Message Manager

The QuickDraw GX Message Manager is a part of the message-passing printing architecture of QuickDraw GX. Read this chapter if you want to use the Message Manager to develop printing extensions or printer drivers.

Because QuickDraw GX uses the Message Manager for printing, you should be familiar with the chapter "Introduction to QuickDraw GX Printing" in *Inside Macintosh: QuickDraw GX Printing* before reading this chapter.

If you want to use the Message Manager to create printing extensions and printer drivers, you should also read *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*

This chapter introduces the Message Manager as it is used for printing with QuickDraw GX. It then shows how to use Message Manager functions to

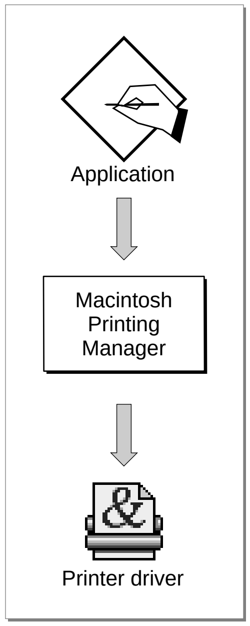
- allocate memory for and dispose of global data
- store global data for a single message handler instance
- store global data for multiple message handler instances
- send and forward messages

This chapter also contains reference information for constants, data types, and functions associated with the Message Manager.

About the Message Manager

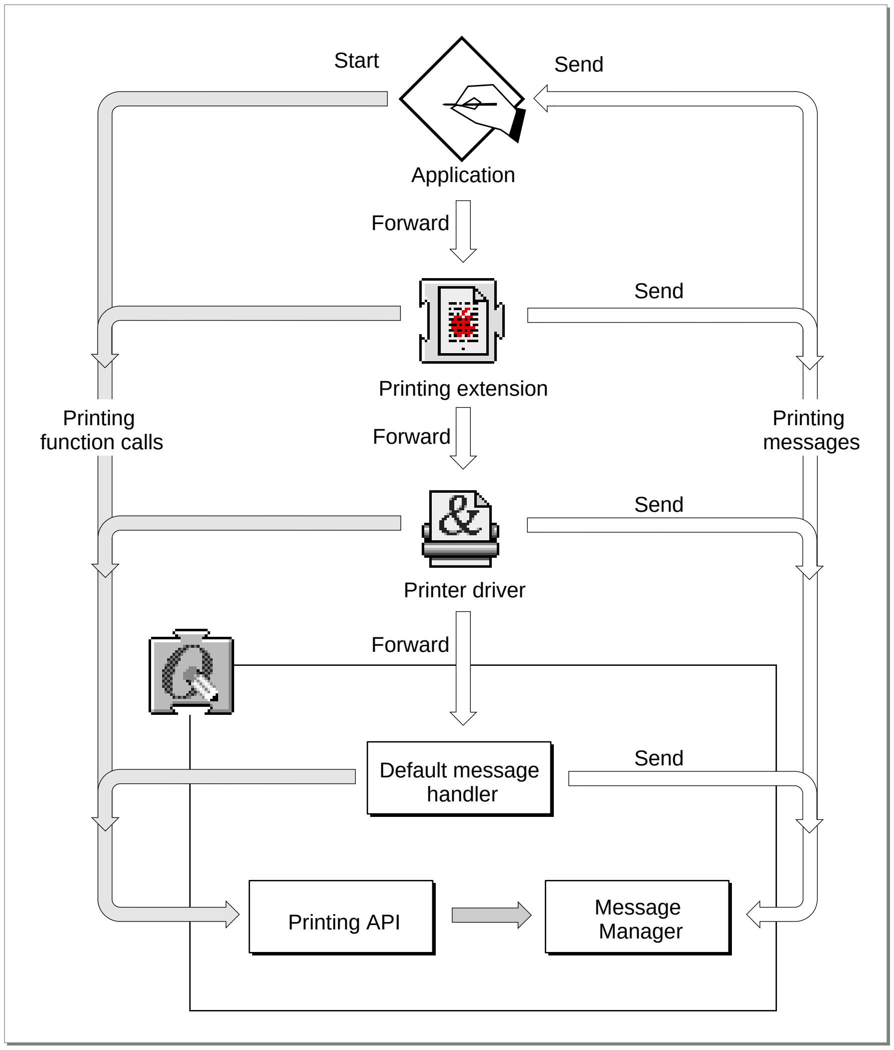
On Macintosh systems in which QuickDraw GX is not installed, the Macintosh Printing Manager calls the printer driver by loading appropriate code resource for the printer driver, as shown in Figure 6-1.

