Handling Mount Requests in Your Resource Manager

Authored by: **Thomas Fletcher**
Updated by: **Thomas Fletcher**

Mount requests can provide a very convenient and flexible interface for programs that need to enable and disable components of their resource managers' systems.

But the procedure remains a bit of a black art, partly because it's one of the less-documented features of QNX Neutrino. (It was added after our initial documentation was written.) Hopefully this article will help to shed some light on this relatively simple and useful feature.

Mount Components

There are three main areas to consider when using and building mount functionality into your resource manager. They are:

- mount utility
- mount function call
- mount callout in the resource manager

These components represent the stream of communication for the mount request. For the purposes of this article, we'll start in the middle with the mount function call and work our way out.

The mount function call is at the bottom of the mount utility and represents a client's access point to the resource manager. The function is implemented in the C library, is defined in <*sys/mount.h*>, and described in the *Neutrino Library Reference*.

Mount Function Call

Here's a typical example of a mount function call:

```
int mount(const char *special_device,
          const char *mount_directory,
          int flags,
          const char *mount_type,
          const void *mount_data,
          int mount_datalen);
```

Most of the arguments are clear and well documented, but one that may need a bit more explanation is the interpretation of the flags field.

To support the mounting of non-existent special devices (like NFS devices) or arbitrary strings (such as shared object or DLL names), we needed to massage the arguments to this function slightly. Such adjustment is necessary because the mount utility has two methods (**-T** and **-t**) for specifying the mount type.

In the general case where **special_device** is an actual device, a typical mount utility command may look like:

```
% mount -t qnx4 /dev/hd0t77 /mnt/fs
```

In this case the **special_device** is */dev/hd0t77* and the **mount_directory** is */mnt/fs* and the mount type is qnx4. In this case, the mount request should be directed only to the process responsible for managing the **special_device**. That is to say the resource manager that has provided the */dev/hd0t77* path into the pathname space. In this type of scenario the resource manager is given an OCB for the **special_device** (*/dev/hd0t77*) rather than the string */dev/hd0t77*. This simplifies the processing in the resource manager since having the OCB, which is an internal data pointer, for the special device implies that the server doesn't have to recursively communicate with itself to get a handle for the device.

A less frequently used, but very useful case, is where the **special_device** is not an actual device. An example of this situation may look like:

```
% mount -T io-net /lib/dll/npm-qnet.so
```

Note that mountpoint is missing from the command line. In this case, NULL (or */*) acts as an implied **mount_directory** which will cause the process handling the request (ie. **io-net**) to take the appropriate action when it receives the mount request. The special_device is */lib/dll/npm-qnet.so* and the type is **io-net**. In these cases, you want to avoid having the special device interpreted as being provided by the same process that will handle the mount request. So while the file */lib/dll/npm-qnet.so* is probably handled by some filesystem process, we're actually interested in mounting a network interface that is managed by the network manager process. Ideally, the mount callout will receive only the special device string */lib/dll/npm- qnet.so* and not the **OCB** for the device.

The behavioral difference between the **-t** and **-T** options for the mount utility can be obtained by **or**'ing in **_MFLAG_OCB** to the standard mount flags parameter. If you ***don't*** want to use the **OCB** method of performing the mount request, use **-T ==> _MFLAG_OCB**.

Mount requests are connection requests, which means they operate on a path in the same way that the **open**() or **unlink**() calls do. The requests are sent along the path specified by **dir**. When a resource manager receives a request to mount something, the information is already provided in the same way that it would be for an **open**() for creation request, namely in the **msg->connect.path** variable.

Mount in the Resource Manager

Your resource manager will be called upon to perform a mount request via the **mount**() **resmgr_connect** function callout, defined as:

```
int mount(resmgr_context_t *ctp,
          io_mount_t *msg,
          RESMGR_HANDLE_T *handle,
          io_mount_extra_t *extra);
```

The only field here that differs from the other connect functions is the **io_mount_extra_t** structure. It's defined in <*sys/iomsg.h*> as:

```
typedef struct _io_mount_extra {
 uint32_t flags; /* _MOUNT_? or ST_? flags above */
 uint32_t nbytes; /* Size of entire structure */
 uint32_t datalen; /* Length of the data structure following */
 uint32_t zero[1];
 union { /* If EXTRA_MOUNT_PATHNAME these set*/
  struct { /* Sent from client to resmgr framework */
   struct _msg_info info; /* Special info on first mount, path info on
remount */
  } cl;
  struct { /* Server receives this structure filled in */
   void * ocb; /* OCB to the special device */
   void * data; /* Server specific data of len datalen */
   char * type; /* Character string with type information */
   char * special; /* Optional special device info */
   void * zero[4]; /* Padding */
  } srv;
 } extra;
} io_mount_extra_t;
```

This structure will be provided with all of the pointers already resolved, so you can use it without doing any extra fiddling.

