When the Real Time Manager wants to determine if there is enough processing time still available to install a new task, it uses a simple algorithm to decide which of the two available values, the GPB estimate or the GPB actual value, it should use for each module in its calculations. This selection is based on the state of the `UseActualGPB` flag in each module header.

For smooth algorithms, this flag is always set. The selection algorithm is this: if the actual value is non-zero and the flag is set, use the GPB actual value as the current value; otherwise, use the GPB estimate. This algorithm is designed to give the most accurate accounting of the available GPB at any given time. However, the estimated value is used until the module has a chance to run at least once. After that, the actual value is used, whether it is smaller or larger than the estimate. This is how the GPB system automatically adapts to different CPU configurations.

GPB for Lumpy Algorithms

The simple approach to GPB used for smooth algorithms does not work for lumpy algorithms. A somewhat different approach is required for this case. First, it is necessary to separate lumpy algorithms into two different classes: **smart lumpy algorithms** and **dumb lumpy algorithms.** A smart lumpy algorithm determines cases when it is executing code that will result in maximum utilization of GPB. A dumb lumpy algorithm cannot determine when this may be the case.

An example of a smart lumpy algorithm is a multirate modem. There are various stages of the modem, including initialization, setup, and data transfer. The maximum GPB use is usually taken by one of the steady-state data processing programs. When this algorithm is reached, the DSP program calls the `GPBSetUseActual` routine.

An example of a dumb lumpy algorithm is a Huffman decoder. This decoder takes longer to decode some bit streams than others, and there is no way to tell beforehand how long it will take. In fact, the processing time can grow without limit in the case of random noise input.

Two different mechanisms handle these two cases. For smart lumpy algorithms, the DSP program knows where the maximum GPB usage is in the code, and so is required to set the `UseActualGPB` flag with the `GPBSetUseActual` routine. The DSP operating system does not actually set the flag until the GPB calculations for this module are completed. This forces the Real Time Manager to continue using the estimated value until after the peak use frame has occurred. After that, the actual value correctly reflects the processing needed by this module on this hardware configuration. The DSP operating system continues to use the peak detection algorithm for computing the actual value, so future peaks may slightly increase the actual value because of variations in I/O and bus utilization.

For dumb lumpy algorithms, the DSP program can check on the available processing time left in the real-time frame, and shut down the process if an overrun is about to happen or has already happened.

There are two macro calls to the DSP operating system that support the dumb lumpy algorithm. The `GPBExpectedCycles` macro returns the expected processing time; the `GPBElapsedCycles` macro returns the amount of processing time used so far. If

the amount used so far is getting close to the expected time, the module must execute its processing termination procedure. This procedure should end the processing in whatever manner is appropriate for this algorithm. If the processing duration has exceeded the time, the `UseActualGPB` flag should be set, and the processing termination procedure should be followed.

If the dumb lumpy algorithm exceeds its GPB estimate, it may cause a frame overrun. If this happens, the offending real-time task that includes this module is set inactive by the DSP operating system, and the application is notified by an interrupt. This process is described in "Frame Overruns," later in this chapter.

Dumb lumpy algorithms are tricky to program correctly. If at all possible, such algorithms should not be done in real time, but in timeshare, where length of execution is not a vital factor.

Fast Execution Versus Real-Time Execution

A task executes faster as a real-time task than as a timeshare task only if the real-time task list is using most of the processing bandwidth of the DSP. In many cases, running in the timeshare list will yield more processing time. By carefully analyzing what applications need real-time processing and what need "run as fast as you can go" processing, you can decide which tasks should go into the timeshare list. Candidates for timesharing normally include tasks such as lossless compression of disk files, graphics animation, and video decompression. All such tasks should use as much DSP bandwidth as possible, because the more they run the sooner they finish. Such tasks must not be confused with real-time tasks, which require a specific amount of data be processed during a specific time period.

Processor Allocation for Timeshare Tasks

Timeshare processing is considerably different than real-time processing. A timeshare task often has no way to determine how much processing time it will have in a given frame. It is even possible to load the real-time task list so that no timeshare task execution is possible. Bear in mind that it takes a significant amount of processing time to load and unload a timeshare task. If there is not sufficient time to perform both operations the task will not execute during that frame.

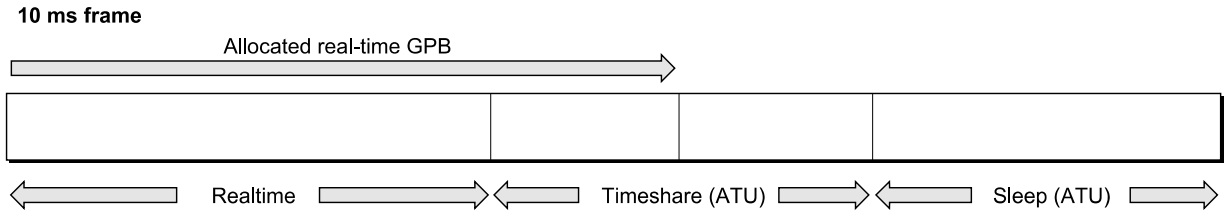
Two numbers can help an application determine if it is worth installing a timeshare task in a given DSP task list. The two numbers are

- **Average timeshare available (ATA).** This is effectively the average sleep time that the DSP is getting per frame. It represents actual unused DSP processing, averaged over several frames.
- **Average timeshare used (ATU).** This number is effectively the average amount of timeshare being consumed by timeshare tasks that are already installed.

Adding the two numbers above yields the **average total timeshare (ATT)** available. Figure 3-16 diagrams this concept.

Note
The application calculates the ATA and ATU using the maximum number of cycles for the processor, the number of real-time cycles allocated, the number of real-time cycles used during the last frame, and the number of timeshare cycles used during the last frame. ◆

Figure 3-16 Timeshare capacity figures



As shown in Figure 3-16, the ATT value is not necessarily the difference between the frame processing capacity and the GPB allocated to real-time tasks. It is often the case that real-time tasks are inactive, or not running at full processing bandwidth. This makes additional timeshare processing available.

The averaging process is used to calculate the timeshare processing numbers because they will usually fluctuate with time. The numbers are provided to allow an application to determine if installing a timeshare task is effective at any given time.

Once a timeshare task is installed, it is recommended that the application check the value of the ATT every so often to make sure that it is still getting service from its timeshare task. Alternatively, the timeshare task itself can report to the application on its activity level. The Real Time Manager does not warn the application when timeshare tasks are not being executed.

Frame Overruns

When several tasks have been installed on the DSP, or if one large and lumpy task is installed, and if the estimated GPB requirements are not accurate, it is possible for the DSP to still be processing data when the next frame interrupt is received. This results in a frame overrun. There are three categories of frame overrun:

- Category one: the DSP acknowledges the current frame interrupt after the next interrupt is received, but before a second interrupt. The DSP misses only one interrupt.
- Category two: the DSP acknowledges the frame interrupt after two interrupts have been received, but before a third interrupt. The DSP misses two interrupts.
- Category three: the DSP has not acknowledged the frame interrupt for five successive interrupts. The DSP misses five or more interrupts.

Category One Frame Overrun

The DSP operating system detects a frame overrun if the interrupt line is asserted before it has been acknowledged. When a category one frame overrun occurs the DSP operating system attempts to recover during the next frame. Since the DSP operating system cannot tell if one or more frames has passed it assumes only one frame has been skipped. To recover, the DSP operating system checks all modules for their current GPB usage, the task with the module having the worst overage is set inactive, and the application is notified. All other clients (such as toolbox routines) are notified that the DSP has skipped a frame.

When an application receives a task inactive message from the Real Time Manager it should deallocate the offending module. This will update the DSP Prefs file with the correct GPB value for that module. The application can then reallocate the module and attempt to reinstall the task. When an application receives a skipped frame message it can do anything from ignoring it to removing and reinstalling the task.

Category Two Frame Overrun

Since the DSP cannot determine how many frames have passed, the external interrupt logic must detect a category two frame overrun. To recover, the interrupt logic sends a hardware interrupt to the main processor and the Real Time Manager executes its DSP overrun recovery code. The Real Time Manager checks with the DSP operating system for the offending module and sets it inactive. If the DSP operating system cannot identify the worst-case module then the Real Time Manager will determine which module is at fault. The Real Time Manager then issues the DSP a reset command and the application that installed the offending module is notified that the task is inactive. All other clients (such as Toolbox routines) are notified that the DSP has been restarted.

The application that receives the task inactive message should respond in the same way as for a category one overrun. When an application receives the DSP restart message it should check the task's memory for possibly corrupted data or code. The recommended response is to remove, rebuild, and reinstall the task.

Category Three Frame Overrun

In the event that the DSP does not respond to interrupts by the sixth frame, the frame overrun logic will issue a hardware reset to the DSP and I/O subsystems. In this case it is assumed that both the DSP and the main processor have crashed. It is important that the DSP and I/O subsystems be reset to prevent possible problems in the output subsystems—for example, a fixed sound on the speaker or the telecom system left offhook.

Recovery from a category three frame overrun is impossible. All clients, including application and Toolbox routines, must start over and install their tasks from the beginning.

Data Structures

As explained in "Real-Time Processing Architecture," earlier in this chapter, it is important to distinguish DSP modules from DSP tasks:

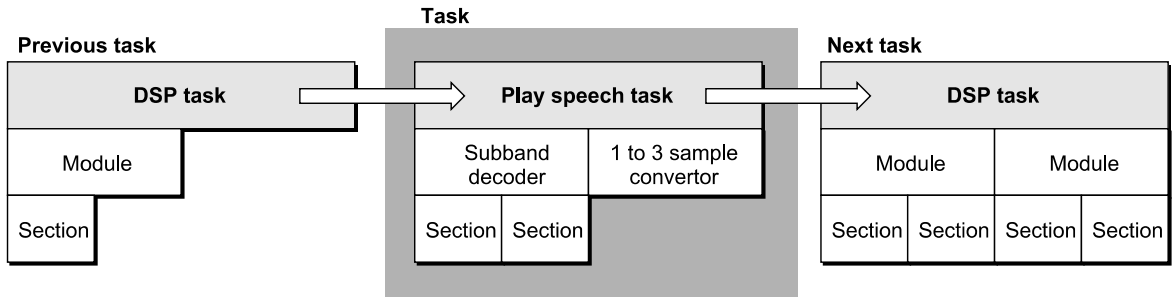
- DSP *modules* are the basic building blocks of digital signal processing software. They always include DSP code. They also usually include some data, input and output buffers, and parameter blocks. There are an infinite number of combinations possible, depending on the desired function.
- A DSP *task* is made up of one or more DSP modules. The purpose of this grouping is to place together, in the appropriate order and with the appropriate I/O buffer connections, all of the DSP modules needed to complete a particular job. A DSP task will frequently contain only one DSP module.

The DSP module is *provided to* the Macintosh program as a resource, and is loaded into a DSP task using the Real Time Manager. A DSP task is *constructed by* the Macintosh application using a series of calls to the Real Time Manager. These calls create the task structure, load and connect modules in the desired arrangement, allocate the required memory, and install the completed task into the DSP task list. The reason for combining modules into tasks is to ensure that the combined function is always executed as a set.

A good example of a task is one that plays compressed speech that was recorded via the telecom subsystem. The data is recorded via the subband decoder at 8 kHz sample rate and compressed before being stored on a disk drive. To play the data over the speaker, it must be decompressed back to 8 kHz samples, and then the sample rate must be converted to 24 kHz data to match the sample rate of the speaker system. A diagram of this example is shown in Figure 3-17.

This task is executed by following the chain of modules from left to right. The task is activated or deactivated as a single unit. It is also installed and removed from the DSP task list as a unit.

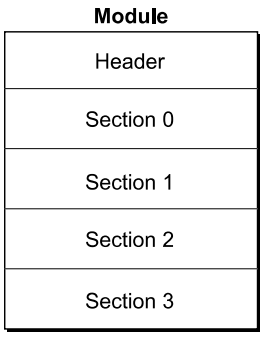
Figure 3-17 Task with two modules



Sections Defined

The internal structure of the DSP module is compartmentalized into code and data blocks. It is this design of the DSP module that gives the real-time data processing architecture its real power and flexibility. Each module is made up of a header and one or more *sections*, as shown in Figure 3-18.

