GetSectionAddress

The `GetSectionAddress` macro returns the physical address of the specified section.

```
GetSectionAddress (theSectionPtr,theSectionName)
```

`theSectionPtr`	Returns a value, physical location of section.
`theSectionName`	The name of section to locate.

REGISTER USAGE

The `GetSectionAddress` macro does not alter the contents of any registers except `theSectionPtr`.

DESCRIPTION

The `GetSectionAddress` macro calculates the physical address of `theSectionName` and copies the address into the `theSectionPtr` register, which may be any `cau` register `r1-r18`.

GetSectionLabel

The `GetSectionLabel` macro returns a physical pointer to a label in the specified section.

```
GetSectionLabel (theSectionLabelPtr,theSectionLabel)
```

`theSectionLabelPtr`	Returns a pointer, physical location of section.
`theSectionLabel`	Label used within the section.

REGISTER USAGE

The `GetSectionLabel` macro does not alter the contents of any register except `theSectionLabelPtr`.

DESCRIPTION

The `GetSectionLabel` macro returns a physical pointer to a label designated by `theSectionLabel`. The pointer is returned in `theSectionLabelPtr`, which may be any `cau` register `r1-r18`.

GetSectionSize

The `GetSectionSize` macro returns the size of the specified section.

```
GetSectionSize (theSectionSize,theSectionName)
```

`theSectionSize`	The size of the section.
`theSectionName`	The section name.

REGISTER USAGE

The `GetSectionSize` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GetSectionSize` macro calculates the size of `theSectionName` and copies it into the `theSectionSize` register, which may be any `cau` register `r1-r18`.

PopSection

The `PopSection` macro caches the specified section off-chip.

```
PopSection (theSectionName)
```

`theSectionName`	The section name.

REGISTER USAGE

The `PopSection` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `PopSection` macro caches `theSectionName`. The actual caching operation performed depends upon the section's caching flags.

For static sections, `PopSection` caches the section data from its primary container to its secondary container. For non-static sections, `PopSection` caches the section data from the top of the demand cache stack to its primary container.

Note
The Save flag must be set (caching flags) for the specified section if data is to be moved. The memory space is automatically reclaimed by the DSP operating system. ◆

▲ **WARNING**
Sections must use `PopSection` in the reverse order that they use `PushSection`. ▲

PushSection

The `PushSection` macro loads the specified section on-chip.

```
PushSection (theSectionName)
```

`theSectionName` The section name.

REGISTER USAGE

The `PushSection` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `PushSection` macro caches `theSectionName`. The actual caching operation performed depends upon the section's caching flags.

For static sections `PushSection` caches the section data from its secondary container to its primary container. For non-static sections, `PushSection` caches the section data from its primary container to the top of a demand cache stack.

Note
You must set the Load flag (caching flags) for the specified section if data is to be moved. The Clear flag must be set if the section is to be cleared. Either the Bank A or Bank B flag should also be set. If no Bank flag or the Don't Care flag is selected the DSP operating system will use Bank A. ◆

Module Manipulation Macro

The `SetSkipCount` macro helps you program DSP modules.

SetSkipCount

The `SetSkipCount` macro sets the skip count (number of modules to be jumped over).

```
SetSkipCount (theSkipCount)
```

`theSkipCount` The number of modules to skip over.

REGISTER USAGE

The `SetSkipCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SetSkipCount` macro sets the skip count for the currently executing module. The current module continues its execution. When the module finishes its execution, the new skip count takes effect.

The `theSkipCount` parameter is a 32-bit constant or any `cau` register in the range `r1` through `r17`.

Task Manipulation Macros

The macros described in this section help you work with tasks.

GetNumRealTimeFrames

The `GetNumRealTimeFrames` macro returns the number of real-time frames that have been executed.

```
GetNumRealTimeFrames (numFrames)
```

`numFrames` The number of frames executed.

REGISTER USAGE

The `GetNumRealTimeFrames` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GetNumRealTimeFrames` macro is used to get the number of real-time frames that have been executed since the DSP was started or reset.

SetTaskInactive

The `SetTaskInactive` macro turns off the task associated with the section that is using it.

```
SetTaskInactive ()
```

REGISTER USAGE

The `SetTaskInactive` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SetTaskInactive` macro sets the owner task for the currently executing module inactive. Setting the task inactive does not take effect until the next frame. The task's modules complete their execution for the current frame.

FIFO Manipulation Macros

The macros described in this section help you work with FIFO buffers.

▲ **WARNING**
Although FIFO manipulations deal with byte counts, all operations must be done in longword (4 bytes) increments only. Use of the FIFO calls with non-longword counts will cause unpredictable results. ▲

FIFOGetReadCount

The `FIFOGetReadCount` macro returns the available number of data bytes in the FIFO.

```
FIFOGetReadCount (theFIFOName)
```

`theFIFOName` The FIFO name.

REGISTER USAGE

The `FIFOGetReadCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOGetReadCount` macro returns, in `r2`, the current number of bytes available in the FIFO that can be read. A value of 0 indicates an empty FIFO.

FIFORead

The `FIFORead` macro copies FIFO data into the specified section.

```
FIFORead (theSectionName)
```

`theSectionName` The section name.

REGISTER USAGE

The `FIFORead` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFORead` macro takes one argument, the section name of an AIAO FIFO section. The AIAO FIFO section must be located within the same DSP module as the current section.

The `FIFORead` macro copies data to the AIAO FIFO section from the FIFO that's connected to it.

The size of AIAO is used as the number of bytes to read from the FIFO. If the FIFO empties during the read, only the actual number available will be read. The remaining bytes in the section are cleared to 0.

In the event that an underrun occurs (the FIFO does not contain enough data to fill the AIAO), a `kdspFIFOUnderrunMessage` message is sent to the FIFO's message handler if the FIFO's `kdspEnableOverUnderMessage` flag is set. Also, if the FIFO's `kdspOverUnderTaskInactive` flag is set, the owner task of the currently executing module is set inactive.

Note
Reads and writes to the buffers must occur on longword boundaries. ◆

FIFOReadN

The `FIFOReadN` macro copies the requested number of bytes of FIFO data into the specified section.

```
FIFOReadN (theFIFOName, theCount)
```

`theFIFOName` The FIFO name.
`theCount` The number of bytes to copy.

REGISTER USAGE

The `FIFOReadN` macro destroys the contents of `cau` registers `r1-r4, r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOReadN` macro reads the specified number of bytes in `theCount` from the named FIFO to the section. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO empties during the read, only the actual number of bytes available are read. The remaining bytes in the section are cleared to 0.

FIFOReadNBuffer

The `FIFOReadNBuffer` macro copies the requested number of bytes of FIFO data into the specified section.

```
FIFOReadNBuffer (theFIFOName, theCount, theBufferPtr)
```

`theFIFOName`	The FIFO name.
`theCount`	The number of bytes to copy.
`theBufferPtr`	The section data is being copied to.

REGISTER USAGE

The `FIFOReadNBuffer` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOReadNBuffer` macro reads the specified number of bytes in `theCount` from the named FIFO to the section pointed to by `theBufferPtr`. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO empties during the read process, only the actual number of bytes available will be read. The remaining bytes in the section are cleared to 0.

FIFOGetWriteCount

The `FIFOGetWriteCount` macro returns the number of empty bytes available in the FIFO.

```
FIFOGetWriteCount (theFIFOName)
```

`theFIFOName` The FIFO name.

REGISTER USAGE

The `FIFOGetWriteCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOGetWriteCount` macro returns in `r2` the current number of bytes available in the FIFO that can be written—in other words, how much empty space is available. A value of 0 indicates a full FIFO.

FIFOWrite

The `FIFOWrite` macro copies section data into the specified FIFO.

```
FIFOWrite (theSectionName)
```

`theSectionName` The section name.

REGISTER USAGE

The `FIFOWrite` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOWrite` macro writes from the AIAO section to the named FIFO. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

The size of the AIAO section is used as the number of bytes to write to the FIFO.

In the event that an overrun occurs (the FIFO does not contain enough space to hold the AIAO's data), a `kdspFIFOOverrunMessage` message is sent to the FIFO's message handler if the FIFO's `kdspEnableOverUnderMessage` flag is set. Also, if the FIFO's `kdspOverUnderTaskInactive` flag is set, the owner task of the currently executing module is set inactive.

Note
Reads and writes to the FIFO and the buffer may occur on longword boundaries only. ◆

FIFOWriteN

The `FIFOWriteN` macro copies the specified number of bytes of section data into the specified FIFO.

```
FIFOWriteN (theFIFOName, theCount)
```

`theFIFOName`	The FIFO name.
`theCount`	The number of bytes to copy.

