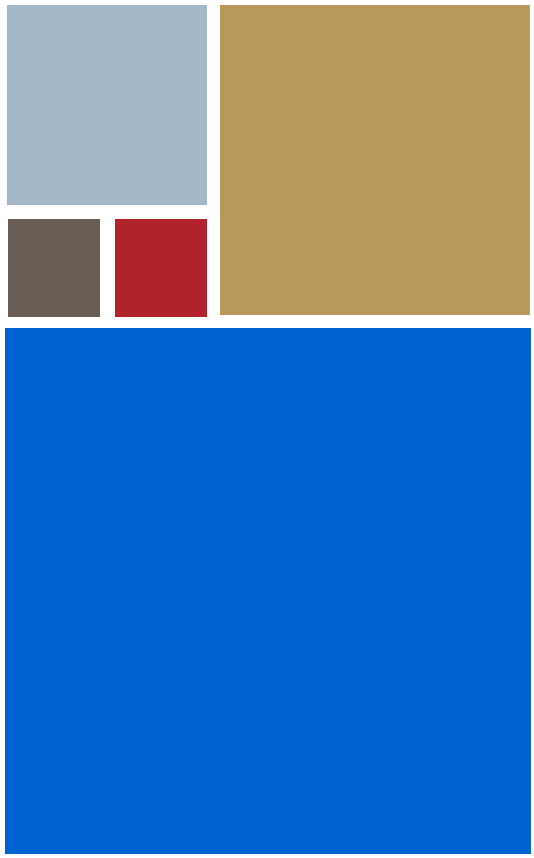


Home

Using MAUI®

Version 3.2

RadiSys
THE POWER OF WE

www.radisys.com

Revision A • July 2006

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.

Table of Contents

Chapter 1: Introduction to MAUI

This chapter is an overview of the Multimedia Application User Interface (MAUI®) architecture and components. Read this chapter to help you understand how the MAUI components relate to each other.

Most of this manual deals with the interface between the application and the MAUI Application Program Interfaces (APIs), but it is important that you read this chapter to get an overall understanding of the MAUI components and how they relate to each other.

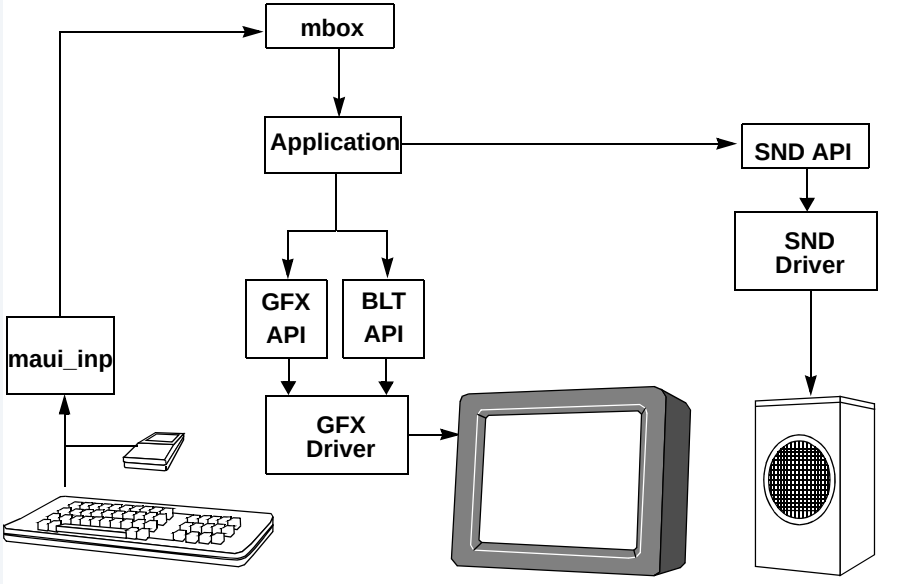
MAUI System Design

The MAUI system is a complete set of multimedia development libraries that are modular, fast, and hardware independent.

The philosophy used in developing the OS-9® and OS-9 for 68K operating systems stressed modularity; we designed the MAUI system in accordance with that philosophy. MAUI APIs are building blocks you use to design applications that range from very simple to very complex, depending on your environment and needs. You can adapt MAUI to any size system, from small ROM-based embedded applications to large-scale network-based development systems.

The MAUI system enables you to write applications that are portable to a wide variety of hardware configurations. MAUI is written in the ANSI-C language so you can port it to any CPU supported by the Microware Ultra-C compiler. More importantly, the MAUI hardware drivers and protocol modules insulate applications from problems that arise due to differences in multimedia hardware.

Figure 1-1 The MAUI System Architecture



As shown in Figure 1-1, the MAUI system comprises several components. At the heart of the MAUI system is a powerful set of APIs. These form the application interface to the MAUI system. For simplicity, the diagram shows most of the APIs in one block. However, this is actually a hierarchy of APIs. Each API has its own scope and objectives.

MAUI Input Process

The MAUI Input Process is responsible for insulating the API layers from differences in pointer and keycode devices. This process reads raw input from Serial Character File manager (SCF) devices and uses protocol modules to translate the data into standardized MAUI messages. These messages are then inserted into the application's mailbox so that the application may act on the user input.

Device Drivers

The lowest layers of the MAUI system comprise a set of drivers that handle all interaction with the multimedia hardware. This includes a graphics driver and a set of Serial Character File Manager (SCF) drivers for the pointer and keycode devices.

The graphics driver is responsible for graphics device management. The interface between the graphics driver and the API is standardized to insulate the API layer from hardware differences.

In addition to the standard entry points found in all OS-9/OS-9000 drivers, the MAUI graphics driver has a special set of fast entry points. These fast entry points enable the API code to call hardware-specific functions to perform operations that would otherwise be performed by the main CPU. For example, if a graphics chip has built-in support for Bit-BLT operations, its driver provides a set of fast entry points for the API's use.

Application Programming Interfaces

The MAUI APIs form a comprehensive set of functions that enable you to build your multimedia application. A summary of each API is presented below.

MAUI System API

Ease of use is important for any technology. The MAUI System API provides a simple method for applications to initialize and terminate all of the MAUI APIs in the correct order with a single function call.

Shaded Memory API

Efficient memory management is a requirement for any graphical environment. Graphical environments tend to be very dynamic when it comes to allocating and freeing memory segments. Therefore, unless an application takes specific steps to avoid it, memory fragmentation can become a serious problem.

The Shaded Memory API provides the facilities applications (and other APIs) required to manage multiple pools of memory.

Configuration Description Block API

The Configuration Description Block (CDB) API makes it possible for the application to examine a system configuration and retrieve information about devices. The source of this information is the Configuration Description Block module. It consists of individual Device Description Records that represent a device in the system.

Graphics Device API

The Graphics Device API is responsible for insulating applications from differences in graphics hardware. This is performed primarily through the use of two high level abstractions, drawmaps and viewports.

A *drawmap* is an object that defines a rectangular piece of pixel memory, and a *viewport* is a mechanism that allows you to make drawmaps visible on the display.

Bit-BLT API

All pixel manipulation in the MAUI system eventually filters down to the operations supported by the Bit-BLT API. Therefore, it is of the utmost importance to have fast Bit-BLT functions. The Bit-BLT API provides functions that rapidly perform a range of operations, from drawing individual pixels to copying blocks from one pixel depth to another (dynamic pixel expansion).

If a hardware Bit-BLT engine is available, the graphics driver supports a set of fast entry points so the API can take advantage of the hardware engine. This is transparent to the application.

Drawing API

The Drawing API provides a suite of functions for drawing to a drawmap. These functions include points, lines, poly-lines, polygons, circles, and rectangles.

Shapes can be drawn in outline or solid mode. Attributes such as patterns and line styles may be applied to these shapes.

Text API

The Text API provides a suite of functions for writing multi-byte and wide-character strings to a drawmap. These functions provide support for multiple fonts (mono and proportional spaced), and methods for controlling the padding between characters.

Animation API

The Animation API makes it possible for the application to move sprites quickly and smoothly on the display. The sprite mechanism creates the illusion of continuous movement by changing image frames and location.

Messaging API

In many cases, one of the most difficult transitions you face when delving into the world of graphics programming is giving up the ideas of procedural programming and adopting the philosophy of event-driven programming. However, you must use and understand event-driven programming methods if you want to develop applications that allow for user control.

The MAUI system, like virtually all other popular GUI platforms, is tailored toward applications that run in an event-driven manner. The Messaging API is the part of MAUI that enables event-driven applications. This philosophy is explored in Chapter 3, which guides you through writing and executing your first MAUI program.

Input API

Input in the MAUI system is not limited to a keyboard and a mouse; you can use any type of input device (for example, remote control, joystick, game controller). The MAUI system uses protocol modules to convert the raw SCF input received from the device into standardized pointer and key symbol messages. Through the MAUI Input Process, an application can use any number of input devices. This is especially useful in multi-player game applications.

Table 1-1 API Descriptions and Dependencies

API	Prefix	Purpose	Dependency
MAUI System	`maui_`	Top level convenience functions.	All other APIs
Shaded Memory	`mem_`	Manages memory dynamics.	None
Configuration Description Block	`cdb_`	Enables the application to examine a set top box system configuration and retrieve information about attached devices.	`mem_`
Graphics Device	`gfx_`	Manages the graphics devices.	`mem_`

Table 1-1 API Descriptions and Dependencies (continued)

API	Prefix	Purpose	Dependency
Bit-Block Transfer (Bit-BLT)	`blt_`	Performs block drawing and block transfer operations.	`mem_ gfx_`
Drawing	`drw_`	Draws geometric shapes.	`mem_ blt_`
Text	`txt_`	Performs graphical text output.	`mem_ blt_`
Animation	`anm_`	Performs sprite animation.	`mem_ blt_`
Messaging	`msg_`	Performs inter- and intra-process message exchanging.	`mem_`
Input	`inp_`	Manages pointer device input.	`mem_ msg_`
Windowing	`win_`	Performs windowing operations	`mem_ gfx_ blt_ msg_ inp_`

MAUI Application Profile

Table 1-2 on page 22 defines the contents of the MAUI Application Profile. The application profile is available in three forms:

- A shared library used to access the shared library module
- An i-code link library
- An o-code link library.

The shared library is the normal way to link MAUI applications. This library contains small bindings that call into the shared library module. This is similar in concept to `csl.l` and `csl`. Using the MAUI shared library has the advantage of making individual applications as small as possible. In addition, applications need not recompile between minor updates of MAUI as the bulk of the API code is located in the shared library module. The disadvantage of using the shared library is that there is a small speed penalty for crossing the application/shared library boundary. This is a result of having to switch between the application global and constant pointers to the shared library global and constant pointers.

The i-code and o-code link libraries are provided to aid debugging and for some environments where use of the shared library is not feasible. While every attempt is made to maintain compatibility between MAUI releases, where the goals of the shared library and link libraries conflict, the shared library has precedence.

The MAUI application profile consists of all thirteen APIs and is fully functional. This is the profile required by JAVA.

Table 1-2 The MAUI Application Profile

API	Shared Library and Module[1]		Static Link Libraries		
	maui.l	maui	mauilib.l	mauilib.il	mfm.l
MAUI (System)	stub	full	full	full	
MEM	stub	full	full	full	
CDB	stub	full	ull	full	
GFX	stub	full	full	full	
BLT	stub	full	ful	full	
DRW	stub	full	full	full	
TXT	stub	full	full	full	
ANM	stub	full	full	full	
MSG	stub	full	full	full	
INP	stub	full + maui_inp	full	full	
WIN	stub	full + maui_win	full	full	
_os_gfx			as required		full
_os_snd					full

1 Shared libraries may not be available for all processors.

Printing to stderr

The MAUI shared library can to print to `stderr` without pulling in the `fprintf()` code. This is enabled by using the callbacks `maui_vfprintf()` and `maui_fflush()`. Functions such as `mem_list_segments()` and `mem_list_overflows()` work as documented in the shared library.

If you do not want the shared library to print or do not want to incur the code size overhead of the print functions in the binary of the application, simply define the following functions in the application:

```
#include <stdio.h>
  int maui_vfprintf(FILE *fp, const char *fmt, va_list ap)
      {
        return 0;
      }
      int maui_fflush(FILE *fp)
      {
        return 0;
      }
```

Chapter 2: MAUI Concepts

With MAUI, you build your applications on top of a data structure foundation. Knowledge of the MAUI data structures helps you understand the information in the example programs.

This chapter introduces you to the types of MAUI data structures and how they inter-relate.

MAUI Data Structures

The MAUI system contains two types of data structures: public and private. Public data structures have a format that is known to the application. The application may change members of the structure directly with mechanisms supported by the ANSI-C language. Private data structure formats are hidden from the application and accessible only through dedicated API function calls.

Public Data Structures

A public data structure has two distinct advantages over a private data structure:

- Because the structure members may be accessed directly, function calls are not required to change them. This allows for more flexibility when you modify the structure members.
- You may create a public data structure using any mechanism supported by Ultra-C.

For example, the following code segment defines a drawmap that represents a bitmap in a public data structure. This data structure represents a 16x16 pixel checkerboard pattern.

```
const GFX_PIXEL checker_pixmem =
{
   0xf0f0f0f0, 0xf0f0f0f0,/* Lines 0 - 3 */
   0x0f0f0f0f, 0x0f0f0f0f,/* Lines 4 - 7 */
   0xf0f0f0f0, 0xf0f0f0f0,/* Lines 8 - 11 */
   0x0f0f0f0f, 0x0f0f0f0f,/* Lines 12- 15 */
};
const GFX_PALETTE checker_palette = {
   0,                   /* Start entry */
   2,                   /* Number of entries */
   GFX_COLOR_RGB;       /* color type */
   {0xff0000, 0x00ff00}/* Red, green */
};
```

```
const GFX_DMAP checker_board =
{
   GFX_CM_1BIT,         /* Coding method */
      16, 16,           /* Width and height */
      2,                /* Line size */
      &checker_pixmem,  /* Pixel memory address*/
      0,                /* Pixmem shade */
      2 * 16,           /* Pixmem size */
      &checker_palette; /* Palette address*/
};
```

In this example, it is important to note that the entire checkerboard pattern is defined by initialized data. Furthermore, because this code uses `const`, the storage allocated for the pattern is in the code area rather than the data area.

The above example shows one method of using the C language to create MAUI objects. You can also use MAUI functions to create and modify these objects.

Private Data Structures

Private data structures are used when the information must be hidden from the application. There are several possible reasons for hiding this information from the application:

- The implementation may be hardware specific. In this case, it is not wise to burden the application with hardware-specific details. For example, the implementation of viewports on each hardware platform can be completely different because of specific hardware requirements.
- The information may be implementation specific. In this case, we do not want to burden the application with details that may change in future releases. An example of this is the Bit-BLT context object (an object used for drawing).

Graphics Device

Regardless of what your graphics program does internally, at some point you need to make your interface visible to the user. The physical device that does this is the *graphics device*. The MAUI component that allows you to manipulate this device is the Graphics Device API.

The Graphics Device API is not limited to a particular type of graphics device. MAUI provides your application with the information needed to adjust to the hardware capabilities. Differences in graphics device capabilities fall into three general categories:

- display resolutions
- coding methods supported
- viewport complexity

Display Resolutions

Display resolutions vary from one device to another. For example, VGA devices typically support resolutions such as 640x480, 800x600 or 1024x768, and CD-i resolutions are 384x240 or 768x480. The resolution is also affected by such factors as whether the output is a PAL or NTSC format.

Different hardware can also affect the size of pixels. For example, a VGA monitor has square pixels, while NTSC and PAL devices develop a pixel that is rectangular (e.g. NTSC 1:1.19). This is referred to as pixel aspect ratio.

Coding Methods

Coding methods present a challenge for applications that must run on several devices because all devices do not support the same coding methods. Common coding methods include RGB (direct color values), CLUT (Color Look-Up Table), YUV, and Run Length.

Because pixel depth is a factor in coding methods, MAUI supports 1-, 2-, 4-, 8-, 16- and 32-bit pixels.

Graphics Device Capability

MAUI provides the `gfx_get_dev_cap()` function to address graphics device capability issues. This function supplies applications with information about the host hardware capabilities so adjustments can be made to run on the target hardware.

The `GFX_DEV_CAP` structure provides certain information about the graphics device. However, a complete explanation of the capabilities of a particular graphics device should be available from the graphics driver manufacturer that ported MAUI to your hardware. When you design your application, refer to the graphic driver manufacturer's specifications for each device you plan to support.

The following example shows the `GFX_DEV_CAP` and related data structures returned by `gfx_get_dev_cap()` for an potential SVGA device. This device claims to support three resolutions and three coding methods:

```
GFX_DEV_CAP gdv_dev_cap = {
  "SVGA",          /* Hardware type */
  "Doc example", /* Hardware sub-type name */
  FALSE,           /* Supports viewport mixing */
  FALSE,           /* Supports external video */
  TRUE,            /* 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 */
};
```

The `GFX_DEV_CAP` structure contains two pointers (`res_info` and `cm_info`) to separate data structures. Both are arrays containing additional device capability information.

The first array indicates the set of resolutions that are supported by this device. The first entry is considered the default resolution. In the current example, the device supports three resolutions:

```
GFX_DEV_RES gdv_res_info[] = {
  {640, 480, 60, GFX_INTL_OFF, 1, 1},  /* Default resolution
640x480*/
  {800, 600, 60, GFX_INTL_OFF, 1, 1},
  {1024, 768, 60, GFX_INTL_OFF, 1, 1}
};
```

The second array indicates the set of coding methods that are supported by this device. The first entry is considered the default coding method. In the current, example the device also supports three coding methods:

```
GFX_DEV_CM gdv_cm_info[] = {
  {GFX_CM_8BIT  |gfx_set_cm_depth(4),TRUE, 1,1,GDV_NUMCOLORS,
    gdv_valid_colors},
  {GFX_CM_RGB555|gfx_set_cm_depth(5),FALSE,1,1,0, NULL},
  {GFX_CM_RGB888|gfx_set_cm_depth(6),FALSE,1,1,0, NULL}
};
```

Typically (but not always) the first `res_info` will work with the first `cm_info`.

In MAUI 3.1, a new extensible device capabilities structure, `GFX_DEV_CAPEXTEN`, and call, `gfx_get_dev_capexten()`, were added to supplement the information in `GFX_DEV_CAP`.

The following example shows the `GFX_DEV_CAPEXTEN` and the `GFX_DEV_MODES` structure it references, returned by `gfx_get_dev_capexten()` for the above device. To make this more interesting we assume this device only has 2MB of graphic memory and cannot display all combinations:

```
GFX_DEV_MODES gdv_dev_modes[] = {
  {0, 0, "640x480x8"},
  {1, 0, "800x600x8"},
  {2, 0, "1024x768x8"},
  {0, 1, "640x480x555"},
  {0, 2, "640x480x888"}
};
```

```
const GFX_DEV_CAPEXTEN gdv_dev_capexten = {
  sizeof(GFX_DEV_CAPEXTEN), /* Version of structure */
  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 */
};
```

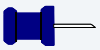
The new `GFX_DEV_CAPEXTEN` structure is designed such that it can easily be updated in future releases, but this places a larger burden on the application to validate the fields requested are valid.

First, the application must verify that the call returned successfully since not all MAUI systems implement this call. Systems with a MAUI shared library prior to version MAUI 3.1 do not know about this call and will return `EOS_ITRAP`. If the MAUI shared library knows about the new call or you are statically linking the application, but the driver is older than 3.1, the call returns `EOS_UNKSVC`. If the driver was compiled against the MAUI 3.1 graphic driver common code, but the developer did not supply the a `GFX_DEV_CAPEXTEN` structure, the call returns `EOS_MAUI_NODVSUPPORT`.