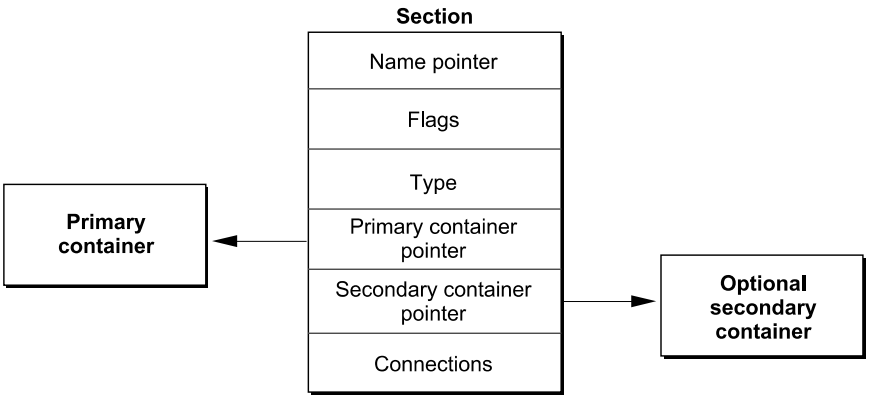
Figure 3-18 The module data structure



The header contains information about the entire module, such as its name, GPB information, and control flags. Also included in the header is a count of the number of sections in the module. This allows the module data structure to be of variable length.

Each section also has a name, flags, and data-type fields. In addition, each section has pointers to two containers. It is the containers that actually hold the data or code for the section. The sections are the building blocks of the module. A section can point to code, data tables, variables, buffers, parameters, work space, or any other resource needed to provide the desired function. The only requirement is that the first section must always point to code. A simplified diagram of a section is shown in Figure 3-19.

Figure 3-19 The section data structure



Note
The section does not contain the actual code or data used by the DSP chip. Rather, it is a data structure that contains pointers to the code or data block. The DSP operating system uses the section structure and flags to cache the actual code or data block as required. ◆

The Section Control Flags and Data Types are used to control caching and manage buffers. The connection data is also used for buffer management internally to the Real Time Manager. These operations are discussed in "Buffer Connections Between Modules," later in this chapter.

The two containers are called the *primary container* and the *secondary container*. A primary container is always required. The secondary container is optional. The primary container is usually allocated in the cache, but can also be in local memory. The secondary container is usually allocated in local memory, but in special cases can be allocated in the cache. Allocated memory for the containers must be in either local or cache memory.

The visible caching system moves data from the secondary container to the primary container, which is usually moving the contents from local memory to cache memory. This is called a **cache load.** The visible caching system also moves data from the primary container to the secondary container, which is usually moving the contents from cache memory to local memory. This is called a **cache save.**

In cases where no caching is required, only one container is needed. The primary container in this case is located in local memory if it contains fixed data or parameters for communication between the main processor application and the module, or in cache memory if it is simply work space.

The section concept was developed to facilitate creating modules with generic functions that can be used in many different applications. It also forms the basis of the plug-and-play module architecture, where input and output data streams can be interconnected between off-the-shelf modules to create new functions. In addition, it supports several different execution models and is easily adapted to future hardware advances, such as significantly larger cache memories and hardware instruction caches.

AutoCache

With the AutoCache caching model (discussed in "Visible Caching," earlier in this chapter), the section data is moved from the secondary to the primary container, before the module runs, if the Load flag is set. Likewise, the section data is moved from the primary to the secondary location after the module runs if the Save flag is set. During execution of an AutoCache module, the primary and secondary pointers never change.

DemandCache

With the DemandCache caching model, two container sections are used much the same way as they are used in AutoCache. The only difference occurs when a section is pushed or popped.

When a section is pushed it changes from a one-container to a two-container section. Data is moved from the secondary to the primary location if the Load flag is set. When a section is popped it changes from a two-container to a one-container section. Again, data is moved from the primary to the secondary container if the Save flag is set.

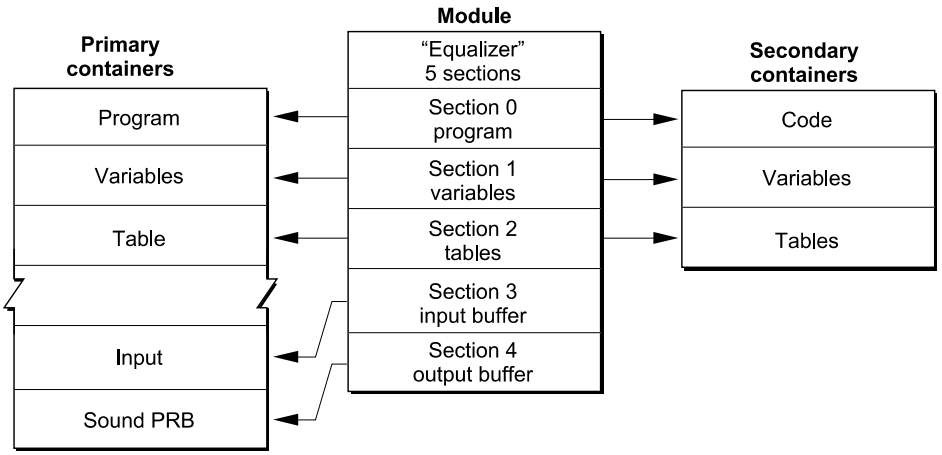
Sections and Caching

The actual operation, with either AutoCache or DemandCache, loads code or data by section into the cache prior to its use, and then saves data back from the cache when completed. The section data structure contains flags, pointers, and other information to support these functions.

For every section there are two possible containers (buffers): the primary container and the secondary container. The caching function moves data between the secondary and primary containers prior to module execution, and moves data between the primary and secondary containers after module execution. Only the minimum required moves are made. For example, it is only necessary to move code into the cache from the secondary container. It is not necessary to move it back, assuming the code is not self-modifying.

A diagram of a sample AutoCache module, including its primary and secondary containers, is shown in Figure 3-20. This example shows five sections in the module: the program (code) section, state variables, a data table, an input buffer, and an output buffer. The first three sections have two containers each, while the last two have only a primary container.

Figure 3-20 Dual-container AutoCache example



In the example, the code, variables, and table sections are loaded into the cache before the code section is executed. After execution completes, only the variables are saved back to local memory. It is important to recognize that the input and output buffers are not moved, but exist in the cache. This buffer mechanism is described in "Buffer Connections Between Modules" and "Buffer Connections Between Tasks," later in this chapter.

Note
This discussion of the caching system is primarily applicable to AutoCache. More detailed information about AutoCache and DemandCache, including the differences between them, is presented in "Execution Models," later in this chapter. ◆

Container Memory Allocation

The structure of modules and sections requires several different blocks of memory. The example shown in Figure 3-20 uses nine different blocks: the module itself, five primary containers, and three secondary containers. The module and the secondary containers are in local RAM, and the primary containers are in the cache.

Substantial memory management and allocation effort is needed to support this type of data structure. Fortunately for the programmer, the work is done automatically by the DSP operating system. The allocation and memory management is done in two phases. When the client loads the module into memory from a resource file, the Real Time Manager allocates all the required blocks in local memory to hold the structure. In the example shown in Figure 3-20, the allocation includes the module itself and three secondary containers. The containers are then loaded with data from the resource file. This completes the first phase of memory allocation.

The application must also specify the I/O connections for the module, a process covered in "Buffer Connections Between Modules," later in this chapter. Once all of this is done, the Real Time Manager calls one of its routines to take care of cache allocation; this is the second phase of allocation. The task is now ready to install. For DemandCache, additional allocation is performed by the DSP operating system at run time.

There are many factors that the Real Time Manager must take into consideration when placing section containers in the cache. First, it must be aware of any reserved memory in the cache. This includes areas for the DSP operating system as well as buffers. Next, it must be aware of the bank configuration of the cache. For some DSP implementations, it is important to locate different sections in different banks to ensure highest performance operation. This is not true for the DSP3210, but it was for the DSP32C and will be true for future versions of the DSP3200 family.

It is important to properly mark the sections for bank preference to ensure correct placement for all future DSP3200 processors. This takes the form of Bank A and Bank B preference flags. If both are set, this indicates that any bank will do. If neither are set, it indicates the section should be located outside of the cache. In the example above, the program, variables, and table sections (primary containers) are located in Bank A. The I/O sections (primary containers) are located in Bank B. The architectural concept behind this bank organization is explained in the AT&T DSP3210 manual.

Other allocation decisions are related to the connections between module I/O buffers. The Real Time Manager attempts to arrange the sections in the cache in such a way as to eliminate as much buffer movement as possible. If a buffer can be set and left in one place without being moved between modules or tasks, it reduces the overhead for maintaining the buffer.

A Complete Software Example

Figure 3-21 diagrams a typical structure of digital signal processing software with sections, modules, and tasks. It shows a dual task list (real-time and timeshare) and adds multiple DSPs and DSP client controls.

Figure 3-21 shows two DSP devices (two separate DSP subsystems), where the structure detail is shown only for the first device. The first device might be a DSP located on the main logic board and the second device might be a DSP located on either a PDS or NuBus card. In machines not having a DSP on the main logic board both DSPs would be located on accessory cards.

For each device, there can be a number of clients. A DSP **client** is either a system Toolbox routine or an application that wishes to use a DSP. An application cannot use a DSP without first signing in as a client. The client must sign in to each device that it intends to use.

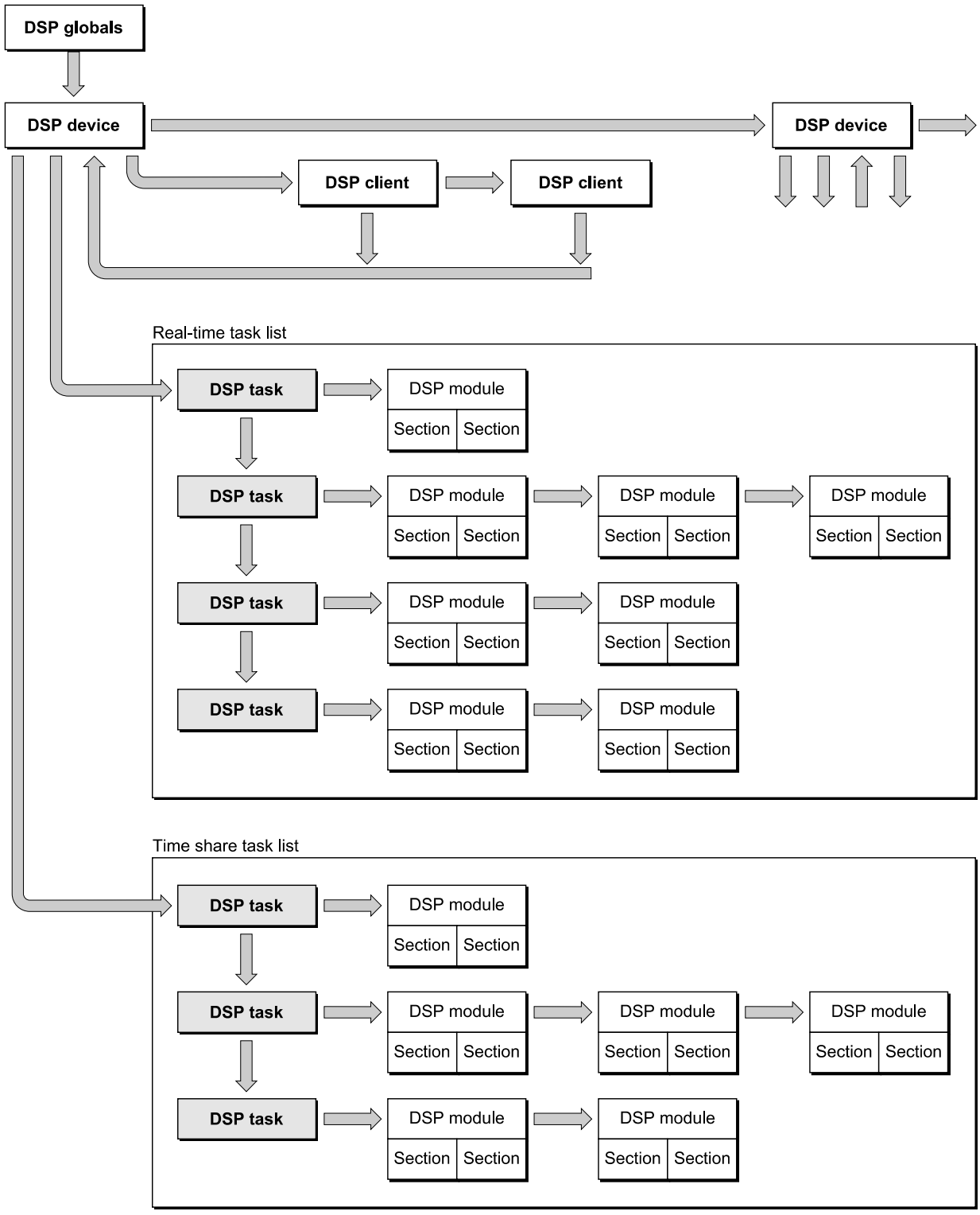
Each device has two task lists. The primary one is for real-time task execution; it is executed once and only once in each frame. The Real Time Manager ensures that the clients do not install too much work in this list, so that the entire list can always be executed by the end of the frame.

