Introduction to Resource Managers

QNX SOFTWARE SYSTEMS

NOTES:

You will learn:

- what a resource manager is
- how to use the QNX Neutrino resource manager framework:
  - initialization
  - handling read and write

NOTES:

Topics:

→ **Overview**

A Simple Resource Manager

– Initialization

– Handling *read()* and *write()*

Conclusion

NOTES:

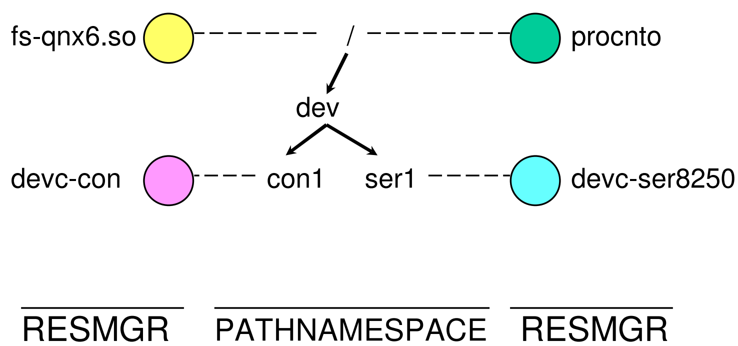
What is a resource manager?

- a program that looks like it is extending the operating system by:
  - creating and managing a name in the pathname space
  - providing a POSIX interface for clients (e.g. *open()*, *read()*, *write()*, ...)
- can be associated with hardware (such as a serial port, or disk drive)
- or can be a purely software entity (such as queuing or logging)

Let's take a look at the pathname space

NOTES:

Name mapping:



NOTES:

The prefix tree:

- is the root of the pathname space
  - every name in the pathname space is a descendant of some entry in the prefix tree
- is maintained by the Process Manager
  - stored as a table
- Resource Managers add and delete entries
- associates a **`nd, pid, chid, handle`** with a name
- is searched for the longest slash-delimited whole-word matching prefix

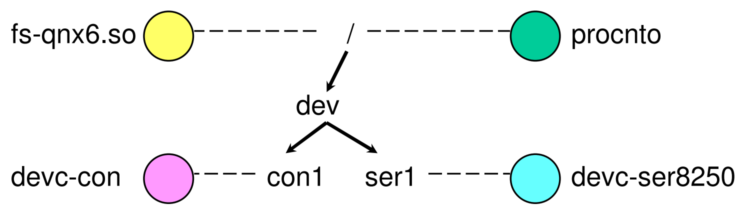
NOTES:

The **`handle`** is used by the resource manager library when a resource manager has more than one name registered. The **`handle`** is used to distinguish between the names.

For example, to resolve the pathname:

/dev/ser1

```
fd = open("/dev/ser1",...);
```



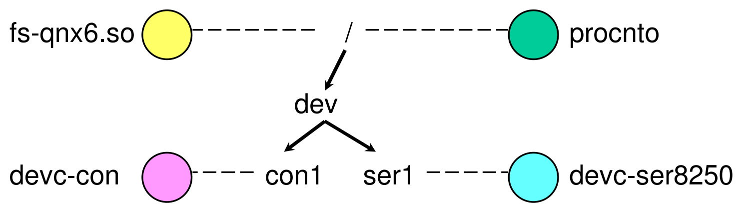
The longest match is /dev/ser1, which points to **`devc-ser8250`**

NOTES:

Or,

/home/bill/spud.dat

```
fd = open("/home/bill/spud.dat",...);
```



The longest match is /, which points to **`procnto`** and **`fs-qnx6.so`**. **`procnto`** would fail the open, and **`fs-qnx6.so`** would then handle requests.

NOTES:

A Client requests a service:

```
fd = open ("/dev/ser1", O_RDWR);
```

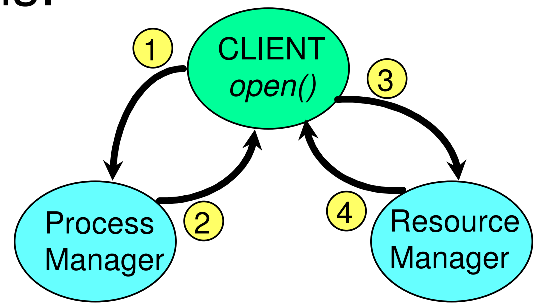
or

```
fp = fopen ("/home/bill/abc", "w");
```

which results in the client's library code (ultimately "**open**") sending a message to the process manager...

NOTES:

Interactions:



1. *open()* sends a “query” message
2. Process Manager replies with who is responsible (**`nd`**, **`pid`**, **`chid`**, **`handle`**)
3. *open()* establishes a connection to the specified resource manager (**`nd`**, **`pid`**, **`chid`**), and sends an open message (containing the **`handle`**)
4. Resource manager responds with status (pass/fail)

All further communication goes directly to the resource manager.

NOTES:

The application doesn’t have to worry about these details -- it’s all handled by the *open()* function in the C shared library.

