developing a plugin to be used in conjunction with a Catalina C program consists of three parts:

1. The plugin itself. Plugins are usually implemented in PASM, and are often adapted from existing code.
2. A set of C “wrapper” functions that allows the services provided by the plugin to be more easily invoked from C.
3. One or more targets that support the plugin (i.e. that know how to load it).

This document provides a brief overview all these aspects of developing Catalina plugins. However, It is not a “step-by-step” or “hands-on” tutorial. Instead, it simply discusses various aspects of the process - but a fully documented example of a working “generic” plugin, plus some C wrapper functions that show how to invoke it, plus a Catalina target that loads it, plus some demo programs that use it – are all provided, and a referred to in various places in this document.

The remainder of this document also assumes you are at least slightly familiar with the C language, SPIN, PASM and also with the Propeller architecture.

Installing and Configuring the Generic Plugin

With this document should have come a zip file called **plugin.zip.** This file can simply be unzipped over an existing Catalina 2.6 (or later) release. The zip file is applicable to both Windows and Linux. The zip file contains two subdirectories:

custom	This is a complete – but *minimalist* – Target Directory. It contains a single LMM *target*, and a single *plugin* that this target knows how to load. The LMM target is called **default**, which means it does not normally need to be specified on the Catalina command line. The plugin is defined in the file **Catalina_Plugin.spin**, and is used by defining the symbol **PLUGIN** on the Catalina command line.
custom_demos	This directory contains a set of wrapper functions (defined in **generic_plugin.h** and implemented in **generic_plugin.c**) that allow easy access to the services provided by the plugin. It also contains a couple of demonstration programs that use them.

In each of these directories (and their subdirectories) you will find **README.txt** files, which contain detailed information in addition to what is contained in this tutorial.

To build both the plugin and the demonstration programs for a HYDRA, simply go to the **custom_demos** folder and enter the command:

```
build_all
```

If you do not have a HYDRA platform, you will first need to edit the configuration file **Catalina_Common_Input.spin** file and specify your own platform details (as a CUSTOM platform) and instead use a command such as:

```
build_all CUSTOM
```

If you don't do the configuration step first, you will see an error message during compilation such as:

```
ERROR : CUSTOM PLATFORM HAS NOT BEEN CONFIGURED YET!
```

Editing the **Catalina_Common_Input.spin** file to suit a custom platform is quite trivial – in most cases all you need to do is locate the following section of the file:

```
#ifdef CUSTOM
'
' Comment out the following line when you have configured the Custom platform:
'
ERROR : CUSTOM PLATFORM HAS NOT BEEN CONFIGURED YET!

'===============================================================================
'
' 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)
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)
'
' Custom platform Clock definitions:
'
CLOCKMODE = xtal1 + pll16x     ' (Custom)
XTALFREQ  = 5_000_000          ' (Custom)
CLOCKFREQ = 80_000_000         ' (Custom) Nominal clock frequency
```

For the purposes of this tutorial, all you need to do is set the clock frequency information correctly, and remove (or comment out) the ERROR line. You don't need to modify any pin definitions, because neither the demo programs nor the generic plugin use them.

Finally, note that the **build_all** script may need to be edited if Catalina is not installed in the default location – i.e. *C:\Program Files\Catalina* (under Windows) or *\usr\local\lib\catalina* (under Linux).

The Generic Plugin

Examine the file **Catalina_Plugin.spin** in the *custom* Target Directory. It embodies many of the aspects common to all plugins, and the rules that all plugins must follow:

- It contains a CON section. However, note that any constants defined here can only be used within the plugin itself, or within the target that loads it. If any of these constants are required in C, they will need to be redefined in a C header file.
- It contains an OBJ section with includes **Catalina_Common**. This is a "processed" version of the **Catalina_Common_Input.spin** configuration file, and is where all common platform dependent information (such as I/O pin definitions and clock speeds) will be stored – such things should not be duplicated in the plugin itself.
- It contains no SPIN methods, other than a **start** method which will be invoked by the target at load time (but note that if the plugin is to be re-loaded dynamically, this method cannot be called, but its functionality will have to be duplicated in C).
- It contains no VAR section. While it is possible to have a VAR block and use it during initialization, after initialization any variables defined in the VAR block would no longer exist in Hub RAM. Instead, a plugin that needs Hub RAM should be told what Hub RAM to use dynamically - either during startup (i.e. in the **start** method) or later by passing initialization parameters via the registry. Doing this allocation statically at initialization time is ok for plugins that are never intended to be unloaded or reloaded, but in this case the registry technique is used since we want to be able to reload and restart the plugin dynamically. However, this does not mean that we have to wait and initialize the plugin from C - the **start** method of this plugin actually uses the registry to initialize the plugin at load time.
- Now look at the DAT section. This contains the PASM that implements the plugin. The first thing the plugin does (see the code at label **entry**) is register itself. This is sometimes done in the start method, but for plugins that may be loaded dynamically, it is better if they register themselves on startup. All plugins are passed the address of the registry on startup – this will appear in the **par** register. From this, they use their cog id to calculate their registry block address that is used for all subsequent communications. See the next section for more details on registration.
- This plugin accepts various service requests, so the next part of the code is about waiting for a service request to appear in the registry (see the code at label **get_service**). In this case, the plugin accepts four different service requests. By convention, the service request code is put in the upper 8 bytes of the registry request block (note that plugins do not *need* to accept any service requests). See the next section for more details on using the registry.
- Not all plugins simply wait idly for service requests. Many spend most of their time doing other things (in the example, see the stub code at label **do_stuff**). However, all plugins that accept service requests should check periodically, since the kernel will halt when a service request is made until the request it acknowledged by the plugin.

- At the end of processing each service request, (or earlier, if the request is to be processed asynchronously) the plugin should acknowledge receipt of the request by writing zero to the same location (see the code at label **done_service**). See the next section for more details on using the registry.
- A service request can be invoked using a *short* request or a *long* request. However, this must be fixed at design time since the interpretation of the registry is slightly different for each kind of request, and a service cannot accept both. In this example, service 1 is a long request (it accepts a 32 bit pointer), service 2 is a short request (it accepts 24 bits of pin data), service 3 is a long request (it accepts 32 bits of pin data) and service 4 is a long request (it accepts a 32 bit pointer). Service 4 also illustrates how more than one 32 bit value must be passed – i.e. by passing an address of a temporary memory block that contains the actual parameters. See the next section for more details on the registry.
- A services that must return a result can also return them via the registry. See the next section for more details on using the registry.

Note that the plugin is intended mainly as an example – but it does implement some trivial functions – i.e. it toggles I/O pins. This means that it is possible to see the plugin in action when using the demo programs provided – e.g. on the Hydra or Hybrid, the operation of the plugin can be verified because it will toggle the Debug LED on or off.

