Chapter 6
Array Processing

An *array processor* is a digital hardware device capable of performing mathematical computations on arrays of data with tremendous efficiency and speed. This chapter outlines the design of C programs that use the Motorola DSP56001, or *DSP*, as an array processor. In addition, the C functions provided by the NeXT DSP system and array processing libraries are summarized, and instructions for creating your own array processing functions (from DSP56001 assembly language macros) are given.

Digital signal processing is a type of array processing that performs computations on signal data. Signals are typically one-dimensional streams produced by measuring the changes in physical phenomena, such as sound, over a period of time. The DSP is used as a signal processor when, for example, it synthesizes music or converts sound data for playback. Real-time signal processing with the DSP is discussed at the end of this chapter.

Design Philosophy

Traditionally, array processors have been employed in fields such as weather forecasting and other physical modeling pursuits to perform "number crunching" on large amounts of data. To attain the speed necessary to process these mountains of data within a tolerable time, array processors use parallel hardware and a computational pipeline in which numerical operations, data input/output, program fetching, and address updating are all done in parallel. Furthermore, an array processor is generally provided as a peripheral device with its own private high-speed memory, allowing it to work efficiently and exclusively on the task at hand. After initiating a process on an array processor, the host processor is free to continue with other chores, unburdened by the array processing operations.

The fundamental difference between array processing and more conventional mathematical computing is the explicit use of arrays—rather than individual numbers—as the primitive objects upon which the computations are performed. The speed and parallelism provided by an array processor are made possible by the use of arrays: Because array elements are stored contiguously, they can be fetched in parallel with numerical computations, and the address of the next element can be automatically computed.

NeXT computers use the Motorola DSP56001 as a general-purpose, fixed-point array processor. This state-of-the-art microprocessor can execute up to 12.5 million instructions per second and, with a single instruction, can perform a 24- by 24-bit fixed-point multiply, a 48- plus 56-bit addition, two 16-bit address updates, and three parallel memory moves. In addition, 8K 24-bit words of zero-wait-state static RAM are available to the DSP for private data storage. Thus, the DSP provides the architectural features of an array processor.

Fundamental array processing operations, such as vector add and matrix multiply, are provided as C functions in the NeXT array processing library. A key design element is the extensibility of this software. It's a simple matter to create your own array processing function from a DSP assembly language macro by using the **dspwrap** utility. Creating your own macro requires familiarity with DSP assembly language; to help acquaint you with this language, NeXT provides a number of macros in source code form—many of which were used to generate the array processing functions—as programming examples. You can write and wrap your own DSP macros, or you can combine the macros provided by NeXT to create more complex and efficient array processing functions.

Creating an Array Processing Program

Programs that access the DSP as an array processor follow a basic five-step design:

1. Initialize the DSP.
2. Transfer data from host memory to DSP memory.
3. Download and perform array processing functions on the data in the DSP.
4. Transfer the processed data back to host memory.
5. Free the DSP.

The DSP is assigned to a single program until it's done processing that program's data. Initializing the DSP will assign it (step 1); no other program will be able to use the DSP until it's freed (step 5). This means, for example, that you can't launch an application that synthesizes music or that depends on the DSP for sound playback conversion while you're running your array processing program.

A program running on the host processor not only sends data to the DSP (step 2), it also sends instructions for processing the data (step 3). After the DSP has completed its processing, the program must ask for the processed data to be sent back to the host (step 4).

The following program fragment demonstrates the five main steps involved in an array processing program:

```
/*
 * Perform array processing operations on two arrays (a[] and b[]),
 * returning results to a third array (c[]).  To compile:
 *     cc thisFile.c -larrayproc -ldsp_s -lsys_s
 */

#import <dsp/arrayproc.h>
#define N 200                   /* number of elements in each array */
#define AADR DSPAPGetLowestAddress()  /* address of a in DSP memory */
#define BADR (AADR+N)           /* address of b in DSP memory */
#define CADR (BADR+N)           /* address of c in DSP memory */
#define INC 1                   /* element increment for all arrays */

main() {
/* The arrays are declared to contain data of type float. */
    float  aArray[N], bArray[N], cArray[N];

    /* Place initial values in aArray[] and bArray[]. */

    . . .

    /* Step 1, initialize the DSP. */
    DSPAPInit();

    /* Step 2, transfer the data arrays to the DSP. */
    DSPAPWriteFloatArray(aArray, AADR, INC, N);
    DSPAPWriteFloatArray(bArray, BADR, INC, N);

    /*
     * Step 3, download and perform array processing functions on the
     * data.  As an example, the vector plus vector function is used
     * here.
     */
    DSPAPvpv(AADR, INC, BADR, INC, CADR, INC, N);

    /* Step 4, return the result to the host. */
    DSPAPReadFloatArray(cArray, CADR, INC, N);

    /* Step 5, release the DSP. */
    DSPAPFree();

    /* Do something interesting with cArray[]. */
    . . .
}
```