Figure 6-1 Printing with the Macintosh Printing Manager



In contrast, QuickDraw GX provides a low-level software manager called the **Message Manager** to transfer control to the printer driver. Whenever an application makes a printing call, QuickDraw GX interacts with the printer driver by calling the Message Manager to request that the appropriate message be sent to the printer driver. QuickDraw GX printing extensions may be inserted between QuickDraw GX and the printer driver to modify the behavior of printing without changing the printer driver. This approach greatly increases the flexibility of printing and allows printing enhancements to be developed quickly and easily. Figure 6-2 shows the relationship of the QuickDraw GX printing software components.

Figure 6-2 Printing with QuickDraw GX



QuickDraw GX predefines over a hundred messages. An application starts the printing process by calling the QuickDraw GX printing application programming interface (API). QuickDraw GX may perform the task itself or call the Message Manager to send one or more messages to the application to initiate one or more steps in the following sequential message chain: application, printing extensions, printer driver, and the default message handler.

The key to the QuickDraw GX extensible printing architecture is the sequential relationship of the application, printing extensions, printer driver, and default message handler for printing. Applications, printing extensions, and printer drivers are located in the message stream so that they may override messages before the message gets to the default message handler. This is the end of the line for any message that makes it to the end of the chain. QuickDraw GX defines the normal printing characteristics that occur unless modified by an application, printing extensions, or the printer driver. Printing modification may occur when one or more messages are overridden. QuickDraw GX sends a large number of printing messages during the printing process. Since many messages are not normally overridden, QuickDraw GX provides a default printing behavior for most messages via the default message handler.

A partial message override occurs when the application, printing extensions, or printer driver perform one or more tasks in response to a message and then forward the message to the next step in the message chain. A complete message override occurs when the application, printing extensions, or printer driver perform one or more tasks in response to a message and do not forward the message to the the next message handler in the chain. Any message that is not explicitly overridden by a printing extension or printer driver is implicitly forwarded to the next link in the sequential message chain. A complete override of a message prevents the next extension, printer driver, or default implementation in the chain from receiving the overridden message.

The Message Manager is not the only initiator of messages. Applications, printing extensions, printer drivers, and QuickDraw GX not only make printing function calls, but they can also initiate messages.

For additional information about printing with QuickDraw GX, see *Inside Macintosh: QuickDraw GX Printing*. For additional information about how to use the QuickDraw GX Message Manager and messages, see *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

Message Terminology

In working with the Message Manager there are a number of terms that are useful to describe the software components and their interactions.

A **message objec**t is the loose equivalent of an object in a fully object-oriented system. It is the recipient of messages. A message object may also send messages to itself or to another message object.

A **message** is a form of notification passed to a message object in order to have that message object perform some operation.

A **message handler** is a component of a message class. A message class may consist of one or more handlers, each of which overrides zero or more messages. Each message handler may override some portion of the functionality of the handler below it in the message class. Message classes are built up from message handlers, in a manner similar to that in which a class in an object-oriented language is derived from other classes. To **forward** is to invoke the override of the next handler in the chain for the current message.

A **message override** is the loose equivalent of a method. It is the implementation, in actual code, of a given message. The override performs the operation requested by sending a message to a message object.

A **message class** is the loose equivalent of a class in a fully object-oriented system. It defines the set of messages that message objects instantiated from it understand and encapsulates the message handlers that implement the overrides corresponding to those messages. Message classes define the acceptable set of messages for all handlers that they encapsulate.

An **instance** is one copy of a message handler in memory.

Global Data Storage for Printing Extensions and Printer Drivers

Printing extensions and printer drivers are stand-alone code and do not enjoy the full status of an application. When an application is launched, a memory block is automatically allocated for the storage of globals. Unlike applications, stand-alone code is never launched. It is simply loaded, and therefore no memory for globals is allocated.

As a result, if your printing extension or driver requires global data, it must allocate and deallocate memory for this data. Global data can be stored as a constant, a handle, a pointer, or in a so-called A5 world by the use of QuickDraw GX Message Manager functions. QuickDraw GX will not dispose of your globals for you. You must explicitly dispose of them yourself when you are done using them.

