How to Write DSP Modules for the NeXT Computer™

by J.O. Smith
(Last updated Aug. 4, 1994)
(Last updated by D. Jaffe, Feb. 9, 1993)

Introduction

This document describes how to write signal processing modules to run on the Motorola DSP56001 digital signal processing chip in the NeXT Computer. Such modules may be designed to aug-ment the array processing module library, the Music Kit "unit generator" library, or both.

Needed Materials

To get started, you need at least:

A NeXT Computer
The DSP56000/1 User's Manual (obtainable from Motorola)
The CCRMA Music Kit and DSP Tools Distribution

You might also want to get a copy of the DSP56001 Data Sheet to obtain a succinct overview of the DSP. (The manual pretty much jumps right into an exhaustive coverage.) Documents from Motorola can be obtained by calling (512)891-2030.

Needed Documents

After installation of the CCRMA distribution, look in
/LocalLibrary/Documentation/DSP
/LocalLibrary/Documentation/MusicKit/DSP
for various information pertaining to DSP programming.
The DSP56000/1 macro assembler and linker manuals can be found online in
/LocalLibrary/Documentation/Motorola/dsp_assembler_manual.txt
The Music Kit programming examples, including an example unit generator, are
/LocalDeveloper/Examples/MusicKit/*

Development Outline

The basic steps in the development of an array processing module or Music Kit unit generator are as follows:

1. Copy the directory /usr/local/lib/dsp/test to a local directory of your own.

Example:
```
cp -r -p /usr/local/lib/dsp/test ~/dsptest
cd ~/dsptest
chmod +w *
```

2. Copy the closest pre-existing example from the on-line DSP sources.

The sources are located in the following directories:

/usr/local/lib/dsp/apsrc/*.asm = CCRMA-supplied array processing macros.
/usr/local/lib/dsp/ugsrc/*.asm = CCRMA-supplied unit generator macros.

Examples:

```
cd ~/dsptest                               [created in step 1 above]
cp /usr/local/lib/dsp/apsrc/vpv.asm .  ["vector plus vector" macro]
cp /usr/local/lib/dsp/ugsrc/ramp.asm .
["ramp" unit generator]
```

3. Adapt one of the test programs from /usr/local/lib/dsp/test to your new macro.
The test program ap_ex1.asm tests vpv.asm (copied in the above example).
The test program mk_ex1.asm tests ramp.asm (also copied above).

4. Assemble the test program to produce a DSP-loadable ".lod" file.

Examples:

```
cd ~/dsptest           [created in step 1 above]
dspasm ap_ex1          [the shell script dspasm is in this directory]
dspasm mk_ex1          [unit generator example]
```

5. Load and run the test .lod file into DSP debugger Bug56 to see if it works

Example:

```
open /LocalLibrary/Documentation/Ariel/Bug56Reference.wn      [doc]
open /LocalDeveloper/Apps/Bug56.app                           [app]
[Menu select: File, Load and Erase Symbols, ap_ex1.lod]
[Menu select: Run]    [or select SSTEP in the control panel]
```

6. Run dspwrap to produce a C function for the array processing module, or
Objective C classes for the unit generator.

Examples:

```
cd ~/dsptest                          [our examples directory]

dspwrap -ap -nodoc vpv.asm            [man dspwrap will describe the options
        See the new files DSPAPvpv.h and DSPAPvpv.c
        See /LocalDeveloper/Examples/DSP/ArrayProcessing/libap
                for what to do with the new C function.
        See /LocalDeveloper/Examples/DSP/ArrayProcessing/myAP
                for examples of how to do dynamic loading of the DSP.]

dspwrap -ug -nodoc ramp.asm           [man dspwrap will describe the options]
```

Important Warning

If the last instruction of a unit generator is restrictive in any way (such as restoring an index register), you should add a final "nop". Since the Music Kit supports arbitrary dynamic loading, there is no telling what the next instruction will be.

Array Processing Modules Versus Unit Generator Modules

Array Processing

- o Non-Real Time
- o Arbitrary Vector (Array) Size
- o Real or Complex Vectors or Matrices
- o Matrix Access = Vector Access using ‘‘Skip Factor’’
- o Complex Access = Real Access using ‘‘Skip Factor’’
- o Simple Load and Go Protocol

Unit Generators

- o Real Time Signal Processing
- o Short Fixed Vector Size
- o Real Sampled Data Streams (needing DMA buffers)
- o Contiguous Signal Vectors
- o Timed Message Support (needing message buffer)
- o Untimed and ‘‘Timed 0’’ Messages

Structure of an Array Processing Program on the DSP

```
start:   Initialize  R_X          ; R_X points to arguments in DSP memory
         AP Module 1              ; Each module leaves R_X pointing one
          ...                     ; past the module's memory arguments.
         AP Module N              ; All memory arguments are in space x.
         jmp end_main             ; Tell host the AP program finished.
```

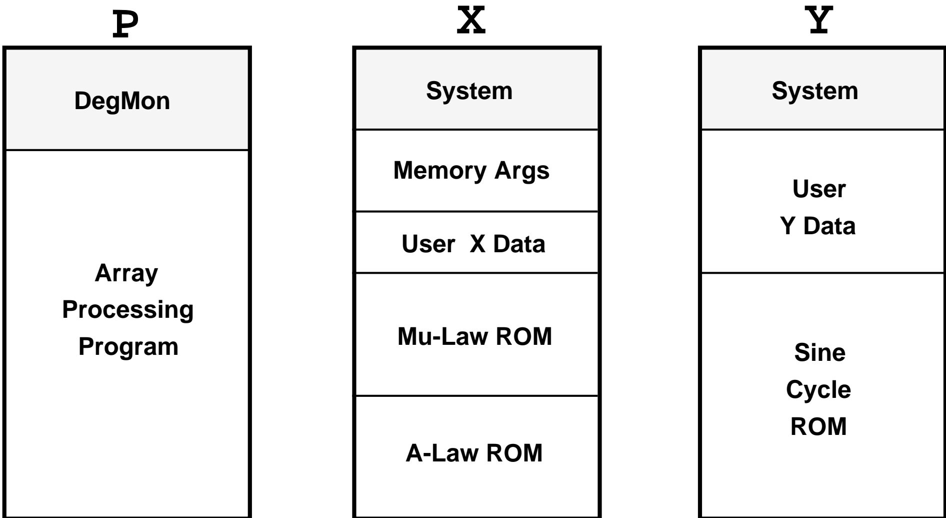
The DSP software is constructed not on a model of subroutines written in DSP56001 assembler, compiled, and linked. Rather, each module is a macro, compiled in-line. Of course, your macro can simply call a subroutine if that’s what you prefer.