Next, since `GFX_DEV_CAPEXTEN` is an extensible structure, it is possible that the driver may have been build with different versions of the structure than the application. It is assumed that this structure will always increase in size, that the application may only read this structure, and that the first member of the structure contains the size of `GFX_DEV_CAPEXTEN` as initialized by the driver. Based on these assumption, the application can validate access to any particular field using the `GFX_DEV_CAPEXTEN_VALIDATE(ptr, member)` macro. For instance, to determine if the `vpdm_complexity` member was initialized by the driver, do the following:

```
const GFX_DEV_CAPEXTEN *edcap;
  ...
if (GFX_DEV_CAPEXTEN_VALIDATE(edcap, vpdm_complexity))
  printf("vpdp = %d\n", edcap->vpdm_complexity);
```

This macro is relatively inexpensive as it is reduced by the compiler to a simple compare of `edcap->version` to a constant.



Note

You are not required to use the `GFX_DEV_CAPEXTEN_VALIDATE()` macro for 3.1 members of the `GFX_DEV_CAPEXTEN` structure, since drivers should define at least the 3.1 members if they define the structure at all. But it is good practice to validate use of all members of this structure.

For More Information

See Chapter 7: Using the Graphics Device API for more information about display types.

Drawmaps

Drawmaps are specific allocations of pixel memory that store graphic images. Drawmaps are not directly displayed on the screen; they are simply a storage device in which a graphic image is loaded. When a drawmap is created, it is described in the following structure:

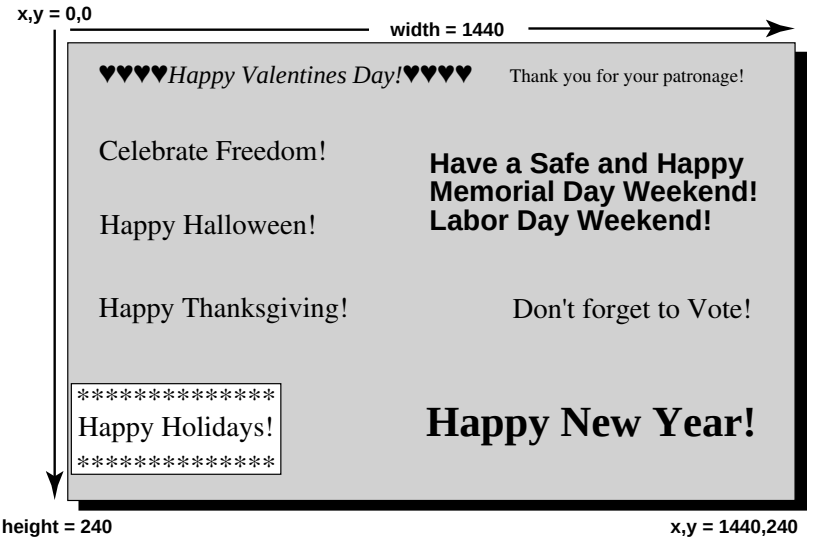
```
typedef struct _GFX_DMAP {
   GFX_CM coding_method;/* Coding method */
   GFX_DIMEN width;     /* Width in pixels */
   GFX_DIMEN height;    /* Height in pixels */
   size_t line_size;    /* Size of line in bytes */
   GFX_PIXEL *pixmem;   /* Ptr to pixel memory */
   u_int32 pixmem_shade;/* Shade used for pixmem */
   size_t pixmem_size; /* Pixmem size in bytes */
   GFX_PALETTE *palette;/* Ptr to color palette */
} GFX_DMAP;
```

The coding method defines the image type and can be a CLUT, RGB, Run Length, or DYUV image type. The width and height parameters specify the maximum image size stored in the drawmap. The width and height are limited only by the size of memory allocated by MAUI for pixel memory, and are not related to the size of the display. After a drawmap is created, an image can be loaded into the drawmap.

Images stored in drawmaps can be displayed in a number of ways. You can display the entire image in the drawmap, or define a small portion of a drawmap for display. The example image in Figure 2-1 shows an image that is 1440 pixels wide by 240 pixels tall, and contains many small messages. Each message can be displayed anywhere on the display screen, or several small messages can be arranged on the

display screen. For example, **Have a Safe and Happy** can be combined on a display screen with **Memorial Day Weekend!** to make the single message display: **Have a Safe and Happy Memorial Day Weekend!**

Figure 2-1 Drawmap



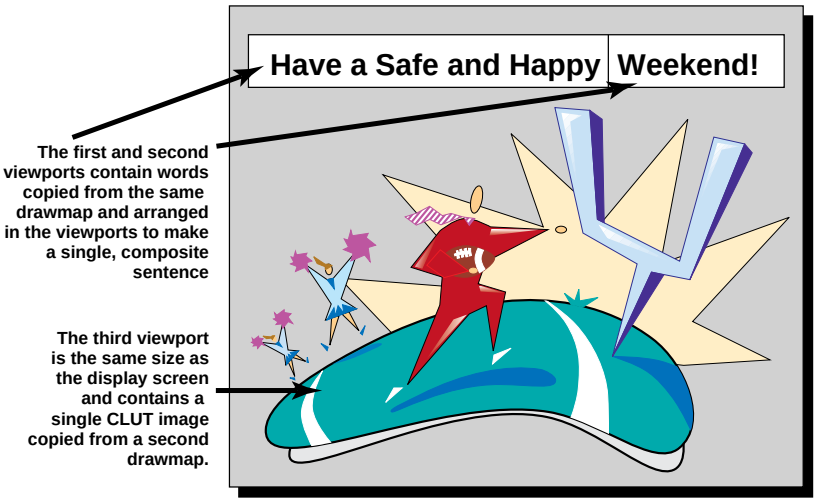
To display a portion of the image from the drawmap, identify the portion by the x,y coordinate of the top left corner of the partial image, and define the area in width and height of the partial. For example, the message Happy Holidays in the sample image is located at x,y coordinate 10,200, and occupies an area 350 pixels wide by 40 pixels high.

When all or part of a drawmap displays, it appears on the screen by way of a viewport. The next section describes viewports and their relationship to drawmaps.

Viewports

Viewports are defined areas of the screen that display still graphic images. The images displayed in the viewport are loaded from drawmaps. A display screen may contain a single viewport that is as large as the screen, or several viewports of various sizes and positions. The number, size, and transparency capabilities of the viewports are determined by the hardware in the user's system. Using the drawmap example in the previous section, the following illustration constructs a display screen from two different drawmaps using three viewports.

Figure 2-2 Viewports



Viewports that overlap are stacked from front to back. When viewports are stacked, they may be opaque so the underlying viewports are partially or completely covered by the front viewport (as is the case in the figure Viewports). Viewports may also be mixed so there is some level of transparency from front to back. The level of transparency is determined by the intensity level of each viewport. For example, if the front viewport lies over another viewport, and both have defined intensities of 100, each viewport contributes 50% to the overlapped area when mixing is on.

For More Information

This technique is covered in detail in Chapter 7: Using the Graphics Device API.

When an external video signal such as MPEG video or a composite video signal displays, viewports may be overlaid on the external video plane. External video does not display in a viewport and always displays on the background plane behind all active viewports. External video may be turned on or off. If external video is turned off, the background will be a single, solid color. While all these features are available with MAUI, your hardware capabilities determine to what extent you can use them. The graphics device capabilities are stored in a structure called `GFX_DEV_CAP`, and lists all the available features for a particular system. As MAUI 3.1, there is a secondary, optional `GFX_DEV_CAPEXTEN` structure that contains additional device capabilities information. Your applications can read the device capabilities and adjust to the hardware environment at run-time.

Constructing a Display

To construct a display like the one in the viewport example requires several steps. Let's assume that the two drawmaps have been created and pictures loaded in the drawmaps. The first step in constructing the display is to define the size of the display. This is done with the `gfx_set_display_size()` function. In this example, the display size is 720 x 240. The top, left corner is coordinates 0,0. The specific hardware capabilities of the display can be examined by using the function `gfx_get_dev_cap()`.

The next step is to create three viewports and specify their size and position on the display screen. Size and position are specified with display and viewport coordinates.

Creating Viewports

A viewport is a private object that defines a rectangular area on the display screen. Viewports are created with the `gfx_create_vport()` function.

The first viewport contains the partial image *Have a Safe and Happy*. It is 580 pixels wide by 20 pixels high, and the top, left corner is located at display coordinates 40, 25.

The second viewport contains the partial image *Weekend*. It is 200 pixels wide by 20 pixels high, and the top, left corner is located at display coordinates 620, 25.

The third viewport contains the background image. It is 720 pixels wide by 240 pixels high, and the top, left corner is located at display coordinates 0,0.

Now that the viewports have been created and their position defined on the screen, you can display images from the drawmaps in the viewports.

Displaying Images in Viewports

The first image displayed is the partial image from our messages drawmap. Images are assigned to a particular viewport with the function `gfx_set_vport_dmap()`. In this function, you define which drawmap is being used, the drawmap x,y coordinates of the top, left corner of the image you are displaying, and the width and height of the image.

For example, the partial image *Have a Safe and Happy* is located at drawmap coordinates 860,58, and is 580 pixels wide by 20 pixels high. The location, width, and height of the image being displayed are all expressed relative to the drawmap coordinates.

Pixel Size Differences

Pixel size can vary between display and drawmap coordinate systems because many graphics devices allow the pixel size to change on various areas of the display. For example, the VDSC (used in CD-i systems) supports eight-bit pixels that are twice the physical width as four-bit pixels. Viewports supporting both of these modes may be visible on the display at the same time.

To avoid creating a confusing display coordinate system, describe the system using the smallest (physical size) pixels. This makes it possible to have pixels within a viewport that take up more than one pixel in the display coordinate system.

Drawing

Drawmaps are used for more than just displaying images.You can use drawmaps to collect blocks of graphic data from other drawmaps, draw shapes, and store fonts for text manipulation. Drawmaps can be thought of as working areas where your application can assemble and create screens behind the scenes before transferring the assembled graphic to the viewport for display. Drawing operations performed on drawmaps fall into three general categories:

- Block transfers
- Shape drawing
- Text

Block Transfers

The Bit-BLT API is responsible for all block drawing and block transfer operations. These include the following operations:

- Draw a solid color block
- Copy a block from a source to a destination drawmap
- Draw a horizontal line (block with a height of one pixel)
- Draw a vertical line (block with a width of one pixel)
- Draw a pixel (block with a width and height of one pixel)

To make these routines as fast as possible, parameters that affect the block transfer operations are kept in a separate object called a Bit-BLT context. This context object is configured before calling the block transfer functions. The Bit-BLT context parameters include:

- Foreground and background colors
- Source and destination drawmaps
- Mixing mode
- Palette offset for color expansion

Because these parameters remain the same throughout many block transfer operations, the API saves you work by dealing with the parameters only when they change, instead of in every block transfer function.

The context information is kept in an object so your application may create as many context objects as it deems necessary.

For More Information

Refer to ***Chapter 8: Using the Bit-BLT API*** for more information about block transfer and block drawing operations.

Shape Drawing

The Drawing API is responsible for all shape drawing. The functions in this API compute points on a shape and call the Bit-BLT API to do the drawing. The shapes supported by the Drawing API are as follows:

- Straight and diagonal lines
- Polylines and polygons
- Rectangles
- Circles

The drawing API uses parameters set in a *drawing context* to perform drawing operations. This context object includes parameters such as the following:

- Drawing pattern
- Line style
- Bit-BLT context

For More Information

Refer to Chapter 9: Using the Drawing API for more details.

Text Drawing

The Text API is responsible for all text drawing. At the heart of this API is the definition of the MAUI font object. You can find this definition in the `maui_txt.h` header file. Also, the font object is fully defined in Chapter 10: Using the Text API. The following is a definition of the font object:

```
typedef struct _TXT_FONT
{
   TXT_FONTYPE font_type;/* Font type */
   GFX_DIMEN width;     /* Maximum cell width */
   GFX_DIMEN height;    /* Cell height */
   GFX_DIMEN ascent;    /* Ascent */
   GFX_DIMEN descent;   /* Descent */
   wchar_t default_char;/* Default character */
   u_int8 num_ranges;   /* Number of ranges */
   TXT_RANGTBL *range_tbl;/* Ptr to range table */
} TXT_FONT;
```

The `font_type` defines whether the font is mono-spaced or proportional. Even if it is defined as proportional, it may be used as a mono-spaced font. This is useful when you are using the TTY emulation capabilities of the text API. TTY emulation requires mono-spaced characters, but MAUI allows you to use a proportional font by automatically adjusting the proportional font to display as a mono-spaced font.

```
typedef struct _TXT_RANGTBL
{
   wchar_t first_char; /* First character */
   wchar_t last_char;  /* Last character */
   GFX_OFFSET *offset_tbl;/* Ptr to offset table */
   GFX_DIMEN *dimen_tbl;/* Ptr to dimension tbl */
   GFX_DMAP *bitmap;   /* Pointer to bitmap */
} TXT_RANGTBL;
```

MAUI fonts can have any number of sub-ranges. This is important for Asian fonts, which require several ranges to encompass the entire character set.

This API uses parameters that are set in a *text context* for all drawing. This context object includes parameters such as the following:

- font object
- Bit-BLT context

For More Information

Refer to Chapter 10: Using the Text API for a complete explanation of the Text functions.

Message Loop

Like other graphical environments, MAUI assumes the user is driving the application, rather than the application driving the user. The mechanism that makes this possible is the *message*.

MAUI messages are packets of information that tell an application it is time to do something. The contents of the message determines what must be done.

Most of the time the application is in a *message loop*. The following code segment is an example of a simple message loop.

```
main()
{
   while (1)
   {
      msg_read(mbox, &msg, MSG_TYPE_ANY, MSG_BLOCK);
      msg_dispatch(&msg);
   }
}
```

This is an incomplete example, but it gives you an idea of how a message loop works. A complete MAUI program, including a message loop, is presented in *Aloha MAUI.* The point here is that most of the time your program sits idle, waiting for the user to do something, or for some other event to happen.

When your message loop receives a message, the message contains common information in addition to the information specific to the type of message. This common information is the only part of the message defined by the Messaging API. It is defined in `maui_msg.h` as:

```
typedef struct _MSG_COMMON
{
   u_int32 type;        /* Message type */
   u_int32 time_queued;/* Time msg was queued */
   process_id pid;      /* process ID of writer */
   void (*callback)(const void *msg);
                        /* Message callback */
} MSG_COMMON;
```

Every message begins with this common structure. It contains the message type, the time the message was queued, and a pointer to a callback function. The callback function is called directly by the `msg_dispatch()` function in your message loop.

User Input

This section provides a summary of user input. Briefly, user input falls into one of two classes:

- Pointer symbols
- Key symbols

The header file `maui_inp.h` defines message types for each symbol type.

```
typedef struct _MSG_PTR
{
   MSG_COMMON com;      /* Common section */
   INP_PTR_SUBTYPE subtype;/* Type of ptr message */
   INP_DEV_ID device_id;/* Device ID */
   u_int8 button;       /* Button number */
   u_int8 button_state;/* State of all buttons */
   GFX_POINT position; /* New position */
   wchar keysym;        /* Keysym if simulation */
                        /* caused the message */
} MSG_PTR;
typedef struct _MSG_KEY
{
   MSG_COMMON com;      /* Common section */
   INP_KEY_SUBTYPE subtype;/* type of key */
                        /* symbol message */
   INP_DEV_ID device_id;/* Device ID */
   wchar_t keysym;      /* Key symbol */
   INP_KEYMOD key_modifiers;
                        /* Key modifiers */
} MSG_KEY;
```

Pointer Messages

Pointer messages are generated when the user interacts with a pointer device. Pointer devices supported by MAUI include, but are not limited to, the following:

- Mouse
- Tablet
- Touch screen
- Light pen
- Joystick

Key Symbol Messages

Key symbol messages are generated by the user's interaction with a device that generates key symbols. MAUI devices that support this type of input include, but are not limited to, the following:

- Keyboard
- Remote control
- Game controller

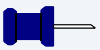
The key symbols generated by MAUI-supported devices are standardized to insulate applications from the protocols of the physical devices used to generate them.

For More Information

See Chapter 13: Using the Input API for detailed function descriptions.

Chapter 3: Writing a MAUI Application

This chapter guides you through the typical steps necessary to write a MAUI program.



Note

Example source code has been included in your MAUI development package in the directory: `MWOS/SRC/MAUI/DEMOS`. Each demo is accompanied by a “read me” or .pdf file describing its use.

Analyzing a Typical MAUI Program

This chapter describes the fundamentals of programming with MAUI and illustrates some of the most important MAUI concepts and programming issues.

In the following sections, example code is provided and described to illustrate the concepts being introduced.

You will notice a lack of error checking in these examples. This was deliberate. We wanted to keep the examples simple so you could stay focused on the most important topics. Error handling is covered later in this chapter.

Include files

The include file `<maui.h>` is needed by virtually all MAUI programs. This file contains declarations of structure types and defined constants used in MAUI functions as well as function prototypes. Many of the structures and constant definitions are described in this manual with the functions that are used.

```
#include <maui.h>
```

Color and Palette Definitions

```
GFX_RGB colors[2] =
{
   0x1010ef,            /* CCIR Blue */
   0xefefef,            /* CCIR White */
};
```

```
GFX_PALETTE palette =
{
   0,                   /* Starting entry */
   2,                   /* Number of colors */
   GFX_COLOR_RGB,
   colors,              /* Colors */
};
```

For simple drawing, the easiest way to set up a color palette is to use initialized data as shown in the definitions of palette and colors above.

For More Information

Refer to the ***MAUI Programming Reference Manual*** for a detailed explanation of the MAUI data types `GFX_PALETTE` and `GFX_RGB`.

Variables

```
   /* Primary MAUI objects */
   GFX_DEV_ID gfxdev;  /* Graphics device ID */
   GFX_DMAP *dmap;     /* Drawmap pointer */
   GFX_PIXEL exptbl[] = {0,1};/* Pixel expansion */
                       /* table */
   GFX_VPORT_ID vport; /* Viewport ID */
   TXT_CONTEXT_ID txt_ctx;/* Text context ID */
   TXT_FONT *font;     /* Font */
```

The variable definition section defines the global variables `dmap`, `vport`, `font`, `blt_ctx`, and `txt_ctx`. These are the primary objects required to make a drawmap visible on the display and to draw text. Also, the variable `exptbl` defines color entries used for the text background (entry `0`) and the foreground (entry `1`).

For More Information

Refer to the ***MAUI Programming Reference Manual*** for a detailed explanation of the MAUI data types.

Initialize MAUI APIs

```
maui_init();
```

The first MAUI function called in any MAUI application is `maui_init()`. When you call this function, all APIs within MAUI are initialized. Until you call `maui_init()`, all other MAUI functions return `E_MAUI_NOINIT`.

Open a Graphics Device

```
/* Open the graphics device */
{
   /* temporary device name */
   char devname[CDB_MAX_DNAME];
   cdb_get_ddr(CDB_TYPE_GRAPHIC, 1, devname,
      NULL);
   gfx_open_dev(&gfxdev, devname);
}
```

Before you use a graphics device, you should open it. In this example, we use the CDB API to extract the device name from the Configuration Description Block. Then, `gfx_open_dev()` opens the device and returns the device ID.