Each instance of a message handler can only see its data. If you want to limit access to one instance of your message handler's data, see the section "Setting and Getting Global Data for a Single Handler Instance" beginning on page 6-10.

If you want to use common global data that is accessible to all instances of your handlers, see the section "Setting and Getting Global Data for Multiple Handler Instances" beginning on page 6-12.

To create an A5 world that limits the access of your global data to one copy of your message handler, see the section "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

For more information about the A5 world, see *Inside Macintosh: Memory*.

Message Sending and Forwarding

QuickDraw GX provides functions that allow you to send a specific message to the top of the message chain (your application), forward a specific message to the next message handler, or forward the current message to the next message handler. For additional information about message sending and forwarding, see the section "Sending and Forwarding Messages" beginning on page 6-15.

Using the Message Manager

This section describes how to

- determine the version of the Message Manager
- allocate and deallocate memory for globals
- create and retrieve global data for a single instance of the message handler
- create and retrieve global data for multiple instances of a message handler
- send and forward messages

Determining the Version of the Message Manager

To determine the current version of the QuickDraw GX Message Manager, you can call the `Gestalt` function with the `gestaltMessageMgrVersion` selector `'mess'`. The `gestaltMessageMgrVersion` selector returns a 2-byte value indicating the version of the QuickDraw GX Message Manager that is currently installed. The high-order byte is the major version number and the low-order byte is the minor revision number.

The selector '`mess`' is defined in the section “Message Manager Gestalt Selector” beginning on page 6-16.

For more information about the `Gestalt` function, see the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities.*

Allocating Memory for and Disposing of Global Data

You can use the `NewMessageGlobals` function to request and allocate memory for globals. You should only call this function while your application is performing a message override.

You should always initialize your global data from a function other than the one in which you call the `NewMessageGlobals` function. Otherwise, your development environment may generate code with bad data references.

Listing 6-1 gives an example of using the `NewMessageGlobals` function to create an A5 world from an MPW programming environment.

Listing 6-1 Creating an A5 world for global data

```
gxShape  gMyShape;
Handle   gMyHandle;

OSErr MyInitGlobalData()
{
   OSErr err;
   gMyShape = nil;
   gMyHandle = TempNewHandle(1024, &err);
   return err;
}

OSErr MyInitialize()
{
   OSErr err;

/*
   Create an A5 world, and initialize the
   global data.
*/
   err = NewMessageGlobals(A5Size(), A5Init);

   if (!err) err = MyInitGlobalData();

   return err;
}
```

The `MyInitalize` function is the override for the `GXInitialize` message. The `MyInitialize` function first sets up an A5 world, as required if an extension is going to use global data. In this case the global data is the `MyShape` structure. Once you create the A5 world by calling the `NewMessageGlobals` function, your global data will be valid whenever your printing extension or printer driver is called. Once the `NewMessageGlobals` function has been called, the extension or driver can initialize its global data. In this example, the code uses a function called `MyInitGlobalData` to do this.

If you have allocated memory for your globals using the `NewMessageGlobals` function, you must use the `DisposeMessageGlobals` function to dispose of the globals and deallocate their memory blocks when they are no longer needed.

Note that `DisposeMessageGlobals` does not dispose of data and handles. These must be disposed of by your code. First, you deallocate any memory that you have allocated and then let QuickDraw GX deallocate memory that it has allocated for your global data.

Listing 6-2 shows how to dispose of global data and deallocate the memory that was allocated in Listing 6-1.

Listing 6-2 Disposing of global data and deallocating memory

```
OSErr MyShutDown()
{/* Dispose of our global data */
   if (gMyHandle != nil)
      DisposHandle(GMyHandle);
/* dispose of the A5 world that was created in MyInitialize */
   DisposeMessageGlobals();
   return noErr;
}
```

The `NewMessageGlobals` function is described on page 6-17. The `DisposeMessageGlobals` function is described on page 6-18.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Setting and Getting Global Data for a Single Handler Instance

You can use the `SetMessageHandlerInstanceContext` function to store data that can be used by a single instance of a message handler. A new instance of your message handler is created each time a new printing job is created. For example, if four printing jobs are created, four instances of the message handler are created. Each job has a unique context called the instance context.

Listing 6-3 uses this function to store global data whenever a new printing job is initiated. If there are multiple print jobs, this function will be called when each job is started.

