Using NullFM

Version 1.0

MICROWARE™

Intelligent Products For A Smarter World

Copyright and Publication Information

This manual reflects version 2.2 of DAVID.

Revision:	B
Publication date:	November 1999

Disclaimer

Reproduction Notice

Trademarks

Address

Microware Systems Corporation
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Table of Contents

Chapter 1: Using NullFM

This chapter describes the Microware null file manager (NullFM) and the basic concept of direct driver-to-driver interfaces. It includes the following sections:

- **NullFM Overview**
- **Using NullFM for Direct Drivers**
- **NullFM Programming Reference**

NullFM Overview

NullFM (Null File Manager) is an OS-9 file manager used by higher-level drivers that need direct driver-to-driver interfaces. With NullFM, drivers can attach to other drivers and, once attached, can call them directly from then on.

The basic design elements of this system include the following:

- The NullFM file manager
- A standard OS-9/DPIO driver
- A standard OS-9/DPIO device descriptor
- Common interface library
- Direct driver class-specific interface library

NullFM File Manager

File managers are OS-9 modules that manage specific device classes. Devices with similar sets of characteristics are of the same class and are controlled by a single file manager, regardless of underlying controller hardware differences.

By using NullFM, the direct drivers can be initialized and terminated using the standard `_os_attach()` and `_os_detach()` calls. This greatly reduces the complexity of the higher-level drivers in setting up their links to the direct drivers.

In addition, NullFM defines a common area for all direct driver static storage blocks. This simplifies the calling interfaces by guaranteeing a certain minimum number of fields available for libraries or drivers to access.

NullFM provides the common interface library function `_drvr_direct()`.

Direct Drivers

Under previous architectures, a direct driver was not associated with a file manager. Instead it operated essentially as a sub-routine or system module for the high-level driver. In the new architecture, a direct driver is a full-fledged driver. It can be called via the standard OS-9 functions, or directly by another driver.

A direct driver class provides a specific interface library, which can be linked to and called by any higher level drivers. These specific interface functions accept parameters from the caller and call NullFM's common interface library function `_drvr_direct()`, which takes the following three parameters:

- A pointer to a device list entry structure
- A function code defined and known by the underlying direct driver
- A pointer to the parameter block specific to the driver's function being called

NullFM and the common interface library are common to all direct drivers and are provided by Microware. Thus, future design and implementation of any direct driver involves only three entities:

- The device driver
- The device descriptor
- The direct driver class-specific interface library

Direct Drivers for this version of NullFM include the following:

- SPI (Serial Peripheral Interface)
- FLASH

Device Descriptors

The direct driver's device descriptor is a standard OS-9/DPIO device descriptor. There are no special considerations involved in its creation. Since the direct drivers use NullFM, their device descriptor description files are kept in `MWOS/SRC/DPIO/NULLFM/DESC`.

Common Interface Library

Direct driver-specific class interface library functions call the direct driver through the _drvr_direct() function. This function is in the library MWOS/<OS>/<CPU>/LIB/ddc.l.

Function prototypes for this library are in the include file MWOS/OS9000/SRC/DEFS/ddc.h.

Note

The location of the include file may change in a future release.

Direct Driver Class-Specific Interface Library

Direct driver class-specific interface library functions are used by higher level drivers to access direct drivers' services. These functions are in the library MWOS/<OS>/<CPU>/LIB/<driver class name>.l. for the direct driver classes SPI and Flash.

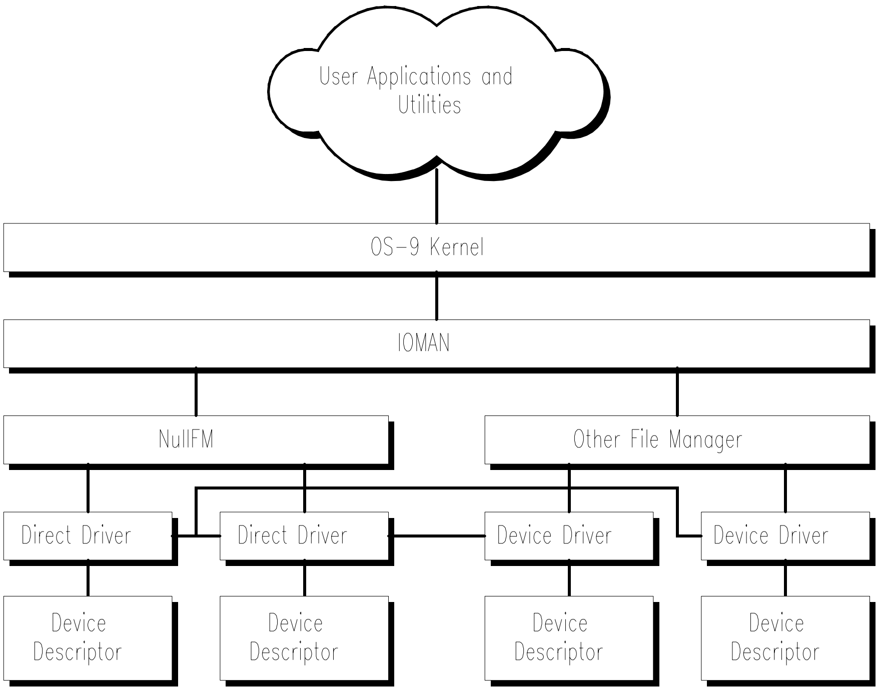
Function prototypes for the library are in the include file MWOS/OS9000/SRC/DEFS/<driver class name>.h.

Note

The general location for all driver class function prototype include files may change in a future release.

