Introduction to Resource Managers

You will learn:

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

Topics:

→ **Overview**

A Simple Resource Manager

– Initialization

– Handling *read()* and *write()*

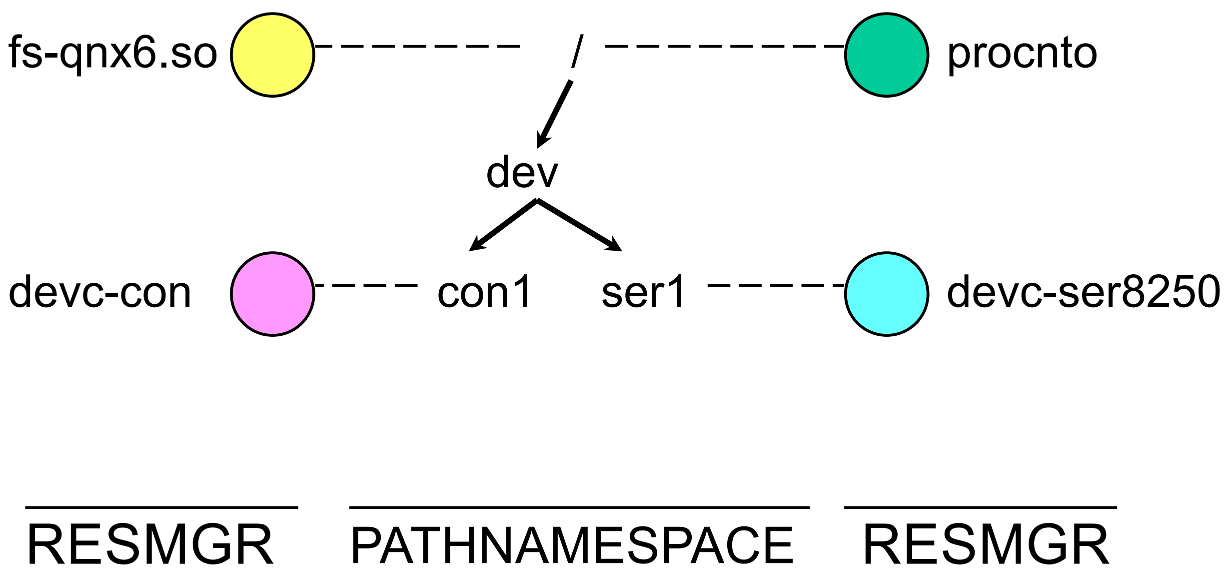
Conclusion

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

Name mapping:



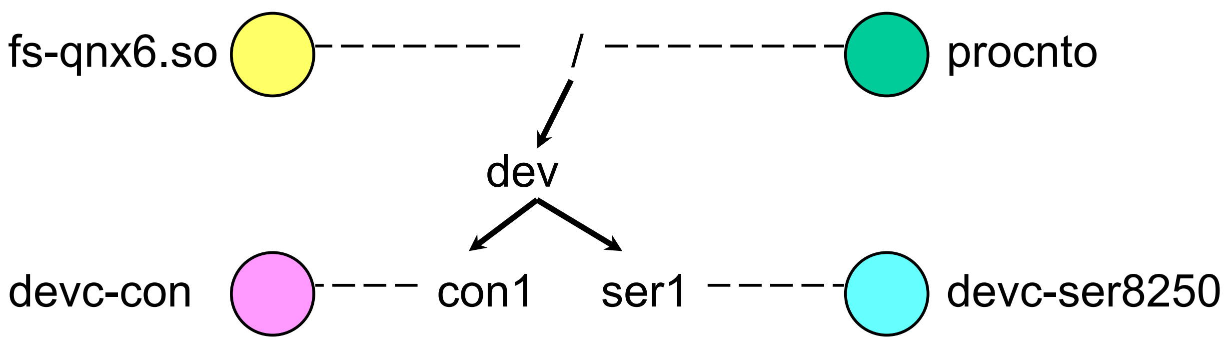
The prefix tree:

