MAUI Porting Guide

Version 3.2

www.radisys.com

World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
November 2001

Copyright and publication information

This manual reflects version 3.2 of MAUI.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

November 2001

Table of Contents

Chapter 1: Configuration Description Block

The Configuration Description Block (CDB) contains specific information about your system configuration. This chapter explains what the CDB is, where the files are located, and how to modify, build, and test your CDB.

Overview of the CDB

The Configuration Description Block (CDB) is one or more data modules that describe each device in your system. Applications read the CDB and adjust how they operate at run-time according to what devices are present and what capabilities each device has.

The CDB is central to the concept of application portability. Applications search the CDB data modules via the MAUI CDB API. This API searches memory for all modules of type (5<<8)+1.

Each device in your system is represented by an entry in CDB. The entries are known as Device Description Records (DDRs). DDRs begin with `dc.b` followed by a text string enclosed in quotes. When you customize the `cdb.a` file, you simply modify the `dc.b` lines, adding or subtracting lines as needed.

All DDRs do not have to be in the same CDB data module. CDB data modules may be linked and unlinked from memory as new devices, drivers, and descriptors are added and deleted from the system.

Each `dc.b` line is constructed in the same way:

```
dc.b "dev_type:device_name:parameters:",13
```

Each line begins with `dc.b`, then is followed by the DDR (a string that describes the device). The `,13` serves as the line terminator for each DDR. Following is a simple example of a CDB from an imaginary system that describes eleven devices:

```
    psect cdb,(5<<8)+1,$8000,200,0,entry
    org 0
entry:
    dc.b       "0:sys:CP=\"PPC603\":OS=\"OS9000\":RV=\"2.0\":
               DV=\"2.1\":SR#12288,1:GR#512,128:",13
    dc.b "3:/gfx:AI=\"MAUI\":GR#2048,128:",13
    dc.b "4:/nvr:",13
    dc.b "5:/kx0/mp_xtkbd:",13
    dc.b "5:/m0/mp_bsptr:",13
    dc.b "9:/pipe:",13
    dc.b "20:/term:",13
    dc.b "20:/t1:",13
    dc.b "90:/mv:",13
    dc.b "91:/ma:",13
    dc.b "113:/sp0/lapb/x25:",13
```

```
dc.b "114:/r0:HD:",13
dc.b "1000:/win:",13
dc.b 0
ends
```

The following rules apply to DDR syntax and parameters:

- Each device in your system is represented by an entry. Each entry is a single line beginning with `dc.b` followed by the DDR inside quotation marks.
- DDR entries are delimited by a colon (:). Separate parameters in each entry are also delimited by colons.
- Parameters that contain quotes (string parameters) must include a back slash (\) preceding each quotation mark such as in the CP definition “PPC603” in the example below:

  ```
  dc.b    "0:sys:CP=\"PPC603\":",13
  ```

- Each entry ends with the characters `,13` to denote an end of record.
- Though it is possible to have several devices of the same type in the system, each device must have a unique name. Names must be less than 80 characters long.
- If the parameter is numeric, it consists of a two character mnemonic part, pound sign (#), and the numeric part. The numeric part consists of one or more decimal values separated by commas. For example, a GR#512,128 in the system device DDR (device 0) means that the system has a 512K bank of graphics accessible memory as color 128 (0x80). Optionally, this is followed by a comma character and another numeric parameter. The value comprises a variable length string of characters in the range 0x30 through 0x39.
- If the parameter is a string, it consists of a two character mnemonic part, an equal sign (=), and a string. OS="OS9000" indicates that the operating system is OS-9000. Within CDB source files strings are enclosed in quotation marks. Quotation marks in parameters are preceded by a back slash (\) to differentiate them from the closing quotation marks that follow each DDR entry.

- If the parameter is boolean (yes/no type), it is represented by two characters that indicate the particular system capability. For example, HD in the data channel DDR (device 114) stands for *hardware direct*. If the parameter is present, it indicates yes, there is a hardware direct connection. If the parameter is absent it indicates no, there is no hardware direct connection.

Device Types, Device Names, Device Parameters

The ***Maui Programing Reference*** manual lists all the valid device types, device names, and device parameters. Use these descriptions as your reference for device description records when you build or modify a CDB. Device DDRs may appear in any order in the CDB, and parameters within a DDR may appear in any order. If a device type and name in a DDR does not include a specific parameter, the default values are assumed. If the device type and name has no default value listed for a parameter, you must supply the parameter value.

Device Types and Names

Device types are a numeric assignment, for example, Device 3 is always a graphics device. A CDB may list more than one device of a given type, but each device must have a unique name. Names are arbitrary, but must be less than 80 characters long.

The System Description, device 0, is the only required DDR. It should appear at least once in one of the CDB data modules on the system. The format of the device name of the System Description is:

`0:sys:parameters:`

As this is not a physical device, the name is not preceded with a slash. On all other DDR entries, the device names are preceded with a slash as in the following example:

`3:/name:parameters:`

When a device requires a protocol module, the name also includes the name of the protocol module directly following the name and preceded with a slash. An example of a device requiring a protocol module is a PS/2 mouse. The DDR entry for a mouse is similar to the following:

```
5:/m0/mp_bsptr:TY="ptr":
```

Example of the Source File

- cdb.a

```
  psect cdb,(5<<8)+1,$8000,200,0,entry

  org 0

entry:

  dc.b "0:sys:CP=\"PPC603\":OS=\"OS9000\":RV=\"2.0\":DV=\"2.1\":SR#12288,1:",13
  dc.b "2:/snd:",13
  dc.b "3:/gfx:AI=\"MAUI\":GR#2048,128:",13
  dc.b "5:/kx0/mp_xtkbd:TY=\"key\":",13
  dc.b "5:/m0/mp_bsptr:TY=\"ptr\":",13
  dc.b "9:/pipe:",13
  dc.b "20:/term:",13
  dc.b "20:/t1:",13
  dc.b "1000:/win:",13
  dc.b 0

  ends
```

Example of the Makefile

- makefile

```
# Makefile
#*****************************************************************************
#* This makefile builds a MAUI CDB module
#*****************************************************************************
#* Copyright 1995 by Microware Systems Corporation                           **
#* Copyright 2001 by RadiSys Corporation                                     **
#* Reproduced Under License                                                  **
#*                                                                           **
#* This source code is the proprietary confidential property of Microware    **
#* Systems Corporation, and is provided to licensee solely for documentation**
#* and educational purposes. Reproduction,publication, or distribution in    **
#* any form to any party other than the licensee is strictly prohibited.     **
#*****************************************************************************

PORT    =   ../..
TRGTS       = cdb

include $(PORT)/../make.com

ODIR    = $(PORT)/CMDS/BOOTOBJS/MAUI
SDIR    = .
OPTS    = -to=$(OS) -tp=$(CPU) -k

$(TRGTS): DIRS $(ODIR)/$(TRGTS)
    $(COMMENT)

$(ODIR)/cdb : $(SDIR)/cdb.a
    $(CODO) $@
    $(CC) $(SDIR)/$*.a $(OPTS) -fd=$@
    $(FIXMOD0) $@

DIRS: .
    $(MAKDIR) $(ODIR)

_clean _purge: .
    $(CODO) $(ODIR)/cdb
    -$(DEL) $(ODIR)/cdb
```

How to Modify the CDB

1. Create a directory *YOURPORT* for your ported files in:
   MWOS/*OS*/*CPU*/PORTS
2. Define and create a *CDB* directory. This directory is referred to in this chapter as *CDB* and assumes the pathname:
   MWOS/*OS*/*CPU*/PORTS/*YOURPORT*/MAUI/*CDB*
3. Using a text editor, modify the device list in the `cdb.a` file to reflect your configuration. Check that your CDB module follows these important rules:
   - Do not modify anything except the `dc.b` lines.
   - Do not modify the last `dc.b 0` line. This is the end-of-file marker.
   - At least one CDB module must include a system description DDR.
   - Your CDB may have more than one device of a given type.
   - Each device in the CDB should have a unique name.
   - Make sure every DDR line ends with a `,13`. This is the end-of-record marker.
   - Make sure you place a back slash (\) before each quote mark that appears inside the DDR. This does not apply to the quote marks that begin and end the DDR string.
   - Make sure the last lines of your file are:
     ```
     dc.b 0
     ends
     ```
4. Save the modified file in directory *CDB*.

How to Build the CDB

1. Change directories to *CDB* directory

   ```
   cd CDB
   ```

2. To make the new CDB module, type:

   ```
   os9make
   ```

How to Test the New CDB

Load the produced CDB data module(s) into your target's memory, use the MAUI CDB API calls to verify that all the relevant data can be retrieved from your CDB(s).

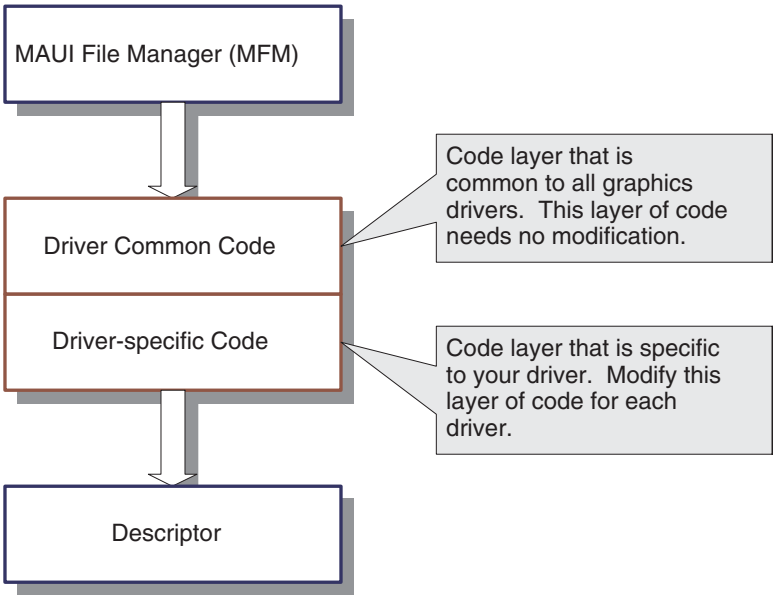
Chapter 2: Graphics Driver Interface

MAUI graphics drivers insulate applications from hardware differences in target systems. This chapter explains the graphics device capabilities; the relationship between the file manager, graphics driver, and descriptors; and how to build, modify, and verify your drivers.

Overview of Graphics Driver Interface

MAUI graphics drivers interface between the graphics device and the MAUI file manager. The graphics driver contains all device-specific code so that MAUI applications and the MAUI APIs are insulated from hardware differences in the target system. The following figure shows the relationship between the MAUI file manager (MFM), graphics driver, and descriptor:

Figure 2-1 MFM–Driver–Descriptor Relationship



The graphics device driver consists of a common code layer and a device-specific code layer. All graphics drivers share the same set of common code, which provides functions and definitions needed by all drivers. Some of the common code is conditionally compiled for individual systems. The device-specific code handles all the functions and definitions unique to each device. When porting a graphics driver, modify the device-specific code in the sample driver to reflect the graphics device in your system.

The device descriptor is the handle used by applications to reference a device. The descriptor indicates the file manager, driver, and the driver initialization data required to access the device.

Graphics RAM

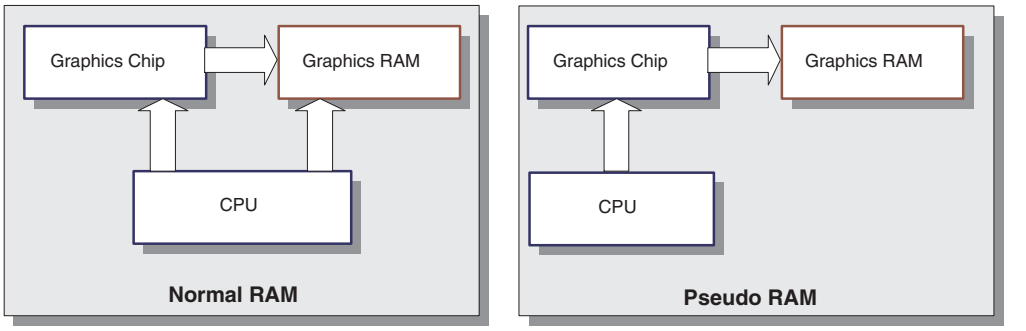
MAUI classifies graphics RAM as one of two types:

- Normal RAM (accessible by the CPU)
- Pseudo RAM (not accessible by the CPU)

For normal RAM, the CPU has direct access to the graphics RAM. For pseudo RAM, the CPU must access the graphics chip which, in turn, accesses the graphics RAM. This is determined by the chip or board manufacturer.

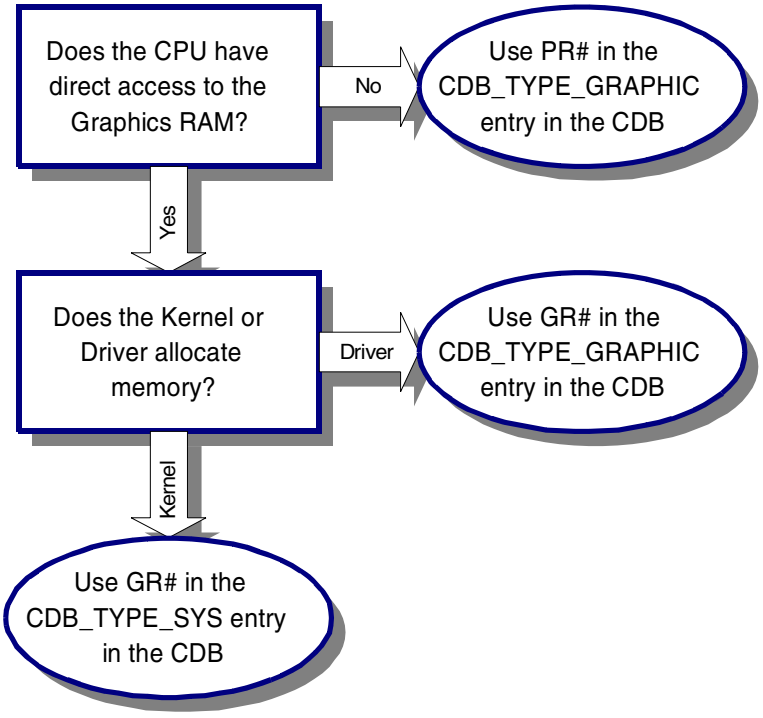
While normal RAM allows easier and normally faster access to the graphics RAM by the application, it draws the penalty of consuming CPU time for display memory updates. The following figure illustrates the two types of graphics RAM:

Figure 2-2 Normal RAM and Pseudo RAM



If your device uses normal RAM, the kernel or the driver may manage the memory. If the kernel manages the memory, it must be set up by the `sys_init` module or appear in the memory list in the `init` module (See ***OS-9 Technical Reference*** or ***OS-9 for 68K Technical Reference***).The configuration of RAM is described in the CDB. Use the following decision tree to help you complete your CDB.

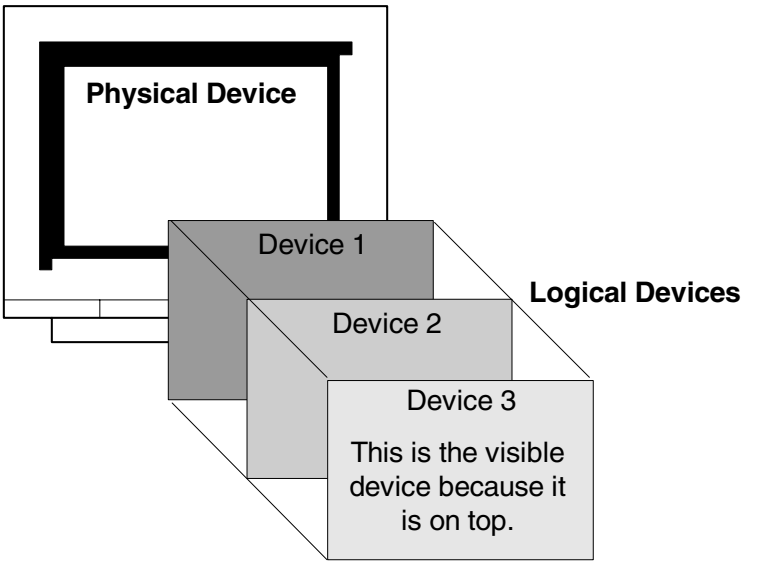
Figure 2-3 RAM Allocation



Graphics Device

When a process opens a path to a physical device, a logical device is created for that process. There may be several paths open to the physical device at any one time, so multiple logical devices may exist. The logical device that is at the top of the logical device stack is the visible device as shown in the following figure:

Figure 2-4 Physical and Logical Graphics Devices



The physical device is simply the physical hardware that displays graphics such as a television, monitor, or LCD.

As each path is opened to the device with `gfx_open_dev()`, each path is given a logical device ID. Only the top-most logical device is visible.

A process may clone a logical device opened by another process by calling the function `gfx_clone_dev()`. When this function is called, the logical device is shared by both processes.

Device Capabilities

One important function of your device driver is specifying the capabilities of the device. Graphics device capabilities are defined in a set of data structures within the `global.h` file. Written specifications for the sample drivers included with MAUI are located in the same directory as the source files.

A hardware specification is particularly valuable when writing your `global.h` file. Your specification may be different, but similar to the specifications of the sample drivers. Following are examples of the data structures that define device capabilities.

GFX_DEV_CAP Device Capabilities

This data structure defines the set of display capabilities. The structure is defined as follows:

```
typedef struct _GFX_DEV_CAP {
    char *hw_type;                  /* hardware type */
    char *hw_subtype;               /* hardware sub-type */
    BOOLEAN sup_vpmix;              /* supports viewport mixing */
    BOOLEAN sup_extvid;             /* supports external video */
    BOOLEAN sup_bkcolor;            /* supports background color */
    BOOLEAN sup_vptrans;            /* supports viewport transparency */
    BOOLEAN sup_vpinten;            /* supports viewport intensity */
    BOOLEAN sup_sync;               /* supports retrace synchronization */
    u_int8 num_res;                 /* number of display resolutions */
    GFX_DEV_RES *res_info;          /* array of display resolutions */
    u_int8 dac_depth;               /* depth of DAC in bits */
    u_int8 num_cm;                  /* number of coding methods */
    GFX_DEV_CM *cm_info;            /* array of coding methods */
    BOOLEAN sup_decode;             /* supports video decoding */
} GFX_DEV_CAP;
```

Following is an example of a `GFX_DEV_CAP` data structure of a driver, which supports standard VGA graphics chip set mode 12H and “X”-mode:

```
GFX_DEV_CAP gdv_dev_cap = {
    "VGA",              /* hardware type */
    NULL,               /* hardware sub-type name (filled in later) */
    FALSE,              /* supports viewport mixing */
    FALSE,              /* supports external video */
    FALSE,              /* supports backdrop color */
    FALSE,              /* supports viewport transparency */
    FALSE,              /* supports vport intensity */
    FALSE,              /* supports retrace synchronization */
    sizeof(gdv_res_info)/sizeof(*gdv_res_info), /* Num res_info */
    gdv_res_info,       /* pointer to display resolution information */
    6,                  /* depth of DAC in bits */
    sizeof(gdv_cm_info)/sizeof(*gdv_cm_info), /* Num cm_info */
    gdv_cm_info,        /* pointer to coding method information */
    FALSE               /* supports video decoding into a drawmap */
};
```

This structure references two other structures `GFX_DEV_RES gdv_res_info` and `GF_DEV_CM gdv_cm_info` described below. Both of these are pointers to an array of structures of their respective types.

GFX_DEV_RES Device Resolution

The `GFX_DEV_RES` structure provides a description of display setting/resolutions supported by the driver. The driver provides an array of `GFX_DEV_RES` structures describing each supported display setting. The first resolution defined in this data structure is the default resolution. This data is structured as follows:

```
typedef struct _GFX_DEV_RES {
    GFX_DIMEN disp_width;     /* width */
    GFX_DIMEN disp_height;    /* height */
    u_int16 refresh_rate;     /* refresh rate */
    GFX_INTL_MODE intl_mode;  /* interlace mode */
    u_int16 aspect_x;         /* x aspect ratio */
    u_int16 aspect_y;         /* y aspect ratio */
} GFX_DEV_RES;
```

Following is an example of a `GFX_DEV_RES` data structure that defines two display resolutions:

```
GFX_DEV_RES gdv_res_info[] = {
    {640, 480, 60, GFX_INTL_OFF, 1, 1}, /* default mode 12H */
    {360, 480, 60, GFX_INTL_OFF, 1, 1}  /* X-mode */
};
```

GFX_DEV_CM Coding Methods

The GFX_DEV_CM data structure describes a coding method. The driver provides an array of GFX_DEV_CM data structures and contains an entry for each coding method supported by the graphics device. Coding methods are specific formats for graphic data. The first coding method entry is the default coding method, followed by additional supported coding methods. This data is structured as follows:

```
typedef struct _GFX_DEV_CM {
    GFX_CM coding_method;          /* coding method */
    BOOLEAN clut_based;            /* TRUE if CLUT-based */
    u_int16 dm2dp_xmul;            /* multiplier to convert X coordinate */
    u_int16 dm2dp_ymul;            /* multiplier to convert Y coordinate */
    u_int8 num_color_types;        /* number of color types */
    GFX_COLOR_TYPE *color types;   /* array of color types*/
} GFX_DEV_CM;
```

A CLUT-based coding method uses an index of colors called a Color Look-Up Table (CLUT).

