This chapter describes the Process Manager, the part of the Macintosh Operating System that provides a cooperative multitasking environment. The Process Manager controls access to shared resources and manages the scheduling and execution of applications. The Finder uses the Process Manager to launch your application when the user opens either your application or a document created by your application. This chapter discusses how your application can control its execution and get information—for example, the number of free bytes in the application's heap—about itself or any other open application.

Although earlier versions of system software provide process management, the Process Manager is available to your application only in system software version 7.0 and later. The Process Manager provides a cooperative multitasking environment, similar to the features provided by the MultiFinder option in earlier versions of system software. You can use the `Gestalt` function to find out if the Process Manager routines are available and to see which features of the `Launch` function are available.

You should read the chapter "Introduction to Processes and Tasks" in this book for an overview of how the Process Manager schedules applications and loads them into memory. If your application needs to launch other applications, you need to read this chapter for information on the high-level function that lets your application launch other applications and the routines you can use to get information about open applications.

To use this chapter, you need to be familiar with how your application uses memory, as described in the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory*. You should also be familiar with how your application receives events, as discussed in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter provides a brief description of the Process Manager and then shows how you can

- control the execution of your application
- get information about your application
- launch other applications or desk accessories
- get information about applications launched by your application
- generate a list of all open applications and information about each one
- terminate the execution of your application

About the Process Manager

The Process Manager schedules the processing of all applications and desk accessories. It allows multiple applications to share CPU time and other resources. Applications share the available memory and access to the CPU. Several applications can be open (loaded into memory) at once, but only one uses the CPU at any one time.

Note
For a complete description of how the Process Manager schedules applications and desk accessories for execution, see the chapter "Introduction to Processes and Tasks" in this book. ◆

The Process Manager also provides a number of routines that allow you to control the execution of processes and to get information about processes, including your own. You can use the Process Manager routines to

- control the execution of your application
- get information about processes
- launch other applications
- launch desk accessories

The Process Manager assigns a process serial number to each open application (or desk accessory, if it is not opened in the context of an application). The process serial number is unique to each process on the local computer and is valid for a single boot of the computer. You can use the process serial number to specify a particular process for most Process Manager routines.

When a user opens or prints a file from the Finder, it uses the Process Manager to launch the application that created the file. The Finder sets up the information from which your application can determine which files to open or print. The Finder information includes a list of files to open or print.

In system software version 7.0 and later, applications that support high-level events (that is, that have the `isHighLevelEventAware` flag set in the `'SIZE'` resource) receive the Finder information through Apple events. The chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication* describes how your application processes Apple events to open or print files.

Applications that do not support high-level events can call the `CountAppFiles`, `GetAppFiles`, and `ClrAppFiles` routines or the `GetAppParms` routine to get the Finder information. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on these routines.

Using the Process Manager

This section shows how you can use the Process Manager to

- obtain information about open processes
- launch applications and desk accessories
- terminate your application

Getting Information About Other Processes

You can call the `GetNextProcess`, `GetFrontProcess`, or `GetCurrentProcess` functions to get the process serial number of a process. The `GetCurrentProcess` function returns the process serial number of the process currently executing, called the **current process.** This is the process whose A5 world is currently valid; this process can be in the background or foreground. The `GetFrontProcess` function returns the process serial number of the foreground process. For example, if your process is running in the background, you can use `GetFrontProcess` to determine which process is in the foreground.

The Process Manager maintains a list of all open processes. You can specify the process serial number of a process currently in the list and call `GetNextProcess` to get the process serial number of the next process in the list. The interpretation of the value of a process serial number and of the order of the list of processes is internal to the Process Manager.

When specifying a particular process, use only a process serial number returned by a high-level event or a Process Manager routine, or constants defined by the Process Manager. You can use these constants to specify special processes:

```
CONST
   kNoProcess        = 0;        {process doesn't exist}
   kSystemProcess    = 1;        {process belongs to OS}
   kCurrentProcess   = 2;        {the current process}
```

In all Process Manager routines, the constant `kNoProcess` refers to a process that doesn't exist, the constant `kSystemProcess` refers to a process belonging to the Operating System, and the constant `kCurrentProcess` refers to the current process.

To begin enumerating a list of processes, call the `GetNextProcess` function and specify the constant `kNoProcess` as the parameter. In response, `GetNextProcess` returns the process serial number of the first process in the list. You can use the returned process serial number to get the process serial number of the next process in the list. When the `GetNextProcess` function reaches the end of the list, it returns the constant `kNoProcess` and the result code `procNotFound`.

You can also use a process serial number to specify a target application when your application sends a high-level event. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to use a process serial number when your application sends a high-level event.

You can call the `GetProcessInformation` function to obtain information about any process, including your own. For example, for a specified process, you can find

- the application's name as it appears in the Application menu
- the type and signature of the application
- the number of bytes in the application partition
- the number of free bytes in the application heap
- the application that launched the application

The `GetProcessInformation` function returns information in a process information record, which is defined by the `ProcessInfoRec` data type.

```
TYPE ProcessInfoRec =
   RECORD
      processInfoLength:   LongInt;       {length of process info record}
      processName:         StringPtr;     {name of this process}
      processNumber:       ProcessSerialNumber;
                                          {psn of this process}
      processType:         LongInt;       {file type of application file}
      processSignature:    OSType;        {signature of application file}
      processMode:         LongInt;       {'SIZE' resource flags}
      processLocation:     Ptr;           {address of partition}
      processSize:         LongInt;       {partition size}
      processFreeMem:      LongInt;       {free bytes in heap}
      processLauncher:     ProcessSerialNumber;
                                          {process that launched this one}
      processLaunchDate:   LongInt;       {time when launched}
      processActiveTime:   LongInt;       {accumulated CPU time}
      processAppSpec:      FSSpecPtr;     {location of the file}
   END;
```

You specify the values for three fields of the process information record: `processInfoLength`, `processName`, and `processAppSpec`. You must either set the `processName` and `processAppSpec` fields to `NIL` or set these fields to point to memory that you have allocated for them. The `GetProcessInformation` function returns information in all other fields of the process information record. See “Process Information Record” on page 2-16 for a complete description of the fields of this record.

Listing 2-1 shows how you can use the `GetNextProcess` function with the `GetProcessInformation` function to search the process list for a specific process.

Listing 2-1 Searching for a specific process

```
FUNCTION FindAProcess (signature: OSType;
                       VAR process: ProcessSerialNumber;
                       VAR InfoRec: ProcessInfoRec;
                       myFSSpecPtr: FSSpecPtr;
                       myName: Str31): Boolean;
BEGIN
   FindAProcess := FALSE;              {assume FALSE}
   process.highLongOfPSN := 0;
   process.lowLongOfPSN := kNoProcess; {start at the beginning}

   InfoRec.processInfoLength := sizeof(ProcessInfoRec);
```

```
   InfoRec.processName := myName;
   InfoRec.processAppSpec := myFSSpecPtr;

   WHILE (GetNextProcess(process) = noErr) DO
   BEGIN
      IF GetProcessInformation(process, InfoRec) = noErr THEN
      BEGIN
         IF (InfoRec.processType = LongInt('APPL')) AND
            (InfoRec.processSignature = signature) THEN
         BEGIN                          {found the process}
            FindAProcess := TRUE;
            Exit(FindAProcess);
         END;
      END;
   END; {WHILE}
END;
```

The code in Listing 2-1 searches the process list for the application with the specified signature. For example, you might want to find a specific process so that you can send a high-level event to it.