- 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 **`pid`**, **`chid`**, and **`handle`** with a name
- is searched for the longest slash-delimited whole-word matching prefix

For example, to resolve the pathname:

/dev/ser1

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

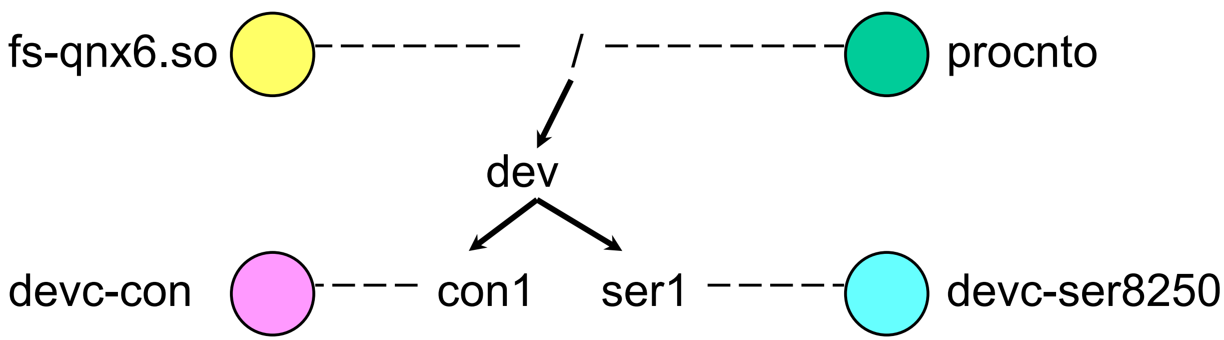


The longest match is /dev/ser1, which points to **`devc-ser8250`**

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.

Overview

A Client requests a service:

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

or

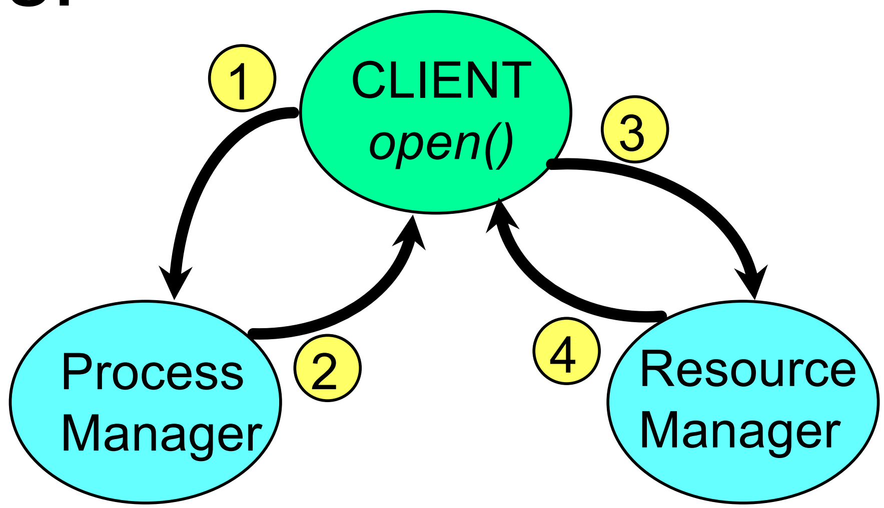
```
fp = fopen ("/home/bill/abc", "w");
```

or

```
mq = mq_open("/myqueue", O_RDONLY);
```

which results in the client's library code sending a message to the process manager...

Interactions:



1. *open()* sends a "query" message
2. Process Manager replies with who is responsible (**`pid`**, **`chid`**, **`handle`**)
3. *open()* establishes a connection to the specified resource manager (**`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.

