Plugins and Targets

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 a *Hardware Abstraction Layer* so that most programs do not need to know the details of the platform-dependent code (e.g. hardware drivers) that perform various functions on their behalf.

On the Propeller, with its multi-core capabilities, such code is often implemented in one or more separate cogs to the main program. As well as hardware 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 other functions that would be too inefficient to code in C, or which can execute in parallel or asynchronously.

Catalina uses the term ***Plugin*** to refer to such components – i.e. any program designed to run on a separate cog. This term was chosen to highlight the fact that plugins are often developed independently of the programs that make use of them, that can be used by many programs, and also that may be dynamically loaded as needed and unloaded again when no longer required (although most 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 them. This is done by using a **Registry** – which is a just section of upper Hub RAM reserved for use by the plugins. The Registry is described in detail in the next section. However, the Registry is not usually accessed directly by application programs – instead, a set of low level registry functions to do so are provided, often written in PASM, then "wrapped" in a set of user-friendly functions that are plugin-specific. For example, to access a Real-Time Clock plugin, a function such as **get_time()** might be provided, which implements all the low-level work of interacting with the clock plugin using the Registry.

To manage plugins, Catalina also introduces the concept of a **Target**. A target defines a execution environment provided by a particular Propeller platform – including the hardware configuration (e.g. pin definitions, physical addresses, clock speeds) the software configuration (e.g. drivers and other plugins) and the kernel itself (e.g. CMM, LMM, NMM or XMM kernels, or one of the special function kernels such as the multi-threading kernels). Catalina groups all these components together into a **Target Directory** (sometimes called a **Target Package**) for convenience.

Each Target Directory may support one or more Propeller platforms, and may contain either (or both) a *p1* and a *p2* sub-directory, depending on whether it includes support for Propeller 1 or Propeller 2 platforms (or both). For example, the default Target Directory includes support for the *C3*, *FLIP*, *Hydra*, *Hybrid*, *TriBladeProp*, *DracBlade*, *RamBlade*, and *Demo* platforms on the Propeller 1, and the *P2 Edge* and *P2 Evaluation* platforms on the Propeller 2. Each sub-directory typically contains a set of plugins supported by one or more of the platforms (e.g. various *HMI*, *SD card*, *Real-Time Clock*, and *Floating Point* plugins). Each sub-directory will also have one or more include files which contain the specific configuration data for the platform (e.g. *C3_XXX.inc*, *HYDRA_XXX.inc* on the Propeller 1, or *P2EDGE.inc* or *P2EVAL.inc* on the Propeller 2).

Catalina supports 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 which have significantly different plugins or capabilities.

Within each Target Directory, there may be multiple Targets. Each Target determines the kernel and the plugins that will be loaded, whether or not debugger support will be included, whether the cache is included, and how the plugins supported by the target must be loaded and initialized.

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

A plugin to be used in conjunction with a Catalina C program typically consists of three parts:

1. The plugin itself. Plugins are usually implemented in PASM (although they can also be written in other languages such as Spin), 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 is typically accomplished on the Propeller 1 via the *Extras.spin* file in the target directory, or on the Propeller 2 via the *plugins.inc* file in the relevant Target Directory.

The Registry

At its simplest, the Catalina **Registry** is simply a number of consecutive longs – one long per cog[1] – located somewhere in Hub RAM (usually in the upper area of Hub RAM). The location is fixed at compile time, and it is quite feasible for different Catalina programs to use different locations for the Registry provided they do not execute on the same Propeller at the same time.

Each long is referred to as the plugin’s **Registry Entry**. The Registry Entry contains the following information:

1. The upper 8 bits of the Registry Entry are used to indicate the type of plugin loaded into each cog. The value **$FF** (in Spin) or **0xFF** (in C) indicates no plugin is loaded (although this is not a guarantee that the cog is actually unused – it simply means no plugin has been registered in that cog – the cog may be being used for some other purpose). Setting this value is referred to as “registering” the plugin.

1 The registry size is always 8 on the Propeller 1. On the Propeller 2 the size is defined by the constant **MAX_COGS** in *constants.inc*, which is currently 8, but which might be 16 in a future version of the chip.

2. The lower 24 bits of the Registry Entry point to another location in Hub RAM – this is the plugin's **Request Block**, and consists of at least two consecutive longs. Normally, these **Request Blocks** exist immediately *above* the Registry itself. 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 and initialized – but it cannot usually be changed *after that*, since most plugins read this location into a local cog register only during initialization and thereafter may not refer to the Registry Entry at all. 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 live permanently in upper Hub RAM.