Launching Other Applications

You can launch other applications by calling the high-level `LaunchApplication` function. This function lets your application control various options associated with launching an application. For example, you can

- allow the application to be launched in a partition smaller than the preferred size but greater than the minimum size, or allow it to be launched only in a partition of the preferred size
- launch an application without terminating your own application, bring the launched application to the front, and get information about the launched application
- request that your application be notified if any application that it has launched terminates

Earlier versions of system software used a shorter parameter block as a parameter to the `_Launch` trap macro. The `_Launch` trap macro still supports the use of this parameter block. Applications using the `LaunchApplication` function should use the new launch parameter block (of type `LaunchParamBlockRec`). Use the `Gestalt` function and specify the selector `gestaltOSAttr` to determine which launch features are available.

Most applications don't need to launch other applications. However, if your application includes a desk accessory or another application, you might use either the high-level `LaunchApplication` function to launch an application or the `LaunchDeskAccessory` function to launch a desk accessory. For example, if you have implemented a spelling checker as a separate application, you might use the

LaunchApplication function to open the spelling checker when the user chooses Check Spelling from one of your application's menus.

You specify a launch parameter block as a parameter to the LaunchApplication function. In this launch parameter block, you can specify the filename of the application to launch, specify whether to allow launching only in a partition of the preferred size or to allow launching in a smaller partition, and set various other options—for example, whether your application should continue or terminate after it launches the specified application.

The LaunchApplication function launches the application from the specified file and returns the process serial number, preferred partition size, and minimum partition size if the application is successfully launched.

Note that if you launch another application without terminating your application, the launched application does not actually begin executing until you make a subsequent call to WaitNextEvent or EventAvail.

The launch parameter block is defined by the LaunchParamBlockRec data type.

```
TYPE LaunchParamBlockRec =
   RECORD
      reserved1:              LongInt;             {reserved}
      reserved2:              Integer;             {reserved}
      launchBlockID:          Integer;             {extended block}
      launchEPBLength:        LongInt;             {length of block}
      launchFileFlags:        Integer;             {app's Finder flags}
      launchControlFlags:     LaunchFlags;         {launch options}
      launchAppSpec:          FSSpecPtr;           {location of app's file}
      launchProcessSN:        ProcessSerialNumber; {returned psn}
      launchPreferredSize:    LongInt;             {returned pref size}
      launchMinimumSize:      LongInt;             {returned min size}
      launchAvailableSize:    LongInt;             {returned avail size}
      launchAppParameters:    AppParametersPtr;    {high-level event}
   END;
```

In the launchBlockID field, specify the constant extendedBlock to identify the parameter block and to indicate that you are using the fields following it in the launch parameter block.

```
CONST
   extendedBlock              = $4C43;    {extended block}
```

In the `launchEPBLength` field, specify the constant `extendedBlockLen` to indicate the length of the remaining fields in the launch parameter block (that is, the length of the fields following the `launchEPBLength` field). For compatibility, you should always specify the length value in this field.

```
CONST
   extendedBlockLen            = sizeof(LaunchParamBlockRec) - 12;
```

The `launchFileFlags` field contains the Finder flags for the application file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the Finder flags.) The `LaunchApplication` function sets this field for you if you set the bit defined by the `launchNoFileFlags` constant in the `launchControlFlags` field. Otherwise, you must get the Finder flags from the application file and set this field yourself (by using the File Manager routine `FSpGetFInfo`, for example).

In the `launchControlFlags` field, you specify various options that control how the specified application is launched. See the section "Launch Options" on page 2-15 for information on the launch control flags.

You specify the application to launch in the `launchAppSpec` field of the launch parameter block. In this field, you specify a pointer to a file system specification record (`FSSpec`). See the chapter "File Manager" in *Inside Macintosh: Files* for a complete description of the file system specification record.

The `LaunchApplication` function sets the initial default directory of the application to the parent directory of the application file.

If it successfully launches the application, `LaunchApplication` returns, in the `launchProcessSN` field, a process serial number. You can use this number in Process Manager routines to refer to this application.

The `LaunchApplication` function returns the `launchPreferredSize` and `launchMinimumSize` fields of the launch parameter block. The values of these fields are based on their corresponding values in the `'SIZE'` resource. These values may be greater than those specified in the application's `'SIZE'` resource because the returned sizes include any adjustments to the size of the application's stack. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how the size of the application stack is adjusted. Values are always returned in these fields whether or not the launch was successful. These values are 0 if an error occurred—for example, if the application file could not be found.

The `LaunchApplication` function returns a value in the `launchAvailableSize` field only when the `memFullErr` result code is returned. This value indicates the largest partition size currently available for allocation.

The `launchAppParameters` field specifies the first high-level event sent to an application. If you set this field to `NIL`, the `LaunchApplication` function automatically creates and sends an Open Application event to the launched application. (See the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication* for a description of this event.) To send a particular high-level event to

the launched application, you can specify a pointer to an application parameters record. The application parameters record is defined by the data type `AppParameters`.

```
TYPE AppParameters =
   RECORD
      theMsgEvent:      EventRecord;       {event (high-level)}
      eventRefCon:      LongInt;           {reference constant}
      messageLength:    LongInt;           {length of buffer}
      messageBuffer:    ARRAY [0..0] OF SignedByte;
   END;
```

You specify the high-level event in the fields `theMsgEvent`, `eventRefCon`, `messageLength`, and `messageBuffer`.

Listing 2-2 demonstrates how you can use the `LaunchApplication` function.

Listing 2-2 Launching an application

```
PROCEDURE LaunchAnApplication (mySFReply: StandardFileReply);
VAR
   myLaunchParams:      LaunchParamBlockRec;
   launchedProcessSN:   ProcessSerialNumber;
   launchErr:           OSErr;
   prefSize:            LongInt;
   minSize:             LongInt;
   availSize:           LongInt;
BEGIN
   WITH myLaunchParams DO
   BEGIN
      launchBlockID := extendedBlock;
      launchEPBLength := extendedBlockLen;
      launchFileFlags := 0;
      launchControlFlags := launchContinue + launchNoFileFlags;
      launchAppSpec := @mySFReply.sfFile;
      launchAppParameters := NIL;
   END;
   launchErr := LaunchApplication(@myLaunchParams);

   prefsize := myLaunchParams.launchPreferredSize;
   minsize := myLaunchParams.launchMinimumSize;
   IF launchErr = noErr THEN
      launchedProcessSN := myLaunchParams.launchProcessSN
   ELSE IF launchErr = memFullErr THEN
      availSize := myLaunchParams.launchAvailableSize
```

```
   ELSE
      DoError(launchErr);
END;
```

Listing 2-2 indicates which application file to launch by using a file system specification record (perhaps returned by the `StandardGetFile` routine) and specifying, in the `launchAppSpec` field, a pointer to this record. The `launchControlFlags` field indicates that `LaunchApplication` should extract the Finder flags from the application file, launch the application in a partition of the preferred size, bring the launched application to the front, and not terminate the current process.

By default, `LaunchApplication` brings the launched application to the front and sends the foreground application to the background. If you don't want to bring an application to the front when it is first launched, set the `launchDontSwitch` flag in the `launchControlFlags` field of the launch parameter block.

In addition, if you want your application to continue to run after it launches another application, you must set the `launchContinue` flag in the `launchControlFlags` field of the launch parameter block. For a complete description of the available launch control options, see "Launch Options" on page 2-15.