REGISTER USAGE

The `FIFOWriteN` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOWriteN` macro writes the specified number of bytes in `theCount` to the named FIFO from the section. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

FIFOWriteNBuffer

The `FIFOWriteNBuffer` macro copies the specified number of bytes of section data into the specified FIFO.

```
FIFOWriteNBuffer (theFIFOName, theCount, theBufferPtr)
```

`theFIFOName`	The FIFO name.
`theCount`	The number of bytes to copy.
`theBufferPtr`	The section data is being copied from.

REGISTER USAGE

The `FIFOWriteNBuffer` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOWriteNBuffer` macro writes the specified number of bytes in `theCount` to the named FIFO from the buffer pointed to by `theBufferPtr`. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

Note
Reads and writes to the FIFO and the buffer are on longword boundaries only. ◆

GPB Manipulation Macros

The macros described in this section help you manage the GPB for a module. GPB is discussed in "Guaranteed Processing Bandwidth," in Chapter 3.

GPBElapsedCycles

The `GPBElapsedCycles` macro returns the number of DSP cycles used by this module up to the point it is called.

```
GPBElapsedCycles (theCycles)
```

`theCycles` Elapsed cycles since start or reset.

REGISTER USAGE

The `GPBElapsedCycles` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GPBElapsedCycles` macro returns the number of DSP instruction cycles that have elapsed since this module started execution. By comparing this value with the expected value returned from the `GPBExpectedCycles()` macro, a dumb lumpy algorithm can determine if it should cease processing. Dumb lumpy algorithms are discussed in "Smooth and Lumpy Algorithms," in Chapter 3.

GPBExpectedCycles

The `GPBExpectedCycles` macro returns the computed number of DSP cycles this module is expected to need based on the supplied GPB estimate.

```
GPBExpectedCycles (theCycles)
```

`theCycles` Expected cycles for this module.

REGISTER USAGE

The `GPBExpectedCycles` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GPBExpectedCycles` macro returns the expected number of DSP instruction cycles to complete this module. This is used in conjunction with the `GPBElapsedCycles` macro (page 235) to control the execution of a dumb lumpy algorithm.

GPBSetUseActual

The `GPBSetUseActual` macro tells the DSP operating system to use the actual GPB required instead of the estimated value.

```
GPBSetUseActual ()
```

REGISTER USAGE

The `GPBSetUseActual` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GPBSetUseActual` macro sets the `UseActualGPB` flag for the module. This flag is set immediately, so this routine should not be called until the module is in its worst-case GPB usage.

Semaphore Manipulation Macros

The macros described in this section help you work with semaphores.

SemaphoreClear

The `SemaphoreClear` macro clears the specified semaphore in a locked environment.

```
SemaphoreClear (theSemaphorePtr, theMask, theOldSemaphoreValue)
```

`theSemaphorePtr`	Pointer to the semaphore.
`theMask`	Mask of new semaphore value.
`theOldSemaphoreValue`	Returns the value of the old semaphore.

REGISTER USAGE

The `SemaphoreClear` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SemaphoreClear` macro locks the system bus and performs the following operation:

```
[lock the bus]
*theSemaphorePtr = ((theOldSemaphoreValue = *theSemaphorePtr) &
                    ~theMask)
[unlock the bus]
```

The value of `theSemaphorePtr` must be a `cau` register in the range `r1` through `r17` containing a physical pointer.

The value of `theMask` may be any register in the range `r1` through `r17`, or a constant.

The `SemaphoreClear` macro performs `dolock` on the bus to prevent host access and then reads the semaphore location. The old semaphore value is `AND`-combined with `NOT` of the mask and this new value is written back to the semaphore location.

RETURN VALUE

The value of `theOldSemaphoreValue` is the value of the semaphore before it was `AND`-combined with the one's-complement of the value of `theMask`.

SemaphoreSet

The `SemaphoreSet` macro sets the specified semaphore in a locked environment.

```
SemaphoreSet (theSemaphorePtr, theMask, theOldSemaphoreValue)
```

`theSemaphorePtr`	Pointer to the semaphore.
`theMask`	Mask of new semaphore value.
`theOldSemaphoreValue`	Returns the value of the old semaphore.

REGISTER USAGE

The `SemaphoreSet` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SemaphoreSet` macro locks the system bus and performs the following operation:

```
[lock the bus]
*theSemaphorePtr = ((theOldSemaphoreValue = *theSemaphorePtr) |
                      theMask)
[unlock the bus]
```

The value of `theSemaphorePtr` must be a `cau` register `r1-r17` containing a physical pointer.

The `SemaphoreSet` macro performs `DoLock` on the bus to prevent host access and then reads the semaphore location. The old semaphore value is `OR`-combined with the mask and this new value is written back to the semaphore location.

RETURN VALUE

The value of `theOldSemaphoreValue` is the value of the semaphore before it was `OR`-combined with `theMask`.

Message Manipulation Macro

The `SendMessageToHost` macro helps you work with DSP messages.

SendMessageToHost

The `SendMessageToHost` macro sends a message from the module to the host using the interrupt handler.

```
SendMessageToHost (theDSPMessagePtr)
```

`theDSPMessagePtr` Pointer to the message vector.

REGISTER USAGE

The `SendMessageToHost` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SendMessageToHost` macro calls the `msVector` (interrupt handler) in the Real Time Manager structure that then passes the message to the interrupt handler. When used by a module to send a message to the client application the `msData [0]` through `msData[2]` fields are not defined when using this macro.

The value of `theDSPMessagePtr` must be a `cau` register `r1-r17` containing a physical pointer to a DSP message.

Note
The `msVector` field of the message must be initialized to a valid interrupt handler. Fields `msData [0]` through `msData[2]` can be used by the programmer as needed. ◆

When the Real Time Manager uses this routine to send a message to the client application `msData [0]` contains `theErrorMessage` constant. The message is sent to the interrupt vector of the owner task for the currently executing module. The owner task is then set inactive.

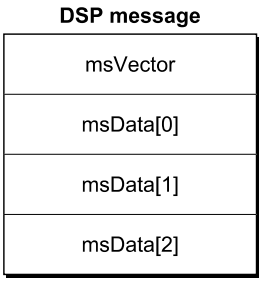
The `theErrorMessage` constant is a DSP message constant or a register containing a DSP message constant. The Apple-defined DSP message constants are defined in the next section.

When the host interrupt vector for the task is called, a complete `DSPMessage` structure is passed on the stack containing the following information:

The owner task's interrupt vector	→ `msVector`
`theErrorMessage`	→ `msData[0]`
The task's reference number	→ `msData[1]`
The current module's reference number	→ `msData[2]`

The DSP message structure is diagrammed in Figure 5-3.

Figure 5-3 DSP message structure



The corresponding routine in the Macintosh API is the `MessageActionProc` routine.