Create and Configure a Drawmap

```
/* Create and configure drawmap */
mem_create_shade(SHADE_PLANEA, MEM_SHADE_NORMAL,
   0x80, 4096, 4096,
   MEM_OV_ATTACHED, TRUE);
gfx_create_dmap(&dmap, SHADE_PLANEA);
gfx_set_dmap_size(dmap, GFX_CM_8BIT, 360, 240);
gfx_set_dmap_pixmem(dmap, NULL, SHADE_PLANEA, 0);
dmap->palette = &palette;
```

Although you can use initialized data to create public data structure drawmaps, that approach is most useful on small drawmaps (for example, with patterns or cursors). For larger drawmaps, such as those you plan to use in viewports on the display device, the steps are a bit more complicated. This is mainly because you may want your application to adjust itself to the resolution of various display devices.

Step 1. Create a drawmap object.

Step 2. Set the drawmap size.

Step 3. Allocate pixel memory for the drawmap.

This concept is fully explored in the ***MAUI Programming Reference Manual***.

Create a Drawmap Object

There are three steps in creating a drawmap object.

Step 1. Allocate the drawmap object. The drawmap object is a small C structure. You may be wondering why we do not code it as shown below instead of calling `gfx_create_dmap()`:

```
GFX_DMAP dmap;
```

The advantage of calling the MAUI create function is that `gfx_create_dmap()` automatically sets each member of the structure to a default value, saving you the trouble of individually setting each structure member's value.

For More Information

See `GFX_DMAP` in the ***MAUI Programming Reference Manual*** for information about the default values.

Step 2. Set the drawmap size. Although you could set the members of the drawmap structure yourself, it is more convenient to use the MAUI supplied function `gfx_set_dmap_size()`. The code segment above sets the width of the drawmap to 360 pixels and height of the drawmap to 240 pixels.

Step 3. Call `gfx_set_dmap_pixmem()` to allocate the drawmap's pixel memory. The pixmem parameter is set to NULL to indicate that the graphics driver should allocate the memory. The allocation is done from shade 0x80.

Create Viewport and Display Drawmap

```
/* Create viewport and put the drawmap in it */
gfx_create_vport(&vport, gfxdev, 0, 0, 720, 240,
     GFX_VPORT_FRONT);
gfx_set_vport_dmap(vport, dmap, 0, 0);

/* Set display parameters and show the viewport */
gfx_set_vport_state(vport, TRUE);
gfx_update_display(gfxdev, FALSE);
```

Viewports allow you to make the drawmap visible on the display. In this example we open a 720 x 240 pixel viewport by calling `gfx_create_vport()`. This example places this viewport on the display at screen coordinates 0,0.

To map the drawmap to the viewport, call `gfx_set_vport_dmap()`. Use an offset of 0,0 because the drawmap is the same size as the viewport. If the drawmap was larger, we could use the offset to tell MAUI what portion of the drawmap should be visible in the viewport.

There are still several more concepts to introduce before anything is visible on the display. MAUI does not display anything until your first call to `gfx_update_display()`. Usually, you want to make several changes to the display, but you do not want to see them happen sequentially; you want them to accumulate until you are ready to make them appear simultaneously.

Create Font Structure and Text Context Objects

```
/* Create font structure from a UCM font module */
get_ucm_font(&font, SHADE_PLANEA, "default.fnt");

/* Create text context */
txt_create_context(&txt_ctx, gfxdev);
txt_set_context_dst(txt_ctx, dmap);
txt_set_context_font(txt_ctx, font);
txt_set_context_exptbl(txt_ctx, 2, exptbl);
```

Currently, a viewport is being displayed, and all that is left is to draw into it. This example is preparing to draw text into the viewport. Drawing in MAUI is done via context objects.

These objects and their capabilities are explained in the ***MAUI Programming Reference Manual*** but right now all you need to know is that they are the objects used to set attributes, such as the colors to draw with and the font required.

The font structure is created by calling `get_ucm_font()`*. The default colors are adequate for our example, so the only parameter we need to set is the destination drawmap to use for Bit-BLT operations.

The text context is created by calling `txt_create_context()`, and parameters within it are set with `txt_set_context()` functions. The only parameters the text context needs are the Bit-BLT context, the font to use for text drawing, and the expansion table, which defines the colors of the text.

Draw the Text String

```
   /* Draw the text string */
   {
/* temporary working variables */
```

*.`get_ucm_font()` is not a part of the standard MAUI library. It is located in `mauidemo.l` and `mauidemo.h`. Both the library and the header file are located where the other MAUI libraries and header files are located.

```
GFX_DIMEN pwidth;       /* Pixel width of a text */
                        /* string write */
char *string;           /* Pointer to text string */
size_t len;             /* Length of text string */
string = "Hello MAUI...";
len = ULONG_MAX;        /* draw the whole string */
txt_draw_mbs(&pwidth, txt_ctx, string, &len,
                20, 40, NULL);
   }
```

Drawing a multi-byte string to a drawmap is as simple as calling `txt_draw_mbs()`. Our example code draws the string "Hello MAUI..." at the coordinates 20,40 in the drawmap specified by `blt_ctx`.

Destroy All Objects and Terminate APIs

```
/* Destroy everything, terminate each API, */
/* and exit */

gfx_set_vport_state(vport, FALSE);
gfx_update_display(gfxdev, TRUE);
txt_destroy_context(txt_ctx);
release_ucm_font(font);
mem_free(dmap->pixmem);
gfx_destroy_vport(vport);
gfx_destroy_dmap(dmap);
gfx_close_dev(gfxdev);
mem_destroy_shade(SHADE_PLANEA);
maui_term();
exit(0);
```

This example shows how to clean up and exit. Objects are usually destroyed in the reverse order in which they were created. Therefore, we perform the following steps in sequence:

Step 1. Destroy the text context object.

Step 2. Destroy the Bit-BLT context object.

Step 3. Destroy the viewport.

Step 4. Destroy the drawmap.

Step 5. Terminate the MAUI APIs.

The call to `maui_term()` is used to terminate the MAUI APIs. This should be the last call to MAUI that your application makes. If you call a MAUI function (besides `maui_init()`) after calling the terminate function, MAUI generates the `E_MAUI_NOINIT` error.

Adding Error Checking

The examples in the previous section are fine as long as the MAUI functions are able to do their work, but what happens when something goes wrong? For example, if `gfx_set_dmap_pixmem()` fails to allocate the pixel memory because there is insufficient memory, subsequent calls to MAUI that require pixel memory will also fail. This is referred to as *cascade failure*.

The obvious solution to this problem is to check the return value from each MAUI function to make sure it succeeded. For example, the code segment in the previous section that creates and configures the drawmap could be modified as follows:

```
error_code ec;
if ((ec = gfx_create_dmap(&dmap, 0)) != SUCCESS)
   exit(ec);
if ((ec = gfx_set_dmap_size(dmap, GFX_CM_BIT1,
                        100,100))
   != SUCCESS)
   exit(ec);
if ((ec = gfx_set_dmap_pixmem(dmap, NULL, 0x80, 0))
   != SUCCESS)
   exit(ec);
```

Although this method works, it tends to get a bit messy, especially if you do this type of checking on every call to a MAUI function. MAUI offers *error handlers* as an alternative.

To understand MAUI error handlers, you must first understand the error levels that MAUI can generate. MAUI errors fall into the following categories (or levels):

- Fatal Errors
- Non-fatal Errors
- Warnings

Fatal Errors

Fatal errors restrict operations likely to cause other operations to fail. This type of error usually sets off a cascade failure. For example, if `gfx_create_dmap()` fails because of insufficient memory, this is considered a fatal error. Attempting to use the drawmap in future calls only leads to more errors.

Non-fatal Errors

Non-fatal errors indicate the operation you were attempting did not succeed, however its failure will not cause other functions to fail. For example, if you call `blt_draw_block()` with a bit-BLT context that has no destination drawmap set, you receive a non-fatal error because it has no context to draw to, but the error will not affect other functions.

Warnings

Warnings indicate something went wrong, but that it was not serious enough to be considered an error. For example, if you call `mem_free()` with a memory segment that was never allocated, you receive a warning. Warnings alert you to unexpected behavior that may occur in your program.

For More Information

MAUI has several functions that deal with error handling. All are explained in the ***MAUI Programming Reference Manual***.

Input Processing

The code segments in the previous sections dealt with output. Most display programs require some type of user input. MAUI uses messages to deliver user input to the application.

Step 1. To add input processing to a program, add a signal handler function and register it with OS-9.

```
/* install signal handler */
intercept (sig_hand);
```

Step 2. Create a MAUI mailbox to receive messages from the input device, and then open the input device.

```
/* create a mailbox */
msg_create_mbo      x(&mbox, “mbox”, 1,
sizeof(INP_MSG),
     MEM_ANY);

/* open input device */
{
   char devname[CDB_MAX_DNAME];
   cdb_get_ddr(CDB_TYPE_REMOTE, 1, devname, NULL);
   inp_open_dev(&inpdev, mbox, devname);
   inp_set_callback(inpdev, mbox, process_keys);
}
```

All error handling is performed through the mailbox function. `msg_create_mbox()` provides a mailbox named mbox to store error messages. `MEM_ANY` sets the error level to generate an error message when an error of any level occurs.

Step 3. Add a section of code called the *message loop*. This is a common term used to refer to code that causes you to wait for the next message and dispatch callback functions to respond to input.

The message loop for a program should be placed following the initialization of the environment. The first line of an event loop usually appears as follows:

```
done= FALSE; /* defined as a global */
while(done == FALSE){
     msg_read(mbox, &inp_msg, MSG_TYPE_ANY,
MSG_BLOCK);
     msg_dispatch(&inp_msg);
}
```

The code for defining the variables and for cleaning up before exiting is not shown here.

The last parameter to `inp_set_callback()` is the name of the callback function to attach to input messages. This is the function that `msg_dispatch()` calls to process the message. In this example it is named `process_keys()` and appears in the code as follows:

```
void process_keys(MAUI_MSG *msg)
{  GFX_DIMEN pwidth;
   char s[80];
   sprintf(s, "You pressed key 0x%4.4x",
      msg->key.keysym);
   txt_draw_mbs(&pwidth,txt_ctx, s, 80, 20, 40,
NULL);
   if (msg->key.keysym == INP_KEY_EXIT)
      done = TRUE;
}
```

This callback function prints a message to the display indicating which key was pressed. When the user presses the `exit` key, it sets the `done` flag, causing the message loop to terminate and the program to exit.

Chapter 4: Using the MAUI System API

The MAUI System API enables applications to initialize, terminate, and set the error action of all supported MAUI APIs available in the application environment with a single function call.

This chapter classifies the MAUI System API functions and explains how they are used.

MAUI System Functions

To use any MAUI API it must first be initialized. If an API is dependent upon another MAUI API, then that API must be initialized first.

The MAUI System API does the following:

- Initializes all the MAUI APIs in the correct order.
- Sets the initial error action for all MAUI APIs.
- Terminates all the MAUI APIs in the correct order.

This provides an easy way for applications to begin initializing their environment without having to understand the MAUI API dependency hierarchy.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual.***

Initialize and Terminate

`maui_init()`	Initialize all MAUI APIs by calling their respective `*_init()` routines in the correct order.

Terminate

`maui_term()`	Terminate all MAUI APIs by calling their respective `*_term()` routines in the correct order.

Setting Error Action

`maui_set_error_action()` Set the Error Action of all MAUI APIs by calling their respective `*_set_error_action()` routines.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Chapter 5: Using the Shaded Memory API

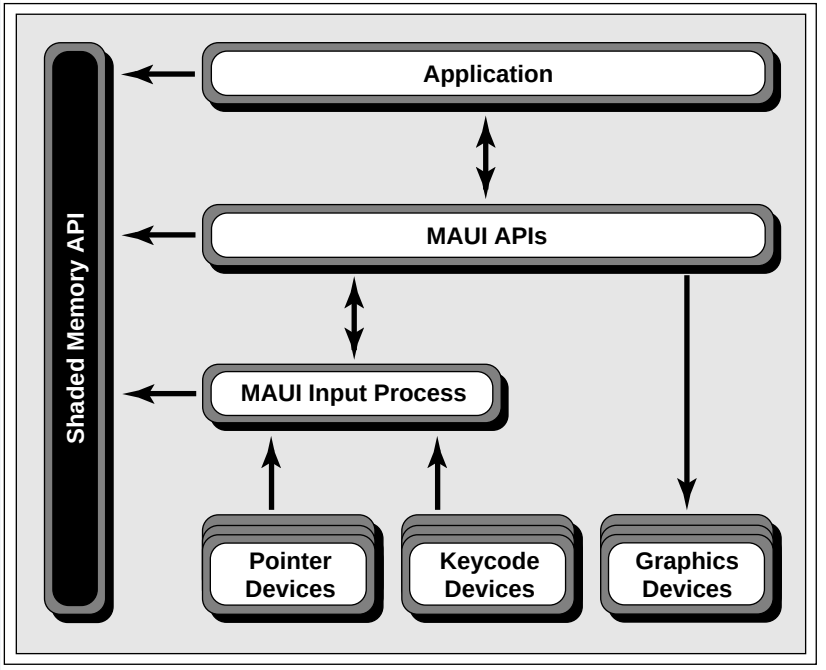
The Shaded Memory API enables applications to decrease the amount of memory fragmentation that occurs when applications perform frequent allocations and de-allocations of memory. The API is based on the memory management features supported by OS-9 and OS-9000, but takes the concept a step further by introducing shading as an additional tool for managing memory partitioning.

This chapter classifies the Shaded Memory API functions and explains how they are used. An example program is included at the end of this chapter that demonstrates the basic techniques for using the Shaded Memory API.

Architecture

All MAUI APIs rely and depend on the Shaded Memory API for memory management. See Figure 5-1 for the relationship between the API and the other components of the MAUI architecture.

Figure 5-1 Shaded Memory API Dependencies



Colors and Shades

The Shaded Memory API:

- Builds on the concept of normal memory, as implemented in the OS-9/OS-9000 operating systems.
- Partitions memory to minimize memory fragmentation during frequent memory allocations.
- Subdivides system memory colors into separate shades with different allocation characteristics.

These characteristics are tied to the block concept. Blocks are pieces of memory allocated when there is not sufficient memory in the shade to satisfy an allocation request.

Important block characteristics are the initial size and the grow size. The initial size is the size of the initial memory block allocated during the shade creation. If it is equal to `0`, no memory is allocated when the shade is created. The grow size determines the block size that can be requested from the system memory when the shade is growing.

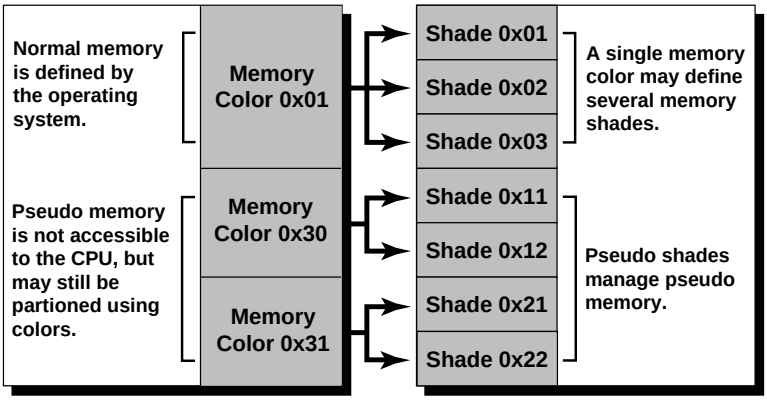
If the `MEM_GROW_MULTIPLE` method is used (see **Creating and Destroying Shades**), the block size is a multiple of the grow size. In the case of `MEM_GROW_LARGER`, the block size is determined by comparing the grow size and the size requested by an application and using the highest number. If a grow size is equal to `0`, the shade is not allowed to grow.

Two types of shades are defined by this API:

- Normal shades use a specific color of system memory.
- Pseudo shades control the allocation and de-allocation of memory from an area not accessible to the CPU. In this case, the operating system is not able to manage the memory.

Refer to Figure 5-2 to see the relationships between normal memory, pseudo memory, and shades.

Figure 5-2 Relationships Between Normal Memory, Pseudo Memory, and Shades



In this diagram, the operating system has defined one color of memory (color 0x01). The application is using this color to define shades 0x01, 0x02, and 0x03.

In addition, the application must provide allocation and de-allocation functions to support the pseudo memory identified by colors 0x30 and 0x31. The application uses color 0x30 to define shades 0x11 and 0x12, and uses color 0x31 to define shades 0x21 and 0x22.

Using Normal Shades

Before allocating memory from a normal shade, you must initialize the shaded memory functions and define the normal shades. The following code segment defines three shades, numbered 1 through 3.

```
#include <MAUI/maui_mem.h>
main()
{
   /* Initialize API and create normal shade */
   mem_init();
   mem_create_shade(1, MEM_SHADE_NORMAL, 0, 8192,
      1024, MEM_OV_SEPARATE, TRUE);
   mem_create_shade(2, MEM_SHADE_NORMAL, 0x80, 0, 1,
      MEM_OV_SEPARATE, TRUE);
   mem_create_shade(3, MEM_SHADE_NORMAL, 0x81, 0,
      4096,MEM_OV_SEPARATE, TRUE);

   /* Call mem_malloc(), mem_calloc() or */
   /* mem_realloc() to allocate each segment. */

   /* Call mem_free() to free each segment */

   /* Destroy shades and terminate the API */

   mem_destroy(1);
   mem_destroy(2);
   mem_destroy(3);
   mem_term();
}
```

These definitions show the diversity in the way an application may choose to manage memory. Shade 1 uses color 0, which allows the system to satisfy the request from any color. The initial size of 8192 bytes forces the initial block to be allocated immediately. This block is not returned to the system until the shade is destroyed. The grow size of 1K forces the shade to grow by blocks that are at least 1024 bytes in size. For example, if you try to allocate 50 bytes and there is no free memory in this shade, a 1K block is allocated from the system and the first 50 bytes are used. The remaining 974 bytes are put in the free list and used to satisfy future requests.

Shade 2 forces all allocations to be satisfied from system memory with color 0x80. Since the initial size is 0, no memory is allocated until a request is made. Since the grow size is 1, this shade always grows by the size of the allocations being made by the application.

Shade 3 has no initial size, but its grow size is 4096 bytes. This means that blocks allocated from the system for this shade must always be multiples of 4K. This shade also requires the memory to come from color 0x81.

Using Pseudo Shades

Pseudo shades differ from normal shades because memory is not allocated from the system to satisfy requests from the application. Instead, allocation and de-allocation functions are provided by the application.

Applications should use `mem_create_shade()` to create a pseudo shade specified as `=_PSEUDO`. Most functions in this API operate on both normal shades and pseudo shades. The only significant difference is that the memory being managed by a pseudo shade is not accessible by the CPU and is therefore never written to by it.

Shaded Memory Functions

The Shaded Memory API functions are classified into four categories: Initialize and Terminate, Create and Destroy Shades, Allocate and Deallocate Memory Segments, and Status and Debugging.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`mem_init()`	Initialize Shaded Memory API
`mem_term()`	Terminate Shaded Memory API
`mem_set_error_action()`	Set the Error Action

To initialize the Shaded Memory API, call `mem_init()`. This automatically creates the default shade `MEM_DEF_SHADE`, with an initial size of 4K and a grow size of 4K using memory color `MEM_SYS`. Because several APIs depend on the Shaded Memory API, another way to initialize the Shaded Memory API is to call `maui_init()` or any of the following functions:

`anm_init()`	`blt_init()`
`cdb_init()`	`drw_init()`
`gfx_init()`	`inp_init()`
`msg_init()`	`txt_init()`

When the Shaded Memory API is not required, it can be deactivated (terminated) using `mem_term()` or `maui_term()`.