If you want your application to be notified about the termination of an application it has launched, set the `acceptAppDiedEvents` flag in your `'SIZE'` resource. If you set this flag and an application launched by your application terminates, your application receives an Application Died Apple event (`'aevt' 'obit'`). See "Terminating an Application" on page 2-11 for more information on the Application Died event.

Launching Desk Accessories

In system software version 7.0 and later, the Process Manager launches a desk accessory in its own partition when that desk accessory is opened, giving it a process serial number and an entry in the process list. The Process Manager puts the name of the desk accessory in the list of open applications in the Application menu and also gives the active desk accessory its own About menu item in the Apple menu containing the name of the desk accessory. This makes desk accessories more consistent with the user interface of small applications.

Although you can use the `LaunchDeskAccessory` function to launch desk accessories, you should use it only when your application needs to launch a desk accessory for some reason other than the user's choosing a desk accessory from the Apple menu. Beginning in system software version 7.0, the Apple menu can contain any Finder object that the user decides to add to the menu. When the user chooses any such user-added item from the Apple menu, your application should respond by calling the `OpenDeskAcc` function instead.

Terminating an Application

The Process Manager automatically terminates a process when the process either exits its main routine or encounters a fatal error condition (such as an attempt to divide by 0).

When a process terminates, the Process Manager takes care of any required cleanup operations; these include removing the process from the list of open processes and releasing the memory occupied by the application partition (as well as any temporary memory the process still holds). If necessary, the Process Manager sends an Application Died event to the process that launched the one about to terminate.

Your application can also terminate itself directly by calling the `ExitToShell` procedure. In general, you need to call `ExitToShell` only if you want to terminate your application without having it return from its main routine. This might be useful when your initialization code detects that some essential system capability is not available (for instance, when the computer running a stereo sound-editing application does not support stereo sound playback). Listing 2-3 shows one way to exit gracefully in this situation.

Listing 2-3 Terminating an application

```
PROCEDURE CheckForStereoSound;
VAR
   myErr:      OSErr;           {result code from Gestalt}
   myFeature:  LongInt;         {features bit flags from Gestalt}
   myString:   Str255;          {text of alert message}
   myItem:     Integer;         {item returned by StopAlert}
CONST
   kAlertBoxID    = 128;        {resource ID of alert template}
   kAlertStrings  = 128;        {resource ID of alert strings}
   kNoStereoAlert = 5;          {index of No Stereo alert text}
BEGIN
   myErr := Gestalt(gestaltSoundAttr, myFeature);
   IF myErr = noErr THEN
      IF BTst(myFeature, gestaltStereoCapability) = FALSE THEN
         BEGIN
            GetIndString(myString, kAlertStrings, kNoStereoAlert);
            ParamText(myString, '', '', '');
            myItem := StopAlert(kAlertBoxID, NIL);
            ExitToShell;        {exit the application}
         END
   ELSE
      DoError(myErr);
END;
```

The procedure `CheckForStereoSound` defined in Listing 2-3 checks whether the computer supports stereo sound playback. If not, `CheckForStereoSound` notifies the user by displaying an alert box and terminates the application by calling `ExitToShell`.

Note
The `ExitToShell` procedure is the only means of terminating a process. It is always called during process termination, whether by your application itself, the Process Manager, or some other process. ◆

If your application launches another application that terminates, either normally or as the result of an error, the Process Manager can notify your application by sending it an Application Died event. To request this notification, you must set the `acceptAppDied` flag in your application's `'SIZE'` resource. (For a complete description of the `'SIZE'` resource, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

Application Died—inform that an application has terminated

Event ID	`kAEApplicationDied`
Required parameters	
Keyword	`keyErrorNumber`
Descriptor type	`typeLongInteger`
Data	A sign-extended `OSErr` value. A value of `noErr` indicates normal termination; any other value indicates that the application terminated because of an error.
Keyword	`keyProcessSerialNumber`
Descriptor type	`typeProcessSerialNumber`
Data	The process serial number of the application that terminated.
Requested action	None. This Apple event is sent only to provide information.

The Process Manager gets the value of the `keyErrorNumber` parameter from the system global variable `DSErrCode`. This value can be set either by the application before it terminates or by the Operating System (when the application terminates as the result of a hardware exception or other problem).

Process Manager Reference

This section describes the constants, data structures, and routines that are specific to the Process Manager.

Constants

You can use Process Manager constants to get information about the attributes of the Process Manager, identify certain special processes, and specify launch options.

Gestalt Selector and Response Bits

You can determine if the Process Manager is available and find out which features of the launch routine are available by calling the `Gestalt` function with the selector `gestaltOSAttr`.

```
CONST
   gestaltOSAttr              = 'os  ';{O/S attributes}
```

The `Gestalt` function returns information by setting or clearing bits in the `response` parameter. The following constants define the bits currently used:

```
CONST
   gestaltLaunchCanReturn     = 1;  {can return from launch}
   gestaltLaunchFullFileSpec  = 2;  {LaunchApplication available}
   gestaltLaunchControl       = 3;  {Process Manager is available}
```

Constant descriptions

`gestaltLaunchCanReturn`
: Set if the `_Launch` trap macro can return to the caller. The `_Launch` trap macro in system software version 7.0 (and in earlier versions running MultiFinder) gives your application the option to continue running after it launches another application. In earlier versions of system software not running MultiFinder, the `_Launch` trap macro forces the launching application to quit.

`gestaltLaunchFullFileSpec`
: Set if the `launchControlFlags` field supports control flags in addition to the `launchContinue` flag, and if the `_Launch` trap can process the `launchAppSpec`, `launchProcessSN`, `launchPreferredSize`, `launchMinimumSize`, `launchAvailableSize`, and `launchAppParameters` fields in the launch parameter block.

`gestaltLaunchControl`
: Set if the Process Manager is available.

Process-Identification Constants

The Process Manager provides three constants that can be used instead of a process serial number to identify a process:

```
CONST
   kNoProcess        = 0;        {process doesn't exist}
   kSystemProcess    = 1;        {process belongs to OS}
   kCurrentProcess   = 2;        {the current process}
```

Constant descriptions

`kNoProcess` Identifies a process that doesn't exist.

`kSystemProcess`
Identifies a process that belongs to the Operating System.

`kCurrentProcess`
Identifies the current process.

Launch Options

When you use the `LaunchApplication` function, you specify the launch options in the `launchControlFlags` field of the launch parameter block. These are the constants you can specify in the `launchControlFlags` field:

```
CONST
   launchContinue          = $4000;
   launchNoFileFlags       = $0800;
   launchUseMinimum        = $0400;
   launchDontSwitch        = $0200;
   launchInhibitDaemon     = $0080;
```

Constant descriptions

`launchContinue` Set this flag if you want your application to continue after the specified application is launched. If you do not set this flag, `LaunchApplication` terminates your application after launching the specified application, even if the launch fails.

`launchNoFileFlags`
Set this flag if you want the `LaunchApplication` function to ignore any value specified in the `launchFileFlags` field. If you set the `launchNoFileFlags` flag, the `LaunchApplication` function extracts the Finder flags from the application file for you. If you want to supply the file flags, clear the `launchNoFileFlags` flag and specify the Finder flags in the `launchFileFlags` field of the launch parameter block.

`launchUseMinimum`
Clear this flag if you want the `LaunchApplication` function to attempt to launch the application in the preferred size (as specified in the application's `'SIZE'` resource). If you set the `launchUseMinimum` flag, the `LaunchApplication` function attempts to launch the application using the largest available size greater than or equal to the minimum size but less than the preferred size. If the `LaunchApplication` function returns the result code `memFullErr` or `memFragErr`, the application cannot be launched under the current memory conditions.