Summary of the DSP Operating System

Constants

```
================================================================
//                DSP MODULE/SECTION DEFINITIONS
================================================================

----------------------------------------------------------------
// FIFOFlags
----------------------------------------------------------------
#define  kdspFIFOMaskAllMessages          0x00000000  // disable all
            //messages (p) priority of FIFO messages in descending order

#define  kdspFIFOEnableOverUnderMessage   0x00000001  // (3) enable
            // message when FIFO transfer causes an overrun or underrun
```

```
#define  kdspFIFOEnableFullEmptyMessage   0x00000002  // (2) enable
            // message when FIFO goes full or empty

#define  kdspFIFOEnableHighLowMessage     0x00000004  // (1) enable
            // message when FIFO goes at least half full or half empty

#define  kdspFIFOEnableLinkMessage        0x00000008  // (4) enable
            // message when FIFO's link is traversed

#define  kdspFIFOOverUnderTaskInactive    0x00000010  // if task
            // accessing FIFO causes either FIFO overrun or underrun then
            // set task inactive

#define  kdspFIFOFullEmptyTaskInactive    0x00000020  // if task
            // accessing FIFO causes either FIFO full or FIFO empty then set
            // task inactive

//-------------------------------------------------------------------
// ModuleFlags
//-------------------------------------------------------------------
#define  kdspAutoCache           0x00000000  // select auto cache model
#define  kdspDemandCache         0x00000001  // select demand cache model
#define  kdspOnChipSectionTable  0x00000004  // put section table on-chip
#define  kdspOnChipStack         0x00000020  // a stack of the specified
                                             // size will be created on-chip
#define  kdspOffChipStack        0x00000040  // a stack of the specified
                                             // size will be created off-chip

//-------------------------------------------------------------------
// GPBFlags (see DSPConstantsPrivate.h for the complete list of flags)
//-------------------------------------------------------------------
#define  kdspLumpyModule         0x00000000  // use bnEstimate
#define  kdspSmoothModule        0x00000001  // see DSPConstantsPrivate.h

//-------------------------------------------------------------------
// SectionFlags
//-------------------------------------------------------------------
// Costs the DSP one instruction to use the following flags:
#define  kdspLeaveSection        0x00000000  // do not load or save this
                                             // section
#define  kdspLoadSection         0x00000001  // load this section
#define  kdspSaveSection         0x00000002  // save this section
#define  kdspClearSection        0x00000004  // fill this section with zeroes
```

```
#define  kdspSaveOnContextSwitch 0x00000008  // save this section on context
                                              // switch

#define  kdspExternal            0x00000000  // never loaded on-chip
#define  kdspBankA               0x00000020  // load in Bank A if possible
#define  kdspBankB               0x00000040  // load in Bank B if possible
#define  kdspAnyBank             (kdspBankA | kdspBankB) // load anywhere
#define  kdspStaticSection       0x00000080  // section statically allocated
                                              // before runtime
#define  kdspFIFOSection         0x00000100  // section is a FIFO buffer
#define  kdspReservedSectionFlag0200   0x00000200  // reserved
#define  kdspLoadFIFOSection     0x00000400  // when loading convert from a
                                              // FIFO
#define  kdspSaveFIFOSection     0x00000800  // when saving convert to a FIFO

#define  kdspHIHOSection         0x00001000  // this is a HIHO section
#define  kdspReservedForToggleSectionTbl  0x00002000  // this flag holds the
                                              // kdspToggleSectionTable flag
                                              // from the module's flag

#define  kdspLoadHIHOSection     0x00004000  // when loading convert from a
                                              // HIHO
#define  kdspSaveHIHOSection     0x00008000  // when saving convert to a HIHO

// Costs the DSP two instructions to use the following flags:
#define  kdspNotIOBufferSection  0x00010000  // all cases other than below
#define  kdspInputBuffer         0x00020000  // section is an input buffer
#define  kdspOutputBuffer        0x00040000  // section is an output buffer
#define  kdspITBSection          0x00080000  // section is an intertask
                                              // buffer
#define  kdspScalableSection     0x00100000  // section size can be scaled
#define  kdspSectionAllocated    0x00200000  // reserved for use by the DSP
                                              // Manager
#define  kdspDSPUseOnly          0x00400000  // only DSP should modify this
                                              // memory

//---------------------------------------------------------------
// SectionDataTypes
//---------------------------------------------------------------
#define  kdspNonData          0x00000000  // data in section is beyond
                                          // description
#define  kdsp3200Float        0x00000001  // 3200 float
#define  kdspIEEEFloat        0x00000002  // IEEE float format
```

```
#define  kdspInt32            0x00000003  // 32bit integer
#define  kdspInt1616          0x00000004  // 16bit integer packed
#define  kdspInt8888          0x00000005  // 8bit integer packed
#define  kdspmuLaw            0x00000006  // muLaw format
#define  kdspALaw             0x00000007  // Alaw format
#define  kdspAppSpecificData  0x0000FFFF  // application-specific

//=====================================================================
//                DSP CLIENT DEFINITIONS
//=====================================================================
//---------------------------------------------------------------------
// constants used by a client to specify where to insert a task
//---------------------------------------------------------------------
                                          // insert at list:
#define  kdspHeadInsert          0x00000004     // head
#define  kdspTailInsert          0x00000008     // tail
#define  kdspBeforeInsert        0x00000010     // before reference link
#define  kdspAfterInsert         0x00000020     // after reference link
#define  kdspAnyPositionInsert   kdspHeadInsert // anywhere

//---------------------------------------------------------------------
// constants for messages received by client tasks
//---------------------------------------------------------------------
#define  kdspBIOPinChangedState  0x62696f70  //'biop' (bio pin has changed
                                             // state)

// constants used for FIFO:
#define  kdspFIFOMessage         0x66000000  // 'f   ' (messages)
#define  kdspFIFOLinkMessage     0x666c6e6b  // 'lnk ' (link was traversed)
#define  kdspFIFOOverrunMessage  0x666f7672  // 'fovr' (buffer filled before
                                             // FIFO write completed)
#define  kdspFIFOUnderrunMessage 0x66756e64  //'fund' (buffer emptied before
                                             // FIFO read completed)

// constants used for FIFO buffer:
#define  kdspFIFOFullMessage     0x6666756c  // 'fful' (exactly full)
#define  kdspFIFOEmptyMessage    0x66656d70  // 'femp' (exactly empty)
#define  kdspFIFOHighMessage     0x66686967  // 'fhig' (at least half full
                                             //  but not exactly full)
#define  kdspFIFOLowMessage      0x666c6f77  // 'flow' (at least half empty
                                             //  but not exactly empty)
#define  kdspFIFOPrimeMessage    0x66707269  // 'fpri' (application-specific)
```

```
// constants used for dsp exception messages
#define  kdspExceptionMessage             0x78000000  // 'x   '
#define  kdspExceptionReset               0x78727374  // 'xrst'
#define  kdspExceptionBusError            0x78627573  // 'xbus'
#define  kdspExceptionIllegalOpcode       0x78696c6c  // 'xill'
#define  kdspExceptionReservedOne         0x78727631  // 'xrv1'
#define  kdspExceptionAddressError        0x78616472  // 'xadr'
#define  kdspExceptionDAUOverUnderflow    0x78646175  // 'xdau'
#define  kdspExceptionNotANumber          0x786e616e  // 'xnan'
#define  kdspExceptionReservedTwo         0x78727632  // 'xrv2'
#define  kdspExceptionExternalIntZero     0x78657830  // 'xex0'
#define  kdspExceptionTimer               0x7874696d  // 'xtim'
#define  kdspExceptionReservedThree       0x78727633  // 'xrv3'
#define  kdspExceptionSIOInputBufFull     0x78736962  // 'xsib'
#define  kdspExceptionSIOOutputBufEmpty   0x78736f62  // 'xsob'
#define  kdspExceptionSIODMAInputFrame    0x78736966  // 'xsif'
#define  kdspExceptionSIODMAOutputFrame   0x78736f66  // 'xsof'
#define  kdspExceptionExternalIntOne      0x78657831  // 'xex1'
#define  kdspExceptionRuntimeError        0x78657272  // 'xerr'

#define  kdspGPBMessage                   0x67000000  // 'g   ' (prefix used
                                                      // for GPB messages)
#define  kdspGPBTaskActive                0x67616374  // 'gact' (task is
                                                      // active)
#define  kdspGPBTaskInactive              0x67696e61  // 'gina' (task is
                                                      // inactive)
#define  kdspGPBFrameOverrun              0x676f7672  // 'govr' (task was
                                                      // involved in a frame
                                                      // overrun and is now
                                                      // inactive)

#define  kdspGPBFrameSkip                 0x67736b70  // 'gskp' (task has
                                                      // skipped one or more
                                                      // frames due to a
                                                      // frame overrun)

//-------------------------------------------------------------
// read/write permission constants for clients
//-------------------------------------------------------------
#define  kdspWritePermission     0x0001
#define  kdspReadPermission      0x0002
#define  kdspReadWritePermission (kdspWritePermission | kdspReadPermission)
```

```
//----------------------------------------------------------------
// constants for indexed devices
//----------------------------------------------------------------
// CPU processor types
#define  kdsp3210          '3210'
#define  kdsp32C           '32C '

//----------------------------------------------------------------
// constants for DSP API functions
//----------------------------------------------------------------
#define  kevtMessageToHost       (17)
#define  kevtCacheSection        (22)
#define  kevtCopyFIFO            (23)
#define  kevtGetSectionSize      (43)
#define  kevtGPBSetUseActual     (44)
#define  kevtGPBExpectedCycles   (45)
#define  kevtGPBElapsedCycles    (46)
#define  kevtSemaphoreSet        (47)
#define  kevtSemaphoreClear      (48)
#define  kevtSetSkipCount        (50)
#define  kevtSetTaskInactive     (51)
#define  kevtBlockMove           (53)
#define  kevtNumreal-timeFrames  (113)

//================================================================
// constants for errors returned by Macintosh DSP API
//================================================================

//----------------------------------------------------------------
// misc errors
// the next available error code number is -733
// if you add an error, also add it to the DSPErrorStrings.r file
//----------------------------------------------------------------
#define  kdspUnimplemented    (-692)   // feature is not implemented
#define  kdspParamErr         (-704)   // bad parameter

//----------------------------------------------------------------
// DSPFIFO errors
//----------------------------------------------------------------
#define  kdspNotAFIFOSection     (-700)   // not a FIFO section
#define  kdspNoMessageInterrupt  (-702)   // no message passing without a
                                          // vector
```

```
#define  kdspFIFOInUseByDSP      (-719)   // this FIFO is currently being
                                          // accessed by the DSP
#define  kdspTaskMustBeInActive  (-720)   // can only dipose of inactive
                                          // structures

#define  kdspNotFirstFIFO        (-721)   // the FIFO must be the first FIFO
                                          // in the link to wrap it

//-------------------------------------------------------------------
// DSPList errors
//-------------------------------------------------------------------
#define  kdspPositionIllegalErr     (-666)   // illegal DSPPosition type
#define  kdspPositionBusyErr        (-667)   // DSPPosition already occupied
#define  kdspInvalidReferenceErr    (-668)   // illegal insertion request
#define  kdspNonExistantReferenceErr(-669)   // reference element does not
                                             // exist
#define  kdspNonExistantElementErr  (-670)   // deletion element not found

//-------------------------------------------------------------------
// DSPMemory errors
//-------------------------------------------------------------------