Multipliers are used to convert values in the drawmap coordinate system to equivalent values in the display coordinate system. When the display resolution is different from the drawmap resolution the dm2dp_xmul and dm2dp_ymul values are other than 1. Following is an example of a GFX_DEV_CM data structure that defines two supported coding methods, 4bpp (16 colors) coding method for VGA mode 12H and 8bpp (256 colors) for ‘X’-mode:

```
GFX_DEV_CM gdv_cm_info[] = {
    {GFX_CM_4BIT, TRUE, 1, 1, GDV_NUMCOLORS, gdv_valid_colors},
    {GFX_CM_8BIT, TRUE, 1, 1, GDV_NUMCOLORS, gdv_valid_colors},
};
```

GFX_DEV_CAPEXTEN Extended Device Capabilities

As of MAUI 3.1 an extended or secondary device capabilities structure should be supported by all drivers. This data structure provides additional information about the display capabilities of the device. The structure is defined as follows:

```
typedef struct _GFX_DEV_CAPEXTEN {
    u_int16 version;            /* == sizeof(GFX_DEV_CAPEXTEN) used to
                                   determine revision of struct */
    u_int16 num_modes;          /* Number of modes in mode_info */
    GFX_DEV_MODES *mode_info;   /* Array of supported modes (maybe subset) */
    GFX_VPC vp_complexity;      /* Hint regarding supported viewport
                                   complexity */
    GFX_VPDMC vpdm_complexity;  /* Hint regarding supported drawmap
                                   viewport complexity */
} GFX_DEV_CAPEXTEN;
```

Following is an example of a GFX_DEV_CAPEXTEN data structure for the driver described above:

```
const GFX_DEV_CAPEXTEN gdv_dev_capexten = {
    sizeof(GFX_DEV_CAPEXTEN),   /* Size/Version of structure, NEVER CHANGE */
    sizeof(gdv_dev_modes)/sizeof(*gdv_dev_modes),/* Number of modes */
    gdv_dev_modes,              /* Mode info */
    GFX_VPC_ONE_EXACT,          /* Supports only one viewport the
                                   exact size of the display */
    GFX_VPDMC_LARGER            /* Can display sub-drawmaps */
};
```

This structure references the structure GFX_DEV_MODES gdv_dev_modes described below. This is a pointer to an array of device modes structures which indicate compatible device resolution and coding method combinations supported by the graphics device.

GFX_DEV_MODES Device Modes

The GFX_DEV_MODES data structure is referenced by GFX_DEV_CAPEXTEN to indicate compatible device resolution and coding method combinations (modes) supported by the graphics device. This is ideally all of the supported modes, but can be a subset if there are too many. This data is structured as follows:

```
typedef struct _GFX_DEV_MODES {
    u_int16 res_idx;            /* res_info index index */
    u_int16 cm_idx;             /* cm_info index */
    char *desc;                 /* Description of mode */
} GFX_DEV_MODES;
```

Following is an example of a GFX_DEV_MODES data structure that defines a set of coding method and display resolutions pairs:

```
GFX_DEV_MODES gdv_dev_modes[] = {
  {0,   0, "640x480x4"},
  {0,   1, "640x480x8"},
  {1,   0, "360x480x4"},
  {1,   1, "360x480x8"}
};
```

Note that the `res_idx` and `cm_idx` fields are indexes into `GFX_DEV_CAP`'s `GFX_DEV_RES gdv_res_info` and `GF_DEV_CM gdv_cm_info` arrays, not pointers.

Note

See the MAUI Programming Reference manual for complete description of each of the data structures.

Compile State for Graphics Drivers

This section provides information regarding the compile state for MAUI graphics drivers, including the names of the driver files and the functions implemented within them.

IOBLT and HWBLT Drivers

Table 2-1 IOBLT and HWBLT

Compiled to User State	Compiled to System State (GDC_FE_SYSATE)
gdv_blt.c	**gdv_fe.c**
_gdv_blt_drwmix	_os_ss_blt_drwmix
_gdv_blt_cpymix	_os_ss_blt_cpymix
_gdv_blt_pix	_os_ss_blt_pix
_gdv_blt_src	_os_ss_blt_src
_gdv_blt_dst	_os_ss_blt_dst
_gdv_blt_ofs	_os_ss_blt_ofs
_gdv_blt_mask	_os_ss_blt_mask
_gdv_blt_trans	_os_ss_blt_trans
_gdv_blt_exptbl	_os_ss_blt_exptbl

IOBLT Driver

Table 2-2 Specific to IOBLT

Compiled to User State	Compiled to System State (GDC_FE_SYSATE
gdv_copy.c	**gdv_fe.c**
_gdv_ioblt_copyblk	_os_ss_ioblt_copyblk
_gdv_ioblt_copynblk	_os_ss_ioblt_copynblk
gvd_draw.c	_os_ss_ioblt_copynblk
_gdv_ioblt_drawblk	_os_ss_ioblt_drawblk
_gdv_ioblt_drawhline	_os_ss_ioblt_drawhline
_gdv_ioblt_drawvline	_os_ss_ioblt_drawvline
_gdv_ioblt_drawpixel	_os_ss_ioblt_drawpixel
gdv_expd.c	_os_ss_ioblt_expdblk
_gdv_ioblt_expdblk	_os_ss_ioblt_expdnblk
_gdv_ioblt_expdnblk	

HWBLT Driver

Table 2-3 Specific to HWBLT

Compiled to User State	Compiled to System State (GDC_FE_SYSATE)
hwblt.c _gdv_hwbt_drawblk	**gdv_fe.c** _os_ss_hwblt_drawblk

Driver Code

MAUI graphics drivers consist of two types of code: common code that is already written for your driver, and device-specific code that you write. The common code makes up a large portion of the graphics driver and does not have to be modified. When porting a graphics driver, you modify the device-specific code in the sample drivers to reflect the capabilities of the graphics device in your system and implement the functionality it can support. The device-specific code consists of a number of files, of which some are required and others are optional, depending on your system. The following files are required in every graphics driver:

- `config.h` contains the definitions that control the configuration of the driver including the names of functions defined by the device-specific code.
- `drvr.tpl` `os9make` “include” file.
- `global.h` contains the global definitions for the driver including device capabilities and prototypes.
- `hardware.c` defines functions that deal with the hardware device setup routines such as init and terminate.
- `hardware.h` contains hardware-specific definitions.
- `static.c` initializes and terminates static storage areas used by the driver.
- `static.h` contains the definitions for static storage areas available to the driver.
- `updtdpy.c` updates the display with queued changes and optionally synchronizes changes with vertical retrace.
- `vpdmap.c` sets a drawmap area to be displayed in the viewport.
- `vpdmpos.c` sets a position of the drawmap in the viewport.
- `vppos.c` sets a position of the specified viewport.

- `vprestak.c` restacks a viewport within the viewport stack.
- `vpsize.c` sets a size of the specified viewport.
- `vpstate.c` sets a state of the specified viewport to active or not active.

Several other files may be included in the device-specific code at your option. These files include:

- `dvbkcol.c` sets a background color for the display.
- `dvextvid.c` sets an external video on and off.
- `dvtran.c` sets transparent color.
- `dvvpmix.c` sets viewport mixing on and off.
- `ioblt.c` enables driver-supported bit-BLT (using I/O registers).
- `hwblt.c` enables driver-supported bit-BLT (using H/W acceleration).
- `hwcur.c` enables a H/W cursor.
- `irq.c` defines interrupt service functions.
- `vpintens.c` sets an intensity of the specified viewport.

When modifying the driver code, you should organize your work to modify the files in this order:

1. Modify header files.
2. Modify required display functions.
3. Modify required viewport functions.
4. Modify optional functions.

Device-Specific Code

Sample driver files are located in the directory:

`MWOS/SRC/DPIO/MFM/DRVR/GX_SAMP`

You can use these files as templates for building your own device-specific code.

Where the Files are Located

MAUI is delivered with one directory of sample files and several complete drivers. The complete drivers are example drivers that you can modify to make your driver. The sample files contain instructions for building your own .h and .c files.

- MAUI standard header files are located in
  `MWOS/SRC/DEFS/MAUI`

WARNING

These header files should never be modified by the user

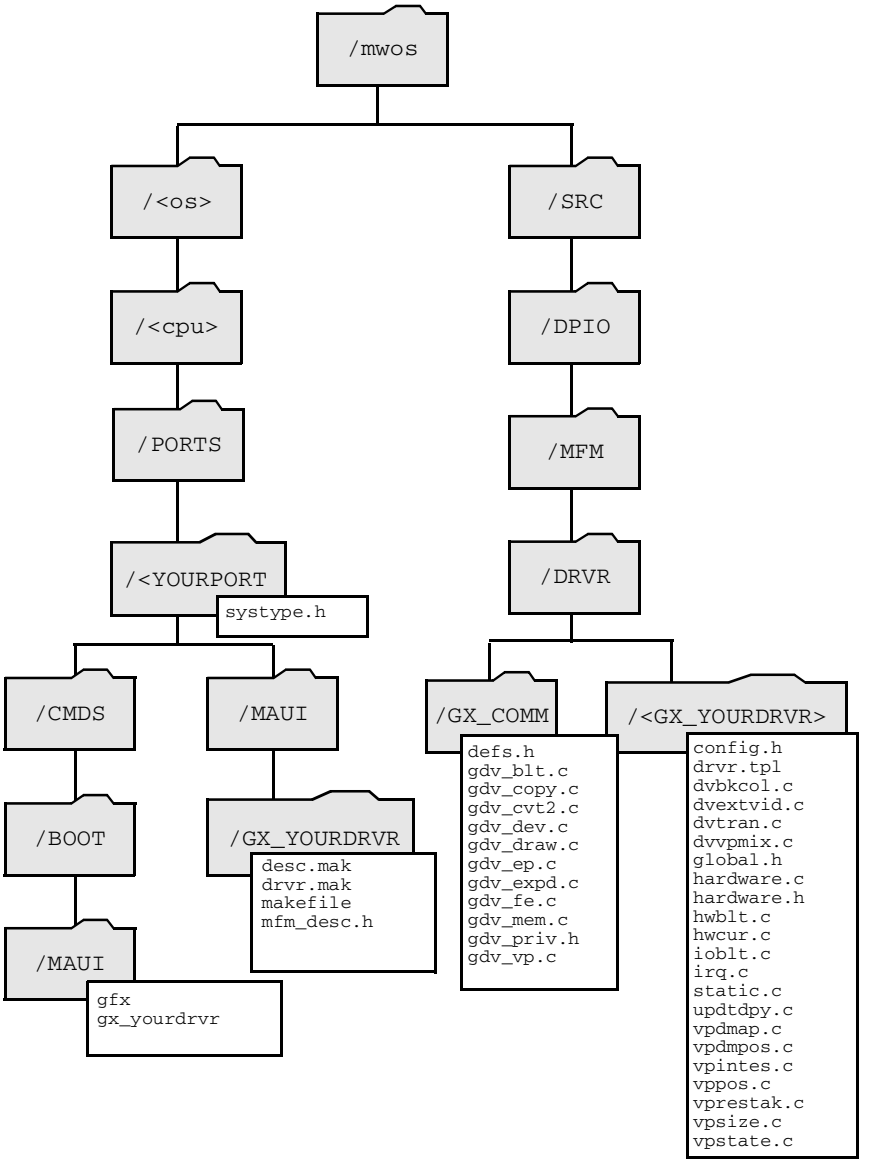
- MAUI common driver code is located in
  `MWOS/SRC/DPIO/MFM/DRVR/GX_COMM`
  This directory is referred to as *common* throughout this chapter. Normally you should not need to modify files in this directory. If your implementation does have special requirements that necessitates modifying the common code, make a copy of the relevant file(s) to your driver specific directory and make your modifications there.
- The sample driver template files are located in:
  `MWOS/SRC/DPIO/MFM/DRVR/GX_SAMP`
- Depending on the software package purchased, other complete driver sources are found under:
  `MWOS/SRC/DPIO/MFM/DRVR/GX_*`

How to Port Your Graphics Driver

Create the directory structure for your port

Before beginning to port your graphics driver, you must create a directory structure to store your new files. That directory structure is shown on the next page in the figure **Directory Structure for Your Graphics Driver Port**.

Figure 2-5 Directory Structure for Your Graphics Driver Port



Step 1. Define and create a source directory. This directory is referred to in this chapter as *SOURCE* and assumes the pathname:
MWOS/SRC/DPIO/MFM/DRVR/*GX_YOURDRVR*
We recommend that the directory name start with "GX_" followed by an uppercase descriptive name for your driver.

Step 2. Copy all of the files from:
MWOS/SRC/DPIO/MFM/DRVR/GX_SAMP
into your new *SOURCE* directory. Verify that the following files are now in your *SOURCE* directory:

config.h	drvr.tpl
dvbkcol.c	dvextvid.c
dvtran.c	dvvpmix.c
global.h	hardware.c
hardware.h	hwblt.c
hwcur.c	ioblt.c
irq.c	static.c
static.h	updtdpy.c
vpdmap.c	vpdmpos.c
vpintens.c	vppos.c
vprestak.c	vpsize.c
vpstate.c	

Step 3. Define and create a ports directory. This directory is referred to in this chapter as *YOURPORT* and assumes the pathname:
MWOS/*OS*/*CPU*/PORTS/*YOURPORT*

Step 4. Define and create a make directory. This directory is referred to in this chapter as *MAKE* and assumes the pathname:
MWOS/*OS*/*CPU*/PORTS/*YOURPORT*/MAUI/*GX_YOURDRVR*

Step 5. Copy or create the following files in your *MAKE* directory:

desc.mak	drvr.mak
makefile	mfm_desc.h

Here are examples of these files:

- `makefile` Make both MAUI graphics descriptor and driver

```
# Makefile
################################################################################
#                                                                              #
#   Copyright 1996 by Microware Systems Corporation                            #
#   Copyright 2001 by RadiSys Corporation
#                              All Rights Reserved                             #
#                            Reproduced Under License                          #
#                                                                              #
#  This software is confidential property of Microware Systems Corporation,  #
#  and is provided under license for internal development purposes only.     #
#  Reproduction, publication, distribution, or creation of derivative works #
#  in any form to any party other than the licensee is strictly prohibited,  #
#  unless expressly authorized in writing by Microware Systems Corporation.  #
#                                                                              #
################################################################################

#
# Conditionally call driver makefile automatically for BSP vs DEVKITS
#
if exists(drvr.mak)
DRVRMAKE = drvr.mak
else
DRVRMAKE =
endif

PORT    =   ../..
TRGTS   =   desc.mak $(DRVRMAKE)

include $(PORT)/../makesub.com

$(TRGTS):
    -$(MAKESUB) -f=$@

################################################################################
```

- `desc.mak` Make the MAUI graphics descriptor

```
# Makefile
#*******************************************************************************
#* Makefile for MAUI Graphics Descriptors
#*******************************************************************************
#* Copyright 1996 by Microware Systems Corporation                             **
#* Copyright 2001 by RadiSys Corporation                                       **
#* Reproduced Under License                                                    **
#*                                                                             **
#* This source code is the proprietary confidential property of                **
#* Microware Systems Corporation, and is provided to licensee                  **
#* solely for documentation and educational purposes. Reproduction,            **
#* publication, or distribution in any form to any party other than            **
#* the licensee is strictly prohibited.                                        **
#*******************************************************************************
#### Add New Descriptor Names Here ############################################
#                                                                             #
TRGTS       =   gfx
DRVR        =   GX_YOURDRVR
USER_OPTS   =   -oln=gfx
#                                                                             #
###############################################################################

PORT        =   ../..
MAKENAME    =   desc.mak
include $(PORT)/../make.com

RDIR        =   RELS/DESC
ODIR        =   $(PORT)/CMDS/BOOTOBJS/MAUI
SDIR        =   $(MWOS)/SRC/DPIO/MFM/DESC
DESCDIR     =   .

include $(SDIR)/desc.tpl

_purge _clean:
for TMP in $(TRGTS)
    -$(CODO) $(ODIR)/$(TMP)
    -$(DEL) $(ODIR)/$(TMP)
endfor

###############################################################################
```

- `drvr.mak` Make the MAUI graphics driver

```
# Makefile
#*****************************************************************************
#* Makefile for MAUI Graphics Driver                                         **
#*****************************************************************************
#* Copyright 1996 by Microware Systems Corporation                             **
#* Copyright 2001 by RadiSys Corporation                                       **
#* Reproduced Under License                                                    **
#*                                                                             **
#* This source code is the proprietary confidential property of                **
#* Microware Systems Corporation, and is provided to licensee                  **
#* solely for documentation and educational purposes. Reproduction,            **
#* publication, or distribution in any form to any party other than            **
#* the licensee is strictly prohibited.                                        **
#*****************************************************************************


#### Put Driver Names and Options Here ##########################################
#                                                                               #
TRGTS   =   gx_yourdrvr
DRVR    =   GX_YOURDRVR

# Definitions, specified by the driver writer, and seen
# in driver-specific portion of the driver source code
# Any defines, useful as a compile-time option for
# controlling the driver configuration.


# Example
#       -d=PWR_AWARE    : register driver with power management
#                         subsystem (if applicable)
USR_DEFINES = -dPWR_AWARE -dENABLE_ATTRIBUTE

# To turn on the debug option, use: -g
DEBUG   =
#DEBUG  = -g

#                                                                               #
################################################################################

PORT    =   ../..
MAKENAME = drvr.mak
include $(PORT)/../make.com
ODIR    =   $(PORT)/CMDS/BOOTOBJS/MAUI
RDIR    =   RELS/DRVR
IDIR    =   $(RDIR)/$(HOSTTYPE)
DESCDIR =   .

# See driver template (drvr.tpl) to understand, how the
# following defines can be used to specify compilation rules.
# Usually, it is helpful to set them, if driver has additional
# source files outside GX_COMMON and GX_YOURDRVR directories.
# Example
```

```
#       ADDITIONAL_IFILES = $(IDIR)/yourfile.i
ADDITIONAL_IFILES   =

# List additional libraries, required for the driver
# Example
#       ADDITIONAL_LIBS = -l=$(MWOS_LIBDIR)/yourlib.l
ADDITIONAL_LIBS =

# Include driver template
include $(MWOS)/SRC/DPIO/MFM/DRVR/$(DRVR)/drvr.tpl

# Additional build rules (required, if ADDITIONAL_IFILES
# flag is set
# Example
#       $(IDIR)/yourfile.i: yourfile.c $(DEPENDFILES)
#                   $(COMPILE) yourfile.c

# you can use the following to change the revision # of the driver
# this is added to the end of the link rule in drvr.tpl
$(ODIR)/$(TRGTS): $(RFILES) $(MAKENAME)
    fixmod -ugua=a001 $@

#                                                                              #
################################################################################
```

- `mfm_desc.h` The MAUI graphics driver descriptor header (modify to reflect your descriptor settings)

```
/*****************************************************************************
**
* FILENAME : mfm_desc.h
*
* DESCRIPTION :
*
*   This file contains definitions for the MAUI device descriptors.
*
* COPYRIGHT:
*
*   This source code is the proprietary confidential property of Microware
*   Systems Corporation, and is provided to licensee solely for documentation
*   and educational purposes. Reproduction, publication, or distribution in
*   form to any party other than the licensee is strictly prohibited.
*
#ifndef _MFM_DESC_H
#define _MFM_DESC_H

#include "../../systype.h"

/*****************************************************************************/
/* Generic VGA Graphic Descriptor                                            */
/*****************************************************************************/
#if defined(MFM_DESC_GFX) || defined(gfx)

/*************************************************************/
/*  Descriptor's common portion (has to be present and set)    */
/*************************************************************/
#define MEM_COLOR 0x80
#define SHARE      TRUE           /* allow multiple open paths to */
                                  /* the graphics device */
#define LUN        0              /* Logical unit number */
#define PORTADDR   0x00000000     /* Base address of hardware */
#define MODE   S_IREAD | S_IWRITE /* Device mode capabilities */
#define DRV_NAME   "gx_yourdrvr"  /* Driver name */

/*
 * Base IRQ vector for OS-9000 is equal to 0x40. Therefore
 * the resulting IRQ vector number to be set is 0x40 plus
 * physical vector number.
 *
 * Note: if the driver do not support interrupts, both
 * INTERRUPT_ENABLED and GDV_IRQ_EVNAME have to be set to
 * zero (NULL).
 */
#define INTERRUPT_ENABLED 0   /* 0 - vertical interrupts disabled */
                                 /* 1 - vertical interrupts enabled */

#define GDV_IRQ_NUM       0x40    /* IRQ vector number */
#define GDV_IRQ_PRIORITY  0       /* IRQ priority */
```

```
#define GDV_IRQ_EVNAME     NULL    /* IRQ event name (NULL if
none) */

/*************************************************************/
/*  Descriptor's specific portion (optional and driver-dependent) */
/*************************************************************/
/* Definitions for the driver static storage (static.h) */
/* Add your own defines here */
#define MEM_BASE_ADDRESS   0xA0000
#define   MEM_SIZE             0x10000

/*
 * for following GDV_HW_SUBTYPE definitions, see MFM/DRVR/XXX/static.h's
 * typedef enum { ... } HW_SUBTYPE;
 */
#define GDV_HW_SUBTYPE  VGA    /* Hardware sub-type */
#define GDV_HW_SUBNAME "VGA_GENERIC" /* Hardware sub-type name */

#endif /* MFM_DESC_GFX */

#endif /* _MFM_DESC_H_ */

/*************************************************************/
```

Step 6. Create the directory *YOURPORT*/CMDS/BOOTOBJS/MAUI. During the make process two object files are created and stored in MAUI:

- gfx — Descriptor object
- gx_*yourdrvr* — Driver object
  This is typically a lower case version of the directory name in step 1.