The second list is the timeshare task list. It is executed using any time left over in each frame after all real-time tasks have been run. The DSP operating system will repeatedly execute timeshare tasks until it either runs out of time (the next frame begins) or until it makes it through the list once without finding anything to do. If the DSP operating system does not find an active task prior to the frame ending, the DSP is put into sleep mode until the start of the next frame.

Data Buffering

In digital signal processing it is often desirable to connect input and output buffers from several different algorithms, using signal flow techniques. There are routines in the Real Time Manager that accomplish this. The programmer needs to specify the number and format of these buffers (for example, input or output buffers, 32-bit floating-point format, other formats). The buffers can be connected at run time to similar buffers in other, separately designed, algorithms. The application makes calls to the Real Time Manager to specify which connections are desired. The Real Time Manager must attempt to connect these buffers in an efficient manner to minimize the loss of DSP time used in moving buffers around.

Figure 3-21 Data structure overview



FIFO Buffers

First-in, first-out (FIFO) buffers are used to buffer data between processors or processes. Essentially a FIFO is an asynchronous buffer. In the sound player example (see "Software Architecture," earlier in this chapter), FIFOs are used as buffers between the main processor application and the DSP system for music, speech, and sound-effect data. Likewise, a FIFO is used between the DSP and the speaker I/O port, as shown in Figure 3-22.

Figure 3-22 Example of FIFO buffers

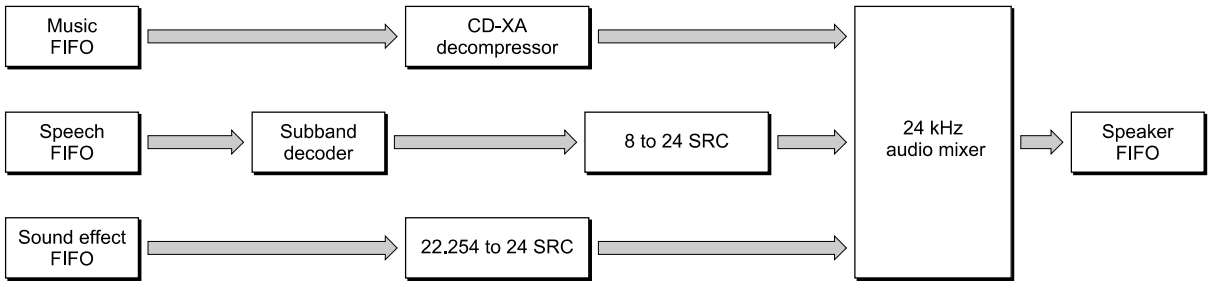


Figure 3-22 shows how FIFOs can be used in a typical application. The speaker FIFO is required because the DSP must keep one frame ahead of the audio serial DMA port. The data FIFOs are necessary because of the slow response time of the disk drive and main processor application. Typically, a buffer in the range of 20 KB to 40 KB is used to buffer the disk to the DSP, depending on the data rate. The disk fills the buffer, and the DSP removes a block every frame. When the FIFO is half empty, the DSP operating system, which handles the FIFO for the DSP module, sends a message to the main processor application. This message tells the main processor application to refill the FIFO from the disk.

FIFOs are also used to buffer output from the DSP to a main processor application—in sound recorders, for example. They work exactly like the FIFOs described above, except in the opposite direction.

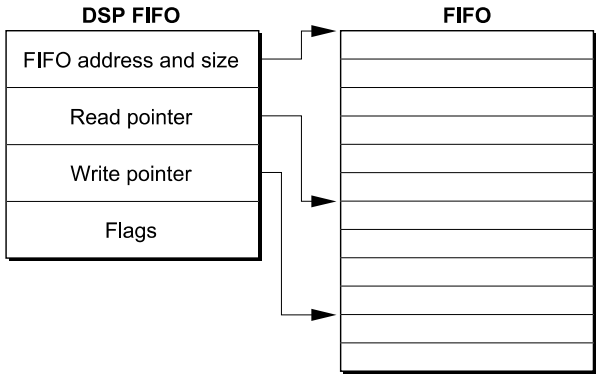
Another use for FIFOs is to handle data that is not synchronized to the frame rate. For example, if data is produced at a rate of 22,254.54 samples per second, the amount of data per frame is either 222 or 223 samples (at 100 frames per second). Using a FIFO allows the processes that are filling and emptying the buffer to read or write exactly the amount of data they need. One prime characteristic of any FIFO is its status. It can be empty or full, half empty or half full, or it can be overrun (following an attempt to put more data into the FIFO than it can contain). An overrun happens if the data consumer cannot keep up with the data producer. It is important to make the FIFO large enough to prevent this from occurring or provide a mechanism in the application to halt data production.

Note
Large FIFOs are usually placed in main memory, because local memory is limited and the data rate is usually small. Small FIFOs can be located in DSP local memory. ◆

FIFOs can also be underrun. This happens when the data receiver is not able to read as much data from the FIFO as it needs to produce one frame's worth of data. The FIFO routines help by automatically doing a zero fill of the unused buffer. For sound, either in DSP floating-point format, 8-bit integer packed format, or 16-bit integer packed format, a zero fill is equivalent to silence. For those functions that require it, the actual amount of data retrieved is reported.

FIFOs are accessed by making `DSPFIFORead` and `DSPFIFOWrite` calls to the DSP operating system. The DSP operating system is responsible for handling status conditions, such as empty or full, half-empty or half-full, and overrun or underrun. The DSP operating system is also responsible for updating the FIFO pointers, and sending messages to the client as required. Typical messages include FIFO Empty (DSP is reading from the FIFO) and FIFO Full (DSP is writing to the FIFO). In order for the DSP operating system to manage this, the FIFO has a header block called `DSPFIFO`. This data structure is shown in Figure 3-23.

Figure 3-23 The FIFO and its data header



Each FIFO requires two separate blocks of memory: the `DSPFIFO` structure located in local memory and the FIFO itself located in either local memory or main memory. Usually, large FIFOs are placed in main memory by the client, to conserve the limited local memory space.

You must write to a FIFO to add data to it and you must read from a FIFO to look at the data in it. Hence you need two separate move operations for each datum: a `DSPFIFOWrite` and a `DSPFIFORead`. Usually, two different processors or processes are responsible for the two operations. For example, the application playing the sound writes to the FIFO, while the sound player task reads from the FIFO.

Real-time data processing FIFOs can read from or write to the DSP side only in longwords. This restriction is necessary because of the real-time cost of reading bytes and reordering them. However, the Real Time Manager supports byte reads and writes to FIFOs from the main processor side. It is also important to note that the DSP operating system masks the lower two bits of the main processor write pointer (for DSP FIFO reads) before using the value to determine the amount of data available in the FIFO. Thus, if the main processor writes six bytes to the FIFO, the DSP will process only four of them. If the main processor writes another three bytes, the DSP will process four more bytes, and so on. This forces all FIFO read/write operations from the DSP to use longwords.

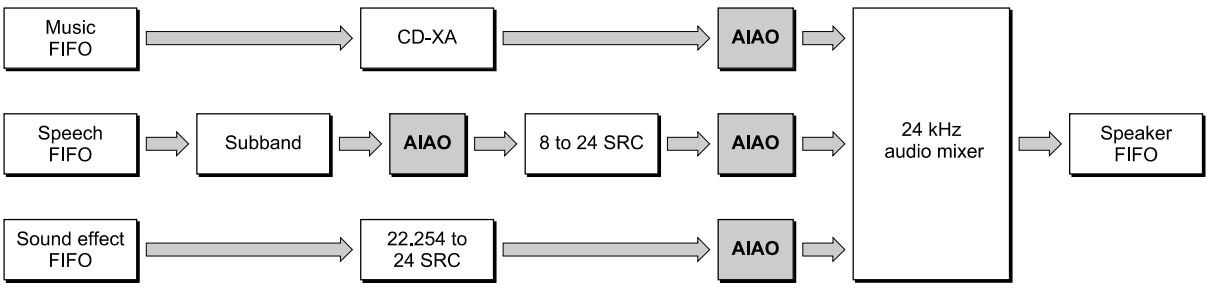
While the FIFO algorithm is ideal for many buffering operations, it requires the DSP operating system to manage the `DSPFIFO` structure and its flags and pointers and also requires dual data movements. These operations make it inefficient for many common buffering operations. It was this realization that resulted in the creation of a new type of buffer, called an AIAO buffer, described in the next section.

AIAO Buffers

AIAO stands for *all-in/all-out*, a naming convention derived from FIFO. AIAO buffers transfer data from one module to another during a given frame. The buffer is transient and acts like a data bucket between modules; the first module fills the buffer, the following module empties it. Another way of thinking about an AIAO is as a frame-synchronous buffer containing only one frame of data.

To understand the need for AIAO buffers, consider the sound player example shown in Figure 3-10. The expanded diagram in Figure 3-24 shows the addition of the AIAO blocks.

Figure 3-24 Code module data flow with AIAOs



AIAO buffers are structured differently from FIFO buffers; they do not contain the header block found in the FIFO. They have a standard section structure and can be used in place, without being moved. It is these characteristics that make the AIAO buffer so efficient.

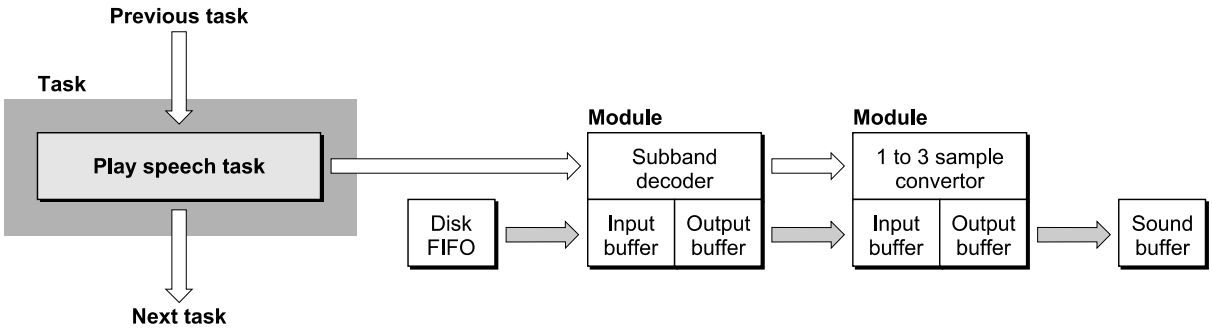
Each module in the example has a fixed number of input and output data streams. These data streams are connected to one another through AIAO buffers. These connections are made between modules by appropriate calls to the Real Time Manager when the task is built. In effect the modules are wired together in a processing network. This plug-and-play module architecture is an important feature of Macintosh real-time data processing.

AIAO buffers can be used to pass data between modules within a task, or to pass data between tasks. The latter case is called an intertask buffer (ITB). The buffers are handled in a similar way, but additional calls are necessary to support them. ITBs require setting aside DSP on-chip memory to pass the data between tasks. See "Buffer Connections Between Tasks," later in this chapter, for more information about ITBs.

Buffer Connections Between Modules

As previously described, AIAO buffers allow very efficient buffering between modules and tasks. To illustrate how they are used in this context, the speech player task shown in Figure 3-17 is expanded in Figure 3-25.

Figure 3-25 Connections between modules



Notice that there are two types of connections in the diagram. First is the control flow, indicated with solid arrows. These are the connections that the DSP operating system follows to execute the real-time task list. The second type is the data flow connection, indicated by shaded arrows. In this example, data flows from the disk FIFO to the subband decoder module, from the decoder to the sample rate converter module, and from the converter module to the sound buffer.

If FIFOs were used for all of these transactions, a total of six `DSPFIFORead` and `DSPFIFOWrite` calls would be needed, not including the FIFO I/O needed to get data into the disk buffer from the disk or to get the sound data into the sound output DMA channel. These six data moves would be:

- read the disk FIFO to cache (10 samples)
- write the decoded data to a connection FIFO (80 samples)
- read the decoded data from the connection FIFO to cache (80 samples)

- write converted data to the sound buffer FIFO (240 samples)
- read the sound buffer FIFO to cache for mixing (240 samples)
- write the mixed sound to the speaker FIFO (240 samples)

The output data from the decoder would be collected in the cache and then written to the FIFO as a single block, for efficiency. This is because the FIFO would usually be in local memory, which would take advantage of the multiple byte move hardware mechanism described in "Access Timing," in Chapter 2. Also, there is considerable overhead for each FIFO DSP operating system call, so it would be time-consuming to make a call for each data point.