Programming Examples

The directory **/NextDeveloper/Examples/DSP/ArrayProcessing** contains example array processing programs in the following subdirectories:

apsound	Time-reverse a sound file.
matrix	Multiply two matrices.
fdfilter	Perform convolution in the frequency domain.

In addition, the subdirectory **libap** demonstrates how to create your own array processing library, and the **fuse**

subdirectory illustrates the “fusing” of several array processing macros into a single C function.

DSP Memory Map

Figure 6-1 and Figure 6-2 show the layout of DSP memory for array processing. The actual addresses delimiting each memory region are in the header file **dsp/dsp_memory_map_ap.h**. Generally, the array processing monitor occupies the lowest and highest addresses in each space. On-chip p memory is nominally reserved for the array processing program itself, on-chip x memory is for array processing function arguments, and on-chip y memory is available for the user. Note, however, that the array processing functions provided by NeXT use only the x memory space for all data arrays.

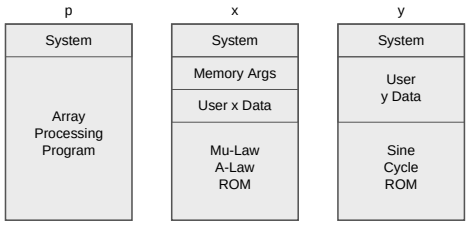

Figure 6-1. DSP Internal Memory Map

There are two images of external memory:

- In image 1 (shown below in Figure 6-2), external memory appears as one giant array that can be carved arbitrarily into x, y, and p segments.
- In image 2 (not shown), the external x and y memory banks are physically separate, as they are on the chip.

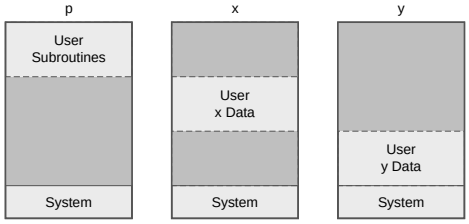

Figure 6-2. DSP External Memory Map (Image 1)

More information on the physical memory map is provided in Appendix F of *Reference*.

Data Formats

The DSP stores data in a 24-bit, fractional, two’s complement, fixed-point number representation that’s declared on the host as type DSPFix24. A DSPFix24 number with bits $b_0, b_1, ..., b_{23}$ (each bit being 0 or 1) has the value

$$-b_0 + b_1 / 2 + b_2 / 4 + ... + b_{23} / 2^{23}$$

The minimum representable value is −1 (binary 10...0), and the maximum value is $1-2^{-23}$, which is often referred to as “1 minus epsilon” (binary 01...1). If n denotes a DSPFix24 number, then $-1.0 \le n < +1.0$. Inside the DSP, numbers are routinely clipped to the interval [−1,1) as they’re transferred from one of the 56-bit accumulators to 24-bit memory. It’s left to the programmer to ensure that computations in the DSP don’t create values outside this range (other than temporarily within an accumulator).

Since a DSPFix24 number n is within the range $-1.0 \le n < +1.0$, a C **float** number f destined for the DSP must also be

within $-1.0 \le f < +1.0$. A C **int** number i must be within the range $-2^{23} \le i < 2^{23}$, or between −8388608 and 8388607. On the host, a DSPFix24 number is stored as an **int;** 24 bits are right-justified in 32 bits and the sign extension isn't required.

Complex Array Format

Array processing functions that use complex data interleave the real and imaginary parts of the data. This means that arrays of complex numbers are stored in a single DSP memory space as $[x(1), y(1), x(2), y(2), ..., x(N), y(N)]$, where $x(i)$ is the real part and $y(i)$ is the imaginary part of the ith element of the length N complex array.

The one exception to this is the Fast Fourier Transform function, **DSPAPfftr2a()**, which requires that the real part be stored in x memory and the imaginary part in y memory.