How does *open()* find the process manager? Simple, it’s a well known nd, pid, chid.

Resource managers are built on message passing:

Client

Thread 1

① `write(){`

② `MsgSend()`

`}`

Resource Manager

Thread 2

`MsgReceive()` ③

.

.

.

`MsgReply()` ④

1. client calls the POSIX *write()*
2. this builds and sends a message
3. the resource manager receives and processes the **`write`** message appropriately
4. the resource manager replies with a result and this is returned to the client from *write()*

NOTES:

Most of the standard operating service programming interfaces are based on message passing in this way, including:

- *read()*
- *write()*
- *readdir()*
- *stat()*
- *close()*
- *devctl()*

A resource manager performs the following steps:

- creates a channel
- takes over a portion of the pathname space
- waits for messages & events
- processes messages, returns results

NOTES:

Resource Manager Messages

There are three major types of messages:

Connect messages:

- pathname-based (eg: `open ("spud.dat", …)`)
- may create an association between the client process and the resource manager, which is used later for I/O messages

I/O messages:

- file-descriptor- (`fd-`) based (eg: `read (fd, …)`)
- rely on association created previously by connect messages

Other:

- pulses, private messages, etc

NOTES:

QNX SOFTWARE SYSTEMS

Connect Messages:

message	**client call**
`_IO_OPEN`	*open()*
`_IO_UNLINK`	*unlink()*
`_IO_RENAME`	*rename()*

Defined in `<sys/iomsg.h>`

NOTES:

I/O Messages (frequently used):

message	**client call**
`_IO_READ`	*read()*
`_IO_WRITE`	*write()*
`_IO_CLOSE`	*close()*
`_IO_DEVCTL`	*devctl(), ioctl()*

Defined in `<sys/iomsg.h>`

continued...

NOTES:

I/O Messages (continued):

`_IO_NOTIFY,`	`_IO_STAT,`
`_IO_UNBLOCK,`	`_IO_PATHCONF,`
`_IO_LSEEK,`	`_IO_CHMOD,`
`_IO_CHOWN,`	`_IO_UTIME,`
`_IO_LINK,`	`_IO_FDINFO,`
`_IO_LOCK,`	`_IO_TRUNCATE,`
`_IO_SHUTDOWN,`	`_IO_DUP`

NOTES:

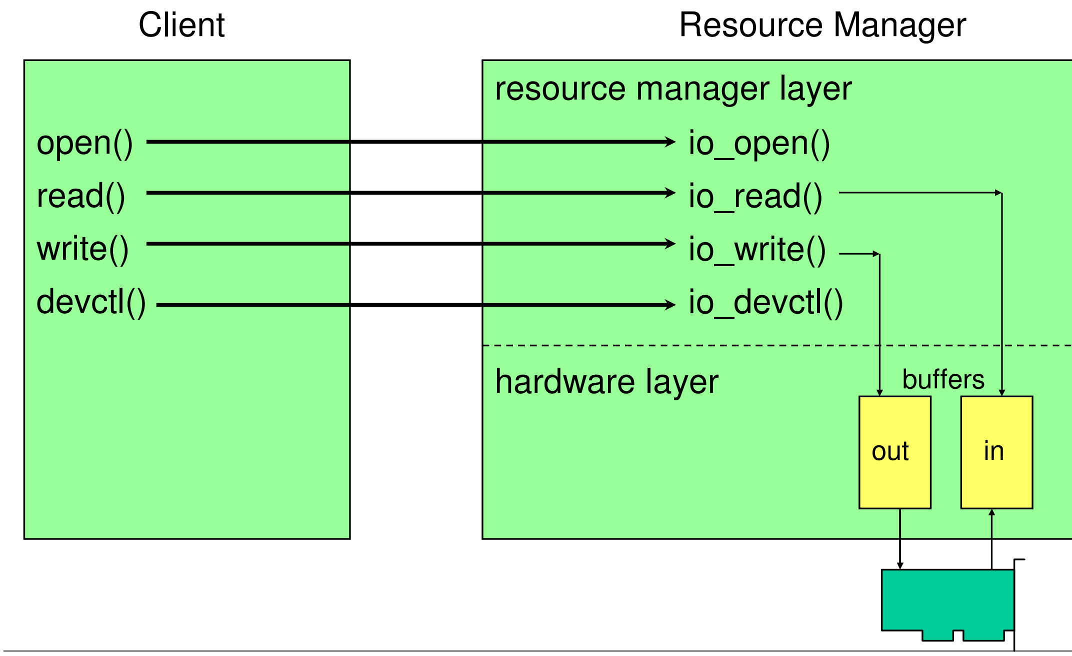
Writing resource managers is simplified greatly with a resource-manager shared library that:

- simplifies main receive loop (table-driven approach)
- has default actions for any message types that do not have handlers specified in tables

NOTES:

The resource manager shared library is part of libc.so.