Likewise, the data would have to be read from the FIFO into a cache buffer where it could be directly accessed by the conversion algorithm. Once the conversion process was completed, the results would have to be written to the sound buffer FIFO. This process would require a separate stage mixer for the sound buffer. Each sound player would have to feed to a separate sound buffer FIFO, which in turn would have to be read by a mixer and summed, then written to the speaker FIFO. This FIFO would have to be in local memory, since an unknown number of channels of sound must be mixed. The cache is too small to maintain very many buffers and leave enough room for the code and data for the module.

All of this data movement would eat up enormous amounts of DSP processing bandwidth. This would be especially true since many of the FIFOs would be located outside the high-speed cache. Using FIFO buffers would reduce the processing capacity by at least half, due to the relatively slow external memory access.

The enormous FIFO overhead just described is eliminated by using AIAO buffers. Using AIAO buffers in this example, the software needs move the data only once, resulting in a sixfold reduction in buffer overhead. The single move is the initial `DSPFIFORead` by the subband decoder module. Only 10 samples are moved between external memory and the cache with AIAOs, but 890 samples must be moved without AIAOs. This represents an 89 to 1 improvement in bus bandwidth utilization.

This model assumes that the output data from the decoder module is left in the cache, and becomes the input data for the rate converter module. The converter samples the data, summing the results to the on-chip sound buffer.

Note
This interconnection of on-chip buffers is accomplished by the Real Time Manager using the `DSPConnectSections` routine. In effect, this routine sets the section pointers for both AIAOs to the same primary container in cache memory, which flags the DSP Manager to reserve the memory space and not to move the data off-chip between modules. ◆

Notice two important points in the example just given:

- First, the input data is directly available to the sample rate conversion algorithm. The converter can access the data as many times as it needs in the high speed cache without a memory-access speed penalty. Multiple accesses are often required for filtering algorithms, such as rate converters.

- Second, it uses a partial result buffer (PRB) for the sound buffer. A PRB is a buffer that contains a partial mix, and must be summed into, thereby adding to the mix. This is different than a complete result buffer (CRB), which is written into rather than summed into.

The application can directly access the on-chip PRB sound output buffer in the high speed cache. It can read, add to the signal, and write the new results with very little processing impact. This eliminates the entire mixer chain; the mixing function becomes part of the output architecture for the module.

Note
This particular use of AIAOs is specifically for sound functions. It is not a general requirement that AIAO buffers have summing outputs, nor that they have any particular data format. However, if an AIAO is to be connected to the AIAO of another module, the size and data type must match. The specifics of the sound architecture for Macintosh real-time data processing are covered in "Standard Sound Task List," later in this chapter. ◆

The method of operation just described depends heavily on connections being made between the I/O sections of modules. The connection process is handled by the Real Time Manager under the direction of the client. The client specifies what connections are to be made, using calls to the Real Time Manager. Once all connections are completed, Real Time Manager allocation routines place each section in the cache in an appropriate location that ensures the best use of available resources. Every attempt is made to avoid moving buffers. If this is not possible, an attempt is made to move them within the cache between module execution. If all else fails, buffers are temporarily moved into local memory.

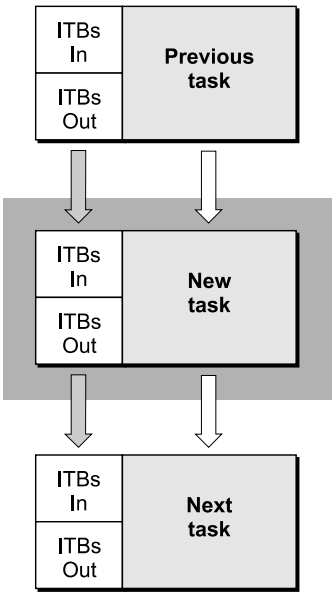
It is this connection mechanism that makes it possible to create generic modules that can be connected in various configurations, depending on the function desired. It is possible to create completely new functions by connecting together existing modules in novel ways.

Buffer Connections Between Tasks

In the speech player task example described in the previous section, the sound buffer is an intertask buffer (ITB). In operation, the AIAO buffer connection between the two modules is an **intermodule buffer.** The Real Time Manager also allows similar connections between tasks, using ITBs.

When the Real Time Manager installs a task, it automatically connects the task to any existing ITBs. The outgoing ITB list from the previous task becomes the incoming ITB list for the new task. Likewise, the incoming ITB list from the next task now becomes the outgoing ITB list for the new task. A diagram of this arrangement is shown in Figure 3-26.

Figure 3-26 ITB connections for previous and next tasks



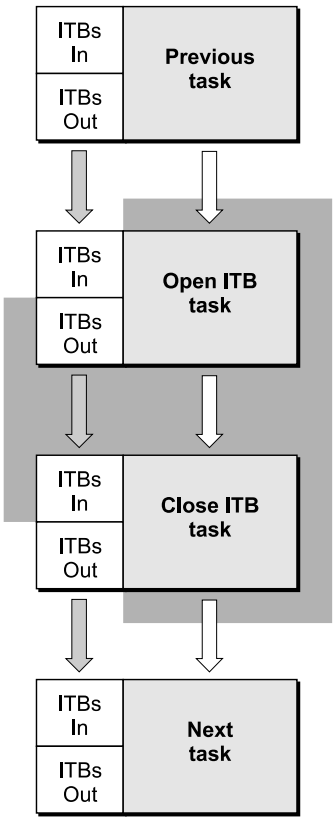
Notice that the diagram above does not represent the data structures associated with the task list, but rather the concept of ITB handling. As shown, each task can import ITBs from previous tasks, and can export ITBs to the next task. Usually, the import and export lists are identical. For sound tasks installed at the appropriate location in the task list, the ITBs are used for stereo PRBs. If a task is installed at the end or beginning of the task list, there will be no ITBs—the ITB pointers will be `nil`.

When the Real Time Manager is asked to create a new ITB, it adds the new ITB to the ITB out list. There is then a mismatch between the output list of the new task and the input list of the next task, because the new ITB was installed prior to the creation of the next task. This new task thus *opens* a new ITB.

The next task the application should install is the *close* task, directly after the first new task. Using the same approach for creating the ITB lists, this task will have the same input list as the previously installed task's output list, and should have the original list as its output list. This concept is illustrated in Figure 3-27.

In Figure 3-27, any additional tasks installed between the open ITB and close ITB tasks will have the additional ITB in both its input and output ITB list. This ITB can be used to pass data between the tasks without requiring off-chip memory. Multiple ITBs can be added in this same way.

Figure 3-27 ITB open and close task configuration



Note

Intertask buffers should be created by the first task installed by an application and should always be deleted by the last task installed. An application installing tasks does not have any information about existing ITBs and cannot pass along information about ITBs that it creates. The application must keep track of creating and deleting ITBs. Subsequent applications installing tasks must assume that only system ITBs are available and cannot use ITBs created by other applications. ◆

IMPORTANT

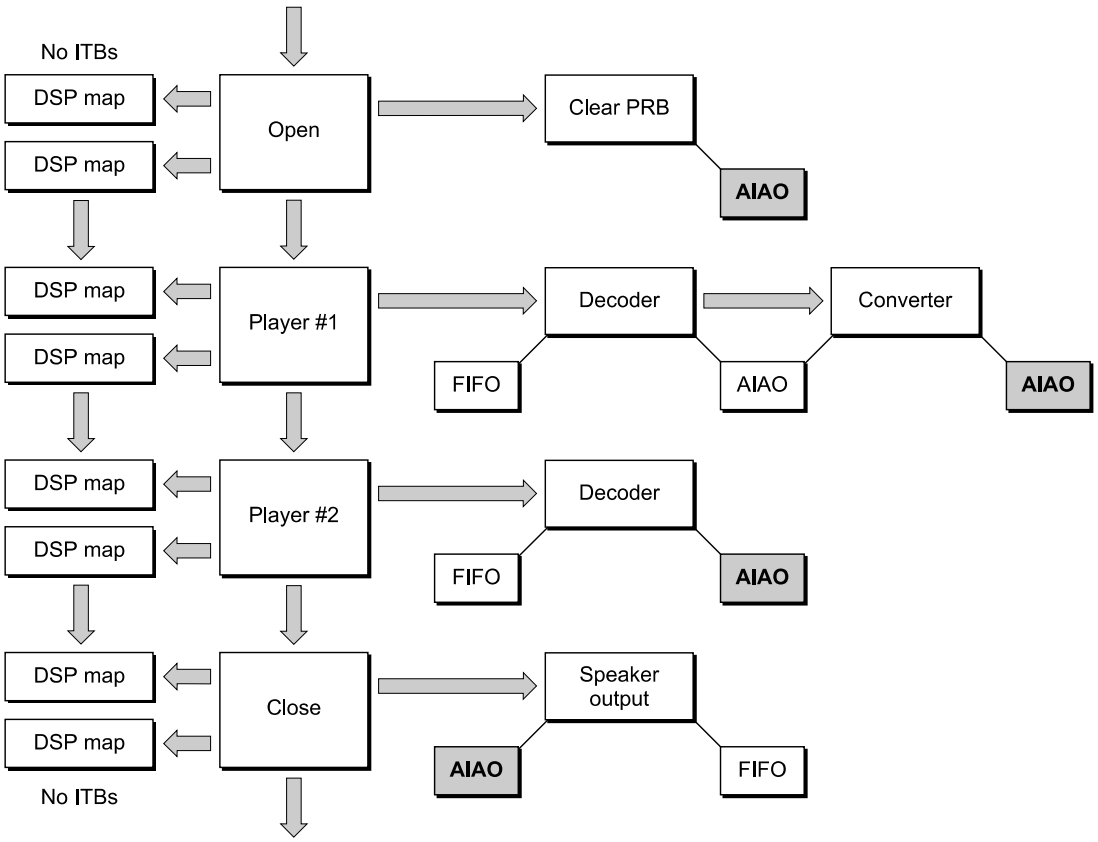
Each ITB uses additional on-chip memory. Installing ITBs can cause a section to be saved and restored between modules because of a lack of on-chip memory. ITBs should only be allocated when absolutely necessary and should be removed as soon as they are not needed. ▲

In the sound player example, a sound output buffer is required to pass the summed output data between all sound player tasks. At the beginning of the chain of sound player tasks, the buffer must be opened and cleared. Then, the output buffer of each player must be connected in turn to this buffer. This is accomplished by reserving cache memory for the buffer using the mechanism described above. A structure called the **DSP map** is used to contain the ITB information. The DSP map provides information to the Real Time Manager's cache allocation routines to make the ITB connections and forestall overruning reserved memory.

During initialization, the Sound Driver automatically signs in to the DSP subsystem and installs the standard sound task list. This creates system ITBs that can be used by other tasks inserted into the standard sound task list. Once the last sound task has finished its operation on the ITB, the ITB can be sent to the speaker FIFO. Once the ITB is dumped to the output FIFO, it must be closed so subsequent tasks will have access to the cache memory that the ITB had been using.

Figure 3-28 is a diagram illustrating this concept. This diagram includes an open buffer task that defines the intertask buffer and clears it, followed by two sound player tasks, and then the close task that closes the ITB after dumping the results to the speaker FIFO.

Figure 3-28 Example of intertask buffers



The intertask sound AIAO buffers in Figure 3-28 are shown in gray. They are connected as the input and output sections of the modules using the ITB connection mechanism. This results in the DSP maps that are used to define the ITBs. There is always an input DSP map and an output DSP map for every task. An open task always has more buffers in the output DSP map than in the input DSP map. Likewise, a close task always has more buffers in the input DSP map than are in the output DSP map.

Note
In Figure 3-28 the open and close operations have been shown as separate tasks. These operations could also be included as part of player 1 and player 2, respectively. ◆

Routines are provided in the Real Time Manager to open and close ITBs. ITBs are also used as the backbone of the sound subsystem, as described in "Standard Sound," later in this chapter.

Unified I/O Architecture

There are many cases where a DSP module could be connected to either an AIAO buffer or a FIFO buffer. For example, the CD-XA decompressor module can be used to supply the input to another module, using an AIAO buffer, but can also be used to create a disk file of uncompressed sound, using a FIFO buffer.

For a module with one input and one output buffer, there are four possible cases for using FIFOs and AIAOs: FIFO to FIFO, AIAO to AIAO, FIFO to AIAO, and AIAO to FIFO. There are two ways to avoid having four different modules to support these four cases:

- A standard FIFO-to-AIAO and AIAO-to-FIFO module can be used whenever an application needs to connect a module to a FIFO.
- A mechanism can be provided to allow either a FIFO or AIAO connection to an AIAO input or output buffer. This has the advantage of reducing the overhead associated with parsing and loading a module. It also provides a somewhat more general unified I/O concept.