Structure of an Array Processing Module Macro

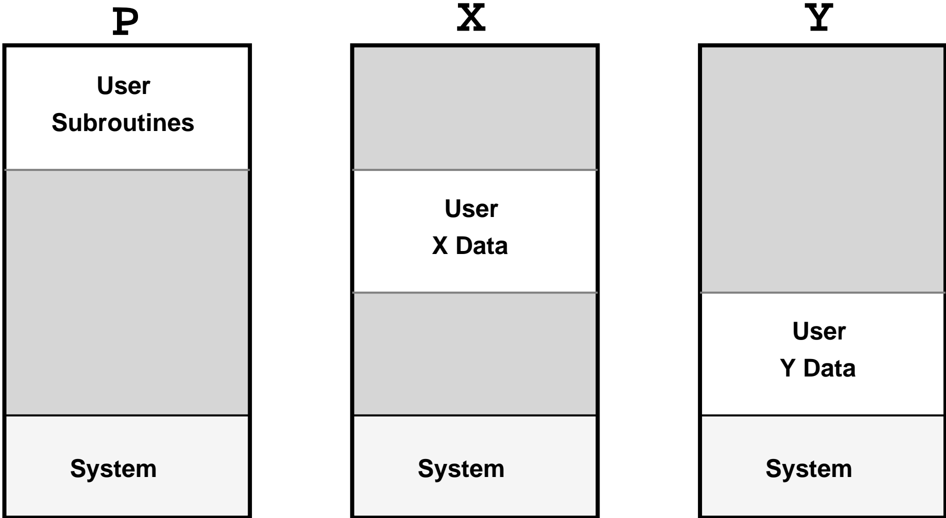
Each AP module is a section of DSP code. Module macros are expanded in line (no subroutines unless you make them yourself). Vectors are passed as pointers to arbitrary data vectors in x memory. Each AP module operates as follows:

- o Load ALU and address registers from memory arguments (“move x:(R_X)+,<reg>”).
- o Execute requested vector operation in a hardware DO loop.
- o Leave memory-argument pointer R_X pointing to args for next AP module.

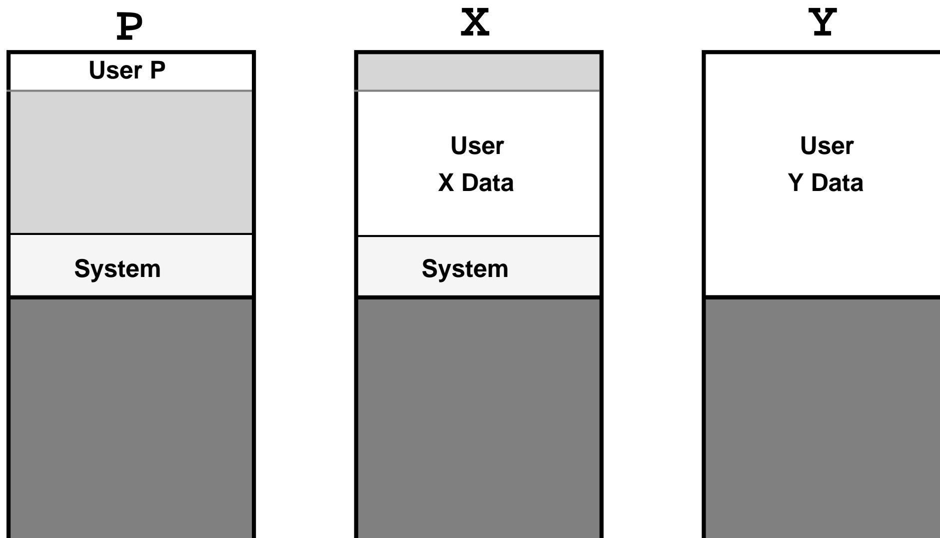
Array Processing Internal DSP Memory Map



Array Processing External DSP Memory Map 1



Array Processing External Memory Map 2



- X and Y physically separate
- P overlays X
- Memory halved in size

The Orchestra Loop is a DSP sound synthesis program. The loop executes once for each "tick" of output sound. A "tick" is currently 16 samples. All timed messages occur on a tick boundary. The Orchestra Loop has the following structure:

System Updates

Loop: Switch DMA Buffers and issue DMA requests, if necessary
Execute all timed messages having time-stamp greater than or equal to "now"
Add 16 to 48-bit sample counter
Reset the 3 memory-argument pointers (x, y, and l banks)

Sound Synthesis

Unit Generator 1
• • •
Unit Generator *n*

Possibly Leap to Off-Chip Memory
Unit Generator *n*+1
• • •
Unit Generator *m*
Go to Loop

An Example Stand-Alone Orchestra Program

The following DSP source code will actually assemble to produce a stand-alone DSP program which can be loaded and executed in the DSP. Such a program would normally be used only for debugging a unit generator. In real usage, the Music Kit constructs the orchestra program dynamically, loading unit generators as needed to build the synthpatches used for synthesizing sound in real time. The parameters of the unit generators, set by macro arguments in the example below, are set in real usage by "timed messages" which were triggered by user events or scorefile data via the Music Kit.

```
----------------------------------------------------------------------------------------------------------
;; DSP56001 Assembly Language for DSP Orchestra Loop
;; To assemble: asm56000 -A -B -L -OS,SO -I/usr/local/lib/dsp/smsrc/ filename

ASM_SYS set 1                       ; Request Music Kit monitor be assembled too
    include 'music_macros'
    beg_orch 'test'                 ; Orchestra program
        new_yib yvec,16,0           ; Allocate y patchpoint
        beg_orcl                    ; Begin orchestra loop
            unoise p,1,y,yvec,0     ; Noise unit generator
            out2sum p,1,y,yvec,0.5,0.5 ; Panning stereo output
        end_orcl                    ; End orchestra loop
    end_orch 'test'                 ; End test program
----------------------------------------------------------------------------------------------------------
```

Important point: Note that the -I option to asm56000 tells you that ALL macros invoked in this simple orchestra program example are to be found in the directory **/usr/local/lib/dsp/smsrc/**. Whenever you see a macro invocation and you would like to know what it does, just go look it up in the **smsrc** ("system macro source") directory. Macros of the form "beg_*" and "end_*" are defined in **beginend.asm**, and macros of the form "new_*" are in **allocusr.asm**. The unit generator macros themselves are in **/usr/local/lib/dsp/ugsrc/** ("unit generator source") under their own names (e.g. **unoise.asm**). See **music_macros.asm** itself to find out about the entire Music Kit DSP monitor. If you do not know how to use **grep**, please first learn about it from its UNIX "man page" (i.e., type "*man grep*" in a Terminal window, or ask DigitalLibrarian to search for it among the UNIX manual pages).