DSP System Library

The DSP system C library, **/usr/lib/libdsp_s.a**, provides generic data transfer and conversion functions that support both array processing and the sound synthesis operations of the Music Kit. In addition, it lets you boot the DSP with an arbitrary monitor and provides functions for reading and writing the host interface, as described in Chapter 5, "Programming the DSP."

The header file **dsp/dsp.h** provides procedure prototypes (either directly or by including other such header files) for the functions in the DSP system library. The DSP system functions have the prefix "DSP".

Data Format Conversion

Data on the host must be converted to DSPFix24 format before it's transferred to the DSP. Similarly, data returned to the host by the DSP is in this format and should be converted to a data type, such as **float** or **int**, that's useful to your program.

System functions that transfer data to or from the DSP, such as **DSPAPWriteFloatArray()** and **DSPAPWriteIntArray()**, do the necessary data type conversions for you. However, for greater control you can convert and transfer data by calling two separate C functions: one that does the conversion and another that performs the transfer. The following example shows separate DSP system function calls for data conversion and data transferral:

```
/*
 * Program example demonstrating the separation of data conversion
 * from data transferral.
 */

#include <dsp/arrayproc.h>
#define N 200
#define AADR DSPAPGetLowestAddress()
#define BADR (AADR+N)
#define CADR (BADR+N)
#define INC 1

main()
{
    /* Arrays for float data. */
    float  aFloat[N], bFloat[N], cFloat[N];

    /* Analogous arrays for DSPFix24 data. */
    DSPFix24 aFix[N], bFix[N], cFix[N];

    /* Place float data in aFloat[] and bFloat[]. */
    . . .

    /* Initialize the DSP. */
    DSPAPInit();

    /* Convert data from float to DSPFix24 format. */
    DSPFloatToFix24Array(aFloat, aFix, N);
    DSPFloatToFix24Array(bFloat, bFix, N);

    /* Transfer DSPFix24 data to the DSP without conversion. */
    DSPAPWriteFix24Array(aFix, AADR, INC, N);
    DSPAPWriteFix24Array(bFix, BADR, INC, N);
```

```
        /* Perform array processing functions on the data. */
        . . .

        /* Return result data to the host and put it in c[]. */
        DSPAPReadFix24Array(cFix, CADR, INC, N);

        /* Convert DSPFix24 to float, if necessary. */
        DSPFix24ToFloatArray(cFix, cFloat, N);

        /* Free the DSP. */
        DSPAPFree();

        /* Do something interesting with cFloat[]. */
        . . .
    }
```

DSPFloatToFix24Array() and **DSPFix24ToFloatArray()** are data conversion functions that take three arguments:

- Data is read from an array in host memory given as the first argument.
- The data is converted and then written to a host array given as the second argument.
- The third argument is a count of the number of elements in either array.

Analogous functions are provided to convert to and from type **int**, and to and from type **double**. Keep in mind that the data conversion functions don't scale the data for you; values that aren't within the bounds described in the section "Data Formats" above are clipped.

Note: int arrays can be cast to type DSPFix24 and written to the DSP without explicit conversion; the uppermost byte of each **int**—which, if properly scaled, should only contain the sign extension—is ignored during the transfer. Similarly, if data that's retrieved from the DSP is to be interpreted as nonnegative integers, it isn't necessary to convert from DSPFix24 to **int**. On the other hand, negative integers from the DSP require sign extension in the uppermost byte, as provided by **DSPFix24ToIntArray()**.

Data that's converted with a data conversion function can be transferred with the functions **DSPAPWriteFix24Array()** and **DSPAPReadFix24Array()**, as shown in the example. These are the fastest, lowest level DSP array transfer functions—no data format conversion is carried out before or after the transfer.

The procedure prototypes for the conversion functions are in **dsp/DSPConversion.h**.

Note: In addition to the DSPFix24 data type, the DSP also accepts 16-bit and 8-bit numbers, but these exist primarily for the Sound Kit. The Sound Kit uses the 16-bit format to process sound for CD-quality output and the 8-bit format for the mu-law encoded voice-quality sound input.

Array Processing Library

The array processing library, **/usr/lib/libarrayproc.a**, contains C functions that provide many of the elementary array operations needed for array processing (and signal processing) applications. In addition, the library provides functions that help communicate with and control the DSP in a manner that's best suited for array processing needs.