Step 7. Verify your directory structure contains the correct files as shown in the figure **Directory Structure for Your Graphics Driver Port** on page 35.

Common Source Files

The following files are located in the `GX_COMM` directory:

`defs.h`	Global definitions file
`gdv_blt.c*`	I/O and H/W Bit-BLT support common code
`gdv_copy.c*`	I/O Bit-BLT support for copy operations
`gdv_cur.c*`	H/W cursor support common code
`gdv_cvt2.c`	Color conversion functions
`gdv_dev.c`	Common graphics device functions
`gdv_draw.c*`	I/O Bit-BLT support for draw operations
`gdv_ep.c`	Entry point functions
`gdv_expd.c*`	I/O Bit-BLT support for expand operations
`gdv_fe.c*`	Fast-entry-point functions
`gdv_main.c`	Main function for the driver
`gdv_mem.c*`	Graphics memory management functions
`gdv_priv.h`	Definitions private to the common code
`gdv_vp.c`	Viewport functions

* Optional files. Do not include in `drvr.tpl` if your driver does not support these functions.

Device-specific Files

The following files are located in the *SOURCE* directory:

config.h	Configures the capabilities of the driver and inclusion/exclusion of common code
drvr.tpl	os9make "include" file.
dvbkcol.c*	Backdrop color function
dvextvid.c*	External video function
dvtran.c*	Transparent color function
dvvpmix.c*	Viewport mixing function
global.h	Global definitions
hardware.c	Hardware function
hardware.h	Definitions for hardware functions
hwblt.c*	H/W Bit-BLT support in driver
hwcur.c*	H/W cursor support in driver
ioblt.c*	I/O Bit-BLT support in driver
irq.c*	Interrupt service functions
static.c	Code to initialize/terminate static storage areas
static.h	Definitions for static storage areas
updtdpy.c	Update display function
vpdmap.c	Viewport drawmap function
vpdmpos.c	Viewport drawmap position function
vpintens.c*	Viewport intensity function
vppos.c	Viewport position function
vprestak.c	Viewport restack function
vpstate.c	Viewport state (active/inactive) function
vpsize.c	Viewport size function

* Optional files. Delete these files if not supported by your driver.

Modify SOURCE Files

Step 1. Construct your `drvr.tpl` file to include or not include the files marked * depending on whether your driver supports those functions.

Step 2. Update your `drvr.tpl` to reflect the changes made in step 1.

Modify the config.h file to reflect your system.

Step 1. Define `GDV_INCLUDE_MEM` only if your graphics driver must handle memory management. This must be done if the graphics memory is pseudo memory or it is not accessible by the CPU at the time the kernel does a memory search. It also has to be done, if graphics memory allocation should be on the specific boundary. Review **Graphics RAM** for a discussion of memory considerations.

If the CPU can access the graphics memory, it is best to let the kernel handle memory management. See the ***OS-9 Porting Guide*** for more information about kernel memory management.

Step 2. Set `GDV_MEM_PREFIX` to the required size in bytes if the hardware requires a header at the beginning of drawmap memory for display. `GDV_INCLUDE_MEM` flag should also be set in this case.

Step 3. Set `GDV_MEM_POSTFIX` to the required size in bytes if the hardware requires a trailer at the end of drawmap memory for display. `GDV_INCLUDE_MEM` flag should also be set in this case.

Step 4. Define the `GDV_INCLUDE_CVT2_*` labels that match the color types required by the hardware. Delete all others.

Step 5. Define `GDV_FE_SYSSTATE` if fast entry points must execute in system state. If not, delete `GDV_FE_SYSSTATE`. This should only be defined if code in fast entry points perform privileged instructions or require system-state I/O access. Enabling this option adds the overhead of a context switch to all fast entry points.

Step 6. Define function names for GDV_INIT_HW through GDV_UPDATE_DPY. All values must be defined, although you normally use the default values. These are required functions.

Step 7. Define function names for GDV_INIT_IRQS through GDV_GET_ATTRIBUTE. Only include definitions for functions supported by your driver. These are optional functions.

Here is a brief description of each function.

- GDV_INIT_IRQS - Initialize interrupts
- GDV_TERM_IRQS - Terminate interrupts
- GDV_INIT_DVATCH - Initialize gfx device attachment
- GDV_TERM_DVATCH - Term gfx device attachment
- GDV_INIT_VPATCH - Initialize viewport attachment
- GDV_TERM_VPATCH - Terminate viewport attachment
- GDV_SET_BKCOL - Set backdrop color
- GDV_SET_EXTVID - Set external video on/off
- GDV_SET_TRANSCOL - Set transparency color
- GDV_SET_VPMIX - Set viewport mixing on/off
- GDV_SET_VPINTEN - Set viewport intensity
- GDV_SET_ATTRIBUTE - Set a device attribute
- GDV_GET_ATTRIBUTE - Get a device attribute

The prototypes for each of the functions may be found in SRC/DPIO/MFM/DRVR/GX_COMM/defs.h.

Step 8. Define GDV_MEM_TOP_BIT if user-state process can not reference memory addresses with the most significant bit set (e.g. SuperH architecture). Every time driver common code allocates memory for the structure, which will be used by the API in user-state mode, it clears the most significant bit of the resulting pointer, if GDV_MEM_TOP_BIT is set. When GDV_MEM_TOP_BIT is not defined, no modification to the structure pointers is made. The most significant bit is set back when the driver common code has to deallocate the memory back to the system.

Step 9. Define GDV_PIXMEM_BNDRY if allocations of and access to the graphics memory must be on a boundary. The value should indicate the required boundary size in bytes.

Step 10. Define GDV_INCLUDE_IOBLT if I/O Bit-BLT support is required. If GDV_INCLUDE_IOBLT is defined, you must define the remaining entries in steps 10 through 15. If not defined, delete the following up to the GDV_INCLUDE_IOBLT flag and go to Step 17.

Step 11. Define GDV_IOBLT_WORDSIZ as the size of each word in bytes. GDV_IOBLT_WORDSIZ is a restriction typically imposed by graphics hardware. It is the boundary (byte, word, longword, quadword, etc.) at which video buffer should be accessed. It is also the smallest segment of graphics RAM that must be read or written through the I/O port.

Step 12. Define GDV_IOBLT_WORDSFT as the shift value derived from the GDV_IOBLT_WORDSIZ value. For example, if GDV_IOBLT_WORDSIZ is equal to 8 (video memory has to be accessed on 8-byte boundary), then GDV_IOBLT_WORDSFT is equal to 3 (1<<3 is 8).

Step 13. Define GDV_IOBLT_LINESIZ as the maximum line size in bytes. GDV_IOBLT_LINESIZ is the size of the largest video buffer line, which IOBLT driver can store into internal data structure. This value should be large enough to hold the content of a video line of any dimention. For example, if IOBLT operations are to performed on the video drawmap with the size 640x480x16bits/pixel, then GDV_IOBLT_LINESIZ should be set to 640 * 16/8.

Step 14. Set GDV_IOBLT_GFXRAM through GDV_IOBLT_WRITE_PIX to the names of the functions provided by the device-specific code. The default values are used in most cases.

Step 15. Optionally define GDV_IOBLT_OFFSETS as the function to compute the odd and even offsets for interlace support.

Step 16. Define GDV_IOBLT_SEP_CHROMA. This enables support of separate lumina and chroma sections of the drawmap.

Step 17. Define GDV_INCLUDE_HWBLT if H/W Bit-BLT support is implemented. If GDV_INCLUDE_HWBLT is defined, you must define the remaining entries in steps 18 through 20.

Step 18. Set GDV_HWBLT_DRWMIX through GDV_HWBLT_DST to the names of the functions provided by the device-specific code. The default values are used in most cases. These functions are required by the template.

Step 19. Define GDV_HWBLT_BCATCH and GDV_HWBLT_BCATCH and in case driver-specific part of Bit-BLT attachment is specified (see also GDV_BCATCH_SPECIFICS in static.h). These functions are optional.

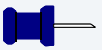
Step 20. Set GDV_HWBLT_DRAWBLK through GDV_HWBLT_GETPIXEL to the names of the functions provided by the device-specific code. The default values are used in most cases. These functions are independent and should be set only in the case when the driver wants to support any of the correspondent Bit-BLT operations in H/W acceleration mode.

Step 21. Define GDV_HW_CURSOR if H/W cursor is supported by the driver. If GDV_HW_CURSOR is defined, you must set the remaining entries in step 22. If not defined, delete the following and skip to the section " *Modify the global.h file to reflect your graphics device capabilities*".

Step 22. Set GDV_CURSOR_CREATE through GDV_CURSOR_SET_POS to the names of the functions provided by the device-specific code. The default values are used in most cases. These functions are required by the template.

Modify the global.h file to reflect your graphics device capabilities

Step 1. Modify the `gdv_dev_cap` data structure. Please note, that two of its fields depend on the number of entries in the following data structures.



Note

This and the following data structures are documented in the MAUI Programming Reference as well an example in the **Device Capabilities** section of this manual.

Step 2. Modify the `gdv_res_info` data structure.

Step 3. Modify the `gdv_cm_info` data structure.

Step 4. Modify the `gdv_dev_capexten` data structure. Please note, that one of its fields depends on the number of entries in the following data structure.

Step 5. Modify the `gdv_dev_modes` data structure.

Step 6. Prototype the functions you need for device-specific code in the `PROTOTYPE` area. This area is used to prototype functions that must be visible to multiple device-specific files.

Modify the static.h file to define your static storage areas.

Now it is time to define the device, logical unit and context specific static storage. You may not be able to completely define all the variables until you get further along with the port, make an attempt to define what you can now, and refine this file as the port proceeds. Steps 2 and 3 define the data per physical device. Step 4 defines the data per logical device. Step 5 defines the data per viewport. Step 6 defines the data per Bit-BLT context (define this structure only in case driver is going to support H/W accelerated Bit-BLT functions).

Step 1. Modify `HW_SUBTYPE` to define all the sub-types known by the driver. The descriptor determines the sub-type for a specific device. The driver uses this value to make run-time decisions based on the sub-type. This could be helpful, if the same driver has to support several subtypes of graphics controllers, that have a minor differences among each other (e.q. SVGA cards by the same manufacturer).

Step 2. Modify `GDV_LU_SPECIFICS` with the variable names needed by the driver. This file is setup with the values in `GDV_LU_SPECIFICS_INIT` when the driver is initialized. Also the fields of this structure can be used as a global storage containing the current state of the physical device. This structure should include, but is not limited to the following:

- Address of each bank of graphics memory.
- Address of groups or individual I/O registers.
- Place holders for shadow contents of I/O registers.

Step 3. Modify `GDV_LU_SPECIFICS_INIT` to include the values from the descriptor to compute the initializers. Minimize the number of definitions required in the descriptor to reduce its size.

Step 4. Modify `GDV_DVATCH_SPECIFICS` to specify the device-specific members of the graphics device structure. This area is allocated and initialized when the graphics device is opened. This data structure should contain enough data to fully define the current state of the graphics device. The following considerations are important when modifying this file:

- The normal `GFX_DEV` and `GFX_DEV_SHARED` structures define the queued-up state of the device, not the current visible state. This allows the application to make any changes to the logical device, which is not on top and not visible, without affecting the physical device. All changes to the hardware will happen only after the device, which is on top, becomes visible and the “update display” function is called.

- The driver needs information about the current visible state whenever this logical device is put on top to set physical device operating mode properly.
- Each time you open the graphics device, a new path to the device is established and a new logical device structure is created. Each logical device maintains its own state so when it is put on top, the physical device should reflect all changes correctly. It is possible to use the GDV_LU_SPECIFICS structure to store the current state of the physical graphics device.

Step 5. Modify GDV_VPATCH_SPECIFICS to specify the device-specific members of a viewport. This structure is allocated/initialized when the viewport is created. GDV_VPATCH_SPECIFICS maintains information about the current visible state of the viewport. This is similar to the requirements for GDV_DVATCH_SPECIFICS.

Step 6. Modify GDV_BCATCH_SPECIFICS to specify the device-specific members of a Bit-BLT context. Define this structure only in the case when the driver is going to support H/W accelerated Bit-BLT functions. This structure is allocated/initialized when the Bit-BLT context is created. GDV_BCATCH_SPECIFICS maintains information about the current state of the H/W acceleration registers and provides additional storage per Bit-BLT context, which can be useful for implementing the H/W BLT layer.

Step 7. Modify GDV_CPATCH_SPECIFICS to specify the device-specific members of a H/W cursor structure. Define this structure only in the case when the driver is going to support H/W cursors. This structure is allocated/initialized when the H/W cursor is created. GDV_CPATCH_SPECIFICS maintains information about the current state of the H/W cursor registers and provides additional storage per cursor, which can be useful for implementing H/W cursor.

Modify the hardware.h file to reflect your system hardware definitions

Step 1. Modify `hardware.h` to include all necessary hardware related definitions.

Modify the hardware.c file to initialize your hardware

Step 1. Modify the `hardware.c` file with the following considerations:

- `dr_init_hw()` is called when the device is initialized. This function initializes the hardware and calls `gdv_create_mem_color()` to create a color of memory for each bank of graphics memory.
- `dr_term_hw()` is called when the device is terminated. Be sure to return any resources allocated in `dr_init_hw()`.
- `dr_show_topdev()` is called when the changes to the logical device stack have been made (device open/close/restack) in the common portion of the driver GX_COMM or when the “update display” function is called. This function updates the physical device state (operational mode, resolution, CLUT) according to the state of the top-most visible logical device in the stack (depending on how many resolutions and pixel depths the driver can support).
- Different logical devices in the stack require different hardware mode settings. Plus, viewports in the viewport stack can have different CLUT palette settings. `dr_show_topdev()` decides what the current physical device state should be to match it with the top-most visible logical device and the palette of the top-most visible viewport associated with this logical device.

Modify the static.c file to initialize and terminate static storage areas

Step 1. Define the function `dr_init_dvatch()`. This function is called when a device is opened. You do not need to allocate the space for the `GDV_DVATCH_SPECIFICS` structure, but you may allocate other structures and point to them from `GDV_DVATCH_SPECIFICS`. Be sure to initialize all members of `GDV_DVATCH_SPECIFICS`.

If `GDV_INCLUDE_MEM` is defined in `config.h`, call `gdv_create_mem_shade()` to create a shade of memory for each color of graphics memory.

If `GDV_FE_SYSSTATE` is NOT defined in `config.h`, then permit any address space, which is used to set the graphics device up (e.q. I/O registers). Memory will be permitted for the process which called an “open device” function.

Step 2. Define the function `dr_term_dvatch()`. This function is called when a device is terminated. Be sure to de-allocate any resources allocated by the `dr_init_dvatch()` function.

If `GDV_INCLUDE_MEM` is defined, call `gdv_destroy_mem_shade()` to destroy any shades created in the `dr_init_dvatch()` function.

If `GDV_FE_SYSSTATE` is NOT defined in `config.h`, make sure you protect any memory space which was permitted in `dr_init_dvatch()`. Memory will be protected from the process which called an “close device” function.

Step 3. Define the function `dr_init_vpatch()`. This function is called when a viewport is created. You do not need to allocate the space for the `GDV_VPATCH_SPECIFICS` structure, but you may allocate other structures and point to them from `GDV_VPATCH_SPECIFICS`. Be sure to initialize all of its members.

Step 4. Define the function `dr_term_vpatch()`. This function is called when a viewport is terminated. Be sure to deallocate any resources allocated by the `dr_init_vpatch()` function.

Step 5. If `GDV_INCLUDE_HWBLT` is defined, implement `dr_init_bcatch()` function. This function is called when a Bit-BLT context is created. You do not need to allocate the space for the `GDV_BCATCH_SPECIFICS` structure, but you may allocate other structures and point to them from `GDV_BCATCH_SPECIFICS`. Be sure to initialize all of its members.

Step 6. If `GDV_INCLUDE_HWBLT` is defined, implement `dr_term_vpatch()` function. This function is called when a Bit-BLT context is terminated. Be sure to deallocate any resources allocated by the `dr_init_bcatch()` function.

Modify the remaining display functions

Step 1. Modify the display functions in the following files only if they are supported by your hardware.

- `dvbkcol.c` sets the background color for the display.
- `dvextvid.c` sets external video on and off.
- `dvtran.c` sets transparent color.
- `dvvpmix.c` sets viewport mixing on and off.

Modify the remaining viewport functions

Step 1. Modify the viewport functions in the following files. All of these functions are mandatory, and must be included.

- `updtdpy.c` updates the display with queued changes and optionally synchronizes changes with vertical retrace.

- `vpdmap.c` sets the drawmap area to be displayed in a viewport.
- `vpdmpos.c` sets the position of the drawmap in the viewport.
- `vppos.c` sets the position of the specified viewport.
- `vprestak.c` restacks a viewport within the viewport stack.
- `vpsize.c` sets the size of the specified viewport.
- `vpstate.c` sets the state of the specified viewport to active or not active.

Step 2. Modify the viewport functions in the following files only if they are supported by your hardware.

- `hwcur.c` enables driver-supported H/W cursor.
- `irq.c` defines interrupt service functions.
- `vpintens.c` sets the intensity of the specified viewport.

Step 3. Modify the following file only if either I/O BLT functions, or H/W BLT functions or both are supported by your hardware.

- `ioblt.c` enables driver-supported Bit-BLT (using I/O registers).
- `hwblt.c` enables driver-supported Bit-BLT (using H/W acceleration).

How to Build your Graphics Driver

Step 1. Change directories to *MAKE* directory:

```
cd MWOS/OS/CPU/PORTS/YOURPORT/MAUI/GX_YOURDRVR
```

Step 2. To make both the graphics driver and descriptor, type:

```
os9make
```

The `makefile` invokes `desc.mak` and `drvr.mak`

To make only the graphics descriptor, type:

```
os9make -f desc.mak
```

The `desc.mak` makefile builds the graphics descriptor and places it in the directory *YOURPORT*`/CMDS/BOOTOBJS/MAUI`.

To make only the graphics driver, type:

```
os9make -f drvr.mak
```

The `drvr.mak` builds the graphics driver and places it in the directory *YOURPORT*`/CMDS/BOOTOBJS/MAUI`.

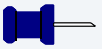
How to Test Your Driver

Run the demo programs included with MAUI on your target platform to test your driver. Demo program sources are located below the following directory:

`MWOS/SRC/MAUI/DEMOS`

Demo program objects are located in the following directory:

`MWOS/OS/CPU/CMDS/MAUIDEMO`



Note

These demos are not designed to be a comprehensive test of the graphics driver.

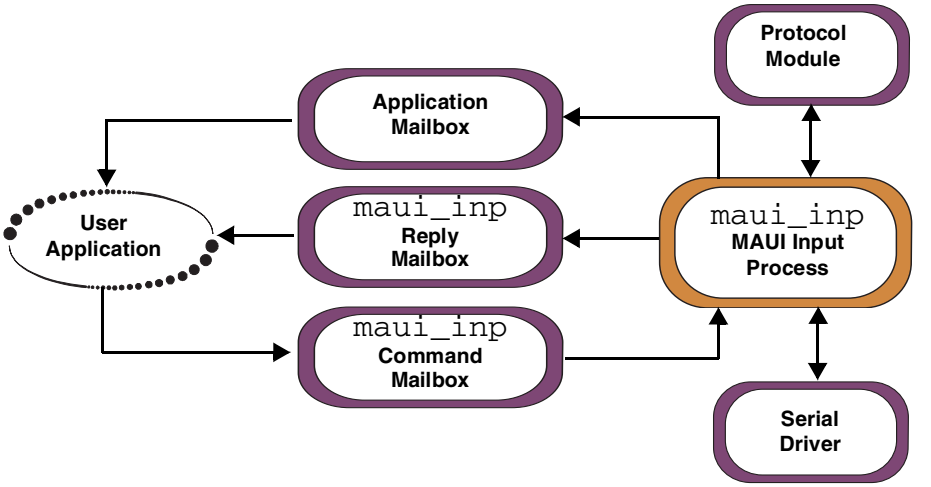
Chapter 3: Input

The MAUI Input System (MIS) provides an abstraction layer between the application and the raw serial output from the input hardware and their drivers. This abstraction layer insulates the applications from many of the hardware differences between target systems. The MIS provides key code translation, asynchronous messaging, pointer and key simulation, and device arbitration. This chapter explains how to build, modify, and verify MAUI Input Process Protocol modules.