`launchDontSwitch`
Set this flag if you do not want the launched application brought to the front. If you set this flag, the launched application runs in the background until the user brings the application to the front—for

example, by clicking in one of the application's windows. Note that most applications expect to be launched in the foreground. If you clear the `launchDontSwitch` flag, the launched application is brought to the front, and your application is sent to the background.

`launchInhibitDaemon`
Set this flag if you do not want `LaunchApplication` to launch a background-only application. (A background-only application has the `onlyBackground` flag set in its `'SIZE'` resource.)

Data Structures

This section describes the data structures that you use to provide information to the Process Manager or that the Process Manager uses to return information to your application.

Process Serial Number

The Process Manager uses process serial numbers to identify open processes. A process serial number is a 64-bit quantity whose structure is defined by the `ProcessSerialNumber` data type.

IMPORTANT
The meaning of the bits in a process serial number is internal to the Process Manager. You should not attempt to interpret the value of the process serial number. If you need to compare two process serial numbers, call the `SameProcess` function. ▲

```
TYPE ProcessSerialNumber   =
   RECORD
      highLongOfPSN:    LongInt;    {high-order 32 bits of psn}
      lowLongOfPSN:     LongInt;    {low-order 32 bits of psn}
   END;
```

Field descriptions

`highLongOfPSN` The high-order long integer of the process serial number.

`lowLongOfPSN` The low-order long integer of the process serial number.

Process Information Record

The `GetProcessInformation` function returns information in a process information record, which is defined by the `ProcessInfoRec` data type.

```
TYPE ProcessInfoRec =
   RECORD
      processInfoLength:   LongInt;       {length of process info record}
      processName:         StringPtr;     {name of this process}
```

```
   processNumber:       ProcessSerialNumber;
                                {psn of this process}
   processType:         LongInt;       {file type of application file}
   processSignature:    OSType;        {signature of application file}
   processMode:         LongInt;       {'SIZE' resource flags}
   processLocation:     Ptr;           {address of partition}
   processSize:         LongInt;       {partition size}
   processFreeMem:      LongInt;       {free bytes in heap}
   processLauncher:     ProcessSerialNumber;
                                {process that launched this one}
   processLaunchDate:   LongInt;       {time when launched}
   processActiveTime:   LongInt;       {accumulated CPU time}
   processAppSpec:      FSSpecPtr;     {location of the file}
END;
```


Field descriptions

`processInfoLength`
The number of bytes in the process information record. For compatibility, you should specify the length of the record in this field.

`processName` The name of the application or desk accessory. For applications, this field contains the name of the application as designated by the user at the time the application was opened. For example, for foreground applications, the `processName` field contains the name as it appears in the Application menu. For desk accessories, the `processName` field contains the name of the `'DRVR'` resource. You must specify `NIL` in the `processName` field if you do not want the application name or the desk accessory name returned. Otherwise, you should allocate at least 32 bytes of storage for the string pointed to by the `processName` field. Note that the `processName` field specifies the name of either the application or the `'DRVR'` resource, whereas the `processAppSpec` field specifies the location of the file.

`processNumber` The process serial number. The process serial number is a 64-bit number; the meaning of these bits is internal to the Process Manager. You should not attempt to interpret the value of the process serial number.

`processType` The file type of the application, generally `'APPL'` for applications and `'appe'` for background-only applications launched at startup. If the process is a desk accessory, this field specifies the type of the file containing the `'DRVR'` resource.

`processSignature`
The signature of the file containing the application or the `'DRVR'` resource (for example, the signature of the TeachText application is `'ttxt'`).

`processMode` Process mode flags. These flags indicate whether the process is an application or desk accessory. For applications, this field also returns information specified in the application's `'SIZE'` resource. This information is returned as flags. You can refer to these flags by using these constants:

```
CONST
   modeDeskAccessory            = $00020000;
   modeMultiLaunch              = $00010000;
   modeNeedSuspendResume        = $00004000;
   modeCanBackground            = $00001000;
   modeDoesActivateOnFGSwitch   = $00000800;
   modeOnlyBackground           = $00000400;
   modeGetFrontClicks           = $00000200;
   modeGetAppDiedMsg            = $00000100;
   mode32BitCompatible          = $00000080;
   modeHighLevelEventAware      = $00000040;
   modeLocalAndRemoteHLEvents   = $00000020;
   modeStationeryAware          = $00000010;
   modeUseTextEditServices      = $00000008;
```

`processLocation`
The beginning address of the application partition.

`processSize` The number of bytes in the application partition (including the heap, stack, and A5 world).

`processFreeMem`
The number of free bytes in the application heap.

`processLauncher`
The process serial number of the process that launched the application or desk accessory. If the original launcher of the process is no longer open, this field contains the constant `kNoProcess`.

`processLaunchDate`
The value of the `Ticks` global variable at the time that the process was launched.

`processActiveTime`
The accumulated time, in ticks, during which the process has used the CPU, including both foreground and background processing time.

`processAppSpec`
The address of a file specification record that stores the location of the file containing the application or `'DRVR'` resource. You should specify `NIL` in the `processAppSpec` field if you do not want the `FSSpec` record of the file returned.

Launch Parameter Block

You specify a launch parameter block as a parameter to the `LaunchApplication` function. The launch parameter block is defined by the `LaunchParamBlockRec` data type.

```
TYPE LaunchParamBlockRec =
   RECORD
      reserved1:              LongInt;             {reserved}
      reserved2:              Integer;             {reserved}
      launchBlockID:          Integer;             {extended block}
      launchEPBLength:        LongInt;             {length of block}
      launchFileFlags:        Integer;             {app's Finder flags}
      launchControlFlags:     LaunchFlags;         {launch options}
      launchAppSpec:          FSSpecPtr;           {location of app's file}
      launchProcessSN:        ProcessSerialNumber; {returned psn}
      launchPreferredSize:    LongInt;             {returned pref size}
      launchMinimumSize:      LongInt;             {returned min size}
      launchAvailableSize:    LongInt;             {returned avail size}
      launchAppParameters:    AppParametersPtr;    {high-level event}
   END;
```

Field descriptions

`reserved1` Reserved.

`reserved2` Reserved.

`launchBlockID` A value that indicates whether you are using the fields following it in the launch parameter block. Specify the constant `extendedBlock` if you use the fields that follow it.

`launchEPBLength`
The length of the fields following this field in the launch parameter block. Use the constant `extendedBlockLen` to specify this value.

`launchFileFlags`
The Finder flags for the application file. Set the `launchNoFileFlags` constant in the `launchControlFlags` field if you want the `LaunchApplication` function to extract the Finder flags from the application file and to set the `launchFileFlags` field for you.

`launchControlFlags`
The launch options that determine how the application is launched. You can specify these constant values to set various options:

```
CONST
   launchContinue       = $4000;
   launchNoFileFlags    = $0800;
```

```
launchUseMinimum     = $0400;
launchDontSwitch     = $0200;
launchInhibitDaemon  = $0080;
```

See “Launch Options” on page 2-15 for a complete description of these flags.

`launchAppSpec` A pointer to a file specification record that gives the location of the application file to launch.

`launchProcessSN`
The process serial number returned to your application if the launch is successful. You can use this process serial number in other Process Manager routines to refer to the launched application.