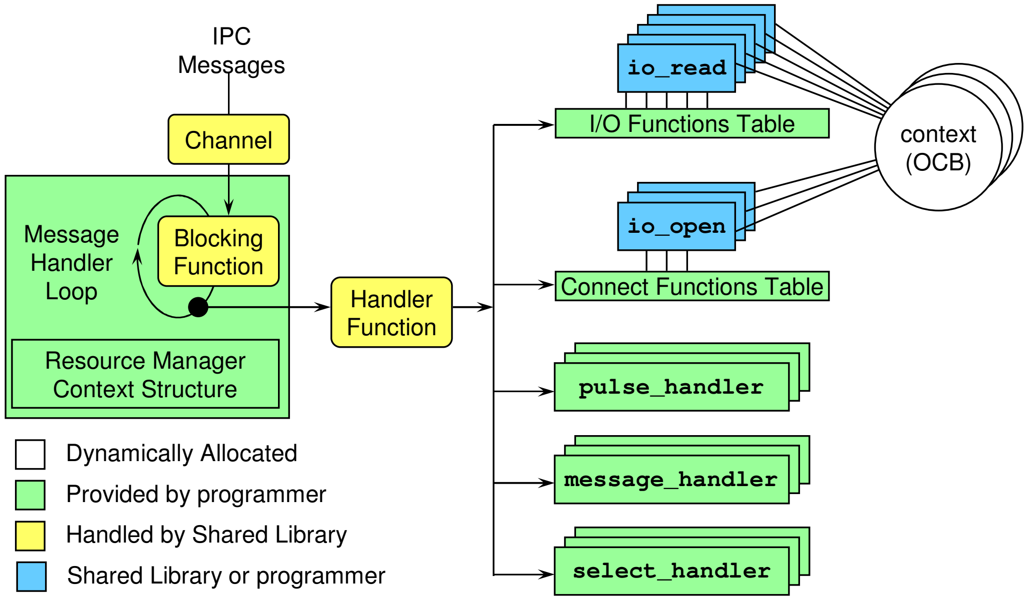
Client calls go to handlers:



NOTES:

Resource Manager Structure

The resource manager layer:



NOTES:

Topics:

Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Conclusion

NOTES:

To talk about setting up a resource manager, we'll use an **`example`** resource manager:

Client side:

- Here's how it behaves from a client's point of view:
  - **`read`** always returns 0 bytes
  - **`write`** of any size always works
  - other things behave as expected

NOTES:

The default resource manager behavior is equivalent to that of the /dev/null system device. We'll first handle that, then extend it to actually move some data on read and write.

The example resource manager:

- create & initialize structures:
  - a dispatch structure
  - list of *connect* message handlers
  - list of *I/O* message handlers
  - device attributes
  - resource-manager attributes
  - dispatch context
- attach a pathname, passing much of the above
- from the main loop:
  - block, waiting for messages
  - call a handler function; the handler function handles requests and performs callouts to your specified routines.

NOTES:

Setting things up - dispatch_create()

First, create a dispatch structure:

```
dispatch_t *dpp;

dpp = dispatch_create ();
```

dpp

– this is the glue the resource manager framework uses to hold everything together

– the contents are hidden (it is an opaque type)

NOTES:

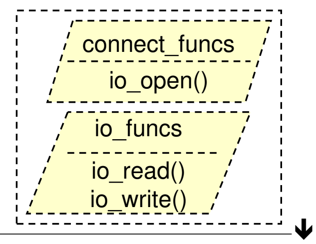
Setting things up - Connect and I/O functions

Next, we set up two tables of functions:

- connect functions
  - these are called as a result of POSIX calls that take a filename

    e.g.: *open (filename, ...)*, *unlink (filename)*, ...
- I/O functions
  - these are called as a result of POSIX calls that take a file descriptor

    e.g.: *read (fd, ...)*, *write (fd, ...)*, ...



NOTES:

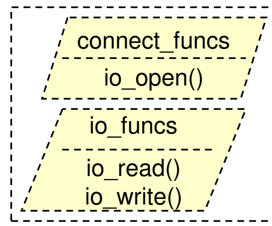
Our example won't actually define an *open()* handler, because the default handler works in most basic cases. It is generally needed for file system resource managers.

Example of declaring and initializing the connect- and the I/O-functions structures:

```
resmgr_connect_funcs_t connect_funcs;
resmgr_io_funcs_t      io_funcs;

iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
          _RESMGR_IO_NFUNCS, &io_funcs);

connect_funcs.open = io_open;
io_funcs.read    = io_read;
io_funcs.write   = io_write;
```



iofunc_func_init() places default values into the passed connect- and I/O-functions structures, based on the number of values that you have specified via the first and third integer arguments. It is recommended that you use the **_RESMGR_CONNECT_NFUNCS** and **_RESMGR_IO_NFUNCS** constants for those two arguments.

NOTES:

Note that by specifying the number of functions to be the size of the entire structure as it existed at compile time (**_RESMGR_CONNECT_NFUNCS** and **_RESMGR_IO_NFUNCS**), we are in effect building in a “version number”. In the future, if the shared object containing the resource manager framework should change, the shared object could examine the number passed and decide what sort of default behavior was appropriate for any new entries added later.