The above example allocates a length 16 signal vector called yvec in internal y DSP memory with the invocation of the **new_yib** macro ("new y internal block of memory"). Inside the orchestra real-time loop, delimited by the **beg_orcl** and **end_orcl** macros, there is an instance of the **unoise** unit generator which simulates white noise, and an instance of the **out2sum** unit generator which sums its argument into the sound-out buffer. (This buffer is precleared by code emitted in the **beg_orcl** macro.) The **end_orcl** macro simply jumps back to a label at the beginning of the orchestra loop defined in **beg_orcl**. This loop executes until the chip is reset or Host Flag 0 is set in the DSP host interface. This is all there is to an orchestra program running in the DSP.

More elaborate examples are built up by inserting more unit generators like **unoise** into the orchestra loop. To do this while the loop is running requires precisely timed software downloads. To be able to drastically rebuild the orchestra loop between tick computations, there must exist an adequate supply of buffering of the output sound data in the DSP. We currently use 1024 words total for the dual stereo sound-out DMA buffers.

The macro **beg_orcl** emits code to perform the following once-per-tick services in the orchestra loop:

• Request DMA transfer of the completed sound-out buffer if necessary.

• Compare the time stamp of the next timed message (queued in DSP memory) to the current DSP time (in samples), and execute all messages timed for "now" or earlier.

• Add 16 to the 48-bit current time variable to update it for one iteration of the orchestra loop.

• Reset the three unit-generator "memory argument pointers" R_X, R_Y, and LARGS to their beginning values for the orchestra loop.

The Importance of Vectorized Computations

In the current implementation, the tick size is 16 samples. Since the overhead of setting up DSP index and ALU registers is often comparable to the amount of actual work done in the inner loop of a tick computation, a tick size of 16 brings the set-up overhead for each unit generator down under ten percent in most cases.

On the other hand, it's also desirable not to make the tick size any larger than necessary because it also determines the size of a patchpoint, memory that's used for communication between unit generators. It's highly desirable that all patchpoints fit on-chip in the DSP. This is because the three-way parallel data move capability of the DSP requires that at least two of the data moves be to or from on-chip memory. Only one parallel external memory read or write is possible because only one set of data and address pins is brought out of the chip. Note that patchpoints can generally be reused; for example, most unit generators can overwrite (one of) their input patchpoints for output. An efficient synthpatch should try to get by using only one *x* and one *y* patchpoint. This point is discussed further below.

The hardware DO instruction in the Motorola DSP56001 further enhances the benefits of vectorized computations by performing the loop test and branch in parallel with the block iteration; while there is a three instruction-cyle (six clock-cycle) overhead incurred to set up the loop, the individual loop iterations suffer no test and branch overhead.

Finally, the dual, parallel indexing ALUs provide zero-overhead memory address updates for two parallel data transfers, with skip factors, modulo (wrap-around) addressing, and even bit-reverse indexing for FFT data shuffling provided as indexing modes.

Thus, vectorized computations are far more efficient than computing a single audio sample per iteration of the orchestra loop. The price for this efficiency is a loss of control bandwidth since parameter updates (envelope break-points, filter coefficients, etc.) are only installed once at the beginning of each tick.

Unit Generators

The *unit generator* is a fundamental building block of sound synthesis. It can also be regarded as one computational "black box" in a real-time signal processing diagram. Almost every unit generator has an output signal that it writes into a patchpoint every time it runs (once per tick). At the heart of every unit generator is a DO loop that executes 16 times (once per sample) to produce an output tick. Most unit generators also have one or more signal inputs that are also 16-sample long vectors.

By carefully arranging the order of execution of unit generators within the orchestra loop, it's often possible to significantly reduce the number of patchpoints required. For example, if unit generators A, B, and C are arranged in a linear chain, i.e., A—> B—> C, and if no other unit generators depend on their outputs, then only one patch-point is needed if A runs before B and B runs before C. The patchpoint between A and B is simply reused as the patchpoint between B and C.

Note also that the order in which unit generators are executed in the orchestra loop determines whether or not there is an 16-sample delay in the connection between them. For example, if in the above example, the order of execution is C,B,A, then then C's output will be delayed two ticks relative to A's output plus whatever delay is built into B and C. For this reason, the Music Kit provides a way to control the order of execution of unit generators.

Examples of Existing Unit Generator Macros

Asymp
One segment of a piecewise exponential envelope.

SlpDur
Piecewise linear envelope from specification of the slope
and duration of each segment.

Biquad
Two-pole, two-zero filter section.

Delay
Digital delay line.

Dswitch
Delayed switch (for switching a connection at a particular sample time).

Oscgafi
Oscillator with **g**eneral address mask, and envelopes on
amplitude and **f**requency. The wave table is **i**nterpolated.

Scl1Add2
Add scaler times first input signal to the second.

There are many additional oscillator, mixer, and filter variations, and various signal generators (noise, im-pulses, etc.). Some have tick-based versions computing 1 rather than 16 values. A more complete list of unit generators is given in an appendix.

Structure of a Unit Generator Macro

o Each Unit Generator is a section of DSP synthesis code.
o Unit-Generator macros are expanded in line (no subroutines in typical cases).
o Signals are passed as pointers to 16-sample “patchpoints.”
o Tables and delay-lines used are typically a multiple of 16 long.
o Parameter update rate therefore equals the audio sampling-rate divided by 16.

Each Unit Generator operates as follows:

Load ALU Registers from Memory Arguments
    Move input/output signal pointers to Rn in Address ALU.
    Move coefficients and state (e.g. filter state) to Data ALU.
    Access arguments using auto-increment addressing.

Execute Tick Loop
    Perform desired function in a length 16 hardware DO loop.
    Access signals with auto-increment addressing.

Save State to Memory Arguments
    Save run-time state (e.g. oscillator phase, filter delays).
    Leave memory arg pointers set for next unit generator.