`launchPreferredSize`
The preferred partition size for the launched application as specified in the launched application’s `'SIZE'` resource. `LaunchApplication` sets this field to 0 if an error occurred or if the application is already open.

`launchMinimumSize`
The minimum partition size for the launched application as specified in the launched application’s `'SIZE'` resource. `LaunchApplication` sets this field to 0 if an error occurred or if the application is already open.

`launchAvailableSize`
The maximum partition size that is available for allocation. This value is returned to your application only if the `memFullErr` result code is returned. If the application launch fails because of insufficient memory, you can use this value to determine if there is enough memory available to launch in the minimum size.

`launchAppParameters`
The first high-level event to send to the launched application. If you set this field to `NIL`, `LaunchApplication` creates and sends the Open Application Apple event to the launched application.

Application Parameters Record

You specify an application parameters record in the `launchAppParameters` field of the launch parameter block whose address is passed to the `LaunchApplication` function. This record specifies the first high-level event to be sent to the newly launched application. The application parameters record is defined by the `AppParameters` data type.

```
TYPE AppParameters =
   RECORD
      theMsgEvent:      EventRecord;       {event (high-level)}
      eventRefCon:      LongInt;           {reference constant}
```

```
    messageLength:    LongInt;          {length of buffer}
    messageBuffer:    ARRAY [0..0] OF SignedByte;
END;
```

Field descriptions

`theMsgEvent`	The event record specifying the first high-level event to be sent to the launched application.
`eventRefCon`	A reference constant. Your application can use this field for its own purposes.
`messageLength`	The length of the buffer specified by the `messageBuffer` field.
`messageBuffer`	A buffer of data. The nature of this data varies according to the event being sent.

Routines

This section describes the Process Manager routines you can use to get information about any currently open applications, to control process execution, to launch other applications, and to terminate your application.

Getting Process Information

You can use the Process Manager to get the process serial number of a particular process, to generate a list of all open processes, to get information about processes, or to change the scheduling status of a process.

GetCurrentProcess

Use the `GetCurrentProcess` function to get information about the current process, if any.

```
FUNCTION GetCurrentProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On output, the process serial number of the current process.

DESCRIPTION

The `GetCurrentProcess` function returns, in the `PSN` parameter, the process serial number of the process that is currently running, that is, the one currently accessing the CPU. This is the application associated with the `CurrentA5` global variable. This application can be running in either the foreground or the background.

Applications can use this function to find their own process serial number. Drivers can use this function to find the process serial number of the current process. You can use the returned process serial number in other Process Manager routines.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetCurrentProcess` function are

Trap macro	Selector
`_OSDispatch`	$0037

RESULT CODE

`noErr`	0	No error

GetNextProcess

Use the `GetNextProcess` function to get information about the next process, if any, in the Process Manager's internal list of open processes.

```
FUNCTION GetNextProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On input, the process serial number of a process. This number should be a valid process serial number returned from `LaunchApplication`, `GetNextProcess`, `GetFrontProcess`, or `GetCurrentProcess`, or else the defined constant `kNoProcess`. On output, the process serial number of the next process, or else `kNoProcess`.

DESCRIPTION

The Process Manager maintains a list of all open processes. You can derive this list by using repetitive calls to `GetNextProcess`. Begin generating the list by calling `GetNextProcess` and specifying the constant `kNoProcess` in the `PSN` parameter. You can then use the returned process serial number to get the process serial number of the next process. Note that the order of the list of processes is internal to the Process Manager. When the end of the list is reached, `GetNextProcess` returns the constant `kNoProcess` in the `PSN` parameter and the result code `procNotFound`.

You can use the returned process serial number in other Process Manager routines. You can also use this process serial number to specify a target application when your application sends a high-level event.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetNextProcess` function are

Trap macro	Selector
`_OSDispatch`	$0038

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid
`procNotFound`	–600	No process in the process list following the specified process

GetProcessInformation

Use the `GetProcessInformation` function to get information about a specific process.

```
FUNCTION GetProcessInformation (PSN: ProcessSerialNumber;
                                VAR info: ProcessInfoRec): OSErr;
```

`PSN` The process serial number of a process. This number should be a valid process serial number returned from `LaunchApplication`, `GetNextProcess`, `GetFrontProcess`, `GetCurrentProcess`, or else a high-level event. You can use the constant `kCurrentProcess` to get information about the current process.

`info` A record containing information about the specified process.

DESCRIPTION

The `GetProcessInformation` function returns, in a process information record, information about the specified process. The information returned in the `info` parameter includes the application's name as it appears in the Application menu, the type and signature of the application, the address of the application partition, the number of bytes in the application partition, the number of free bytes in the application heap, the application that launched the application, the time at which the application was launched, and the location of the application file. See "Getting Information About Other Processes" on page 2-5 for the structure of the process information record.

The `GetProcessInformation` function also returns information about the application's `'SIZE'` resource and indicates whether the process is an application or a desk accessory.

You need to specify values for the `processInfoLength`, `processName`, and `processAppSpec` fields of the process information record. Specify the length of the process information record in the `processInfoLength` field. If you do not want information returned in the `processName` and `processAppSpec` fields, specify `NIL` for these fields. Otherwise, allocate at least 32 bytes of storage for the string pointed to by the `processName` field and, in the `processAppSpec` field, specify a pointer to an `FSSpec` record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetProcessInformation` function are

Trap macro	Selector
`_OSDispatch`	$003A

SPECIAL CONSIDERATIONS

Do not call `GetProcessInformation` at interrupt time.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid

SameProcess

Use the `SameProcess` function to determine whether two process serial numbers specify the same process.

```
FUNCTION SameProcess (PSN1, PSN2: ProcessSerialNumber;
                      VAR result: Boolean): OSErr;
```

`PSN1` A process serial number.

`PSN2` A process serial number.

`result` A Boolean value that indicates whether the process serial numbers passed in `PSN1` and `PSN2` refer to the same process.

DESCRIPTION

The `SameProcess` function compares two process serial numbers and determines whether they refer to the same process. If the process serial numbers specified in the `PSN1` and `PSN2` parameters refer to the same process, the `SameProcess` function returns `TRUE` in the `result` parameter; otherwise, it returns `FALSE` in the `result` parameter.

Do not attempt to compare two process serial numbers by any means other than the `SameProcess` function, because the interpretation of the bits in a process serial number is internal to the Process Manager.

The values of `PSN1` and `PSN2` must be valid process serial numbers returned from `LaunchApplication`, `GetNextProcess`, `GetFrontProcess`, `GetCurrentProcess`, or a high-level event. You can also use the constant `kCurrentProcess` to refer to the current process.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SameProcess` function are

Trap macro	Selector
`_OSDispatch`	$003D

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid

GetFrontProcess

Use the `GetFrontProcess` function to get the process serial number of the front process.

```
FUNCTION GetFrontProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On output, the process serial number of the process running in the foreground.

DESCRIPTION

The `GetFrontProcess` function returns, in the `PSN` parameter, the process serial number of the process running in the foreground. You can use this function to determine if your process or some other process is in the foreground. You can use the process serial number returned in the `PSN` parameter in other Process Manager routines.

If no process is running in the foreground, `GetFrontProcess` returns the result code `procNotFound`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetFrontProcess` function are

Trap macro	Selector
`_OSDispatch`	$0039

RESULT CODES

noErr	0	No error
paramErr	–50	Process serial number is invalid
procNotFound	–600	No process in the foreground

SetFrontProcess

Use the SetFrontProcess function to set the front process.