QNX SOFTWARE SYSTEMS

Next, fill the device-attributes structure, for passing to resmgr_attach():

device structure
iofunc_attr_t

```
iofunc_attr_t ioattr;

iofunc_attr_init (&ioattr, S_IFCHR | 0666, NULL,
                  NULL);
```

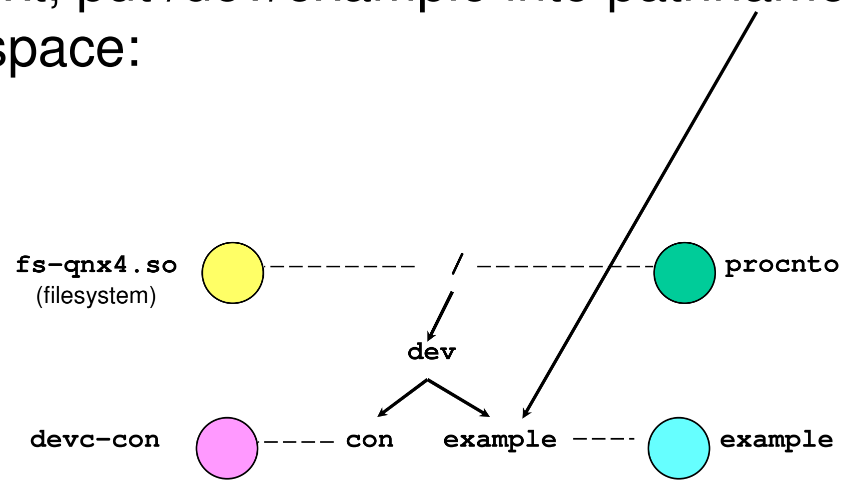
– this is your device-specific data structure
– used by the *iofunc_*()* helper functions
– it is possible to extend this structure so that it can contain your own data too

NOTES:

`S_IFCHR` is defined in **`<sys/stat.h>`** and says this device is a character special device.

Putting /dev/example into the pathname space

Next, put /dev/example into pathname space:




```
resmgr_attach(…,"/dev/example",…);
```


NOTES:

You must be root (userid 0) to do this.

The parameters are:

```
id = resmgr_attach (dpp, &rattr, path, file_type,
             flags, &connect_funcs, &io_funcs, handle);
```

dpp = pointer returned by *dispatch_create()*

rattr = **NULL** or structure of further paramaters

path = **"/dev/example"**

file_type = **_FTYPE_ANY** (the usual case)

flags = **0** or control flags...

connect_funcs and **io_funcs** point to the tables of functions we just created

handle = pointer to device attribues

id = id of this pathname, used for *resmgr_detach()* call

☞ You must be root (userid 0) for this function to work.

NOTES:

You likely would not use any of the other types for the **file_type** member. For example, **_FTYPE_MQUEUE** is used by the POSIX Message Queue process.

resmgr_attach() - flags

The resmgr_attach(..., <u>flags</u>, ...):

_RESMGR_FLAG_BEFORE and

_RESMGR_FLAG_AFTER

this resource manager will handle the pathname BEFORE or AFTER all others that have attached the same pathname

_RESMGR_FLAG_DIR

allow pathnames that extend past the registered pathname to be handled by this resource manager

used by filesystem resource managers (e.g.: **/cdrom/…**)

NOTES:

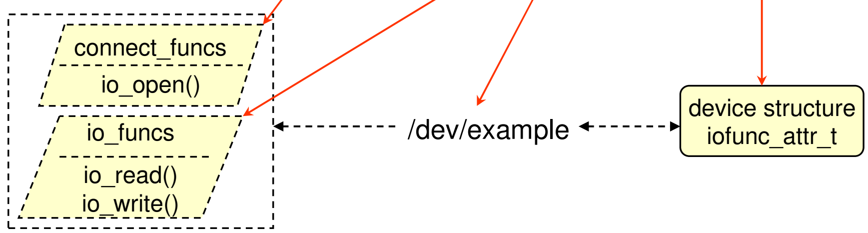
If multiple registrations with BEFORE or AFTER set, then all BEFORE will be first in FIFO order, then non-flagged in LIFO order, then AFTER in LIFO order. (First AFTER stays as last, first BEFORE stays at first.)

resmgr_attach()

When you call *resmgr_attach*(), you are:

- creating your device
- associating data and handlers with it

```
id = resmgr_attach (dpp, &rattr, "/dev/example", _FTYPE_ANY,
                    NULL, &connect_funcs, &io_funcs, &ioattr);
```



☞The library does not make copies of these structures

NOTES:

resmgr_attach()

resmgr_attach() puts the name in the pathname space:

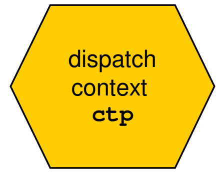
- your resource manager becomes visible to clients
  - clients will expect you to be ready to handle messages
- before doing so, you should have completed most of your initialization:
  - hardware detection and initialization
  - buffer allocation and configuration
- if something fails, don't attach name

NOTES:

Setting things up - dispatch_context_alloc()

Lastly, allocate a dispatch context structure:

```
dispatch_context_t *ctp;
ctp = dispatch_context_alloc (dpp);
```



- this is the operating parameters of the message receive loop
- it is passed to the blocking function and the handler function
- it contains things like the **`rcvid`**, pointer to the receive buffer, and message info structure
- it will be passed as the **`ctp`** parameter to your connect and I/O functions

NOTES:

Putting together what we have so far:

```
dpp = dispatch_create ();

iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_funcs);
connect_funcs.open = io_open;
io_funcs.read    = io_read;
io_funcs.write   = io_write;

iofunc_attr_init (&ioattr, S_IFCHR | 0666, NULL,
                  NULL);

id = resmgr_attach (dpp, NULL, "/dev/example", _FTYPE_ANY,
            0, &connect_funcs, &io_funcs, &ioattr);

ctp = dispatch_context_alloc (dpp);
```


NOTES:

So now that everything's set, the loop:

```
while (1) {
    ctp = dispatch_block (ctp);
    dispatch_handler (ctp);
}
```

– *dispatch_block()* blocks waiting for messages,

– *dispatch_handler()* handles them, including calling any callout functions which you've provided (connect function, I/O functions)

NOTES:

We could have done this instead:

```
resmgr_context_t *ctp;
...
ctp = resmgr_context_alloc (dpp);
while (1) {
    ctp = resmgr_block (ctp);
    resmgr_handler (ctp);
}
```

–but *resmgr_block()* and *resmgr_handler()* do not support pulse handlers, message handlers and select handlers.

–Basically, the *dispatch_*()* functions act as a central point for handling multiple types of things. The *resmgr_*()* functions are a little quicker, but are for handling only **_IO_*** messages. *dispatch_handler()* ultimately calls *resmgr_handler()*.

EXERCISE

Exercise:

- in your **resmgr** project, look at **example.c**
- it is missing much of the initialization, finish the initialization
- run it as:

```
example -v
```

- you will need a command line on your target to test it, try:

```
echo Hello >/dev/example
ls /dev
ls -l /dev/example
cat /dev/example
```


NOTES:

Topics:

Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Conclusion

NOTES:

The example Resource Manager

Let’s see what happens when a client uses the new **/dev/example** device (by, for example, doing “**cat /dev/example**”):

Internally, “**cat**” basically does:

```
fd = open("/dev/example",O_RDONLY);
while (read (fd, buf, BUFSIZ) > 0)
    /* write buf to stdout */
close (fd);
```


NOTES:

Which results in:

- Communications with the Process Manager:
  - an inquiry message to the process manager:
    - "who is responsible for **`/dev/example`**?"
    - returns a reply, "(nd, pid, chid)" (our resource manager, **`example`**), "is responsible"
- Communications with **`example`**:
  - an open message
    - "open this device for read"
    - returns a reply, "yes, open succeeded, proceed"
    - the *open()* library call returns a file descriptor, **`fd`**
  - a read message
    - "get me some data"
    - returns a reply, "here are 0 bytes" (I.e. EOF)
  - a close message

NOTES:

On the client side, all of the messaging interaction that is taking place above is done inside the C library by the library calls *open(), read(),* and *close()*.

The example Resource Manager's io

Let's look at example's I/O functions:

```
int   io_read (resmgr_context_t *ctp,
                io_read_t *msg,
                RESMGR_OCB_T *ocb);
int   io_write (resmgr_context_t *ctp,
                 io_write_t *msg,
                 RESMGR_OCB_T *ocb);
```

They both share the **`ctp`** and **`ocb`**...

NOTES:

QNX SOFTWARE SYSTEMS

ctp

- pointer to a resource-manager context structure
- information about the received message
- contains at least:

```
typedef struct _resmgr_context {
  int                    rcvid;
  struct _msg_info       info;
  resmgr_iomsgs_t        *msg;
  unsigned               msg_max_size;
  int                    status;
  int                    offset;
  IOV                    iov [1];
} resmgr_context_t;
```

dispatch
context
ctp

NOTES:

The **rcvid** is the return value from the *MsgReceive()*, and will be used for replying to the client.

The **info** structure is the message information structure from the *MsgReceive()* call.

The ***msg** field is the receive buffer, and is declared as a union of ALL possible message types.

msg_max_size is the size of the receive buffer pointed to by **msg**.

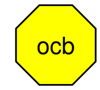
The **status** field is used for the reply to the client, holding the reply status, often used for the number of bytes handled.

offset is the size of extra headers skipped in the receive buffer before this handler function was called.

size is the size of the current message being processed, which may be smaller than the amount received if this is a component of a multi-part message.