An Example Unit Generator Macro

Below is an example unit-generator source file. It contains a single unit generator macro, **add2** which is found by the Motorola DSP assembler when assembling a stand-alone orchestra program. For the Music Kit, the macro is automatically wrapped (by **dspwrap**) in a small main DSP program, assembled in “relative mode” to make it relocatable, and packaged into an Objective C object class which is called upon by the Music Kit to dynamically load an instance of the unit generator into the DSP during a musical performance. Below is the unit generator source code:

```
______________________________________________________________
;;  Modification history
;; ____________________
;;  10/19/87/jos - initial file created from scale.asm
;;
;;———————————————————DOCUMENTATION——————————————————--
;;  NAME
;;          add2 (UG macro) - add two signals to produce a third
;;
;;  SYNOPSIS
;;          add2 pf,ic,sout,aout0,i1spc,i1adr0,i2spc,i2adr0
;;
;;  MACRO ARGUMENTS
;;          pf     = global label prefix (any text unique to invoking macro)
;;          ic     = instance count (such that pf\ _add2_\ ic\ _ is globally unique)
;;          sout      = output memory space ('x' or 'y')
;;          aout0     = initial output address in memory sout
;;          i1spc     = input 1 memory space ('x' or 'y')
;;          i1adr0    = initial input address in memory i1spc
;;          i2spc     = input 2 memory space ('x' or 'y')
;;          i2adr0    = initial input address in memory i2spc
;;
;;  DSP MEMORY ARGUMENTS
;;          Arg access      Argument use              Initialization
;;          __________      ____________              __________
```

```
;;          x:(R_X)+          address of input 1 signal        i1adr0
;;          y:(R_Y)+          address of input 2 signal        i2adr0
;;          y:(R_Y)+          address of output signal         aout0
;;
;;  DESCRIPTION
;;          The add2 unit-generator sums two patch-points, forming a
;;          third. The output can be the same patch-point as either input.
;;          The inner loop is two instructions if the memory spaces
;;          for in1, in2, and out are x,y,x, respectively. In all other cases the
;;          inner loop is three instructions.
;;
;;  DSPWRAP ARGUMENT INFO
;;          add2 (prefix)pf,(instance)ic,
;;             (dspace)sout,(output)aout0,(dspace)i1spc,(input)i1adr0,(dspace)i2spc,(input)i2adr0
;;
;;  MAXIMUM EXECUTION TIME
;;     112 DSP clock cycles for one "tick" which equals 16 audio samples.
;;
;;  MINIMUM EXECUTION TIME
;;     78 DSP clock cycles for one "tick".
;;
          beg_def 'add2'                              ; begin macro definition
          define add2_pfx "pf\ _add2_\ ic\ _"         ; pf = <name>_pfx of invoker
          define add2_pfxm """add2_pfx"""             ; this form need in macro args
add2      macro pf,ic,sout,aout0,i1spc,i1adr0,i2spc,i2adr0
          beg_mac 'add2'                              ; begin macro body
          new_xarg add2_pfxm,i1adr,i1adr0             ; allocate x memory argument
          new_yarg add2_pfxm,i2adr,i2adr0             ; allocate y memory argument
          new_yarg add2_pfxm,aout,aout0               ; allocate y memory argument
          move y:(R_Y)+,R_I2                          ; input 2 address to R_I2
          move y:(R_Y)+,R_O                           ; output address to R_O
          move x:(R_X)+,R_I1                          ; input 1 address to R_I1

          move i2spc:(R_I2)+,A                        ; load input 2 to A
          move i1spc:(R_I1)+,X0                       ; load input 1 to X0
          do #I_NTICK,add2_pfx\ tickloop              ; enter do loop
             add X0,A  i1spc:(R_I1)+,X0               ; do add and fetch input 1
             if "i1spc"=='x'&&"i2spc"=='y'
                if "ospc"=='x' ; xyx
                      move A,sout:(R_O)+ i2spc:(R_I2)+,A   ; optimal
                else ; xyy
                      move A,sout:(R_O)+
                      move i2spc:(R_I2)+,A
                endif
             else
                move A,sout:(R_O)+
                move i2spc:(R_I2)+,A
             endif
add2_pfx\ tickloop
          end_mac 'add2'
          endm
          end_def 'add2'
```

Discussion

There are several points to note: The documentation at the top of the file is turned into a UNIX-style man page by the **dspwrap** program. The **DSPWRAP ARGUMENT INFO** field is used by **dspwrap** to generate Objective C code, and it does not appear in the man page. The first two macro arguments, prefix (pf) and instance count (ic), are used to generate unique global symbols within the macro. They can be arbitrary text, although, by convention, the instance count increments through the integers from 1, and the prefix is a unique label prefix passed down from above. The purpose of having both a prefix and an instance count is to support unique label generation at all levels in a nested macro expansion.

The benefit of this use of prefix and instance count is the availability of global handles on all interesting quantities at all levels of macro expansion.

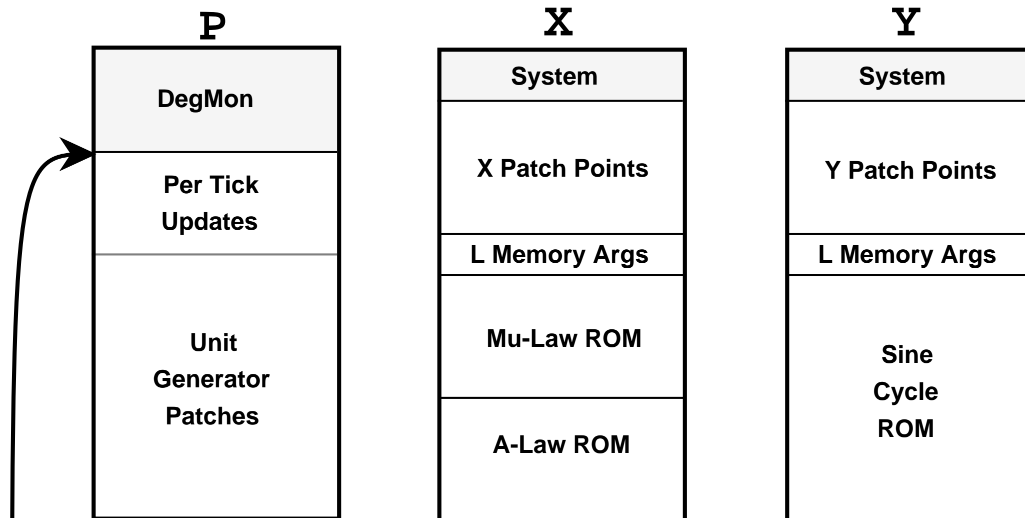
After the prefix and instance count arguments of the add2 unit generator, there is the macro argument sout which can be either *y* or *x*. It specifies the memory space of the output signal vector. The next macro argument, aout0, specifies the address of the output signal. Similarly, i1spc, i1adr0, i2spc, and i2adr0 specify the memory space and memory address of the two length 16 input patchpoints.