`mem_set_error_action()` sets the proper error handler reaction depending on the severity of the error. You can find the initialization and termination examples in the `mem.c` program at the end of this chapter.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Creating and Destroying Shades

`mem_create_shade()`	Create a Normal or Pseudo shade
`mem_set_grow_method()`	Set grow method for a shade
`mem_destroy_shade()`	Destroy a shade of memory
`mem_set_alloc()`	Set allocator function for a shade
`mem_set_alloc_bndry()`	Set boundary size for allocation
`mem_set_dealloc()`	Set de-allocator function for a shade

The normal shade is created by calling `mem_create_shade()`. To create a normal shade, you must determine:

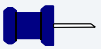
- Desired shade number
- Memory color for the shade
- Initial and grow sizes
- A method for overhead memory allocation
- Whether the overflow/underflow detection is needed

Pseudo shades are also created using `mem_create_shade()`. This call requires pointers to allocation and deallocation functions, provided by an application and called when the shade needs to grow.

The grow method for the shade is set by `mem_set_grow_method()`. To destroy a shade, either normal or pseudo:

Step 1. Call `mem_destroy_shade()`.

Step 2. Deallocate all segments from this shade, otherwise the information in these segments will be printed on the screen (see `mem_list_segments()` in **Status And Debugging**).



Note

Be aware that the initial block of the shade is not returned to the system when it does not have segments allocated.

Allocating and De-allocating Memory Segments

`mem_malloc()`	Allocate shaded memory
`mem_calloc()`	Allocate and clear shaded memory segment
`mem_realloc()`	Reallocate shaded memory
`mem_free()`	Free memory segment from a normal shade
`mem_sfree()`	Free memory segment from the specified shade
`mem_sfree_all()`	Free all segments from the specified shade

Another memory allocation concept is a *segment*. Segments are pieces of memory allocated to an application.

Allocating a Segment

To allocate a segment of memory:

- Use `mem_calloc()` to allocate and clear memory for an array of data structures.
- Use `mem_malloc()` to allocate memory of a specified size.

Reallocating a Segment

To reallocate the memory segment previously allocated from a normal shade use `mem_realloc()`. Although the size of this segment can be changed (the main goal for calling this function), its contents within the reallocated part are kept unchanged.

Deallocating a Segment

To deallocate a segment:

- Use `mem_free()` for normal shades.
- Use `mem_sfree()` for both normal and pseudo shades.

Status And Debugging

`mem_get_shade_status()`	Get shade status
`mem_list_segments()`	Print a listing of allocated segments
`mem_list_tables()`	Print a listing of memory tables
`mem_list_overflows()`	Print a listing of underflows/overflows

The Shaded Memory API allows an application to request memory status information and to perform debugging operations related to memory allocations.

Returning Current Status

To return the current status of the specified shade call `mem_get_shade_status()`.

Printing a List of Allocated Segments

To print the list of allocated segments call `mem_list_segments()`.This enables you to keep track of allocations and deallocations.

Printing a List of Shades, Blocks, Segments

To print the list of shades, blocks, segments and their properties call `mem_list_tables()`.

Printing a List of Overflows/Underflows

To print the list of overflows/underflows call `mem_list_overflows()`. When a shade is created with the check overflows option `true`, safe areas are created at the beginning and the end of the segment. If these safe areas are overwritten, the overflow/underflow situation is reported by `mem_list_overflows()`.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Chapter 6: Using the CDB API

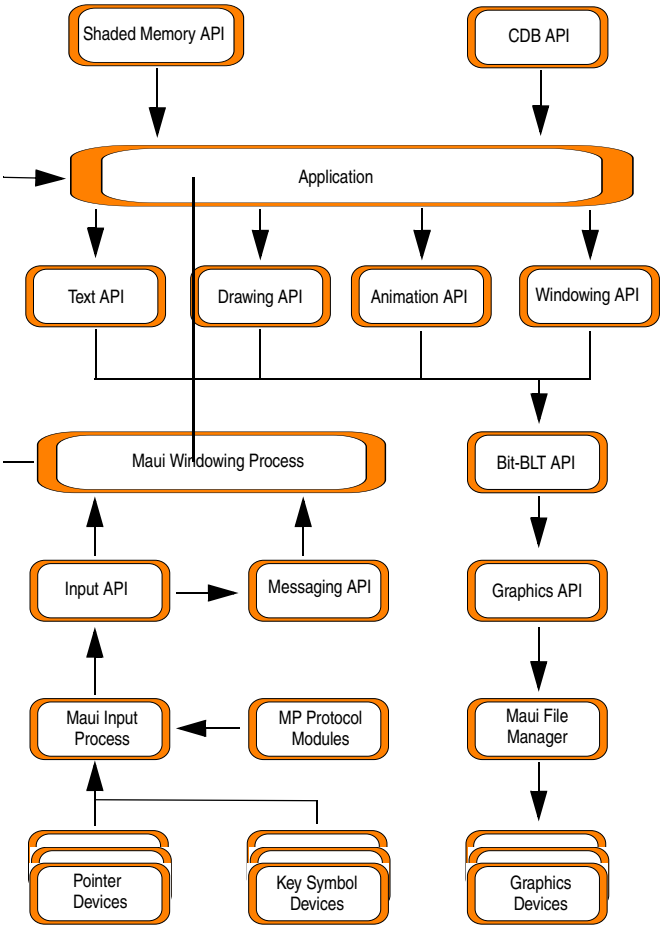
The Configuration Description Block (CDB) API makes it possible for an application to examine a system configuration and retrieve information about attached devices. This information is derived from entries in Device Descriptor Records (DDRs) located in the CDB module.

This chapter classifies the CDB API functions and explains how they are used.

Architecture

The CDB API relies on the Shaded Memory API for memory allocation. The following figure shows the relationship between the CDB API and the other components that comprise the MAUI architecture.

Figure 6-1 Configuration Description Block (CDB) API Dependencies



CDB API Functions

The functions provided by the CDB API are classified into two groups: Initialize and Terminate Functions, and Retrieving Functions.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate Functions

`cdb_init()`	Initialize CDB API
`cdb_term()`	Terminate CDB API
`cdb_set_error_action()`	Set error action

The system-wide functions `maui_init()` and `maui_term()` initialize and terminate all the MAUI APIs, including the CDB API. When the CDB API is initialized, the Shaded Memory API is also initialized.

Initializing the CDB API

To initialize the CDB API separately Call `cdb_init()`. This function initializes the API to prepare it for use. No other functions in the CDB API (excluding `cdb_set_error_action()`) can be called before `cdb_init()`. Any attempt to call CDB API functions prior to the initialization returns the error `EOS_MAUI_NOINIT`.

Terminating the CDB API

To terminate the CDB API call `cdb_term()`. Use `maui_term()` to terminate all APIs. Once this is done, you cannot call any other CDB function.

Changing The Default Actions on Errors

The CDB API has a built-in error handling mechanism. `cdb_set_error_action()` sets the action to take in the error handler when a function in this API detects an error.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Retrieving Functions

`cdb_get_ddr()`	Get device description
`cdb_get_size()`	Get size of the CDB
`cdb_get_copy()`	Get copy of the CDB
`cdb_get_ncopy()`	Get copy of the CDB up to n bytes

The CDB block is a textual description of the system configuration contained in Device Description Records (DDRs). Each DDR corresponds to one device and has three parts:

Device Type	This is an unsigned integer representing the device (such as a WAN, graphic overlay, printer, magnetic disk, pointing device, or keyboard.)
Device Name	The name used to access the device.

Device Parameters	Indicates the functionality of the device. The contents are specific to the particular device type.

Reading Device Description Records

The following rules apply to DDR syntax and parameters:

- Though you may have several devices of the same type in the system, each device must have a unique name.
- Parts of the DDR are separated by a colon (:). Within the parameter part, separate parameters are also delimited by colons.
- If the parameter is boolean (yes/no type), it is represented by two characters to indicate the particular system capability.
- A numeric parameter consists of a pound sign (#) and the numeric part. The parameter name must be two characters. For example, in a graphics device entry, PL#8 means the presence of 8 image planes for a graphics processor. 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. For example, in a system entry, OS = “OS9” indicates that the operating system is OS-9.

The following is an example of DDR entries in a CDB:

```
0:sys:CP=”68340”:OS=”OS9”:RV=”3.0”:DV=”2.0”:SR#2048,
   1:VR#512,80:VR#512,81:
3:/gfx:AI=”MAUI”
4:/nvr:
5:/rem/genrem.mpm
9:/pipe:
```

Retrieving Information from the CDB

`cdb_get_ddr()` returns the name and the parameter string for the device if you specify the device type and its sequential number within this type.

`cdb_get_copy()` retrieves the entire CDB. This function copies the CDB into a buffer that is automatically allocated by the function. If you use this function, remember to de-allocate this memory with the `mem_free()` call. The copy returned is a single, NULL-terminated character string.

The disadvantage of `cdb_get_copy()` is that you don't know how large of an allocation it will perform nor do you have control over how this memory is allocated. This was resolved in MAUI 3.1 with the following two calls.

`cdb_get_size()` determines the buffer size required to make a copy of the entire CDB. This size include space for a NULL at the end of the CDB string.

`cdb_get_ncopy()` retrieves up to `N` bytes of the entire CDB. This function copies the CDB into a caller supplied buffer. The copy returned is a single, NULL-terminated character string. The size field is updated to indicate the number of bytes copied including the NULL at the end (i.e. `strlen()+1`).

Chapter 7: Using the Graphics Device API

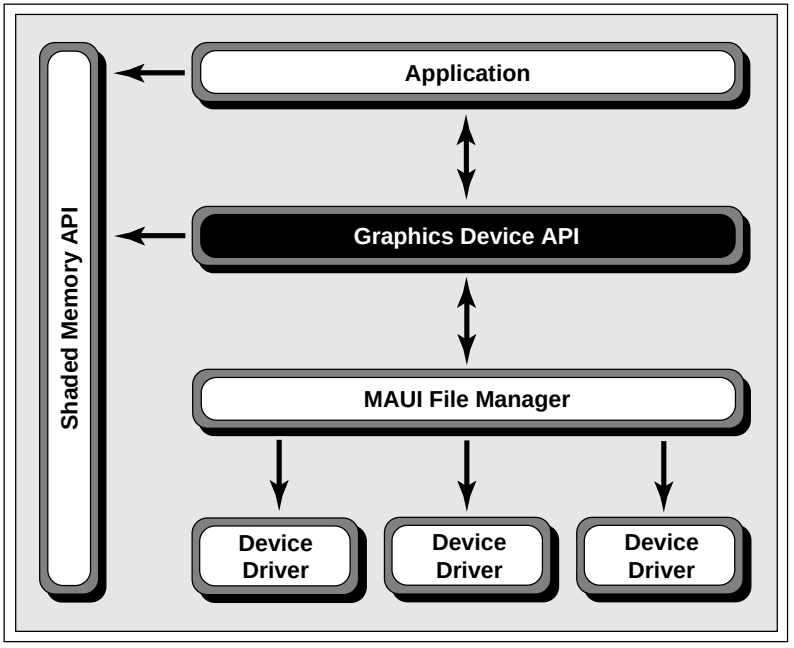
The Graphics Device API provides a hardware-independent interface to the graphic device(s) used by your application. Hardware independence is achieved by requiring the API to reach the hardware through the MAUI file manager and graphics device drivers.

This chapter classifies the Graphics Device API functions and explains how they are used.

Architecture

The Graphics Device API relies on the Shaded Memory API for memory allocation. The following figure shows the relationship between the Graphics Device API, the file manager, the application, and the Shaded Memory API.

Figure 7-1 Graphics Device API Dependencies.



An application uses the Graphics Device API to perform hardware-dependent operations. These operations are transparent to the application. To make the Graphics Device API hardware-independent, the API must talk to the hardware through the MAUI file manager and graphics device drivers.

Graphics Device API Functions

The functions provided by the Graphics Device API are classified into five groups: Initialize and Terminate, Graphics Device, Drawmap, Viewport, and Miscellaneous.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`gfx_init()`	Initialize Graphics Device API
`gfx_term()`	Terminate Graphics Device API
`gfx_set_error_action()`	Set action to take in error handler

All MAUI APIs are self-contained APIs. An API can require the presence of other APIs, but the internal aspects of one API are not affected or known by any other API.

Initializing the Graphics Device API

To initialize the Graphics Device API call `gfx_init()`. This function initializes the API and prepares it for use. No other functions in the Graphics Device API can be called before `gfx_init()` except `gfx_set_error_action()`.

Terminating the Graphics Device API

To terminate the Graphics Device API Call `gfx_term()`. `gfx_set_error_action()` can be called before `gfx_init()`. Any attempt to call Graphics Device API functions prior to initialization returns the `EOS_MAUI_NOINIT` error. `gfx_set_error_action()` sets the action to take in the error handler when a function in the Graphics Device API detects an error.

You may have noticed that the `gfx_init()` and `gfx_term()` functions were not used in the `hello` and `aloha` example programs in the previous chapters. `hello` and `aloha` use the `maui_init()` and `maui_term()` convenience functions to call `gfx_init()` and `gfx_term()`. These convenience functions initialize and terminate all of the MAUI APIs, rather than an individual API.

If you plan on using all the MAUI APIs, use `maui_init()` and `maui_term()`. If you plan to use only a few of the APIs, call the `init` and `term` functions for each API. `maui_init()` and `maui_term()` are used in examples throughout this chapter.

For More Information

See **Chapter 1, Printing to stderr** for more information about printing with `mem_` functions.

Graphics Device

`gfx_open_dev()`	Open graphics device
`gfx_clone_dev()`	Clone a graphics device
`gfx_close_dev()`	Close graphics device
`gfx_restack_dev()`	Change the position of a graphics device in the stack
`gfx_get_dev_cap()`	Get device capabilities

`gfx_get_dev_status()`	Get device status
`gfx_set_display_size()`	Set display screen size
`gfx_set_display_vpmix()`	Set viewport mixing on/off
`gfx_set_display_extvid()`	Set external video on/off
`gfx_set_display_bkcol()`	Set backdrop color
`gfx_set_display_transcol()`	Set transparent color
`gfx_alloc_mem()`	Allocate graphics memory
`gfx_dealloc_mem()`	De-allocate graphics memory
`gfx_set_decode_dst()`	Set destination for video decoding

Opening the Device

To open the device call `gfx_open_dev()`. Although most applications need to only open one device, you can open as many devices as you need. To close the device, call `gfx_close_dev()`.

Determining the Device Capabilities

To determine the device capabilities call `gfx_get_dev_cap()`. This function provides your application with the information it needs to adjust to various devices.

For More Information

The `show_img` program introduced later in this chapter shows how to use this information to enable your application to make these adjustments.

Receiving Current Values for Parameters

To receive the current values for parameters that affect the device characteristics call `gfx_get_dev_params()`.

Setting the Parameter Values

To set the parameter values use the following functions:

`gfx_set_display_size()` Set device size

`gfx_set_display_vpmix()` Set device mixing mode

`gfx_set_display_extvid()` Set device external video mode

`gfx_set_display_bkcol()` Set device backdrop color

`gfx_set_display_transcol()` Set device transparency color

If you do not set one or more of these parameters, MAUI uses the default value.When you call any of these functions to change the device parameters, the changes are not immediately visible on the display. MAUI queues the changes until you call `gfx_update_display()`. This enables you to code your application so it can synchronize several display modifications

Drawmap

`gfx_create_dmap()`	Create drawmap
`gfx_destroy_dmap()`	Destroy drawmap
`gfx_set_dmap_size()`	Set coding method and drawmap size
`gfx_set_dmap_pixmem()`	Set pixel memory pointer in drawmap

A drawmap is an object that defines a rectangular area of pixel memory. This object can be created by calling `gfx_create_dmap()` or created directly by the application using initialized data. The data structure name for a drawmap is `GFX_DMAP`.

The following is a drawmap and palette data structure:

```
typedef struct _GFX_DMAP
{
   GFX_CM coding_method;/* Coding method */
   GFX_DIMEN width;     /* Width in pixels */
   GFX_DIMEN height;    /* Height in pixels */
   size_t line_size;    /* Size of line in bytes */
   GFX_PIXEL *pixmem;   /* Ptr to pixel memory */
   u_int32 pixmem_shade;/* Shade used for pixmem */
   size_t pixmem_size; /* Size of pixmem */
   GFX_PALETTE *palette;/* Ptr to color palette */
}  GFX_DMAP;
typedef struct _GFX_PALETTE
{
   u_int16 start_entry;/* Starting entry */
   u_int16 num_colors; /* Number of colors */
   GFX_COLOR_TYPE color_type;/* Type of color table */
   union
   {
      GFX_RGB *colors;  /* Array of num_colors RGB */
      GFX_YUV * yuv;    /* Array of num_colors YUV */
      GFX_A1_RGB *a1_rgb; /*Array of num_colors A1_RGB*/
   }
}  GFX_PALETTE;
```

For More Information

Refer to the ***MAUI Programming Reference Manual*** for a complete description of all MAUI data structures.

There are many methods you can use to create and manage drawmaps. Here are three common approaches that you may find useful for your applications:

- Create a drawmap of a known size (preferred).
- Create a drawmap with a maximum size.
- Create a drawmap and set the size later.

Creating a Drawmap of Known Size (preferred)

In most cases, you know the dimensions and coding method for a drawmap before you open it. To create a drawmap that is the size of the display:

Step 1. Call `gfx_get_dev_cap()` to determine the size parameters.

Step 2. Call `gfx_create_dmap()` to create the drawmap, and pixel memory shade.

The following code creates a drawmap of the coding method, size, and palette defined in the data structure `GFX_DMAP`:

```
GFX_DMAP *dmap;
gfx_create_dmap(&dmap, 0x81);
```

The parameter `0x81` tells MAUI to allocate the memory from shade `0x81`. Only one allocation is made, and it is the combined size of the drawmap structure, pixel memory, and palette size.

This example is the preferred method for creating a drawmap because this method:

- allocates everything you need for the drawmap in one function call
- only allocates one memory segment so it is also the most efficient way to create a drawmap

Creating a Drawmap with Maximum Size

Unfortunately, it may not always be possible or practical to use the preferred method previously described. Cases arise when you simply want to create a drawmap of the maximum size you need, and change its parameters when required.

This method is similar to the previous method except that you use maximum values for the size and coding method. For example, if the largest size you need is `720 x 480` with a coding method of `GFX_CM_RGB555`, create your drawmap as follows:

```
GFX_DMAP *dmap;
gfx_create_dmap(&dmap, 0x81);
```

Calling the create function with these parameters ensures that the pixel memory and palette will be large enough. If you decide later to set the size to `360 x 240` with a coding method of `GFX_CM_4BIT`, make the following function call:

```
gfx_set_dmap_size(&dmap, GFX_CM_4BIT, 360, 240);
```

You can change the dimensions of this drawmap as often as necessary in your application, as long as you never try to make it larger than its original size.

Creating Drawmap and Setting Size Later

In the previous two methods, memory for the pixel memory and the palette is allocated as part of the drawmap structure. In some cases, however, you may want to separate this allocation into two steps because:

- you don't know the size of the pixel memory you are going to need.
- you are creating the drawmap structure with initialized data. In this case, you eventually want to allocate the pixel memory.

Use `gfx_create_dmap()` to create a drawmap structure with no pixel memory or palette by calling it as follows:

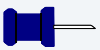
Step 1.
```
GFX_DMAP *dmap;
gfx_create_dmap(&dmap, 0);
```

Step 2. Set drawmap size and coding method, allocate pixel memory, and set the palette by calling the following functions:
```
gfx_set_dmap_size(&dmap, GFX_CM_8BIT, 360, 240);
gfx_set_dmap_pixmem(&dmap, GFX_PIXMEM_CALC, 0x81);
```

Be careful when using this method (or similar methods). When you use `gfx_set_dmap_pixmem()`, remember that the memory allocation for this item is separate from the drawmap structure. To avoid a memory leak, destroy the pixel memory when you no longer need it.