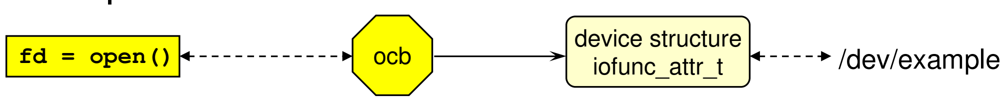
The **iov** array is an array size (default 1) as specified by the **rattr** passed to *resmgr_attach()* that can be used for building multi-part replies.

I/O Functions Arguments -- ocb



`ocb`

- Open Control Block
- one **`ocb`** per *open()*
- maintains context between the *open()* call and subsequent I/O calls, i.e. *iofunc_open_default()* allocates and initializes it, I/O functions use it.
- will be either an **`iofunc_ocb_t`**, or
- an **`iofunc_ocb_t`** encapsulated within your own structure with additional state information, etc.
- points to the attribute structure for the device opened



NOTES:

Open Control Block is a name that dates back to some of the internal data structures used in UNIX for filesystems -- basically, you can think of it as a context block.

The message you'll receive is:

```
typedef union
{
  struct _io_read  i;
} io_read_t;

struct _io_read
{ // contains at least the following
  unsigned short   type;    // message type = _IO_READ
  long             nbytes; // number of bytes to be read
  uint32_t         xtype;   // extended type
};
```


NOTES:

The one-element union for the **io_read_t** is declared that way to parallel other message structures that will have both **i** (incoming) and **o** (outgoing) message structure in the union.

The message will be generated by the *read()* call which looks like:

```
bytes_read = read( fd, buf, nbytes )
{
  struct _io_read hdr;
  hdr.type = _IO_READ;
  hdr.nbytes = nbytes;
  ...
  return( MsgSend( fd, &hdr, sizeof(hdr), buf, nbytes ) );
}
```

For the reply:

– if successful:

- the reply message would be your data (i.e. there is no header to worry about)
- the return value from the *read()*'s *MsgSend()* would be the number of bytes successfully read. To set this, do:

```
_IO_SET_READ_NBYTES (ctp, nbytes_read);
SETIOV (ctp->iov, data, nbytes_read);
return (_RESMGR_NPARTS(1));
```

When you return to the resource manager library, it will pass **nbytes_read** as the status parameter to *MsgReplyv()*. The *read()* will return this value.

– if failed, do:

```
return (errno_value);
```


NOTES:

QNX SOFTWARE SYSTEMS

The xtype (extended-type) member:

- will most often be:

  `_IO_XTYPE_NONE`

- most resource managers check for **`_IO_XTYPE_NONE`**. If xtype is not this, then they return **`ENOSYS`**

NOTES:

For some of the other possible **`xtype`** values see the "Writing a Resource Manager" section in the Programmer's Guide.

example's **read** function:

```
int
io_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
  int status;

  if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK)
    return (status);

  if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
    return (ENOSYS);

  _IO_SET_READ_NBYTES (ctp, 0); /* 0 bytes successfully read */

  if (msg->i.nbytes > 0) /* mark access time for update */
    ocb->attr->flags |= IOFUNC_ATTR_ATIME;

  return (_RESMGR_NPARTS (0));
}
```


NOTES:

The *iofunc_read_verify()* will do access-permission checks for you as well as other things.

According to POSIX, when a read returns zero bytes, this means end-of-file.

POSIX says that you must mark the access time for update when a read is done where the number of bytes requested is greater than 0 and the read is successful. The actual time doesn't have to be updated until a stat is done or the file is closed. So, here, set a bit in the attribute structure that will be looked at in the default stat and close handlers. This avoids an extra kernel call to get the time on each read.

If you want the access time to be the time of the read, set the flag as above and then call *iofunc_time_update()*.

Exercise:

- in your **resmgr** project, go back to **example.c** (we've already seen most of it)
- modify the **io_read** handler to return some data
  - make up some data
- run it as:

```
example -v
```

- test it from a command line on your target:

```
cat /dev/example
```


NOTES:

It is normal and expected behaviour that you will get an unending stream of data from "cat /dev/example". The ocb->offset entry is commonly used to track current position within a file, and to determine when to return end of file.

The message you'll receive is:

```
typedef union
{
  struct _io_write i;
} io_write_t;
/* the data to be written usually follows the io_write_t */

struct _io_write
{ // contains at least the following
  unsigned short   type;    // message type = _IO_WRITE
  long             nbytes;  // number of bytes to write
  uint32_t         xtype;   // extended type
};
```


NOTES:

The message will be generated by the *write()* call, which looks like:

```
bytes_written = write( fd, buf, nbytes )
{
  struct _io_write hdr;
  iov_t iov[2];
  hdr.type = _IO_WRITE;
  hdr.nbytes = nbytes;
  ...
  SETIOV(&iov[0], &hdr, sizeof(hdr));
  SETIOV(&iov[1], buf, nbytes );
  return( MsgSendv( fd, iov, 2, NULL, 0 ) );
}
```

The case where the data does not follow the **io_write_t** in the message buffer is when the client calls one of the *pread*()* or *pwrite*()* functions. See the resource manager documentation for how to handle this case. Look for information on handling the case where the **xtype** member in the mesage header is **_IO_XTYPE_OFFSET**.