The Real Time Manager provides the second capability, by letting a FIFO section be connected to an AIAO section. In this case, the AIAO section is "converted" to a FIFO section. Likewise, an AIAO section can be connected to a FIFO rather than another AIAO section, by creating a FIFO with the `DSPNewFIFO` routine. This mechanism operates without requiring control by the module. Data appears in the input buffer automatically (having been piped from the output AIAO of the previous module) or by having been read from a FIFO by the DSP operating system. This works equally well for output buffers.

This automatic connection feature provides a powerful generic I/O system for DSP modules. It allows modules to be used in many different ways without having to add "helper" modules. It also results in substantially less processing overhead.

Another extension of the FIFO system is the support of linked buffers, which allow the DSP to work in a virtual memory environment. When data is being passed from the disk file system, it is also not necessary to move the data from the disk buffer to a separate FIFO to play it. This reduces the overhead for disk-related I/O.

Execution Models

As explained in "Software Architecture," earlier in this chapter, there are two different execution models for real-time data processing: AutoCache and DemandCache. AutoCache works for many types of functions and is easier to program and use, making it the model of choice in most cases.

DemandCache is the more general model, and can be used to implement very complex functions. Unlike AutoCache modules, DemandCache modules can generate run-time errors. In general, the DemandCache model requires more work from the programmer.

The difference between the two models is in the caching mechanism. For AutoCache, the caching is done automatically, based on source-code flags the programmer sets up for each section in the module. Caching is further supported by a series of "standard" section macros, such as `NewInputFIFOAndScalableBufferSection`, `NewCachedProgramSection`, `NewOutputPRBSection`, `NewParameterSection`, and others. Each macro sets all the necessary flags for the proper caching operation for the section, making it much less likely for errors to be made. For example, the `NewCachedProgramSection` macro sets up caching for a load operation, but not for a save operation. It also specifies the correct cache bank (Bank A) and the section data type (code).

DemandCache, on the other hand, requires that the program explicitly activate the caching functions for each section in the module, using DSP operating system routines. The programmer is responsible for the proper sequencing of these routines. This adds to the workload but also allows various combinations of sections to be cached as needed.

Both models require that the first section be a code section. However, in DemandCache, this section is in off-chip memory when called, and therefore will run much slower than in AutoCache. Normally only a small bootstrap program should be included in this section, which caches the "real" code section and calls it once it is in on-chip memory. It is often more efficient to cache even small code sections on-chip rather than executing them off-chip.

Both models use a section table that is constructed by the Real Time Manager's allocation routines during the installation process. This table contains a list of pointers to appropriate containers, and is used to determine the active location of each section by the module code. These pointers provide the basis for *section-relative addressing* for the Real Time Manager. Relative addressing is required to allow the Real Time Manager the freedom to locate sections in the cache in the most efficient manner. The program must find the location of the sections by using the appropriate macros.

Section Control Flags

The flags that affect how a section is handled by AutoCache and DemandCache are listed below. It is important to distinguish the difference between source-code flags and the run-time flags. Since the Real Time Manager can clear some of the flags when making connections, the flag settings may be different at run time than are indicated by the source code.

- The Load flag indicates that a section should be loaded on-chip. In the general case, this flag indicates a move from one container to another, and can be cleared by the Real Time Manager to eliminate a buffer move. This is useful if the buffer is already on-chip from the module that generated the data.
- The Clear flag indicates that a section should be zeroed before use. This flag should not be set if the Load flag is set, since loading a section and then clearing it does not make sense. It is normally used in conjunction with the Save flag. The Real Time Manager can clear this flag if data already exists in a PRB, and will leave it set if this module is the first to be connected to a PRB.
- The Save flag indicates that a section should be saved off-chip. In the general case, this flag indicates a reverse move from one container to another, and can be cleared by the Real Time Manager to prevent a buffer move. This would be the case if a following module wanted to use the on-chip buffer as input data.
- The Static flag indicates that a section is to be allocated using the static allocation method rather than the dynamic allocation method. Static allocation is done at installation time by the Real Time Manager. Dynamic allocation is done at run time by the module program. This flag facilitates connected buffer management for DemandCache. It is ignored in AutoCache modules; the Real Time Manager will always set the Static flag for AutoCache sections, so its setting doesn't matter in the source code when using AutoCache.
- Bank flags include two preference flags, the Bank A flag and the Bank B flag. If neither are set, this condition specifies external memory for the primary container. If either flag is set, it specifies a cached section and which on-chip bank is preferred. If both flags are set it also indicates a cached section; however, either bank may be used. Notice that it is an error if you specify a Static flag section with no Bank flags set. It is also illegal to specify a Load flag or Save flag with no Bank flags set. For a better understanding of the purpose of banked memory for DSP's, refer to the AT&T DSP3210 manual or other signal processing literature.

It is very important to correctly understand these flags and how they operate. One of the most common errors in programming the DSP is setting an illegal or inappropriate combination of flags. Table 3-3 and Table 3-4, later in this chapter, list combinations of flag settings to watch out for.

Setting Up Input and Output for Connections

When specifying a section as an input buffer, the Load flag should be set, along with a Bank flag (usually Bank B). If you connect this section to another module's section using the Real Time Manager, the Load flag will be cleared if the section is already on-chip from a previous module. On the other hand, if the section is off-chip, or has to be moved off-chip to make room in the cache for an intervening module, the flag will not be cleared.

When specifying a section as a partial result buffer, the Clear flag and Save flag should be set, along with a Bank flag (usually Bank B). If the buffer is connected to another module and will remain on-chip, the Real Time Manager will clear the Save flag. The Clear flag is always cleared except for the first module summing into a PRB. The PRB operations pertaining to sound are covered in "Standard Sound Task List," later in this chapter.

When specifying a section as a complete result buffer, only the Save flag should be set. This is the usual setting for modules other than sound modules.

AutoCache Execution Model

The AutoCache function is activated by a flag in the module header. When the DSP operating system prepares to execute the module it begins with a pre-cache process that preloads any sections indicated by the Load flag in the section data structures. Likewise, any section having the Clear flag set in the section structure is cleared. This is accomplished by the DSP operating system during the parsing of the section structure. This process supports up to 32 sections.

In the example shown in Figure 3-20, the program, variables, and table sections are loaded into the cache. The input buffer is connected to an existing AIAO buffer in the cache, so the Load flag is cleared by the Real Time Manager. Since the output buffer is not the first to connect to the sound output PRB, the Clear flag is cleared. The Save flag is also cleared, since the buffer is to remain on-chip after the module completes execution.

The section table for AutoCache always contains the primary container pointers. These pointers are on-chip for cached sections and workspace, and off-chip for noncached sections, such as parameter buffers used for main processor/DSP interaction. The section table does not change during the execution of the module and is always on-chip.

Once the precache process is complete the first section is called. The first section must always be a code or program section. The code section uses the section table to access information in the other sections. Programmers should use the standard macros to obtain the base address of the section (`GetSectionAddress`) or the address of a label within a section (`GetSectionLabel`). When execution is complete, the module returns control to the DSP operating system.

At this point, the postcache process occurs. All sections with Save flags are copied back to their secondary sections. In the example shown in Figure 3-20, only the variables section must be cached back to local memory. The output buffer is left on-chip. It is important to keep in mind that the Real Time Manager can clear only unneeded flags, based on the connections made. This prevents the Real Time Manager from overriding cases where the programmer decides that a section should always be off-chip to conserve cache space.

Table 3-3 summarizes all the possible flag combinations for AutoCache at run time. The three groups of cases are for one-container sections, two-container sections, and illegal combinations. The Real Time Manager uses the Bank flags to determine the number of containers. If no Bank flags are set, there is a single container off-chip. If one or more Bank flags are set, and either the Load flag or Save flag is set, then two containers are used: one off-chip and one on-chip. If a connection is made which eliminates the need for a secondary container, it is deleted, and the section becomes a one-container section.

The Real Time Manager can also set up cases where both primary and secondary containers are on-chip. It uses this mechanism to automatically move sections to conserve cache space.

Table 3-3 Run-time AutoCache flag combinations

Flags					
Bank	Static	Load	Save	Clear	**Comments**
					Single-container case—Bank flags not set
–	■	–	–	–	Off-chip section (parameters, workspace)
–	■	–	–	■	Off-chip initialized workspace
■	■	–	–	–	Allocate on-chip workspace
■	■	–	–	■	Allocate a cleared on-chip workspace
					Dual-container case—one or more Bank flags set
■	■	–	■	–	Save primary to secondary on exit
■	■	–	■	■	Allocate a clear primary on entry, save to secondary on exit
■	■	■	–	–	Load primary from secondary on entry
■	■	■	■	–	Load primary from secondary on entry, save to secondary on exit
					Illegal cases—caching with one container or dynamic allocation
?	–	?	?	?	Illegal—can't AutoCache with a dynamic section
–	■	■	?	?	Illegal—can't AutoCache with one container
–	■	?	■	?	Illegal—can't AutoCache with one container
■	■	■	–	■	Useless/illegal
■	■	■	■	■	Useless/illegal

NOTE ■ = flag set, – = flag cleared, ? = do not care what flag is set to

▲ **WARNING**

The foregoing method of providing buffer moves cannot be used for DemandCache. ▲

DemandCache Execution Model

DemandCache is used for complex situations such as the following:

- More than 32 sections are required.
- A module must make a decision as to which sections must be cached, based on some program status.
- A module has one or more large functions which must be broken down into smaller steps that will fit in the cache. This requires a sequence of cached sections.

The DemandCache execution model supports any number of sections, but the section table cannot be placed in the cache if the number of sections gets too large. The programmer must make the trade-off between faster execution (in-cache section table) and more cache space available (out-of-cache section table). This selection is done by setting a flag in the module header.

The Real Time Manager determines if two containers are needed by observing the Static flag. If it is set (static allocation), two containers are required. Otherwise, only a single container is required (dynamic allocation). In effect, a static section is allocated in the same way sections are allocated in AutoCache, at module installation rather than during run time.

Note
Since the Static flag is ignored for AutoCache, standard section macros, such as `NewPRBOutputSection`, set this flag. This eliminates the need to have two sets of macros. ◆

Note
While two containers are allocated initially for static sections, the section may become a one-container section once the Real Time Manager has done its work. Clearly, if neither Save flags nor Load flags are set the secondary container is not needed. ◆

Caching is accomplished with calls to the DSP operating system. There are only two routines: `PushSection` and `PopSection`. These routines can do different things, depending on the state of the section control flags. Whenever a section is accessed with these routines, the section table is updated.

The DSP operating system names `PushSection` and `PopSection` reflect the stack-like structure of DemandCache. The `PushSection` routine pushes a section onto the cache stack, and the `PopSection` routine pops a section off of the cache stack. There are actually two different stacks: the Bank A stack and the Bank B stack. Two stacks are necessary to support the maximum performance operation for dual bank caches.

▲ **WARNING**
The `PushSection` and `PopSection` routines will cause a run-time error if neither Bank flag is set, because a section without a bank preference is an external section. ▲

As in AutoCache, the section table is prebuilt during the load process by the Real Time Manager. For AutoCache, the values in the table are always the primary container pointers. DemandCache is more complex—the values in the section table are different for dynamic and static sections. Each case is described separately below.

DemandCache for Dynamic Sections

Most of the sections in a DemandCache module are dynamic, and are allocated by the programmer on the stack. Only a single container is required for these sections. The primary container is always an off-chip buffer, and the secondary container pointer is `nil`. The `PushSection` routine creates a temporary container on the stack, and the `PopSection` routine removes it. Both `PushSection` and `PopSection` routines update the section table: `PushSection` changes the pointer from the primary container to the newly created stack container and `PopSection` does the reverse.

Note
The container pointers in the section data structure are not modified by either of these routines. ◆

All section flags affect the operation of the `PushSection` and `PopSection` routines. A `PushSection` routine uses the Load flag, Clear flag, and Bank flags. The `PopSection` uses only the Save flag. The Static flag's impact on `PushSection` and `PopSection` operations is discussed next in "DemandCache for Static Sections."

When `PushSection` is called, a block of memory is allocated in the appropriate stack (selected by the Bank preference flags). If the Load flag is set, the contents of the primary container is copied to the stack. Otherwise, the allocated space is work space. If the Clear flag is set, the allocated stack space is cleared. In effect, you can specify that the work space be either cleared or not cleared with this mechanism. Both flags should not be set.

When `PopSection` is called, the stack space is copied back to the primary container if the Save flag is set. Then the stack space is deallocated.