For example:

- If you use `gfx_create_dmap()` to create a drawmap, call `gfx_destroy_dmap()` to destroy it.
- If you use `gfx_set_dmap_pixmem()` to allocate or assign pixel memory to a drawmap, use `mem_free()` or `mem_sfree()` to de-allocate it.



Note

To avoid memory problems, be sure to call the corresponding MAUI function to destroy or de-allocate the function when it is no longer needed.

Viewport

`gfx_create_vport()`	Create viewport
`gfx_clone_vport()`	Clone a viewport
`gfx_destroy_vport()`	Destroy viewport
`gfx_get_vport_status()`	Get viewport status
`gfx_set_vport_position()`	Set viewport position
`gfx_set_vport_size()`	Set viewport size
`gfx_set_vport_dmpos()`	Set viewport drawmap position
`gfx_set_vport_state()`	Set viewport state
`gfx_set_vport_intensity()`	Set viewport intensity

`gfx_set_vport_dmap()`	Set drawmap for use in a viewport
`gfx_restack_vport()`	Change placement in viewport stack

Viewports are defined areas of the screen that display still graphic images loaded from drawmaps. A display screen may contain a single viewport that is as large as the screen, or several viewports of various sizes and positions. The number, size, and transparency of viewports are determined by the hardware capabilities. Unlike drawmaps, viewports are private data structures. This means that applications cannot directly access the structure members. The structures can only be examined or changed by using the functions provided in the Graphics Device API.

Creating a Viewport

To create a viewport call `gfx_create_vport()`.You may create as many viewports as you wish, but the hardware will limit what you can activate. A viewport is not visible until it is activated.

Destroying a Viewport

To destroy a viewport call `gfx_destroy_vport()`.

Receiving the Current Value of Viewport Parameters

To receive the current value of viewport parameters call `gfx_get_vport()`. Viewports have several parameters or characteristics.

Setting Viewport Parameters and Characteristics

To set viewport parameters and characteristics use the following viewport functions:

`gfx_set_vport_position()`	Set viewport position
`gfx_set_vport_size()`	Set viewport size
`gfx_set_vport_state()`	Set viewport state

`gfx_set_vport_intensity()`	Set viewport intensity
`gfx_set_vport_dmap()`	Set drawmap used in a viewport
`gfx_set_vport_dmpos()`	Set drawmap position

When you call any of the above functions to change viewport parameters and characteristics, the changes are not immediately visible on the display. This behaves similar to changes to device parameters where MAUI queues up viewport changes until you call `gfx_update_display()`.

This ability to synchronize changes to the display allows applications to make clean transitions from one arrangement to another.

Miscellaneous

`gfx_sync_retrace()`	Wait for vertical retrace
`gfx_update_display()`	Update display
`gfx_calc_pixmem_size()`	Calculate size of pixel memory
`gfx_find_vport()`	Find the viewport at the specified position
`gfx_cvt_dppos_dmpos()`	Convert display to drawmap position
`gfx_cvt_dmpos_dppos()`	Convert drawmap to display position

If a vertical retrace capability is present in the hardware, the `gfx_sync_retrace()` function synchronizes the changes made by `gfx_update_display()` with the vertical retrace.

Use the `gfx_sync_retrace()` function in other areas of your application where you want to perform this type of synchronization (for example, to control the animation speed).

Example Program

The `show_img` program uses the functions that were explained earlier in this chapter. In this example, the `show_img` program loads an image from a data file into a drawmap and shows the image centered on the display. The executable source for this program is not included, but is printed here to illustrate the concepts explained in this chapter.

Load the Image from the Data File

```
mem_create_shade(SHADE_PLANEA, MEM_SHADE_NORMAL,
0x80,
    1024*4, 1024*4, MEM_OV_ATTACHED, TRUE);
```

Before loading the image to the memory, create a memory shade in the graphics memory (color 0x80).

```
{
   error_code ec;
   if ((ec = load_image(&dmap, MEM_DEF_SHADE,
      SHADE_PLANEA,
         (argc != 1) ? argv[1] :
      "board.d"))
         != SUCCESS)
   {
      gfx_term();
      exit (ec);
   }
}
```

`load_image()` creates a drawmap containing the image held in the `board.d` file.

The Graphics Device API does not assume a particular module or file format for your images, so it only defines the format of the drawmap structure, `GFX_DMAP`. Therefore, code is required (in this case, `load_image()`) that reads an image from a file and creates the drawmap object to hold it.

The `load_image()` function is not part of any MAUI API, but is presented here so that you can get an idea of what you need to provide so you can load IFF and other types of images (such as DYUV and JPEG).

Show the Image on the Graphics Display

```
{
   /* temporary device name */

   char devname[CDB_MAX_DNAME];

   cdb_init();
   cdb_get_ddr(CDB_TYPE_GRAPHIC, 1, devname,
      NULL);
   cdb_term();
   gfx_open_dev(&gfxdev, devname);
}
```

Before the graphics device is used, it must be opened. To retrieve the device name from the Configuration Description Block, use CDB API functions.

`gfx_open_dev()` opens the graphics device. The first parameter is a pointer to where the device ID will be written. In this case, it is `&dev`. The variable `dev` is used throughout the program for all operations that affect the graphics device.

```
gfx_get_dev_cap(&devcap, gfxdev);
cm_info = devcap->cm_info;
num_cm = devcap->num_cm;
while (cm_info->coding_method !=
   dmap->coding_method)
{
   if (--num_cm == 0)
   {
      fprintf (stderr,
         "Cannot display coding method %d\n",
         dmap->coding_method);
      gfx_term();
      exit (EOS_MAUI_BADCODEMETH);
   }
```

```
        cm_info++;
        }

    /* compute size and centering information */

    width = dmap->width*cm_info->dm2dp_xmul;
        if (width > devcap->res_info->disp_width)
        width = devcap->res_info->disp_width;
        x = (devcap->res_info->disp_width - width)/2;
    height = dmap->height*cm_info->dm2dp_ymul;
        if (height > devcap->res_info->disp_height)
    height = devcap->res_info->disp_height;
    y = (devcap->res_info->disp_height - height)/2;

    fprintf (stderr, "dmap: cm %d w %d h %d\n",
          dmap->coding_method,
          dmap->width, dmap->height);
    fprintf (stderr, "vport: x %d y %d w %d h %d\n",
          x, y, width, height);

    /* Create vport */

    gfx_create_vport(&vport, gfxdev, x, y, width,
        height, GFX_VPORT_FRONT);
}

/* Map the drawmap to the viewport and show it */

gfx_set_vport_dmap(vport, dmap, 0, 0);
{
    GFX_COLOR color = {GFX_COLOR_RGB,0x7f7f7f};
    gfx_set_display_bkcol(gfxdev, &color);
}
gfx_set_vport_state(vport, TRUE);
gfx_update_display(gfxdev, TRUE);
```

Because the capabilities of graphics devices vary, `gfx_get_dev_cap()` is called to get information about this device. The width and height of the first supported (default) resolution is used to center the viewport upon its creation.

To put the drawmap containing the image into this viewport and make it visible `gfx_set_vport_dmap()`, `gfx_set_vport_state()`, and `gfx_update_display()` are called respectively.

Destroy Viewport

```
_os9_sleep(20);

/* Destroy everything, terminate each API and exit */

gfx_set_vport_state(vport, FALSE);
gfx_destroy_vport(vport);
mem_free(dmap->pixmem);
gfx_destroy_dmap(dmap);
mem_destroy_shade(SHADE_PLANEA);
gfx_close_dev(gfxdev);
gfx_term();
```

This code centers the image on the display. After waiting 20 seconds, the viewport is destroyed and the API is terminated.

Chapter 8: Using the Bit-BLT API

The Bit-Block-Transfer (Bit-BLT) API performs all block transfer and block drawing operations. It is the foundation for all APIs that perform drawing, including the Drawing, Text, and Animation APIs.

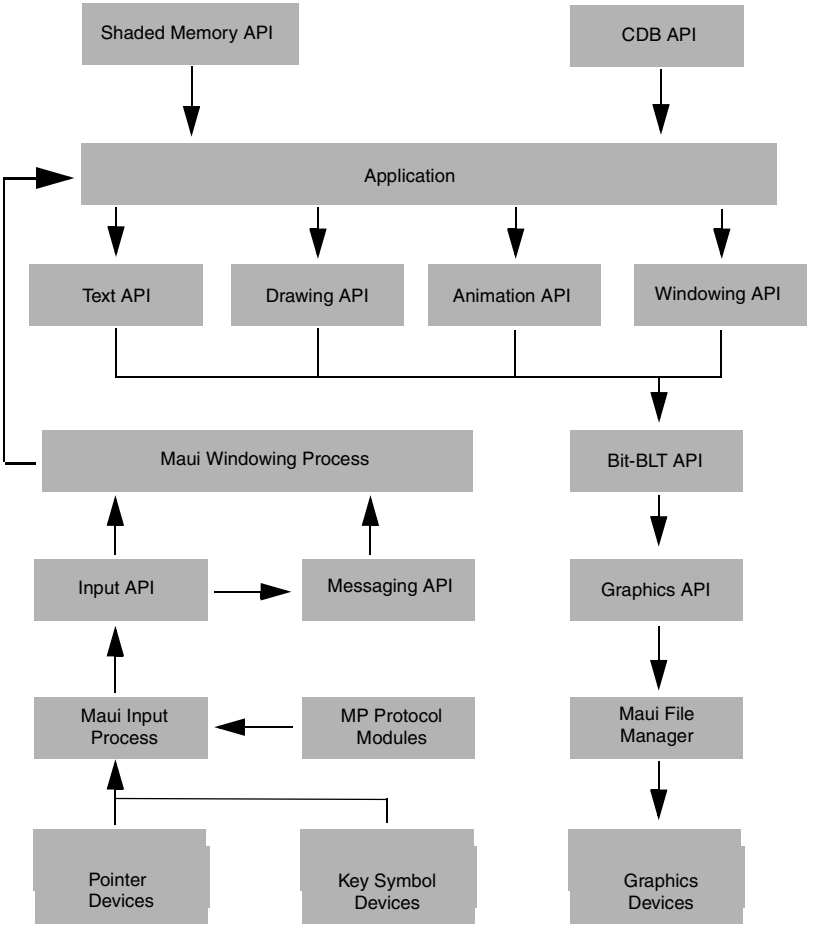
The Bit-BLT API also interfaces with the graphics driver to take advantage of hardware Bit-BLT engines or to draw to graphics memory that is not accessible to the CPU.

This chapter classifies the Bit-BLT API functions and explains how they are used. Example programs are included for each Block Transfer Operation (`fastcopy` and `fastdraw`). You can execute these programs to illustrate the effects of the Bit-BLT functions.

Architecture

The relationship between the Bit-BLT API, Shaded Memory API, Graphics Device API, and the Graphics Driver Interface is shown in the figure Figure 8-1.

Figure 8-1 Bit-BLT Dependencies



The Bit-BLT API can perform all block drawing and copy operations to drawmaps. However, if the CPU cannot access the pixel memory in the drawmap, the Bit-BLT function calls the driver to perform the operation.

Bit-BLT API Functions

The Bit-BLT API functions are classified into four groups: Initialize and Terminate, Bit-BLT Context, Block Transfer Operations, and Draw Block Operations.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`blt_init()`	Initialize Bit-BLT API
`blt_term()`	Terminate Bit-BLT API
`blt_set_error_action()`	Set action to take in error handler

Initializing the Bit-BLT API

To initialize the Bit-BLT API call the `blt_init()` function to prepare it for use. No other functions in this API can be called before `blt_init()` except `blt_set_error_action()`.

Terminating the Bit-BLT API

To terminate the Bit-BLT API call `blt_term()`.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Bit-BLT Context Object

`blt_create_context()`	Create Bit-BLT context object
`blt_destroy_context()`	Destroy Bit-BLT context object
`blt_get_context()`	Get Bit-BLT context parameters
`blt_set_context_cpymix()`	Set mixing mode for copy operations
`blt_set_context_expmix()`	Set mixing mode for expand operations
`blt_set_context_drawmix()`	Set mixing mode for draw operations
`blt_set_context_pix()`	Set pixel value for drawing
`blt_set_context_exptbl()`	Set background and foreground pixel values
`blt_set_context_src()`	Set source drawmap
`blt_set_context_trans()`	Set transparent pixel value
`blt_set_context_mask()`	Set mask drawmap
`blt_set_context_ofs()`	Set offset color
`blt_set_context_dst()`	Set destination drawmap

The Bit-BLT context object is largely responsible for the speed of this API. It is a private data structure containing the current parameters for performing block transfer operations. These transfer operations are fast because:

- The parameters are stored in the context object

- The functions used to change the parameters also compute many other variables stored in the context

Computing these values when the context parameters change is more efficient than doing it on every block transfer operation. This efficiency is noticeable because applications typically change the parameters in the context at a much lower frequency than they perform block transfer operations. Applications are not limited to one context object. You can create as many context objects as you need.

Creating Bit-BLT Context Objects

To create Bit-BLT context objects call `blt_create_context()`.

Destroying Bit-BLT Context Objects

To destroy Bit-BLT context objects call `blt_destroy_context()`.

Modifying Bit-BLT Context Objects

To modify a Bit-BLT context object call the `blt_set_context_*()` functions. There is one function for each parameter (for example, `blt_set_context_dst()` sets the destination drawmap).

Block Transfer Operations

The `blt_copy_block()`, `blt_expd_block()`, and `blt_draw_block()` are the main block transfer functions. All other block transfer functions are variations of these three. For example, `blt_draw_hline()` is the same (yet more efficient) as `blt_draw_block()` with a height of `1`.

`blt_copy_block()`	Copy pixel block
`blt_copy_next_block()`	Copy next pixel block
`blt_expd_block()`	Expand pixel block
`blt_expd_next_block()`	Expand next pixel block

`blt_draw_block()`	Draw pixel block
`blt_draw_hline()`	Draw horizontal line of pixels
`blt_draw_vline()`	Draw vertical line of pixels
`blt_draw_pixel()`	Draw single pixel
`blt_get_pixel()`	Get pixel value

The following sections describe each of these Bit-BLT operations. Example programs are included to help you understand each function.

Copy Block Operations

The copy block operations include all functions that copy rectangular blocks of pixels from the source drawmap to the destination drawmap.

`blt_copy_block()`	Copy a block of pixels
`blt_copy_next_block()`	Copy the next block of pixels

These functions use the current mixing mode specified for copy operations in the Bit-BLT context. The mixing mode is set with the `blt_set_context_cpymix()` function.

Fastcopy

The `fastcopy` program located at `MWOS/SRC/MAUI/DEMOS/FCOPY` shows you how quickly MAUI performs copy operations. Print a copy of the source to understand how `fastcopy` works.

Expand Block Operations

Expand block operations include all functions that copy rectangular blocks of pixels from the source drawmap to the destination drawmap, while expanding the pixels. This is required when the depth of the source pixels is less than that of the destination. Functions that expand pixels as they are copied include the following:

`blt_expd_block()`	Expand pixel block
`blt_expd_next_block()`	Expand next pixel block

These functions use the current mixing mode specified for expand operations in the Bit-BLT context. This value is set with the function `blt_set_context_expmix()`.

Draw Block Operations

The draw block operations draw rectangular blocks of pixels to the destination drawmap. Functions that draw blocks include the following.

`blt_draw_block()`	Draw block of pixels
`blt_draw_hline()`	Draw horizontal line of pixels
`blt_draw_vline()`	Draw vertical line of pixels
`blt_draw_pixel()`	Draw a single pixel

These functions use the current mixing mode specified for draw operations in the Bit-BLT context. This value is set with the function `blt_set_context_drwmix()`.

Fastdraw

The `fastdraw` program demonstrates how quickly MAUI performs draw block operations. The executable source for this program is located in the directory `MWOS/SRC/MAUI/DEMOS/FDRAW`.

Chapter 9: Using the Drawing API

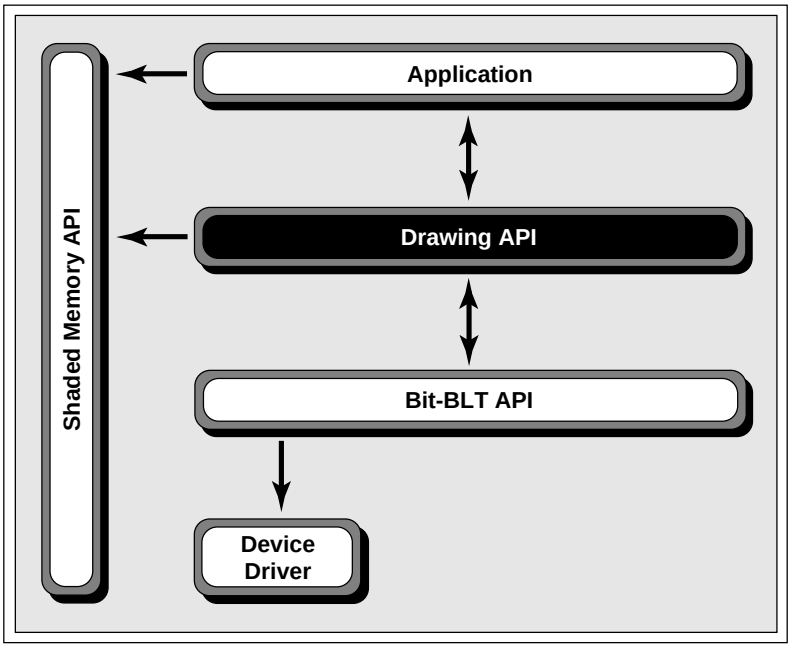
The Drawing API contains functions for drawing geometric shapes to a drawmap. These shapes include lines, polylines, points, circles, rectangles, and polygons. Shapes can be drawn in outline or solid mode and attributes such as patterns and line styles can be applied to these operations.

This chapter classifies the Drawing API functions and explains how they are used.

Architecture

The relationship between the Drawing API, Shaded Memory API, Bit-BLT API, and the Graphics Device API is shown in Figure 9-1. The Drawing API depends on the Shaded Memory API for memory allocation, the Bit-BLT API for block transfer operations, and the Graphics Device API for device capabilities information.

Figure 9-1 Drawing API Dependencies.



Drawing API Functions

The Drawing API functions are classified into three groups: Initialize and Terminate, Drawing Context, and Shape Drawing.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`drw_init()`	Initialize drawing API
`drw_term()`	Terminate drawing API
`drw_set_error_action()`	Set error action

Like all other MAUI APIs, the Drawing API automatically initializes and terminates APIs it depends on.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Initializing the Drawing API

To initialize the Drawing API call `drw_init()`. After initializing the drawing API, other steps you might take to prepare for drawing include (not necessarily in this order):

- setting display parameters
- creating a drawmap and initializing memory for a drawmap
- creating a viewport and assigning a drawmap to the viewport
- activating the viewport

Drawing Context Object

`drw_create_context()`	Create drawing context object
`drw_destroy_context()`	Destroy drawing context object
`drw_get_context()`	Get drawing context parameters
`drw_set_context_fm()`	Set fill mode
`drw_set_context_ls()`	Set line style
`drw_set_context_dash()`	Set dash pattern
`drw_set_context_pix()`	Set pixel value
`drw_set_context_ofs()`	Set offset pixel value
`drw_set_context_trans()`	Set transparent pixel value
`drw_set_context_mix()`	Set mixing mode
`drw_set_context_dst()`	Set destination drawmap

The drawing context object is a set of common data defining current drawing parameters for the drawing functions. If you use this context object, you can increase drawing performance.