Listing 6-3 Storing global data for a single message handler instance

```
typedef struct MyDataRec {
   long something;
   long somethingElse;
} MyDataRec, **MyDataHdl;

OSErr MyInitialize()
{
   OSErr       err;
   MyDataHdl   dataHandle;

/*
   Create a new temporary memory handle, initialize
   it, and store it as the message handler's instance
   context.
*/
   dataHandle = (MyDataHdl) TempNewHandle(sizeof(MyDataRec),
                                                    &err);

   if (err == noErr)
   {
      MyInitDataHandle(dataHandle);
      SetMessageHandlerInstanceContext(dataHandle);
   }

   return err;
}
```

In Listing 6-3, you begin by creating a handle to store global data for the `MyDataRec` structure. Each message handler instance has a unique copy with unique values for the fields of the data structure. If there is insufficient memory to create the handle, an error will be generated. If the handler is successfully created, the handler is initialized. The `SetMessageHandlerInstanceContext` function is then used to store a reference to the handle that can then be used by this message handler's overrides. If you use this code in an extension and four jobs were created for it, each job would have a handle to a unique copy of a record for the structure.

You can use the `GetMessageHandlerInstanceContext` function to retrieve the data that you stored with the `SetMessageHandlerInstanceContext` function. Listing 6-4 uses the `GetMessageHandlerInstanceContext` function to return and dispose of the handle containing the global data that was previously stored in Listing 6-3.

Listing 6-4 Getting and disposing of global data

```
OSErr MyShutDown()
{
   MyDataHdl dataHandle;

/*
   Retrieve the message handler's instance context.  If the
   value returned isn't nil, it's a handle that we stored
   earlier. Dispose of the handle and set the instance
   context to nil to "clear" it.
*/
   dataHandle = (MyDataHdl) GetMessageHandlerInstanceContext();

   if (dataHandle != nil)
   {
      DisposHandle((Handle) dataHandle);
      SetMessageHandlerInstanceContext(nil);
   }

   return noErr;
}
```

In Listing 6-4, the `GetMessageHandlerInstanceContext` function is used to get the previously stored handle containing the global data. If the handle isn't `nil`, it's the `handle` that was previously stored and it is disposed of. Finally, the `SetMessageHandlerInstanceContext` function is used to set the context data to `nil`. If the instance context is `nil`, the handle was previously disposed of.

The `SetMessageHandlerInstanceContext` function is described on page 6-19. The `GetMessageHandlerInstanceContext` function is described on page 6-20.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Setting and Getting Global Data for Multiple Handler Instances

You can use the `SetMessageHandlerClassContext` function to store data that can be used by multiple copies of your message handler in memory. This common data can be accessed by multiple print jobs and eliminates the need for storing redundant data. Listing 6-5 shows how to use the `SetMessageHandlerClassContext` function to store global data that can be used by multiple handler instances.

Listing 6-5 Storing global data for multiple handler instances

```
typedef struct MySharedDataRec {
   unsigned long  ownerCount;
   long           someData;
   long           someMoreData;
} MySharedDataRec, **MySharedDataHdl;

OSErr MyInitialize()
{
   OSErr             err = noErr;
   MySharedDataHdl   sharedDataHdl;

/*
   Retrieve the message handler's class context.  If the
   value returned is nil, the class context isn’t set up.  In
   that case, create a new handle, initialize
   it, set its owner count to 1, and store it in our class
   context.

   If the class context has been set up, retrieve the data
   handle and increment its owner count.  (We will use the
   owner count in our gxShutDown message override.)
*/
   sharedDataHdl = (MySharedDataHdl)
               GetMessageHandlerClassContext();

   if (sharedDataHdl == nil)
   {
      sharedDataHdl = (MySharedDataHdl)
                  TempNewHandle(sizeof(MySharedDataRec), &err);
      if (!err)
      {
         MyInitSharedDataHandle(sharedDataHdl);
         (*sharedDataHdl)->ownerCount = 1;
         SetMessageHandlerClassContext(sharedDataHdl);
      }
   }
   else
      ++(*sharedDataHdl)->ownerCount;

   return err;
}
```

In contrast to the instance context that is always `nil` as you enter into an initialize routine, with the class context you can't assume that the context is `nil`. For example, you may be the third instance of this message handler. As a result, you need to test to see if the class context is already set up. If it is, you increment the owner count. If it isn't you se tup the context. This ensures that the class context is only set up once Listing 6-5 shows how to use the owner count to set up the class context. If the class context is not `nil`, then you increment the owner count. Otherwise, create the handle, set the owner count to 1, and store the class context.