If you access a dynamic section prior to a `PushSection` call or after a `PopSection` call, you will access the primary container in external memory. Accesses between the `PushSection` and `PopSection` calls will be directed to the section in the cache stack.

▲ **WARNING**
It is the programmer's responsibility to update the pointers after using the `PushSection` or `PopSection` routine. The DSP operating system only updates the section table, not the registers used by the programmer in the module. ▲

Notice that there are run-time errors that can occur with these routines. For `PushSection`, insufficient space in the stack or no Bank flags will cause an error. For `PopSection`, an error will result if the section space in the stack is not at the top of the stack. In either case, the DSP operating system will mark the task as inactive and send a message to the client indicating the type of error.

DemandCache for Static Sections

Static sections are used in DemandCache as a mechanism to provide efficient inter-module buffer management. This use is supported by the Real Time Manager and is one of the central services of the real-time data processing architecture. This connection mechanism allows the creation of generic modules that can be interconnected in many different ways for different purposes, without explicit programming. To support all possible cases, however, the module programmer must follow the rules for the I/O sections that are to be connected.

Static sections are allocated at module installation time by the Real Time Manager. This is very similar to the allocation of sections in the AutoCache model. When `PushSection` is called, the section table is updated with the primary container pointer. If the Load flag is set, the contents of the secondary container is copied to the primary container. Otherwise, the primary container is work space. If the Clear flag is set, the primary container is cleared. In effect, you can specify that the work space be either cleared or not cleared with this mechanism. Both flags should not be set.

If the Save flag is set when `PopSection` is called, the primary container is copied back to the secondary container and the section table is updated with the secondary container pointer. If the Save flag is not set, the section table is updated with the secondary container pointer only if the pointer is not `nil`. The pointer is `nil` for single container sections and contains a valid address for loaded sections.

To summarize, static sections work very much like AutoCache except that caching operations are carried out under the control of the module programmer rather than prior to and after module execution.

No run-time errors can occur with static sections.

Connections in DemandCache

It is possible to make connections to either dynamic or static sections using the Real Time Manager. If a connection is made to a dynamic section the primary container is shared with the other connected sections in local memory, external to the cache. This can be used to reduce local memory requirements or pass parameters between modules. This type of connection is made without specific module programming. In effect, the primary container pointer is changed to point to some other container and the original container is deleted.

For static sections, connections are made as with AutoCache, using two containers. Such connections are used to pass buffers between modules without the overhead of moving them off-chip and then back on-chip for a subsequent module. The section table contains the secondary container pointer until a `PushSection` call is made, after which it contains the primary container pointer. When a `PopSection` call is made, the section table is restored to the secondary container pointer.

For some types of connections, the secondary container is not required. For example, if an input section is already on-chip from a previous module, the Load flag is cleared and there is no use for the secondary container. In this case, the secondary container is

eliminated and the secondary pointer is set to `nil`. The primary pointer is placed in the section table in place of the `nil` secondary pointer. Notice that neither `PushSection` nor `PopSection` actually move memory in this case.

▲ **WARNING**

This means that the DemandCache programmer must always make the `PushSection` and `PopSection` calls in the code for static input and output sections. By making the calls, the correct function is executed, depending on the connections made. This makes the generic connection mechanism available to the DemandCache programmer. ▲

For example, suppose you have a simple module with a single input section (ignoring other sections). This input can be connected by the client that is installing the module into the task. Since it is a general-purpose input buffer, the Load flag and Static flags should be set.

If the client makes a connection to an output buffer from a previous module, and the buffer is already on-chip, then the Load flag will be cleared, and the primary container pointer will point on-chip to that buffer. The secondary container is deleted, and `nil` is stored in the secondary pointer. The primary container pointer is placed in the section table. When `PushSection` and `PopSection` are called, no action is taken. Notice that the program may access the buffer before the push operation or after the pop operation, if desired

If the client instead makes a connection to a previous module with an off-chip output buffer, the Real Time Manager will not clear the Load flag, and the secondary container will be valid. The secondary container pointer will be put into the section table. The `PushSection` routine will load the off-chip buffer into the on-chip primary container and update the section table pointer. The `PopSection` routine will restore the secondary container pointer in the section table. Accessing the secondary container prior to the push gives access to the input data off-chip (slow access). It does not make sense to access the data after the pop operation, since the data is not saved.

Hence, off-chip buffer accesses are not recommended prior to a push or after a pop operation. Since the location pointed to by the section table is certain to be on-chip only after the push and before the pop operation, accessing data on-chip can be guaranteed only between these routines.

Table 3-4 summarizes the flag combinations for DemandCache at run time.

Notice in Table 3-4 that although it is not illegal to specify or use a section that does not have any Bank flags set, it is always illegal to push or pop such a section. Other illegal combinations include static sections with no bank preference flags, and cached sections (Load flag or Save flag set) with no bank preference.

Table 3-4 Run-time DemandCache flag combinations

Flags						
Bank	Static	Load	Save	Clear	`PushSection`	`PopSection`
					Single-container dynamic case—Static flag is not set	
■	–	–	–	–	Allocate space in stack	Deallocate space in stack
■	–	–	–	■	Allocate and clear stack workspace	Deallocate space in stack
■	–	–	■	–	Allocate space in stack	Save to primary and deallocate
■	–	–	■	■	Allocate and clear stack workspace	Save to primary and deallocate
■	–	■	–	–	Allocate and load stack from primary	Deallocate space in stack
■	–	■	■	–	Allocate	Save to primary and deallocate
					Single-container static case—Static flag is set	
■	■	–	–	–	No operation	No operation
■	■	–	–	■	Clear primary	No operation
					Dual-container static case—Static flag is set	
■	■	–	■	–	Section table change only	Save primary to secondary
■	■	–	■	■	Clear primary	Save primary to secondary
■	■	■	–	–	Load primary from secondary	Section table change only
■	■	■	■	–	Load primary from secondary	Save primary to secondary
					Illegal cases—if no Bank flags are set	
	–	–	–	?	Off-chip section, push illegal	Off-chip section, pop illegal
–	■	?	?	?	Static illegal if no Bank flag set	Static illegal if no Bank flag set
–	?	■	?	?	Load illegal if no Bank flag set	Load illegal if no Bank flag set

continued

Table 3-4 Run-time DemandCache flag combinations (continued)

Flags						
Bank	Static	Load	Save	Clear	`PushSection`	`PopSection`
–	?	?	■	?	Save illegal if no Bank flag set	Save illegal if no Bank flag set
■	–	■	–	■	Useless/illegal	Useless/illegal
■	–	■	■	■	Useless/illegal	Useless/illegal
■	■	■	–	■	Useless/illegal	Useless/illegal
■	■	■	■	■	Useless/illegal	Useless/illegal

NOTE ■ = flag set, – = flag cleared, ? = do not care what flag is set to

FIFO Connections

It is possible to connect standard AIAO input and output buffers to a FIFO buffer. This can be done in one of two ways:

- Connect a FIFO section to an AIAO section with the `DSPConnectSections` routine.
- Create a FIFO for an AIAO I/O section with the `DSPNewFIFO` routine.

In the first case, the AIAO section is marked as a FIFO section, and the secondary container becomes the `DSPFIFO` data structure. One section should be an output section (Save flag set), and the other should be an input section (Load flag set). In the second case, a new `DSPFIFO` structure is created as the secondary container, and the AIAO section is marked as a FIFO section.

The DSP operating system recognizes these flag combinations, FIFO plus Load flag or FIFO plus Save flag, and does a `DSPFIFORead` or `DSPFIFOWrite` automatically, in place of a normal block copy. This allows the client to make connections with FIFOs without changing the module code. For AutoCache, the FIFO operations occur during the precache and postcache processes. For DemandCache, the FIFO operations occur during push and pop operations.

▲ **WARNING**
In DemandCache, it is inappropriate to access an input or output section before a push or pop operation, because a FIFO connection may have been made by the client. ▲

The `SetTaskInactive` flag in FIFO operations makes it possible to do useful things with FIFO connections. For example, it is possible to set up a sound player and connect its input to a sound FIFO. By setting the `SetTaskInactiveOnEmpty` and `SetTaskInactiveOnUnderrun` flags, the player will automatically stop when the buffer has played. A zero fill is done on the last `DSPFIFORead`, which creates silence at the end of the sound if it is not an integral number of frames in length.

Grouped Modules

AutoCache and DemandCache provide enough flexibility for most functions. If a job cannot be accomplished with AutoCache, it is almost certain to be possible with DemandCache.

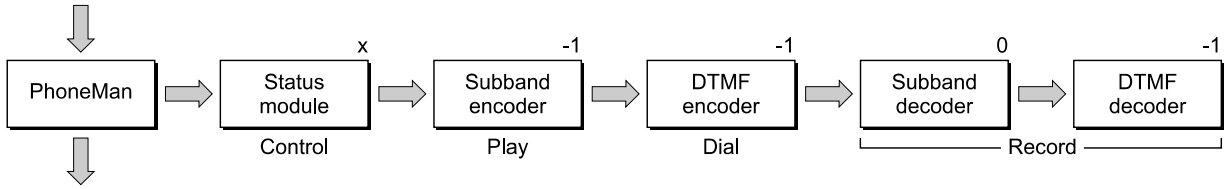
There are cases where having one big complex module to handle a complex case is not a good solution. This is where a set of already existing standard modules, if properly used, could get the job done. Another case is when automatic function replacement is desirable. For example, a system might consist of several very distinct functions, all implemented as separate modules. It is possible to replace a single module with a new and better one if the functions are separate rather than integrated into a single module. This is especially important if the modules were created by different programmers.

To support this type of building-block philosophy, a field called *skipcount* is added to each module. The skipcount number is normally set to zero, which indicates that the DSP operating system should proceed normally to the next module in the task or jump to the next task if the next module pointer is `nil`. The skipcount can also be set to –1, which means that the DSP operating system should exit the task at the completion of the module, ignoring any additional modules that may follow.

If the skipcount has any value from 1 up, it tells the DSP operating system how many modules to skip over in the current task before continuing module execution. If the DSP operating system runs out of modules in the task while skipping, it automatically exits the task, and begins the next one. This is not considered a run-time error. The skipcount from one task cannot affect a following task.

Here is an example of how the skipcount can be used in a telephone answering function, where a status and control module is required. The module is responsible for detecting rings, taking the phone line offhook, and hanging up the phone. Other functions needed include a recorder function, a play message function, and a dial function. The recorder function may need a DTMF decoder to detect remote control functions. The recorder may use a compression module, and the player may use a decompression module. One possible arrangement of these modules is shown in Figure 3-29.

Figure 3-29 Example of DSP task for telephone answering



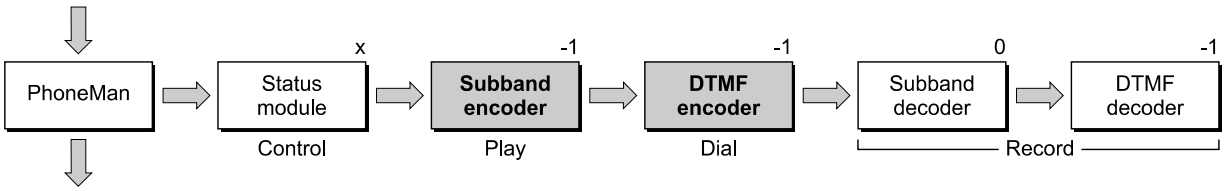
In this example, the status module contains the control function. It normally has a skipcount of –1 when the answering machine is idle. When a play function is needed the skipcount is set to 0. When the dial function is needed the skipcount is set to 1. When the record function is needed the skipcount is set to 2. The fact that the skipcount has several possible values is indicated with an “x” in Figure 3-29.

The skipcount can be set initially by calls to the Real Time Manager. Since the count is normally 0, the value –1 should be stored in the status module, subband encoder, DTMF encoder, and subband decoder modules. The last module count does not matter but should be set to –1. The status/control module will change its own skipcount as needed using the DSP operating system routines provided.

GPB for Grouped Modules

The telephone-answering task described in the last section never actually uses all of the modules within it at the same time. Hence it would be inappropriate to add up the GPB requirements for all the modules and allocate that much time for this function. A flag, called `DontCountThisModule`, is included in the module to handle this case. If set, it tells the Real Time Manager not to include the GPB of a module in the total GPB requirements for the task. It is up to the client that is installing this group to determine which is the worst-case utilization of modules, and to mark all of the other modules with the flag. In the example, the record case is probably the worst case. Therefore, the dial and play modules should have their `DontCountThisModule` flags set. This is shown in Figure 3-30, where modules with the flag set are shown with shading. Only the unshaded modules will have their GPB requirements added to the task total.