Each plugin is told the location of the Registry on startup, and from that (plus its own cog id) it determines which Registry Entry belongs to it specifically, and either the startup code registers the plugin once it determines it has started, or the plugin registers itself.

While the Registry can be used directly to locate and communicate with plugins, there is another set of important entries immediately *below* the **Registry** that offer a further level of abstraction and concurrency protection when multiple programs may make concurrent requests. Each word immediately *below* the Registry can be a **Service Entry** and is referenced by a service id which is the negative offset (in words) from the Registry. See the diagram below, which shows the whole Registry structure.

Each Service Entry contains the following information:

1. The upper 4 bits of the Service Entry are used to specify the cog (i.e. the plugin) that offers the service (the upper bit is set if the service entry is unused).
2. The next 5 bits are used to hold the id of a lock[2] that can be used to prevent contention in accessing the service (or all bits set if no lock is required).
3. The next 7 bits represent the request id in the plugin that performs the service.

Note that request ids are local to each plugin, and so the same request id may be used by different plugins to mean different things, but the service id is unique across *all* plugins. So, for example, service **svc** might be represented (in Spin) by **word[REGISTRY-svc]** and this word might contain the value **cog<<12 + lock<<7 + req**, which indicates that service **svc** is actually request id **req** offered by plugin **cog**, and calling it must be protected by first successfully locking the specified **lock**.

This additional level of indirection allows concurrency protection and also more flexibility as to which services are offered by which plugin. As an example (this is a genuine use case) the service that gets the time can be offered by *either* a stand-alone real-time clock plugin *or* by code that exists in the same plugin as the another plugin such as he SD plugin, which saves a cog.

On the Propeller 1, the basic Registry structure and support code is defined in *Catalina_Common.spin*. On the Propeller 2 it is defined in *common.inc* and *plugsup.inc*. If you examine these files, you will see definitions for two important locations –

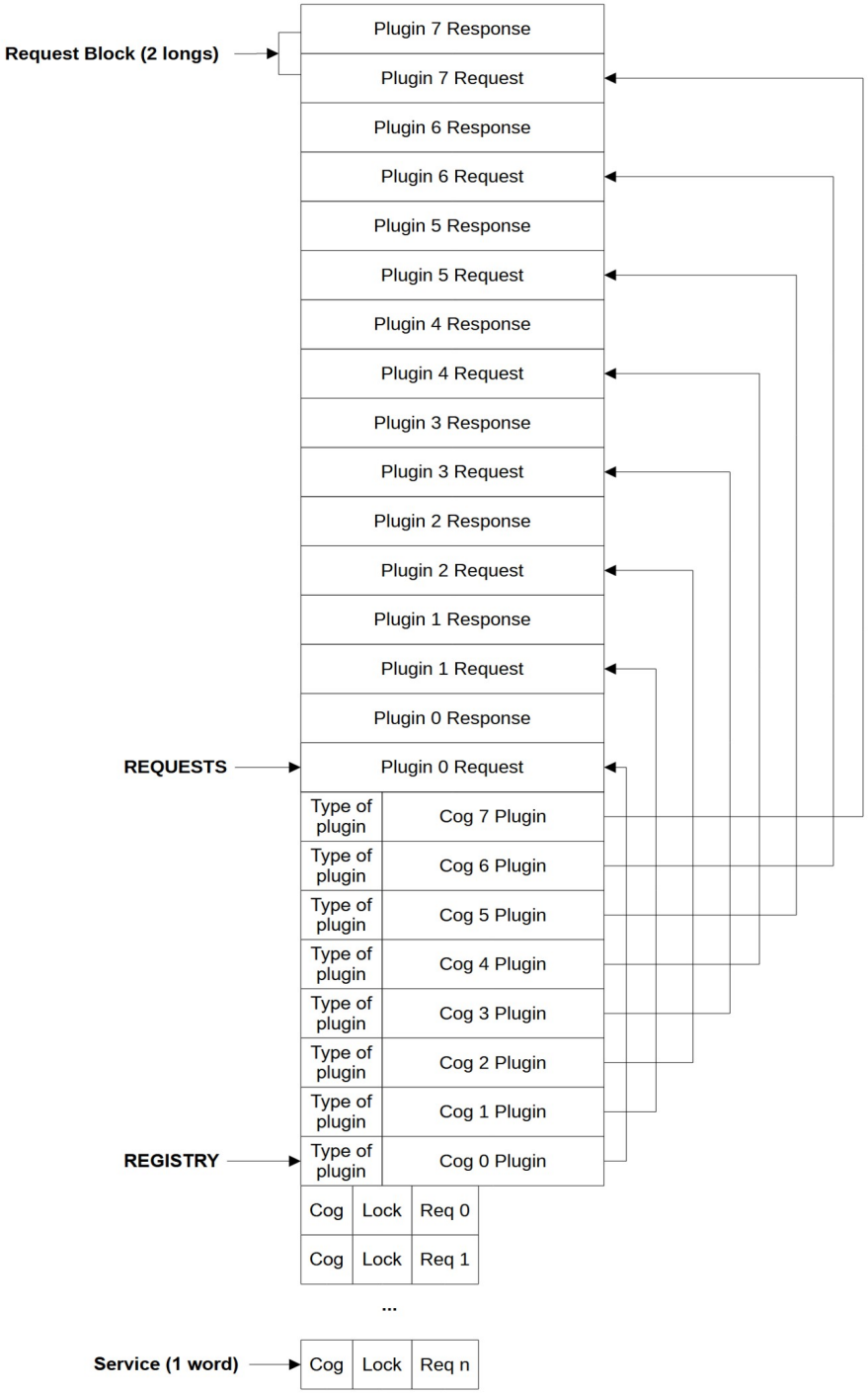
2 On the Propeller 1, there are only 8 locks, but on the Propeller 2 there are 16, so to make the code common between the two, 5 bits are used on both platforms.