NullFM System Architecture

Figure 1-1 Basic NullFM System Architecture

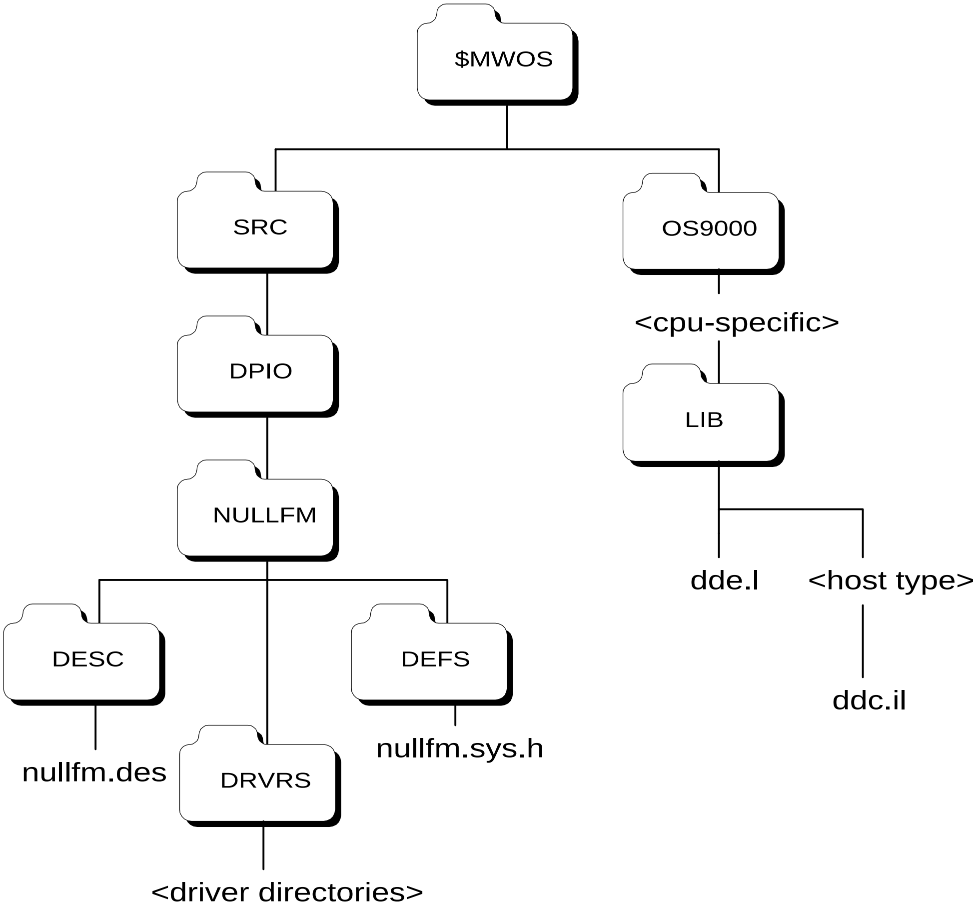


Through NullFM, direct drivers are initialized and terminated using the standard `_os_attach()` and `_os_detach()` calls.

The direct driver is, in virtually every sense, a standard device driver. It is possible to call any of the standard entry points in a direct driver (`read`, `write`, `getstat`, `setstat`, `init`, `term`), although in typical situations only `init` and `term` will be used. Most accesses will occur through the seventh entry point (`v_dd_entry`). This direct entry point is intended for use by other device drivers as a quick and efficient mechanism to get into a direct driver.

NullFM System Files

Figure 1-2 System Files Directory Tree



Using NullFM for Direct Drivers

NullFM (Null File Manager) is used by higher-level direct drivers for direct driver-to-driver interfaces. It is designed, as much as possible, to be invisible to the drivers and passes all application calls down to the driver. It provides no special services or resources to the driver, other than initializing the system globals pointer (`v_sysglob`) in the driver static storage.

Specifically, direct drivers use NullFM for two reasons.

1. Initialization and Termination

   Through NullFM, direct drivers can be initialized and terminated using the standard `_os_attach()` and `_os_detach()` calls. This greatly reduces the complexity of the high-level drivers in setting up their links to the direct drivers.

2. Driver Static Storage Blocks

   NullFM defines a common area for all direct driver static storage blocks. This simplifies the calling interfaces by guaranteeing a certain minimum number of fields available for libraries or drivers to access. These fields are defined in `MWOS/SRC/DPIO/NULLFM/FM/DEFS/nullfm.h`.

   In addition to the standard IOMAN entry points there is one additional entry point for gaining direct access to the driver and a pointer to the system globals, which is initialized by NullFM.

For More Information

The `_driver_statics` structure is described in the **NullFM Programming Reference** section of this chapter.

Calling NullFm

The calling interface to NullFM is a standard OS-9 file manager. NullFM provides support for access to all of the driver entry points except for `v_dd_entry`. This entry point is accessed only through a special library intended to be used by other device drivers requiring this direct access.

Using Direct Drivers

Direct drivers are generally used to:

- Arbitrate access to a specific set of device registers on behalf of several high-level drivers.
- Regulate communication on a device bus (such as SCSI or SPI) with multiple peripherals controlled by multiple high-level drivers.

They typically perform the following services for a higher level driver:

- Initialize/deinitialize a device
- Set/clear interrupt enable conditions
- Install/execute an interrupt service routine
- Read/write blocks of data from/to a device
- Return status conditions for the device
- Get/set operating parameters for the device

All of these calls, except for the initialize/deinitialize calls, can come through the same direct entry point in the direct driver, which examines a function code in the parameter block to determine the appropriate action to take.

Data Dependencies

A direct driver depends on a generic header file common to all devices of similar functionalities (for example, `spi.h`, `atn.h`). It also depends on a header file that defines the system structures of the driver (for example, `spi_sys.h`).

The function prototype for interface function `_drvr_direct()` and the structure definitions for the parameter block are located in:

```
MWOS/OS9000/SRC/DEFS/ddc.h.
```

Direct Driver Interface

The direct driver interface conforms to the definitions of a standard OS-9/DPIO device driver. The function interfaces for all of the standard driver entry points are the same as those defined by other OS-9 file managers (for example, SCF and RBF), including the following:

- `drv_init()`
- `drv_term()`
- `drv_setstat()`
- `drv_getstat()`
- `drv_read()`
- `drv_write()`

Direct Calling Entry Point

The direct calling entry point is defined as:

```
drv_dd_entry()
```

Note

The caller WILL NOT set up the direct driver's static storage before calling. If the direct driver needs access to the static storage, it must get a pointer to its static storage from the device list entry that is passed to the function.

For More Information

The format of the default `dd_entry_pb` parameter block, which is defined in the `ddc.h`, is described in the **NullFM Programming Reference** section of this chapter.

drv_dd_entry() Design

The `drv_dd_entry()` function in a direct driver works much like the standard OS-9 functions `drv_setstat()` or `drv_getstat()`. The driver looks at the first part of the parameter block to determine the function code, and then executes an appropriate set of actions based on that function request.

Sleeping Rules for Direct Drivers

The direct driver should never sleep in a function called through the direct entry point. These entry points may, in fact, be called from a higher level driver's interrupt service routine. If it is necessary to sleep in a given function, the driver should implement a mechanism to get to the function through the `drv_setstat()` entry point, which is allowed to sleep during its operation. However, it still may not be called from an interrupt context. Instead, direct drivers should implement non-blocking services with callbacks — to the higher-level driver requesting service — when services are complete. This permits efficient CPU utilization.

Writing a Direct Driver for NullFM

There are two potential situations in which you would implement a driver for NullFM.

1. You need a driver for a new device that is in an existing direct driver class (for example SPI or Flash).

   In this case, you choose the appropriate driver (SPI or FLASH) and use it as a model — making modifications as required by your hardware device.