Resource managers are built on message passing:

Client	Resource Manager
Thread 1	Thread 2
1 `write(){`	`MsgReceive()` 3
2 `MsgSend()`	.
`}`	.
	.
	`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()*

Overview

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

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

Connect Messages:

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

Defined in `<sys/iomsg.h>`

I/O Messages (frequently used):

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

Defined in `<sys/iomsg.h>`

continued...

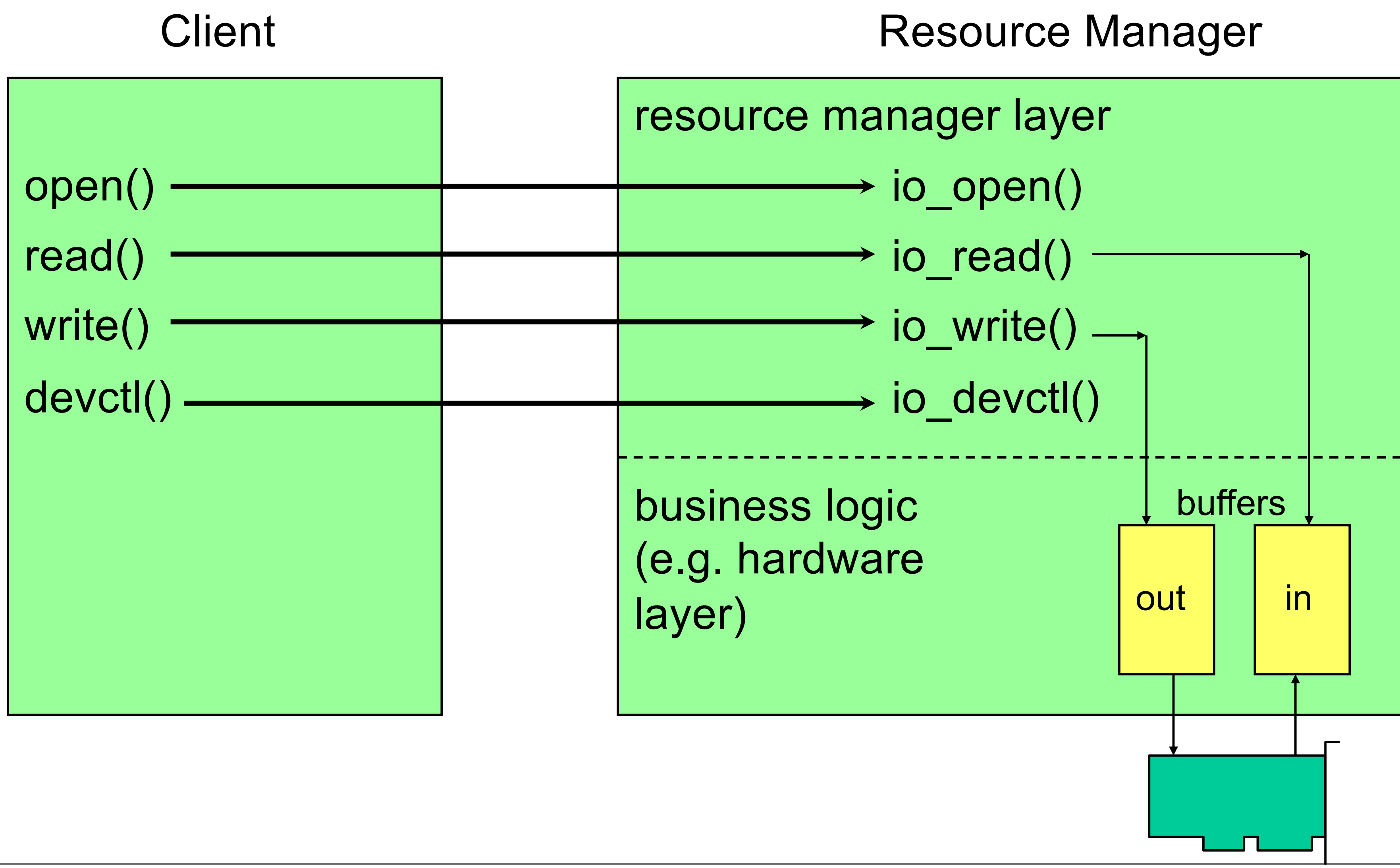
I/O Messages (continued):

```
_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
```