```
FUNCTION SetFrontProcess (PSN: ProcessSerialNumber): OSErr;
```

PSN — The process serial number of the process you want to move to the foreground. This number should be a valid process serial number returned from LaunchApplication, GetNextProcess, GetFrontProcess, GetCurrentProcess, or a high-level event. You can also use the constant kCurrentProcess to refer to the current process.

DESCRIPTION

The SetFrontProcess function schedules the specified process to move to the foreground. The specified process moves to the foreground after the current foreground process makes a subsequent call to WaitNextEvent or EventAvail.

If the specified process serial number is invalid or if the specified process is a background-only application, SetFrontProcess returns a nonzero result code and does not change the current foreground process.

If a modal dialog box is the frontmost window, the specified process remains in the background until the user dismisses the modal dialog box.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SetFrontProcess function are

Trap macro	Selector
_OSDispatch	$003B

SPECIAL CONSIDERATIONS

Do not call SetFrontProcess interrupt time.

RESULT CODES

noErr	0	No error
procNotFound	–600	Process with specified process serial number doesn't exist or process is suspended by high-level debugger
appIsDaemon	–606	Specified process runs only in the background

WakeUpProcess

Use the WakeUpProcess function to make a process suspended by WaitNextEvent eligible to receive CPU time.

```
FUNCTION WakeUpProcess (PSN: ProcessSerialNumber): OSErr;
```

PSN — The process serial number of the process to be made eligible. This number should be a valid process serial number returned from LaunchApplication, GetNextProcess, GetFrontProcess, GetCurrentProcess, or a high-level event. You can also use the constant kCurrentProcess to refer to the current process.

DESCRIPTION

The WakeUpProcess function makes a process suspended by WaitNextEvent eligible to receive CPU time. A process is suspended when the value of the sleep parameter in the WaitNextEvent function is not 0 and no events for that process are pending in the event queue. This process remains suspended until the time specified in the sleep parameter expires or an event becomes available for that process. You can use WakeUpProcess to make the process eligible for execution before the time specified in the sleep parameter expires.

The WakeUpProcess function does not change the order of the processes scheduled for execution; it only makes the specified process eligible for execution.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the WakeUpProcess function are

Trap macro	Selector
_OSDispatch	$003C

RESULT CODES

noErr	0	No error
procNotFound	–600	Suspended process with specified process serial number doesn't exist

Launching Applications and Desk Accessories

Your application can use the `LaunchApplication` function to launch other applications and the `LaunchDeskAccessory` function to launch desk accessories.

LaunchApplication

You can use the `LaunchApplication` function to launch an application.

```
FUNCTION LaunchApplication (LaunchParams: LaunchPBPtr): OSErr;
```

`LaunchParams`
A pointer to a launch parameter block specifying information about the application to launch.

Parameter block

→	launchBlockID	Integer	Extended block
→	launchEPBLength	LongInt	Length of following fields
→	launchFileFlags	Integer	Finder flags for the application file
→	launchControlFlags	LaunchFlags	Flags for launch options
→	launchAppSpec	FSSpecPtr	Location of application file to launch
←	launchProcessSN	ProcessSerialNumber	Process serial number
←	launchPreferredSize	LongInt	Preferred application partition size
←	launchMinimumSize	LongInt	Minimum application partition size
←	launchAvailableSize	LongInt	Maximum available partition size
→	launchAppParameters	AppParametersPtr	High-level event for launched application

DESCRIPTION

The `LaunchApplication` function launches the application from the specified file and returns the process serial number, preferred partition size, and minimum partition size if the application is successfully launched.

Note that if you launch another application without terminating your application, the launched application is not actually executed until you make a subsequent call to `WaitNextEvent` or `EventAvail`.

Set the `launchContinue` flag in the `launchControlFlags` field of the launch parameter block if you want your application to continue after the specified application is launched. If you do not set this flag, `LaunchApplication` terminates your application after launching the specified application, even if the launch fails.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and registers on entry and exit for `LaunchApplication` are

Trap macro

```
_Launch
```

Registers on entry

A0 Pointer to launch parameter block

Registers on exit

A0 Pointer to launch parameter block

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to allocate the partition size specified in the `'SIZE'` resource
`memFragErr`	–601	Not enough room to launch application with special requirements
`appModeErr`	–602	Memory mode is 32-bit, but application is not 32-bit clean
`appMemFullErr`	–605	More memory is required for the partition size than the amount specified in the `'SIZE'` resource
`appIsDaemon`	–606	Application runs only in the background, and launch flags don't allow background-only applications

LaunchDeskAccessory

You can use the `LaunchDeskAccessory` function to launch desk accessories. Use this function only when your application needs to launch a desk accessory for some reason other than the user's choosing one from the Apple menu. (When the user chooses any Apple menu item that is not specific to your application, use the `OpenDeskAcc` function.)

```
FUNCTION LaunchDeskAccessory (pFileSpec: FSSpecPtr;
                              pDAName: StringPtr): OSErr;
```

`pFileSpec` A pointer to a file system specification of the resource fork to search for the specified desk accessory.

`pDAName` The name of the `'DRVR'` resource to launch.

DESCRIPTION

The `LaunchDeskAccessory` function searches the resource fork of the file specified by the `pFileSpec` parameter for the desk accessory with the `'DRVR'` resource name specified in the `pDAName` parameter. If the `'DRVR'` resource name is found, `LaunchDeskAccessory` launches the corresponding desk accessory. If the desk accessory is already open, it is brought to the front.

Use the `pFileSpec` parameter to specify the file to search. Specify `NIL` as the value of `pFileSpec` if you want to search the current resource file and the resource files opened before it. Otherwise, use a pointer to an `FSSpec` record to specify the file.

In the `pDAName` parameter, specify the `'DRVR'` resource name of the desk accessory to launch. Specify `NIL` as the value of `pDAName` if you want to launch the first `'DRVR'` resource found in the file as returned by the Resource Manager. Because the `LaunchDeskAccessory` function opens the specified resource file for exclusive access, you cannot launch more than one desk accessory from the same resource file.

If the `'DRVR'` resource is in a resource file that is already open by the current process or if the driver is in the System file and the Option key is pressed, `LaunchDeskAccessory` launches the desk accessory in the application's heap. Otherwise, the desk accessory is given its own partition and launched in the system heap.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LaunchDeskAccessory` function are

Trap macro	Selector
`_OSDispatch`	$0036

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Resource not found

Terminating Processes

You can use the `ExitToShell` procedure to have your application terminate itself directly. In general, you need to call `ExitToShell` only if you want your application to terminate without reaching the end of its main routine.

ExitToShell

Call `ExitToShell` to terminate your application directly.