For the reply:

– if successful:

- there is no data to reply with
- the return value from the *write()*'s *MsgSendv()* would be the number of bytes successfully written. To set this do:

```
_IO_SET_WRITE_NBYTES (ctp, nbytes_written);
return (_RESMGR_NPARTS(0));
```

When you return to the resource manager library, it will pass **nbytes_written** as the status parameter to *MsgReplyv()*. The *write()* will return this value.

– if failed, do:

```
return (errno_value);
```


NOTES:

WRITE

example's **`write`** function:

```
int io_write (resmgr_context_t *ctp, io_write_t *msg,
              RESMGR_OCB_T *ocb)
{
  int status;
  if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK)
    return (status);
  if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
    return (ENOSYS);
  // msg -> i.nbytes is the number of byte to be written,
  // we are telling it that we wrote everything (msg -> i.nbytes)
  _IO_SET_WRITE_NBYTES (ctp, msg -> i.nbytes);
  if (msg->i.nbytes > 0) /* mark times for update */
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;
  return (_RESMGR_NPARTS (0));
}
```


NOTES:

The *iofunc_write_verify()* will do access permission checks for you as well as other things.

As with read, POSIX says that you must mark the access time for update when a write is done where the number of bytes requested is greater than 0 and the write is successful. The actual time doesn't have to be updated until a stat is done or the file is closed. So, here set a bit in the attribute structure that will be looked at in the default stat and close handlers. This avoids an extra kernel call to get the time on each write.

If you want the times to be the time of the write, set the flags as above and then call *iofunc_time_update()*.

WRITE - Getting the data

example doesn't do anything with the data to be written, but what if you want to?

– The data usually follows the **`io_write_t`** in the message buffer.
– But it may not all have been received, what happens in the following case?

```
MsgSend(coid, smsg, sbytes, ...);

  smsg =

          sbytes = 3000
```

```
MsgReceive(chid, rmsg, rbytes, ...);

    rmsg =

      rbytes = 1000
```

As you know, the kernel copies the lesser of the two sizes, so in this case only 1000 bytes will have been received. How do you handle this?

NOTES:

In your io_write callback:

`int io_write (resmgr_context_t *ctp, io_write_t *msg, …)`

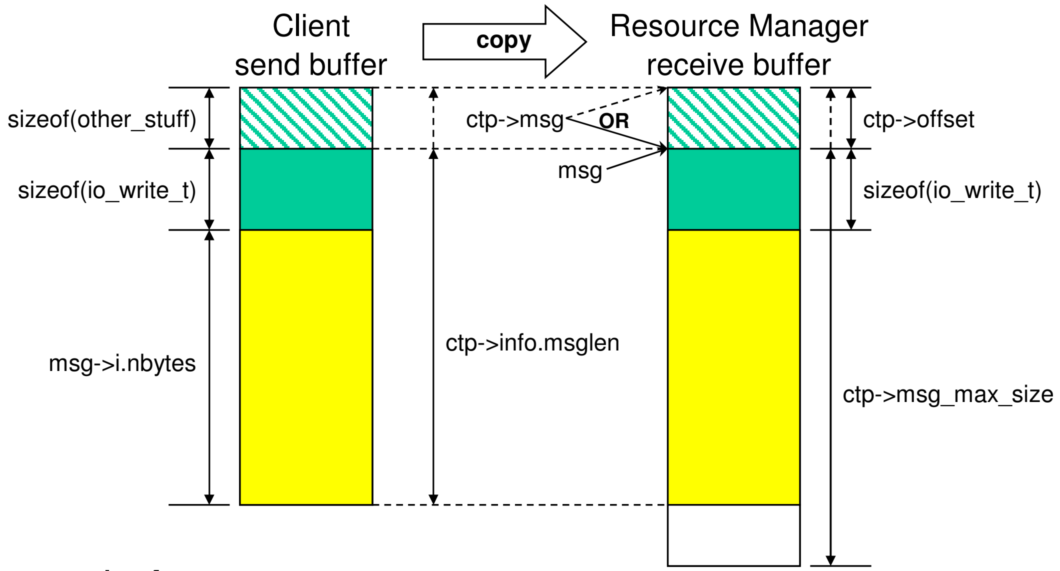
You have several pieces of information:

- **`msg->i.nbytes`**
  - is the number of bytes passed to the client's *write()* call: **`write(fd, buf, nbytes)`**
- **`ctp->info.msglen`**
  - is the number of bytes that have actually been copied into the receive buffer
  - includes the **`io_write_t`** header and any headers before it
- **`ctp->offset`**
  - is the size of any headers before the **`io_write_t`**
  - **`msg = (char *)(ctp->msg) + ctp->offset`**
- **`ctp->msg`**
  - is a pointer to the actual receive buffer
- **`ctp->msg_max_size`**
  - is the size of the receive buffer, **`ctp->msg`**