#define  kdspMemFullErr          (-671)   // heap full,allocation failed
#define  kdspAddressNotInZone    (-672)   // address is not in a zone
#define  kdspNilAddress          (-683)   // trying to dispose of nil
#define  kdspContainingNilAddress(-684)   // trying to dispose of (nil, nil)
#define  kdspInvalidZoneSize     (-685)   // heap size must be factor of four
#define  kdspInvalidZoneBase     (-686)   // heap base must be longword
                                          // aligned

//-------------------------------------------------------------------
// DSPClient errors
//-------------------------------------------------------------------
#define  kdspDeviceNotFound         (-673)   // no device matching given name
#define  kdspInvalidIndexErr        (-674)   // no device (or whatever)
                                             // matching index given
#define  kdspDeviceHasActiveClients (-675)   // can't sign out device with
                                             // clients
#define  kdspInvalidPermission      (-688)   // invalid permission for
                                             // operation
#define  kdspWritePermissionDenied  (-689)   // client already exists with
                                             // write permission
#define  kdspClientNameInvalid      (-690)   // client name must be [1……31]
                                             // bytes
```

```
#define  kdspInvalidOptionSelector  (-691)   // options selector not
                                             // recognized
#define  kdspInvalidIODeviceType    (-707)   // invalid io device type,
                                             // index out of range
#define  kdspInvalidClientICON      (-717)   // an invalid ICON was passed
#define  kdspDeviceCantBeSlave      (-728)   // specified cpu device cannot
                                             // be a slave

//--------------------------------------------------------------------
// resource loader errors
//--------------------------------------------------------------------
#define  kdspModuleNotFound               (-676)   // module does not exist
#define  kdspModuleUncompatibleRate       (-677)   // incompatible frame or
                                                   // sample rate
#define  kdspUnknownDSPFResourceVersion   (-679)   // DSPF resource not
                                                   // recognized
#define  kdspUnknownDSPSectionTag         (-680)   // DSPF resource not
                                                   // recognized
#define  kdspZeroGPB                      (-714)   // module has GPB set to
                                                   // zero
#define  kdspTwoStacks                    (-731)   // cannot have both an
                                                   // on-chip and an
                                                   // off-chip stack

//--------------------------------------------------------------------
// DSPStorage errors
//--------------------------------------------------------------------
#define  kdspStorageNotFound        (-695)   // the amount and location do
                                             // not exist
#define  kdspNotEnoughOnChipMemory  (-696)   // not enough on-chip memory to
                                             // allocate

//--------------------------------------------------------------------
// DSPAllocation errors
//--------------------------------------------------------------------
#define  kdspCouldNotAllocate          (-678)   // could not allocate the
                                                // module
#define  kdspMoreThanOneModule         (-687)   // can allocate only one
                                                // module for now
#define  kdspSectionAlreadyConnected   (-693)   // one of the sections has
                                                // already been
                                                // connected (i.e. FIFO
                                                // sections)
```

```
#define  kdspSectionsDoNotMatch        (-694)   // the sections which are
                                                 // being connected either do
                                                 // not have the same size
                                                 // or the same type or are
                                                 // both input or both output
#define  kdspSectionsNotInSameModule   (-706)   // the sections that are
                                                 // being connected are not
                                                 // in the same module
#define  kdspSectionNotFound           (-697)   // could not find the
                                                 // specified section
#define  kdspBothFIFOsAllocated        (-698)   // both FIFO sections have
                                                 // already been attached to
                                                 // FIFOs
#define  kdspHadToUseOffChipMemory     (-699)   // section which was
                                                 // supposed to be on-chip
                                                 // was set up off-chip; the
                                                 // module will still run,
                                                 // but not as quickly
#define  kdspAlreadyAllocated          (-701)   // you cannot make a new ITB
                                                 // or connect sections if
                                                 // task has already been
                                                 // allocated
#define  kdspTooManyITBs               (-703)   // you cannot have more than
                                                 // MAX_MAP_SECTIONS ITBs
#define  kdspInvalidModuleAddress      (-712)   // passed in a nil module
                                                 // address
#define  kdspAIAOMustLoadOrSave        (-715)   // when connecting a FIFO to
                                                 // an AIAO,the AIAO must
                                                 // move data or the
                                                 // connection will not work
#define  kdspFIFOsNotConnected         (-716)   // you cannot insert a task
                                                 // if all the FIFOs are not
                                                 // connected to other FIFOs
#define  kdspNotAllocated              (-718)   // you must insert the task
                                                 // before you can call
                                                 // DSPGetSectionData
#define  kdspTaskNotInstalled          (-732)   // you cannot get the
                                                 // available on-chip
                                                 // memory until after the
                                                 // task is installed
```

```
//---------------------------------------------------------------------
// DSPTask errors
//---------------------------------------------------------------------
#define  kdspTaskRefNumAlreadyAllocated(-681)   // trying to reuse used
                                                // DSPTaskRefNum
#define  kdspNilMessageActionProc      (-705)   // passed in nil where
                                                // MessageActionProc
                                                // required

#define  kdspInvalidCPUDevicePtr       (-708)   // passed in nil for
                                                // DSPCPUDeviceParamBlkPtr
#define  kdspInvalidTaskRefNumPtr      (-709)   // passed in nil for the
                                                // DSPTaskRefNumPtr
#define  kdspInvalidTaskAddress        (-710)   // passed in nil for the
                                                // DSPTaskAddressPtr
#define  kdspInvalidTaskRefNum         (-711)   // passed in nil for the
                                                // DSPTaskRefNum
#define  kdspInvalidTaskName           (-713)   // length of name must
                                                // be > 0 and < 31

#define  kdspNoMasterSlaveRelationship (-722)   // tasks to be synchronized
                                                // must be on one
                                                // DSP or on DSPs that have
                                                // master-slave relationship
#define  kdspAllTasksMustBeRealTime    (-723)   // tasks to be synchronized
                                                // that are on different
                                                // DSPs must all be in
                                                // the real-time task list
#define  kdspNotEnoughTime             (-724)   // didn't have enough time
                                                // to successfully
                                                // synchronize all the tasks
#define  kdspChangingState             (-725)   // task is in the process of
                                                // going (in)active
#define  kdspAlreadyActive             (-726)   // task is already active
#define  kdspAlreadyInactive           (-727)   // task is already inactive

//---------------------------------------------------------------------
// DSPGPB errors
//---------------------------------------------------------------------
#define  kdspNotEnoughGPB  (-682)   // not enough real time for allocation
```

```
//---------------------------------------------------------------------
// DSP address fixup errors
//---------------------------------------------------------------------
#define  kdspOnChipPatchup       (-729)   // auto-init using address already
                                          // on-chip
#define  kdspBadRelocationType   (-730)   // internal assert - unrecognized
                                          // relocation type from linker

//=====================================================================
//                DSP REGISTER ASSIGNMENTS
//=====================================================================
//
// DSP3210 Register Model
//
// REGISTER    USAGE       DESCRIPTION
// r1-r4       scratch     The contents of these registers are not
// r15-r17                 saved or restored. The contents of these
// a0-a1                   registers may be destroyed by DSP API calls.
//
// r5-r14      protected   The contents of these registers must
// a2-a3                   always be saved and restored when they
//                         are used by the programmer.
//
//                         The contents of these registers are always
//                         saved before and restored after they are
//                         used by the DSP API calls.
//
// r18         return      The DSP operating system always calls the first
//                         instruction in the entry section of
//                         each module. r18 contains the return
//                         vector to get back to the DSP operating system
//                         when the module has finished executing.
//
//                         Before jumping to the return vector, all
//                         protected registers must be restored to the
//                         same values they contained upon initial
//                         entry to the module.
//
// r19         reserved    This register is reserved by Apple. Do not alter
//                         its contents.
//
// r20         reserved    This register is reserved by Apple. Do not alter
//                         its contents.
//
```

```
// r21         stack       This register is the common stack pointer
//                         register shared by the programmer and the
//                         DSP operating system. r21 always points to the
//                         next available stack location. Therefore, r21
//                         is pre-decremented for pops and
//                         post-incremented for pushes.

// r22         reserved    This register is reserved by Apple. Do not alter
//                         its contents.
//
//=============================================================

#define  TMP   r17   // temporary register
#define  RV    r18   // return vector
#define  MXPB  r19   // reserved
#define  ERRT  r20   // reserved
#define  SP    r21   // stack pointer
#define  EVTP  r22   // reserved

//=============================================================
//                         DSP API CONSTANTS
//=============================================================

#define kLongWordSize      (4)
#define kStr31Size         (32)

// DSPMessage
#define  msVector          (0)
#define  msData            (msVector + kLongWordSize)
#define  DSPMessageSize    (msData + 3*kLongWordSize)

#define  kEVTPad           (512)
#define  Find(theSelector) EVTP + kEVTPad + theSelector*kLongWordSize

#define  OSCall(theSelector)\
      RV  = (long)  Find(theSelector);\
      RV  = (long) *RV;\
      nop;\
      call RV (RV);\
      nop
```

```
// Selectors/Options for DSP API macros
#define  kdspSelectPush             0x00000000  // selector for direction
#define  kdspSelectPop              0x00000001  // selector for direction
#define  kdspOptionSpecifyBuffer    0x00000002  // option for specifying
                                                // buffer
#define  kdspOptionSpecifyCount     0x00000004  // option for specifying
                                                // count
#define  kdspOptionNoCopyJustCount  0x00000008  // option for just counting
```