```
PROCEDURE ExitToShell;
```

DESCRIPTION

The `ExitToShell` procedure terminates the calling process. The Process Manager removes your application from the list of open processes and performs any other necessary cleanup operations. In particular, all memory in your application partition and any temporary memory still allocated to your application is released. If necessary, the Application Died Apple event is sent to the process that launched your application.

If your application was the foreground process at the time it called `ExitToShell`, its name is removed from the Application menu. The Process Manager selects a new foreground process, switches it into the foreground, and propagates the scrap to the new foreground application.

If your application was the last one running and the shell program is not the Finder, the Process Manager displays a dialog box that gives the user the choice of restarting the computer or shutting it down.

SPECIAL CONSIDERATIONS

Any trap patches installed by your application are removed immediately by `ExitToShell`. They will not affect any trap calls made by `ExitToShell` itself.

RESULT CODES

When `ExitToShell` exits, the system global variable `DSErrCode` holds its result code.

SEE ALSO

See “Terminating an Application” on page 2-11 for details on the parameters passed to the Application Died event.

Summary of the Process Manager

Pascal Summary

Constants

```
CONST
      {Gestalt selector and response bits}
      gestaltOSAttr              = 'os  ';   {O/S attributes selector}
      gestaltLaunchCanReturn     = 1;        {can return from launch}
      gestaltLaunchFullFileSpec  = 2;        {LaunchApplication is available}
      gestaltLaunchControl       = 3;        {Process Manager is available}

      {process identification constants}
      kNoProcess                 = 0;        {process doesn’t exist}
      kSystemProcess             = 1;        {process belongs to OS}
      kCurrentProcess            = 2;        {the current process}

      {launch control flags}
      launchContinue             = $4000;    {continue after launch}
      launchNoFileFlags          = $0800;    {ignore launchFileFlags}
      launchUseMinimum           = $0400;    {use minimum or greater size}
      launchDontSwitch           = $0200;    {launch app. in background}
      launchAllow24Bit           = $0100;    {reserved}
      launchInhibitDaemon        = $0080;    {don't launch background app.}

      {launch parameter block length and ID}
      extendedBlockLen           = sizeof(LaunchParamBlockRec) - 12;
      extendedBlock              = $4C43;    {extended block}

      {flags in processMode field}
      modeDeskAccessory          = $00020000;   {process is desk acc}
      modeMultiLaunch            = $00010000;   {from app file's flags}
      modeNeedSuspendResume      = $00004000;   {from 'SIZE' resource}
      modeCanBackground          = $00001000;   {from 'SIZE' resource}
      modeDoesActivateOnFGSwitch = $00000800;   {from 'SIZE' resource}
      modeOnlyBackground         = $00000400;   {from 'SIZE' resource}
      modeGetFrontClicks         = $00000200;   {from 'SIZE' resource}
```

```
   modeGetAppDiedMsg          = $00000100;   {from 'SIZE' resource}
   mode32BitCompatible        = $00000080;   {from 'SIZE' resource}
   modeHighLevelEventAware    = $00000040;   {from 'SIZE' resource}
   modeLocalAndRemoteHLEvents = $00000020;   {from 'SIZE' resource}
   modeStationeryAware        = $00000010;   {from 'SIZE' resource}
   modeUseTextEditServices    = $00000008;   {from 'SIZE' resource}
```


Data Types

Process Serial Number

```
TYPE
   ProcessSerialNumber     =
   RECORD
      highLongOfPSN:       LongInt;             {high-order 32 bits of psn}
      lowLongOfPSN:        LongInt;             {low-order 32 bits of psn}
   END;

   ProcessSerialNumberPtr  = ^ProcessSerialNumber;
```

Process Information Record

```
   ProcessInfoRec          =
   RECORD
      processInfoLength:   LongInt;             {length of record}
      processName:         StringPtr;           {name of process}
      processNumber:       ProcessSerialNumber; {psn of the process}
      processType:         LongInt;             {file type of app file}
      processSignature:    OSType;              {signature of app file}
      processMode:         LongInt;             {'SIZE' resource flags}
      processLocation:     Ptr;                 {address of partition}
      processSize:         LongInt;             {partition size}
      processFreeMem:      LongInt;             {free bytes in heap}
      processLauncher:     ProcessSerialNumber; {proc that launched this one}
      processLaunchDate:   LongInt;             {time when launched}
      processActiveTime:   LongInt;             {accumulated CPU time}
      processAppSpec:      FSSpecPtr;           {location of the file}
   END;

   ProcessInfoRecPtr       = ^ProcessInfoRec;
```

Application Parameters Record

```
AppParameters           =
RECORD
   theMsgEvent:         EventRecord;         {event (high-level)}
   eventRefCon:         LongInt;             {reference constant}
   messageLength:       LongInt;             {length of buffer}
   messageBuffer:       ARRAY [0..0] OF SignedByte;
END;

AppParametersPtr        =   ^AppParameters;
```

Launch Parameter Block

```
LaunchFlags             =   Integer;

LaunchParamBlockRec     =
RECORD
   reserved1:           LongInt;             {reserved}
   reserved2:           Integer;             {reserved}
   launchBlockID:       Integer;             {extended block}
   launchEPBLength:     LongInt;             {length of block}
   launchFileFlags:     Integer;             {app's Finder flags}
   launchControlFlags:  LaunchFlags;         {launch options}
   launchAppSpec:       FSSpecPtr;           {location of app's file}
   launchProcessSN:     ProcessSerialNumber; {returned psn}
   launchPreferredSize: LongInt;             {returned pref size}
   launchMinimumSize:   LongInt;             {returned min size}
   launchAvailableSize: LongInt;             {returned avail size}
   launchAppParameters: AppParametersPtr;    {high-level event}
END;

LaunchPBPtr             =  ^LaunchParamBlockRec;
```

Routines

Getting Process Information

```
FUNCTION GetCurrentProcess  (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetNextProcess     (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetProcessInformation
                            (PSN: ProcessSerialNumber;
                             VAR info: ProcessInfoRec): OSErr;
```

```
FUNCTION SameProcess        (PSN1: ProcessSerialNumber;
                             PSN2: ProcessSerialNumber;
                             VAR result: Boolean): OSErr;
FUNCTION GetFrontProcess    (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION SetFrontProcess    (PSN: ProcessSerialNumber): OSErr;
FUNCTION WakeUpProcess      (PSN: ProcessSerialNumber): OSErr;
```

Launching Applications and Desk Accessories

```
FUNCTION LaunchApplication  (LaunchParams: LaunchPBPtr): OSErr;
FUNCTION LaunchDeskAccessory(pFileSpec: FSSpecPtr; pDAName: StringPtr):
                             OSErr;
```

Terminating a Process

```
PROCEDURE ExitToShell;
```


C Summary

Constants

```
/*Gestalt selector and response bits*/
#define gestaltOSAttr               'os  '   /*O/S attributes selector*/
#define gestaltLaunchCanReturn      1        /*can return from launch*/
#define gestaltLaunchFullFileSpec   2        /*LaunchApplication available*/
#define gestaltLaunchControl        3        /*Process Manager is available*/

/*process identification constants*/
enum {
      kNoProcess                    0,       /*process doesn't exist*/
      kSystemProcess                1,       /*process belongs to OS*/
      kCurrentProcess               2        /*the current process*/
};

/*launch control flags*/
enum {
      launchContinue            = 0x4000,   /*continue after launch*/
      launchNoFileFlags         = 0x0800,   /*ignore launchFileFlags*/
      launchUseMinimum          = 0x0400,   /*use minimum or greater size*/
      launchDontSwitch          = 0x0200,   /*launch app. in background*/
```

```
      launchAllow24Bit          = 0x0100,   /*reserved*/
      launchInhibitDaemon       = 0x0080    /*don't launch background app.*/
};

/*launch parameter block length and ID*/
#define extendedBlockLen           (sizeof(LaunchParamBlockRec) - 12)
#define extendedBlock              ((unsigned short)'LC')