The array processing library functions—except those with prefix "DSPAPGet", as described below—return an error code that's 0 for success and nonzero for failure. The nature of the error is displayed if the **DSPEnableErrorFile()** function has been called, as explained in Chapter 5.

All the functions in the array processing library have the prefix "DSPAP".

System Support Functions

System support functions for array processing provide DSP control, data transfer between DSP memory and host memory, and error handling.

DSP Control Functions

DSP control functions manage the acquisition of the DSP, providing exclusive access to the DSP chip. The most important

of these are **DSPAPInit()** and **DSPAPFree()**:

- **DSPAPInit()** acquire and initializes the DSP for array processing.
- **DSPAPFree()** releases and resets the DSP.

The other DSP control functions, listed in the header file **dsp/DSPControl.h**, needn't be called directly if you're using **dspwrap**-generated functions. However, you can use them to implement advanced features, such as stacking of array processing functions (which are all relocatable modules) and overlapping of data transfers and program execution on the DSP.

Data Transfer Functions

These functions provide array copying between host memory and DSP memory. The functions operate on specific data types or configurations and all come in pairs, one for sending (writing) an array from the host to the DSP and another for retrieving (reading) data from the DSP to the host. They each take four arguments:

- A pointer to the array on the host
- The address of the array on the DSP
- A "skip factor," the DSP address increment used in the transfer
- The number of array elements to transfer

The data type of the first argument depends on the array that's being written or read. The other three arguments are **int**s. Keep in mind that the skip factor always applies to the array in DSP memory, whether you're reading or writing. For example, if you write an array with a skip factor of 3, the first element from the host array is written as the first word in the DSP array, the second element from the host is the fourth word on the DSP, the third host element is the seventh DSP word, and so on. Similarly, if you read an array with a skip factor of 3, the first host element is taken from the first DSP word, the second host element from the fourth DSP word, and so on. To write and read contiguous elements, use a skip factor of 1.

There are seven write/read pairs of transfer functions, representing seven different data types or configurations:

- **DSPAPWriteFix24Array()** and **DSPAPReadFix24Array()** transfer arrays of type DSPFix24. These functions are also used to transfer unpacked byte arrays. Array data on the host is right-justified in 24 bits on the DSP.
- **DSPAPWriteIntArray()** and **DSPAPReadIntArray()** transfer arrays of type **int**. The write function is the same as that for DSPFix24 arrays. The read function, on the other hand, is different from the DSPFix24 read function: The 24-bit values received from the DSP are sign-extended.
- **DSPAPWritePackedArray()** and **DSPAPReadPackedArray()** transfer arrays of data packed in type **unsigned char**. Each successive three bytes of host data is transferred to or from a single word on the DSP.
- **DSPAPWriteShortArray()** and **DSPAPReadShortArray()** transfer arrays of 16-bit data packed in type **int**. These are used primarily for processing sound data; each 32-bit word in the source array provides two successive 16-bit samples in the DSP. The DSP receives each 16-bit word right-justified in 24 bits, with no sign extension.
- **DSPAPWriteByteArray()** and **DSPAPReadByteArray()** transfer arrays of 8-bit data packed in type **unsigned char**. These are also used for processing sound data; each 32-bit word in the source array provides four successive 8-bit samples in the DSP, right-justified in 24 bits and with no sign extension.
- **DSPAPWriteFloatArray()** and **DSPAPReadFloatArray()** transfer arrays of **float**s. The data is converted to type DSPFix24 before being sent to the DSP, and converted back to **float** when read. Otherwise, these functions are the same as those that read and write **int** data. Host floating-point data must lie between −1.0 and 1.0 in order to be accurately represented in DSP fixed point.
- **DSPAPWriteDoubleArray()** and **DSPAPReadDoubleArray()** transfer arrays of **double**s. The data is converted to and from type DSPFix24. Note that double-precision offers no advantage relative to single-precision **float** data; the larger double-precision mantissa isn't representable in DSP fixed point.

Functions Returning Address Limits

The DSP memory map addresses that pertain to array processing are defined as constants in the header file **dsp/dsp_memory_map_ap.h**. The following functions return these values and are provided so you can avoid compiling in

constants that may change in the future. None of these functions take arguments.