Routines

```
//================================================================
//                DSP PROGRAM MACROS
//================================================================

#define  NewModule (Name, GPBFlags, ModuleFlags, EntryName)\
   @lowmod (Name, GPBFlags, ModuleFlags, EntryName)

#define  NewSection (Name, SectionFlags, SectionDataType, ModuleName)\
   @lowseg (Name, SectionFlags, 0, SectionDataType, ModuleName)
#define  AppendSection (Name)\
   .rsect"Name", TEXT

#define  NewInputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
   ModuleName)\ NewSection (Name,\
                kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspInputBuffer | kdspFIFOSection,\
                SectionDataType,\
                ModuleName)\
                BufferSize * long 0

#define NewOutputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
   ModuleName)\NewSection (Name,\
                kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                |kdspOutputBuffer | kdspFIFOSection,\
                SectionDataType,\
                ModuleName)\
                BufferSize * long 0
```

```
#define NewInputFIFOAndScalableBufferSection (Name, BufferScale,
   SectionDataType, ModuleName)\NewSection (Name,\
                kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspInputBuffer | kdspFIFOSection
                | kdspScalableSection,\
                SectionDataType,\
                ModuleName)\
                BufferScale * long 0

#define NewOutputFIFOAndScalableBufferSection (Name, BufferScale,
   SectionDataType, ModuleName)\NewSection (Name,\
                kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspOutputBuffer | kdspFIFOSection
                | kdspScalableSection,\
                SectionDataType,\
                ModuleName)\
                BufferScale * long 0

#define NewInputAIAOSection (Name, AIAOSize, SectionDataType, ModuleName)\
   NewSection (Name,\
                kdspLoadSection | kdspBankB
                | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspInputBuffer | kdspStaticSection,\
                SectionDataType,\
                ModuleName)\
                AIAOSize * long 0

#define NewOutputPRBSection (Name, AIAOSize, SectionDataType, ModuleName)\
   NewSection (Name,\
                kdspClearSection | kdspSaveSection | kdspBankB
                | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspInputBuffer | kdspOutputBuffer
                | kdspStaticSection,\
                SectionDataType,\
                ModuleName)\
                AIAOSize * long 0

#define NewOutputCRBSection (Name, AIAOSize, SectionDataType, ModuleName)\
   NewSection (Name,\
                kdspSaveSection | kdspBankB
                | kdspSaveOnContextSwitch | kdspDSPUseOnly
                | kdspOutputBuffer | kdspStaticSection,\
                SectionDataType,\
                ModuleName)\
                AIAOSize * long 0
```

```
#define NewScalableInputAIAOSection (Name, AIAOScale, SectionDataType,
   ModuleName)\NewSection (Name,\
                  kdspLoadSection | kdspBankB
                  | kdspSaveOnContextSwitch | kdspDSPUseOnly
                  | kdspInputBuffer | kdspStaticSection
                  | kdspScalableSection,\
                  SectionDataType,\
                  ModuleName)
                  AIAOScale * long 0

#define NewTempScalableAIAOSection (Name, AIAOScale, SectionDataType,
   ModuleName)\NewSection (Name,\
                  kdspLeaveSection | kdspBankB
                  | kdspSaveOnContextSwitch | kdspDSPUseOnly
                  | kdspScalableSection | kdspClearSection,\
                  SectionDataType,\
                  ModuleName)\
                  AIAOScale * long 0

#define NewScalableOutputPRBSection (Name, AIAOScale, SectionDataType,
   ModuleName)\NewSection (Name,\
                  kdspClearSection | kdspSaveSection | kdspBankB
                  | kdspSaveOnContextSwitch | kdspDSPUseOnly
                  | kdspInputBuffer | kdspOutputBuffer
                  | kdspStaticSection | kdspScalableSection,\
                  SectionDataType,\
                  ModuleName)\
                  AIAOScale * long 0

#define NewScalableOutputCRBSection (Name, AIAOScale, SectionDataType,
   ModuleName)\NewSection (Name,\
                  kdspSaveSection | kdspBankB
                  | kdspSaveOnContextSwitch | kdspDSPUseOnly
                  | kdspOutputBuffer | kdspStaticSection
                  | kdspScalableSection,\
                  SectionDataType,\
                  ModuleName)\
                  AIAOScale * long 0

#define NewCachedProgramSection (Name, ModuleName)\
   NewSection (Name,\
                  kdspLoadSection | kdspBankA | kdspDSPUseOnly
                  | kdspNotIOBufferSection,\
                  kdspNonData,\
                  ModuleName)
```

```
#define NewExternalProgramSection (Name, ModuleName)\
   NewSection (Name,\
                  kdspLeaveSection | kdspNotIOBufferSection,\
                  kdspNonData,\
                  ModuleName)

#define NewParameterSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                  kdspExternal | kdspNotIOBufferSection,\
                  SectionDataType,\
                  ModuleName)

#define NewTableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                  kdspLoadSection | kdspBankA | kdspDSPUseOnly
                  | kdspNotIOBufferSection,\
                  SectionDataType,\
                  ModuleName)

#define NewStateVariableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                  kdspLoadSection | kdspSaveSection | kdspBankB
                  | kdspDSPUseOnly | kdspSaveOnContextSwitch
                  | kdspNotIOBufferSection,\
                  SectionDataType,\
                  ModuleName)

#define NewTempVariableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                  kdspLeaveSection | kdspBankB
                  | kdspSaveOnContextSwitch | kdspDSPUseOnly
                  | kdspNotIOBufferSection,\
                  SectionDataType,\
                  ModuleName)
```

```
//=====================================================================
//                       DSP API MACROS
//=====================================================================
//=====================================================================
//                          GENERAL
//=====================================================================

#define  BlockMove(theSrcPtr,theDestPtr,theCount)\
      r1  = (long)  theSrcPtr;\
      r2  = (long)  theDestPtr;\
      r3  = (long)  theCount;\
      RV  = (long)  Find(kevtBlockMove);\
      RV  = (long) *RV;\
      r15 = (ushort24) 0x0004;\
      call RV (RV);\
      r16 = (ushort24) 0x0004

#define  PcLabel(theSectionLabel) \
      pc + ((theSectionLabel)-(.+8))

#define  Pop(theRegister)\
      SP = (long) SP--;\
      theRegister = (long) *SP;\
      nop

#define  Push(theRegister) *SP++ = (long) theRegister

//=====================================================================
//                       SECTION MANIPULATION
//=====================================================================

#define  CallSection (theSectionName)\
      RV  = (long) MXPB + sectn theSectionName;\
      RV  = (long) *RV;\
      nop
      call RV (RV);\
      nop

#define  GetSectionAddress(theSectionPtr,theSectionName)\
      theSectionPtr = (long) MXPB + sectn theSectionName;\
      theSectionPtr = (long) *theSectionPtr;\
      nop
```

```
#define  GetSectionSize(theSectionSize,theSectionName)\
      RV  = (long)  Find(kevtGetSectionSize);\
      RV  = (long) *RV;\
      r1  = (short) sectn theSectionName;\
      call RV (RV);\
      nop;\
      theSectionSize = (long) r2

#define  GetSectionLabel(theSectionLabelPtr,theSectionLabel)\
      theSectionLabelPtr = (long) MXPB + sectn theSectionLabel;\
      theSectionLabelPtr = (long) *theSectionLabelPtr;\
      nop;\
      theSectionLabelPtr = (long) theSectionLabelPtr + offset theSectionLabel

#define  PopSection (theSectionName)\
      RV  = (long)  Find(kevtCacheSection);\
      RV  = (long) *RV;\
      r1  = (short) sectn theSectionName;\
      call RV (RV);\
      r2  = (ushort24) kdspSelectPop

#define  PushSection (theSectionName)\
      RV  = (long)  Find(kevtCacheSection);\
      RV  = (long) *RV;\
      r1  = (short) sectn theSectionName;\
      call RV (RV);\
      r2  = (ushort24) kdspSelectPush

//=================================================================
//                      MODULE MANIPULATION
//=================================================================

#define  SetSkipCount(theSkipCount)\
      RV  = (long)  Find(kevtSetSkipCount);\
      RV  = (long) *RV;\
      r1  = (long) theSkipCount;\
      call RV (RV);\
      nop
```

```
//=====================================================================
//                       TASK MANIPULATION
//=====================================================================
#define  GetNumRealTimeFrames(numFrames)\
      numFrames = (long)  Find(kevtNumRealTimeFrames);\
      numFrames = (long) *numFrames;\
      nop

#define  SetTaskInactive()    OSCall(kevtSetTaskInactive)

//=====================================================================
//                       FIFO MANIPULATION
//=====================================================================

#define  FIFOGetReadCount(theFIFOName)\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV ;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspOptionNoCopyJustCount | kdspSelectPush)

#define  FIFOGetWriteCount(theFIFOName)\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspOptionNoCopyJustCount | kdspSelectPop)

#define  FIFORead(theSectionName)\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theSectionName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPush)

#define  FIFOReadN(theFIFOName,theCount)\
      r4  = (long)  theCount;\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPush | kdspOptionSpecifyCount)
```

```
#define  FIFOReadNBuffer(theFIFOName,theCount,theBufferPtr)\
      r3  = (long)  theBufferPtr;\
      r4  = (long)  theCount;\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPush | kdspOptionSpecifyCount
            | kdspOptionSpecifyBuffer)

#define  FIFOWrite(theSectionName)\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theSectionName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPop)

#define  FIFOWriteN(theFIFOName,theCount)\
      r4  = (long)  theCount;\
      RV = (long)  Find(kevtCopyFIFO);\
      RV = (long) *RV;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPop | kdspOptionSpecifyCount)

#define  FIFOWriteNBuffer(theFIFOName,theCount,theBufferPtr)\
      r3  = (long)  theBufferPtr;\
      r4  = (long)  theCount;\
      RV  = (long)  Find(kevtCopyFIFO);\
      RV  = (long) *RV;\
      r1  = (short) sectn theFIFOName;\
      call RV (RV);\
      r2  = (ushort24) (kdspSelectPop | kdspOptionSpecifyCount
            | kdspOptionSpecifyBuffer)


//=====================================================================
//                      GPB MANIPULATION
//=====================================================================

#define  GPBElapsedCycles(theCycles)\
      OSCall(kevtGPBElapsedCycles);\
      theCycles = (long) r2
```

```
#define  GPBExpectedCycles(theCycles)\
      OSCall((kevtGPBExpectedCycles);\
      theCycles = (long) r2

#define  GPBSetUseActual()    OSCall(kevtGPBSetUseActual)

//=====================================================================
//                      MESSAGE MANIPULATION
//=====================================================================

#define  SendMessageToHost(theDSPMessagePtr)\
      RV  = (long)  Find(kevtMessageToHost);\
      RV  = (long) *RV;\
      r1  = (long) theDSPMessagePtr;\
      call RV (RV);\
      nop

//=====================================================================
//                      SEMAPHORE MANIPULATION
//=====================================================================

#define  SemaphoreClear (theSemaphorePtr, theMask,
                           theOldSemaphoreValue)\
      RV  = (long)  Find(kevtSemaphoreClear);\
      RV  = (long) *RV;\
      r1  = (long) theSemaphorePtr;\
      r2  = (long) theMask;\
      call RV (RV);\
      nop;\
      theOldSemaphoreValue = (long) r3

#define  SemaphoreSet (theSemaphorePtr, theMask,
                          theOldSemaphoreValue)\
      RV  = (long)  Find(kevtSemaphoreSet);\
      RV  = (long) *RV;\
      r1  = (long) theSemaphorePtr;\
      r2  = (long) theMask;\
      call RV (RV);\
      nop;\
      theOldSemaphoreValue = (long) r3
```