Overview

The MAUI Input System (MIS) consists of several components; the INP API, the MAUI Input Process (`maui_inp`), and the MAUI Input Process Protocol Modules (MPPMs). The `maui_inp` components, protocol modules, and the relationship to other components in a typical MAUI application are depicted in **MAUI Input Process System Diagram**.

Figure 3-1 MAUI Input Process System Diagram



Input API

The INP API handles all communication between the application and the `maui_inp` process. Although the communication between the API and `maui_inp` take place via messaging, this is transparent to the application. When each application initializes the INP API, the API creates a uniquely named Reply Mailbox. It then opens the `maui_inp` Command Mailbox and sends an `init` message to inform `maui_inp` that it is there. When an application opens an input device, the pathname specifies both the name of the serial device and protocol (MPPM) to use.

MAUI Input Process

`maui_inp` provides routing, connection management, and message handling services. The `maui_inp` process is responsible for managing multiple applications, simultaneously using multiple input devices, with many different protocols. `maui_inp` monitors which applications are using the INP API and keeps track of which applications have open paths to which input sources. When data is received, the `maui_inp` process applies the appropriate protocol to the appropriate device, depending on which application has the focus for that device. When multiple applications open the same device with the same protocol, `maui_inp` also provides key reservation.

`maui_inp` is hardware and protocol independent. It uses MAUI Input Process Protocol Modules (MPPMs) to insulate it from hardware and protocol differences between ports.

When `maui_inp` starts, it creates a single Command Mailbox named `mp_mbox`. All messages from the INP API to `maui_inp` and the MPPMs are sent to this Command Mailbox. `maui_inp` replies to these messages via a Reply Mailbox that is created by the application's INP API.

MAUI Input Protocol Modules

The MAUI Input Process Protocol Modules (MPPMs) provide the command control and response, as well as data interpretation. MPPMs are generally hardware independent (that is the job of drivers), but protocol dependent.

MPPMs are implemented as raw subroutine modules. In the interest of minimizing the overhead of calling MPPM functions, these modules are called directly without correcting the static or constant storage pointers. As such, they do not have any memory of their own. They operate completely on the stack of the process that calls them, in this case `maui_inp`. **Static and global variables are not allowed in MPPMs**. Use of variables that are not allocated on the stack can severely damage `maui_inp`'s internal data structures. If your MPPM requires persistent memory, include those variables in the `PMEM` structure.

OEMs are encouraged to take the basic MPPM examples, and modify them as appropriate for their input devices.

WARNING

Static and global variables are not allowed in MPPMs.

Where the Files are Located

MAUI header files are located in:

```
MWOS/SRC/DEFS/MAUI
```

WARNING

These header files should never be modified by the user.

There are several example source directories of MPPMs. Two basic MPPMs are:

```
MWOS/SRC/MAUI/MP/MP_KYBRD
MWOS/SRC/MAUI/MP/MP_MSPTR
```

`MP_KYBRD` is example source code for a key device, specifically, a serial port connected to a VT100 terminal or communications program. `MP_MSPTR` is example source code for a pointer device, specifically a two-button Microsoft®-compatible type M mouse. See your release notes for a complete list and description of what protocol modules are included in your package. Depending on the type of your device (key or pointer), pick one of the example MPPMs as your starting point.

Within the above source directories are the following source files:

- `_key.h` port-specific header definitions.
- `init.c` initialize static memory and register processes.
- `mppmstrt.a` sub-routine entry point table.
- `procdata.c` interprets device data.
- `procmsg.c` process commands from `maui_inp`.
- `term.c` un-register processes.

How to Port Your Protocol Module

The first step in porting to a new input device is to determine what kind of device is being ported. MAUI divides input devices into three classes, key devices, pointer devices and a hybrid (combination pointer and key) device.

A key device generates key symbol data. Examples include keyboards and remote controls.

A pointer device generates coordinate information, either absolute or relative. They may also have buttons. Examples include mice, joysticks, touchscreens, rollerballs, pens, and tablets.

The third class of device is a hybrid device. This device can generate both coordinate information and key symbol information.

Porting a Key Device

For key devices, use the `MP_KYBRD` example located in the following directory:

```
MWOS/SRC/MAUI/MP/MP_KYBRD
```

Create the directory structure for your port

Before beginning to port your protocol module, you must create a directory structure to store your new files.

Step 1. Define and create a source directory. This directory is referred to in this chapter as *SOURCE* and assumes the pathname:
`MWOS/SRC/MAUI/MP/`*SOURCE*

Step 2. Copy all of the files from:
`MWOS/SRC/MAUI/MP/MP_KYBRD`
into your new *SOURCE* directory. Verify the following files are now in your *SOURCE* directory:

`_key.h`	port-specific header definitions.
`init.c`	initialize static memory and register processes.
`mppmstrt.a`	sub-routine entry point table.
`procdata.c`	interprets device data.
`procmsg.c`	process commands from `maui_inp`.
`term.c`	un-register processes.

Step 3. Define and create a ports directory. This directory is referred to in this chapter as *YOURPORT* and assumes the pathname:
`MWOS/OS/CPU/PORTS/`*YOURPORT*

Step 4. Define and create a *MP_YOURMPPM* directory. This directory is referred to in this chapter as *MP_YOURMPPM* and assumes the pathname:
`MWOS/OS/CPU/PORTS/`*YOURPORT*`/MAUI/`*MP_YOURMPPM*

Step 5. Create the following file in *MP_YOURMPPM*

`makefile` Make the MPPM

```
# Makefile
#*******************************************************************************
#* This makefile builds a MAUI Input Process Protocol Module
#*******************************************************************************
#* Copyright 1995 by Microware Systems Corporation                             **
#* Copyright 2001 by RadiSys Corporation                                       **
#* Reproduced Under License                                                    **
#*                                                                             **
#* This source code is the proprietary confidential property of                **
#* Microware Systems Corporation, and is provided to licensee                  **
#* solely for documentation and educational purposes. Reproduction,            **
#* publication, or distribution in any form to any party other than            **
#* the licensee is strictly prohibited.                                        **
#*******************************************************************************

PORT    =   ../..

TRGTS       = mp_kybrd

USER_OPTS   =
USER_HEADERS =
USER_RFILES =
USER_LIBS   =

include $(PORT)/../make.com
```

```
ODIR    = $(PORT)/CMDS/BOOTOBJS/MAUI
SDIR    = $(MWOS)/SRC/MAUI/MP/MP_KYBRD
RDIR    = RELS
IDIR    = $(RDIR)/$(HOSTTYPE)

include $(SDIR)/../pmod.com

#
# Put USER_RFILES rules (if any) here
#
```

_key.h

This file defines the default settings for the protocol module.

For a key device, you normally only change the `DEV_CAP_*` definitions. These definitions are used to fill out the `INP_DEV_CAP` structure.

The `PMEM` structure is the static memory for the protocol module. It is allocated by `maui_inp` on behalf of the protocol module for each opened device. Extend this field if you have additional memory requirements.

init.c

This file has two entry points, `mppm_initsize()` and `mppm_init()`. Normally, no changes are required in this file.

mppmstrt.a

This assembly source file contains the subroutine module entry point table.

WARNING

Do not modify `mppmstrt.a`!

procdata.c

This file contains the functions necessary to process raw data from the input device and build key and pointer messages. `mppm_process_data()` is this file's only external entry point. This function is called whenever there is data to be processed.

For key devices, there is usually only one section to modify in `procdata.c`. That section parses the raw input data and translates that data into standardized key symbol data.

```
/* fill keybuf */
key = *(*buf)++;
(*buf_size)--;

/* do any required key translation */
   switch (key) {
   case 0x7f: key= INP_KEY_CLEAR; break;
   case 0x80: key= INP_KEY_PLAY; break;
   case 0x81: key= INP_KEY_STOP; break;
   case 0x82: key= INP_KEY_PAUSE; break;
   case 0x85: key= INP_KEY_REWIND; break;
   case 0x86: key= INP_KEY_FASTFWD; break;
   case 0x88: key= INP_KEY_CUR_U; break;
   case 0x89: key= INP_KEY_CUR_D; break;
   case 0x8a: key= INP_KEY_CUR_R; break;
   case 0x8b: key= INP_KEY_CUR_L; break;
   case 0x91: key= INP_KEY_LASTCHAN; break;
   case 0x92: key= INP_KEY_EXIT; break;
   case 0x94: key= INP_KEY_STORE; break;
   case 0x99: key= INP_KEY_CHAN_U; break;
   case 0x9a: key= INP_KEY_CHAN_D; break;
   case 0x9c: key= INP_KEY_MENU; break;
   case 0x9d: key= INP_KEY_VIP; break;
   case 0x9e: key= INP_KEY_VDT; break;
   case 0xac: key= INP_KEY_VOL_U; break;
   case 0xad: key= INP_KEY_VOL_D; break;
   case 0xae: key= INP_KEY_MUTE; break;
   case 0xed: key= INP_KEY_RECORD; break;
   }
```

Modify this section to parse and translate the data from your input device. The above example code is based on a remote that generates single-byte key data. Other remotes may require more translation. If you have more keys that require translation, you may wish to use a translation table rather than a `switch` statement.

If you have a multi-byte input packet, you can find an example of how to deal with incomplete packets in `procdata.c` of `MP_MSPTR`.

procmsg.c

This file contains all the functions necessary for processing messages from `maui_inp`.

`mppm_process_msg()` is this file's only external entry point. `mppm_process_msg()` routes a message to the appropriate function (listed next) based on the command code found in the `cmd_msg->any.dcom.cmd` variable.

```
static error_code cmd_get_dev_cap();
static error_code cmd_get_dev_status();
static error_code cmd_set_ptr_pos();
static error_code cmd_set_sim_meth();
static error_code cmd_set_ptr_limit();
static error_code cmd_set_msg_callback();
static error_code cmd_set_msg_mask();
static error_code cmd_release_key();
static error_code cmd_reserve_key();
static BOOLEAN cmd_check_keys();
```

Usually, the only function you need to modify is `cmd_check_keys()`. Modify this function to return `TRUE` for key ranges present on the device.

term.c

This file contains the functions for terminating the use of this protocol module. The two entry points are `mppm_term()` and `mppm_detach()`. Normally, no changes are required for this file.

Porting a Pointer Device

For pointer devices, use the MP_MSPTR example located in the following directory:

MWOS/SRC/MAUI/MP/MP_MSPTR

Create the directory structure for your port

Before beginning to port your protocol module, you must create a directory structure to store your new files.

Step 1. Define and create a source directory. This directory is referred to in this chapter as *SOURCE* and assumes the pathname:
MWOS/SRC/MAUI/MP/*SOURCE*

Step 2. Copy all of the files from:
MWOS/SRC/MAUI/MP/MP_MSPTR
into your new *SOURCE* directory. Verify the following files are now in your *SOURCE* directory:

_key.h	port-specific header definitions.
init.c	initialize static memory and register processes.
mppmstrt.a	sub-routine entry point table.
procdata.c	interprets device data.
procmsg.c	process commands from maui_inp.
term.c	un-register processes.

Step 3. Define and create a ports directory. This directory is referred to in this chapter as *YOURPORT* and assumes the pathname:
MWOS/OS/CPU/PORTS/*YOURPORT*

Step 4. Define and create a *MP_YOURMPPM* directory. This directory is referred to in this chapter as *MP_YOURMPPM* and assumes the pathname:
MWOS/OS/CPU/PORTS/*YOURPORT*/MAUI/*MP_YOURMPPM*

Step 5. Create the following file in *MP_YOURMPPM*

`makefile` Make the MPPM

```
# Makefile
#*****************************************************************************
#* This makefile builds a MAUI Input Process Protocol Module
#*****************************************************************************
#* Copyright 1995 by Microware Systems Corporation                           **
#* Copyright 2001 by RadiSys Corporation                                     **
#* Reproduced Under License                                                  **
#*                                                                           **
#* This source code is the proprietary confidential property of              **
#* Microware Systems Corporation, and is provided to licensee                **
#* solely for documentation and educational purposes. Reproduction,          **
#* publication, or distribution in any form to any party other than          **
#* the licensee is strictly prohibited.                                      **
#*****************************************************************************

PORT    =   ../..

TRGTS       = mp_msptr

USER_OPTS   =
USER_HEADERS =
USER_RFILES =
USER_LIBS   =

include $(PORT)/../make.com

ODIR    = $(PORT)/CMDS/BOOTOBJS/MAUI
SDIR    = $(MWOS)/SRC/MAUI/MP/MP_MSPTR
RDIR    = RELS
IDIR    = $(RDIR)/$(HOSTTYPE)

include $(SDIR)/../pmod.com

#
# Put USER_RFILES rules (if any) here
#
```

_key.h

This file defines the default settings for the protocol module. Modify the `DEV_CAP_*` definitions to reflect the capabilities of your device. These definitions are used to fill out the `INP_DEV_CAP` structure.

Modify the PMEM structure if you have additional memory requirements. The PMEM structure is the static memory for the protocol module. It is allocated by maui_inp on behalf of the protocol module for each opened device.

Modify NUM_IMSG based on the number of messages that can be queued. This is usually (DEV_CAP_PTR_BUTTONS+1).

Modify NUM_PKT_BUF based on the size of a data packet. In the MP_MSPTR example, this is three bytes because the mouse generates three byte packets. This definition is used to determine the size of pktbuf in PMEM.

init.c

This file has two entry points, mppm_initsize() and mppm_init(). Normally, no changes are required in this file.

mppmstrt.a

This assembly source file contains the subroutine module entry point table.

WARNING

Do not modify mppmstrt.a!

procdata.c

This file contains all functions necessary to process raw data from the input device and build key and pointer messages. mppm_process_data() is this file's only external entry point. This function is called whenever there is data to be processed.

First modify the section that builds a complete mouse packet in the packet buffer. Modify this section to synchronize and packetize the raw data from your input device.

```
/************************************************
**
 * build a mouse packet in the packet buffer
 ************************************************/

/* If on 1st byte of packet, advance 1 byte at a time
   until we get a good start byte (bit 6 set to 1) */
if (pmem->pktcnt == 0)
{
   while (*buf_size && !(*(*buf)&1<<6))
   {
      (*buf)++;/* advance buffer pointer */
      (*buf_size)--;/* decrement buffer counter */
   }
}

/* fill pktbuf */
while (*buf_size && pmem->pktcnt < 3)
{
   pmem->pktbuf[pmem->pktcnt++] = *(*buf)++;
   (*buf_size)--;
}

/* if the packet is not complete, leave until it is */
if (pmem->pktcnt < 3)
{
   *inp_msg = NULL;/* don't send a msg yet */
   return SUCCESS;
}
```

The next section decodes the button and coordinates data from the packet. Modify this section to decode your data packet into button and coordinate information.

```
/************************************************
 * decode the mouse data packet
 ************************************************/

/* save off the old position */
old_x = status->ptr_cur.x;
/* compute the new position */
status->ptr_cur.x += (int8)(((pmem->pktbuf[0] << 6)
      & 0xc0) | (pmem->pktbuf[1] & 0x3f));
/* keep it in bounds */
```

```
LIMIT (status->ptr_cur.x, status->ptr_min.x,
       status->ptr_max.x);
/* compute the real change */
new_x_delta = status->ptr_cur.x - old_x;

/* save off the old position */
old_y = status->ptr_cur.y;

/* compute the new position */
status->ptr_cur.y += (int8)(((pmem->pktbuf[0] << 4)
      & 0xc0) | (pmem->pktbuf[2] & 0x3f));
/* keep it in bounds */
LIMIT (status->ptr_cur.y, status->ptr_min.y,
      status->ptr_max.y);
/* compute the real change */
new_y_delta = status->ptr_cur.y - old_y;

/* grab the new button state */
new_button_state = ((pmem->pktbuf[0] >> 5) & 1)
      | ((pmem->pktbuf[0] >> 3) & 2);
button_change = status->button_state ^ new_button_state;
status->button_state = new_button_state;
```

procmsg.c

This file contains all functions necessary for processing messages from `maui_inp`.

`mppm_process_msg()` is this file's only external entry point. `mppm_process_msg()` routes a message to the appropriate function (listed below) based on the command code found in the `cmd_msg->any.dcom.cmd` variable. For pointer devices, functions in this file usually do not need modification.

```
static error_code cmd_get_dev_cap();
static error_code cmd_get_dev_status();
static error_code cmd_set_ptr_pos();
static error_code cmd_set_sim_meth();
static error_code cmd_set_ptr_limit();
static error_code cmd_set_msg_callback();
static error_code cmd_set_msg_mask();
static error_code cmd_release_key();
static error_code cmd_reserve_key();
static BOOLEAN cmd_check_keys();
```

term.c

This file contains functions for terminating the use of this protocol module. The two entry points are `mppm_term()` and `mppm_detach()`. Normally, no changes are required in this file.

How to Build Your Protocol Module

Step 1. Change directories to *MP_YOURMPPM* directory:

```
cd MP_YOURMPPM
```

Step 2. To make the protocol module, type:

```
os9make
```

How to Test Your Protocol Module

Use the Input API to exercise the protocol module.

Testing Key Devices

Verify that `inp_check_keys()` returns true for all keys on the device. Verify that each key returns the proper key code.

Testing Pointer Devices

Verify the protocol module responds correctly to events such as simultaneous movement and button presses.

Input Protocol Module Entry Points

MAUI Input Process Protocol Modules (MPPMs) have seven entry points. This section describes each entry point and their responsibilities.

Summary of MAUI Hardware-Layer Functions

Table 3-1 contains a list of all MAUI hardware-layer functions.

Table 3-1 MPPM Entry Point Functions

Function	Description
`mppm_attach()`	attaches a device.
`mppm_detach()`	detaches a device.
`mppm_init()`	initializes a device.
`mppm_initsize()`	gets the protocol module's static memory size requirements.
`mppm_process_data()`	interprets device data.
`mppm_process_msg()`	processes command messages.
`mppm_term()`	terminates device.

Location of MAUI Hardware-Layer Functions

MAUI hardware-layer functions are located in the files shown in **Table 3-2**.

Table 3-2 Location of MPPM Entry Points

Function	File Name
mppm_attach()	init.c
mppm_detach()	term.c
mppm_init()	init.c
mppm_initsize()	init.c
mppm_process_data()	procdata.c
mppm_process_msg()	procmsg.c
mppm_term()	term.c

mppm_attach()

Attaches to a Device

Syntax

```
#include <mppm.h>
error_code mppm_atttach(MP_DEV *mp_dev);
```

Description

`mppm_attach()` notifies a protocol module that a new device has been opened.

`mppm_attach()` is called when an application calls `inp_open_dev()`.

Parameters

`mp_dev`	points to the data structure that represents an opened input device path.

Direct Errors

`SUCCESS(0)` if no error occurred.

See Also

`inp_open_dev()` (See ***MAUI Programming Reference Manual***)

`MP_DEV`

mppm_detach()

Detaches Device

Syntax

```
#include <mppm.h>
error_code mppm_detach(MP_DEV *mp_dev);
```

Description

`mppm_detach()` notifies the protocol module that an application has closed the input device. `mppm_detach()` is called by `maui_inp` when an application calls `inp_close_dev()` or `inp_term()`.

`mppm_detach()` releases any reserved keys for the calling application before closing the device.

Parameters

`mp_dev`	points to the data structure that represents an opened input device path.

Direct Errors

`SUCCESS(0)` if no error occurred.

See Also

`MSG_CLOSE_DEV` (See ***MAUI Programming Reference Manual***)
`MSG_INP_TERM` (See ***MAUI Programming Reference Manual***)
`MP_DEV`

mppm_init()

Initializes Static Memory

Syntax

```
#include <mppm.h>
error_code mppm_init(MP_MPPM *mppm,
   void *mem_buf, size_t mem_size);
```

Description

`mppm_init()` initializes the protocol module's static memory `mem_buf`. `mppm_init()` is called by `maui_inp` after `maui_inp` allocates the amount of memory specified by `maui_initsize()`.

Parameters

`mppm`	points to a data structure that points to the device and protocol module.
`mem_buf`	points to the protocol module's static memory. `mem_buf` is allocated and attached to `mppm` by `maui_inp`.
`mem_size`	contains the size of the protocol module's static memory as returned by `mppm_initsize()`.

Direct Errors

`SUCCESS(0)` if no error occurred.

See Also

`inp_init()`(See ***MAUI Programming Reference Manual***)
`mppm_initsize()`
`MP_MPPM`

RadiSys. MICROWARE SOFTWARE

mppm_initsize()

Gets Static Memory Requirements

Syntax

```
#include <mppm.h>
error_code mppm_initsize(MP_MPPM *mppm,
   size_t *mem_size);
```

Description

`mppm_initsize()` returns the protocol module's static memory size requirements in `mem_size`. This call also sets the protocol module's compatibility level in the mppm structure.