Context Parameters

Several parameters are available for controlling the way you draw shapes. For example, depending on the line style parameter value, the `drw_line()` function produces a solid or dashed line.

Applications cannot access context variables directly. However, you can use the Drawing API function calls to query and modify drawing contexts.

Following are the drawing parameters and the functions you use to set them.

Table 9-1 Drawing Parameters

drw_set_context	Sets this parameter	Default value
`_fm()`	Fill mode (solid or outline)	outline
`_ls()`	Line style (solid or dashed)	solid
`_dash()`	Dash pattern and magnification	`0x555555554`
`_mix()`	Mix mode (mix the drawing with the existing image)	`BLT_MIX_REPLACE`
`_pix()`	Drawing pixel value	`1`
`_trans()`	Transparent pixel value	`0`
`_ofs()`	Offset pixel value	`0`
`_dst()`	Destination drawmap	`NULL`

Creating Context Objects

To create context objects call `drw_create_context()`.

Destroying Previously Created Context Objects

To destroy previously created context objects call `drw_destroy_context()`.

Querying for Context Object Parameters

To query for context object parameters call `drw_get_context()`.

Shape Drawing

The following drawing operations enable you to draw shapes:

`drw_point()`	Draw a point
`drw_line()`	Draw a line
`drw_rectangle()`	Draw a rectangle
`drw_circle()`	Draw a circle
`drw_polyline()`	Draw a polyline
`drw_polygon()`	Draw a polygon

Each of these functions behaves according to the context parameters set in the appropriate `drw_context_*()` function. If parameters are not set, the drawing operations uses the default parameters.

For example, `drw_polygon()` draws outlined polygons when fill mode is set to `DRW_FM_OUTLINE`, and filled polygons when the mode is set to `DRW_FM_SOLID`.

Drawing functions check to determine if shapes are located within the screen and, if necessary, are clipped to the boundaries of the drawmap. If a shape is clipped, the clipped area remains intact.

Chapter 10: Using the Text API

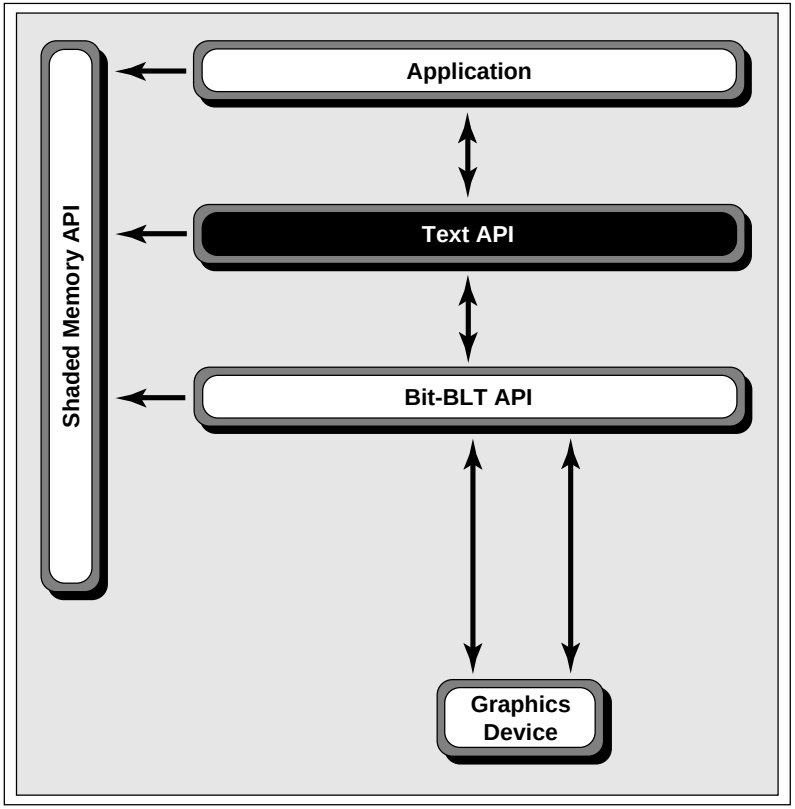
The Text API contains functions for writing multi-byte and wide-character strings to any drawmap. These functions support multiple fonts, (mono and proportionally spaced), and methods for controlling the padding between characters.

This chapter classifies the Text API functions and explains how they are used.

Architecture

The Text API relies on the Shaded Memory API for memory allocation and the Bit-BLT API for block transfer operations. Refer to Figure 10-1 Text API Dependencies to see the relationships between these APIs.

Figure 10-1 Text API Dependencies



Text API Functions

The functions provided by the Text API are classified into five groups: Initialize and Terminate, Text Context Object, Text Font Object, Text Drawing Operations, and Miscellaneous Functions.

For More Information

For a function list, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`txt_init()`	Initialize the Text API
`txt_term()`	Terminate the Text API
`txt_set_error_action()`	Set error action

Like all other MAUI APIs, the Text API automatically initializes and terminates APIs it depends on.

For More Information

See **Chapter 1** for more information about printing with `mem_` functions.

Initializing the Text API

To initialize the Text API call `txt_init()`.

Terminating the Text API

To terminate the Text API call `txt_term()`.

Text Context Object

`txt_create_context()`	Create context object
`txt_destroy_context()`	Destroy context object
`txt_get_context()`	Get text context parameters
`txt_set_context_cpad()`	Set character padding in text context
`txt_set_context_dst()`	Set destination drawmap in text context
`txt_set_context_exptbl()`	Set pixel expansion table
`txt_set_context_font()`	Set font in text context
`txt_set_context_mix()`	Set mixing mode in text context
`txt_set_context_ofs()`	Set offset pixel in text context
`txt_set_context_trans()`	Set Transparent pixel in text context

The behavior of the Text API is defined by the text context object. The text context object is a private data structure that encapsulates the properties listed above.

Creating an Instance of a Text Context Object

To create an instance of a text context object call `txt_create_context()`.

Destroying Text Context Objects

To destroy text context objects call `txt_destroy_context()`.

Modifying Members of the Text Context Object

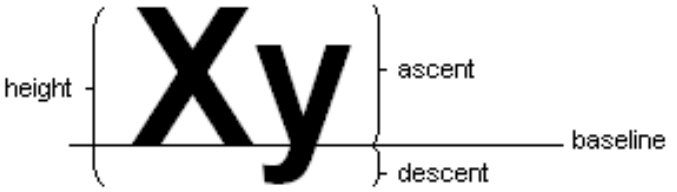
To modify members of the text context object call the `txt_set_context_*()` functions.

Text Font Object

`txt_create_font()`	Create a font object
`txt_destroy_font()`	Destroy a font object

The text font object is a public data structure that defines all the characters in a font. **Figure 10-2** details the important character attributes supported by the MAUI Text API.

Figure 10-2 Character Attributes



ascent	the number of pixels in the character cell above the baseline
descent	the number of pixels in the character cell below the baseline
baseline	an imaginary line between pixels that make up the ascent and the descent
height	the sum of ascent pixels and descent pixels

Below are some additionally important data items for this structure:

font type	fixed or proportional
width	maximum character width
default character	default character in font
number of ranges	number of character ranges in font
range table	array of character ranges in font

Creating a Text Font Object

To create a text font object call `txt_create_font()`. An application can use this function to create a text font object or it can create an instance directly. After creation, the text font object's members should be initialized with data (probably stored in a font module).

Text Drawing Operations

Note

The text API draws from the character's baseline, not the standard 0,0 coordinates.

`txt_draw_mbs()`	Draw a multi-byte string
`txt_draw_wcs()`	Draw a wide character string
`txt_get_mbs_width()`	Get pixel length of a multi-byte string
`txt_get_wcs_width()`	Get pixel length of a wide character string

The Text API supports two types of encoding methods for character strings: multi-byte and wide character. Both types are supported by ANSI-C and allow the API to support multiple locals.

`txt_draw_mbs()` and `txt_draw_wcs()` are two functions in the Text API you can use to place text of a specified font at `x,y` coordinates. The call to these functions can include a parameter that determines the padding between characters in the output string. This parameter is an array of values indicating the padding between the first and second characters, second and third, and so forth. Padding for all strings drawn with a context can be defined with a call to `txt_set_context_cpad()`.

Any text drawn that does not fit into the destination drawmap is clipped. You can determine the width in pixels of any string with a call to either `txt_get_mbs_width()` or `txt_get_wcs_width()`. As an option, these functions can also return the width of each character in the specified string.

Chapter 11: Using the Animation API

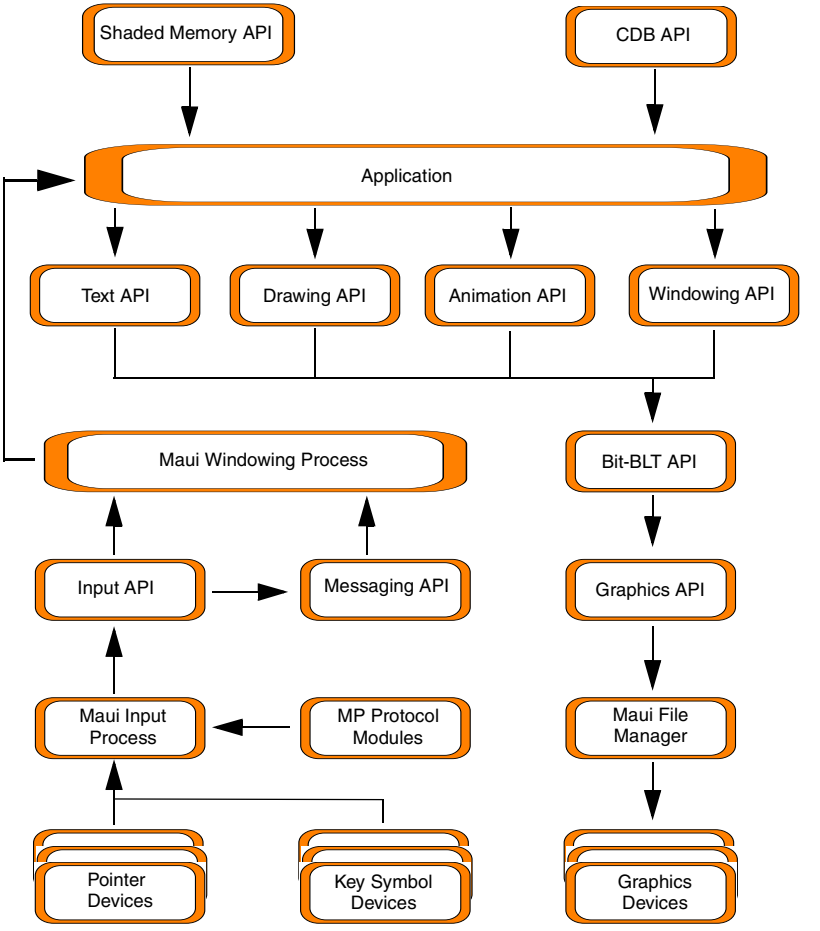
The Animation API makes it possible for the application to move images quickly and smoothly on the display. The sprite mechanism creates the illusion of continuous movement by changing image frames and locations.

This chapter classifies the Animation API functions and explains how they are used.

Architecture

The Animation API relies on the Shaded Memory API for memory allocation, the Bit-BLT API for block transfer operations, and the Graphics Device API for device capabilities information. Where the Animation API fits into the MAUI API architectural scheme is shown in Figure 11-1 Animation API Dependencies.

Figure 11-1 Animation API Dependencies



Animation API Functions

The functions provided by the Animation API are classified into four categories: Initialize and Terminate, Sprites, Animation Groups, and Animation Objects.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`anm_init()`	Initialize Animation API
`anm_term()`	Terminate Animation API
`anm_set_error_action()`	Set error action

The system-wide functions `maui_init()` and `maui_term()` initialize and terminate all the MAUI APIs, including the Animation API.

Initializing the Animation API

To initialize the Animation API call `anm_init()`. This automatically initializes the Shaded Memory API, Bit-BLT API, and the Graphics Device API.

Terminating the Animation API

To terminate the Animation API call `anm_term()`. This automatically terminates the Animation API in addition to the Shaded Memory API, Bit-BLT API, and the Graphics Device API unless they were initialized before the Animation API. The three APIs remain initialized in this situation.

`anm_set_error_action()` can be called before `anm_init()`. Any attempt to call Animation API functions prior to initialization returns the `EOS_MAUI_NOINIT` error. `anm_set_error_action()` sets the action to take in the error handler when a function in the Animation API detects an error. You can find the initialization and termination examples in the `animate.c` program at the end of the chapter.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Sprites

`anm_create_sprite()`	Create a Sprite
`anm_destroy_sprite()`	Destroy a Sprite

Sprites are objects that contain a set of frames which represent different phases of the image motion. When these frames display in succession at different screen locations, the illusion of motion appears. Technically, frames are parts of a source drawmap where images are stored. The source drawmap pointer is stored in a sprite data structure. In the simplest case, the motion can be imitated with just one frame changing its screen locations. But if you want more complex animation (3-D imitation, morphing, behavioral image changes), multiple frames will do the job.

Creating a Sprite

To create a Sprite call `anm_create_sprite()`.

Destroying a Sprite

To destroy a sprite call `anm_destroy_sprite()`.

Sprites are part of a more complex entity called an animation object.

Animation Groups and Objects

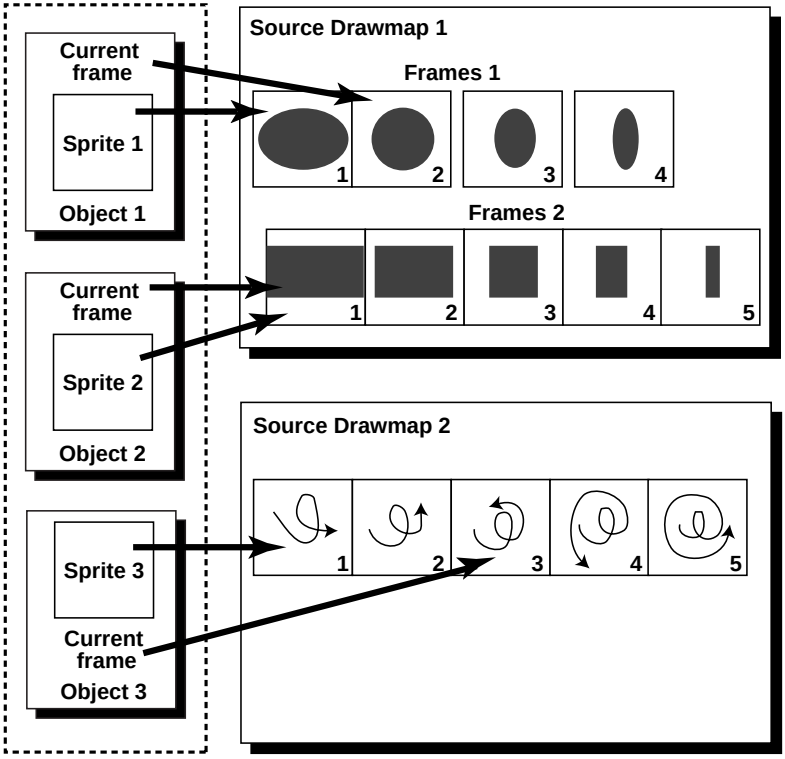
The animation object represents a separate moving image. You can group animation objects together in animation groups to simplify their processing.

To an application developer, an animation object consists of:

- A pointer to a sprite
- Destination drawmap pointer
- A current frame to be displayed
- The position of the object on the screen
- An object drawing method (to be explained below)
- Active/passive state flag

The relationship between sprites, objects, frames, and source drawmaps is shown in Figure 11-2 Animation Groups and Objects. Frames do not necessarily address a contiguous area of a drawmap (like Frames 1).

Figure 11-2 Animation Groups and Objects



Animation Group

`anm_create_group()`	Create the animation group
`anm_get_group()`	Get animation group parameters
`anm_set_group_bkg()`	Set group background color
`anm_set_group_dst()`	Set group destination drawmap
`anm_destroy_group()`	Destroy an animation group
`anm_draw_group()`	Draw objects in a group
`anm_process_group()`	Process objects in a group

Before any new object is created, you must create a group to which this object belongs.

Creating an Animation Group

To create an animation group call `anm_create_group()`.

Destroying the Existing Animation Group

To destroy the existing animation group call `anm_destroy_group()`.

Drawing All the Active Objects of a Group on a Screen

To draw all the active objects of a group on a screen call `anm_draw_group()`.

Processing Objects in a Group

To process objects in a group call `anm_process_group()`. This calls a behavior function for every object in a group and the objects on a screen according to results of these calls. Since all objects within the same group are drawn to the same drawmap, this drawmap is stored in a group object and can be set with `anm_set_group_dst()`.

Assigning a Background Image or Color

To assign a background image or color to an animation group call `anm_set_group_bkg()`. This function specifies either a drawmap or a pixel value for a background within a single graphics memory plane.

Animation Object

`anm_create_object()`	Create an animation object
`anm_destroy_object()`	Destroy an animation object
`anm_get_object()`	Get animation object parameters
`anm_restack_object()`	Re-stack animation objects
`anm_set_object_state()`	Set state for an object
`anm_set_object_sprite()`	Set sprite for an object
`anm_set_object_frame()`	Set frame for an object
`anm_set_object_bhv()`	Set behavior for an object
`anm_set_object_pos()`	Set position for an object
`anm_set_object_meth()`	Set drawing method for an object

Setting the Object State

To set the object state call `anm_set_object_state()`. This sets an object state to be either active or passive. Only active objects within a group are processed and drawn on a screen.

Assigning a Sprite to an Object

To assign a sprite to an object call `anm_set_object_sprite()`.

Changing a Current Frame

To change a current frame call `anm_set_object_frame()`.

Updating the Position of an Object on the Screen

To update the position of an object on the screen call `anm_set_object_pos()`.

To change the object display order

To change the object display order call `anm_restack_object()`. Objects on a screen can overlap each other because of the incorrect order they are displayed.

Transparency Checks

There are two drawing methods for drawing an object on the screen: with or without a transparency check.

- With a transparency check:
  Objects that overlap each other should check for the transparency of their pixels when drawing. You can choose between two different transparency check implementations: transparent pixels and transparency mask. The transparency pixel implementation checks to see if pixels have a transparent value, and, if `yes`, ignores them. In the transparency mask implementation, a bit set to `1` in the mask transfers corresponding source to the destination.
- Without transparency check:
  Ignores the pixel transparency. For each bit in the mask that is `0`, the corresponding bit in the source is ignored.

Defining the Drawing Method

To define what drawing method to use when an object is being drawn on a screen call `anm_set_object_meth()`.

Defining a Behavior

The fundamental concept of animation is movement. Since different images are moving differently, every object has a special function assigned. This function defines the object's behavior, such as speed, acceleration, rotation, and frame change.

To supply a behavior function and tie it with an object call `anm_set_object_bhv()`.