PART THREE

Speech Synthesis and Recognition

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* explains the facilities in the Macintosh Quadra 840AV and Macintosh Centris 660AV system software for generating and understanding human speech. It contains three chapters:

- Chapter 6, "Speech Manager," describes a new Macintosh system software manager that provides a standardized way for applications to generate synthesized speech. The Speech Manager also lets an application control one or more speech synthesizers, which generate spoken sound in specific languages, intonations, and speaking styles.
- Chapter 7, "Introduction to Speech Recognition," contains a basic tutorial for the Speech Setup control panel. This control panel provides commands for controlling the speech recognition functions of the Macintosh Quadra 840AV and Macintosh Centris 660AV computers.
- Chapter 8, "Speech Rules," describes the speech rules that are built into the Macintosh Quadra 840AV and Macintosh Centris 660AV system software.

CHAPTER 6

Speech Manager

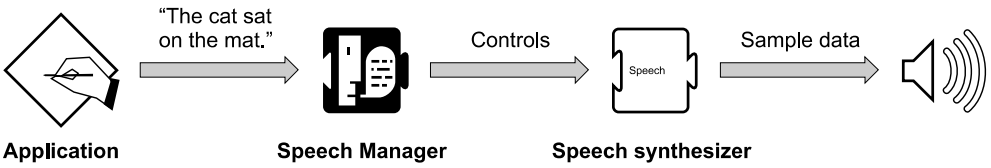
This chapter describes Apple's Speech Manager, which provides a standardized method for Macintosh applications to generate synthesized speech.

This chapter provides an overview of the Speech Manager followed by general information about generating speech from text. The necessary information and calls needed by all text-to-speech applications are given next, followed by a simple example of speech generation. More advanced calls and special-purpose routines are described last.

Speech Manager Overview

A complete system for speech synthesis consists of the elements shown in Figure 6-1.

Figure 6-1 Speech synthesis components



An application calls routines in the Speech Manager to convert character strings into speech and to adjust various parameters that affect the quality or character of the spoken output. The Speech Manager is responsible for dispatching these requests to a speech synthesizer. The speech synthesizer converts the text into sound and creates the actual audio output. Hardware support for speech generation in the Macintosh Quadra 840AV and Macintosh Centris 660AV is described in "Sound I/O," in Chapter 2.

The Apple-supplied voices, pronunciation dictionaries, and speech synthesizer may reside in a single file or in separate files. These files are clearly identifiable as Speech Manager–related files and are installed and removed by being dragged into or out of the System Folder. Additional voices can be provided by bundling the resources in the resource forks of specific applications. These resources are considered private to that particular application. It is up to the individual developers to decide whether the voice resources they provide are usable on a systemwide basis or only from within their applications.

In the first release of the Speech Manager, pronunciation dictionaries are managed entirely by the application. The application is free to store dictionaries in either the resource or the data fork of a file. The application is responsible for loading the individual dictionaries into RAM and then passing a handle to the dictionary data to the Speech Manager.

Applications that use the Speech Manager must provide their own human interface for selecting voices and/or controlling other speech characteristics. If voices are provided in separate files, the speech synthesizer developer is responsible for providing a method for

installing these resources into the System Folder or Extensions folder. The computer must be rebooted after speech synthesizers are added to or removed from the System Folder for the desired changes to be recognized.

Speech Manager Concepts

On a simple level, speech synthesis from text input is a two-stage process. First, plain-language English text is converted into **phonemic** representations for the individual words. Phonemes stand for specific sounds; for a complete explanation, see "Summary of Phonemes and Prosodic Controls," later in this chapter. The resulting sequence of phonemes is converted into audible sounds by mapping of the individual phonemes to a series of waveforms, which are sent to the sound hardware to be played.

In reality, each stage is more complicated than this description suggests. For example, during the text-to-phoneme conversion stage, number strings, abbreviations, and special symbols must be detected and converted into appropriate words before being converted into phonemes. When a sentence such as "He earned over $2,000,000 in 1990" is spoken, it would normally be preferable to say "He earned over two million dollars in nineteen-ninety" rather than "He earned over dollar-sign, two, comma, zero, zero, zero, comma, zero, zero, zero, in one, nine, nine, zero." To produce the desired spoken output automatically, knowledge of these sorts of constructions is built into the synthesizer.

The phoneme-to-sound conversion stage is also complex. Phonemes by themselves are often not sufficient to describe the way a word should be pronounced. For example, the word "object" is pronounced differently depending on whether it is used as a noun or a verb. (When it is used as a noun, the stress is placed on the first syllable. When it is used as a verb, the stress is placed on the second syllable.) In addition to stress information, phonemes must often be augmented with pitch, duration, and other information to produce intelligible, natural-sounding speech.