/*flags in processMode field*/
enum {
      modeDeskAccessory         = 0x00020000,  /*process is desk acc*/
      modeMultiLaunch           = 0x00010000,  /*from app file's flags*/
      modeNeedSuspendResume     = 0x00004000,  /*from 'SIZE' resource*/
      modeCanBackground         = 0x00001000,  /*from 'SIZE' resource*/
      modeDoesActivateOnFGSwitch = 0x00000800,  /*from 'SIZE' resource*/
      modeOnlyBackground        = 0x00000400,  /*from 'SIZE' resource*/
      modeGetFrontClicks        = 0x00000200,  /*from 'SIZE' resource*/
      modeGetAppDiedMsg         = 0x00000100,  /*from 'SIZE' resource*/
      mode32BitCompatible       = 0x00000080,  /*from 'SIZE' resource*/
      modeHighLevelEventAware   = 0x00000040,  /*from 'SIZE' resource*/
      modeLocalAndRemoteHLEvents = 0x00000020,  /*from 'SIZE' resource*/
      modeStationeryAware       = 0x00000010,  /*from 'SIZE' resource*/
      modeUseTextEditServices   = 0x00000008   /*from 'SIZE' resource*/
};
```

Data Types

Process Serial Number

```
struct ProcessSerialNumber {
      unsigned long        highLongOfPSN;       /*high-order 32 bits of psn*/
      unsigned long        lowLongOfPSN;        /*low-order 32 bits of psn*/
};

typedef struct ProcessSerialNumber ProcessSerialNumber;
typedef ProcessSerialNumber *ProcessSerialNumberPtr;
```

Process Information Record

```
struct ProcessInfoRec {
      unsigned long        processInfoLength;   /*length of record*/
      StringPtr            processName;         /*name of process*/
      ProcessSerialNumber  processNumber;       /*psn of the process*/
```

```
      unsigned long        processType;         /*file type of app file*/
      OSType               processSignature;    /*signature of app file*/
      unsigned long        processMode;         /*'SIZE' resource flags*/
      Ptr                  processLocation;     /*address of partition*/
      unsigned long        processSize;         /*partition size*/
      unsigned long        processFreeMem;      /*free bytes in heap*/
      ProcessSerialNumber  processLauncher;     /*proc that launched this */
                                                /* one*/
      unsigned long        processLaunchDate;   /*time when launched*/
      unsigned long        processActiveTime;   /*accumulated CPU time*/
      FSSpecPtr            processAppSpec;      /*location of the file*/
};

typedef struct ProcessInfoRec ProcessInfoRec;
typedef ProcessInfoRec *ProcessInfoRecPtr;
```

Application Parameters Record

```
struct AppParameters {
      EventRecord          theMsgEvent;         /*event (high-level)*/
      unsigned long        eventRefCon;         /*reference constant*/
      unsigned long        messageLength;       /*length of buffer*/
};

typedef struct AppParameters AppParameters;
typedef AppParameters *AppParametersPtr;
```

Launch Parameter Block

```
typedef unsigned short LaunchFlags;

struct LaunchParamBlockRec {
      unsigned long        reserved1;           /*reserved*/
      unsigned short       reserved2;           /*reserved*/
      unsigned short       launchBlockID;       /*extended block*/
      unsigned long        launchEPBLength;     /*length of block*/
      unsigned short       launchFileFlags;     /*app's Finder flags*/
      LaunchFlags          launchControlFlags;  /*launch options*/
      FSSpecPtr            launchAppSpec;       /*location of app's file*/
      ProcessSerialNumber  launchProcessSN;     /*returned psn*/
      unsigned long        launchPreferredSize; /*returned pref size*/
      unsigned long        launchMinimumSize;   /*returned min size*/
      unsigned long        launchAvailableSize; /*returned avail size*/
      AppParametersPtr     launchAppParameters; /*high-level event*/
```

```
};

typedef struct LaunchParamBlockRec LaunchParamBlockRec;
typedef LaunchParamBlockRec *LaunchPBPtr;
```

Routines

Getting Process Information

```
pascal OSErr GetCurrentProcess
                            (ProcessSerialNumber *PSN);
pascal OSErr GetNextProcess (ProcessSerialNumber *PSN);
pascal OSErr GetProcessInformation
                            (const ProcessSerialNumber *PSN,
                             ProcessInfoRecPtr info);
pascal OSErr SameProcess    (const ProcessSerialNumber *PSN1,
                             const ProcessSerialNumber *PSN2,
                             Boolean *result);
pascal OSErr GetFrontProcess
                            (ProcessSerialNumber *PSN);
pascal OSErr SetFrontProcess
                            (const ProcessSerialNumber *PSN);
pascal OSErr WakeUpProcess  (const ProcessSerialNumber *PSN);
```

Launching Applications and Desk Accessories

```
pascal OSErr LaunchApplication
                            (const LaunchParamBlockRec *LaunchParams);
pascal OSErr LaunchDeskAccessory
                            (const FSSpec *pFileSpec,
                             ConstStr255Param pDAName);
```

Terminating a Process

```
pascal void ExitToShell     (void);
```

Assembly-Language Summary

Data Structures

Process Serial Number

0	highLongOfPSN	long	high-order 32-bits of process serial number
4	lowLongOfPSN	long	low-order 32-bits of process serial number

Process Information Record

0	processInfoLength	long	length of this record
4	processName	long	name of process
8	processNumber	2 longs	process serial number of the process
16	processType	long	type of application file
20	processSignature	long	signature of application file
24	processMode	long	flags from 'SIZE' resource
28	processLocation	long	address of process partition
32	processSize	long	partition size (in bytes)
36	processFreeMem	long	amount of free memory in application heap
40	processLauncher	2 longs	process that launched this one
48	processLaunchDate	long	value of Ticks at time of launch
52	processActiveTime	long	total time spent using the CPU
56	processAppSpec	long	location of the file

Application Parameters Record

0	theMsgEvent	16 bytes	the high-level event record
16	eventRefCon	long	reference constant
20	messageLength	long	length of buffer
24	messageBuffer	byte	first byte of the message buffer

Launch Parameter Block

0	reserved1	long	reserved
4	reserved2	word	reserved
6	launchBlockID	word	specifies whether block is extended
8	launchEPBLength	long	length (in bytes) of rest of parameter block
12	launchFileFlags	word	the Finder flags for the application file
14	launchControlFlags	word	flags that specify launch options
16	launchAppSpec	long	address of FSSpec that specifies the application file to launch
20	launchProcessSN	2 longs	process serial number
28	launchPreferredSize	long	application's preferred partition size
32	launchMinimumSize	long	application's minimum partition size
36	launchAvailableSize	long	maximum partition size available
40	launchAppParameters	long	high-level event for launched application

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`LaunchApplication`	`_Launch`
`ExitToShell`	`_ExitToShell`

Trap Macros Requiring Routine Selectors

`_OSDispatch`

Selector	Routine
$0036	`LaunchDeskAccessory`
$0037	`GetCurrentProcess`
$0038	`GetNextProcess`
$0039	`GetFrontProcess`
$003A	`GetProcessInformation`
$003B	`SetFrontProcess`
$003C	`WakeUpProcess`
$003D	`SameProcess`

Result Codes

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid
`memFullErr`	–108	Not enough memory to allocate the partition size specified in the `'SIZE'` resource
`resNotFound`	–192	Resource not found
`procNotFound`	–600	No eligible process with specified process serial number
`memFragErr`	–601	Not enough room to launch application with special requirements
`appModeErr`	–602	Addressing mode is 32-bit, but application is not 32-bit clean
`appMemFullErr`	–605	Partition size specified in `'SIZE'` resource is not big enough for launch
`appIsDaemon`	–606	Application is background-only