2. You need a driver for a new device that is in a new (custom) direct driver class.

   In this case, writing the driver is more complicated and requires architectural design. To obtain architectural advice or review of a custom direct driver class, contact Microware Professional Services.

NullFM Programming Reference

Direct Driver Common Interface Functions

Driver-specific interface library functions call the direct driver through the `_drvr_direct()` function. This function is in `MWOS/<OS>/<CPU>/LIB/ddc.l`. Function prototypes for this library are in `MWOS/SRC/DEFS/ddc.h`.

The following lists and describes the functions that compose the `ddc.l` library.

Table 1-1 ddc.l Library Functions and Descriptions

Function	Description
`_drvr_direct()`	Request Service of a Direct Driver

_drvr_direct() Request Service of a Direct Driver

Syntax

```
#include <ddc.h>
error_code _drvr_direct(
     Dev_list     dev,
     u_int32      direct_code,
     void         pb); /* Driver_class_functionpb */
                       /* pointer_type */
```

Libraries

ddc.l

Description

`_drvr_direct()` looks in the device list entry for a pointer to the driver statics. The driver statics are used to obtain a pointer to the driver's direct entry point. If this function pointer is not `NULL`, the library will call the function, passing a pointer to the `dd_entry_pb` created from the remaining parameters, and returning whatever value is returned by the direct entry point function. If the direct entry point is `NULL`, the library will return `E_UNKSVC`.

Parameters

`dev`	Pointer to the direct device driver's device list entry. This should be the same value as that returned from `_os_attach()`.
`direct_code`	Function code defined by the direct driver class
`pb`	Pointer to a parameter block for the direct function being called in the driver. This parameter block is unique for each function, and defined by the direct driver class.

Errors

`SUCCESS`	Function executed successfully
`E_UNKSVC`	Either the driver does not support a direct entry point or the specified function is not supported
`E_PARAM`	Illegal or out-of-range parameters specified for this function
`E_NOTRDY`	Device is not ready to accept this command
`E_DEVBSY`	Device is not able to execute this function at this time

Direct Driver Common Interface Structures

This section includes the structure definitions in `ddc.h` and `nullfm.h` that are used by NullFM and direct drivers under NullFM. The structures are listed in the following table.

Table 1-2 `ddc.h` and `nullfm.h` Structures and Definitions

Structure	Definition
`dd_entry_pb`	Direct Driver Entry Parameter Block
`_driver_statics`	Driver Statistics Structure

dd_entry_pb

Direct Driver Entry Parameter Block

Declaration

The dd_entry_pb structure is declared in the file ddc.h file as follows:

```
typedef struct
{
     error_code   error;
     u_int32      func_code;
     void         *param_blk;
} dd_entry_pb, *Dd_entry_pb;
```

Description

This structure is built by the _drvr_direct() function in ddc.l to pass the direct_code and pb parameters to the driver and to provide a mechanism for the driver to return an error or success code.

Fields

error	Error code returned by direct driver function
func_code	Direct function code
*param_blk	Direct function-specific parameters

_driver_statics — Driver Statistics Structure

Declaration

The _driver_statics structure is declared in the file nullfm.h file as follows:

```
typedef struct
{
     error_code
          (*v_init)(),
          (*v_read)(),
          (*v_write)(),
          (*v_getstat)(),
          (*v_setstat)(),
          (*v_term)(),
          (*v_dd_entry)();
     sysglobs
          *v_sysglob;
} drvr_statics, *Drvr_statics;
```

Description

This structure is the interface between NullFM and a direct driver. The function pointers are provided when a driver static storage is initialized by IOMAN. NullFM initializes the system globals pointer before calling the driver.

Fields

`(*v_init)()`	Address of driver's `init` function
`(*v_read)()`	Address of driver's `read` function
`(*v_write)()`	Address of driver's `write` function
`(*v_getstat)()`	Address of driver's `get_status` function
`(*v_setstat)()`	Address of driver's `set_status` function
`(*v_term)()`	Address of driver's `terminate` function
`(*v_dd_entry)()`	Address of driver's `direct entry` function
`*v_sysglob`	System globals — initialized by file manager

Chapter 2: Using the SPI Direct Driver Class

This chapter describes the SPI direct driver class. It includes the following sections:

- **SPI Direct Driver Overview**
- **Using SPI Direct Driver**
- **SPI Direct Driver Programming Reference**
- **Porting SPI Direct Drivers**

SPI Direct Driver Overview

A Serial Peripheral Interface (SPI) driver provides data transfer services through an SPI host interface to SPI bus devices. Both master and slave interface operations are supported.

High-level drivers access the SPI driver through the SPI interface library. Applications or other entities can also use an SPI driver's services.

SPI is a low-level driver used to communicate with devices on an SPI bus. SPI is invisible to application programs. It is only accessed by other drivers that are written to control devices on an SPI bus.

The Microware SPI driver is used primarily by original equipment manufacturers (OEMs) when porting DAVID to their consumer device.

Both master and slave interface operations are supported. However, dynamic switching between master mode and slave mode requires that external (hardware) signalling and handshake be handled by the slave application or high-level driver. These would use the SPI Class driver services for the actual data transfer.

SPI Interface Library

A high-level driver uses SPI class driver services via the SPI interface library. The SPI interface library formats parameter blocks appropriately and passes the service requests on to the SPI class driver via the low-level driver common interface library.

To access the SPI Class driver's device list entry, the high-level driver (or an entity on it's behalf) must attach the SPI device (using `_os_attach()`) before using it. Similarly, the high-level driver must detach the device after completing its operation.

The SPI interface library function prototypes and attribute value definitions can be found in:

`MWOS/OS9000/SRC/DEFS/spi.h`

The functions themselves can be found in:

MWOS/<OS>/<CPU>/LIB/spi.l and
MWOS/<OS>/<CPU>/LIB/<HOSTTYPE>/spi.il

For More Information

The SPI functions are described in the **SPI Direct Driver Programming Reference** section of this chapter.

SPI Data Dependencies

High-level drivers must use the SPI interface library to access an SPI class driver's services.

SPI Interface

Each service provided by an SPI class driver has a specific parameter block structure, containing the parameters passed to and from the high-level driver, via the SPI interface library.

For More Information

The SPI parameter block structures are defined in the **SPI Direct Driver Programming Reference** section of this chapter.