The speech synthesizer has many built-in rules for automatically converting text into the complex phonemic representation described above. However, there will always be words and phrases that are not pronounced the way you want. The Speech Manager allows you to provide raw phonemic information directly in order to enable very precise control over the spoken output.

By default, speech synthesizers expect input in normal language text. However, using the input mode controls of the Speech Manager, you can tell the synthesizer to process input text in raw phonemic form. By using the embedded commands described in the next section, you can even mix normal language text with phonemic text within a single string or text buffer.

See "Summary of Phonemes and Prosodic Controls," later in this chapter, for a listing of the phonemic character set and each character's interpretation.

Using the Speech Manager

This section describes the routines used to add speech synthesis features to an application. It is organized into three sections: "Getting Started" (easy), "Essential Calls—Simple and Useful" (intermediate), and "Advanced Routines."

Getting Started

If you're just getting started with text-to-speech conversion using the Speech Manager, the following routines will get you up and running with minimal effort. If you're developing an application that does not need to choose voices, use more than one channel of speech, or exercise real-time control over the synthesized speech, these may be the only routines you need.

Determining If the Speech Manager Is Available

You can find out if the Speech Manager is available with a single call to the Gestalt Manager.

Use the `Gestalt` toolbox routine and the selector `gestaltSpeechAttr` to determine whether or not the Speech Manager is available, as shown in Listing 6-1. If `Gestalt` returns `noErr`, then the parameter argument will contain a 32-bit value indicating one or more attributes of the installed Speech Manager. If the Speech Manager exists, the bit specified by `gestaltSpeechMgrPresent` is set.

Listing 6-1 Determining if the Speech Manager is available

```
Boolean SpeechAvailable (void) {
   OSErr    err;
   long     result;
   err = Gestalt(gestaltSpeechAttr, &result);
   if ((err != noErr) || !(result &
         (1 << gestaltSpeechMgrPresent)))
      return FALSE;
   else
      return TRUE;
}
```

Determining Which Version of the Speech Manager Is Running

Once you have determined that the Speech Manager is installed, you can see which version of the Speech Manager is running by calling `SpeechManagerVersion`.

SpeechManagerVersion

Returns the version of the Speech Manager installed in the system.

```
pascal NumVersion SpeechManagerVersion (void);
```

DESCRIPTION

`SpeechManagerVersion` returns the version of the Speech Manager installed in the system. This call should be used to determine the compatibility of your program with the currently installed Speech Manager.

RESULT CODES

None

Making Some Noise

The most basic operation of the Speech Manager is accomplished by using the `SpeakString` call. This call passes a specific text string to be spoken to the Speech Manager.

SpeakString

The `SpeakString` function passes a specific text string to be spoken to the Speech Manager.

```
pascal OSErr SpeakString (StringPtr s);
```

`s` Text string to be spoken.

DESCRIPTION

`SpeakString` attempts to speak the Pascal-style text string contained in `myString`. Speech is produced asynchronously using the default system voice. When an application calls this function, the Speech Manager makes a copy of the passed string and creates any structures required to speak it. As soon as speaking has begun, control is returned to the application. The synthesized speech is generated transparently to the application

so that normal processing can continue while the text is being spoken. No further interaction with the Speech Manager is required at this point, and the application is free to release or purge or trash the original string.

If `SpeakString` is called while a prior string is still being spoken, the audio currently being synthesized is interrupted immediately. Conversion of the new text into speech is then initiated. If an empty (zero length) string or a null string pointer is passed to `SpeakString`, it stops the synthesis of any prior string but does not generate any additional speech.

As with all Speech Manager routines that expect text arguments, the text may contain embedded speech control commands.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to speak
`synthOpenFailed`	–241	Could not open another speech synthesizer channel

Determining If Speaking Is Complete

Once an application starts a speech process with `SpeakString`, the next thing it will probably need to know is whether the string has been completely spoken. It can use `SpeechBusy` to determine whether or not the system is still speaking.

SpeechBusy

The `SpeechBusy` routine is useful when you want to ensure that an earlier speech request has been completed before having the system speak again.

```
pascal short SpeechBusy (void);
```

DESCRIPTION

`SpeechBusy` returns the number of channels of speech that are currently synthesizing speech in the application. If you use just `SpeakString` to initiate speech, `SpeechBusy` will always return 1 as long as speech is being produced. When `SpeechBusy` returns 0, all initiated speech has finished.

RESULT CODES

None

A Simple Example

The example shown in Listing 6-2 demonstrates how to use the routines introduced in this section. It first makes sure the Speech Manager is available. Then it starts speaking a string (hard-coded in this example, but more commonly loaded from a resource) and loops, doing some screen drawing, until the string is completely spoken. This example uses the `SpeechAvailable` routine shown in Listing 6-1.

Listing 6-2 Elementary Speech Manager calls

```
OSErr err;
if (SpeechAvailable()) {
   err = SpeakString("\pThe cat sat on the mat.");
   if (err == noErr)
      while (SpeechBusy() > 0)
         CoolAnimationRoutine();
   else
      NotSoCoolAlertRoutine(err);
}
```

Essential Calls—Simple and Useful

While the routines presented in the last section are simple to use, their applicability is limited to a few basic speech scenarios. This section describes additional routines that let you work with different voices and adjust some basic characteristics of the synthesized speech.

Working With Voices

When describing a person's voice, we talk about the particular set of characteristics that help us to distinguish that person's voice from another. For example, the rate at which one speaks (slow or fast) and the average pitch (high or low) characterize a particular speaker on a crude level. In the context of the Speech Manager, a voice is the set of parameters that specify a particular quality of synthesized speech. This portion of the Speech Manager is used to determine which voices are available and to select particular voices.

Every specific voice has a unique ID associated with it, which is the primary way an application refers to it. Every voice is also associated with a `VoiceSpec` structure that is set up by the `MakeVoiceSpec` routine.

The Speech Manager provides two routines to count and step through the list of currently available voices. `CountVoices` is used to compute how many voices are available with the current system. `GetIndVoice` uses an index, starting at 1, to return information about all currently installed voices.

Use the `GetIndVoice` routine to step through the list of available voices. It will fill a `VoiceSpec` record that can be used to obtain descriptive information about the voice or to speak using that voice.

Any application that wishes to use multiple voices will probably need additional information about the available voices beyond the `VoiceSpec` structure, such as the name of the voice and perhaps what script and language each voice supports. This information might be presented to the user in a "voice picker" dialog box or voice menu, or it might be used internally by an application trying to find a voice that meets certain criteria. Applications can use the `GetVoiceDescription` routine for these purposes.

MakeVoiceSpec

To maximize compatibility with future versions of the Speech Manager, you should always use `MakeVoiceSpec` instead of setting the fields of the `VoiceSpec` structure directly.

```
pascal OSErr MakeVoiceSpec (OSType creator, OSType id, VoiceSpec
*voice);
```

```
typedef struct VoiceSpec {
   OSType   creator; // determines which synthesizer is required
   OSType   id;      // voice ID on the specified synth
} VoiceSpec;
```

Field descriptions

`creator`	The synthesizer required by your application.
`id`	Identification number for this voice.
`*voice`	Pointer to the `VoiceSpec` structure.

DESCRIPTION

Most voice management routines expect to be passed a pointer to a `VoiceSpec` structure. `MakeVoiceSpec` is a utility routine provided to facilitate the creation of `VoiceSpec` records. On return, the passed `VoiceSpec` structure contains the appropriate values.

Voices are stored in resources of type `'ttsv'` in the resource fork of Macintosh files. The Speech Manager uses the same search method as the Resource Manager, looking for voice resources in three different locations when attempting to resolve `VoiceSpec` references. It first looks in the application's resource file chain. If the specified voice is not found in an open file, it then looks in the System Folder and the Extensions folder (or in just the System Folder under System 6) for files of type `'ttsv'` (single-voice files) or `'ttsb'` (multivoice files) and in text-to-speech synthesizer component files (file type `'INIT'` or `'thng'`). Voices stored in the System Folder or Extensions folder are normally available to all applications. Voices stored in the resource fork of an application files are private to the application.

RESULT CODE

noErr	0	No error

While the determination of specific voice ID values is mostly left to synthesizer developers, the voice creator values are specified by Apple (they would ordinarily correspond to a developer's currently assigned creator ID). For both the creator and id fields Apple further reserves the set of OSType values specified entirely by space characters and lowercase letters. Apple is establishing a standard suite of voice ID values that developers can count upon being available with all speech synthesizers.

CountVoices

The CountVoices routine returns the number of voices available.

```
pascal OSErr CountVoices (short *voiceCount);
```

voiceCount Number of voices available to the application.