- **DSPAPGetLowestAddress()** returns the lowest address available for the user in external memory, using image 1 of the memory map. This value is represented by the constant DSPAP_XLE_USR and is always the start of external memory (0x2000 in the present hardware). Note that x, y, and p memory spaces are overlaid in image 1; in other words, DSP memory locations x:*n*, y:*n*, and p:*n* refer to the same physical memory cell for each *n* of the DSP external RAM.
- **DSPAPGetHighestAddress()** returns the highest address available for the user in image 1 of external memory (DSPAP_XHE_USR).
- **DSPAPGetLowestAddressXY()** returns the lowest address available for the user in external memory using image 2 (DSPAP_XLE_USG). This value points to the same physical location as **DSPAPGetLowestAddress()**, but in the address partition where x and y memory spaces are physically separated.
- **DSPAPGetHighestXAddressXY()** returns the highest address available for the user in the x partition of external memory, image 2 (DSPAP_XHE_USG).
- **DSPAPGetHighestYAddressXY()** returns the highest address available for the user in the y partition of external memory, image 2 (DSPAP_YHE_USG).
- **DSPAPGetHighestAddressXY()** returns the minimum of DSPAP_XHE_USG and DSPAP_YHE_USG.
- **DSPAPLoadAddress()** returns the start address of the array processing program in DSP on-chip program memory. The start address is equal to DSPAP_PLI_USR (the start of internal program memory for the user) plus the size of the preamble program that appears before the array processing program.

Array Processing Functions

The array processing functions perform manipulations on data residing in the DSP. They're generated through **dspwrap** from DSP56001 assembly language macros: **dspwrap** assembles a binary image from the macro source and wraps a C function around it. When the function is called, the binary image at its heart is downloaded to the DSP and executed there.

The source code files for the macros from which the array processing functions were generated are provided in the directory **/usr/lib/dsp/apsrc**. These files are made available as programming examples and to allow you to wrap combinations of existing macros into new functions.

Naming Conventions

The following abbreviation conventions apply to the DSP macro and array processing function names.

Abbreviation	Meaning
c	Complex
v	Vector
m	Matrix if operand, minus if operator
s	Scalar
t	Times
p	Plus
i	Immediate
b	Backwards
br	Bit-reversed
mag	Magnitude
max	Maximum
min	Minimum
lim	Limit
rand	Random
real	Real part
imag	Imaginary part

For example, the macro name **vtspv** means "vector times scalar plus vector," and **sumvmag** means "sum vector magnitudes." In addition to these abbreviations, there are two other sources of mnemonics: the DSP56001 instruction set itself (for example, **veor** means "vector exclusive or") and the DSP software published by Motorola. Any term not

covered by these conventions is spelled out in full; for example, **vsquare** stands for “vector square,” which squares each element of a vector.

The following sections provide one-line summaries of the array processing functions. For brevity, each function is given by the name of its underlying DSP macro; in other words, without the “DSPAP” prefix. More detailed descriptions of the functions are provided in Volume 2, Chapter 3. A list of the calling sequences (only) for the functions appears in Volume 2, Appendix B.

Real Vector and Matrix Operations

mtm	matrix times matrix
vreal	vector real part extraction
vimag	vector imaginary part extraction
vclear	vector clear
vfill	vector fill with a constant value
vfilli	vector fill immediate (fill from argument)
vmove	vector move
vmoveb	vector move backwards
vmovebr	vector move bit-reversed
vabs	vector absolute value
vnegate	vector negate
vpv	vector plus vector
vpvnolim	vector plus vector, no limiting
vmv	vector minus vector
vtv	vector times vector (pointwise multiply)
vtvpv	vector multiply plus vector
vtvpvtv	vector multiply plus vector multiply
vtvmvtv	vector multiply minus vector multiply
vps	vector plus scalar
vpsi	vector plus scalar immediate
vts	vector times scalar
vtsi	vector times scalar immediate (scalar in argument)
vtsmv	vector times scalar minus vector
vtspv	vector times scalar plus vector
vtvms	vector times vector minus scalar
vtvps	vector times vector plus scalar
vramp	vector ramp
vrampi	vector ramp immediate (slope in argument)
vrand	vector random numbers
vreverse	vector reverse elements
vsquare	vector square
vswap	vector swap

Complex Vector Operations

The complex vector operations perform elementary operations on arrays of complex data. Complex numbers, each consisting of a real part and an imaginary part, arise naturally in the application of spectrum analysis.