Chapter 12: Using the Messaging API

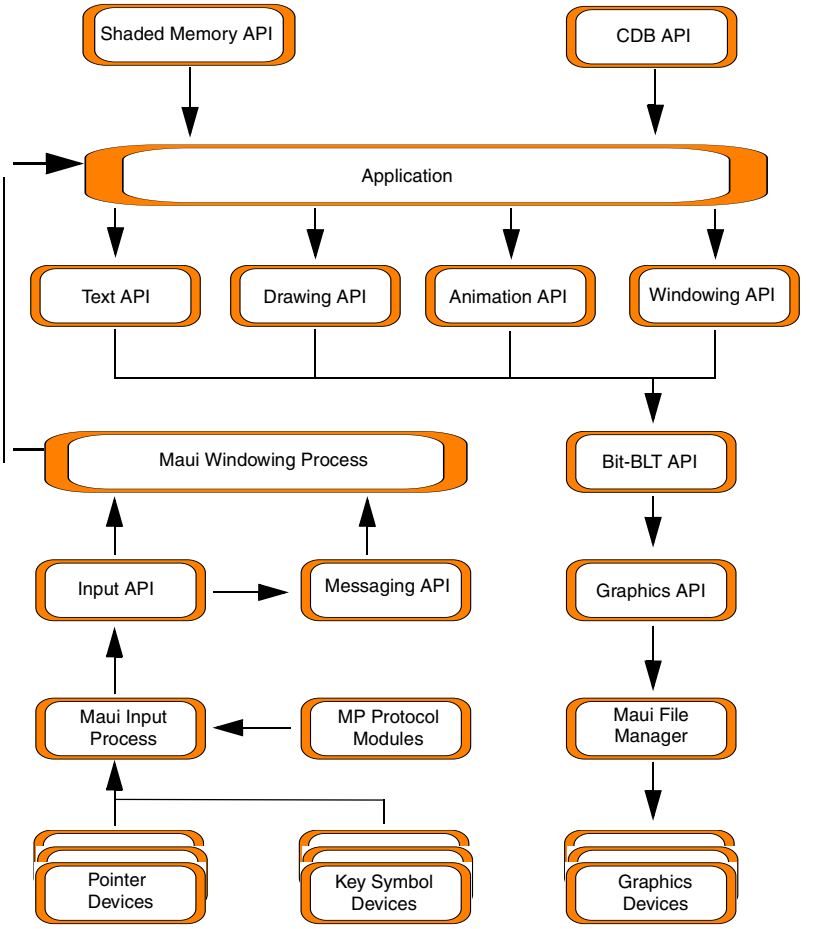
The Messaging API enables the application to read and write to mailboxes that are named and visible to all applications. MAUI uses mailboxes to pass information between and within application processes.

This chapter classifies the Messaging API functions and explains how they are used. An example program at the end of this chapter creates a mailbox, writes a message to the mailbox, and reads the message.

Architecture

The Messaging API relies on the Shaded Memory API for memory allocation. The following figure shows the MAUI architectural scheme and where the Messaging API fits into it.

Figure 12-1 Messaging API Dependencies



Messaging API Functions

The Messaging API functions are classified into three categories: Initialize and Terminate, Mailboxes, and Messages.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`msg_init()`	Initialize Messaging API
`msg_term()`	Terminate Messaging API
`msg_set_error_action()`	Set error action

The Messaging API depends on the shaded memory API. When the Messaging API is initialized, the shaded memory API is initialized automatically. Similarly, the shaded memory API is terminated upon the Messaging API termination. If the shaded memory API was initialized before, no error situation appears, but after you terminate the Messaging API, the shaded memory API remains initialized.

There are system-wide functions `maui_init()` and `maui_term()` which, respectively, initialize and terminate all the MAUI APIs, including the Messaging API.

To activate the Messaging API separately, call `msg_init()`. This function initializes the API to prepare it for use. No other functions in the Messaging API (excluding `msg_set_error_action()`) may be called before `msg_init()`. Any attempt to call Messaging API functions prior to the initialization returns `EOS_MAUI_NOINIT` error. To terminate the use of the Messaging API, call `msg_term()` or call

maui_term() to terminate all APIs together). After this, you cannot call any other messaging function until you call msg_init() or maui_init() again.

If, for any reason, you initialize the Messaging API more than once, in order to deactivate it, you must terminate it the same number of times.

The Messaging API has a built-in error handling mechanism that sets different actions upon different degrees of error severity. msg_set_error_action() specifies severity levels for error message printing, error code returning, and exiting.

Initializing the Messaging API

To initialize the Messaging API call msg_init(). This function initializes the API and prepares it for use. No other functions in the Messaging API can be called before msg_init() except msg_set_error_action().

For More Information

See **Chapter 1, Printing to stderr** for more information about printing with mem_ functions.

Terminating the Messaging API

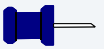
To terminate the Messaging API call msg_term(). After you call msg_term(), you cannot call any other messaging functions unless you first call msg_init().

Watch Service

msg_send_watch()	Request a watch
msg_release_watch()	Release a watch

The Messaging API's watch calls allow a process to request a signal when another process terminates it's use of the Messaging API. This is useful when a process is communicating with another process via the Messaging API and has allocated resources on behalf of that other process. Even if the other process quits without informing anyone, the caller of `msg_send_watch()` can receive a signal and perform any necessary cleanup. To cancel a watch request, use `msg_release_watch()`.

Any application that calls `msg_init()` is eligible for a watch. The watch signal fires with a "watched" process calls `msg_term()`. If a "watched" process exits without calling msg_term(), MAUI detects this and attempts to clean up any open mailboxes and send any outstanding watch signals before the processes terminates.



Note

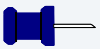
`msg_send_watch()` and `msg_release_watch()` are new as of MAUI 3.1 and require the `mauidev` and `mauidrvr` modules in memory. These calls do not work with statically linked MAUI applications that are older than MAUI 3.1. Older applications that use the MAUI shared library (linked with maui.l) will work because they can dynamically upgrade.

Mailboxes

`msg_create_mbox()`	Create mailbox
`msg_open_mbox()`	Open a mailbox
`msg_close_mbox()`	Close mailbox
`msg_get_mbox_status()`	Get mailbox status

Creating a Mailbox

To create a mailbox call `msg_create_mbox()`. You must supply a mailbox name (a character string, like a module name), the maximum number of messages a mailbox is expected to hold, the size of the largest message, and the color of the memory where a mailbox is to be created. If the function call is successful, it returns a mailbox ID — a handle used to address this mailbox in all the Messaging API functions.



Note

After a mailbox is created, you are linked to that mailbox and do not need to open it.

Opening an Existing Mailbox

To open a mailbox that already exists call `msg_open_mbox()` using the mailbox name and message size. On the successful completion, you receive a mailbox ID.

Closing a Mailbox

To close a mailbox call `msg_close()`. When a mailbox is closed by all the application processes that use it, it is automatically destroyed.

Obtaining Status on a Current Mailbox

To obtain status on a current mailbox call `msg_get_mbox_status()`. This information contains the mailbox name, maximum number of entries, number of free entries, entry size, number of links to this mailbox, mask for filtering messages when writing, and mailbox filter function pointer.

Messages

`msg_peek()`	Search for next message in queue
`msg_peekn()`	Peek at n bytes of the next message in a mailbox
`msg_read()`	Read next message from queue
`msg_readn()`	Read n bytes of the next message from the queue
`msg_unread()`	Remove message from queue and place in mailbox
`msg_unreadn()`	Unread n bytes of the message
`msg_send_sig()`	Send signal on message ready
`msg_release_sig()`	Release signal on message ready
`msg_set_mask()`	Set mask for queueing messages
`msg_write()`	Write message to the queue
`msg_writen()`	Write n bytes of a message to the queue
`msg_flush()`	Flush messages
`msg_dispatch()`	Dispatch message

Retrieving a Message from a Mailbox

To retrieve a message from a mailbox call `msg_read()` or `msg_readn()` using the mailbox ID, the mask, and the blocking mode. This call copies a message into the buffer supplied by your application. After the message is read, it is removed from the mailbox. The mask is used to retrieve messages of a particular type. If you want to read messages of several types, form the mask with an OR statement. `msg_readn()` allows you to specify the maximum number of bytes to read, but otherwise works exactly like `msg_read()`.

Blocking Mode

Blocking mode can be `MSG_BLOCK` or `MSG_NOBLOCK`:

MSG_BLOCK	the program waits until the message of the expected type arrives in the mailbox.
MSG_TYPE_NONE	if there are no messages in the mailbox, the program returns a message of this type. No error codes are set in this case.

Returning a Message Back To a Mailbox

To return the message back to a mailbox call `msg_unread()` or `msg_unreadn()` to mark a message as unread and make it available for the next read operation. `msg_unreadn()` allows you to specify the maximum number of bytes to unread, but otherwise works exactly like `msg_unread()`.

Checking to See if You Have a Message

To check to see if you have a message call `msg_peek()` or `msg_peekn()` to read a message but leave it in a mailbox. Finding the message does not guarantee that it can be read later because there is always a chance that another reader will read it before you. In this case, you can get another message or `MGS_TYPE_NONE` message instead of what you expected. `msg_peekn()` allows you to specify the maximum number of bytes of the message to read, but otherwise works exactly like `msg_peek()`.

Using a Less CPU-intensive Approach to Check Message Arrival

To use a less CPU-intensive approach to check message arrival call `msg_send_signal()`. This function asks for a specified signal to be sent when the message of the specified type is queued to the mailbox. A signal is sent just once per `msg_send_signal()` call. Again, it is possible that another process will “snatch” the message before you can read it.

Cancelling the Signal Request

To cancel the signal request call `msg_release_sig()` to release a pending request for a signal.

Deleting Messages Currently Queued

To delete messages currently queued call `msg_flush()`. Use this if you do not need any messages of a particular type that are currently queued.

Processing Messages Directly after Read

To process messages directly after they are read call `msg_dispatch()`. This call uses the callback function that is found in the common message header.

Message Types

Each message is unique because it carries individual information structures. Yet, all messages share a common header at the beginning, similar to the real-life mail letters and parcels that display their destination addresses in a common, visible place. For MAUI, one of these common header fields is the message type.

The message type distinguishes and filters out the messages that carry the type of information a program expects to receive. For example, the application might process only the keyboard keystrokes and ignore all mouse input.

The message type value is defined by the 32-bit variable. Every bit in this variable represents the separate message type, therefore, only 32 primary message types may be defined in MAUI.

The Messaging API defines the following message types and ranges:

Table 12-1 Message Types

Bit Set	Message Type	Description
None	MSG_TYPE_NONE	This special message type indicates that no messages should be processed. If this type of message is retrieved while reading messages, it means that there are no messages available for reading.
0	MSG_TYPE_PTR	Messages sent by MAUI Input Process for pointer input. (Refer to Chapter 13: Using the Input API).
1	MSG_TYPE_KEY	Messages sent by MAUI Input Process for the key symbol input. (Refer to Chapter 13: Using the Input API).
2	MSG_TYPE_WIN	Messages sent by the MAUI Win Process for window events. (Refer to Chapter 14: Using the Windowing API).
3-23		Reserved for MAUI use.
24-31		Used by the application to define user types of messages.
All	MSG_TYPE_ANY	This type includes all the possible message types.

For multiple message types, you can create an OR statement using any of these types. Other fields of common message headers are:

- Time when the message is queued
- Process ID of the sender

- Pointer to a callback function that is called when the message is dispatched

Example Program

The Message programs illustrate basic principles of the MAUI Messaging API. It consists of two programs:

- `msgwrtr` writes a number of messages to a mailbox, ending with the message of the certain type to indicate the end.
- `msgrdr` reads these messages until the end message is received. For clarity, error checking is omitted.

The executable source for these two programs is located in the directory `MWOS/SRC/MAUI/DEMOS/MSG`.

Chapter 13: Using the Input API

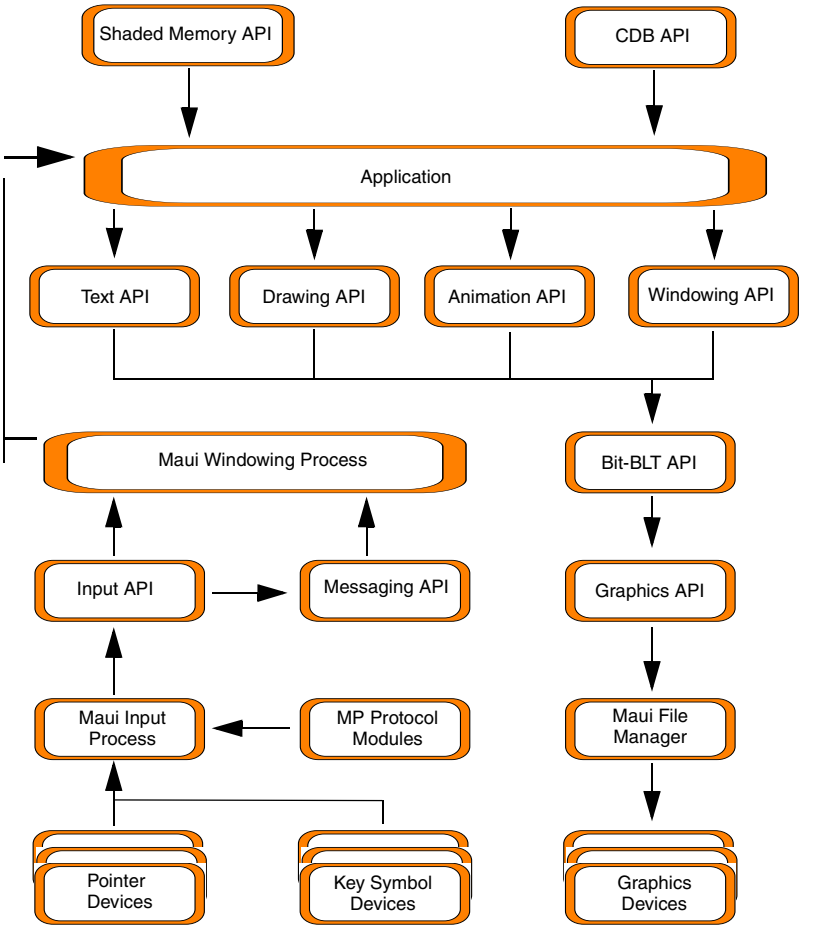
The Input API supports methods for handling input from pointer and key symbol devices. It reads raw input from pointer and key symbol devices and delivers messages to mailboxes. The application may use the Input API to reserve keys on each key symbol device.

This chapter classifies the Input API functions and explains how they are used. An example program at the end of this chapter shows how input can be received from a remote control.

Architecture

The relationships between the Input API, the Shaded Memory API, the Messaging API, and MAUI Input Process are shown in the following figure. The Input API depends on the Shaded Memory API for memory allocation, the Messaging API for sending messages to the mailbox, and MAUI Input Process for message routing

Figure 13-1 Input API Dependencies.



MAUI Input Process and Protocol Modules

The MAUI Input process, `maui_inp` processes all the commands sent by the Input API and translates raw input streams into the sequence of standardized device-independent messages.

In order to make `maui_inp` hardware independent, all translation is performed in separate modules called MAUI Input Process Protocol Modules (MPPMs). These modules are OS-9 subroutine modules that `maui_inp` links to at execution time. Each protocol module is responsible for one input device type.

You must start `maui_inp` before initializing `inp_init()`.

To start `maui_inp`, type:

```
maui_inp&
```

You can also include this command in a start-up script.

Input API Functions

The Input API functions are classified into four groups: Initialize and Terminate, Input Device, Key Symbol Device, and Pointer Device.

For More Information

For detailed function descriptions, refer to the ***MAUI Programming Reference Manual***.

Initialize and Terminate

`inp_init()`	Initialize input API
`inp_term()`	Terminate input API
`inp_set_error_action()`	Set error action

Like all other MAUI APIs, the Input API automatically initializes and terminates APIs it depends on.

Initializing the Input API

To initialize the Input API call `inp_init()`. This function initializes the API and prepares it for use. No other functions in the Input API (except `inp_set_error_action()`) can be called before `inp_init()`. Any attempt to call Input API functions prior to the initialization returns the `EOS_MAUI_NOINIT` error.

Terminating the Input API

To terminate the Input API call `inp_term()`. After you call this function, you cannot call any other input function (excluding `inp_set_error_action()`) until you call `inp_init()` or `maui_init()` again.

There are system-wide functions `maui_init()` and `maui_term()` which, respectively, initialize and terminate all the MAUI APIs, including the Input API.

The Input API has a built-in error handling mechanism that sets different actions depending on degrees of error severity. `inp_set_error_action()` specifies severity levels for printing error messages, error code returning, and exiting.

For More Information

See **Chapter 1**, **Printing to stderr** for more information about printing with `mem_` functions.

Input Device

`inp_open_dev()`	Open input device
`inp_close_dev()`	Close input device
`inp_get_dev_cap()`	Get device capabilities
`inp_get_dev_status()`	Get device status
`inp_check_keys()`	Check key symbol range
`inp_restack_dev()`	Restack an input device
`inp_set_ptr_limit()`	Set pointer limit
`inp_set_ptr_pos()`	Set pointer position
`inp_set_msg_mask()`	Set mask for queueing messages
`inp_set_callback()`	Set callback for queueing messages

The input device is the cornerstone concept of the Input API. The input device could be a mouse, remote control, joystick, touch screen, or other type of input device. From the application program side, input devices are associated with particular mailboxes (see Chapter 12: Using the Messaging API).

Applications can read messages from these mailboxes and interpret them accordingly. Though there are a variety of input devices, message formats are standardized and currently are divided into two groups: pointer messages and key symbol messages. This is accomplished by using the MAUI Input Process and protocol modules.

There are some input device operations which are the same for all input device types.

Opening the Device and Associate it with a Mailbox

To open the device and associate it with a mailbox call `inp_open_dev()`. You must supply the mailbox ID of an open or created mailbox.

The device name format for this call is: `/<dev>/<protocol_module>. <dev>/<protocol_module>` is typically the name supplied by the CDB API (described in Chapter 6: Using the CDB API). In return you get the input device `ID`. This `ID` is the handle used to refer to the device after it is opened.

The message type associated with a pointer device is `MSG_TYPE_PTR`. Along with the common message header (described in Chapter 12: Using the Messaging API), `MSG_TYPE_PTR` contains:

- Input device `ID` (to distinguish between devices if the mailbox is associated with more than one device)
- Subtype of the message (button pressed, button released, pointer moved)
- Number of button that changed, if any
- Current state of all buttons
- Current pointer coordinates
- Key symbol that generated the message (if the pointer is simulated by a key device)

Functions that are specific to pointer devices are:

`inp_set_ptr_limit()`	Defines the maximum and minimum coordinates that the device can return
`inp_set_ptr_pos()`	Assigns new coordinate values to a pointer device

Closing a Device

To close a device call `inp_close_dev()`.

Inquiring About the Device Capabilities

To inquire about the device capabilities call `inp_get_dev_cap()`.

Receiving Current Device Status

To receive current device status call `inp_get_dev_status()`.

Filtering Messages With a Mask

To filter messages with a mark call `inp_set_msg_mask()`.

Specifying the Callback Function for Messages

To specify the callback function for messages call `inp_set_callback()`.

Assigning a Simulation Method

The Input API provides a way to simulate pointer devices using key symbol devices and to simulate key symbol devices using pointer devices. This means that pointer devices can generate `MSG_TYPE_KEY` messages and key symbol devices can generate `MSG_TYPE_PTR` messages.

To assign a simulation method call `inp_set_sim_meth()`.

There are certain Input API functions that only work with a particular input device type.

Key Reservation and Simulation

`inp_release_key()`	Release range of keys
`inp_reserve_key()`	Reserve range of keys
`inp_set_sim_meth()`	Set simulation method

Key symbol devices are used to enter commands and alpha-numeric information. `MSG_TYPE_KEY` is the message type generated by key symbol devices.

Along with the common header and input device `ID`, key symbol messages contain the key symbol and the state of key modifiers (Alt, Ctrl, Meta) at the time the key was pressed or released, if detectable by the device.