DESCRIPTION

Each time CountVoices is called, the Speech Manager searches for new voices. This algorithm supports dynamic installation of voices by applications or users. On return, the voiceCount parameter contains the number of voices available.

RESULT CODE

noErr	0	No error

GetIndVoice

The GetIndVoice routine returns information about a specific voice.

```
pascal OSErr GetIndVoice (short index, VoiceSpec *voice);
```

index	Index value for a specific voice.
*voice	Pointer to the VoiceSpec structure.

DESCRIPTION

As with all other index-based routines in the Macintosh Toolbox, an index value of 1 causes GetIndVoice to return information for the first voice. The order that voices are returned is not presently defined and should not be assumed. Speech Manager behavior when voice files or resources are added, removed, or modified is also presently

undefined. However, calling `CountVoices` or `GetIndVoice` with an index of 1 will force the Speech Manager to update its list of available voices. `GetIndVoice` will return a `voiceNotFound` error if the passed index value exceeds the number of available voices.

RESULT CODES

`noErr`	0	No error
`voiceNotFound`	–244	Voice resource not found

GetVoiceDescription

The `GetVoiceDescription` routine returns information about a voice beyond that provided by `GetIndVoice`.

```
pascal OSErr GetVoiceDescription (VoiceSpec *voice,
         VoiceDescription *info, long infoLength);
```

```
enum {kNeuter = 0, kMale, kFemale};// returned in gender field below

typedef struct VoiceDescription {
   long        length;        // size of structure
   VoiceSpec   voice;         // synth and ID info for voice
   long        version;       // version code for voice
   Str63       name;          // name of voice
   Str255      comment;       // additional text info about voice
   short       gender;        // neuter, male, or female
   short       age;           // approximate age in years
   short       script;        // script code of text voice can process
   short       language;      // language code of voice output speech
   short       region;        // region code of voice output speech
   long        reserved[4];   // always zero - reserved
} VoiceDescription;
```

Field descriptions

`*voice`	Pointer to the `VoiceSpec` structure.
`*info`	Pointer to structure containing parameters for the specified voice.
`infoLength`	Length in bytes of `info` structure.

DESCRIPTION

The Speech Manager fills out the passed `VoiceDescription` fields with the correct information for the specified voice. If a null `VoiceSpec` pointer is passed, the Speech Manager returns information for the system default voice. If the `VoiceDescription`

pointer is null, the Speech Manager simply verifies that the specified `VoiceSpec` refers to an available voice. If `VoiceSpec` does not refer to a known voice, `GetVoiceDescription` returns a `voiceNotFound` error, as shown in Listing 6-3.

To maximize compatibility with future versions of the Speech Manager, the application must pass the size of the `VoiceDescription` structure. Having the application do this ensures that the Speech Manager will never write more data into the passed structure than will fit even if additional information fields are defined in the future. On returning from `GetVoiceDescription`, the length field is set to reflect the length of data actually written by this routine.

Listing 6-3 Getting information about a voice

```
OSErr GetVoiceGender (VoiceSpec *voicePtr, short *gender) {
   OSErr       err;
   VoiceDescriptionvd;
   err = GetVoiceDescription
      (voicePtr,&vd,sizeof(VoiceDescription));
   if (err == noErr) {
      if (vd.length > offsetof(VoiceDescription,gender))
         *gender = vd.gender;
      else
         err = badStructLen;
   }
   return err;
}
```

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error
`memFullErr`	–108	Not enough memory to load voice into memory
`voiceNotFound`	–244	Voice resource not found

Managing Connections to Speech Synthesizers

Using the routines described earlier in this chapter, an application can select the voice with which to speak. The next step is to associate the selected voice with the proper speech synthesizer. This is accomplished by creating a new speech channel with the `NewSpeechChannel` routine. A speech channel is a private communication connection to the speech synthesizer, much as a file reference number is a communication channel to an open file in the Macintosh file system.

The `DisposeSpeechChannel` routine closes a speech channel when the application is finished with it and releases any resources that have been allocated to support the speech synthesizer and are no longer needed.

NewSpeechChannel

The `NewSpeechChannel` routine creates a new speech channel.

```
pascal OSErr NewSpeechChannel (VoiceSpec *voice,
   SpeechChannel *chan);
```

`*voice` Pointer to the `VoiceSpec` structure.

`*chan` Pointer to the new channel.

DESCRIPTION

The Speech Manager automatically locates and opens a connection to the proper synthesizer for a specified voice and sets up a channel at the location pointed to by `*chan` so that it is ready to speak with that voice. If a null `VoiceSpec` pointer is passed to `NewSpeechChannel`, the Speech Manager uses the current system default voice.

There is no predefined limit to the number of speech channels an application may create. However, system constraints on available RAM, processor loading, and number of available sound channels may limit the number of speech channels actually possible.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to open speech channel
`synthOpenFailed`	–241	Could not open another speech synthesizer channel
`voiceNotFound`	–244	Voice resource not found

DisposeSpeechChannel

The `DisposeSpeechChannel` routine disposes of an existing speech channel.

```
pascal OSErr DisposeSpeechChannel (SpeechChannel chan);
```

`chan` Specific speech channel.

DESCRIPTION

This routine disposes of an existing speech channel. Any speech channels that have not been explicitly disposed of by the application are released automatically by the Speech Manager when the application quits.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Invalid `SpeechChannel` parameter

Starting and Stopping Speech

All the remaining routines in this section require a valid speech channel to work properly. Once the application has successfully created a speech channel, it can start to speak. You use the `SpeakText` routine to begin speaking on a speech channel.

At any time during the speaking process, the application can stop the synthesizer's speech. The `StopSpeech` routine will immediately abort any speech being produced on the specified speech channel and force the channel back into an idle state.

SpeakText

The `SpeakText` routine converts designated text into speech.

```
pascal OSErr SpeakText (SpeechChannel chan, Ptr textBuf, long
   textBytes);
```

Field descriptions

`chan`	Specific speech channel.
`textBuf`	Buffer of text.
`textBytes`	Length of `textBuf`.

DESCRIPTION

In addition to a valid speech channel, `SpeakText` expects a pointer to the text to be spoken and the length in bytes of the text buffer. `SpeakText` will convert the given text stream into speech using the voice and control settings for that speech channel. The speech is generated asynchronously. This means that control is returned to your application before the speech has finished (probably even before it has begun). The maximum length of text buffer that can be spoken is limited only by the available RAM. However, it's generally not very friendly to force the user to listen to long uninterrupted text unless the user requests it.

If `SpeakText` is called while it is currently busy speaking the contents of a prior text buffer, it will immediately stop speaking from the prior buffer and will begin speaking from the new text buffer as soon as possible. As with `SpeakString`, described on page 267, if an empty (zero length) string or a null text buffer pointer is passed to `SpeakText`, this will have the effect of stopping the synthesis of any prior text but will not generate any additional speech.

▲ **WARNING**
With `SpeakText`, unlike with `SpeakString`, the text buffer must be locked in memory and must not move during the entire time that it is being converted into speech. This buffer is read at interrupt time, and very undesirable effects will happen if it moves or is purged from memory. ▲

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Invalid `SpeechChannel` parameter

StopSpeech

The `StopSpeech` routine terminates speech delivery on a specified channel.

```
pascal OSErr StopSpeech (SpeechChannel chan);
```

`chan` Specific speech channel.

DESCRIPTION

After returning from `StopSpeech`, the application can safely release any text buffer that the speech synthesizer has been using. The `SpeechBusy` routine, described on page 268, can be used to determine if the text has been completely spoken. (In an environment with multiple speech channels, you may need to use the more advanced status routine `GetSpeechInfo`, described on page 286, to determine if a specific channel is still speaking.) `StopSpeech` can be called for an already idle channel without ill effect.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Invalid `SpeechChannel` parameter

Using Basic Speech Controls

The Speech Manager provides several methods of adjusting the variables that can affect the way speech is synthesized. Although most applications probably do not need to use these advanced features, two of the speech variables, speaking rate and speaking pitch, are useful enough that a very simple way of adjusting these parameters on a channel-by-channel basis is provided. Routines are supplied that enable an application to both set and get these parameters. However, the audible effects of changing the rate and pitch of speech vary from synthesizer to synthesizer; you should test the actual results on all synthesizers with which your application may work.

Speaking rates are specified in terms of words per minute (WPM). While this unit of measurement is difficult to define in any precise way, it is generally easy to understand and use. The range of supported rates is not predefined by the Speech Manager. Each speech synthesizer provides its own range of speaking rates. Furthermore, any specific rate value will correspond to slightly different rates with different synthesizers.

Speaking pitches are defined on a musical scale that corresponds to the keys on a standard piano keyboard. By convention, pitches are represented as fixed-point values in the range from 0.000 through 100.000, where 60.000 corresponds to middle C (261.625