cvcombine	complex vector combine (two real to complex)
cvconjugate	complex vector conjugate
cvfill	complex vector fill with a constant value
cvfilli	complex vector fill immediate (from argument)
cvmandelbrot	complex vector Mandelbrot set generator
cvmcv	complex vector minus complex vector
cvmove	complex vector move

cvnegate	complex vector negate
cvpcv	complex vector plus complex vector
cvreal	complex vector from real (zero imaginary part)
cvtcv	complex vector times complex vector
fftr2a	radix 2 FFT (requires xy memory partition)

Maximum and Minimum Operations

maxmagv	scalar maximum magnitude of vector elements
minmagv	scalar minimum magnitude of vector elements
maxv	scalar maximum of vector elements
minv	scalar minimum of vector elements
vmax	vector maximum of two vectors
vmin	vector minimum of two vectors

Vector Sum Operations

sumv	vector element sum
sumvmag	vector magnitude sum
sumvnolim	vector element sum, no limiting

Vectorized DSP Instruction Operations

This family of array processing functions lets you apply individual DSP instructions to each element (or pair of elements) in an entire vector (or pair of vectors). For example, **DSPAPvand()**, applies the DSP's **and** instruction to each successive pair of elements from the two input vectors. Note that the logical operations **and**, **or**, and **eor**, are bitwise operations on successive vector elements—as opposed to word-oriented operations—and provide a full word of output information.

vand	vector and
vor	vector or
veor	vector exclusive or
vlsl	vector logical shift left
vlsr	vector logical shift right
vasl	vector arithmetic shift left
vasr	vector arithmetic shift right

Creating New Array Processing Functions

You can create your own array processing function by writing DSP56001 assembly language and processing it with **dspwrap**. Programming examples that illustrate this process are provided in the directory **/NextDeveloper/Examples/DSP/ArrayProcessing**. The relevant examples are:

libap	Example of creating a custom library of array processing routines
fuse	Example of fusing supplied array processing macros into a single C function

The basic procedure is to copy an existing array processing macro that's closest to your needs (from the directory **/usr/lib/dsp/apsrc**), modify it to further suit your purposes, run **dspwrap** to generate the C function interface, and link the function into your own array processing library. Note that existing array processing macros can be invoked inside your new array processing macro.

The following demonstrates a typical invocation of **dspwrap**:

```
dspwrap -ap -nodoc mymacro.asm
```

The **-ap** command-line argument indicates that the program is being used to generate an array processing function. **-nodoc** suppresses automatic documentation generation. The file **mymacro.asm** contains a DSP macro for which the program creates a C function named **DSPAPmymacro()**. The C function is written to a file named **DSPAPmymacro.c**.

Further details about **dspwrap** are provided in a UNIX manual page.

Real-Time Digital Signal Processing

A *digital signal* is a sequence of numerical measurements of a physical variable, such as temperature or air pressure, over time. For example, the DSP is used as a signal processor when it synthesizes music or processes recorded sounds. Normally, a signal processing system is thought of as operating on continuous streams of signal data in real time. However, a signal can be broken into a succession of arrays, allowing it to be processed in terms of array processing operations. When implemented using array processing, the operations are performed on a succession of these arrays (or vectors), typically out of real time. It's possible to perform array-oriented operations on signals and keep up with real time, although the output is always delayed by an amount no less than the time it takes to process a single array.

The UnitGenerator class, part of the Music Kit, is designed to support the needs of real-time digital signal processing. In contrast to array processing functions, which cause DSP macros to execute inside the DSP without host interaction, the UnitGenerator class supports real-time communication between the host and the DSP. Parameters of a UnitGenerator can be updated after every "tick" of samples of the output signal have been computed. (The Music Kit presently uses 16 samples per tick.) Thus, at the expense of more DSP resources devoted to real-time communication and buffering, it's possible to configure a network of signal processing modules for real-time processing of signals coming in and going out of a DSP serial port or through the host interface.

Similar to array processing functions, UnitGenerator subclasses are built around DSP assembly language macros. By combining UnitGenerators into an Orchestra object, you can assemble a large amount of DSP code that can be sent to the DSP with a single instruction. One Orchestra can perform a series of computations that would otherwise require several array processing function calls.

For more information on the UnitGenerator class, see Chapter 3, "Music." The DSP source code for the Music Kit monitor is in **/usr/lib/dsp/smsrc/mkmon8k.asm**. It can be adapted to accommodate different tick sizes and buffer lengths.