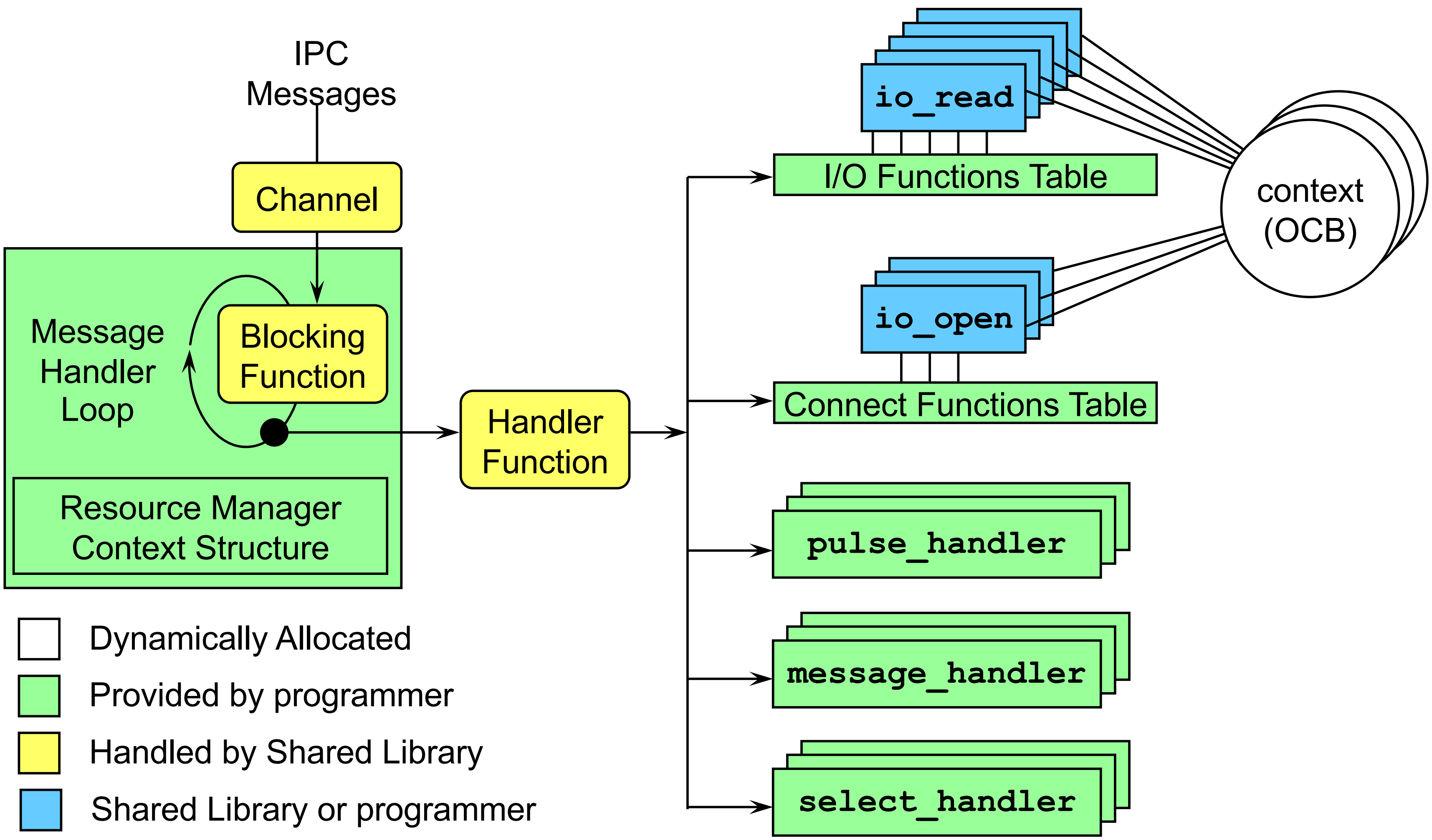
Writing resource managers is simplified greatly with a resource-manager shared library that:

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

Client calls go to handlers:



The resource manager layer:



Message-handling functions:

- may access client data
  - data already in the receive buffer
  - get more data if necessary
- must either:
  - reply to the client with:
    - an error
    - a success status without data
    - a success status and data
- or:
  - delay responding to the client until later

Topics:

Overview

A Simple Resource Manager

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

Conclusion

The example Resource Manager

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

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.

First, create a dispatch structure:

```
dispatch_t *dpp;

dpp = dispatch_create_channel(chid, flags);
```

dpp

- this is the glue the resource manager framework uses to hold everything together
- the contents are hidden (it is an opaque type)

- This is usually called with the paramaters:

```
dispatch_create_channel(-1, DISPATCH_FLAG_NOLOCK);
```

  - create a channel
  - disable some, usually unnecessary, mutex locks during message handling

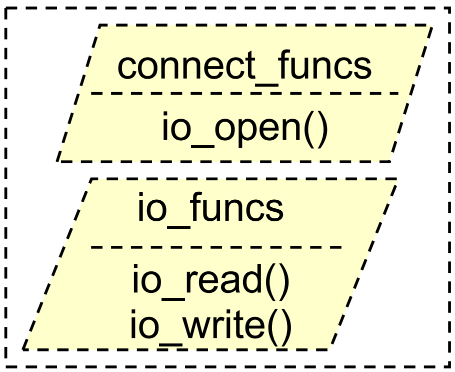
Setting things up - Connect and I/O functions

Next, we set up two tables of functions:

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

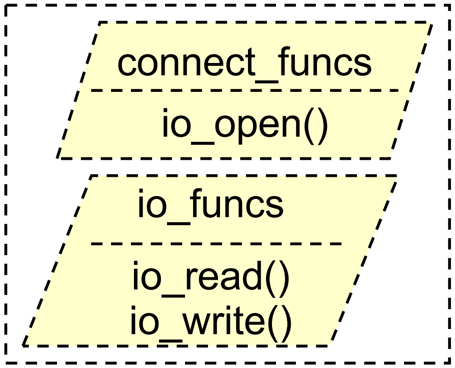
    e.g.: *open (filename, ...)*, *unlink (filename)*, ...
- I/O functions
  - these are called as a result of POSIX calls that take a file descriptor

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



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.