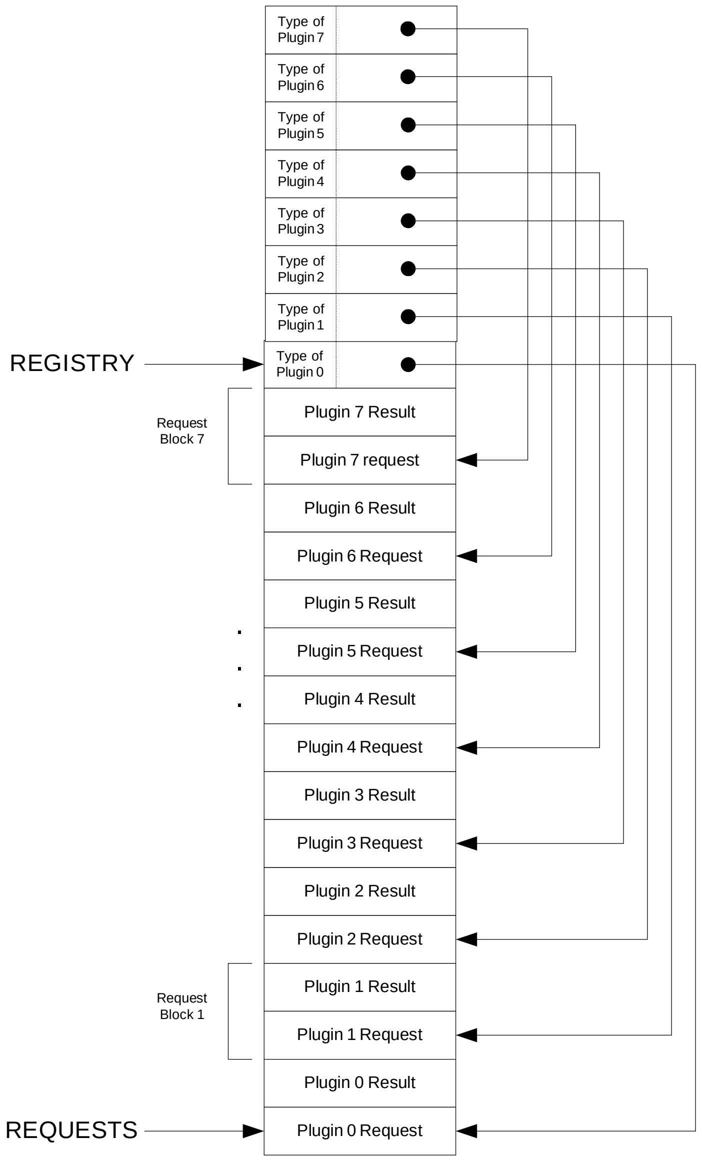
The Registry

At its simplest, the Catalina Registry is simply 8 consecutive longs located somewhere in Hub RAM (usually in the upper area of Hub RAM). The location is fixed only at compile time, and it is quite feasible for two different Catalina programs to use two different locations for the Registry (provided they not run at the same time).

Each plugin is told the location of the registry on startup, and from that (and its own cog id) deduces which of the 8 longs belongs to it specifically. This is referred to as the plugins **Registry Entry**. The Registry Entry is used for two purposes:

1. The upper 8 bits of the Registry Entry are used to indicate the type of plugin loaded into each cog. Zero indicates no plugin is loaded (although this is not a guarantee that the cog is actually unused – it simply means no plugin is registered in that cog). Setting this value is referred to as "registering" the plugin.
2. The lower 24 bits of the Registry Entry point to another location in Hub RAM – this is the plugin **Request Block**, and must consist of at least two consecutive longs. In theory, the Request Block can be longer than two longs, can exist anywhere in Hub RAM, and can be set up anytime up to the time the plugin is loaded – but it cannot usually be changed *after* the plugin is loaded, since most plugins read this location into a local cog register during initialization and thereafter do not refer to the Registry Entry. However, it is so common for plugins to use a simple Request Block consisting of two longs, that Catalina initializes each Registry Entry to point to a Request Block of two longs that also lives permanently in upper Hub RAM.

The basic Registry structure is defined in **Catalina_Common_Input.spin**. If you look in this file, you will see definitions for two important locations – **REGISTRY** and **REQUESTS**, and a method called **InitializeRegistry**. After the registry has been initialized, its structure will be as shown in the diagram below.



However, there is nothing to stop a target (or a C program) creating a new request block if it decides it needs one larger than 2 longs – all it has to do is update the Registry Entry prior to loading the plugin.

When using the registry to communicate with a plugin, a program writes a *request* to the *first* long in the plugin request block. A plugin that is offering interactive services must monitor this long, and process the service request whenever it sees a non-zero value in this long. To indicate the request is complete (or simply to acknowledge receipt of the request), the plugin should write zero to the same long (i.e. the *first* long of the request block). Before doing so, if it wishes to return a result (or a status), the plugin should write a “result” value to the *second* long of the request block.

As has been mentioned, there are several types of service request, dictated by the convention (and it is *only* a convention – plugins are free to use other methods) that plugins that offer multiple services all use the upper 8 bits of the request long as a service code, and the lower 24 bits of the request long in one of two ways:

- To hold up to 24 bits of data. This is known as a *short* request
- To hold the pointer to *another* data block somewhere in Hub RAM that contains the actual data. This is known as a *long* request. It is the responsibility of the caller to allocate the necessary space for this additional data block, and to guarantee that it remains valid for the lifetime of each service request.

As can be seen from examining the code, short requests are simpler and more efficient, and are generally preferred where possible.

Catalina provides C functions for locating the registry, registering and unregistering plugins, and for sending *short* or *long* service requests to a plugin. They are defined in the file **catalina_plugin.h** in the *Catalina\include* directory. The implementation of these functions can be found in the library source code in *Catalina\source\lib\catalina.*

Loading a plugin dynamically is usually as simple as using the *_coginit()* function and then registering the plugin (if it does not do so itself), and stopping it is usually simply a matter of calling *_cogstop()* on the cog in which it is running, and then unregistering it.