The functions specific to key symbol devices are:

`inp_release_key()`	Releases a reserved key
`inp_reserve_key()`	Reserves a key on a particular device, so other applications listening to this device do not receive messages with those key symbols
`inp_set_sim_meth()`	Sets the simulation method to use for the specified device

Example Program

The example input program illustrates input being received from a remote control. The executable source for this program is located in the directory `MWOS/SRC/MAUI/DEMOS/INP`. To help understand how this program works, print the source code listing and follow the comments in the code.

Chapter 14: Using the Windowing API

The MAUI Windowing API manages multiple windows in a windowing device. A windowing device may display graphics and/or text in overlapping or nested windows.

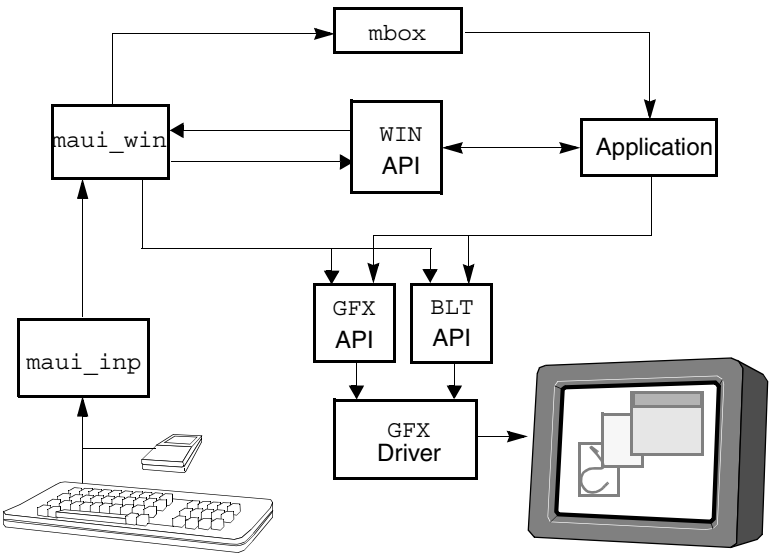
This chapter classifies the Windowing API functions and explains how they are used.

Windowing Concepts

A window is a rectangular area on the windowing device that displays graphics and/or text. Applications can display multiple overlapping and nested windows on one or more windowing devices. The windowing API provides functions that enable applications to move, resize, and re-stack windows. The windowing device can be shared by multiple processes, with each process owning one or more windows. No window is affected by the activity in another window.

The following figure illustrates the components of the windowing system and how they relate to other components in a typical MAUI application.

Figure 14-1 MAUI Windowing System



Applications call windowing functions, which are processed by the Windowing API. The Windowing API returns information to the application such as device and window IDs. Windowing commands are processed by the `maui_win` process.

The `maui_win` process communicates with the Graphics and Bit-BLT APIs to control the graphics device, and sends messages to a mailbox where they are accessed by the application.

Input from pointer or keypad devices are routed to the `maui_win` process. The `maui_win` process routes cursor and ink information directly to the Graphics and Bit-BLT APIs for fast response on the windowing device. Other input information is routed from `maui_win` to the application mailbox.

Windowing Applications

There are two classes of windowing applications. The first type is a window manager type application that consists of an owner process that owns the windowing device and sets up the basic operating parameters for multiple windows. The second type is a single window application that opens a windowing device and uses the window to draw text and graphics.

Window Manager Demonstration

Your MAUI software includes a window manager application that demonstrates how window manager applications are constructed.

The basic operations of a window manager are to create a windowing device, set up the basic appearance and operation of the windowing device, and create windows on behalf of other processes that are then displayed on the windowing device.

Set-up Functions

The first operations performed are the application set-up functions. These operations include:

Step 1. Define the colors used in the background pattern on the windowing device.

Step 2. If a background pattern is required, define a function that applies the pattern to the display screen. This pattern is drawn over the entire screen.

Step 3. Define globals. There are at least two globals necessary: the graphics device ID, and the root window ID.

Main() Functions

The next section of a window manager is `main()`. This portion of the application defines the basic functions of the window manager. `main()` does the following:

Step 1. Initialize the MAUI APIs used. The APIs initialized by this section of code include the Messaging API, the Windowing API, and the Drawing API.

Step 2. Create a message mailbox for the window manager. The mailbox stores messages for the window manager from the `maui_win` process whenever the keyboard or mouse is used.

Step 3. Create and define a windowing device. Creating a windowing device comprises three functions:

```
win_create_dev (&windev, &gfxdev, &root, "/win", "/gfx,
         1,0, mbox);
win_set_callback (root, process_winmsg, NULL);
win_set_msg_mask (root, WIN_MASK_CHILD_CONFIG |
         WIN_MASK_CHILD_CREATE | WIN_MASK_CHILD_DESTROY);
```

`win_create_dev()` creates the windowing device, opens a graphics device and sets the graphics device resolution and coding method. This function creates a root window for the windowing device and returns the ID of the root window.

`win_set_callback()` sets the callback used when queuing messages for the root window. The callback in this example is `process_winmsg()`.

`win_set_mask()` defines the type of messages that are queued to the root window. Three message types, `CHILD_CONFIG`, `CHILD_CREATE`, and `CHILD_DESTROY` are the only message types that need to be queued for the root window.

Root Window

The root window is owned by the creating process (in this case it is our window manager) and has certain privileges such as determining the default cursor, determining the input device, and defining the colormap used by other windows. The colormap is an array of colors used by the window that is assigned to the colormap. The child windows of the window assigned to the colormap may inherit the colormap defined for the root.

```
/* Create the default colormap, add colors, and assign */
/* it to the root window */
win_create_cmap (&cmap, windev, GFX_COLOR_RGB);
color.rgb = 0x000000; win_alloc_cmap_color
   (&pix[COLOR_BLACK], cmap, color);
color.rgb = 0xffffff; win_alloc_cmap_color
   (&pix[COLOR_WHITE], cmap, color);
color.rgb = 0xe7e7e7; win_alloc_cmap_color
   (&pix[COLOR_LGREY], cmap, color);
color.rgb = 0xc0c0c0; win_alloc_cmap_color
   (&pix[COLOR_MGREY], cmap, color);
color.rgb = 0x676767; win_alloc_cmap_color
   (&pix[COLOR_DGREY], cmap, color);
win_set_cmap (root, cmap);

/* Clear the root window background */
window_background (root);

/* Create the default cursor and assign it to the */
/* root window */
win_create_cursor (windev, DEMO_ARROW_CURSOR,
   &demo_arror_cursor);
win_set_cursor (root, DEMO_ARROW_CURSOR);

/* Open the input devices */
win_open_inpdev (&kdev, windev, "/kx0/mp_xtkbd");
win_open_inpdev (&kdev, windev, "/m0/mp_bsptr");
```

Attributes of this process include:

- The default colormap contains five colors and is assigned to the root window.
- The root window is cleared (or a pattern applied) using the function `window_background()`.
- The default cursor (defined in the data type `DEMO_ARROW_CURSOR`) is assigned to the root window.
- Two input devices are opened for the root window: a keyboard and a pointing device.

Message Loop

After the setup is completed, a message loop is defined that continually reads the mailbox and dispatches messages. This simple set of commands reads user input, responds, and reads the next user input.

```
while(TRUE){
   msg_read(mbox, &winmsg, MSG_TYPE_ANY, MSG_BLOCK);
   msg_dispatch(&winmsg);
}
```

Shut-down Procedure

Finally, when the application terminates, a shut-down procedure closes the input device and mailbox, destroys the cursor, colormap, and windowing device, and terminates the MAUI APIs that were initialized.

Colormaps

As stated previously, the child windows may inherit the colormap of the parent window. In some circumstances, you may want to assign a different colormap to a child process. When the child process has a different colormap, the display behaves a bit differently than when all windows share the same colormap. The active window determines the colors for the entire display, so when the cursor moves to a window

assigned to a different colormap, all the windows on the display switch to the colors defined by the active colormap. When all windows share the same colormap, no color switching is seen.

Background Pattern Maintenance

The `window_background()` function is called in `main()` to initialize the background by drawing a pattern on the root window. The most important elements seen in this function are the `win_lock_region()` and `win_unlock_region()` functions. Before any drawing is performed in the window, the region occupied by the window is locked. Since multiple windows may be displayed in a windowing device, locking a region before drawing is critical to ensure that other processes do not interfere with the drawing. For example, if drawing is taking place in a window, and another window is moved on top of the drawing window, the drawing operation can cause both windows to display incorrectly.

```
/* Draw the background */
win_get_win_status (&winstat, win);
win_lock_region (win, 0, 0, winstat.width,
   winstat.height);
for (h=0; (h*32) < winstat.width; h++) {
   for (v = 0; (v*32) < winstat.height; v++) {
      drw_expd_block (drwctx, h*32, v*32, 0, 0, 32, 32);
   }
}
win_unlock_region (win);
```

Always lock a region prior to drawing, and unlock the region when the drawing operation is finished.

Message Process

The message process defined in `process_winmsg()` defines the callback function used by the application. A window manager message process should define a set of actions to taken when specific messages are received. Here is a minimum set of actions a window manager should handle:

- When the user opens a window on the root level, the message process calls a function in the window manager that creates a child window for the calling application.
- When the user closes the child window, the window manager is called to destroy the window.
- When the user wishes to move a window, the window manager is called to process the move request.
- When the user wishes to resize a window, the window manager is called to process the resize request.
- When the user wishes to change the stacking order of the windows, the window manager is called to process the restack request.
- When the user selects a different window to become active (moves the cursor into another window) the window manager is called to change the window state.
- When an area of the root window becomes exposed through one of the above events, the window manager should re-apply the background pattern to the exposed region.

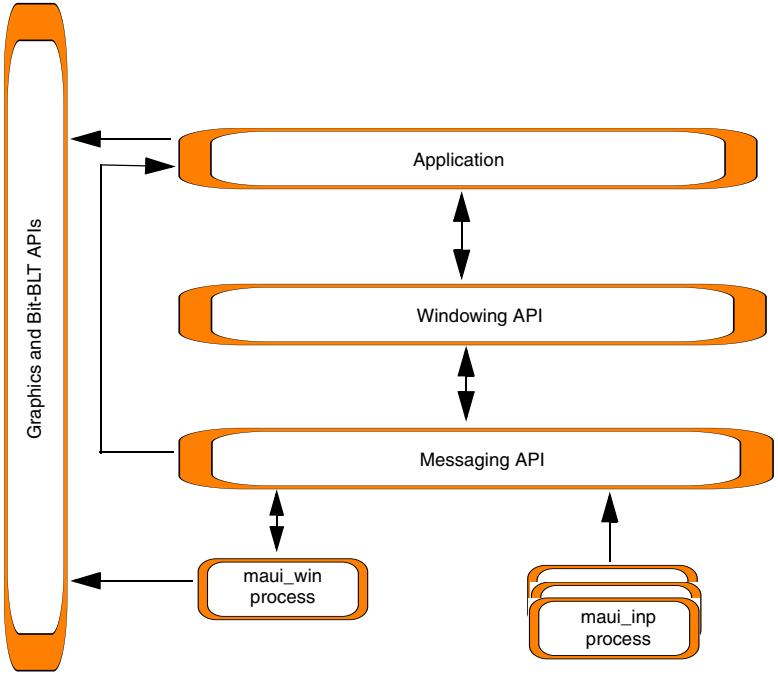
Input Functions

Several input functions are handled automatically by the `maui_win` process and require no intervention on the part of the application. Cursor movement is processed by `maui_win` and directly communicated through the graphics and Bit-BLT APIs to the graphics device. Keyboard inputs are also echoed to the graphics device automatically. The application needs to take no action to perform these functions.

Architecture

The relationship between the Windowing API, Messaging API, Bit-BLT API, and the Graphics Device API is shown in the following figure. The Windowing API depends on the Shaded Memory API for memory allocation, the Bit-BLT API for block transfer operations, and the Graphics Device API for device capabilities. The Messaging API manages communication between the Windowing API and the `maui_win` and `maui_inp` processes

Figure 14-2 Windowing API Dependencies.



Before the Windowing API is used in an application, the Bit-BLT, Graphics, and Messaging APIs must be initialized. After the Windowing API is initialized (`win_init()`) set the action to take in the error handler by calling `win_set_error_action()`. The Windowing API is terminated when your application calls `win_term()`.

The Windowing Device

The windowing device is a group of devices including one or more graphics devices and one or more input devices. When a windowing device is created, several activities take place:

- A graphics device is assigned to the windowing device
- A root window is assigned to the windowing device
- Callback information is set up
- One or more input devices are opened and assigned to the windowing device
- A colormap is defined and assigned to the root window
- A default cursor is created and assigned to the root window

Creating a Windowing Device

The function `win_create_dev()` creates the windowing device. The process that creates the windowing device owns the device. This function contains parameters that name of the windowing device, open the graphics device, create the root window, and identify the mailbox.

After the widowing device is created, set the callback information for the root window. This is accomplished with the function `win_set_callback()`. The callback function defines the action to take when certain messages are received by the application.

One or more input devices can be assigned to the windowing device. Input devices are assigned to a windowing device by using the function `win_open_inpdev()`. This function returns the device ID of the input device, assigns the device to a window, and provides the name of the input device. When an input device is no longer needed, use the function `win_close_inpdev()` to close the device.

When the input device is a keyboard, the function `win_set_focus()` defines which window has the keyboard focus.

The windowing device is opened by using the function `win_open_dev()`. The windowing device may be opened by any process, but the opening process does not own the device.

Before or after opening a device, the process may call `win_get_dev_status()` for specific information about the device. This function returns information including the graphics device ID, root window ID, width and height of root window, device coding method, and which window has the keyboard focus.

The process that opened the windowing device can close the device by calling the function `win_close_dev()`. This function closes the graphics device and input devices, and destroys all windows, colormaps and cursors opened by the process.

The process that owns the windowing device can destroy the device. Windowing devices are destroyed by calling the process `win_destroy_dev()`. This function destroys the graphics device and input devices, and destroys all windows, colormaps and cursors that were created by the process.

Always use `win_close_dev()` to close a device that was opened. Always use `win_destroy_dev()` to close a device that was created.

Colormaps

A colormap is an array of colors used and owned by the process that created the colormap. Colormaps are created with the function `win_create_cmap()`. The maximum number of cells in a colormap is determined by the graphics color type assigned to the colormap when it is created. For example, if the colormap color type is `GFX_CM_8BIT`, the colormap contains 256 cells. You can allocate a color to each of the cells in a colormap up to the maximum number of cells.

Colormaps are assigned to a window or windowing device and inherited by all the child processes of that window or windowing device. Colormaps are managed with a set of six functions:

`win_alloc_cmap_color()`	Assigns a color value to a colormap cell and returns the color.
`win_get_cmap_cells()`	Returns the colors assigned to the specified group of cells.
`win_get_cmap_free()`	Returns information about the amount of free cells in the colormap. The information includes the total number of free cells, the largest contiguous block of free cells, and the number of blocks of one or more free cells.
`win_alloc_cmap_cells()`	Allocates a group of color cells for the private use of the calling process.
`win_set_cmap_cells()`	Assigns colors to a specified group of cells allocated by a process. Only the process that allocated the group of colormap cells can set those cells.
`win_free_cmap_cells()`	De-allocates some or all of the cells previously allocated with `win_alloc_cmap_cells()` or `win_alloc_cmap_color()`.

When a colormap is no longer needed, the owning process can call `win_destroy_cmap()` to destroy the colormap.

Cursors

The root window is assigned a default cursor when it is created. Child windows inherit the default cursor, or you may assign a different cursor to each window as they are opened. The cursor is owned by the process that created it. The two cursor functions available are:

`win_create_cursor()`	Creates a cursor and assigns the cursor to the window specified.
`win_destroy_cursor()`	Destroys the specified cursor.

Cursors are defined in a data structure called `WIN_CURSOR`, that contains a pointer to the cursor bitmap and specifies the cursor hit point. The process that owns the cursor is the only process that can destroy the cursor.

When the input device commands the cursor to move, the movement commands are routed from the input device and `maui_inp` process to the `maui_win` process. From `maui_win`, these commands are routed directly to the Graphics and Bit-BLT APIs for output to the display screen. By keeping the Windowing API uninvolved with cursor movement, the display can be faster and more responsive to user input.

Managing Windows

Windows are managed by a group of functions. Some of the functions control the how a window operates, and others control the appearance of the window.

Create and Destroy

Windows can be created on the root context by any process, or can be created as a child of a window created by the same process. When a window is created as a child of another window, the new window must be wholly contained within the parent window. Two functions are available to create and destroy windows:

`win_create_win()`	Creates a new window as a child of the window specified.
`win_destroy_win()`	Destroys a window and all its child window.

Window Setup

When a window is created, it can inherit the characteristics of the parent window, or you can set the operating parameters for each window individually. Eight functions are available to define how a window is set up:

`win_get_win_status()`	Returns information about the parent, child, and sibling windows, position of the window, size, inking method, callback and drawing, text, and colormap IDs.
`win_set_cursor()`	Defines a cursor to use with the window. The cursor changes as it enters the boundary of the window.
`win_set_callback()`	Sets the callback for queuing messages.

`win_set_msg_mask()`	Determines which messages will be queued.
`win_set_state()`	Sets the state of the window to active or not active.
`win_set_cmap()`	Assigns a colormap to the window.
`win_reparent_win()`	Assigns the window to a new parent window. Both the old and new parent must be owned by the calling process.
`win_restack_win()`	Changes the position of the window in the stack of windows.

Window Appearance

The size and position of a window can change in response to application needs, or user input.

`win_move_win()`	Moves the window to a new location.
`win_resize_win()`	Changes the height and width of the window.

Ink

The Windowing API includes inking functions that enable users to draw in a window using a cursor, pen, or other input device. A single ink point on a display is 3 pixels wide by 3 pixels high. The center pixel of the 9-pixel grid is the hit point.

`win_set_ink_method()`	Sets the inking method to `OFF` or `REPLACE`.
`win_set_ink_pix()`	Sets the pixel value for the ink. This value may be a color value, or an index value in a color table.

`win_erase_ink()`	Erases all the ink in a window. If necessary, this function sends a message to repaint the entire window to restore the background.

When the input device is used to draw in the window, the ink commands are routed to the `maui_win` process, then directly to the Graphics and Bit-BLT APIs for drawing on the display. The windowing API is not directly involved in the actual ink drawing process, resulting in a faster and more responsive display.

Drawing

Drawing text and graphics in a window is very similar to drawing text and graphics on a graphics device. Since a single display may contain several windows, text and graphics are assigned to a specific window rather than a more general graphics device. Two functions are provided to assign contexts to windows prior to drawing and text operations:

`win_set_drw_context()`	Specifies the drawing context for a specific window.
`win_set_txt_context()`	Specifies the text context for a specific window.

The actual drawing is performed by the Drawing API. The Windowing API maintains the origin, drawing area, and clipping are in the drawing context, so you must not call `drw_set_context_origin()`, `drw_set_context_draw()`, or `drw_set_context_clip()`. Likewise, when drawing text you must not call `txt_set_context_origin()`, `txt_set_context_draw()`, or `txt_set_context_clip()`.

Lock and Unlock

Because several windows may be displayed at the same time, several processes may be running at any one time. There is always a risk when drawing text or graphics, that a drawing or text operation may interfere with another process. Two functions are provided in the Windowing API to ensure drawing and text operations are completed safely:

`win_lock_region()`	Locks a region of a window prior to text and drawing operations.
`win_unlock_region()`	Unlocks the region previously locked.

Locking a region for an extended length of time can interfere with other processes. `win_lock_region()` should be called immediately before the drawing functions are called, and `win_unlock_region()` should be called immediately after the drawing operation is finished.

If the window contains an active cursor, it is turned off when the window is locked. Unlocking the window enables the cursor.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

S

T

U

V

W