Setting things up - iofunc_attr_t

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

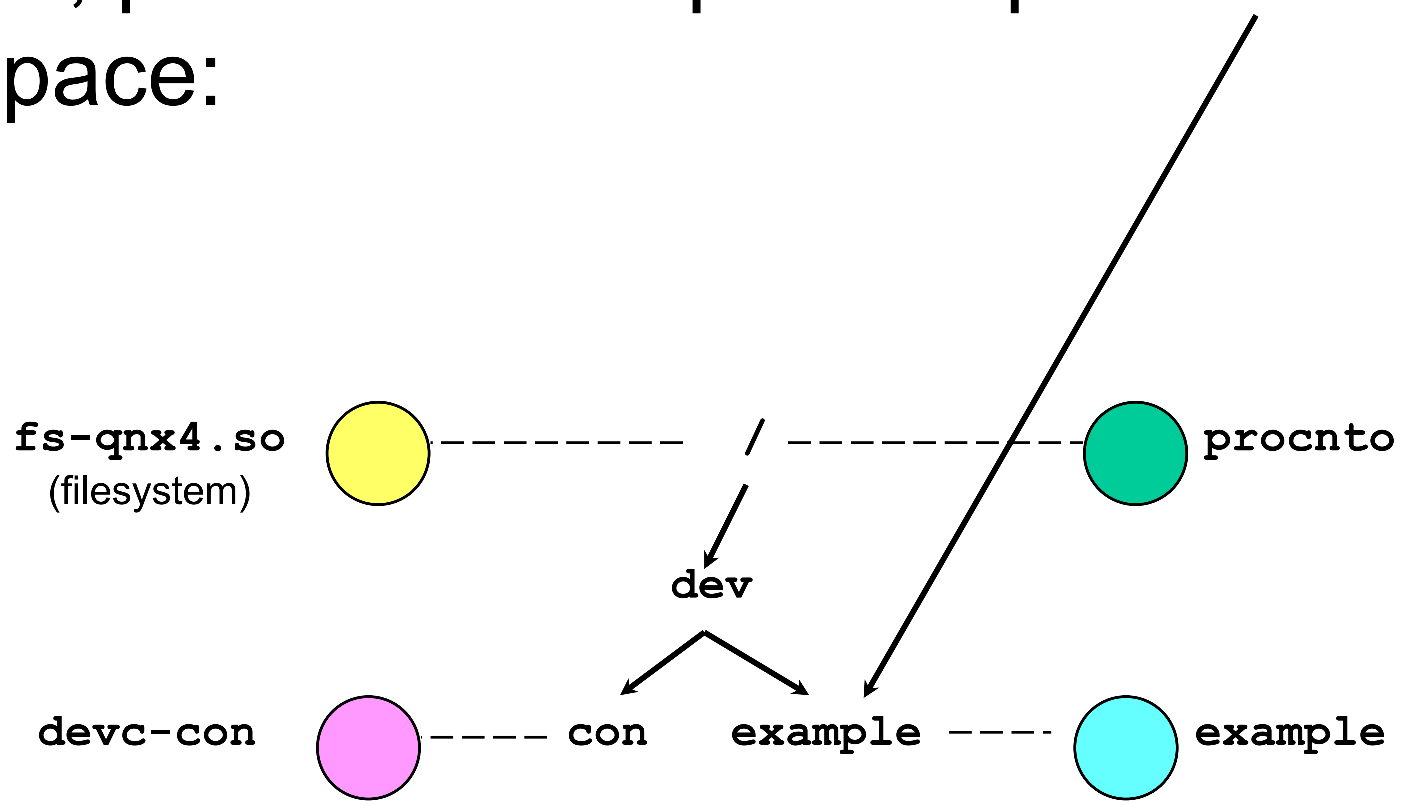
```
iofunc_attr_t ioattr;

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

device structure
iofunc_attr_t

- 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

Putting /dev/example into the pathname space

Next, put /dev/example into pathname space:



`resmgr_attach(…,"/dev/example",…);`

The parameters are:

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

dpp = pointer returned by *dispatch_create_channel()*

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

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 attributes

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

☞ Requires **PROCMGR_AID_PATHSPACE** to succeed.

The resmgr_attach(..., flags, ...):

`_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`

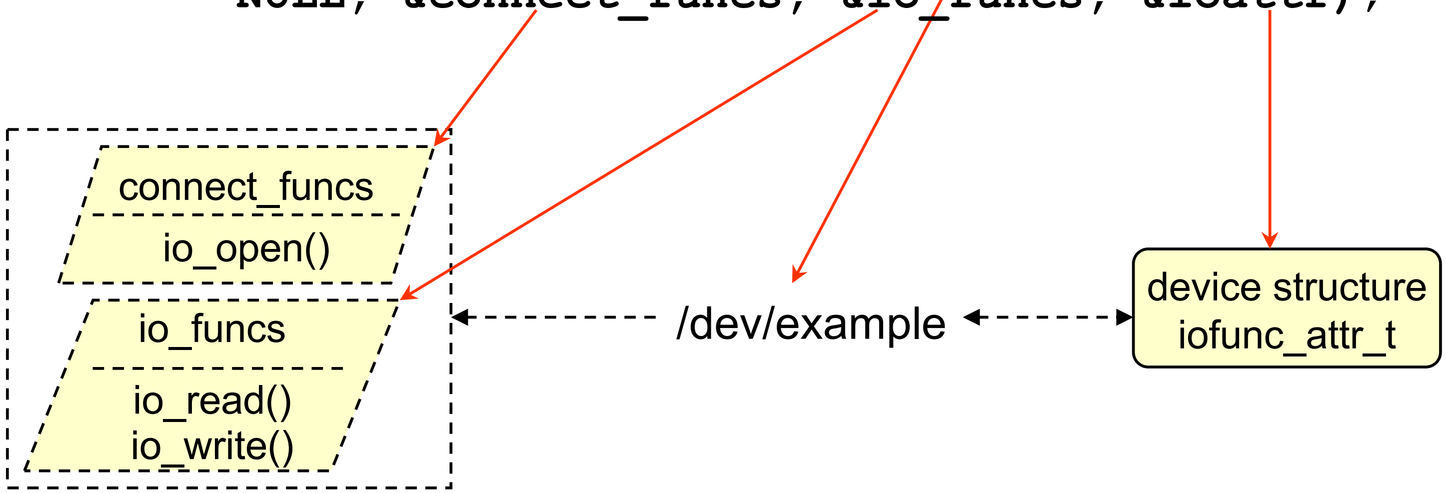
allow pathnames that extend past the registered pathname to be handled by this resource manager

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

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

- creating your device
- associating data and handlers with it

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



☞The library does not make copies of these structures

resmgr_attach()

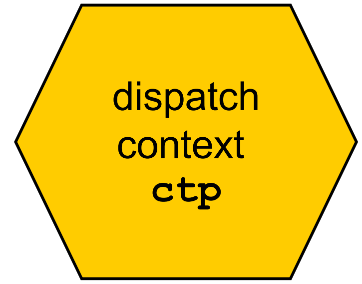
resmgr_attach() puts the name in the pathname space:

- 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

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

Putting together what we have so far:

```
dpp = dispatch_create_channel (-1, DISPATCH_FLAG_NOLOCK);

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);
```

Setting things up - The dispatch loop

So now that everything’s set, the loop:

```
while (1) {
   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)

Exercise:

- in your **`rm_writing`** 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
```

- you should see it printing that it is in the various io functions

Topics:

Overview

A Simple Resource Manager

– Initialization

– Handling *read()* and *write()*

Conclusion

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);
```

Which results in:

- Communications with the Process Manager:
  - an inquiry message to the process manager:
    - “who is responsible for **`/dev/example`**?”
    - returns a reply, “(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

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);
```

- **`msg`** is always a pointer to the current message being handled

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

I/O Functions Arguments -- ctp

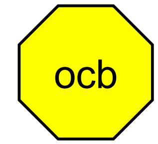
`ctp`

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

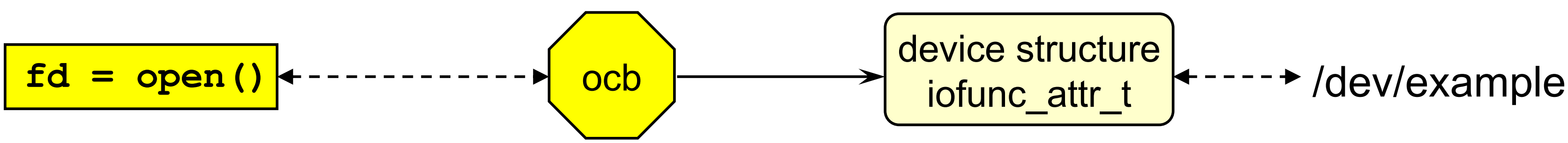
```
typedef struct _resmgr_context {
  int                  rcvid;
  struct _msg_info     info;
  resmgr_iomsgs_t      *msg;
  size_t               msg_max_size;
  size_t               offset;
  size_t               size;
} resmgr_context_t;
```

dispatch
context
`ctp`

`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



READ

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.xtype = _IO_XTYPE_NONE;
  hdr.nbytes = nbytes;
  ...
  return MsgSend( fd, &hdr, sizeof(hdr),
                  buf, nbytes );
}
```