SPI Device Descriptor

The general SPI device descriptor definition file is located in:

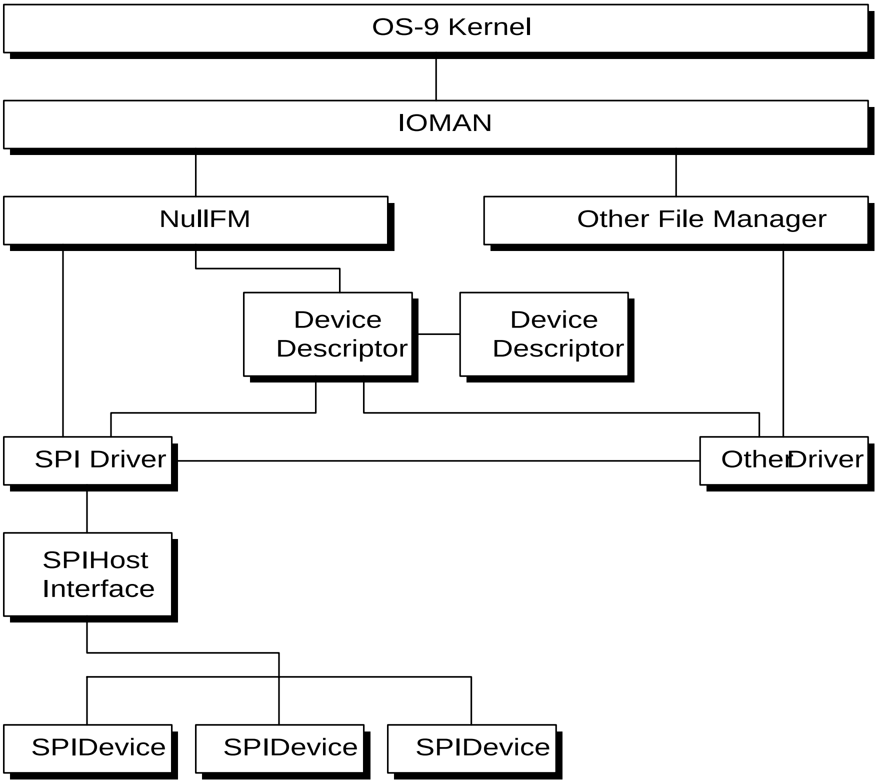
`MWOS/SRC/DPIO/NULLFM/DESC/spi.des.`

In addition, a specific SPI driver may define a device-specific static storage area within its own `editmod` descriptor (`.des`) file.

The example `CPMSPI` driver defines device-specific storage in `MWOS/SRC/DPIO/NULLFM/DRVR/CPMSPI/cpmspi.des`.

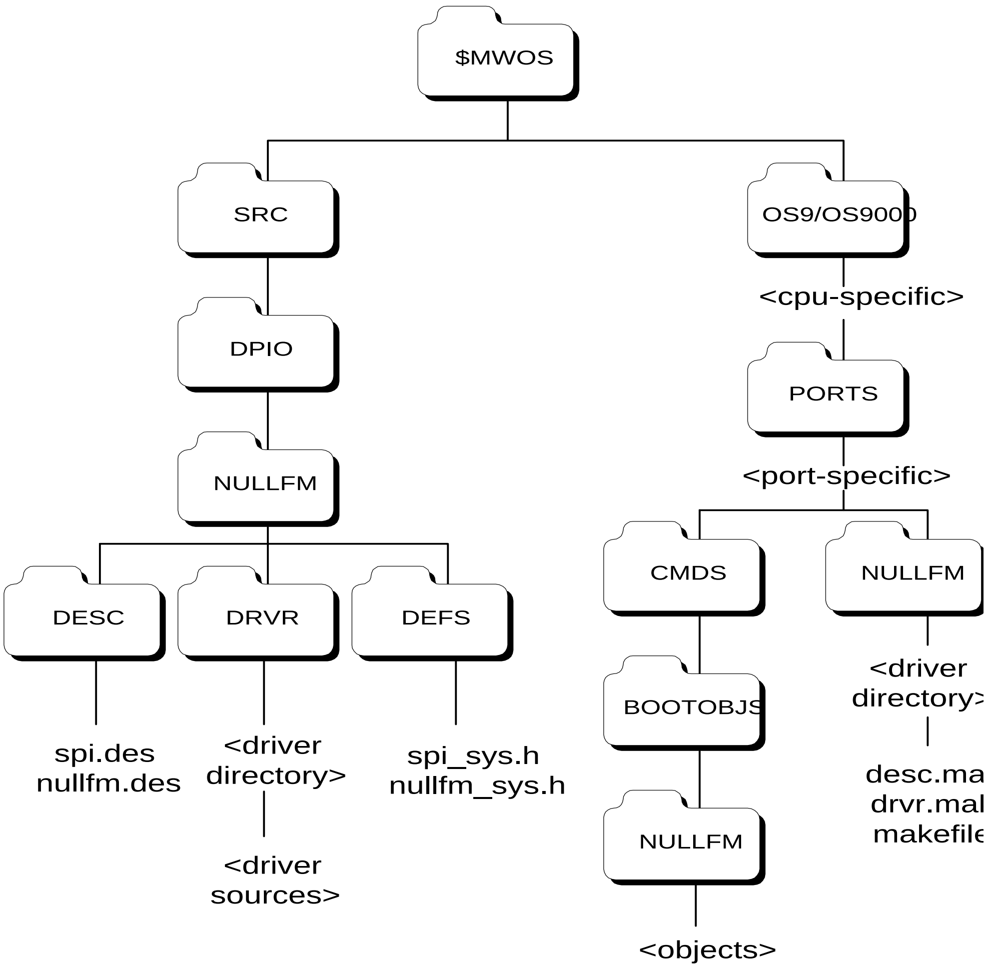
SPI Direct Driver System Architecture

Figure 2-1 SPI Direct Driver Architecture



SPI Direct Driver System Files

Figure 2-2 SPI Direct Driver System Files Directory Tree



Using SPI Direct Driver

Using SPI Direct Drivers

Using an SPI driver involves the following procedures.

1. Building a SPI driver to support the SPI host interface in your system.

2. Building a descriptor to describe the SPI host interface device in your system.

3. Including NullFM, your SPI driver, and your SPI device descriptor in your system boot image, along with any other drivers which make use of the SPI driver services.

Writing SPI Direct Drivers

Writing a new SPI driver typically involves the following basic steps.

Step 1. 1. Make a copy of the example driver sources in a new directory.

Step 2. 2. Identify the SPI host interface-specific routines in the driver and modify or rewrite them for the host interface for which support is desired.

Step 3. 3. Create a build subdirectory in your target port directory. Use the example port directory structure as a guide.

Step 4. 4. Copy the makefile used to build the SPI driver from the example port directory to your target port directory. Modify the SDIR macro to point to your modified driver sources.

Step 5. 5. Build your new driver, test and debug it.