REGISTRY and **REQUESTS**, and (on the Propeller 1) a Spin method called **InitializeRegistry**. or (on the Propeller 2) a PASM function called **INITIALIZE_REGISTRY**.

These files also contain the constants that identify the type of plugin. For historical reasons, these constants are all called **LMM_XXX**[3]. The local request ids are *not* defined in these files, but are instead typically defined in the plugin files themselves plus various C header files, but the service ids – which must be globally unique *are* defined in these files, and are called **SVC_<NAME>** (e.g. **SVC_RTC_GETTIME**).

Once the Registry has been initialized, its structure will be as shown in the diagram below (assuming there are 8 cogs):

3 LMM (Large Memory Mode or Large Memory Model) is what allows PASM instructions to be executed from Hub RAM on the Propeller 1, and therefore makes C programming practical. The Propeller 2 introduced a Native mode (known as NMM), but also still supports LMM.



To use 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 request whenever it sees a non-zero value in this long. To indicate the request is complete (or to tell the calling program that it can proceed), 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.

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 offering multiple services use the upper 8 bits of the request long as a request id, 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 which 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.

Short requests are simpler and more efficient, and are generally preferred where possible. More details on service requests are given in the next section.

Note that there is nothing to stop a target (or a plugin itself) 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 the plugin being used.

Also, it is possible for a target or plugin to create an entirely separate Hub RAM buffer to use for communication. In such cases the plugin can store the location of this buffer in either the lower 24 bits of the Registry Entry or in the Request Block, so that programs can locate it at run-time.

Registry, Plugin and Service functions

Catalina provides C functions and macros for interacting with the Registry. They can be used to record or identify which Plugins are currently loaded, to manage the Request Blocks for each Plugin, and to request the services provided by the Plugins.

These functions are defined in the include file *plugin.h*. The functions are divided into three logical layers:

- Basic Registry Management
- Plugin Requests
- Service Requests

The Basic Registry Management functions are:


```
unsigned _registry();
```

This function returns the address of the registry. This is required (for example) to be passed to a cog when starting a dynamic kernel to execute C code on that cog.

```
void _register_plugin(int cog_id, int plugin_type);
```

This function can be used to register that a plugin of a specified type is running on a particular cog. Plugins must be registered before requests can be sent to them.

```
void _unregister_plugin(int cog_id);
```

This function can be used to unregister a plugin.

Plugin types 0 to 127 are reserved for various basic Catalina plugins (see the file *plugin.h* for a complete list of those currently allocated), but plugin types 128 to 254 are free for users to define for their own purposes. Plugin type 255 is reserved, and indicates no plugin is loaded.

The Basic Registry Management macros are:

```
REGISTRY_ENTRY(c)
```

This macro returns the registry entry for cog c. The parameter can be from 0 and 7 on the Propeller 1, or 0 to **MAX_COGS** on the Propeller 2 – any other value will return an undefined result. The result is an unsigned value.

```
REGISTERED_TYPE(c)
```

This macro returns the registered type for cog c. The parameter can be from 0 and 7 on the Propeller 1, or 0 to **MAX_COGS** on the Propeller 2 – any other value will return an undefined result. The result is an unsigned value.

```
REQUEST_BLOCK(c)
```