You can use the `GetMessageHandlerClassContext` function to retrieve data that has been stored by the `SetMessageHandlerClassContext` function. Listing 6-6 shows how to retrieve a message handler's class context and use the information during shutdown.

Listing 6-6 Retrieving a message handler's class context

```
OSErr MyShutDown()
{
   MySharedDataHdl   sharedDataHdl;

/*
   Retrieve the message handler's class context.  If the
   value returned is nil, the class context isn't set up.
   Otherwise, decrement our data's owner count.
   If the owner count falls below 1, dispose of the
   actual data and set our class context to nil to
   "clear" it.
*/
   sharedDataHdl = (MySharedDataHdl)
               GetMessageHandlerClassContext();

   if (sharedDataHdl != nil)
   {
      if (--(*sharedDataHdl)->ownerCount < 1)
      {
         DisposHandle((Handle) sharedDataHdl);
         SetMessageHandlerClassContext(nil);
      }
   }

   return noErr;
}
```

In Listing 6-6, you use the `GetMessageHandlerClassContext` function to obtain and use the class context during shutdown. If the class context is not `nil`, you decrement the owner count. If the owner count is less than 1, there are no other owners and you may then dispose of the data. Using the owner count during shutdown prevents disposing of data more than once.

The `SetMessageHandlerClassContext` function is described on page 6-21. The `GetMessageHandlerClassContext` function is described on page 6-22.

Global data is discussed in the section “Global Data Storage for Printing Extensions and Printer Drivers” beginning on page 6-7.

Sending and Forwarding Messages

Message objects can send a printing message to other clients in the **message chain.** When a message is sent, QuickDraw GX receives it and sends it to the first message handler in the chain. In Figure 6-2 this is the application.

QuickDraw GX provides two methods of sending messages. You can use a statement with the format:

```
anErr = Send_GXMessageName(arguments);
```

A typical example is

```
anErr = Send_GXCompleteSpoolFile(theSpoolFile);
```

Alternatively, you can use the `SendMessage` function to send a specified message to the top of the message chain.

You can use the `ForwardMessage` function to specify the message to be forwarded to the next message handler. This function takes a selector that indicates the message to be forwarded and has parameters that are message-specific.

For example, a four-up printing extension that maps four document pages onto one physical page at print time may require that the `GXCountPages` message be forwarded. The `GXCountPages` message has the following interface:

```
OSErr GXCountPages (gxSpoolFile thePrintFile, long* numPages);
```

You can use the `ForwardThisMessage` function to forward the current message to the next message handler.

```
anErr = ForwardThisMessage(gxCountPages, thePrintFile, &numPages);
```

All the QuickDraw GX `Forward_xxx` functions, where `xxx` is the QuickDraw GX printing message to forward, are in-line aliases to the `ForwardThisMessage` function with the message-specific parameters added for type-checking purposes. An example of the recommended format for forwarding a message is:

```
anErr = Forward_GXCountPages(thePrintFile, &numPages);
```

The `SendMessage` function is described on page 6-23. The `ForwardMessage` function is described on page 6-24. The `ForwardThisMessage` function is described on page 6-25.

Printing messages are described in the “Printing Messages” chapter of *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

Message Manager Reference

This section provides reference information for constants, data types, and functions that allow you to work with the QuickDraw GX Message Manager.

Constants and Data Types

This section describes the constants and data types used by the Message Manager.

Message Manager Gestalt Selector

The Gestalt selector `'mess'` can be used to determine which version, if any, of the Message Manager is installed.

```
enum {
   gestaltMessageMgrVersion = 'mess'
};
```

Message Globals Initiatialization Procedure

You may supply your own initialization procedure for your globals using this type definition.

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

To install a message globals initialization procedure, use the `NewMessageGlobals` function described on page 6-17.

For more about initializing your globals, see the section “Allocating Memory for and Disposing of Global Data” beginning on page 6-8.

Functions

This section describes the Message Manager functions you can use to

- allocate memory for and dispose of global data
- define and retrieve global data for a single handler instance
- define and retrieve global data for multiple handler instances
- send and forward messages

Allocating Memory for and Disposing of Global Data

This section describes the functions the QuickDraw GX Message Manager provides for allocating and deallocating memory for your global data.