Where:

- **flags** - Flag fields provided to the mount command containing the common mount flags defined in <*sys/mount.h*>
- **nbytes** - Size of the entire mount-extra message and is: `sizeof(_io_mount_extra) + datalen + strlen(type) + 1 + strlen(special) + 1`
- **datalen** - Size of the data pointer.
- **info** - Used by the resource manager layer.
- **ocb** - **OCB** of the special device if it was requested via the **_MOUNT_OCB** flag. NULL otherwise.
- **data** - Pointer to the user data of length **datalen**.
- **type** - NULL terminated string containing the mount type, such as **nfs**, **cifs**, **qnx4**.
- **special** - NULL terminated string containing the special device if it was requested via the **_MOUNT_SPEC** flag. NULL otherwise.

In order to receive mount requests, the resource manager should register a NULL path with an **FTYPE** of **_FTYPE_MOUNT** and with the flags **_RESMGR_FLAG_FTYPEONLY**. This would be done with code that looks something like:

```
mntid = resmgr_attach(dpp, /* Dispatch pointer */
                      &resmgr_attr, /* Dispatch attributes */
                      NULL, /* Attach at "/" */
                      /* We are a directory and only want matching
ftypes */
                      _RESMGR_DIR | _RESGMR_FTYPE_ONLY,
                      _FTYPE_MOUNT,
                      mount_connect, /* Only mount filled in */
                      NULL, /* No io handlers */
                      &handle) /* Handle to pass to mount callout */
```

Again, we're attaching at the root of the pathname space so that we'll be able to receive the full path of the new mount requests in the **msg->connect** structure.

Adding the **FTYPEONLY** flag will ensure that this request will only be used when there is a **FTYPE_MOUNT** style of connection. Once this is done, the resource manager is ready to start receiving mount requests from users.

An outline of a sample mount handler would look something like this:

```
int io_mount( ... ) {
 Do any sanity checks that you need to do.
 Check type against our type w/ strcmp(), since there may be no name
for REMOUNT/UNMOUNT flags.
 Error with ENOENT out if no match.
 If no name, check the validity of the REMOUNT/UNMOUNT request.
 Parse arguments or set up your data structure.
 Check to see if we are remounting (_MOUNT_REMOUNT)
 Change flags, etc., if you can remount.
  Return EOK.
 Check to see if we are unmounting _MOUNT_UNMOUNT
  Change flags, etc., if you can unmount.
  Return EOK.
 Create a new node and attach it at the msg->connect.path point (unless
some other path is implied based on the input variables and the
resource manager) with resmgr_attach().
  Return EOK.
}
```

What's important to notice here is that each resource manager that registers a mount handler will potentially get a chance to examine the request to see if it can handle it. This means that you have to be rigorous in your type and error checking to make sure that the request is indeed destined for your manager. If your manager returns anything other than ENOSYS or ENOENT it's assumed that the request was valid for this manager, but there was some other sort of error. Only errors of ENOSYS or ENOENT will cause the request to "fall through" to other resource managers.

When you unmount, you would perform any cleanup and integrity checks that you need and then call **resmgr_detach**() with the **ctp->id** field. In general, you should only support umounted calls on the root of a mounted filesystem.

Mount Utility

By covering the mount library function and the operation in the resource manager, we have pretty well covered the mount utility. The usage for the utility is shown here for reference:

```
mount [-wreuv] -t type [-o options] [special] mntpoint
mount [-wreuv] -T type [-o options] special [mntpoint]
mount
```

-t Indicates the special device, if it is present, is generally a real device and the same server will handle the mountpoint.

-T Indicates the special device is not a real device but rather a key for the server. The server will autocreate an appropriate mountpoint if **mntpoint** is not specified.

-v Increases the verbosity

-w Mount read/write

-r Mount read only

-u Mount for update (remount)

However, if you're writing a mount handler, there may be occasions when you want to do custom parsing of arguments and provide your own data structure to your server. This is why the mount command will always first try and call out to a separate program named **mount_XXX**, where **XXX** is the type that you specified with the **-t** option. To see just what would be called (in terms of options, etc.), you can use the **-v** options, which should provide you with the command line that would be **exec**()'ed.

In order to help with the argument parsing, there is a utility function that can be called to help strip out common flags. The function is defined in <*sys/mount.h*> as:

```
char *mount_parse_generic_args(char *options, int *flags);
```

and you would use it in the following manner:

```
while ((c = getopt(argv, argc, "o:"))) {
switch (c) {
case 'o':
if ((mysteryop = mount_parse_generic_args(optarg, &flags))) {
//You can do your own getsubopt type processing here
//mysteryop is stripped of the common options.
}
break;
}
}
```

Currently, the stripped options are:

"ro", "rw", "noexec", "exec", "nosuid", "suid",

"nocreat", "creat", "noatime", "atime", "remount",

"update", "before", "after"

Which are in turn converted to appropriate _**MOUNT_XXX** values.

Conclusions

Adding mount capability to your resource manager is relatively pain-free. Along with providing a convenient user interface, it can be used to some extent as a mechanism for implementing dynamic changes to a running system.

For more information on writing a resource manager, read the documentation "Writing a Resource Manager." Rob Krten's book, "Getting Started with Neutrino 2.0" is also an invaluable reference.