This macro returns a pointer to the request block reserved for cog c. The parameter can be from 0 and 7 on the Propeller 1, or 0 to **MAX_COGS** on the Propeller 2 – any other value will return an undefined result. The request block structure pointed to is defined as:

```
typedef struct {
   unsigned int request;
   unsigned int response;
} request_t;
```

The Plugin Request functions are:

```
int _locate_plugin(int plugin_type);
```

This function can be used to find a cog on which a plugin type is executing. Note that if there is more than one plugin of a specified type executing, only the first will be found.

```
int _short_plugin_request (long plugin_type, long code, long param);
```

This function can be used to send a “short” request to a plugin specified by type (e.g. **LMM_HMI**). Short requests have a code and up to 24 bits of parameter. Note that the meaning of the code and the parameters is plugin-

dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

```
int _long_plugin_request (long plugin_type, long code, long param);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). This type of long request has a code and ***one*** 32 bit parameter. Note that the meaning of the code and the parameter is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

```
int _long_plugin_request_2 (long plugin_type,
                            long code,
                            long par1,
                            long par2);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). This type or long request has a code and ***two*** 32 bit parameters. Note that the meaning of the code and the parameters is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request ids.

```
float _float_request (long plugin_type, long code, float a, float b);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). This type of long request has a code and ***two*** 32 bit ***floating point*** values as parameters. Note that the meaning of the code and the parameters is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request ids.

The Service Requests functions are:

```
int _short_service (long svc, long param);
```

This function can be used to send a short request for a specific service (e.g. **SVC_T_CHAR**). Short requests have a code and up to 24 bits of parameter. Note that the meaning of the parameter is service-dependent, and also that different services may require short or long requests to be used.

```
int _long_service (long svc, long param);
```

This function can be used to send a long request for a specific service (e.g. **SVC_RTC_SETFREQ**). Long requests have a code and up to 32 bits of parameter. Note that the meaning of the parameter is service-dependent, and also that different services may require short or long requests to be used.

```
int _long_service_2 (long svc, long par1, long par2);
```

This function can be used to send a long request for a specific service (e.g. **SVC_SD_READ**). This type of long request has a code and ***two*** 32 bit parameters. Note that the meaning of the parameters is service-dependent.

```
float _float_service (long svc, float a, float b);
```

This function can be used to send a long request for a specific service (e.g. **SVC_FLOAT_ADD**). This type of long request has a code and ***two*** 32 bit

floating point values as parameters. Note that the meaning of the parameters is service-dependent.

The Service Requests macros are:

SERVICE_ENTRY(s)

This macro returns the registry entry for service **s**. The parameter should be from 1 to **SVC_MAX** – any other value will return an undefined result. The result is an unsigned short value.

SERVICE_COG(s)

This macro returns the cog containing the plugin which implements service **s**. The parameter should be from 1 to **SVC_MAX** – any other value will return an undefined result. The result is an unsigned value. A value of **0xF** indicates the service is not currently implemented by any loaded plugin.

SERVICE_LOCK(s)

This macro returns the lock that must be successfully set to gain access to service **s**. The parameter should be from 1 to **SVC_MAX** – any other value will return an undefined result. The result is an unsigned value. A value of **0x1F** indicates the service is not currently implemented by any loaded plugin.

SERVICE_CODE(s)

This macro returns the request that will be sent to the plugin to request service **s**. The parameter should be from 1 to **SVC_MAX** – any other value will return an undefined result. The result is an unsigned value. A value of **0x00** indicates the service is not currently implemented by any loaded plugin.

The following miscellaneous utility function are provided:

char *_plugin_name (int type)

This function returns a pointer to a human-readable name for the plugin type. For example “Real-Time Clock” or “Gamepad”.

Note that the same plugin functions can generally be requested using either Plugin Requests or Service Requests. The advantage of using Service Requests is that they implement concurrency control (necessary if you have multiple threads or multiple cogs executing C programs), that you do not need to know the plugin type to request a service (i.e. allowing the same service to be implemented by different plugins in different targets), and also that Service Request access is slightly faster.

Services 1 to 64 are predefined to mean various basic Catalina services (see the file *plugin.h* for a complete list), but services 65 to **MAX_SVC** are free for users to define for their own purposes.

AN IMPORTANT NOTE ABOUT REGISTRY ACCESS: When accessing the registry, any addresses used in the registry, or in a plugin or service request, ***must be HUB RAM addresses***. This is because plugins are normally

implemented as Spin/PASM programs that have ***no access to XMM RAM***. For example, if a service requires a parameter that represents an address where the plugin expects to find data to process, the address ***must be in Hub RAM***. If the data is actually located in XMM RAM, it must be copied to Hub RAM before the service request.