NewMessageGlobals

You can use the `NewMessageGlobals` function to request and allocate memory for globals.

```
OSErr NewMessageGlobals (long msgGlobalsSize,
                        MessageGlobalsInitProc aProc);
```

`msgGlobalsSize`	The size of the memory requested for global data.
`aProc`	A pointer to an application-defined callback function that initializes and allocates global data memory.
function result	An error of type `OSErr` indicating that the requested memory allocation could not be completed.

DESCRIPTION

The `NewMessageGlobals` function sets up a global world for your printing extension or printer driver. This consists of allocating the specified amount of memory and initializing it with the passed procedure. Once you have created a global world, you can access your data just as you would if your printing extension or printer driver were an application. Whenever your extension or driver is called, your data will be valid.

To establish an A5 world for your globals, the `msgGlobalsSize` parameter is the `A5Size` function and the `aProc` parameter is the `A5Init` function. The `A5Size` and `A5Init` functions are both Macintosh Programming Workshop (MPW) library routines. The `A5Size` function determines how much memory is to be allocated for the A5 world. The `A5Init` function takes a pointer to the A5 globals and initializes them to the appropriate values.

When your extension or printing driver no longer needs the globals, you should release the memory allocated by the `NewMessageGlobals` function by calling the `DisposeMessageGlobals` function.

SEE ALSO

Global data and the A5 world are discussed in the sections "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7 and "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

To dispose of printing extension and printer driver globals, use the `DisposeMessageGlobals` function described in the next section.

The prototype for the application-defined callback function for global data initialization is described on page 6-26.

DisposeMessageGlobals

You can use the `DisposeMessageGlobals` function to dispose of globals and deallocate their memory blocks.

```
OSErr DisposeMessageGlobals (void);
```

function result An error of type `OSErr` indicating that the globals are not disposed of.

DESCRIPTION

The `DisposeMessageGlobals` function disposes of all globals and deallocates the memory used by your printing extension or printer driver for globals. You should use this function to free memory whenever your printing extension or printer driver no longer requires globals.

SEE ALSO

Global data and the A5 world are discussed in the sections "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7 and "Allocating Memory for and Disposing of Global Data" beginning on page 6-8.

To allocate memory for globals, use the `NewMessageGlobals` function described in the previous section.

Setting and Getting Global Data for a Single Handler Instance

This section describes the functions the QuickDraw GX Message Manager provides for defining and retrieving global data for a single handler instance.

SetMessageHandlerInstanceContext

You can use the `SetMessageHandlerInstanceContext` function to store data that can be used by a single handler.

```
void *SetMessageHandlerInstanceContext (void *);
```

DESCRIPTION

The `SetMessageHandlerInstanceContext` function is used to store data that can be used by only a single instance of a message handler. This data is specific to your handler's code and is unique to one copy in memory. The stored data can be in the form of a long word constant, handle, or pointer to other data. The passed data can be accessed only by the instance of the message handler that sets the data.

SEE ALSO

To retrieve the data that has been set by the `SetMessageHandlerInstanceContext` function, use the `GetMessageHandlerInstanceContext` function described in the next section.

To define common data that can be used by multiple instances of a handler, use the `SetMessageHandlerClassContext` function described on page 6-21. To retrieve the common data that has been set, use the `GetMessageHandlerClassContext` function described on page 6-22.

The use of the `SetMessageHandlerInstanceContext` function is described in the section "Setting and Getting Global Data for a Single Handler Instance" beginning on page 6-10.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

GetMessageHandlerInstanceContext

You can use the `GetMessageHandlerInstanceContext` function to retrieve data for a single instance of a handler.

```
void *GetMessageHandlerInstanceContext (void);
```

DESCRIPTION

The `GetMessageHandlerInstanceContext` function returns the data that you stored using the `SetMessageHandlerInstanceContext` function. This function returns the data that was stored by the instance of a handler that is calling the `GetMessageHandlerInstanceContext` function.

If the `SetMessageHandlerInstanceContext` function has not been previously called, the `GetMessageHandlerInstanceContext` function will return `nil`. If a constant, handle, or pointer to other data has been stored, the `GetMessageHandlerInstanceContext` function returns the stored data.

SEE ALSO

The `SetMessageHandlerInstanceContext` function is described in the previous section.