An attempt is made to optimize each memory space combination. The **dspwrap** program emits an Objective C subclass of **UnitGenerator.m** for each combination of memory spaces. For example, the **add2** example above would generate *eight* subclasses, corresponding to each combination of memory space for the two inputs and single output.

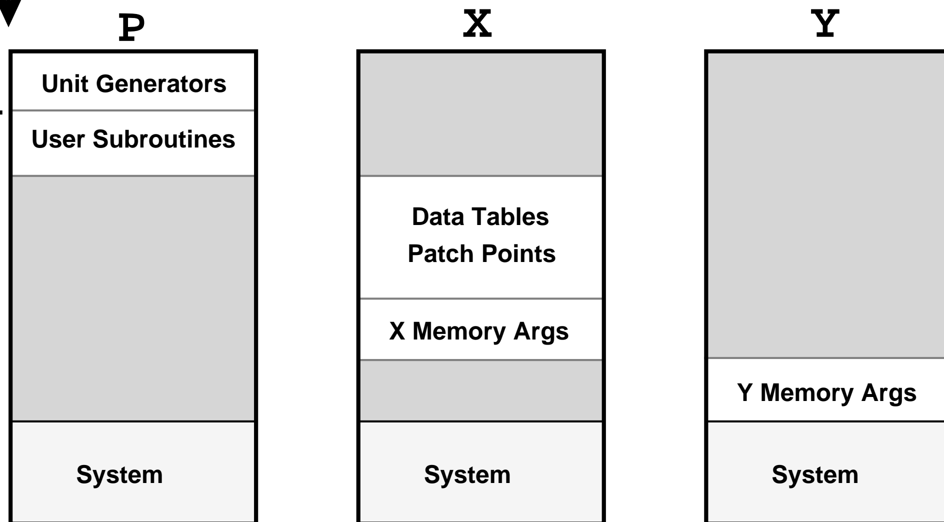
Unit Generator Memory Arguments

The parameters and run-time state of the unit generators are held in so-called *memory arguments*. There are three memory-argument blocks, corresponding to *x*, *y*, and *l* DSP memory. The *l* memory arguments must be on chip since we use the overlaid external memory partition (see DSPMemoryMap.rtf - Fig1). The *x* and *y* memory arguments, however, are off chip, because on-chip memory is more valuable for patch-points (accessed in the inner loop) than for memory arguments (which are accessed outside the inner loop). Long memory arguments are used for oscillator phase and table increment (which determines frequency of oscillation), exponential envelope state, and the slope and current value of the linear amplitude ramp. All other unit generator parameters in the CCRMA set are “single precision,” i.e., 24-bit values.

Music Kit Internal DSP Memory Map



Music Kit External DSP Memory Map



In the execution of an orchestra loop, each unit generator loads its ALU and index registers from its memory arguments. Any state needed for the next tick's computation (such as delayed signals in a digital filter) is written by the unit generator into the memory arguments on exit. To minimize pointer initialization overhead, the memory arguments are accessed sequentially in internal *x* and *y* RAM in the order needed by the unit generator. This eliminates initialization of the three memory argument pointers, except once at the beginning of the orchestra loop. Loading of addresses from memory arguments typically looks as follows:

```
move y:(R4)+,R6          ; output address to output pointer R6
move x:(R0)+,R1          ; input 1 address to input pointer R1
move y:(R4)+,R5          ; input 2 address to input pointer R5
```

Note that addresses cannot be loaded in parallel. Loading of data memory arguments (such as oscillator amplitude or digital filter coefficients), is handled the same way, except that sometimes two data arguments can be load using parallel move syntax, e.g.,

```
move x:(R0)+,X0     y:(R4)+,Y0     ; load gain1 to X0 and gain2 to Y0
```

Each unit generator is responsible for leaving R0, R4, and R2 (*l* arguments) pointing one past its argument block on exit. If a unit generator needs to save run-time state (such as current envelope amplitude), it's best to place this state last in the memory arguments. Having all running state in the memory arguments facilitates memory compaction needed with dynamic loading.

All memory arguments are given symbolic names. These symbols are collected into the assembly output file and are needed to write (or read) the arguments from a host task. The Music Kit™ takes care of this linkage for most users, but these hooks can support more general real-time signal processing applications which need not involve the Music Kit.

Below are the various ways memory-argument accesses appear in unit generator code:

```
move x:(R_X)+,R_I1               ; Input 1 address to R_I1
move y:(R_Y)+,R_I2               ; Input 2 address to R_I2
move y:(R_Y)+,R_O                ; Output address to R_O

move x:(R_X)+,X0 y:(R_Y)+,Y0     ; Load data to X0 and Y0

move x:LARGS,R_I1
move x:(R_I1)+,A                 ; Long datum to A (A1 and A0 written)
move R_I1,x:LARGS
```

Notes:
o Two data loads may be possible in parallel, e.g., the last example above.
o Long loads are always done using two parallel data moves in the DSP.
o Since there is no register assigned to point to L arguments, two extra instructions, one to retrieve the pointer and one to update it.
o There is no possibility of a parallel move when loading an address register.

Use of DSP Address Registers in Unit Generators

The DSP address registers are given mnemonic names by the "define" statements in **/usr/local/lib/dsp/smsrc/defines.asm**. you don't have to use these names, but you do need to know that R_X, R_Y, R_IO, R_IO2, and R_HMS are used by the DSP Music Kit Monitor (the "DSP system"). You are free to save and restore R_X and R_Y in your unit generator code, but you cannot use R_IO, R_IO2 and R_HMS unless you first turn off interrupts. R_IO is used for sound output transfers, R_IO2 is used for sound input transfers, and R_HMS is used for efficient remote procedure calls by the host ('040).

Bank 1	Bank 2
R0 = R_X	R4 = R_Y
R1 = R_I1	R5 = R_I2
R2 = R_IO2	R6 = R_O
R3 = R_IO	R7 = R_HMS

Analogous names are used for the "N" and "M" registers, e.g.,

N0 = N_X
M0 = M_X

etc.