Sample SPI Direct Driver

Sources for the example SPI driver can be found at:

```
MWOS/SRC/DPIO/NULLFM/DRVR/CPMSPI
```

SPI Direct Driver Programming Reference

SPI-Specific Interface Functions

The SPI function prototypes and attribute value definitions reside in `MWOS/OS9000/SRC/DEFS/spi.h`. The functions themselves reside in the libraries, `MWOS/<OS>/<CPU>/LIB/spi.l` and `MWOS/<OS>/<CPU>/LIB/<HOSTTYPE>/spi.il`.

The following table lists and describes the functions that compose the `spi.l` and `spi.il` libraries.

Table 2-1 spi.l and spi.il Library Functions and Descriptions

Function	Description
`_spi_deregister_device()`	Deregister a Device
`_spi_enter_slave_mode()`	Request Slave Mode
`_spi_exit_slave_mode()`	Exit Slave Mode
`_spi_register_device()`	Register Data Transfer Configuration
`_spi_tranfer_data()`	Transfer Data to and from the Target

_spi_deregister_device() — Deregister a Device

Syntax

```
error_code _spi_deregister_device(
     Dev_list     spi_interface_device,
     SPIDeviceHandle handle);
```

Libraries

`spi.l`, `spi.il`

Description

`_spi_deregister_device()` indicates that access to the corresponding SPI device is no longer required. It is called by a higher-level driver. This service must be called before the higher-level driver detaches from SPI interface device.

Parameters

`spi_interface_device`	Pointer to the device list entry returned from `_os_attach()`.
`handle`	The SPI device handle returned by `_spi_register_device()`.

Errors

`EOS_DEVBSY`	Device is not able to execute this function at this time. A data transfer is active.
`EOS_PARAM`	Illegal or out-of-rom parameters specified for this function. The handle was not recognized as valid or the device is in slave mode.

See Also

`_spi_register_device()`

`_spi_enter_slave_mode()` — Request Slave Mode

Syntax

```
error_code _spi_enter_slave_mode(
     Dev_list     spi_interface_device,
     void         (*slave_entered)(void *),
     void         *callback_param,
     SPIDeviceHandle handle);
```

Libraries

`spi.l`, `spi.il`

Description

To make use of the SPI host interface in slave mode, a higher-level driver must call `_spi_enter_slave_mode()`. This initiates the interface to begin processing in that mode. If the SPI interface is not capable of operating in slave mode, an error is returned immediately. Otherwise, the SPI Class driver indicates through a provided callback function that the interface is switched into slave mode. Once the SPI Class driver indicates slave mode operation has been entered, all master mode functions are suspended.

Note

Successful return of the service DOES NOT indicate slave mode entry completion.

The `slave_entered` callback function provided by the higher-level driver must match the following function prototype:

```
void slave_entered (void *callback_parameter);
```

Parameters

`spi_interface_device`	Pointer to the device list entry returned from `_os_attach()`.
`(*slave_entered)(void *)`	Pointer to `slave_entered()` callback function in the higher-level driver.
`*callback_param`	Pointer parameter to be passed to `slave_entered()` callback function.
`handle`	The SPI device handle returned by `_spi_register_device()`.

Errors

`EOS_DEVBSY`	Device is not able to execute this function at this time. A data transfer is active.
`EOS_PARAM`	Illegal or out-of-rom parameters specified for this function. The handle was not recognized as valid or the device is in slave mode.

See Also

`_spi_exit_slave_mode()`

`_spi_exit_slave_mode()` Exit Slave Mode

Syntax

```
error_code _spi_exit_slave_mode(
     Dev_list     spi_interface_device,
     SPIDeviceHandle handle);
```

Libraries

`spi.l`, `spi.il`

Description

`_spi_exit_slave_mode()` causes the SPI interface to exit slave mode and resume master mode operations. If the interface is not in slave mode before you call `_spi_exit_slave_mode()`, an error is returned immediately.

Parameters

`spi_interface_device`	Pointer to the device list entry returned from `_os_attach()`.
`handle`	The SPI device handle returned by `_spi_register_device()`.

Errors

`EOS_DEVBSY`	Device is not able to execute this function at this time. A data transfer is active.
`EOS_NOTRDY`	Enter slave mode request is pending. The `exit_slave_mode` can only be done after the `slave_entered()` callback function has been called.

`EOS_PARAM`	Illegal or out-of-ROM parameters specified for this function. The handle was not recognized as valid or the device is in slave mode.

See Also

`_spi_enter_slave_mode()`

`_spi_register_device()` Register Data Transfer Configuration

Syntax

```
typedef void *SPIDeviceHandle;
error_code _spi_register_device(
     Dev_list     spi_interface_device,
     u_int32      clock_rate,
     u_int32      clock_attributes,
     u_int32      data_attributes,
     u_int32      data_priority,
     void         (*slave_enable)(void *),
     void         *enable_param,
     void         (*slave_disable)(void *),
     void         *disable_param,
     SPIDeviceHandle *handle);
```

Libraries

`spi.l`, `spi.il`

Description

`_spi_register_device()` registers the data transfer configuration of the SPI device it will communicate with. It is called by a higher-level driver. Specific data transfer attributes, as well as slave enable and disable functions are passed to the SPI class driver in this service. After successful registration, a slave device handle is returned, which must be used for all further operations to the device.

The slave enable and disable callback functions provided by the higher-level driver must match the following function prototypes:

```
void slave_enable(void *callback_parameter);
void slave_disable(void *callback_parameter);
```

Parameters

`spi_interface_device`	Pointer to the device list entry returned by `_os_attach()`.
`clock_rate`	Clock rate for slave device.
`clock_attributes`	Clock attributes for slave device (see `spi.h` device values).
`data_attributes`	Attributes for slave device data transfer (see `spi.h` for values).
`data_priority`	Priority for slave device data transfer.
`(*slave_enable)(void *)`	Pointer to `slave_enable()` callback function in the higher-level driver.
`*slave_disable)(void *)`	Pointer to `slave_disable()` callback function in the higher-level driver.
`*enable_param`	Pointer parameter to be passed to `slave_enable()` callback function.
`*disable_param`	Pointer parameter to be passed to `slave_disable()` callback function.
`*handle`	Pointer to handle variable to receive the returned SPI device handle.

Errors

`EOS_NORAM`	All free device control blocks have been consumed. The number of device control blocks allocated by the driver is controlled by the `v_max_slaves` value in the device descriptor (plus 1).