`mppm_initsize()` is called by `maui_inp` upon receipt of a `MSG_INP_INIT` message.

Parameters

`mppm`	points to a data structure that contains information on the static memory space required.
`mem_size`	contains the size of the protocol module's static memory.

Direct Errors

OS-9/OS-9000 error code or `SUCCESS(0)` if no error occurred.

See Also

`mppm_init()`
`MSG_INP_INIT` (See ***MAUI Programming Reference Manual***)
`MP_MPPM`

mppm_process_data()

Interprets Device Data

Syntax

```
#include <mppm.h>
error_code mppm_process_data(MP_MPPM *reply_mppm,
   u_char **buf,
   size_t *buf_size,
   MSG_MBOX_ID *mbox_id,
   INP_MSG **reply_msg);
```

Description

`mppm_process_data()` receives raw data from the SCF device in `buf`, then returns standardized key and pointer messages in `reply_msg`.

Parameters

`mppm`	points to the current device and protocol module static memory associated with the data.
`buf`	points to the buffer where the raw data is stored. `buf` is updated to the next unprocessed byte at the conclusion of this call.
`buf_size`	contains the number of bytes of data available for processing. `buf_size` is updated with the number of unprocessed bytes remaining at the conclusion of this call.
`mbox_id`	is set to an alternative mailbox ID if the message in `reply_msg` needs to be redirected (for example, key reservations).

reply_msg	the maui_inp process forwards the message to the application's mailbox if this pointer is not NULL.

Direct Errors

OS-9/OS-9000 error code or SUCCESS(0) if no error occurred.

EOS_UNFINISHED returned when mppm_process_data() needs to be recalled to complete a task.

EOS_READ returned when there is an error interpreting incoming data.

See Also

INP_MSG (See ***MAUI Programming Reference Manual***)
MP_MPPM
MSG_MBOX_ID (See ***MAUI Programming Reference Manual***)

mppm_process_msg()

Processes Command Messages

Syntax

```
#include <mppm.h>
error_code mppm_process_msg(MP_MPPM *mppm,
   MP_DEV_MSG *cmd_msg,
   MP_DEV_MSG **reply_msg);
```

Description

`mppm_process_msg()` processes all device command messages.

Parameters

`mppm`	points to the current device and protocol module static memory associated with the message.
`cmd_msg`	points to a structure containing the device command message.
`reply_msg`	if not set to `NULL` when this function returns, it points to the reply message.

Direct Errors

OS-9/OS-9000 error code or `SUCCESS(0)` if no error occurred.

`EOS_MAUI_BADACK` returned when command code is not understood.

See Also

`MSG_GET_DEV_CAP`
`MSG_GET_DEV_STATUS`
`MSG_SET_PTR_POS`
`MSG_SET_SIM_METH`
`MSG_SET_PTR_LIMIT`
`MSG_RESERVE_KEY`
`MSG_RELEASE_KEY`
`MSG_SET_MSG_MASK`

MSG_SET_MSG_CALLBACK
MP_DEV_MSG
MP_MPPM

mppm_term()

Terminates Process

Syntax

```
#include <mppm.h>
error_code mppm_term(MPPM *mppm);
```

Description

`mppm_term()` is called by `maui_inp` upon receipt of a `MSG_INP_TERM` message.

Parameters

`mppm`	points to the current device and protocol module static memory associated with the data.

Direct Errors

OS-9/OS-9000 error code or `SUCCESS(0)` if no error occurred.

See Also

`MSG_INP_TERM` (See ***MAUI Programming Reference Manual***)
`MP_MPPM`

Functional Data Reference

This section gives a detailed reference for each of the data types in this interface. These are the only data types defined and recognized by this interface.

Table 3-3 Data Structures/Data Types

Name	Description
MP_DEV	Input device path/mailbox data
MP_MPPM	Device and protocol module data

MP_DEV

Input Device Path/Mailbox Data

Syntax

```
#include <mppm.h>
typedef struct _MP_DEV
{
   u_int32 sync_code;/* syn code - _MP_DEV_SYNC */
   MP_PROC *proc;/* owner proc */
   MP_DEV *proc_next_dev;/* proc linked list */
   MP_MPPM *mppm;/* device and pmod */
   MP_DEV *mppm_next_dev;/* mppm linked list */
   MSG_MBOX_ID app_mbox_id;/* mbox to send msgs */
   INP_DEV_ID device_id;/* ID to return in PTR */
      /* and KEY messages */
} MP_DEV;
```

Description

This data structure represents an opened input device path and is seen by the Input API as `MP_DEV_ID`.

See Also

`INP_DEV_ID` (See ***MAUI Programming Reference Manual***)
`MP_DEV_ID`
`MP_PROC_ID`
`MSG_MBOX_ID` (See ***MAUI Programming Reference Manual***)
`MSG_INP_TERM` (See ***MAUI Programming Reference Manual***)
`MP_MPPM`

RadiSys. MICROWARE SOFTWARE

MP_MPPM

Device and Protocol Module Data

Syntax

```
#include <mppm.h>
typedef struct _MP_MPPM
{
   u_int32 maui_inp_compat_level;/* maui_inp level */

   /* owner info */
   MP_DEV *mp_dev_head;/* current mbox */
   MP_MPPM *next;/* next mbox */
   MP_MPPM *prev;/* previous mbox */

   /* raw device info */
   path_id dev_path;/* device path id */
   char dev_type;/* device type */
   char dev_name[INP_MAX_DEV_NAME];
      /* device name */

   /* protocol module info */
   u_int32 pmod_compat_level;/* pmod level */
   mh_com *pmod_head;  /* pmod module header */
   char pmod_name[INP_MAX_DEV_NAME];
      /* pmod name */
   void *pmod_mem;/* pmod static mem */
   void *pmod_functable;/* pmod entry points */
} MP_MPPM;
```

Description

This data structure represents each unique combination of device path and protocol module. MPPMs use this structure to find their static memory space (`pmod_mem`).

See Also

`INP_MAX_DEV_NAME` (See ***MAUI Programming Reference Manual***)

```
MP_DEV
```

Message reference

This section provides a complete reference for each of the command and reply messages handled by `mppm_process_msg`.

Table 3-4 mppm_process_msg

Command	Description
MSG_CHECK_KEYS	checks for existence of keys.
MSG_GET_DEV_CAP	gets device capabilities.
MSG_GET_DEV_STATUS	gets device status.
MSG_RELEASE_KEY	releases a reserved key.
MSG_RESERVE_KEY	reserves a key for a process.
MSG_RESTACK_DEV	restacks an input device.
MSG_SET_SIM_METH	sets pointer simulation mode.
MSG_SET_MSG_CALLBACK	sets message callback.
MSG_SET_MSG_MASK	sets message write mask.
MSG_SET_PTR_LIMIT	sets pointer limit.
MSG_SET_PTR_POS	sets pointer position.

MSG_CHECK_KEYS

Checks if Key Exists

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_CHECK_KEYS
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_CHECK_KEYS */
   wchar_t min_key;/* first key symbol to /*
                /* reserve */
   wchar_t max_key;/* last key symbol to reserve */
} MSG_CHECK_KEYS;
```

Syntax Reply Structure

```
typedef struct _MSG_CHECK_KEYS_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
      /* CMD_CHECK_KEYS_REPLY */
   BOOLEAN present;/* return TRUE if all present */
   error_code error;/* return error code */
} MSG_CHECK_KEYS_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_check_keys()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_check_keys()` (See ***MAUI Programming Reference Manual***)

`MSG_COMMON_MPCMD`

`MSG_GET_DEV_CAP`

`BOOLEAN` (See ***MAUI Programming Reference Manual***)

MSG_GET_DEV_CAP

Gets Device Capabilities

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_GET_DEV_CAP
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_GET_DEV_CAP */
} MSG_GET_DEV_CAP;
```

Syntax Reply Structure

```
typedef struct _MSG_GET_DEV_CAP_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_GET_DEV_CAP_REPLY */
   INP_DEV_CAP cap;/* device information */
   error_code error;/* return error code */
} MSG_GET_DEV_CAP_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_get_dev_cap()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_get_dev_cap()` (See ***MAUI Programming Reference Manual***)

`INP_DEV_CAP` (See ***MAUI Programming Reference Manual***)

`MSG_COMMON_MPCMD`

MSG_GET_DEV_STATUS

Gets Device Status

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_GET_DEV_STATUS
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_GET_DEV_STATUS */
} MSG_GET_DEV_STATUS;
```

Syntax Reply Structure

```
typedef struct _MSG_GET_DEV_STATUS_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_GET_DEV_STATUS_REPLY */
   INP_DEV_STATUS status;/* device information */
   error_code error; /* return error code */
} MSG_GET_DEV_STATUS_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_get_dev_status()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_get_dev_status()` (See ***MAUI Programming Reference Manual***)
`INP_DEV_STATUS` (See ***MAUI Programming Reference Manual***)
`MSG_TYPE_MPCMD`

MSG_RELEASE_KEY

Releases a Reserved Key

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_RELEASE_KEY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_RELEASE_KEYS */
   wchar_t key; /* first key symbol to */
                /* release */
} MSG_RELEASE_KEY;
```

Syntax Reply Structure

```
typedef struct _MSG_RELEASE_KEY_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /*  CMD_RELEASE_KEYS_REPLY */
   error_code error;/* return error code */
} MSG_RELEASE_KEY_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_release_key()`.

The protocol module is responsible for formatting the reply.

Direct Errors

`EOS_MAUI_NOTRESERVED` returned when the key is not currently reserved.

`EOS_MAUI_NOHWSUPPORT` returned when the specified key is not supported by the hardware.

Indirect Errors

`mppm_process_msg()`

See Also

`inp_release_key()` (See ***MAUI Programming Reference Manual***)
`MSG_TYPE_MPCMD`
`MSG_RESERVE_KEY`

MSG_RESERVE_KEY

Reserves a Key for a Process

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_RESERVE_KEY
{
   MSG_COMMON_MPCMD dcom;
                /* dcom.cmd = CMD_RESERVE_KEY */
   wchar_t key; /* key symbol to reserve */
} MSG_RESERVE_KEY;
```

Syntax Reply Structure

```
typedef struct _MSG_RESERVE_KEY_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_RESERVE_KEYS_REPLY */
   error_code error;/* return error code */
} MSG_RESERVE_KEYS_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_reserve_key()`.

The protocol module is responsible for formatting the reply.

Direct Errors

`EOS_MAUI_ISRESERVED` returned when key is already reserved. `EOS_MAUI_NHWSUPPORT` returned when the protocol module does not support key reservation. Most often because the device does not have keys.

Indirect Errors

`mppm_process_msg()`

See Also

`inp_reserve_key()` (See ***MAUI Programming Reference Manual***)
MSG_COMMON_MPCMD
MSG_RELEASE_KEY

MSG_RESTACK_DEV

Re-stack an Input Device

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_RESTACK_DEV {
   MSG_COMMON_MPCMD dcom;/*dcom.cmd = CMD_RESTACK_DEV */
   INP_DEV_PLACEMENT placement;/*placement in stack of */
         /*devices */
   MP_DEV_ID ref_dev_id;/* reference device */
} MSG_RESTACK_DEV;
```

Syntax Reply Structure

```
typedef struct _MSG_RESTACK_DEV_REPLY {
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
      /* CMD_RESTACK_DEV_REPLY /*
   error_code error;/* return error code */
} MSG_RESTACK_DEV_REPLY;
```

Description

This message instructs `maui_inp` to change the placement of the logical input device `dcom->dev_id` within the current stack of logical devices. The following table shows how placement and `ref_dev_id` specify the new position. The Reference Device column indicates when the `ref_dev_id` is applicable. If successful, this device returns `SUCCESS`.

Table 3-5 Value of Placement in MSG_RESTACK_DEV

Value of Placement	Reference Device	New Position
`INP_DEV_FRONT`	Not applicable	In front of all devices
`INP_DEV_BACK`	Not applicable	In back of all devices

Table 3-5 Value of Placement in MSG_RESTACK_DEV (continued)

Value of Placement	Reference Device	New Position
INP_DEV_FRONT_OF	MP_DEV_ID ref_dev_id	In front of device ref_dev_id
INP_DEV_BACK_OF	MP_DEV_ID ref_dev_id	In back of device ref_dev_id

Direct Errors

EOS_MAUI_BADID is returned when the ID specified by dcom->dev_id or ref_dev_id is not valid.
EOS_MAUI_BADVALUE is returned when placement value is not valid.
EOS_MAUI_DAMAGE is returned when maui_inp has detected inconsistencies in internal data structures.

Indirect Errors

None

See Also

inp_restack_dev() (See ***MAUI Programming Reference Manual***)
MSG_COMMON_MPCMD

MSG_SET_SIM_METH

Sets Simulation Mode

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_SET_SIM_METH
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_CURSOR_SIM */
   INP_SIM_METH sim_meth;/* simulation mode */
   GFX_DELTA speed;/* X/Y speed for simulation */
   wchar_t button_map[INP_MAX_BUTTONS];
                /* button to key mapping */
} MSG_SET_SIM_METH;
```

Syntax Reply Structure

```
typedef struct _MSG_SET_SIM_METH_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_CURSOR_SIM_REPLY */
   error_code error;/* return error code */
} MSG_SET_SIM_METH_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_set_sim_meth()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_set_sim_meth()` (See ***MAUI Programming Reference Manual***)
`GFX_DELTA` (See ***MAUI Programming Reference Manual***)
`INP_CUR_SIM` (See ***MAUI Programming Reference Manual***)
`INP_MAX_BUTTONS` (See ***MAUI Programming Reference Manual***)
`MSG_COMMON_MPCMD`
`MSG_GET_DEV_STATUS`

MSG_SET_MSG_CALLBACK

Sets Message Callback

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_SET_MSG_CALLBACK
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_MSG_CALLBACK */
   void (*callback)(const void *msg);
                /* pointer to callback */
                /* function */
} MSG_SET_MSG_CALLBACK;
```

Syntax Reply Structure

```
typedef struct _MSG_SET_MSG_CALLBACK_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd =
                /*CMD_SET_MSG_CALLBACK_REPLY */
   error_code error;/* return error code */
} MSG_SET_MSG_CALLBACK_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_set_callback()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_set_callback()` (See ***MAUI Programming Reference Manual***)

`MSG_COMMON_MPCMD`

MSG_SET_MSG_MASK

Sets Message Write Mask

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_SET_MSG_MASK
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_MSG_MASK */
   u_int32 write_mask;/* Message write mask */
} MSG_SET_MSG_MASK;
```

Syntax Reply Structure

```
typedef struct _MSG_SET_MSG_MASK_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd =
                /* CMD_SET_MSG_MASK_REPLY */
   error_code error;/* return error code */
} MSG_SET_MSG_MASK_REPLY;
```

Description

This message sets the message write mask and then passes the message to the protocol module via `mppm_process_msg()` when the application calls `inp_set_msg_mask()`.

Direct Errors

None

Indirect Errors

`msg_set_mask()` (See ***MAUI Programming Reference Manual***)
`mppm_process_msg()`

See Also

`inp_set_msg_mask()` (See ***MAUI Programming Reference Manual***)

`MSG_COMMON_MPCMD`

MSG_SET_PTR_LIMIT

Sets Pointer Limit

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_SET_PTR_LIMIT
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_PTR_LIMIT */
   GFX_POINT ptr_min;/* minimum position for the */
                /* pointer */
   GFX_POINT ptr_max;/* maximum position for the */
                /* pointer */
} MSG_SET_PTR_LIMIT;
```

Syntax Reply Structure

```
typedef struct _MSG_SET_PTR_LIMIT_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_PTR_LIMIT_REPLY */
   error_code error; /* return error code */
} MSG_SET_PTR_LIMIT_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `ind_set_ptr_limit()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_set_ptr_limit()` (See ***MAUI Programming Reference Manual***)
`GFX_POINT` (See ***MAUI Programming Reference Manual***)
`MSG_COMMON_MPCMD`
`MSG_GET_DEV_STATUS`

MSG_SET_PTR_POS

Sets Pointer Position

```
#include <mppm.h>
```

Syntax Command Structure

```
typedef struct _MSG_SET_PTR_POS
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_PTR_POS */
   GFX_POINT position;/* New position */
} MSG_SET_PTR_POS;
```

Syntax Reply Structure

```
typedef struct _MSG_SET_PTR_POS_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_SET_PTR_POS_REPLY */
   error_code error;/* return error code */
} MSG_SET_PTR_POS_REPLY;
```

Description

This message is passed directly to the protocol module via `mppm_process_msg()` when an application calls `inp_set_ptr_pos()`.

The protocol module is responsible for formatting the reply.

Direct Errors

None

Indirect Errors

`mppm_process_msg()`

See Also

`inp_set_ptr_pos()` (See ***MAUI Programming Reference Manual***)
`GFX_POINT` (See ***MAUI Programming Reference Manual***)
`MSG_COMMON_MPCMD`
`MSG_GET_DEV_STATUS`

MSG_BADACK_REPLY

Replies to Bad Messages

```
#include <mppm.h>
```

Syntax Reply Structure

```
typedef struct _MSG_BADACK_REPLY
{
   MSG_COMMON_MPCMD dcom;/* dcom.cmd = */
                /* CMD_BADACK_REPLY */
   error_code error;/* return error code */
} MSG_BADACK_REPLY;
```

Description

This message is returned by a protocol module with an error code of `EOS_MAUI_BADACK` when the protocol module does not understand the command code.

Direct Errors

`EOS_MAUI_BADACK` returned when a command code was not understood.

Indirect Errors

None

See Also

`MSG_COMMON_MPCMD`

Message Data reference

This section gives a detailed reference for each of the data types in `mp.h`.

Table 3-6 Message Data Reference Structures

Structure	Type	Description
MP_DEV_CMD	Enumerated	Message command codes
MP_DEV_ID	Data	Input Device ID
MP_DEV_MSG	Data Structure	Union of all messages
MP_MBOX_NAME	Defined Constant	Name of the `maui_inp` command mailbox
MP_MBOX_REPLY_NAME	Defined Constant	Format string for reply mailbox
MP_PROC_ID	Data	Input Process ID
MSG_COMMON_MPCMD	Data Structure	Common section of all messages
MSG_TYPE_MPCMD	Defined Constant	Message type code

MP_DEV_CMD

Message Command Codes

Syntax

```
#include <mppm.h>
typedef enum
{
    CMD_INP_INIT,                 /* Register a process */
    CMD_INP_INIT_REPLY,           /* Reply to CMD_INP_INIT */
    CMD_INP_TERM,                 /* Un-register a process */
    CMD_INP_TERM_REPLY,           /* Reply to CMD_INP_TERM */
    CMD_OPEN_DEV,                 /* Open an input device */
    CMD_OPEN_DEV_REPLY,           /* Reply to CMD_OPEN_DEV */
    CMD_CLOSE_DEV,                /* Close an input device */
    CMD_CLOSE_DEV_REPLY,          /* Reply to CMD_CLOSE_DEV */
    CMD_RESTACK_DEV,              /* Re-stack an input device */
    CMD_RESTAC_DEV_REPLY,         /* Replay to CMD_RESTACK_DEV */
    CMD_SET_MSG_MASK,             /* Set message write mask */
    CMD_SET_MSG_MASK_REPLY,       /* Rply 2CMD_SET_MSG_MASK */
    CMD_CHECK_KEYS,               /* Check if keys exist */
    CMD_CHECK_KEYS_REPLY,         /* Reply to CMD_CHECK_KEYS */
    CMD_GET_DEV_CAP,              /* Get device capabilities */
    CMD_GET_DEV_CAP_REPLY,        /* Reply 2CMD_GET_DEV_CAP */
    CMD_GET_DEV_STATUS,           /* Get device status */
    CMD_GET_DEV_STATUS_REPLY,     /* Reply to CMD_GET_DEV_STATUS */
    CMD_RELEASE_KEY,              /* Release a reserved key */
    CMD_RELEASE_KEY_REPLY,        /* Rply 2 CMD_RELEASE_KEY */
    CMD_RESERVE_KEY,              /* Reserve key for process */
    CMD_RESERVE_KEY_REPLY,        /* Rply 2 CMD_RESERVE_KEY */
    CMD_SET_MSG_CALLBACK,         /* Set message callback */
    CMD_SET_MSG_CALLBACK_REPLY,   /* Reply to CMD_SET_MSG_CALLBACK */
    CMD_SET_SIM_METH,             /* Set pointer sim mode */
    CMD_SET_SIM_METH_REPLY,       /* Reply to CMD_SET_CURSOR_SIM */
    CMD_SET_PTR_POS,              /* Set pointer position */
    CMD_SET_PTR_POS_REPLY,        /* Reply 2 CMD_SET_PTR_POS */
    CMD_SET_PTR_LIMIT,            /* Set pointer limit */
    CMD_SET_PTR_LIMIT_REPLY,      /* Rply CMD_SET_PTR_LIMIT */
    CMD_BADACK_REPLY,             /* Reply to bad messages */
} MP_DEV_CMD;
```

Description

This enumerated type defines the device message command codes.