Mnemonic register names expanded

R_X = x memory argument pointer
R_Y = y memory argument pointer

R_I1 = general-purpose register 1 (arbitrarily named "input 1")
R_I2 = general-purpose register 2 (arbitrarily named "input 2")
R_O = general-purpose register 3 (arbitrarily named "output")

R_IO2 = index register used for sound transfers into dsp
R_HMS = pointer to top of host message stack
R_IO = index register used for sound transfers out of dsp

Memory Argument Pointers

The memory-argument pointers, R_X, R_Y, and LARGS, are set to their starting values at the top of the orchestra loop. Each unit generator assumes these registers point to the first x, y, and l memory arguments, respectively, upon entry to that unit generator.

Similarly, in array processing programs, the memory-argument pointer R_X is set to its starting value at the top of the program. Each processing module assumes R_X points to its first memory-argument, and there are no y or l memory arguments for array processing modules.

If you need to use one or more of the memory-argument pointers in your code, just save them, use them, and restore them before exiting the module. They are never used at interrupt level, so you need not turn off interrupts. Remember that you can save R_X, for example, in N_Y, since the offset registers are not used for memory-argument access. In AP modules, R_Y and LARGS are of course already available for general-purpose use.

Note that because of the banks they are in, R_X can be used in parallel with R_Y. For example, if a unit generator had a first X memory argument called "gain1" and a first Y memory argument called "gain2", the two arguments could be loaded simultaneously via

```
move x:(R_X)+,X0    y:(R_Y)+,Y0    ; X0=gain1, Y0=gain2
```

in the first statement of the unit-generator module.

The "offset registers" N_X and N_Y, are scratch registers. You can use them without saving their previous contents and restoring them on exit.

Input/Output Pointers

The "input/output pointers", R_I1, R_I2, and R_O, are really just "scratch" registers which you are free to use any way you want. You never need to save/restore them.

Similarly, N_I1, N_I2, and N_O are fully unrestricted. The names are chosen based on the most common usage they seem to get.

Address Mode Registers

All “M” registers are assumed to be -1 except for M_IO and M_HMS. That is, M_X, M_Y, M_IO2, M_I1, M_I2, and M_O are assumed to be -1 by all unit generator macros and array processing. If you change an M register to something else, remember to change it back before you are done. The fastest way to restore an M register is to move another one into it. For example, if you have helped yourself to M_Y as a 16-bit temporary register in you module code, but you did not touch M_X, you could restore M_Y as follows:

```
move M_X,M_Y            ; restore M_Y from the (left-alone) M_X value
```

DMA Pointer Register R_IO

The DMA pointer, R_IO is used by the fast, two-word interrupt handler for “host receive” or “host transmit” during a DMA transfer. (A DMA transfer is a “Direct Memory Access” in which data is transferred between the DSP and the main system DRAM memory at very high speed, something like two megabytes per second. All sound data is transferred between the DSP and host in this manner.) The whole interrupt handler is only one line of DSP code, either

```
movep s:(R_IO)+N_IO,x:$FFEB ; DMA read from ’s’ memory, s=x,y, or p
```

or

```
movep x:$FFEB,s:(R_IO)+N_IO ; DMA write to x, y, or p memory
```

Thus, R_IO, N_IO, and M_IO are all used at interrupt level, and must be preserved at all times unless interrupts are turned off. M_IO is set to the buffer length minus 1 in order to make the DMA buffers “modulo” as a safety precaution in case the DMA transfer goes too far.

Host Message Stack Pointer R_HMS

Whenever the host writes a word to the DSP, (out of DMA mode), a “host receive” interrupt, occurs in the DSP. (That means that HRIE is set in the DSP host interface to enable an interrupt on HRDF.) The one-line interrupt handler for this single-word-write from the host is just

```
movep x:$FFEB,y:(R_HMS)- ; write (circular) Host Message Stack
```

The HMS buffer is also “modulo”, so M_HMS must be left alone unless interrupts are turned off.

Turning off interrupts in the DSP

To turn off interrupts so that the system IO pointers can be saved, used, and restored in a unit generator module, do the following:

1. Save HF2 and set it.
2. Save HTIE and clear it.
3. Save HRIE and clear it.
4. Invoke the “mask_host” macro (from /usr/local/lib/dsp/smsrc/misc.asm)

Setting HF2 prevents the host from calling a DSP subroutine which could, for example, start up a DMA transfer. It will normally be off unless there is a “timed-zero message” pending in which case it will be set. To use R_HMS only, only HF2 needs to be set.

Clearing HTIE will “freeze” a DMA transfer to host memory, if one is in progress. This works just fine, and nothing bad happens because the DMA suddenly stops. In fact, for efficiency reasons, all sound-out DMAs to the host are started in the frozen state when the buffer to be sent STARTS filling. (This allows each DMA transfer to be set up in the DMA-completion interrupt service routine in the DSP driver, resulting in one DSP interrupt per sound-out buffer on the host.)

Similarly, clearing HRIE will freeze a DMA transfer into the DSP, if one is in progress.

The “mask_host” macro expands to the following DSP code:

```
ori #1,mr       ; raise IPL to lock out host
nop             ; wait for interrupts to really be locked out
nop             ; wait for interrupts to really be locked out
nop             ; wait for interrupts to really be locked out
nop             ; wait for interrupts to really be locked out
```

You should not leave interrupts turned off when you exit the unit generator (thinking to restore them after a future pass) because system services run at the top of the orchestra loop may change the state of the control bits you have saved. For example, HTIE gets set when the currently filling sound-out buffer becomes full.

Restoring DSP interrupts

1. Invoke the “unmask_host” macro (which is just “andi #$FC,mr”)
2. Restore HTIE
3. Restore HRIE
4. Restore HF2

System IO Pointers in Array Processing Programs

While DMA may be used to load and retrieve arrays in array processing, and the HMS is used to set things up with DSP subroutine calls, the current implementation of the array processing system does not do either of these while an AP program is executing. This means you can clobber the DMA and HMS registers in an AP module with no harmful effect. However, a more sophisticated implementation of the AP system would overlap IO with program execution, perhaps using double-buffering of AP programs and data arrays. Therefore, to be compatible with better AP systems in the future, it is best to treat the DMA and HMS pointers as belonging to the system at interrupt level.