See Also

`_spi_deregister_device()`

`_spi_tranfer_data()` Transfer Data to and from the Target

Syntax

```
error_code _spi_transfer_data(
     Dev_list     spi_interface_device,
     u_char       *transmit_buffer,
     u_char       *receive_buffer,
     u_int32      buffer_size,
     void         (*xfer_complete)(void
*,error_code),
     void         *callback_param,
     SPIDeviceHandle handle);
```

Libraries

`spi.l`, `spi.il`

Description

`_spi_transfer_data()` transfers data to and from the target SPI device. It is called by a higher-level driver. Transmit and receive data buffers must be supplied, and both must be the same size. A callback function (and parameter) must be supplied for the SPI Class driver to indicate when data transfer is complete.

Note

Successful return of the service DOES NOT indicate data transfer completion.

The transfer_complete callback function provided by the higher-level driver must match the following function prototype:

```
void *xfer_complete (void *callback_parameter,
error_code status);
```

Parameters

`spi_interface_device`	Pointer to the device list entry returned from `_os_attach()`
`*transmit_buffer`	Pointer to a valid transmit buffer
`*receive_buffer`	Pointer to a valid receive buffer
`buffer_size`	Size of the transmit and receive buffers (both must be the same size)
`(*xfer_complete)(void *,error_code)`	Pointer to `xfer_complete()` callback function in the higher-level driver
`*callback_param`	Pointer parameter to be passed to `xfer_complete()` callback function
`handle`	The SPI device handle returned by `_spi_register_device()`

Errors

`EOS_DEVBSY`	Device is not able to execute this function at this time. A data transfer is active.
`EOS_NORAM`	All free device control blocks have been consumed. The number of device control blocks allocated by the driver is controlled by the `v_max_slaves` value in the device descriptor [(+1), (+2)].
`EOS_PARAM`	Illegal or out-of-rom parameters specified for this function. The handle was not recognized as valid or the device is in slave mode.

SPI Direct Driver Structures

This section includes the structure definitions in `spi.h` that are used by SPI direct drivers. The structures are listed in the following table.

Table 2-2 `spi.h` Structures and Definitions

Structure	Definition
`_spi_reg_dev_pb`	Register Device Function Parameter Block
`_spi_dereg_dev_pb`	Deregister Device Function Parameter Block
`_spi_xfer_data_pb`	Transfer Device Function Parameter Block
`_spi_enter_slave_pb`	Enter Slave Function Parameter Block
`_spi_exit_slave_pb`	Exit Slave Function Parameter Block

_spi_reg_dev_pb — Register Device Function Parameter Block

Declaration

The _spi_reg_dev_pb structure is declared in the file [*.h file] as follows:

```
typedef struct
{
     u_int32        vers_id;
     u_int32        clock_rate,
                    clock_attributes,
                    data_attributes,
                    data_priority;
     void           (*slave_enable)(void *),
                    (*slave_disable)(void *);
     void           *enable_param,
                    *disable_param;
     SPIDeviceHandle
                    handle;
} spi_reg_dev_pb, *SPI_Reg_Dev_pb;
```

Description

This structure is built by the `spi_register_device()` function in `spi.l` to pass all parameters to the driver via the `_drvr_direct()` function.

Fields

`vers_id`	Structure version identifier
`clock_rate`	Clock rate for slave device
`clock_attributes`	Clock attributes for slave device (see `spi.h` device values)
`data_attributes`	Attributes for slave device data transfer (see `spi.h` for values)
`data_priority`	Priority for slave device data transfer
`(*slave_enable)(void *)`	Pointer to `slave_enable()` callback function in the higher-level driver
`(*slave_disable)(void *)`	Pointer to `slave_disable()` callback function in the higher-level driver
`*enable_param`	Pointer parameter to be passed to `slave_enable()` callback function
`*disable_param`	Pointer parameter to be passed to `slave_disable()` callback function
`*handle`	SPI handle to return to caller device

_spi_dereg_dev_pb Deregister Device Function Parameter Block

Declaration

The _spi_dereg_dev_pb structure is declared in the file spi.h as follows:

```
typedef struct
{
     u_int32      vers_id;
     SPIDeviceHandlehandle;
} spi_dereg_dev_pb, *SPI_Dereg_Dev_pb;
```

Description

This structure is built by the _spi_deregister_device() function in spi.l to pass all parameters to the driver via the _drvr_direct() function.

Fields

vers_id	Structure version identifier.
handle	Parameter from _spi_deregister_device() function in spi.l.

`_spi_xfer_data_pb` Transfer Device Function Parameter Block

Declaration

The `_spi_xfer_data_pb` structure is declared in the file `spi.h` as follows:

```
typedef struct
{
     u_int32      vers_id;
     u_char       *xmit_buf,
                  *rcv_buf;
     u_int32      buf_leng;
     void         (*xfer_complete)(void *,
error_code);
     void         *callback_param;
     SPIDeviceHandle
                  handle;
} spi_xfer_data_pb, *SPI_Xfer_Data_pb;
```

Description

This structure is built by the `_spi_transfer_data()` function in `spi.l` to pass all parameters to the driver via the `_drvr_direct()` function.

Fields

`vers_id`	Structure version identifier
`*xmit_buf`	Pointer to a valid transmit buffer
`*rcv_buf`	Pointer to a valid receive buffer
`buf_leng`	Size of the transmit and receive buffers (both must be the same size)
`(*xfer_complete)(void *,error_code)`	Pointer to `xfer_complete()` callback function in the higher-level driver
`*callback_param`	Pointer parameter to be passed to `xfer_complete()` callback function
`handle`	The SPI device handle returned by `_spi_register_device()`

_spi_enter_slave_pb — Enter Slave Function Parameter Block

Declaration

The _spi_enter_slave_pb structure is declared in the file spi.h as follows:

```
typedef struct
{
     u_int32      vers_id;
     void         (*slave_entered)(void *);
     void         *callback_param;
     SPIDeviceHandlehandle;
} spi_enter_slave_pb, *SPI_Enter_Slave_pb;
```

Description

This structure is built by the _spi_enter_slave() function in spi.l to pass all parameters to the driver via the _drvr_direct() function.

Fields

vers_id	Structure version identifier
(*slave_entered)(void *)	Pointer to slave_entered() callback function in the higher-level driver
*callback_param	Pointer parameter to be passed to slave_entered() callback function
handle	The SPI device handle returned by _spi_register_device()

_spi_exit_slave_pb Exit Slave Function Parameter Block

Declaration

The _spi_exit_slave_pb structure is declared in the file spi.h as follows:

```
typedef struct
{
     u_int32      vers_id;
     SPIDeviceHandlehandle;
} spi_exit_slave_pb, *SPI_exit_slave_pb;
```

Description

This structure is built by the _spi_exit_slave() function in spi.l to pass all parameters to the driver via the _drvr_direct() function.