For the “generic” plugin provided (which accepts fours different service requests) a set of “wrapper” functions are defined in the *custom_demo* directory in the file **generic_plugin.h** (along with other plugin-specific constants that may be required), and implemented in the file **generic_plugin.c**.

It is recommended that these “wrapper” functions be compared with the implementation of each of the services in the file **Catalina_Plugin.spin**. In particular, note that the type of service (i.e. *short* or *long*) must match between the wrapper function and the service implementation.

The Target

The targets in the standard Catalina Target Directory are very complex – but this is largely because they each support so many different plugins, kernel types and platforms.

In order to show that the basics of a fully functional target is actually quite simple, the *custom* Target Directory provided with this tutorial is very simple – it supports only one target (a default LMM target) and that target only supports one plugin (our “Generic” plugin).

The required SPIN target program to do this (all targets are simply SPIN programs) is only a few lines long.

This target program file is in the *custom\input* subdirectory, and is called **lmm_default_Input.spin** (for an explanation of why this is so, see the README.txt file in that directory). Examine the target program file, and note the following, which are common features of all target files:

- It defines the clock frequency and stack size, but gets the values from **Catalina_Common.spin**. Note that the stack size here refers only to the stack used during the initialization phase (not the C program stack which is not constructed till much later).

- It includes **Catalina_Common.spin**, which contains all the platform-dependent features, such as I/O pin definitions etc.
- It includes **Catalina.spin**. This is the actual C code output by the Catalina compiler, wrapped up in a SPIN object. This file is generated on each compile, and (usually) deleted at the end of the compile (if you want to see its contents, include the **–u** flag when invoking the Catalina compiler).
- It includes **Catalina_LMM.spin**, which is the LMM Kernel.
- It includes all the plugins it might have to load (in this case, there is only the one – the generic plugin defined in **Catalina_Plugin.spin**).
- It contains one method (**Start**) which initializes the registry, allocates any Hub RAM required, loads and starts all the plugins, and then invokes the LMM kernel.
- Each plugin has an associated symbol (in this case **PLUGIN**) which controls whether or not the plugin is loaded (using **`#ifdef PLUGIN … #endif`** constructs).
- If a plugin requires any Hub RAM allocated to it, this is allocated *downwards* from the Registry **REQUESTS** block. The C program stack will be constructed starting just *below* any Hub RAM allocated to the plugins (or for the registry). This is different to when a plugin is started dynamically by a C program, where any Hub RAM required will have to be allocated in global RAM, or perhaps on the C stack. Plugins should not assume any particular Hub RAM locations (apart from a very few special cases where data blocks are permanently allocated for specific plugins in the upper Hub RAM area).

Advanced Concepts

EMM & XMM Targets

The *custom* directory provided as an example Target Directory only provides a single LMM target. While the plugin code itself is often identical for EMM, XMM and LMM targets, the means of loading and interacting with plugins may differ.

Refer to the standard Catalina Target Directory for examples of other targets, and how to load plugins for other targets.

Supporting multiple Propeller platforms

The **Catalina_Common_Input.spin** file provided in the *custom\input* subdirectory illustrates the basics of how to support multiple platforms. It includes a default set of platform-dependent definitions (suitable for a HYDRA) and also a customizable set, which can be activated by defining the **CUSTOM** symbol on the command line.

This is intended to show how different platforms are supported – i.e. by using constructs like **`#ifdef CUSTOM … #endif`**. The first step in supporting a new platform is to determine a suitable symbol for it (in this case **CUSTOM**) and then use it consistently in all target files and plugins.

However, all SPIN files in which the symbol is used must be preprocessed as part of the compilation process. Generally, this means they should live in the *custom\input* directory rather than simply in the *custom* directory. For example, if the generic plugin provided had to be made customizable to suit different platform, it may need to contain such **`#ifdef PLATFORM … #endif`** constructs – although in general it is better to “factor out” such differences and include them in **Catalina_Common_Input.spin**. However, sometimes the differences between platforms are so great that this is not possible.

More details on the “input” directory are provided in the README.txt file in the *custom\input* subdirectory.