Structure of a Unit Generator Stand-Alone Test Program

See

/usr/local/lib/dsp/test/*
/usr/local/lib/dsp/smsrc/allocusr.asm
/usr/local/lib/dsp/smsrc/begin_end.asm

Testing a New Unit Generator in the Music Kit Context

See

/LocalDeveloper/Examples/MusicKit/PlayNote/*
/LocalDeveloper/Examples/MusicKit/exampsynthpatch/*

Automatic C and Documentation Generation

Each array-processing or unit-generator macro source file has leading comments resembling a UNIX “man page.” These comments contain information for automatically generating three types of on-line documentation, and C software which which “wraps” each macro, marrying it to the host software environment.

The comment fields are used by program **dspwrap** to generate:

- A file containing a *one-line summary* for each macro
- A file containing *calling-sequences* for each macro
- A file containing a *"man page"* for each macro
- An invoking *C function* for each array processing macro
- A *prototype master* Objective C class for each unit generator
- *Leaf classes* in Objective C for each unit-generator variation
  (There's a variation for each input/output space combination)

Source-Code Comment Fields

Documentation Generation:

NAME
One-line description of the macro's function.

SYNOPSIS
One-line summary of the macro invocation syntax.

MACRO ARGUMENTS
One-line summary of the nature of each *macro* argument.

DSP MEMORY ARGUMENTS
One-line summary of the nature of each *memory* argument.

DESCRIPTION
A complete description of the macro's purpose and function.

PSEUDO-C NOTATION
Describes the macro's function in a DSP-extended C language.

DSPWRAP C SYNTAX
Calling-sequence summary for array processing C function.

C Code Generation:

DSPWRAP ARGUMENT INFO
Declares the type of each macro argument.

General Information:

MINIMUM EXECUTION TIME
MAXIMUM EXECUTION TIME
Execution time per element (AP) or for one tick (UG).

CALLING DSP PROGRAM TEMPLATE
Test main program in DSP56001 assembly language.

SOURCE
Pointer to on-line macro source file.

MEMORY USE

Words of DSP program memory used by macro expansion.

REGISTER USE
Which ALU registers are used and what they contain.

Appendix — List of Unit Generators in /usr/local/lib/dsp/ugsrc/

add2 - add two signals to produce a third
allpass1 - one-pole digital allpass filter section
asymp - one segment of an exponential (ADSR type) envelope
biquad - direct-form, two-pole, two-zero, filter section
constant - generate a constant signal
delay - sample-based delay line using non-modulo indexing
delayticks - tick-based delay line using non-modulo indexing
dswitch - switch from input 1 to input 2 after delay
impulses - periodic impulse-train generator
mul2 - multiply two signals to produce a third
onepole - one-pole digital filter section
onezero - one-zero digital filter section
orchloopbegin - begin DSP orchestra loop (invokes **beg_orcl** macro only)
orchloopend - end DSP orchestra loop (invokes **end_orcl** macro only)
oscg - simplest oscillator with general address mask
oscgaf - oscillator with amplitude and frequency envelopes
oscgf - oscillator with multiplicative frequency input
osci - interpolating oscillator
oscs - simplest oscillator
oscw - oscillator based on 2D vector rotation
out2sum - sum signal vector into sound output buffer.
patch - patch one signal vector to another
sawtooth - sawtooth oscillator
scale - scale a signal vector by a scalar using mpyr
scl1add2 - add scaler times first signal to the second
slpdur - linear envelope generation using slopes/durations
twopole - two-pole digital filter section
unoise - uniform pseudo-random number generator
unoisehp - highpassed uniform pseudo-random number generator

Appendix — DSP56000/1 Parallel Move Descriptions

Below is a notation describing all possible parallel moves in the DSP56000/1.

The set notation "{a,b,c}" means "pick one of a, b, or c". That is, pick any one element from the group delimited by curly braces.

The notation for "optional" is to enclose the optional quantity in square brackets. For example, "(R0)[+]" stands for either "(R0)" or "(R0)+".

The notation "Rn" means "{R0,R1,...R7}", "An" means "{A0,A1,A2}", and so on.

The notation "Rn~" means "any R register in the opposite bank." For example, "R0~" = {R4,R5,R6,R7} and

“R6~” = {R0,R1,R2,R3}.

The notation “+-” means “{+,-}”.

The notation “A --> B” means “A can move to B”.
The notation “A <-- B” means “B can move to A”.
The notation “A <--> B” means either “B can move to A” or “A can move to B”.

The notation “<desc>” means that “desc” describes the quantity. For example, absolute DSP addresses, which are normally indicated in DSP assembly language by a label name, are denoted below as “<absolute>”. Also, “<op>” denotes a DSP operation (such as “add X0,A”, “mac X1,Y1,B”, etc.).

Let AnyRegD={Rn,Nn,Xn,Yn,An,Bn,A,B} (“any data ALU register”).
Let AnyEA={(Rn)[{+,-}[Nn],(Rn+Nn),-(Rn),<absolute>} (“any effective address”).

Then we have the following possible **SINGLE PARALLEL MOVES**:

```
<op>  #<byte>  --> AnyRegD           ; immediate short data
<op>  AnyRegD <--> AnyRegD           ; register to register
<op>  (Rn)+-[Nn] --> Rn              ; register update
<op>  {x,y}:AnyEA <--> AnyRegD       ; x or y memory data
<op>  #<word> --> AnyRegD            ; 24 bit immediate load
```

and the following possible **DUAL PARALLEL MOVES**:

```
<op>  x:AnyEA <--> {Xn,A,B}    {A,B} --> Yn     ; x mem and reg data
<op>  y:AnyEA <--> {Yn,A,B}    {A,B} --> Xn     ; y mem and reg data
<op>  #<word>  --> {Xn,A,B}    {A,B} --> Yn     ; immediate + reg data
<op>  #<word>  --> {Yn,A,B}    {A,B} --> Xn     ; immediate + reg data

<op>  l:AnyEA <--> {X,Y,A,B,A10,B10,AB,BA}
; long move

<op>  (Rn)[+-[Nn]] <--> {Xn,A,B}       y:(Rn~)[+-[Nn]] <--> {Yn,A,B}
```

In addition, it is good to remember that the TFR instruction allows parallel moves:

```
TFR {A,B,Xn,Yn},{A,B}     <dual or single parallel move>
```

For example, “TFR X0,A A,X0 y:(R4)+N4,Y0” will swap X0 with A1 and load Y0 from data memory.