Fields

vers_id	Structure version identifier
handle	The SPI device handle returned by _spi_register_device()

Porting SPI Direct Drivers

The SPI Direct Drivers have the following build options for customization to the target platform.

Table 2-3 SPI Direct Drivers Build Options

Build Option	Description
`CACHE_INHIBITED_BITES`	Defined for a target system whose SPF driver buffer storage resides in cache-inhibited memory
`CACHE_CANSTORE`	Defined for a target system whose cache support module supports cache store functions and the buffer storage resides in cacheable memory
`CACHE_CANVALIDATE`	Defined for a target system whose cache support module supports cache invalidation functions and the buffer storage resides in cacheable memory
`MASTER_ONLY`	Defined for a target system whose SPI host interface can only operate in master mode

Chapter 3: Using the FLASH Direct Driver Class

This chapter describes the FLASH direct driver class. It includes the following sections:

FLASH Direct Driver Overview

FLASH drivers are a specific class of direct drivers that are part of the NullFM file/device manager systems. A FLASH driver provides general read and write services to FLASH devices in a part-independent manner. It also provides part geometry information for those system entities which must have knowledge of it for optimal operation.

Note

The Microware example FLASH driver for the Hellcat provides more functionality than the architecture described here. Future general releases may or may not adopt some of all of that additional functionality.

FLASH Interface Library

System entities (higher-level drivers, module manager and utilities) use FLASH driver services via the standard I/O calls and the FLASH interface library. The FLASH interface library formats parameter blocks appropriately and passes the service requests on to the FLASH Direct Driver via the standard `_os_getstat()` and `_os_setstat()` library calls.

To use the FLASH driver's services, the caller must open a path to the FLASH device (using `_os_open()`). Similarly, the caller must close the path after completing its operation.

The FLASH interface library function prototypes can be found in:

```
MWOS/OS9000/SRC/DEFS/flashlib.h
```

Note

The name and location of the `flashlib.h` include file may change in a future release.

The functions themselves can be found in:

`MWOS/<OS>/<CPU>/LIB/flash.l` and
`MWOS/<OS>/<CPU>/LIB/<HOSTTYPE>/flash.il`

Note

The names of the FLASH library files may change in a future release.

For More Information

The FLASH functions are described in the **FLASH Direct Driver Programming Reference** section of this chapter.

FLASH Data Dependencies

Users of the FLASH direct drivers must use the standard I/O calls and the FLASH interface library to access a FLASH Direct Driver's services.

FLASH Interface

The FLASH driver services require use of existing parameter block structures containing the parameters passed to and from the FLASH users via the FLASH interface library.

For More Information

The FLASH parameter block structure is described in the **FLASH Direct Driver Programming Reference** section of this chapter.

FLASH Device Descriptor

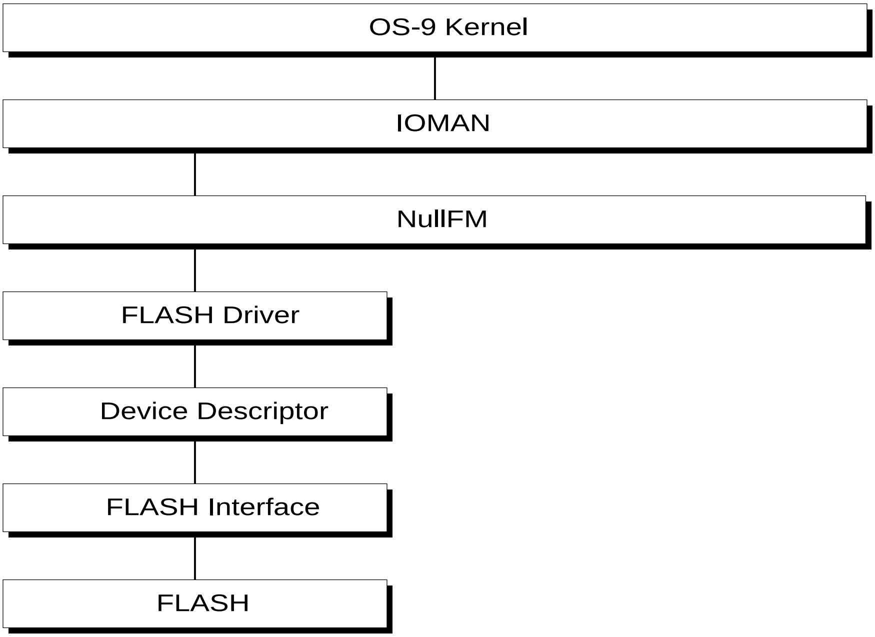
The general FLASH device descriptor definition file is located in:

```
MWOS/SRC/DPIO/NULLFM/DESC/flash.des
```

There are no fields for FLASH devices beyond the standard NullFM device descriptor.