Figure 3-30 Controlling GPB in grouped modules



Module Scaling

The Macintosh real-time data processing architecture makes it possible to use modularized digital signal processing functions in different configurations, reducing or eliminating the need to rewrite DSP modules whenever a slightly different operation is required. To accomplish this, the architecture provides unified I/O buffers and connections to make up new applications from existing modules. It also includes the ability to scale a module's buffers to accommodate different frame rates, sample rates, and GPB requirements.

It is possible for a client to encounter one of three sample rates for sound or telecom, plus one of two (at least) frame rates. It would be unfortunate if a different module was required for each of the possible cases. The real-time data processing architecture solves this problem by providing a means of scaling any module to fit the current frame and sample rate, as well as select the appropriate GPB estimate for the module's functionality.

There are several characteristics that change for a module when either the frame rate or sample rate change. First, the required GPB number changes. A module that takes 5000 GPB units to execute at 100 frames per second will only take 2500 GPB units at

200 frames per second. The second characteristic that changes is the number of repeat loops the module must execute to complete processing for a frame. Again, at the 200 frames per second rate only half as many samples must be processed or generated per frame. A module may also change its GPB requirements based on an outside loading factor.

There are some types of modules that normally do not operate at different frame rates or sample rates—for example, a DTMF decoder. This is because a DTMF decoder looks for specific frequencies and must assume a sample rate. A more advanced version of a DTMF decoder could be created that would adjust automatically to the sample rate.

A good example of a module that can easily function at various sample rates and frame rates is a sample rate converter. If the converter has a fixed conversion rate, it can operate just as well on 32 kHz data as 8 kHz data. Likewise, it can operate just as well on 240 samples per frame as 120 samples per frame. A more advanced sample rate converter would be able to adjust automatically to changes in the input frequency by changing the conversion factor. This capability requires that a module be able to request more or less GPB even though the sample rate and frame rate have remained fixed.

The Real Time Manager provides a mechanism for the module programmer to specify scalability, whenever it is possible. This is done with the use of *scaling vectors* and the GPB mode (AutoCache or DemandCache). A scaling vector contains three values: the frame rate, the scale factor, and the GPB value. Any number of scaling vectors can be supplied for a module. They work for both AutoCache and DemandCache modules. In addition, a scaling vector can have one frame rate and scale factor with multiple GPB estimates.

The module code itself determines the number of samples it has to work with by using the `GetSectionSize` macro. By using this on its input buffer, for example, the module can determine the repeat count of its processing loop. Generally, this is the only information required by the module. It contains both sample rate and frame rate information in a single number. The GPB mode must be specified if the module has variable GPB requirements.

The frame rate value is used to select the appropriate scaling vector. The scale factor is used to determine the size of the scalable I/O buffers for the module.

The scalability of I/O buffers is indicated by setting the `ScalableSection` flag. This flag should be set only for AIAO I/O buffers. The Real Time Manager determines the actual size of the section for a given scaling vector by multiplying the size set in the source code by the scaling vector's scale factor.

In a 2 to 1 sample rate converter, for example, scaling vectors might be generated for the following cases:

- 100 frames/sec. 24 kHz to 12 kHz
- 200 frames/sec. 24 kHz to 12 kHz
- 100 frames/sec. 8 kHz to 4 kHz

- 200 frames/sec. 8 kHz to 4 kHz
- 100 frames/sec. 16 kHz to 8 kHz
- 200 frames/sec. 16 kHz to 8 kHz
- 100 frames/sec. 48 kHz to 24 kHz
- 200 frames/sec. 48 kHz to 24 kHz
- 100 frames/sec. 32 kHz to 16 kHz
- 200 frames/sec. 32 kHz to 16 kHz

Note
The only information the module would need to operate at any of these rates is the amount of cache space and the size of either the input or output buffer. All of the required information to support these 10 cases can be supplied by 10 scaling vectors. ◆

Since this module is a 2 to 1 converter, the size of the input buffer in the source code should be set to 2, and the output buffer should be set to 1. The 10 scaling vectors would look like this:

Mode 0	100, 120, 5000	[100 f/s, scale = 120 (I/O size of 240/120), GPB = 5000]
Mode 1	200, 60, 2500	[200 f/s, scale = 60 (I/O size of 120/60), GPB = 2500]
Mode 2	100, 40, 1666	[100 f/s, scale = 40 (I/O size of 80/40), GPB = 1666]
Mode 3	200, 20, 833	[100 f/s, scale = 20 (I/O size of 40/20), GPB = 833]
Mode 4	100, 80, 3333	[100 f/s, scale = 80 (I/O size of 160/80), GPB = 3333]
Mode 5	200, 40, 1666	[200 f/s, scale = 40 (I/O size of 80/40), GPB = 1666]
Mode 6	100, 240, 10000	[100 f/s, scale = 240 (I/O size of 480/240), GPB = 10,000]
Mode 7	200, 120, 5000	[200 f/s, scale = 120 (I/O size of 240/120), GPB = 5000]
Mode 8	100, 160, 6666	[100 f/s, scale = 160 (I/O size of 320/160), GPB = 6666]
Mode 9	200, 80, 3333	[200 f/s, scale = 80 (I/O size of 160/80), GPB = 3333]

Notice that there are several cases where the same values for GPB and scale factor are used, but a different frame rate is selected. This is necessary to provide a mechanism for the module programmer to exactly specify the conditions under which the module will operate. There may be cases where the module will not operate correctly when run at a different frame rate with the same GPB and scale factor. An example of this is a module that is designed to operate at specific frequencies, such as a DTMF decoder.

Note
In the example, the GPB numbers are neatly set to round numbers. In reality, the numbers may not be so round, because of bus and DSP operating system overhead. ◆

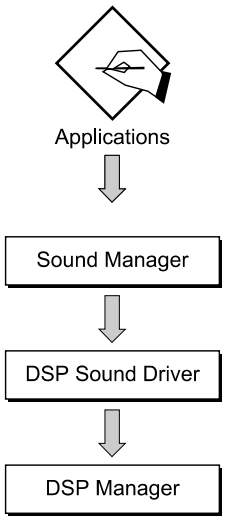
Selecting Module Scale Factor

The process for selecting the scale for a module is simple. When the client wishes to load a module, a Real Time Manager call to `DSPLoadModule` is made. One of the parameters provided to the Real Time Manager is the scale factor. The current DSP frame rate is provided automatically. If there is a matching scaling vector, the scalable sections are scaled appropriately and the correct GPB is used. If there is no matching scaling vector, an error is returned. The client must make a selection based on the required I/O buffer sizes. The associated scale factor is then passed to the loader.

Standard Sound

Standard sound consists of a set of tasks installed in the real-time task list plus the DSP Sound Driver. The Sound Driver acquires control over the sound port, installs and maintains the sound tasks, and handles the interface to the Macintosh Sound Manager. The goal of the Sound Driver is to provide DSP sound support to all applications that do not use the DSP, and to provide a plug-board architecture for sound applications that use the DSP. The relationship between these different layers of software is shown in Figure 3-31.

Figure 3-31 DSP Sound Manager and Sound Driver



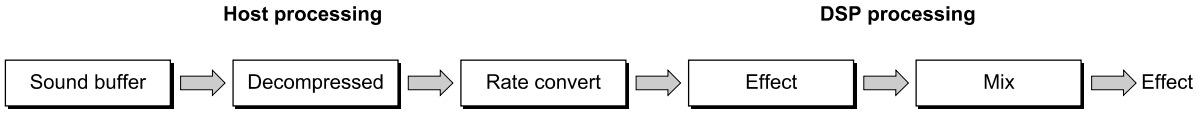
The function of the Sound Driver interface is to provide a hardware-independent interface to the Macintosh Sound Manager that can accept requests for work to be done. The interaction between available GPB and work requests is handled automatically by the Sound Driver. If there is insufficient DSP processing time available, the Sound Driver

rejects the work request, and the Sound Manager does the work itself. This mechanism prevents a heavily loaded DSP from preventing essential sound functions from operating. The DSP is used if it is available, and normally results in better sound.

Sound Manager Interface

The Sound Manager approach is to provide a series of blocks for processing sound from the raw data to the output. This diagram is shown in Figure 3-32.

Figure 3-32 Sound Manager processing



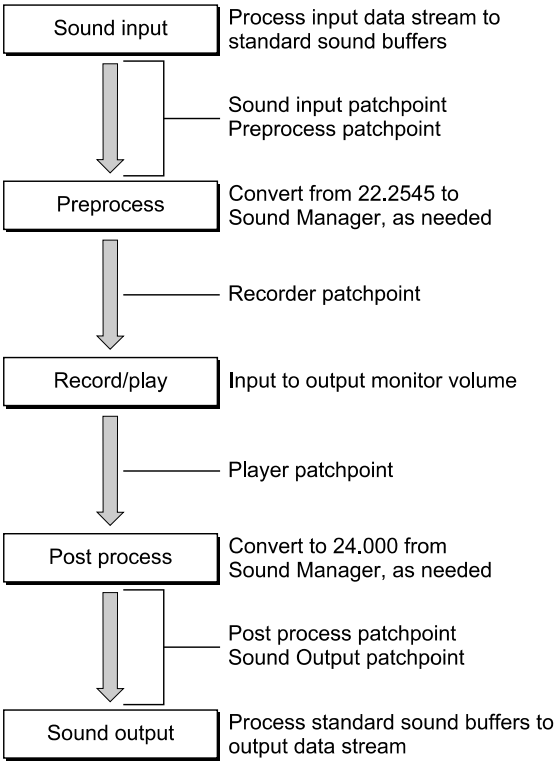
The dividing line between 68000 processing and DSP processing moves from right to left in Figure 3-32, depending on the amount of DSP processing available; the more DSP processing available, the farther left the line goes. The mix and speaker output is always done in the DSP. However, if additional channels of sound are required, and the DSP is fully loaded, the additional channels will have to be "premixed" before arriving at the DSP. If sufficient DSP processing is available, the entire processing chain for each channel will be handled by the DSP. The processing of each channel is done by a task installed in the player patch point, described in the next section. In the example above, the task consists of a decompressor module, a sample rate converter module, and a special effects module. All modules are interconnected via AIAO buffers. The input buffer for the decompressor is a FIFO and the output buffer is the sound output buffer.

Standard Sound Task List

The standard sound task list consists of five separate tasks, as shown in Figure 3-33. The first and last task are responsible for handling the serial port data stream and for opening and closing two ITBs that pass stereo data in DSP floating-point format between all five of the tasks. These buffers are used to provide the standard sound patch points.

Note
The standard sound tasks are not named to represent what the task does but rather are named for the patch points associated with them. ◆

Figure 3-33 Standard sound task list



The sound input task takes the serial data stream from the DMA channel coming from the stereo A/D converter and converts it to DSP floating-point format. For 10 ms frames and 24 kHz sound, this takes the form of two 240-longword buffers. These buffers are established as ITBs by the DSP map structure associated with sound input.

If an application wishes to get access to the raw input data stream, it installs a task after the sound input task. Such a task should not modify the sound input.

The preprocess task provides a stereo or mono data stream to the sound input at any of the standard 8-bit rates of 7, 11, and 22 kHz, and 16-bit sound at the current rates of 24, 32, or 48 kHz.

If an application wishes to preprocess the sound datastream prior to it reaching any recorder tasks, it installs a task prior to the preprocess task.

The record/play task handles the monitor function of the standard sound plug board. This function allows none or all of the input sound to be passed to the output sound.

If an application wishes to record sound, it installs a task before the record/play task. If an application wishes to play sound, it installs a task after the record/play task.

The postprocess task accepts a mono or stereo sound channel from the Sound Manager at the standard 8-bit rates of 7, 11, and 22 kHz, or 16-bit sound at the current rates of 24, 32, or 48 kHz.

If an application wishes to postprocess the mix from all players, it installs a task after the postprocess task.