Appendix — DSP56001 Hardware:

- Motorola DSP56001 clocked at 25MHz
- Memory-mapped Host Interface
- DMA to/from Host Interface (2-5MBytes/sec)
- 8K 24-bit words of zero-wait-state private static RAM
- D-15 connector: SSI and SCI serial ports of the DSP

Appendix — DSP56001 Instruction Set Summary

The following notation is used in the summary:

Notation **Denotes**

‘*’	Instructions that don’t allow parallel data moves
[a,b]	One of a or b
<a,b>	Either a,b or b,a
<n>	A nonnegative integer
#I<n>	n-bit immediate value
A<n>	n-bit absolute address
An	A0, A1, or A2 (similarly for Bn)
Xn	X0 or X1 (similarly for Yn)
Rn	R0, R1, R2, R3, R4, R5, R6, or R7 (similarly for Nn, Mn)
AnyEa	Addressing modes (Rn)[±[Nn]], (Rn+Nn), −(Rn), (similarly for An)
AnyXY	[x,y]:AnyEa
AnyIO	[x,y]:<<pp (x or y peripheral address, 6 bits, 1’s extended)
Creg	Registers Mn, SR,OMR,SP,SSH,SSL,LA,LC
Dreg	Registers Xn,Yn,An,Bn,A,B
Areg	Registers Rn, Nn
AnyReg	Registers Dreg, Areg, Creg
cc	CC(HS) CS(LO) EC EQ ES GE GT LC LE LS LT MI NE NR PL NN

left-justified moves: → [A,B,Xn,Yn]

right-justified moves: → [An,Bn,Rn,Nn]

Arithmetic Instructions

ABS [A,B]	Absolute Value
ADC [X,Y],[A,B]	Add Long with Carry
ADD [X,Xn,Y,Yn,B,A],[A,B]	Add
ADDL [B,A],[A,B]	Shift Left then Add (D=2*D+S)
ADDR [B,A],[A,B]	Shift Right then Add (D=D/2+S)
ASL [A,B]	Arithmetic Shift Left (D1=D1*2)
ASR [A,B]	Arithmetic Shift Right (D1=D1/2)
CLR [A,B]	Clear Accumulator
CMP [Xn,Yn,B,A],[A,B]	Compare (CCR=Sign(D1−S))
CMPM [Xn,Yn,B,A],[A,B]	Compare magnitude (CCR=Sign(D−S))
*DIV [Xn,Yn],[A,B]	Divide Iteration (D/S iteration)
MAC ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply-Add (no X1*X1, Y1*Y1)
MACR ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply, Accumulate, and Round
MPY ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply (no X1*X1, Y1*Y1)
MPYR ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply-Round (no X1*X1, Y1*Y1)
NEG [A,B]	Negate Accumulator
*NORM [A,B]	Normalize Accumulator Iteration
RND [A,B]	Round Accumulator
SBC [X,Y],[A,B]	

Subtract Long with Carry (D = D – S – C)

SUB [X,Xn,Y,Yn,B,A],[A,B]
Subtract (D = D – S)

SUBL [B,A],[A,B]
Shift Left then Subtract (D = 2*D – S)

SUBR [B,A],[A,B]
Shift Right then Subtract (D = D/2 – S)

*Tcc [Xn,Yn,B,A],[A,B]
Transfer Conditionally

TFR [Xn,Yn,B,A],[A,B]
Transfer Data ALU Register

TST [A,B]
Test Accumulator

Logical Instructions

AND [Xn,Yn],[A,B]
Logical AND (D1=D1&S)

*ANDI #I8,[MR,CCR,OMR]
AND Immediate with Control Register

EOR [Xn,Yn],[A,B]
Logical Exclusive OR (D1=D1 XOR S)

LSL [A,B]
Logical Shift Accumulator Left (D1=D1<<1)

LSR [A,B]
Logical Shift Accumulator Right (D1=D1>>1)

NOT [A,B]
Logical Complement on Accumulator (D1=~D1)

OR [Xn,Yn],[A,B]
Logical Inclusive OR (D1=D1S)

*ORI #I8,[MR,CCR,OMR]
OR Immediate with Control Register

ROL [A,B]
Rotate Accumulator Left ([C,D1] ROL)

ROR [A,B]
Rotate Accumulator Right ([D1,C] ROR)

Bit Manipulation Instructions

*BCLR #B5,AnyXY
Bit Test and Clear (C = Selected bit)

*BSET #B5,AnyXY
Bit Test and Set (C = Selected bit)

*BCHG #B5,AnyXY
Bit Test and Change (C = Selected bit)

*BTST #B5,AnyXY
Bit Test on Memory (C = Selected bit)

*JCLR #B5,[AnyXY,AnyIO],xxxx
Jump if Bit Clear

*JSET #B5,[AnyXY,AnyIO],xxxx
Jump if Bit Set

*JSCLR #B5,[AnyXY,AnyIO],xxxx
Jump to Subroutine if Bit Clear

*JSSET #B5,[AnyXY,AnyIO],xxxx
Jump to Subroutine if Bit Set

Loop Instructions

*DO [[x,y]:[AnyEa,A12],AnyReg],L
Start Hardware Loop (L=Label after end)

*ENDDO
Exit from Hardware Loop

Move Instructions

Instruction	Description
*LUA (Rn)[±[Nn]],[Rn,Nn]	Load Updated Register
MOVE (NOP)	Move Data (see parallel move summary)
*MOVEC <AnyXY,Creg>	Move Control Register
*MOVEC [#I16,#I8],Creg	Move Control Register
*MOVEC <Creg,AnyReg>	Move Control Register
*MOVEM <p:AnyEa,AnyReg>	Move Program Memory
*MOVEP <[AnyReg,AnyXY],AnyIO>	Move Peripheral Data
*MOVEP #I24,AnyIO	Move Peripheral Data

Program Control Instructions

Instruction	Description
*Jcc [A12,AnyEa]	Jump Conditionally
*JMP [A12,AnyEa]	Jump
*JScc [A12,AnyEa]	Jump to Subroutine Conditionally
*JSR [A12,AnyEa]	Jump to Subroutine
*NOP	No Operation
*REP [AnyXY,#I12,AnyReg]	Repeat Next Instruction
*RESET	Reset Peripherals
RTI	Return from Interrupt
RTS	Return from Subroutine
*STOP	Stop Processing
*SWI	Software Interrupt
*WAIT	Wait for Interrupt