To define common data that can be used by multiple handlers, use the `SetMessageHandlerClassContext` function described on page 6-21. To retrieve the common data that has been set, use the `GetMessageHandlerClassContext` function described on page 6-22.

The use of the `GetMessageHandlerInstanceContext` function is described in the section “Setting and Getting Global Data for a Single Handler Instance” beginning on page 6-10.

Global data is discussed in the section “Global Data Storage for Printing Extensions and Printer Drivers” beginning on page 6-7.

Setting and Getting Global Data for Multiple Handler Instances

This section describes the functions the QuickDraw GX Message Manager provides for defining and retrieving global data for multiple handler instances.

SetMessageHandlerClassContext

You can use the `SetMessageHandlerClassContext` function to store data that can be used by multiple instances of a message handler.

```
void *SetMessageHandlerClassContext (void *);
```

DESCRIPTION

The `SetMessageHandlerClassContext` function is used to store data that can be used by multiple instances of a message handler in one or more print jobs. The parameter passed is a pointer to the `long` data. The stored data can be in the form of a constant, handle, or pointer to additional data. This reference constant can be used by all instances of a message handler.

SEE ALSO

To retrieve the data defined by the `SetMessageHandlerClassContext` function, use the `GetMessageHandlerClassContext` function described in the next section.

The use of the `SetMessageHandlerClassContext` function is described in the section "Setting and Getting Global Data for Multiple Handler Instances" beginning on page 6-12.

To define data that can be used by only one handler, use the `SetMessageHandlerInstanceContext` function described on page 6-19. To retrieve the data that has been set, use the `GetMessageHandlerInstanceContext` function described on page 6-20.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

GetMessageHandlerClassContext

You can use the `GetMessageHandlerClassContext` function to allow multiple instances of your handler to retrieve common global data.

```
void *GetMessageHandlerClassContext (void);
```

DESCRIPTION

The `GetMessageHandlerClassContext` function returns common data that you defined using the `SetMessageHandlerClassContext` function. This function can be called by any instance of your handler.

If the `SetMessageHandlerClassContext` function has not been previously called, the `GetMessageHandlerClassContext` function will return `nil`. If a constant, handle, or pointer has been stored, the `GetMessageHandlerClassContext` function returns the stored data. This function may be used by your handler to allow multiple print jobs to share common global data.

SEE ALSO

To store the data that is retrieved by the `GetMessageHandlerClassContext` function, use the `SetMessageHandlerClassContext` function described in the previous section.

The use of the `GetMessageHandlerClassContext` function is described in the section "Setting and Getting Global Data for Multiple Handler Instances" beginning on page 6-12.

To store data that can be used by only one instance of a handler, use the `SetMessageHandlerInstanceContext` function described on page 6-19. To retrieve the data that has been set, use the `GetMessageHandlerInstanceContext` function described on page 6-20.

Global data is discussed in the section "Global Data Storage for Printing Extensions and Printer Drivers" beginning on page 6-7.

Sending and Forwarding Messages

This section describes the functions the QuickDraw GX Message Manager provides for sending and forwarding messages.

SendMessage

You can use the `SendMessage` function to send a specified message to the current message target.

```
OSErr SendMessage (long messageSelector,…);
```

`messageSelector`
: The number of the message to be sent to the message handler.

additional parameters
: Parameters associated with the message sent.

function result An error of type `OSErr`.

DESCRIPTION

The `SendMessage` function dispatches a message to the topmost handler in the message class that is the parent of the current message target.

The `messageSelector` parameter indicates which message is to be sent.

The ellipsis character at the end of the parameter list indicates that the remaining *additional parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message identified by the `messageSelector` parameter. By definition, all message overrides return a result of type `OSErr`. It is an error to call the `SendMessage` function except from within a message handler. In any other case, behavior is undefined.

The `OSErr` error returned may indicate that the message could not be sent. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return an error of type `OSErr`.

SEE ALSO

To forward a specified message to the next message handler, use the `ForwardMessage` function described in the next section.

To forward the current message to the next message handler, use the `ForwardThisMessage` function described on page 6-25.

The use of the `SendMessage` function is described in the section “Sending and Forwarding Messages” beginning on page 6-15.