MP_DEV_ID

Input Device ID

Syntax

```
#include <mppm.h>
typedef void * MP_DEV_ID;
```

Description

This data type defines a caller process ID and is returned in `MSG_OPEN_DEV_REPLY`.

See Also

`MSG_OPEN_DEV` (See ***MAUI Programming Reference Manual***)

MP_DEV_MSG

Union of All Messages

Syntax

```
#include <mppm.h>
typedef union _DEV_MSG
{
    MSG_INP_INIT              inp_init;
    MSG_INP_INIT_REPLY        inp_init_reply;
    MSG_INP_TERM              inp_term;
    MSG_INP_TERM_REPLY        inp_term_reply;
    MSG_OPEN_DEV              open_dev;
    MSG_OPEN_DEV_REPLY        open_dev_reply;
    MSG_CLOSE_DEV             close_dev;
    MSG_CLOSE_DEV_REPLY       close_dev_reply;
    MSG_RESTACK_DEV           msg_restack_dev;
    MSG_RESTACK_DEV_REPLY     msg_restack_dev_reply;
    MSG_SET_MSG_MASK          set_msg_mask;
    MSG_SET_MSG_MASK_REPLY    set_msg_mask_reply;
    MSG_GET_DEV_CAP           get_dev_cap;
    MSG_GET_DEV_CAP_REPLY     get_dev_cap_reply;
    MSG_GET_DEV_STATUS        get_dev_status;
    MSG_GET_DEV_STATUS_REPLY  get_dev_status_reply;
    MSG_SET_PTR_POS           set_ptr_pos;
    MSG_SET_PTR_POS_REPLY     set_ptr_pos_reply;
    MSG_SET_SIM_METH          set_cursor_sim;
    MSG_SET_SIM_METH_REPLY    set_cursor_sim_reply;
    MSG_SET_PTR_LIMIT         set_ptr_limit;
    MSG_SET_PTR_LIMIT_REPLY   set_ptr_limit_reply;
    MSG_RESERVE_KEY           reserve_key;
    MSG_RESERVE_KEY_REPLY     reserve_key_reply;
    MSG_RELEASE_KEY           release_key;
    MSG_RELEASE_KEY_REPLY     release_key_reply;
    MSG_CHECK_KEYS            check_keys;
    MSG_CHECK_KEYS_REPLY      check_keys_reply;
    MSG_SET_MSG_CALLBACK      set_msg_callback;
    MSG_SET_MSG_CALLBACK_REPLY set_msg_callback_reply;
    MSG_BADACK_REPLY          badack_reply;
    MSG_COMMON_MPCMD          any;
} MP_DEV_MSG;
```

Description

This union defines a generic reference to all input device command messages. See the message reference for details of these messages.

MP_MBOX_NAME

Name of maui_inp's Command Mailbox

Syntax

```
#include <mppm.h>
MP_MBOX_NAME
```

Description

This constant defines the name of the `maui_inp` command mailbox.

MP_MBOX_REPLY_NAME

Format String for Reply Mailbox

Syntax

```
#include <mppm.h>
MP_MBOX_REPLY_NAME
```

Description

This constant defines the format string for the reply mailbox.

MP_PROC_ID

Inputs Process ID

Syntax

```
#include <mppm.h>
typedef void * MP_PROC_ID;
```

Description

This data type defines a caller process ID and is returned in `MSG_INP_INIT_REPLY`.

See Also

`MSG_INP_INIT` (See ***MAUI Programming Reference Manual***)

MSG_COMMON_MPCMD

Common Section of Control Messages

Syntax

```
#include <mppm.h>
typedef struct _MSG_COMMON_MPCMD
{
   MSG_COMMON com;      /* common section of all */
                        /* messages */
   MP_DEV_CMD cmd;      /* Command code of message */
   MP_DEV_ID dev_id;    /* ID of device */
} MSG_COMMON_MPCMD;
```

Description

This data structure defines the common header at the beginning of all command and reply messages. A message must have this header to be understood by `maui_inp` and its protocol modules.

See Also

`MP_DEV_CMD`
`MP_DEV_ID`
`MSG_COMMON` (See ***MAUI Programming Reference Manual***)

MSG_TYPE_MPCMD

Message Type Code

Syntax

```
#include <mppm.h>
MSG_TYPE_MPCMD
```

Description

This constant defines the type code for all command messages.

Chapter 4: Sound Driver

MAUI sound drivers enable applications to operate independent of hardware differences in target systems. This chapter explains the sound device capabilities, the relationship between the file manager, sound driver, and descriptors, and how to build, modify, and verify your drivers.

Overview of Sound Driver Interface

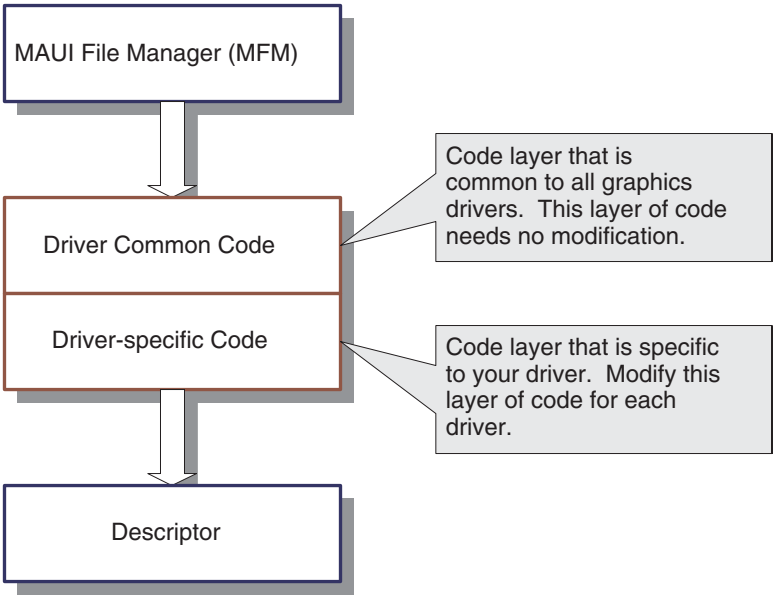
The Sound Driver Interface:

- provides a set of primary entry points, GetStat sub-functions, and SetStat sub-functions through which MAUI applications can control the sound driver.
- is a dual-ported I/O (DPIO) driver that uses the multimedia file manager (MFM). This allows the driver to work under both OS-9 and OS-9000.

The sound driver is sharable (multiple paths may be open to it at the same time). This enables multiple play and record paths, but not concurrent play and record. The Sound Driver Interface is accessible by MAUI applications and directly controls the operation of the sound driver

MAUI sound drivers interface between the sound device and the MAUI File Manager. The sound driver contains all device-specific code so that MAUI applications and the MAUI APIs can operate independent of the hardware in any system. The following figure shows the relationship between the file manager, sound driver, and descriptor:

Figure 4-1 MFM, Driver, Descriptor Relationship



The sound device driver consists of a common code layer and a device-specific code layer. All sound drivers share the same set of common code, which provides functions and definitions needed by all drivers. Some of the common code is conditional to allow individual customization of each port. The device-specific code handles all the functions and definitions unique to each device. When porting a sound driver, modify the device-specific code in the example drivers to reflect the sound device in your system.

The device descriptor is the handle used by applications to reference a device. The descriptor indicates the file manager, driver, and the driver's initialization data required to access the device.

Device Capabilities

One important function of your device driver is identifying the capabilities of the device. Sounds device capabilities are defined in a set of data structures within the `global.h` file. A specification is particularly valuable when writing your `global.h` file.

For More Information

Look in the directory holding the sample drivers for an example of a written specification for the sample sound driver included with MAUI.

Driver Code

MAUI sound drivers consist of two types of code: common code that is already written for your driver, and device-specific code that you write. The common code makes up a large portion of the sound driver.

The simplest and most successful method of developing device-specific code when porting a sound driver is to modify the device-specific source code in an existing sample driver. Your modifications reflect the capabilities and requirements of the sound device in your system.

The device-specific code consists of a number of files, of which some are required and others are optional, depending on your system. The following files are required in every sound driver, although it is possible to change their names:

- `abort.c` contains hardware specific code, if any, to abort a play or record.
- `config.h` contains the definition that controls the configuration of the driver including the names of functions defined by the device-specific code.
- `cont.c` contains hardware specific code to continue a play or record after a pause.
- `drvr.tpl` `os9make` “include” file.
- `gain.c` contains hardware specific code to modify both input and output gain.
- `global.h` contains the global definitions for the driver including device capabilities and prototypes.
- `hardware.c` defines functions that deal directly with the hardware device such as `init` and `term`, and any register modification.
- `hardware.h` contains hardware-specific definitions.
- `irq.c` contains interrupt service functions.
- `path.c` contains hardware specific code, if any, for opening and closing a device.

- `pause.c` contains hardware specific code to pause a play or record.
- `signal.c` contains hardware specific code, if any, for the `_os_ss_sendsig()` functionality.
- `static.h` contains the definitions for static storage areas available to the driver.

The following files may be included or excluded depending on whether play and/or record functionality is needed. These files are:

- `play.c` contains hardware specific code to play sound samples.
- `record.c` contains hardware specific code to record sound samples.

When modifying the driver code, you should organize your work to modify the files in this order:

1. Modify the header files; `config.h`, `global.h`, `static.h`, `hardware.h`, and `mfm_desc.h`.
2. Modify `hardware.c` to access the device hardware.
3. Modify `play.c`, `record.c`, and `irq.c` as required.
4. Modify `gain.c`, `abort.c`, `pause.c`, and `cont.c` as required.
5. Update `drvr.tpl` to reflect any file name changes.

Device-specific Code

This provides specific details for modifying the device-specific code. Within each of these files, some functions are required and some are optional.

Sample driver files are located in the directory:

`MWOS/SRC/DPIO/MFM/DRVR/SD_SAMP`

You can use these files as templates for building your own device-specific code.

Where the Files are Located

MAUI sound driver source is delivered with one directory of sample files and one complete example driver. You may either modify the example driver or the sample files to make your driver. The sample files contain instructions for building your own .h and .c files.

- MAUI Standard header files are located in
  `MWOS/SRC/DEFS/MAUI`

WARNING

These header files should never be modified by the user

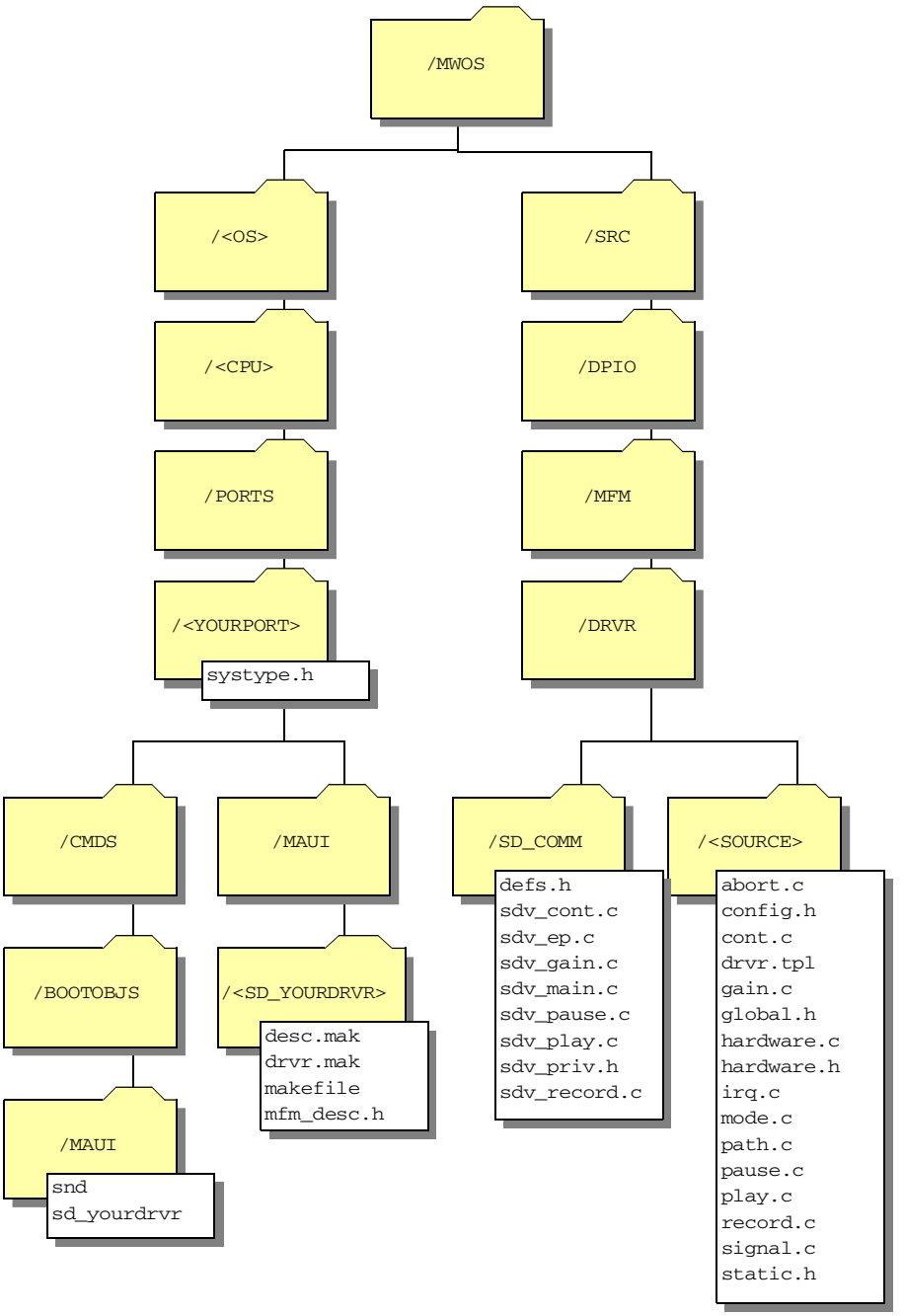
- MAUI common sound driver code is located in:
  `MWOS/SRC/DPIO/MFM/DRVR/SD_COMM`
  This directory is referred to as *common* throughout this chapter. Normally you should not need to modify files in this directory. If your implementation does have special requirements that necessitates modifying the common code, make a copy of the relevant file(s) to your driver specific directory and make your modifications there.
- MAUI example driver source is located in:
  `MWOS/SRC/DPIO/MFM/DRVR/SD_CS`
- The sample driver template files are located in:
  `MWOS/SRC/DPIO/MFM/DRVR/SD_SAMP`

How to Port your Sound Driver

Create the directory structure for your port

Before beginning to port your sound driver, you must create a directory structure to store your new files. That structure is shown in **Figure 4-2 Directory Structure for Your Sound Driver Port**

Figure 4-2 Directory Structure for Your Sound Driver Port



Step 1. Define and create a source directory. This directory is referred to in this chapter as *SOURCE* and assumes the pathname:
MWOS/SRC/DPIO/MFM/DRVR/*SD_YOURDRVR*
It is recommended that the directory name start with “SD_” followed by an uppercase descriptive name for your driver.

Step 2. Copy all of the files from
MWOS/SRC/DPIO/MFM/DRVR/SD_SAMP
into your new *SOURCE* directory. Verify that the following files are now in your *SOURCE* directory:

```
abort.c
config.h
cont.c
drvr.tpl
gain.c
global.h
hardware.c
hardware.h
irq.c
path.c
pause.c
play.c
record.c
signal.c
static.h
```

Step 3. Define and create a ports directory. This directory is referred to in this chapter as *YOURPORT* and assumes the pathname:
MWOS/OS/CPU/PORTS/*YOURPORT*

Step 4. Define and create a make directory. This directory is referred to in this chapter as *MAKE* and assumes the pathname:
MWOS/OS/CPU/PORTS/*YOURPORT*/MAUI/*SD_YOURDRVR*
We recommend that this name match the one in step 1.

Step 5. Copy or create the following files in *SD_YOURDRVR*

```
desc.mak        drvr.mak
makefile        mfm_desc.h
```

Here are examples of these files:

- `desc.mak` Make the MAUI sound descriptor

```
# Makefile
#*****************************************************************************
#   This makefile will make the MAUI Sound descriptors#
#*****************************************************************************
#* Copyright 1996 by Microware Systems Corporation                           **
#* Copyright 2001 by RadiSys Corporation                                     **
#* Reproduced Under License                                                  **
#*                                                                           **
#* This source code is the proprietary confidential property of              **
#* Microware Systems Corporation, and is provided to licensee                **
#* solely for documentation and educational purposes. Reproduction,          **
#* publication, or distribution in any form to any party other than          **
#* the licensee is strictly prohibited.                                      **
#*****************************************************************************

#### Add New Descriptor Names Here ##########################################
#                                   #
TRGTS   =   snd snd10
DRVR    =   SD_YOURDRVR
#                                   #
#############################################################################

PORT    =   ../..
MAKENAME=desc.mak
include $(PORT)/../make.com

RDIR    =   RELS/DESC
ODIR    =   $(PORT)/CMDS/BOOTOBJS/MAUI
SDIR    =   $(MWOS)/SRC/DPIO/MFM/DESC
COMMDIR =    SD_COMM
DESCDIR =   .

include $(SDIR)/snddesc.tpl

_purge _clean:  nulltrg
    $(CODO) $(ODIR)/snd
    -$(DEL) $(ODIR)/snd
    $(CODO) $(ODIR)/snd10
    -$(DEL) $(ODIR)/snd10

#                                   #
#############################################################################
```

- `drvr.mak` Make the MAUI sound driver

```
# Makefile
#*****************************************************************************
#* Makefile for Maui CS4231 Driver
#*****************************************************************************
#* Copyright 1996 by Microware Systems Corporation                           **
#* Copyright 2001 by RadiSys Corporation                                     **
#* Reproduced Under License                                                  **
#*                                                                           **
#* This source code is the proprietary confidential property of              **
#* Microware Systems Corporation, and is provided to licensee                **
#* solely for documentation and educational purposes. Reproduction,          **
#* publication, or distribution in any form to any party other than          **
#* the licensee is strictly prohibited.                                      **
#*****************************************************************************


#### Put Driver Names and Options Here ########################################
#                                   #
TRGTS   =   sd_yourdrvr
DRVR    =   SD_YOURDRVR

#DEBUG  =   -g
DEBUG   =

#                                   #
###############################################################################

PORT    =   ../..
MAKENAME=drvr.mak
include $(PORT)/../make.com

ODIR    =   $(PORT)/CMDS/BOOTOBJS/MAUI
RDIR    =   RELS/DRVR
IDIR    =   $(RDIR)/$(HOSTTYPE)
DESCDIR =   .

# Place user defines here. See the bottom of config.h in the driver
# source directory for the list of defines value for this driver

USR_DEFINES = -dPIO -dXCTL_CONTROL -dPCI_I82378

include $(MWOS)/SRC/DPIO/MFM/DRVR/$(DRVR)/drvr.tpl

#                                   #
###############################################################################
```

- `makefile` Make the MAUI sound descriptor and the MAUI sound driver

```
# Makefile
#*****************************************************************************
#* Call makefiles to build sound driver and descriptor
#*****************************************************************************
#* Copyright 1997 by Microware Systems Corporation
#* Copyright 2001 by RadiSys Corporation
#* Reproduced Under License
#*
#* This source code is the proprietary confidential property of
#* Microware Systems Corporation, and is provided to licensee
#* solely for documentation and educational purposes. Reproduction,
#* publication, or distribution in any form to any party other than
#* the licensee is strictly prohibited.
#*****************************************************************************


MWOS    =   ../../../../../..
TRGTS   =   desc.mak drvr.mak
ALL_TRGTS=p603
MAKENAME=makefile
include $(MWOS)/MAKETMPL/makesub.com


$(TRGTS):  notarget
    $(MAKE) -f $@ $(MAKEOPTS) TARGET=$(TARGET) $(SUBTRGT)


notarget:  .
    $(COMMENT)

#                                   #
#####################################################################################
```

- `mfm_desc.h` The MAUI sound descriptor header file

```
/*****************************************************************************
**
* FILENAME : mfm_desc.h
*
* DESCRIPTION :
*
*   This file contains definitions for the MAUI device descriptors.
*
* COPYRIGHT:
*
*   This source code is the proprietary confidential property of Microware
*   Systems Corporation, and is provided to licensee solely for documentation
*   and educational purposes. Reproduction, publication, or distribution in
*   form to any party other than the licensee is strictly prohibited.
*
```

```
#ifndef _MFM_DESC_H
#define _MFM_DESC_H

#include "../../systype.h"

#define I82378_NCFG_ADDR    ISA_IOBASE          /* Non-Configured I82378 base */
#define BOARD_CFG_REG_ADDR  BOARD_CFG_REG       /* Board Configuration register
*/

/*****************************************************************************
**
 * CS4231A Sound Descriptor

*****************************************************************************
*/
#if defined(MFM_DESC) && (defined(snd) || defined(snd10))

#define SHARE               TRUE        /* Path sharing flag */
#define LUN                 1           /* Logical unit number */
#define PORTADDR            (ISA_IOBASE+0x830) /* Base address of hardware */
#define MODE                S_IREAD | S_IWRITE

#define DRV_NAME            "sd_cs"

/* for SD_COMM/defs.h */
#define SDV_HW_SUBTYPE      CS4231A     /* Hardware sub-type */
#define SDV_HW_SUBNAME      "CS4231A"   /* Hardware sub-type name */

#if defined(snd)
#define SDV_IRQ_NUM         5           /* IRQ number */
#else /*snd10 */
#define SDV_IRQ_NUM         10          /* IRQ number */
#endif

#define SDV_IRQ_PRIORITY    5           /* IRQ priority */
#define SDV_DMA_PLAY_CHAN   6           /* DMA Channel for Playback */
#define SDV_DMA_RECORD_CHAN 7           /* DMA Channel for Capture */

/* for SD_CS/static.h */
#define SDV_TRANSFER_SIZE   (500*64)    /* This is the Maximum transfer size -
make divisable by 16 */

#endif /* MFM_DESC_SND */


#endif /* MFM_DESC_SND */

#endif /* _MFM_DESC_H_ */
```

Step 6. Define and create the directory *YOURPORT*/CMDS/BOOTOBJS/MAUI. During the make process two object files are created and stored in MAUI:

- snd — descriptor object.
- sd_*yourdrvr* — driver object. This is typically a lower case version of the directory name in step 1.