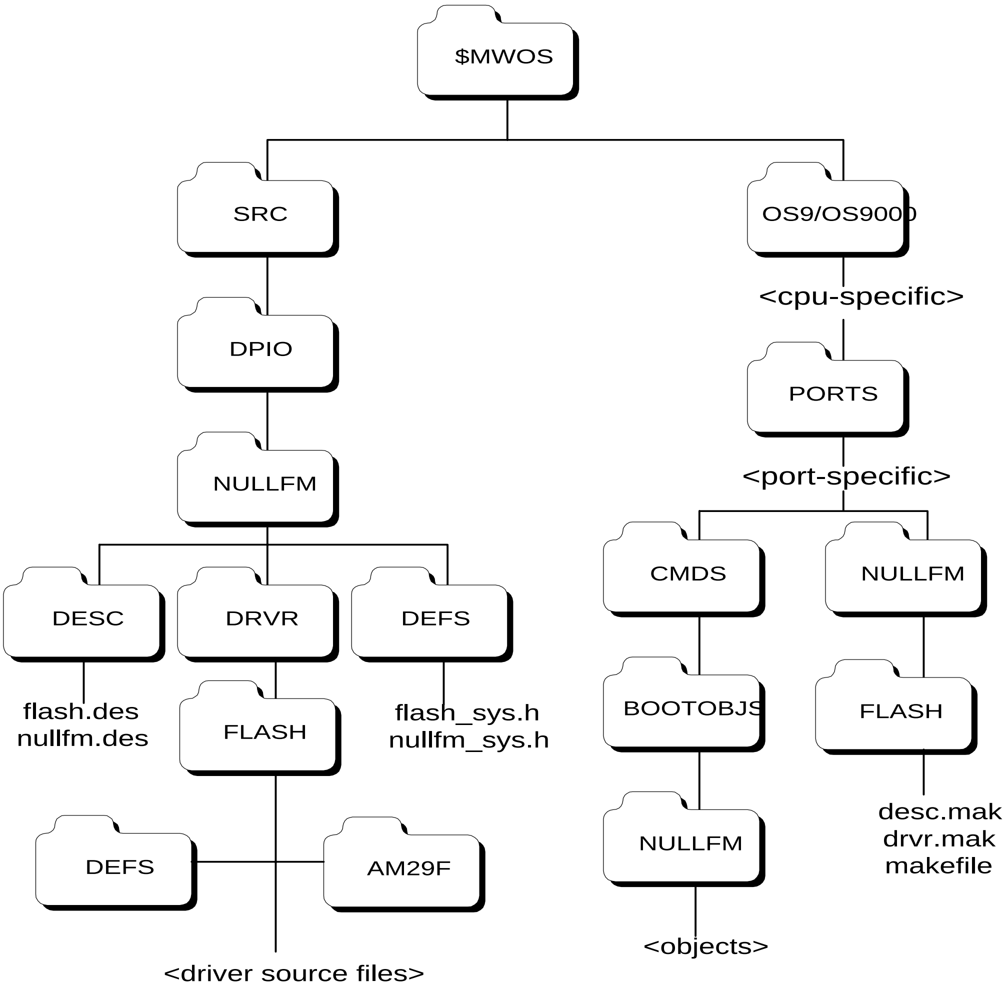
FLASH System Architecture

Figure 3-1 FLASH Direct Driver Architecture



FLASH Direct Driver System Files

Figure 3-2 FLASH Direct Driver System Files Directory Tree



Using FLASH

Using FLASH Direct Driver

Using the FLASH driver involves the following procedures.

1. Building a FLASH driver to support the FLASH hardware in your system.
2. Building a descriptor to describe the mapping of FLASH devices in your system.
3. Including NullFM, your FLASH driver, and you FLASH device descriptor in your system boot image, along with any user code which makes use of the FLASH driver services.

Writing FLASH Direct Driver

Write a new FLASH driver typically involves the following basic steps.

Step 1. Make a copy of the example driver sources in a new directory.

Step 2. Identify the part-specific routines in the driver and modify or rewrite them for the parts for which support is desired.

Step 3. Create a build subdirectory in your target port directory. Use the example port directory as a guide.

Step 4. Copy the makefile used to build the FLASH driver from the example port directory to your target port directory. Modify the SDIR macro to point to your modified driver sources.

Step 5. Build your new driver, test and debug it.

Sample FLASH Direct Driver

Sources for the example FLASH driver can be found in:

```
MWOS/SRC/DPIO/NULLFM/DRVR/FLASH
```

FLASH Direct Driver Programming Reference

Supported Standard I/O Interface Functions

The standard I/O functions supported by the FLASH Direct Driver class include the following:

```
_os_open()
_os_close()
_os_read()
_os_write()
```

The following functions are not supported:

```
_os_readln()
_os_writeln()
```

FLASH-Specific Interface Functions

The FLASH function prototypes reside can be found in:

`MWOS/OS9000/SRC/DEFS/flashlib.h`

Note

The name and location of the `flashlib.h` include file may change in a future release.

The functions themselves reside in:

`MWOS/<OS>/<CPU>/LIB/flash.l` and
`MWOS/<OS>/<CPU>/LIB/<HOSTTYPE>/flash.il`

Note
The names of the FLASH library files may change in a future release.

The following table lists and describes the functions that compose the FLASH libraries.

Table 3-1 FLASH Functions

Function	Description
`_flash_gs_dsize()`	Retrieve Number of Sectors and Current Sector Size
`_flash_gs_pos()`	Retrieve the Current Path Position
`_flash_ss_pos()`	Establish the Current Path Position

`_flash_gs_dsize()` Retrieve Number of Sectors and Current Sector Size

Syntax

```
error_code _flash_gs_dsize(
     path_id      spath,
     u_int32      *sectors,
     u_int32      *size);
```

Libraries

`flash.l, flash.il`

Description

Returns the total number of sectors for the device, and the size of the sector currently positioned in.

If the position is set beyond the size of the device, the value of the size returned is zero.

Parameters

`spath`	Path id returned by `_os_open()`
`*sectors`	Pointer to the number of sectors variable
`*size`	Pointer to the size of sector variable

Errors

None

See Also

`_flash_gs_pos()`

`_flash_ss_pos()`

_flash_gs_pos() Retrieve the Current Path Position

Syntax

```
error_code _flash_gs_pos(
     path_id     spath,
     u_int32     *position);
```

Libraries

`flash.l, flash.il`

Description

The current device byte position for the path is returned at the location pointed to by the position pointer variable.

Parameters

`spath`	Path id returned by `_os_open()`
`*position`	Pointer to a position variable

Errors

None

See Also

`_flash_ss_pos()`

`_flash_ss_pos()` Establish the Current Path Position

Syntax

```
error_code _flash_ss_pos(
     path_id      spath,
     u_int32      position);
```

Libraries

`flash.l, flash.il`

Description

The current device byte position for the path is set to the position variable passed on the call.

If the position is set beyond the size of the device, `EOS_EOF` is returned on the next `_os_read()` or `_os_write()` operation performed.

Parameters

`spath`	Path id returned by `_os_open()`.
`Position`	The position variable.

Errors

None

See Also

`_flash_gs_pos()`

FLASH Direct Driver Structures

There are no FLASH-specific Direct Driver Structures.

Porting FLASH Direct Drivers

The FLASH Direct Drivers have the following build options for customization to the target platform:

Table 3-2 FLASH Build Options

Build Option	Description
`INVALIDATE` (default)	Defined for a target system whose cache support module supports invalidation and the FLASH region is marked as cacheable. The FLASH region must also be marked as Data Write Through.
`NO_INVALIDATE`	Defined for a target system whose cache support module does not support invalidation and the FLASH region is marked as cacheable. Cache disables and enables are done automatically by the driver around block writes.

Table 3-2 FLASH Build Options

Build Option	Description
PART_WIDTH	Defined in systype.h to describe the data width of a FLASH part implemented on a target system. Should be set to one of the following values: ONE_BYTE TWO_BYTES FOUR_BYTES
BANK_WIDTH	Defined in systype.h to describe the width (in parts) of a FLASH bank implemented on a target system. Should be set to one of the following values: ONE_PART TWO_PARTS FOUR_PARTS

Note

The example FLASH driver may not implement all of the above build options. Future releases may expand or eliminate some or all of the build options.

Index

Symbols

A

B

C

D

E

F

I

L

M

N

O

P

R

S

T

U

W

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:_______________________________________

Phone:____

Fax:_____________________Email:__________________

Product Name:

Description of Problem:

____

____

____

____

____

____

____

____

____

____

Host Platform_____________________________________

Target Platform___________________________________