ForwardMessage

You can use the `ForwardMessage` function to specify the message to be forwarded to the next message handler.

```
OSErr ForwardMessage (long messageSelector,…);
```

`messageSelector`
The number of the message to be forwarded.

additional parameters
Parameters associated with the message sent.

function result An error of type `OSErr`.

DESCRIPTION

The `ForwardMessage` function forwards the message specified by the `messageSelector` parameter to the next message handler. This function is like the `ForwardThisMessage` function, except that any message may be forwarded. The `messageSelector` parameter indicates which message is to be forwarded, as in the `SendMessage` function. By definition, all messages return a function result of type `OSErr`.

The ellipsis character at the end of the parameter list indicates that the remaining *additional parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message identified by the `messageSelector` parameter. By definition, all message overrides return a result of type `OSErr`. It is an error to call the `ForwardMessage` function except from within a message handler. In any other case, behavior is undefined.

The `OSErr` error returned may indicate that the message could not be forwarded. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return an error of type `OSErr`.

SEE ALSO

To send a specified message to the current message target, use the `SendMessage` function described in the previous section.

To forward the current message to the next message handler, use the `ForwardThisMessage` function described in the next section.

The use of the `ForwardMessage` function is described in the section "Sending and Forwarding Messages" beginning on page 6-15.

ForwardThisMessage

You can use the `ForwardThisMessage` function to forward the current message to the next message handler.

```
OSErr ForwardThisMessage (…);
```

parameters	Parameters associated with the message sent.
function result	An error of type `OSErr`.

DESCRIPTION

The `ForwardThisMessage` function explicitly inherits the current message by forwarding it to the next handler in the message class of the current message target. By definition, all message overrides return a function result of type `OSErr`.

The `OSErr` error returned may indicate that the message could not be forwarded. If no error occurs, the function result is `noErr`. In addition, the receiving message handler may return a result of type `OSErr`.

The ellipsis character in the parameter list indicates that the *parameters* are unspecified; the caller must pass whatever parameters are expected by the recipient of the message. It is an error to call the `ForwardThisMessage` function except from within a message handler. In any other case, behavior is undefined.

SEE ALSO

To send a specified message to the current message target, use the `SendMessage` function described on page 6-23.

To forward a specified message to the next message handler, use the `ForwardMessage` function described in the previous section.

The use of the `ForwardThisMessage` function is described in the section “Sending and Forwarding Messages” beginning on page 6-15.

Driver- or Extension-Defined Functions

This section describes the callback function that you must provide for QuickDraw GX to call when initializing global data.

MessageGlobalsInitProc

You can create an initialization function that requests and allocates memory for your global data. The initialization function must have a prototype of this form:

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

`messageGlobals`
A pointer to the global data to be initialized.

DESCRIPTION

You must supply the `MessageGlobalsInitProc` function if you use the `NewMessageGlobals` function to allocate memory for your global data. Once this initialization function is installed, QuickDraw GX calls it whenever you use the `NewMessageGlobals` function.

If your programming environment is MPW, you may use the `A5Init` function that MPW provides to establish an A5 world for your global data:

```
void A5Init (void *globalPtr);
```

SEE ALSO

The `NewMessageGlobals` function is described on page 6-17.

For more information on initializing you global data, see the section “Allocating Memory for and Disposing of Global Data” beginning on page 6-8.

Summary of the Message Manager

Constants and Data Types

Message Manager Gestalt Selector

```
#define gestaltMessageMgrVersion 'mess' /* gestalt version selector */
```

Message Globals Inititialization Procedure

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```

Functions

Allocating Memory for and Disposing of Global Data

```
OSErr NewMessageGlobals      (long msgGlobalsSize,
                              MessageGlobalsInitProc aProc);
OSErr DisposeMessageGlobals  (void);
```

Setting and Getting Global Data for Multiple Handler Instances

```
void *SetMessageHandlerClassContext
                             (void *);
void *GetMessageHandlerClassContext
                             (void);
```

Setting and Getting Global Data for a Single Handler Instance

```
void *SetMessageHandlerInstanceContext
                             (void *);
void *GetMessageHandlerInstanceContext
                             (void);
```

Sending and Forwarding Messages

```
OSErr SendMessage            (long messageSelector…);
OSErr ForwardMessage         (long messageSelector, …);
OSErr ForwardThisMessage     (…);
```

Application-DefinedFunctions

Initializing Memory for Global Data

```
typedef void (*MessageGlobalsInitProc) (void *messageGlobals);
```