Step 7. Verify your directory structure contains the correct files as shown in **Figure 4-2 Directory Structure for Your Sound Driver Port**.

Common Code Source Files

The following files are located in the SD_COMM directory:

- defs.h — primary definition file which ties together all the other definition files.
- sdv_abort.c — common abort functions.
- sdv_cont.c — common continue functions.
- sdv_ep.c — entry point functions.
- sdv_gain.c — common gain functions.
- sdv_main.c — main function for the driver.
- sdv_pause.c — common pause functions.
- sdv_play.c — common play functions.
- sdv_priv.h — definitions private to the common code.
- sdv_record.c — common record functions.

Device-specific Source Files

The following files are located in the *SOURCE* directory:

- abort.c — abort play or record function.
- config.h — configures the capabilities of the driver and inclusion/exclusion of common code.
- cont.c — continue play or record functions.

- `drvr.tpl` `os9make` “include” file.
- `gain.c` gain control functions.
- `global.h` global definitions.
- `hardware.c` hardware function.
- `hardware.h`* definitions for hardware functions.
- `irq.c`† interrupt service functions.
- `path.c` hardware Open and Close functions.
- `pause.c` pause play or record functions.
- `play.c`‡ optional hardware play functions.
- `record.c`** optional hardware record functions.
- `signal.c` hardware sendsig and release functions.
- `static.h` fefinitions for static storage areas.

Modify the *SOURCE* files you need

Step 1. Delete the optional files in your *`SOURCE`* directory, if your driver does not support the corresponding function.

Step 2. Update your `drvr.tpl` to reflect the deletions, if any, made in step 1.

*. The example `sd_cs` sound driver has additional hardware definition files that are not depicted in this list. These could have gone into `hardware.h` instead, but were included by `hardware.h` to preserve their original integrity. If you make similar extensions, remember to update `drvr.tpl`.

†.Optional files. Delete these files if not supported by your driver.

‡.Optional files. Delete these files if not supported by your driver.

**.Optional files. Delete these files if not supported by your driver.

Modify the config.h file to reflect your system.

Step 1. Define function names for `HW_ABORT_PLAY` through `HW_TERM`. All values must be defined, although you normally use the default values. These are required functions.

Step 2. Define function names for `HW_INIT_IRQS` through `HW_RECORD_SET_MODE`. Only include definitions for functions supported by your driver. These are optional functions.

Modify the global.h file to reflect your system.

Step 1. Modify the initializer for the `sdv_gain_cap` array to match the gain capabilities of your driver.

Step 2. Modify the initializer for the `sdv_cm_info` array. Show the data structure with a detailed explanation of each member.

Step 3. Modify the initializer for the `sdv_sample_rates` array to include all supported sample rates.

Step 4. Modify the initializer for the `sdv_channel_info` array to include all supported number of channels.

Step 5. Modify the initializer for the `sdv_dev_cap` data structure. Show the data structure with a detailed explanation of each member

Step 6. Modify the initializer for the `sdv_status_gain` array. Include one entry for each device that is controllable on your system.

Step 7. Modify the initializer for the `sdv_mix_lines` array.

Step 8. Prototype the functions you need for device-specific code in the `PROTOTYPE` area. This area is used to prototype functions that must be visible to multiple device-specific files.

Modify the static.h file to define your static storage areas.

This task may be difficult to perform at this time because the variables that must be defined here may not be known yet. Make an attempt to define them now, and refine this file as the port proceeds.

Step 1. Modify `SDV_LU_SPECIFICS` with the variable names needed by the driver. This file is setup with the values in `SDV_LU_SPECIFICS_INIT` when the driver is initialized. This structure should include, but is not limited to the following:

- Address of groups or individual I/O registers.
- Place holders for shadow contents of I/O registers.

Step 2. Modify `SDV_LU_SPECIFICS_INIT` to include the values from the descriptor to compute the initializers. Minimize the number of definitions required in the descriptor. The objective there is to use a few definition in the descriptor to compute a larger number of entries in the `lustat`.

Modify the hardware.h file to reflect your system hardware definitions

Step 1. Modify `hardware.h` to include all necessary hardware related definitions.

Modify the hardware.c files to initialize your hardware

Step 1. Modify the `hardware.c` file with the following considerations:

- `hw_init()` is called when the device is initialized.
- `hw_term()` is called when the device is terminated.
- Be sure that `hw_term()` returns any resources allocated in `hw_init()`.

Modify the play.c, record.c, and irq.c files to support play and/or record

Step 1. Modify the `play.c` file with the following considerations:

- `hw_play_enable()` is called at the start of a play to enable the decoding of sound samples.
- `hw_play_disable()` is called at the conclusion of a play to disable the decoding of sound samples.
- `hw_play_set_mode()` is called to set the hardware mode and IRQ handler for the sound data in the sound map.

Step 2. Modify the `record.c` file with the following considerations:

- `hw_record_enable()` is called at the start of a record to enable the encoding of sound samples.
- `hw_record_disable()` is called at the conclusion of a record to disable the encoding of sound samples.
- `hw_record_set_mode()` is called to set the hardware mode and IRQ handler for the sound data in the sound map.

Step 3. Modify the `irq.c` file with the following considerations:

- `hw_init_irqs()` must be modified to enable interrupts.
- `hw_term_irqs()` must be modified to disable interrupts.
- `hw_isr()` is the entry point for all sound driver interrupts. This function must be modified to read the appropriate status register, act on the interrupt, and clear the interrupt.

Modify the remaining control functions

Step 1. Modify the following control functions only if they are supported by your hardware. If they are not supported, delete the source file and point the function in `config.h` at a function that simply returns `EOS_UNKSVC`. Make sure that the inclusion or exclusion of these files are represented in the device capabilities `(global.h)` and the makefile template `(drvr.tpl)`.

- `abort.c` contains the hardware specific code to abort a play or record.
- `cont.c` contains the hardware specific code to continue a play or record after a pause.
- `gain.c` contains the hardware specific code to modify both input and output gain.
- `pause.c` contains the hardware specific code to pause a play or record. This driver simply stops the timer to pause both play and record.

Modify the remaining device-specific functions

Step 1. Modify the following device-specific functions only if your hardware has specific requirements. These functions are normally handled completely in the sound driver common code. Normally the these device-specific functions simply return `SUCCESS`.

- `signal.c` contains any, if any, hardware specific code for enabling and disabling “send signal on device idle”. Normally this is not necessary.
- `path.c` contains any, if any, hardware specific code to be called when the device is opened or closed. normally this is not necessary.

How to Build your Sound Driver

Step 1. Change directories to *SD_YOURDRVR* directory:

```
cd MWOS/OS/CPU/PORTS/YOURPORT/MAUI/SD_YOURDRVR
```

Step 2. To make both the sound driver and descriptor, type

```
os9make
```

The `makefile` invokes `.desc.mak` and `drvr.mak`.

To make the sound descriptor, type:

```
os9make -f desc.mak
```

The `desc.mak` makefile builds the sound descriptor and places it in the directory *YOURPORT*`/CMDS/BOOTOBJS/MAUI`.

To make the sound driver, type:

```
os9make -f drvr.mak
```

The `drvr.mak` builds the sound driver and places it in the directory *YOURPORT*`/CMDS/BOOTOBJS/MAUI`.

How to Test your Driver

Run the sound demo programs included with MAUI to test your driver. Demo program source is located in the following directory:

`MWOS/SRC/MAUI/DEMOS/SND`

Demo program objects are located in the following directory:

`MWOS/OS/CPU/CMDS/MAUIDEMO`

There are two demo programs:

- `auplay` attempts to play `.au` and `.wav` sound files. Success depends on the capabilities of the sound hardware. `auplay` has many options. Execute `auplay` with a parameter of `-?` or `-h` to get on-line help.
- `aurecord` attempts to record `.au` and `.wav` sound files. Success depends on the capabilities of the sound hardware. `aurecord` has many options. Execute `aurecord` with a parameter of `-?` or `-h` to get on-line help.

These are not complete tests, but should enable you to verify basic functions.

Chapter 5: How to Configure a System for MAUI

This chapter describes how to configure a MAUI enabled system. It includes the following sections:

- **Overview of MAUI Object Modules**
- **Selecting a MAUI System Driver**
- **Using the Configuration Wizard for MAUI**
- **Advanced Wizard Configuration**

Overview of MAUI Object Modules

MAUI is highly modular and configurable, enabling system developers to make design decisions that trade off between size, speed, and functionality. This section describes the objects that make up MAUI and the considerations for configuring a MAUI enabled system.

Common MAUI modules

- `MWOS/OS/CPU/CMDS/BOOTOBJS/mfm`

  MAUI File Manager. Required for the `CDB`, `MSG`, `SND`, and `GFX` APIs.

- `MWOS/OS/CPU/CMDS/BOOTOBJS/mauidev`

  MAUI Device Descriptor. Required for the `CDB` and `MSG` APIs.

- `MWOS/`*`OS`*`/`*`CPU`*`/CMDS/BOOTOBJS/mauidrvr`, `MWOS/`*`OS`*`/`*`CPU`*`/CMDS/BOOTOBJS/mauidrvr_lock`, or `MWOS/`*`OS`*`/`*`CPU`*`/CMDS/BOOTOBJS/mauidrvr_filter`

  MAUI System Driver. Required for the `CDB` and `MSG` APIs. There are three different versions of this driver. Each has the same module name but different file names. See the **Selecting a MAUI System Driver** section for a full description of each:

  `mauidrvr` - Default/recommended version. The smallest, fastest, most secure version of the three. The mailbox format is not run-time compatible with the other two versions.

  `mauidrvr_lock` - Supports queue locks that are compatible with old statically linked MAUI `MSG` applications.

  `mauidrvr_filter` - Supports queue locks and the deprecated `msg_set_filter()` call. This is the largest, slowest, least secure version of the three.

- `MWOS/OS/CPU/CMDS/BOOTOBJS/maui_inp`, `MWOS/OS/CPU/CMDS/BOOTOBJS/MON/maui_inp`, or `MWOS/OS/CPU/CMDS/BOOTOBJS/MON/maui_inl`

Input daemon. Required by the INP API. Only include this module on the system if INP or WIN API support is required. The Input daemon uses the MSG API, so it requires mfm, mauidev, and mauidrvr. There are three different versions of the Input daemon:

maui_inp - Default version. Requires maui shared library module

MON/maui_inp - Debug version. Requires maui shared library module. Includes a command line option to print status and debug information.

MON/maui_inl - Statically linked debug version. Does not require the maui shared library module.

- MWOS/OS/CPU/CMDS/BOOTOBJS/maui_win or MWOS/OS/CPU/CMDS/BOOTOBJS/MON/maui_win

  Window daemon. Required by the WIN API. Only include this module on the system if WIN API support is required. The Window daemon uses the INP and MSG APIs, so it requires mfm, mauidev, mauidrvr, and maui_inp. There are two versions of the Window daemon:

  maui_win - Default version. Requires maui Shared Library module.

  MON/maui_win - Debug version. Requires maui Shared Library module. Includes code to print debug messages.

- MWOS/OS/CPU/CMDS/maui or MWOS/OS/CPU/CMDS/mt_maui

  MAUI Shared Library module. This module is normally present on MAUI systems, but is not required if all MAUI applications link to the static MAUI libraries, mauilib.l/mauilib.il, instead of maui.l/maui.il. While the MAUI Shared Library module is large, linking all MAUI applications against mauilib.l/mauilib.il instead of maui.l/maui.il makes each of those applications larger (including any required daemons such as maui_win and maui_inp). Sometimes this results in an even larger footprint. In addition, statically linking MAUI applications does not provide as

much compatibility with future versions of MAUI.
There are two versions of the MAUI Shared Library. Each has the same module name but different file names:

`maui` - Non-Threaded version. This is smaller and faster, but does not supported “connections” from threaded MAUI applications.

`mt_maui` - Threaded version. This version can accept “connections” from both threaded and non-threaded MAUI applications.

Port-Specific Objects

The port-specific module names described below are common, but not absolute or fully inclusive.

For More Information

Port-specific MAUI modules are described in each appropriate OS-9 Board Guide.

Configuration Description Blocks

There may be one or more Configuration Description Block (CDB) modules on a system. Via the `CDB` API, they appear to the application as a single Configuration Description Block string. CDB modules can have any name, since what defines them as CDB modules is a module type/attribute of 0x501. The primary module is usually (but not always) called `cdb`. This module contains the `CDB_TYPE_SYSTEM` entry and any devices that are in all board configurations.

Other devices can be listed in separate CDB modules so they can be easily added or removed from a system. For example, you might create separate CDB entries for the mouse and touch screen, and then configure the wizard to load these along with the descriptor, drivers, and protocol modules as appropriate.

Graphics Devices

Graphic device modules typically consist of a descriptor and driver. A common descriptor name is `gfx`, but `osd`, `vga` and `lcd` are often used as well. By convention, graphics drivers have a prefix of `gx_`, followed by a name descriptive of the device. Graphic devices use the `mfm` file manager.

MAUI supports multiple graphics devices on a system. It is also possible that more than one descriptor may exist in memory for a particular graphics device. The various descriptors might select different default resolutions or specify some other configurable attribute of the device.

Sound Devices

Sound device modules typically consist of a descriptor and driver. The most common descriptor name is `snd`. By convention, sound drivers have a prefix of `sd_`, followed by a name descriptive of the device. Sound devices use the `mfm` file manager.

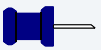
MAUI supports multiple sound devices on a system. It is also possible that more than one descriptor may exist in memory for a particular sound device. The various descriptors might select different defaults or specify some other configurable attribute of the device.

Input Devices

Input devices consist of a MAUI Input Protocol Module and an associated device descriptor and driver. In the configuration wizard, sometimes the input device descriptor and driver are configured as part of the “core OS”, such as with serial ports. For input devices that are solely used by MAUI (e.g. a touch screen), they are configured within the MAUI screens of the Wizard.

The MAUI Input devices use the MAUI Input Process (`maui_inp`) to read a “normal” OS-9 input source, such as a serial port. `maui_inp` is not affected by what file manager or driver supplies the data as long as it supports `_os_open()`, `_os_close()`, `_os_gs_ready()`, `_os_sendsig()`, and `_os_read()`. Some protocol modules may require additional setstat/getstat support.

Like MAUI Graphic Drivers, the names of MAUI Input Protocol Modules are not fixed. By convention, they have a prefix of `mp_`, followed by a name indicating the protocol they support. Microware provides the following standard MAUI Input Protocol Modules.



Note

Not all MAUI Input Protocol Modules are included in all packages.

`mp_bsptr`	Three Button Bus/PS2 Mouse
`mp_hamp`	Hampton Communications Touch Screen Format
`mp_keyptr`	Example Combination Key/Pointer Device
`mp_kybrd`	Generic VT100 Serial Keyboard
`mp_msptr`	Two Button Serial Mouse
`mp_phptr`	Touch Screen CD-i Mouse
`mp_phrem`	Philips Remote Control - In some OEM packages as a source code example.
`mp_pskbd`	Raw PS2 Keyboard
`mp_sakpad`	StrongArm/SideKick 16 Button Numeric Keypad Example
`mp_ssptr`	SmartSet Touch Screen Controllers by Elo Touch Systems, Inc.
`mp_t328ads`	Motorola MC68328ADS Touchpad
`mp_ucb1200`	UCB1200/1300 Touch Screen

mp_usbkbd	USB Keyboard
mp_xtkbd	Generic XT/Scan Code Keyboard

Demo Objects

Demo objects include examples, demos, and their assets. The demo objects described below are located in the following directory:

MWOS/OS/CPU/CMDS/MAUIDEMOS

Their sources are located below MWOS/SRC/MAUI/DEMOS. All of these demos use the MAUI Shared Library module (maui or mt_maui). Readme files located with the demo sources indicate other dependencies.

aloha	Text and Input demo. Requires an input and graphics device. maui_inp must be running. Uses MWOS/OS/CPU/ASSETS/FONTS/default.fnt.
auplay	Playback AU and WAV sound files. Requires a sound device capable of playback.
aurecord	Record AU and WAV sound files. Requires a sound device capable of recording.
fcopy	Graphic copying demo. Uses the IFF images fun.*, mwlogo.* and travel.* from MWOS/OS/CPU/ASSETS/ASSETS/IMAGES, where "*" indicates the bit depth of the display. Requires a graphics device.
fdraw	Graphic drawing demo. Requires a graphics device.
gxdevcap	Print graphic device information. Requires a graphics device.

`hello`	Graphic text demo. Requires a graphics device. Uses `MWOS/OS/CPU/ASSETS/FONTS/default.fnt`.
`inp`	Input demo. Requires an input device. `maui_inp` must be running.
`jview`	Display JPEG images. Requires a graphics device. One or more JPEG images.
`msginfo`	Display information about a MAUI mailbox. Requires the `mauidev/mauidrvr`.
`msgwrtr` and `msgrdr`	Messaging example. `msgwrtr` creates a mailbox, forks `msgrdr` then starts writing messages to the mailbox. `msgrdr` opens the same mailbox and starts reading messages. After a specified number of messages, `msgwrtr` sends a “done” message and waits for `msgrdr` to quit. `msgrdr` quits when it sees a “done” message. This demo has no hardware dependencies, it should be able to run on any OS-9 system. Requires the `mauidev/mauidrvr`.
`sfont`	Display a UCM font (defaults to `default.fnt`). Requires a graphics device.
`showimg`	Display IFF image. Requires a graphics device.
`windraw`	`WIN` API Block Drawing Demo. Requires the demo `winmgr` daemon running on the system.
`winink`	Window Pen/Inking Drawing Demo. Requires the demo `winmgr` daemon running on the system.

`winmgr`	Demo Window Manager Daemon. Requires an input and graphics device. The `maui_win` module must be in memory and `maui_inp` must be running.

Selecting a MAUI System Driver

MAUI Version 3.1 or greater includes the MAUI System Driver, which implements the MSG and CDB functionality.

MSG Support

To close a stability hole in the original design of MAUI Messaging (MSG), MAUI 3.1 re-implemented messaging as a system state service via a driver. By making messaging part of the OS-9 IO system, MAUI can now detect abnormal application termination via SS_CLOSE. This enables automatic cleanup of mailboxes opened by the application. In addition, the MSG API was extended to enable applications to request notification when other applications, which use the MSG API, terminate. For example, maui_inp can now determine if an application using the INP API, which uses the MSG API, quits without calling inp_term(), which calls msg_term(). This allows the application to return memory allocated on behalf of that application.

There is additional overhead to call a driver instead of directly running the message code. This can be partially offset by the driver making faster system calls from system state. To do this, however, the driver must execute entirely in system state, which interferes with the continued support of the msg_set_filter().

CDB Support

MAUI Version 3.1 moved the CDB from user state application code to the MAUI System Driver. This corrected a hole in the original design that allowed small windows where the systems module directory could change while the application searched for CDB modules. This also resulted in a speed improvement because of fewer system calls and less copying of system state structures for accessing in user state.

All versions of the MAUI System Driver have the same CDB implementations.

MAUI System Driver Versions

MAUI provides the following versions of the MAUI system driver:

- `mauidrvr`
- `mauidrvr_lock`
- `mauidrvr_filter`

The three versions of the driver allow the selection of different messaging implementations by installing different versions of the driver.

The mauidrvr Driver

The standard version of the MAUI System Driver, `mauidrvr`, is the smallest and fastest of the three. It does not support `msg_set_filter()` so it can execute 100% in system state. This results in a size savings. This also enables the MAUI System Driver to make accelerated OS calls (speed savings) directly into the kernel as well as remove the old semaphore calls (speed and size savings) that protected the message queues.