READ

The message you'll receive is:

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

struct _io_read
{ // contains at least the following
  _Uint16t        type;   // message type = _IO_READ
  _Uint32t        xtype;  // extended type
  _Uint32t        nbytes; // number of bytes to be read
};
```

For the reply:

– if successful:

```
MsgReply(ctp->rcvid, bytes_read, buffer,
  bytes_read);
return _RESMGR_NOREPLY;
```

- the reply message (buffer, bytes_read) would be your data (i.e. there is no header to worry about)
- the *MsgReply()* status would be the return value from the client's *read(),* the number of bytes successfully read

– if failed, do:

```
return errno_value;
```

The xtype (extended-type) member:

- will most often be:

  `_IO_XTYPE_NONE`

- most *read()* and *write()* handlers check for **`_IO_XTYPE_NONE`**. If xtype is not this, then they return **`ENOSYS`**

example's *read()* handler:

```
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;

  MsgReply(ctp->rcvid,0,NULL,0); /* 0 bytes successfully read */

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

  return _RESMGR_NOREPLY;
}
```

Exercise:

- in your **`rm_intro`** project, go back to **`example.c`**
- modify the *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
```

WRITE

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 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
};
```

For the reply:

- if successful:

```
MsgReply(ctp->rcvid, bytes_written, NULL, 0);
return _RESMGR_NOREPLY;
```

  - there is no data to reply with
  - the *MsgReply()* status would be the return value from the client's *write(),* the number of bytes successfully written

- if failed, do:

```
return errno_value;
```

example's *write()* handler:

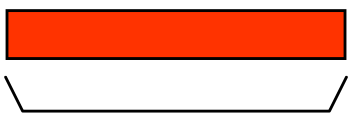
```
int io_write (resmgr_context_t *ctp, io_write_t *msg,
              RESMGR_OCB_T *ocb)
{
  int status;
  if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK)
    return status;

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

  // msg -> i.nbytes is the number of bytes to be written,
  // we are telling it that we wrote everything (msg -> i.nbytes)
  MsgReply(ctp->rcvid, msg->i.nbytes, NULL, 0);

  if (msg->i.nbytes > 0) /* mark times for update */
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;

  return _RESMGR_NOREPLY;
}
```

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, ...);
```

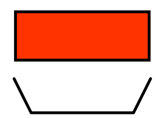
`smsg =`



`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?

WRITE – Getting the data

You need to:

- figure out where to put the data
  - pre-allocated ring buffer
  - hardware cache buffers
  - allocate a buffer
- put the data where it goes:
  - for a single destination buffer:
    - *resmgr_msgget()*
  - for multiple destination buffers:
    - *resmgr_msggetv()*

Get client data with:

```
bytes_copied =
   resmgr_msgget(ctp, buf, bytes, offset);
```

- copy **bytes** of data from client **ctp->rcvid**
- starting at **offset** bytes after the start of **msg**
  - remember, **msg** is a standard parameter to our I/O functions
- to **buf**

- copies locally if possible
- calls *MsgRead()* when needed

Exercise:

- we're going to further modify **`example.c`** in your **`rm_intro`** project
- modify the **`io_write`** function to:
  - print out the number of bytes written
  - print out all the data written
- 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
```

Topics:

Overview

A Simple Resource Manager

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

→ **Conclusion**

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