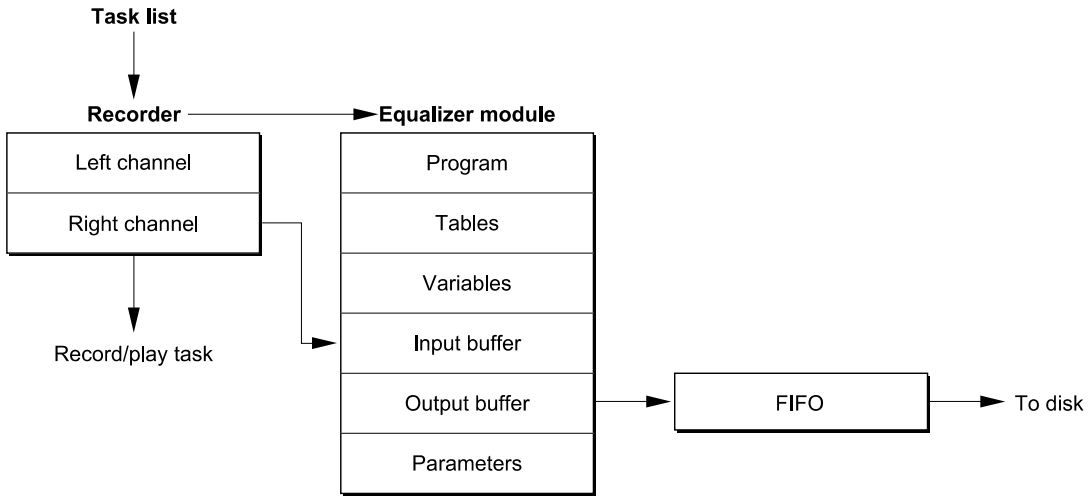
The sound output task accepts the final mixed and processed output data stream in DSP floating-point format. It takes care of the system volume control function, converts the data to the serial port format, and transports it to the sound output buffer. The DMA channel passes this data to the sound D/A converters.

If an application wishes to access the data stream after all players and postprocessing, it installs a task before the sound output task. Such a task should not modify the output sound stream.

The operation of the standard sound plug-board depends on the function of AIAO buffers and the Real Time Manager. The two sound ITBs are opened by the sound input task, and passed to each task in the list. The sound output task closes the ITBs. This operation is covered in "Buffer Connections Between Tasks," earlier in this chapter.

There are two different ways a sound module can be installed in this task list. For these examples, assume the application is installing a five-band active equalizer module (forming one task). This module can be installed as a preprocessor or part of a more complex preprocessor function, as a recorder or part of a recorder, as a player or part of a player, and as a postprocessor or part of a postprocessor. Figure 3-34 shows how the module can be used as a recorder.

Figure 3-34 Equalizer used as a recorder task



In this example, assume that the equalizer is monophonic. Its input buffer is a scalable AIAO buffer, and its output buffer is a scalable AIAO buffer. This means that the Load flag is set for the input buffer, and the Clear flag and Save flags are set for the output buffer. These buffers were created using the `NewScalableInputBuffer` and `NewScalablePRBOutputBuffer` macros.

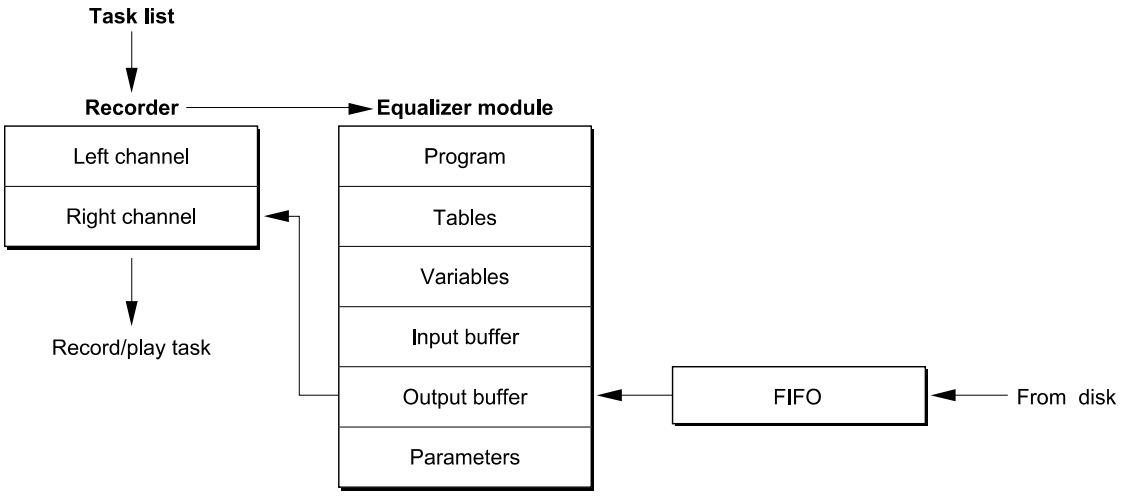
The application uses the Real Time Manager to make a connection between the right channel ITB (shown as part of the record/play task) and the input buffer section of the equalizer. Since the ITB already exists on-chip, the Load flag is cleared by the Real Time Manager, and the input buffer primary container is made to be the same as the right channel buffer. This completes the input connection.

The output buffer section is connected to a FIFO, using the `NewFIFO` routine. The application uses this FIFO to buffer the data to the disk. In this case neither the Clear flag or Save flags are cleared by the Real Time Manager.

When executing, the output buffer is cleared, the equalizer reads its input data from the right channel buffer, and then the equalizer sums its output into the output buffer. Finally, the DSP operating system writes the buffer to the FIFO. The data recorded is the same data format and sample rate as the input data, but has been processed through the equalizer filters.

The next example connects the same module as a player. A player task is installed after the record/play task, as shown in Figure 3-35.

Figure 3-35 Equalizer used as a player task

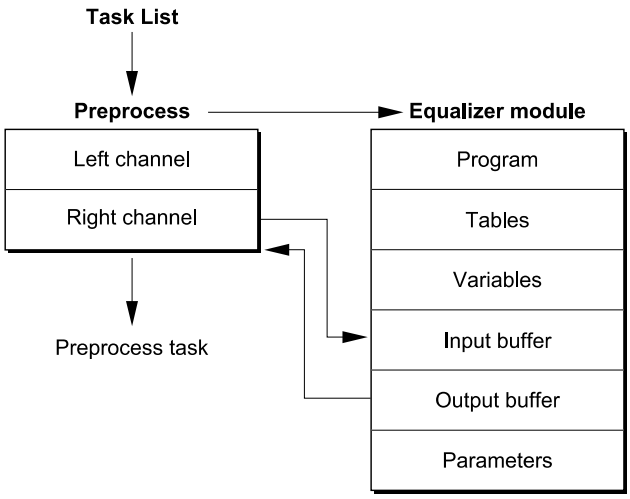


When the connection is made from the output buffer to the right channel buffer, the Clear flag is cleared, since this module is not the first to input to the right channel buffer. Likewise, the Save flag is cleared, since the right channel buffer already exists on-chip. Thus, any already existing signal in the buffer is directly summed into by the equalizer. This is a normal or *direct connection*.

Note
The function of this buffer as a summing buffer is a fundamental property of the standard sound plug board. ◆

In Figure 3-36, the equalizer task is used as a sound preprocessor.

Figure 3-36 Equalizer used as a preprocess task



In this case, both the input buffer and the output buffer are connected to the same ITB. If the normal summing connection was made between the output buffer and the right channel buffer in this case, the desired result would not be achieved. Rather, the sum of the signal and the filtered signal would be in the right channel buffer. In this case, the client must make an *indirect connection* between the two buffers. This type of connection is selected with one of the `ConnectSections` routine parameters.

An indirect connection is the same as a direct connection, except a buffer move is required—either the Save flag of the source buffer or the Load flag of the destination buffer must be set (or not cleared). The Clear flag of the source buffer must be left set. This results in the equalizer summing into a cleared buffer, followed by the DSP operating system saving the filtered sound into the right channel buffer, overwriting the previous sound data.

The final example is to use this same equalizer as a postprocessor. The layout would look exactly like Figure 3-36, except the location of the task would be different.

These examples show the versatility of the standard sound plug-board architecture. In addition, the utility of the Real Time Manager to create connections between FIFOs and AIAOs, and to provide both direct and indirect connections, allows the standard sound modules to be used in very flexible ways.

Sample Rate and Frame Rate Changes

There are two factors that affect the real-time DSP processing of sound: the sample rate of the sound I/O subsystem, and the frame rate of the DSP operating system. While it is possible to handle these two factors separately, it is more convenient to handle them together. This simplifies the standard sound task list.

This approach offers three different “gear shifts” for sound:

- standard sound (24 kHz sample rate, 10 ms frame rate)
- high fidelity sound (32 kHz sample rate, 5 ms frame rate)
- professional sound (48 kHz sample rate, 5 ms frame rate)

The advantage of this approach is that there is sufficient room in the cache to support all of these selections, including the dual intertask buffers. If 48 kHz was supported at 10 ms frames, for example, the buffers would be too big to both fit in the cache along with the DSP operating system and module code.

CHAPTER 4

Real Time Manager

This chapter describes the Macintosh system software routines for accessing, controlling, and monitoring the digital signal processor (DSP) subsystem in the Macintosh Quadra 840AV and Macintosh Centris 660AV. The DSP subsystem provides real-time processing for applications that require a guaranteed throughput. It also provides processing for applications that perform timeshare processing.

Before you read this chapter you should already be familiar with

- Macintosh programming
- using resource files
- the concepts of digital signal processing given in Chapter 3, "Introduction to Real-Time Data Processing"

A brief synopsis of the Real Time Manager is provided first. Valuable information about getting started with the DSP is included in "About the Real Time Manager." This is followed by a top-down explanation of how an application accesses the different levels of the DSP architecture. Next is an example showing the necessary commands in the order they would probably be used. Optional commands are included to provide you with programming alternatives. At the end is a Real Time Manager Reference, organized in top-down order. Routines are listed for each level of the Real Time Manager in logical groupings.

For information about installing and debugging DSP programs in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Appendix A, "DSP d Commands for MacsBug," Appendix B, "BugLite User's Guide," and Appendix C, "Snoopy User's Guide."

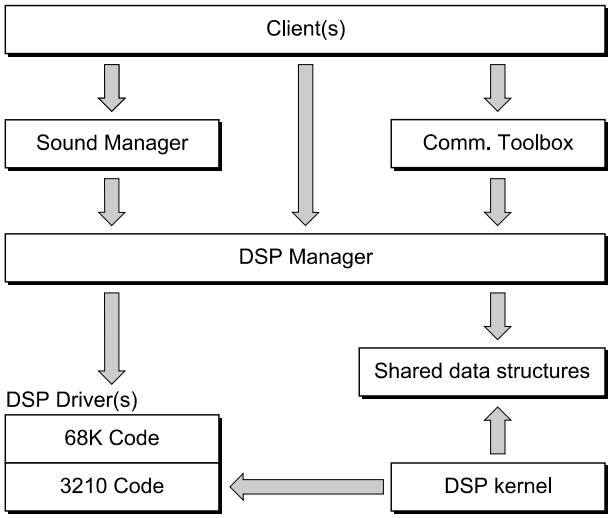
About the Real Time Manager

The Real Time Manager provides access to the capabilities of the Macintosh DSP subsystem. DSP code modules for performing the desired data manipulations must already be available as resources. Apple provides a standard set of DSP code modules as resources for performing sound and telecommunications input and output. These functions include data compression, sample-rate conversion, disk store and recall, and playing sound files directly from disk. Examples are described in "Standard Sound Task List," in Chapter 3.

Real Time Manager Structure

Figure 4-1 gives a high level architectural view of the DSP subsystem. To maintain implementation independence, the Real Time Manager must perform all operations on the DSP subsystem. Programmers wanting to maintain compatibility should always use the Sound Manager and communications toolboxes provided by Apple. For maximum performance the Real Time Manager is accessible directly. However, error handling should be included for use on computers without the DSP.

Figure 4-1 DSP subsystem overview



To determine if the DSP subsystem is available use the Gestalt command. See "Machine Identification," in Chapter 1.

To check compatibility with the version of the Real Time Manager installed in the system, the `DSPManagerVersion` routine must be called. This routine should always be called early in the application to determine if its DSP tasks can be installed.

```
pascal unsigned long DSPManagerVersion(void)
```

The Sound Manager includes code that sets up and maintains a set of standard system tasks to support input and output for the stereo codec and mono/stereo sound player, as well as multiple sample rates and sample rate changes. Calls made to the Sound Manager are always performed by the system. If there is no DSP subsystem available, the system performs the desired function using the main processor.

The Communications Toolbox handles similar functions for the telephone and data communications serial port. Calls made to the Communications Toolbox will only be performed if a DSP subsystem is available.

Memory allocation routines in the Real Time Manager handle the DSP's on-chip memory, module bandwidth allocation, and intertask buffer (ITB) allocation. Since these routines are specifically for the DSP subsystem there are no equivalent system routines.

All blocks of memory indicated by a value of type `DSPAddress` are by definition locked contiguous and non-cacheable. They are locked contiguous so that the DSP does not have to worry about scatter/gather operations when using a `DSPAddress` value. The blocks are locked non-cacheable to eliminate conflicts that would occur when the DSP modifies a memory location that the host processor had cached. Since the DSP can only address physical memory, it cannot use virtual memory.