This version of the MAUI System Driver is not compatible with pre-MAUI 3.1 statically linked applications, which expect all parties accessing the message queue to use semaphore queue locking. Pre-MAUI 3.1 binaries that linked with the MAUI Shared Library (`maui.l`/`maui.il`) are still compatible because those applications will attach to a current MAUI Shared Library Module (`maui`), which uses the MAUI System Driver.

The mauidrvr_lock Driver

The second version of the MAUI System Driver, `mauidrvr_lock`, does not support `msg_set_filter()`, but does provide queue locking via semaphores. This allows the continued use of old versions of statically linked MAUI binaries on a system. This version is both larger and slower than the standard MAUI System Driver because of the additional semaphore code. This version of the driver is not considered as “secure” as the standard driver because the message queue memory is permitted for write access by the old application binaries in user state.

The mauidrvr_filter Driver

The third version of the MAUI System Driver, `mauidrvr_filter`, is the largest, slowest, and least secure of the three versions. However, it is the most compatible with prior versions of MAUI. This version supports both `msg_set_filter()` and queue locking via semaphores. To maintain compatibility with earlier version of MAUI, the `read`, `readn`, `peek`, `peekn`, and `flush` functions have to execute in user state when a filter function is active. This required that the application be permitted both read and write access to the memory containing the message queues. This leaves the message queues vulnerable to errant or malicious data accesses by applications, which could result in lockups and other unpredictable behavior. In addition, because parts of the driver code now execute in user state, the accelerated system state OS calls, plus full semaphore locking of the message queues, is required.

Extreme care should be exercised when implementing filter functions. Filter functions are intended only for inspection of the message data in the queue. Activity in the filter function should be kept to a minimum. The filter function must not attempt to modify the messages in the queue. Programming errors or abnormal termination of an application while executing a filter function can cause damage to message queue and the other processes using that message queue.

The filter function should not make function calls or cause the application to block or abort. While the filter function is called, MAUI locks the message queue of the mailbox, blocking all other applications attempting to access the message queue. Delays or failures in the filter function can impact the stability of other applications accessing the message queue.

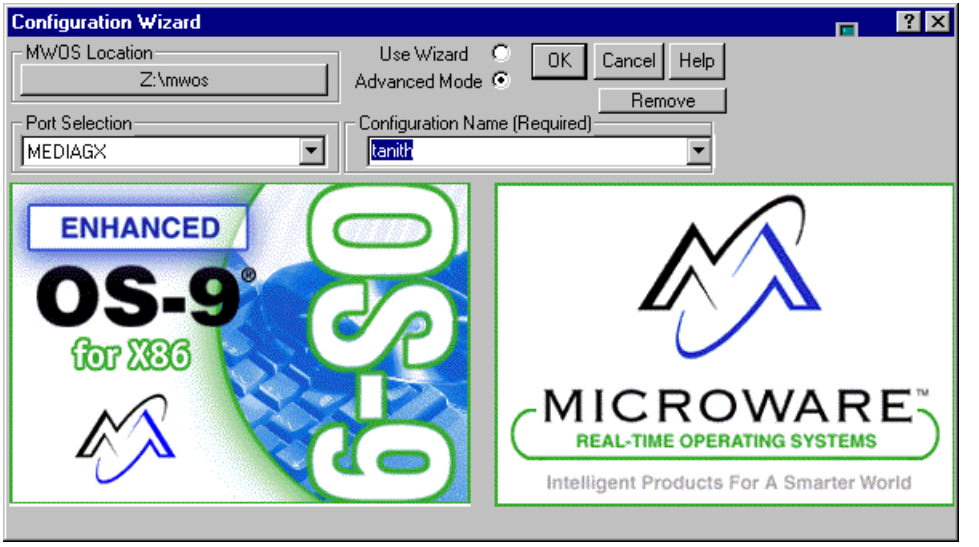
Unless there is a specific need for compatibility with pre-MAUI 3.1 binaries that were statically linked, it is recommended that you use the `mauidrvr` version of the driver.

Using the Configuration Wizard for MAUI

This section describes using the OS-9 configuration wizard to configure your software system for MAUI.

Step 1. Start the configuration wizard by selecting **Start -> Programs -> Microware -> <Product Name> -> Microware Configuration Wizard** from your Windows development system. Select **Advanced Mode** and click **OK**. The configuration wizard executable is located in `mwos\DOC\BIN\os9p.exe` on your Windows development system.

Figure 5-1 Configuration Wizard Starting Window



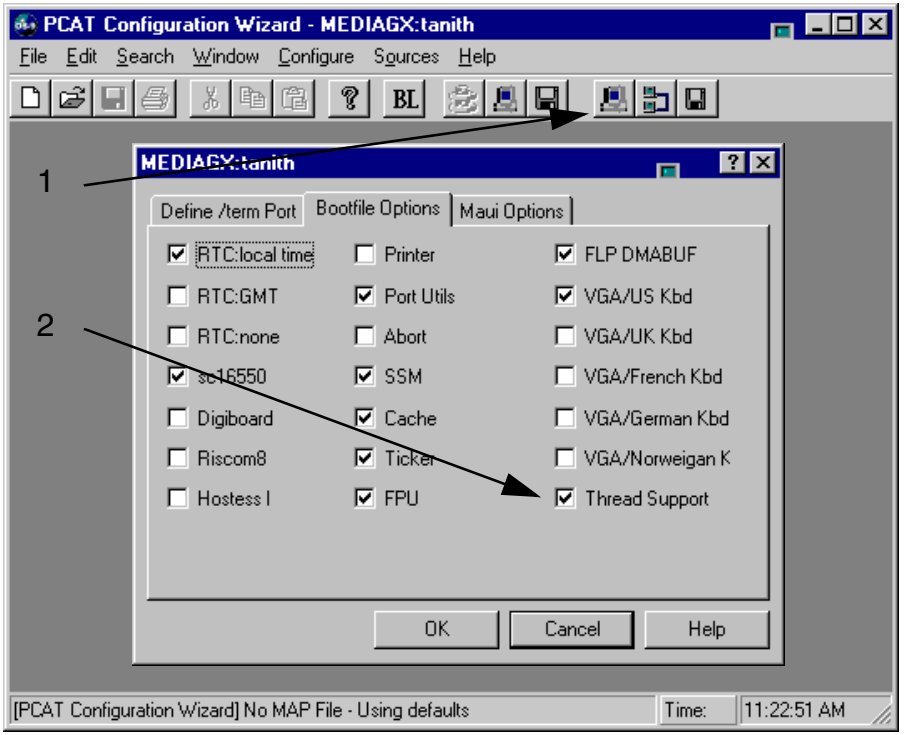
Note

This section assumes that you are familiar with the configuration wizard and can build an OS-9 system that boots your target hardware to an OS-9 shell prompt on one of the serial ports. This information is provided in the appropriate OS-9 Board Guide.

Step 2. Choose your bootfile options by pushing the **Configure Systems Options** button on the toolbar shown by arrow 1. Select the **Bootfile Options** tab.

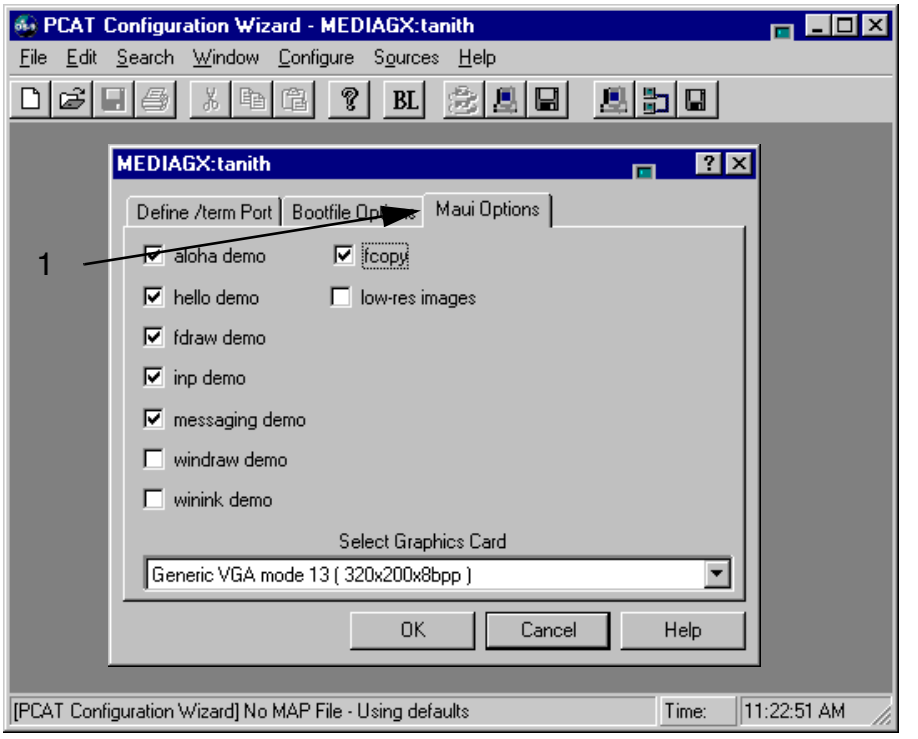
If the target system must support threaded applications, check the **Thread Support** box shown by arrow 2. Clear this box if you do not require thread support. This option selects the appropriate C and MAUI shared library modules. Thread support has a slightly larger footprint and some operations may run a little slower.

Figure 5-2 MAUI Bootfile Options



Step 3. Select the **MAUI Options** tab shown by arrow 1.

Figure 5-3 MAUI Options

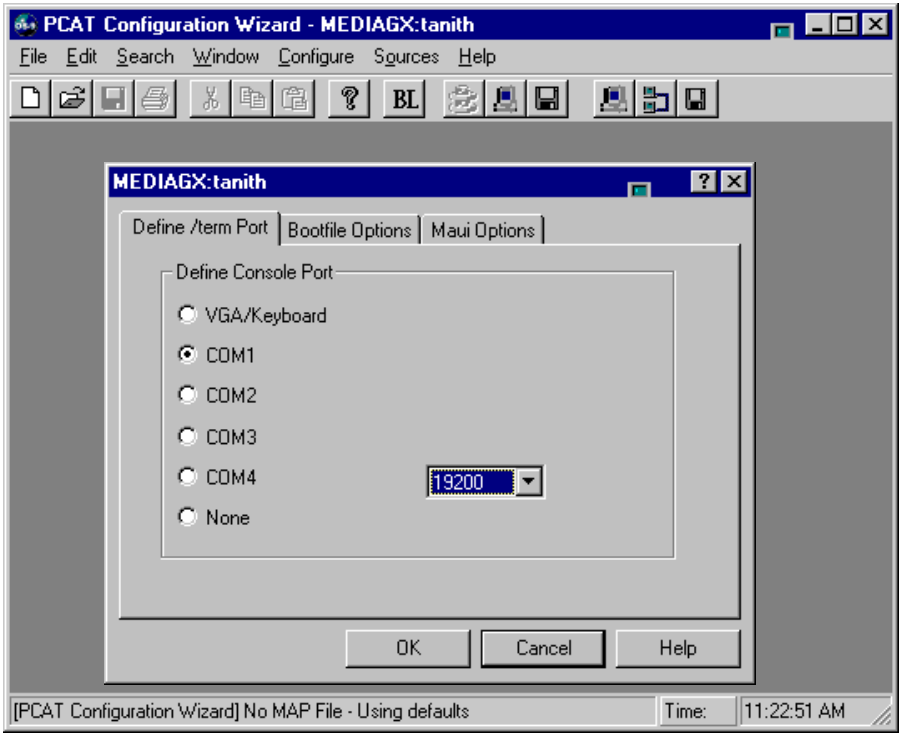


This screen varies depending on the specific port you are working with. Refer to the object module descriptions in the previous section and the appropriate OS-9 Board Guide for descriptions of these options. In this example, you can select the version of the graphics driver to use and what demos to load.

Some systems (for example the PCAT and Sandpoint) boot by default with a command console on the VGA/Keyboard rather than a serial port. An active shell on the same port as used by `maui_inp` can cause

conflicts. It is recommended that when using MAUI on these systems, you switch the Console Port to one of the serial ports. To set the Console Port, select the **Define /term Port** tab.

Figure 5-4 Setting the Console Port

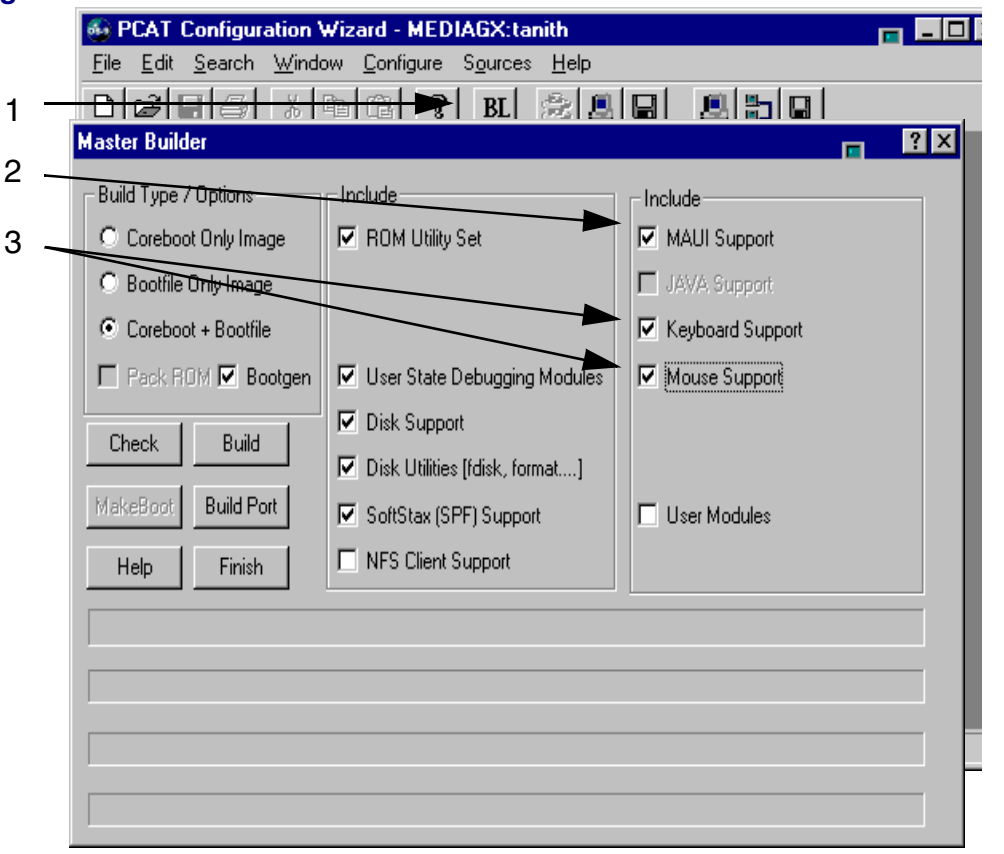


For instance, the screen dump below has selected COM1 rather than the default VGA/Keyboard for the command shell. This allows MAUI applications uncontested access to the scan code keyboard.

Step 4. Click **OK** and close the window.

Step 5. Select the **Build Images** button from the toolbar shown by arrow 1.

Figure 5-5 Master Builder Window



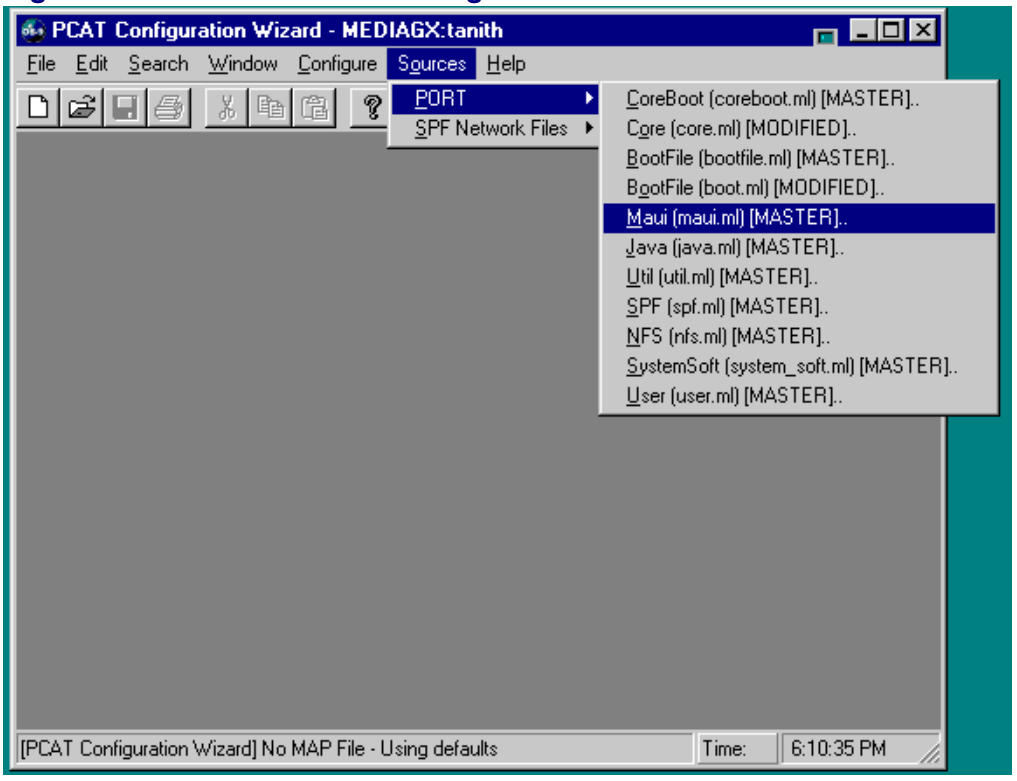
The Master Builder window is displayed. Here you select which components to enable on the system. Select the MAUI Support check box (shown by arrow 2) as well as any other boxes that may be appropriate for the device (shown by arrow 3). Again the specific selections may differ between ports.

Step 6. Click the **Build** button. For advanced MAUI settings, see the **Advanced Wizard Configuration** section.

Advanced Wizard Configuration

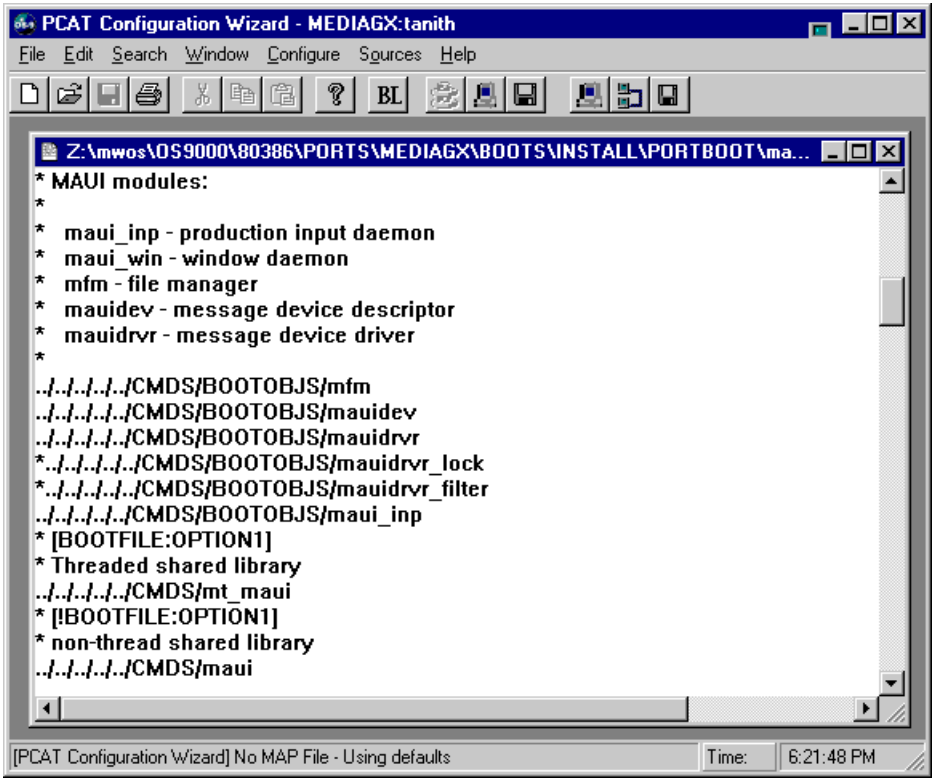
The Advanced Mode of the Wizard provides convenient access to the file lists used to build the boot images. The MAUI file list is called `maui.ml` and can be accessed by selecting **Sources -> Port -> Maui** from the configuration window menu.

Figure 5-6 MAUI Advanced Configuration



This selection opens a text editor window for editing the MAUI file list.

Figure 5-7 Editing the MAUI File List



A “*” in the first column denotes a comment, which is ignored. Comment lines that contain bracketed items such as “`* [BOOTFILE:OPTION1]`” are used by the wizard to find and select files based on the options selected in the Wizard configurations screens.

By removing and adding comment symbols “*”, a developer has absolute control of what MAUI objects are placed in the boot image. This includes changing file selections that are not controllable from the Wizards graphic user interface, such as which `mauidrvr` or `maui_inp` to use.

Index

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

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_____