NOTES:

WRITE - getting the data

We got all the write data:

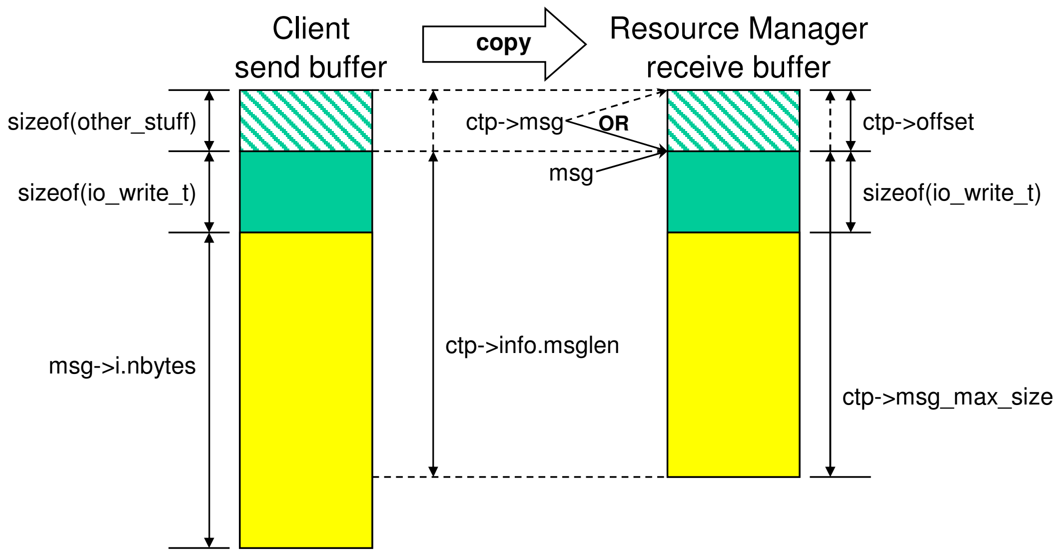



```
msg->i.nbytes ==
     ctp->info.msglen - (ctp->offset + sizeof(io_write_t))
```


NOTES:

WRITE - getting the data

We didn't get all the write data:




```
msg->i.nbytes >
     ctp->info.msglen - (ctp->offset + sizeof(io_write_t))
```


NOTES:

How do you use this?

- if **`msg->i.nbytes`** is equal to
  `ctp->info.msglen -`
  `(ctp->offset + sizeof(io_write_t))`
  - all the write data has already been received, and we can use it directly from the receive buffer
- otherwise, we don't have the whole message
  - need to find somewhere to put the data
  - need to go get the rest of the data using *resmgr_msgread()*

NOTES:

resmgr_msgread() is a cover function for *MsgRead()* that automatically adds in the **`ctp->offset`** for you.

A simple method is to reread all of it from the sender's buffer:

```
int io_write (resmgr_context_t *ctp, io_write_t *msg,
RESMGR_OCB_T *ocb)
{
  if( need_more_data ) {
    char *buf;
    buf = malloc( msg->i.nbytes );
    ...
    resmgr_msgread (ctp, buf, msg -> i.nbytes,
                   sizeof (msg -> i));
    // do something with buf
    free( buf );
    ...
  }
}
```

offset to skip,
in this case the header

NOTES:

WRITE - Getting the Data

But there are other choices, including:

- find available cache buffers and use *resmgr_msgreadv()* to fill them
- use a small buffer and multiple *resmgr_msgread()* calls to work through the client's message, a piece at a time
- copy the already received data from the receive buffer, and then use *resmgr_msgread()* for the rest
- ensure in advance that the receive buffer will be large enough for your largest write

NOTES:

You can modify the size of the receive buffer by by filling in the **`msg_max_size`** member of the **`resmgr_attr_t`** structure that you'd passed to *resmgr_attach()*. You'd set it to the largest message that you'll be receiving, including the headers.

Exercise:

– we're going to further modify **example.c** in your **resmgr** project

– modify the **io_write** handler to:

- print out the number of bytes written
- print out all the data written
- handle the small messages without reading more data

– run it as:

```
example -v
```

– test it from a command line on your target:

```
echo Hello >/dev/example
cp /etc/services /dev/example
cp /etc/termcap /dev/example
```


NOTES:

On some targets, **/etc/services** and **/etc/termcap** may not exist, so copy another large text file, or transfer your source file, **example.c**, to the target and then copy it to **/dev/example**.

Topics:

Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

→ **Conclusion**

NOTES:

Conclusion

You learned:

- that a resource manager is a device driver framework
- how to initialize and register a resource manager
- how to handle *read()* and *write()* client requests

NOTES:

References

Programmer's Guide (QSS)

- contains a chapter titled “Writing a Resource Manager”

Getting Started with QNX Neutrino

- by Rob Krten, but included with QNX documentation
- has good information on writing Resource Managers

NOTES: