OIB: 32010 Playback Code

John Strawn

The Droid Works
3301 Kerner Boulevard
San Rafael, California 94901

October 15, 1986

Table of Contents

1. Purpose ......................................................................................................................... 1
2. Prerequisites .................................................................................................................. 1
3. Capabilities of the playback code .................................................................................. 1
4. The 32010-68000 software interface ............................................................................ 3
4.1. The DMEM memory map ........................................................................................... 3
4.2. The declarations, one by one .................................................................................... 3
4.2.1. Locations in DMEM ............................................................................................... 3
4.2.1.1. Using dmem.h .................................................................................................... 4
4.2.1.2. Initialization parameters (code p. 1) ............................................................... 4
4.2.1.3. Run-time inputs/outputs from/to 32010 (page 2) ........................................ 5
4.2.1.4. Error flags (page 2) .......................................................................................... 5
4.2.1.5. Miscellaneous (page 2) .................................................................................... 6
4.2.1.6. Buffer pointer ("wcma") debugging (page 3) ................................................. 6
4.2.1.7. Debugging information (page 3) ....................................................................... 6
5. The building blocks of the code

5.1. Output sample no. N

5.2. Read one sample from NIE:NI

5.2.1. The first sample for a stream

5.2.2. The next six samples of a stream

5.2.3. The last sample of the stream

5.3. Calculate an output sample

5.4. Handle buffer pointer

5.4.1. Overview; Decrement wc

5.4.2. Reading in a New Buffer Pointer

5.4.2.1. Set buffDoneptr

5.4.2.2. Check hang flag

5.4.2.3. Reading in wc and testing it for 0

5.4.2.4. Getting the other buffer pointer entries

5.4.2.5. Testing wcAl_seen

5.4.2.6. Testing attackbit

5.4.2.7. Testing zerobit

5.4.3. No New Buffer Pointer

5.4.3.1. Store updated wc

5.4.3.2. Increment MEM address

5.4.3.3. Get current flags

5.4.3.4. Test abort bit

5.4.3.5. Test zero bit

5.5. Read gain terms

5.6. Calculate stream gain

Ver. 1.0
5.6.1. Silence buffer? ................................................................. 22
5.6.2. Attack or decay? ................................................................. 22
5.6.3. Process attack ................................................................. 22
5.6.4. Start decay? ................................................................. 23
5.6.5. Process decay ................................................................. 23
5.7. Calculate click count .......................................................... 23
5.8. Check hang flag ................................................................. 24
6. Assembling the building blocks ................................................. 24
   6.1. Timing considerations ....................................................... 25
   6.2. The Overall Plan .............................................................. 25
   6.3. Labels in the code ............................................................ 26
   6.4. Streams A2 and B2 .......................................................... 26
   6.5. Optimizing bioz .............................................................. 27
   6.6. Optimizing It and Ita ....................................................... 27
   6.7. Pipelining considerations ................................................ 28
   6.8. The Initialization Code ..................................................... 28
      6.8.1. TMS RESET .............................................................. 29
      6.8.2. Initialize the chip ....................................................... 29
      6.8.3. Initialize constants (page 11) ....................................... 29
      6.8.4. Initialize the D/As (page 11) ......................................... 30
      6.8.5. Read in parameters (page 12) ....................................... 30
      6.8.6. Set flags in D\1\EM (page 12) ....................................... 30
      6.8.7. Initialize attack and decay gains (page 13) ......................... 31
      6.8.8. Read in first buffer pointer (page 13) ; .......................... 31
      6.8.9. Initialize 1\1\EM (debugging) (page 14) .............................. 32
      6.8.10. Initialize internal sample storage (page 16) ..................... 34
      6.8.11. Set up debugging (page 16) ......................................... 34
      6.8.12. Finish up initialization stage (page 17) ............................ 34
      6.8.13. The init_done flag ..................................................... 35
6.9. I/O for Debugging ............................................................ 35
   6.9.1. The hang flag ............................................................. 35
   6.9.2. Copies of buffer pointers ............................................. 35
   6.9.3. Copies of output samples ............................................ 36
   6.9.4. Copies of input samples ............................................. 36
   6.9.5. Dummy buffer pointers .............................................. 36
   6.9.6. What if you get the Sound of Silence? .............................. 37
7. Using the assembler's output .................................................. 37

Ver. 1.0
8. Starting and stopping the playback code........................................................................................................38
9. Table 1. .......................................................................................................................................................... 39
10. Appendix A. The playback code (listing).................................................................................................... 39
1. Purpose

This manual is intended for anyone who must use, understand, or modify the playback code installed in the on-board 32010.

2. Prerequisites

We assume that you understand the OIB Programmer's Manual (in particular the section on the PAs for the 32010), the TMS 32010 chip and its instruction set, and the as320 assembler. You should have read "OIB: The 32010-68000 Software Interface." The playback code itself, /ut/u0/john/odd/play2.a, is included as an appendix to this document. We expect that the gentle reader will follow the code on a line-by-line basis while reading this manual. Page numbers in this manual refer to the code listing included as Appendix A.

3. Capabilities of the playback code

The playback code in play2.a will do the following for you:

1) read in as many as four sounds ("streams") already placed into :MEM, presumably by the SCSI. The 32010 reads buffer pointers placed into D:MEM by the 68000. The buffer pointer can pointer to any arbitrary location within a buffer in :MEM; this is how trim head and trim tail are implemented. Then the 32010 reads samples from :MEM until the buffer end is reached. Then the 32010 sets a "buffer done" flag in the buffer pointer in D:MEM and reads in a new buffer pointer.

---

1This file lives as /ut/u0/john/odd/play2.f.
2) send out two channels to the A/D.

Following the nomenclature adopted for the DASI, the two D/A channels are arbitrarily called A and B. In the original version of the playback code, streams A1 and A2 were mixed and sent to channel A, and streams B1 and B2 were mixed and sent to channel B. Streams A1 and B1 belong to one stereo cue, and streams A2 and B2 belong to the other. Only streams A1 and B1 have been implemented so far. The space needed for streams A2 and B2 has been reserved, and in some cases code for streams A2 and B2 is already included (see Section 6.4).

3) for each stream,

a) calculate an amplitude envelope consisting of an attack (fadein), a steady-state, and a decay (fadeout). During initialization, you may specify the attack and decay times, which remain set during all time for all streams. The use of the attack and decay is optional, subject to certain restrictions. During initialization, an attack is started for all implemented streams.

b) test whether the stream consists only of silence (a "silence buffer")

c) test whether to abort the stream now. Abort means to start the decay immediately, and to set the wc entry in the buffer pointer so that a new buffer pointer will be read in, at the latest, when the decay is done. This abort feature is to be reworked for the next version of the playback code.

d) calculate a stream gain based on the channel gain mixed with the attack, decay, or "silence" information.

5) for each channel,

a) read in a channel gain term. If the user has controls over L/R balance and overall gain, then it is the 68000's responsibility to convert those terms into individual channel gains.

b) mix two streams to make one sample.

6) test for certain error states, such as

a) buffer pointer with wc = 0

b) MEM DMA-read hung

c) D/A hung
d) no initial buffer pointer for a stream
e) MEM DMA write hung (only for debugging)

7) calculate and output a "click" count

8) output various debugging information

The playback software in play2.a will allow you to play back two mono "cues" or one stereo cue. You can overlap two mono cues, but each cue will go to a different output channel. Once streams A2 and B2 have been added to the code, then you can a) overlap stereo cues b) overlap two mono cues and have them both go to the same channel.

The playback software in play2.a was not intended for two stand-alone users. If the 68000 code
is careful and clever, play2.a can be used for such a purpose, however. The outline is that the
TMS must always run. User 1 uses only stream Al, and user 2 uses only stream B1. When a
stream is not in use, it reads only a short "silence buffer", and the buffer pointer for the silence
buffer always points to itself. When the user wants to hear a cue, the buffer pointer pointing to
the silence buffer is munged so that it will now point to a buffer pointer that "knows about" the
buffer playing the cue. The end of the cue points back to the silence buffer's buffer pointer,
which has meanwhile been changed to point only to itself again. Note that the abort facility
using TMS reset (see Section 8) will cause the sounds from both users to halt.

4. The 32010-68000 software interface

4.1. The DMEM memory map

The rationale behind the memory map is discussed in more detail in "OIB: The 32010-68000
Software Interface." To review briefly, we have:

<table>
<thead>
<tr>
<th>DMEM Locations</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Branch to startup location</td>
</tr>
<tr>
<td>2</td>
<td>Interrupt vector</td>
</tr>
<tr>
<td>3:7</td>
<td>Reserved</td>
</tr>
<tr>
<td>8:159</td>
<td>Miscellaneous variables</td>
</tr>
<tr>
<td>160:255</td>
<td>Buffer Pointers</td>
</tr>
<tr>
<td>256:4095</td>
<td>The code</td>
</tr>
</tbody>
</table>

In the code, the startup location is labelled "spotinit:" (page 10). Location 0 is specified by the
declaration ";=O" in the code (page 7). The starting location for the code is assigned the loca-
tion DMEM[256] with the declaration ";=256" in the code (p. 10). These are convenient items to
search for in the text editor. In Section 7 below we will discuss how the individual variables in
DMEM[8:159] as well as the buffer pointers in [160:255] are assigned by specific locations by the
assembler.

4.2. The declarations, one by one

In this section, each declaration will be explained, following exactly the order of the declarations
in the code. This will get a little tedious; sorry. You may want to just skim this section, then
go on to the manual, and come back here when you encounter a variable name that makes no
sense.

4.2.1. Locations in DMEM

In many cases the same variable name will be used for the location in DMEM as well as the
location in the on-chip RAM. It is important to be able to distinguish the two uses. Therefore,
by convention, all variables that refer to DMEM locations in the source code begin with
"dmem_".
The headings in this subsection approximately match those in the code, starting on the bottom of p. 1.

4.2.1.1. Using dmem.h

Normally, a declaration in the assembler's input is of the form

\[ \text{dmemJILfoo} = \text{xxx} \]

where dmemULfoo is the variable name and xxx is some number, meaning (in this case) a DMEM location. The problem arose during development of this code that the 68000 handler must have the same D:MEM locations as the assembler code. Mike Shantzis helped immensely by setting up a common .h file and a shell script to include it. The details are given in "OIB: The 32010-68000 Software Interface." We mention this here because the code contains statements such as

\[ \text{dmem_.decayincr} = \text{PLAY2_decayincr} \]

In this case, dmem_decayincr is the variable name as found in the 32010 code. PLAY2_decayincr is defined in the .h file. In almost all cases, the "PLAY2_" forms of the variable are used exactly once in the code -- right in this declaration section start on p. 1. Also note that dmem.h contains declarations for PLAY1_ as well as PLAY2_; you may even find PLAY3_ in dmem.h. I found it best to create a different set of variables for each version of the playback code, as the quantity and mixture of variables changed from version to version.

In the list of parameters that follows, the default values given in the source code (pp. 7-10) are listed following the variable name.

4.2.1.2. Initialization parameters (code p. 1)

dmem_abortcnt (1414) When the abotbit for a buffer pointer i set, we is set to the quntity (we MIN dmeULabortcnt). In play3.a, this variable is to be removed. See Section 5.4.3.4.

dmem da_loop_times (1000) How often to loop (on the BIO signal) before deciding that the D/A is hung. See Section 5.1.

dmem mem_times (1000) How often to loop (on the BIO signal) before deciding that a DMA read from MEM has hung. See Section 5.2.

dmem_decayincr (Ox5FFF) The time constant for the one-pole that generates the fadeout. See Sections 5.6.5 and 6.8.7.

dmem_attackincr (Ox5FFF) The time constant for the one-pole that generates the fadein. See Sections 5.6.3 and 6.8.7.

dmem_maxpos (Ox7FFF) Do not change this value from Ox7FFF in the source code. Storing this value in DMEM is a cheap and easy way to get the highest possible maximum number, which is needed for several calculations.
dmem_first_wcmaAl (0) Pointer to the first buffer pointer for stream Al. See Section 6.8.8.

dmem_first_wcmaBl (0) Pointer to the first buffer pointer for stream Bl. See Section 6.8.8.

Pointers to the first buffer pointers for streams A2 and B2 are currently commented out.

4.2.1.3. Run-time inputs/ outputs from/to 32010 (page 2)

dmem_scnthi, dmem_scntlo (0) A 32-bit click counter. See Section 5.7.

dmem_init_done (0) Set to 1 by 32010 at end of initialization. See Section 6.8.13.

dmem_aGain, dmem_bGain (OxFF) Gains for A and B channel, respectively. See Sections 5.5 and 5.6.

dmem_hang (0) Code "hangs" when this location goes non-zero. See Sections 5.8 and 6.9.1.

4.2.1.4. Error flags (page 2)

All "error" flag names begin with "dmem_err_".

dmem_err_da (0) Set to non-zero when D/A hung. See Section 5.1.

dmem_err_rDMA (0) Set to non-zero when DMA read from :MEM hung. See Section 5.2.2.

dmem_err_wDMA (0) Set to non-zero when DMA write to :MEM hung. Used only in debugging initialization under control of INIT:MEM. See Section 6.8.9.

dmem_err_no_wcmaAl (0) Set to non-zero if the initialization code finds zero in dmem_first_wcmaAl. See Section 6.8.8.

dmem_err_no_wcmaBl (0) ditto for stream Bl.

Similar flags for streams A2 and B2 are still commented out.

dmem_err_wcAl_zero (0) Goes non-zero when a buffer pointer containing wc=0 is read in for stream Al. See Section 5.4.2.3.

dmem_err_wcBl_zero (0) Ditto for stream Bl.

dmem_err_wcAl_seen (0) Goes non-zero when a buffer pointer containing a non-zero bufdone flag is read in for stream Al. This does not necessarily indicate an error condition. See Section 5.4.2.5.

dmem_err_wcBl_seen (0) ditto, for stream Bl.
4.2.1.5. Miscellaneous (page 2)

dmem_rgarbage (0) Set to non-zero after first buffer pointer for stream Al is read during initialization. See Section 6.8.8.

dmem_wgarbage (0) Ditto, for stream Bl. This variable name is a hold-over from earlier debugging code.

dmem_tmp (0) In the current code, this is set to non-zero after the flags have been set in DMEM. In general, this location is used for random debugging purposes. See Section 6.8.8.

4.2.1.6. Buffer pointer ("wcma") debugging (page 3)

dmem_err_wcAl_copy This points to the first of several locations into which the current buffer pointer for stream Al is copied once during each click. See Section 6.9.2.

dmem_err_wcBl_copy Ditto, for stream Bl.

4.2.1.7. Debugging information (page 3)

All of the following variables (see the code itself) are DMEM copies of variables from on-chip RAM that will be discussed in the next section.

4.2.2. On-chip RAM locations (page 3)

Many of these names match the names for the variables in DMEM as given in the previous sections. In the case of many such matches, a TBLR instruction is used to read in the value from DMEM as part of initialization. See Section 6.8.

In this section, the number in parentheses after the name of the variable shows the number to which the quantity is explicitly initialized by the 32010.

4.2.2.1. On-chip RAM memory map

There are 144 locations of on-chip RAM. To use [0:127], the RAM page pointer is set to 0 with the ldpk 0 instruction during initialization (see Section 6.8.2). The current code does not use locations [128:143]. To do so, you would have to issue a ldpk 1 instruction to get at those higher-order locations, followed by a ldpk 0 instruction. The current code uses almost all of the lower-order 128 RAM locations. If you add streams A2 and B2, then either you will have to free up some locations in RAM, or you will have to use the higher "page" in RAM.
4.2.2.2. Error and control flags

da_loop_times How often to loop (on the BIO signal) before deciding that the D/A is hung. See Section 5.1 and dmem_da_loop_times in Section 4.2.1.2.

biozdasi_new_samp (1) Contains 1 to select the hardware signal dasi_new_samp for BIOZ loops. See drawings dasi5 and dspl as well as Sections 5.1 and 6.5.

mem_times (1000) How often to loop (on the BIO signal) before deciding that a DMA read from MEM has hung. See Section 5.5.2 and dmem_mem_times in Section 4.2.1.2.

biozMEMdone (0) Contains 0 to select the MEM done flag for bioz loops. See drawing dspl and Sections 5.2.2 and 6.5.

biozaudio In the playback code, this is commented out. For the record code, it is used to signal that the A/D is not working. See drawing dspl.

hang (0) Current contents of dmem_hang. Read once per click. See Section 5.8 and 6.9.1, and dmem_hang in Section 4.2.1.3.

4.2.2.3. Parameters (page 4)

attackincr The time constant for the one-pole that generates the fadein. In play2, this is read in once during initialization. See Section 6.8.7 and dmem_attackincr in Section 4.2.1.2.

decayincr ditto for fadeout.

attackGl, attackG2, decayGl, decayG2 Constants calculated during initialization to optimize execution of the one-poles that implement attack (fadein) and decay (fadeout). See Section 6.8.7.

scnthi, scntlo (0) A 32-bit click counter, copied out to DMEM once per click. See Section 5.7 and dmem_scnthi, dmem_scntlo in Section 4.2.1.3.

decayGlAl During attack and steady-state of a stream's amplitude envelope, this is kept at 0. During decay decayGl is copied here to 1) signal that the decay is in progress and 2) deliver the gain value to the one-pole code. See Sections 5.6.4 and 5.6.5.

decayGIBI Ditto for stream Bl. Note that since decayGl is calculated exactly once during initialization, all streams use the same decayGl value.

4.2.2.4. Buffer pointers (page 4)

The structure of the buffer pointers is discussed in "OIB: The 32010-68000 Software Interface."

4.2.2.4.1. Stream Al

In the on-chip RAM, we first have declarations in the code for the locations wcAl, memhiAl, memloAl, nextWcmaAl, and sdecayAl. When a new buffer pointer is read in, these locations
are copied from the corresponding DMEM locations.

Also, flaglocA1 is calculated when the buffer pointer is read in. This location points to the "flags" stored in memhi, which will be read in later (see Section 5.4.3.3) into flagA1.

Then the buffer pointer code (see Section 5.4.2.4) calculates a pointer (stored in bufdoneptrA1) to the bufdone flag.

decayA1 and attackA1 store the current decay and attack values, respectively. gainA1 is the product of aGain times either decayA1 or attackA1, whichever is currently running.

4.2.2.4.2. Stream B1
The locations just described for stream A1 are duplicated for stream B1.

4.2.2.4.3. Streams A2 and B2
gainA2 and gainB2 are required by the code for calculating a sample value (see Section 5.3). None of the other buffer pointer entries for these two streams currently have on-chip RAM locations set aside for them. Indeed, if those two streams are ever implemented, then either a) some other locations will have to be freed up from the on-chip RAM or b) you'll have to use the higher "page" of the on-chip RAM memory, which will unfortunately involve the ldpk instruction.

4.2.2.5. Storage for input samples (page 5)

Next we find 24 locations for storing samples input from MEM (see Section 5.2). There are 8 locations for stream A1, followed by 8 locations for stream A2, followed by 8 locations for stream B1. Originally I set aside 8 locations for stream B2 as well, but as you will see in the next section, some of those locations have to be used.

We expect these 24 (or 32, with stream B2) to be consecutive locations. They are initialized to 0 by the 32010; see Section 6.8.10.

4.2.2.6. wcma_cntA1 and wcma_cntB1 (page 6)
These two locations are incremented by 1 every time a new buffer pointer is read in. They are written out to MEM once per click (see Sections 5.4.2.4 and 5.4.2.5).

4.2.2.7. Storage for mixing input samples to output samples (page 6)

aGain (0) This is a copy of dmem_aGain given in Section 4.2.1.3.

bGain (0) This is a copy of dmem_bGain given in Section 4.2.1.3.
In outsampAl through outsampB8, we collect the samples that will be sent out to the D/A. outsampAl has the result of aGain X (attack or decay) X insampAl + aGain (attack or decay) X insampA21. See Section 5.3. We expect all 16 locations to be consecutive. They are all set to 0 during initialization; see Section 6.8.10.

4.2.2.8. Miscellaneous constants (page 6)

tmp This is used for a variety of nefarious purposes. This location is not necessarily related with dmem_tmp discussed in Section 4.2.1.5.

maxpos (Ox7FFF) During initialization, dmem_maxpos is read to here. The values in dmem_maxpos (Section 4.2.1.2) and maxpos must never be changed.

zero through eight hold exactly the numerical value given by the name.

twobits holds the quantity 011, which is used for a) setting up auto-increment in MEM addressing and b) starting another MEM read cycle. See Section 5.2.

abortcnt Filled once during initialization with the value in dmem_abortcnt (Section 4.2.1.2). When the abortbit for a buffer pointer is set, wc is set to the quantity (we MIN abortcnt). Note that the flags are read and tested for abort once per click. In play3.a, this variable is to be removed. See Section 5A.3.4.

attackbit (Ox8000) This is a mask for the "start attack" flag in the memhi entry in the buffer pointer. We AND this quantity with memhi to figure out whether to start a fadein now. See Section 5.4.2.6.

zerobit (Ox4000) This is a mask for the "silence buffer" flag in the memhi entry in the buffer pointer. We AND this quantity with memhi to figure out whether the buffer contains all Os. See Section 5.4.2.7 and 5.4.3.5.

notabortbits (Ox7FFF) If the abort bit is found set in a buffer pointer out DMEM, then we use this quantity to turn off the abort bit in DMEM. See Section 5.4.3.4. As I was writing this manual, I realized that the quantity in maxpos is also Ox7FFF, so you could save the location notabortbits right there. But notabortbits is to be removed in the next version of the play code anyway.

5. The building blocks of the code

The code is organized so that one pass through the main code loop produces eight stereo samples from a maximum of four streams. In order to pull this off easily and efficiently, we first create little "building blocks" that do compact lo-level functions like reading in a sample from MEM, writing out a sample to the D/A, etc. Section 5 covers those building blocks. Section 6 will show how they are duplicated as often as needed and assembled into the whole program. Section 6 will also discuss the necessary initialization.

In the code examples here, you will see some labels (always in the left-hand column, followed by a ":"). Typically, a label ends in A1, A2, B1, or B2, showing which stream is being handled. Another kind of label is used for outputting samples 1 through 8; in this case; the label is of the form fooN, where N is 1 through 8. Finally, there are miscellaneous labels that are of the form
Lxyyy where x tells you the sample number from 1 to 8 that is currently being output (this is the second column of Table 1) and yyy is the number of the label within that sample period. Other labels will be discussed in Section 6.3.

5.1. Output sample no. N

During the click loop, the following block of code is invoked eight times, once for each stereo sample to be output. The code given here is reproduced verbatim from play2.a (page 17), and is for the very first (stereo) sample output. I have merely added line numbers in the comment field to simplify the discussion.

```c
samplel: out biozdasi_new_samp, PA5; 1
zals· da_loop_times; 2
Lll: DABIOZ L12; 3
subs one; 4
bgz Lll; 5
lack dmem_err_da; 6
tblw one; 7
zals da_loop_times; 8
b Lll; 9
L12: out outsampAl, PA6; 10
out outsampBl, PA7; 11
```

(These blocks of code are labelled "sampleN").

The core of this excerpt is lines 10 and 11, where the samples are sent to the DASI. outsampAl and outsampBl are on-chip RAM locations for the A and B output samples, respectively. PA6 and PA7 are the DASI D/A interface registers for samples A and B, respectively.

Before the samples can be output, we must ensure that the D/A registers are clear, so that we don't prematurely overwrite a sample that the D/A hasn't seen yet. Lines 1-9 take care of that. In effect, these lines synchronize the 32010 code with the sample rate of the D/A.

The process centers on the BIO signal available in the TMS, and the BIOZ instruction that reads that BIO line. On the OIB, there are several signals that can be fed to the BIO line. Line 1 selects the dasi_new_samp signal from the hardware to go to BIO.

Lines 3 through 5 are a "buzz loop" waiting for the dasi_new_samp signal. Normally, DABIOZ is set to bioz with the line

```c
#define DABIOZ bioz
```

that occurs at the beginning of the code. When you run the source code through cc, cc replaces DABIOZ with bioz. One can also put just "b" instead of "bioz" in the above #define, so that BIO is never tested and you always jump to L12. This is used for stand-alone debugging.

With DABIOZ set to bioz, we branch to L12 when the BIO line goes to 0, which means that the D/A has read the previous sample. While BIO is nonzero, we fall through to line 4. Now in line 2, the accumulator (AC) has been initialized with the user-specified quantity da_loop_times. In line 4, the AC is decremented by 1. In line 5, we jump back to L11 as long as AC >= 0. Once AC reaches 0 (which indicates an error condition), we execute lines 6 through 9 because we believe that for some reason the DASI has hung. Lines 6 and 7 set the DMEM location pointed
to by dmem_err_da to be 1. Then we re-initialize the AC with da_loop_times, and branch back to the start of the "buzz" loop in line 3. The TMS continues to loop here in lines 3-9 until the BIO signal reaches 0.

The 68000 must occasionally check the dmem_err_da location; if it ever goes non-zero, then an error message should be printed out. The quantity in dmem_err_da shows where the error occurred. In the above code, we wrote out a "1"; for sample 2 (page 20 in the code), we write out a "2", and so on, through sample 8. The value in dmem_err_da thus points to the exact part of the code where the error was detected.

Actually, I believe that the TMS normally falls through to line 4 exactly once (maximum). Copies of the above block of code are spaced throughout the code, as we shall see later, so that little or no time is spent in this BIOZ loop. The quantity da_loop_times is initialized to a default value of 1000 in the .a file (this can be overwritten by the 68000). I have never observed a spurious error message in dmem_err_da with da_loop_times set to 1000.

To debug the above loop of code, simply initialize da_loop_times to 0. This will have the effect that you will always drop into line 6 (assuming that BIOZ didn't branch the first time through), so you are guaranteed to have dmem_err_da set to non-zero.

5.2. Read one sample from MEM

As we will see later, we read eight samples for one stream at the beginning of the click loop, then read eight samples for the next stream, and so on. There are thus three different blocks of code: one for the first sample read, six for the next six samples, and one for the final (eighth) sample.

5.2.1. The first sample for a stream

The following three lines read the first sample from stream A1:

```
out
out
out
memloAl, PAO
memhiAl, PA1
twobits, PA3
```

(In the code, the last of the three lines above is labelled "startXXN", i.e. startA11 starts the first sample for stream A1, startB21 starts the first sample for stream B2, and so on).

The first two lines set up the address for MEM, which is contained in PAO and PA1. The concatenation of memhiAl[0:6] and memloAl (supplied from the current buffer pointer for stream A1) form a 23-bit address which is left-shifted by the hardware to form a 24-bit address into MEM (with the LSB set to 0). In the third line, we send the quantity twobits (initialized during TMS 32010 startup to 011) from the on-chip RAM to PA3. The effect of this is 1) to start a MEM read cycle; 2) enable auto-increment in the MEM address hardware.
5.2.2. The next six samples of a stream

The following block of is duplicated 6 times for each stream. This particular excerpt is the code for the second sample of stream A1 (pp. 17-18).

```
out
zais
LOO:  bioz
      subs
bgz
lack
tbiw
zais
b
readAll: in
startA12: out
```

(biozMEMdone, PA5; 1
mem_times; 2
readAll; 3
one; 4
LOO; 5
dmem_err_rDMA; 6
one; 7
mem_times; 8
LOO; 9
insampAll, PA2; 10
twobits, PA3; 11
)

(In the code, the last two lines are labeled so that the label for line 10 refers to sample N, and the label for line 11 refers to sample N+1. To give another example, you would find readB17 on line 10 for reading the 7th sample of stream B1, and on line 11 in the same block of code you would find "startB18;", meaning to start the read for the 8th and final sample of the same stream).

This code should look familiar to those who have read the section above on doing D/A output (and if you haven’t yet, you should now). The core of this code is in lines 10 and 11. First (line 10) we read into the on-chip RAM location the sample from MEM, which is contained in PA2. Then (line 11) we send twobits out to PA3, which (as we saw before) 1) starts a new MEM read; 2) (re-) enables auto-increment in the MEM address hardware.

Lines 1 through 9 are a "buzz" loop to check whether MEM is hung. In line 1, we change the BIO line so that it now looks at the MEMdone signal in the hardware (it was looking at dasi_new_samp before). The branch at line 3 goes to line IO once BIO goes to 0. This means that the new datum has been successfully fetched from MEM.

In line 2 we initialize the accumulator with me:m_times (which is initialized to 1000 in the .a file). Line 4 decrements the AC by 1 each time that BIOZ fails to branch. When the AC reaches 0, we execute lines 6-9, because we believe that MEM has hung doing a read DMA. First (lines 6-7) we set the DMEM location giYen by dme:m_err_rDMA to be 1. Then (lines 8-9) we re-initialize the AC to me:m_times and go back to the "buzz" loop.

As before with the D/A code, the quantity written out to dme:m_err_rDMA shows the sample number (1 through 8) where the MEM read hung. The 68000 should check dmem_err_rDMA occasionally, and print out an error message if that location ever goes non-zero.

5.2.3. The last sample of the stream

The following code excerpt (from p. 23) reads in the last (eighth) sample of stream AL.

```
out
zais
L213:
bgz
lack
```

(biozMEMdone, PAS
mem_times
readA18
one
L213
dmem_err_rDMA
)
(In the code, the label on the last line is of the form readXXN).

This code is identical to that given in the previous section except in two places:

1) The out biozMEMdone in the first line is commented out. This is because the signal biozMEMdone is still in PA5 from before, and there is no reason to waste two instruction cycles repeating old information. In Section 6.5, we will see why the (commented) line is left in the code.

2) There is no line reading

\begin{verbatim}
out twobits, PA3
\end{verbatim}

at the end of this code. This is because there is no need to start another MEM read for this stream. The next MEM read will be for a different stream, which means that the MEM address in PAO and PA1 must be reloaded before the MEM read can be started.

5.3. Calculate an output sample

Recall that the sample for DASI channel A is created by combining the samples for streams A1 and A2, each multiplied by its own gain term. This is accomplished in the following 7 lines of code, taken from page 33:

\begin{verbatim}
lt insampA11; 1
mpy gainA1; 2
pac ; 3
lt insampA21; 4
mpy gainA2; 5
lta insampB11; 6
sach outsampA1, l; 7
\end{verbatim}

(In the code, these blocks of code are given a commented label of the form CALCYZ, where Y is replaced with A or B for the given channel and Z is the sample number varying from 1 through 8).

All of the memory locations in this excerpt are from the on-chip RAM in the 32010. Lines 1 through 3 multiply a sample from stream A1 by the gain for stream A1, and stuff the result into the AC. Lines 4 and 5 multiply a sample from stream A2 by the gain for stream A2. In line 6, we 1) add the product from stream A2 into the AC, where the product for stream A1 already lives; 2) we pre-load a sample from stream B1 into the 32010’s T register, where it will live until we reach a similar block of code. In Section 6.6 we will see how the lta in line 6 can allow us to omit the instruction in line 1 in some instances. The sach in line 7 (with a left-shift of 1) stores the product in a RAM location reserved for the first (of eight) samples for the A side of DASI. The left-shift is necessary because of the fractional multiplies.
5.4. Handle buffer pointer

This is easily the largest and most complicated building block in the playback code. This section of the manual relies heavily on the explanations given in "OIB: The 32010-68000 Software Interface." Many of the variable names given here will be easier to understand is you remember that "wcma" is obscure shorthand for "buffer pointer," and "we" stands for click count.

5.4.1. Overview; Decrement we

The buffer pointer handler has a short initialization (covered in this section) which branches to one of two large blocks. The first block actually reads in a new buffer pointer and takes care of some messy details. The other block (which is the block that is executed most of the time) takes care of a lot of miscellaneous things. At one point, both branches were exactly (!) the same length in terms of machine cycles. When it became clear that streams A2 and B2 would not be implemented, I stopped being so careful.

All of the buffer pointer handler code excerpts will be taken from the code for stream Al, pp. 18-20.

The buffer pointer handler starts out by simply decrementing wc and executing a branch:

```
zals wcAl 1
subs one 2
bnz noWcmaAl ; 3
```

In English: we load wcAl, the click count for stream Al, into the AC, and decrement it. When AC = 0, we fall through to the instruction following the bnz. When AC != 0, we branch (in this case, to noWcmaAl); note that after the branch, AC still has the decremented wc, which will quickly be squirreled away for the next pass through the code.

5.4.2. Reading in a New Buffer Pointer

5.4.2.1. Set bufdoneptr

The first thing to do is to signal to the 68000 that the buffer pointed to by the current buffer pointer is now empty. We do this with two lines of code:

```
zals bufdoneptrAl tblw minus
```

As we will see in Section 5.4.2.4, a new bufdoneptr is calculated every time a new buffer pointer is read in. Remember that bufdoneptr points to the "buf done" flag for the old buffer pointer, not the "buf done" flag for the buffer pointer that we're about to read in.
5.4.2.2. Check hang flag

Actually, the next little bit of code belongs under the heading of "debugging."

```
readWcAl:
    lack  dmem_wcmaHangAl
    nop
    foe: lack  dmem_wcmaHangAl
         tblw  nextWcmaAl  Ox7D15
         tblr  tmp
         zals  tmp
         bnz  foe
```

In the last three lines, we read the contents of dmem_wcmaHangAl into the on-chip RAM location given by tmp, then check tmp for 0. As long as dmem_wcmaHangAl is non-zero, we will loop here.

The key here is the nop. If you replace it with the tblw nextWcmaAl instruction in the comment field (which assembles to Ox7015 under the current variable assignments), then you are guaranteed to have a non-zero dmem_wcmaHangAl. This means that every time a new buffer pointer for the current stream is read in, the TMS will hang in this loop; and you have the added luxury of being able to read, in location dmem_wcmaHangAl, where the buffer pointer starts that you're about to read into the TMS. Alternately, you can set dmem_wcmaHangAl by hand, and the TMS code will hang in this loop the next time that a new buffer pointer is read in for this stream. The code is written so that the looping occurs until you, by hand, set dmem_wcmaHangAl to 0.

5.4.2.3. Reading in we and testing it for 0

As discussed in "OIB: The 32010-68000 Software Interface," we must ensure that we never is 0:

```
readWcAl:
    zals  nextWcmaAl
    tblr  wcAl
    zals  wcAl
    bnz  wcAlOK
    lack  dmem_err_wcAl_zero
    tblw  nextWcmaAl
    b  readWcAl

wcAlOK:
```

(In the code, the first label above is of the form readWcXX.)

The first two lines read in wc for the new buffer pointer. Obviously, nextWcmaAl points to the start of the new buffer pointer, and wcAl is the on-chip RAM location that will hold the click count for the new buffer in MEM. In line 3, we load that new wcAl into the AC, and test it in line 4 for 0. For non-zero wcAl, we branch to wcAlOK and go on our merry way.

For wcAl=0, which is an error condition, we fall through and start hanging in a loop. First, we set the DMEM location given by dmem_err_wcAl_zero to the non-zero value equal to nextWcmaAl. The 68000 should occasionally check dmem_err_wcAl_zero; if it goes to non-zero, an error message should be printed out. After the error flag has been set, we loop back to readWcAl; and continue looping until we pointed to by nextWcmaAl is non-zero.
5.4.2.4. Getting the other buffer pointer entries

The first three lines in the next block simply increment the on-chip RAM location wcma_cntAl by 1. This quantity is eventually written out to a D1v1E1v1 location for debugging purposes. Lines 4-16 are what we're really interested in:

\[
\begin{align*}
\text{wcA1OK:} & \quad \text{zals } wcma\_cntAl, 1 \\
& \quad \text{adds one } 2 \\
& \quad \text{sacl } wcma\_cntAl, 0, 3 \\
& \quad \text{zals } nextWcmaAl, 4 \\
& \quad \text{adds one } 5 \\
& \quad \text{tblr } memhiAl, 6 \\
& \quad \text{sacl } flaglocAl, 0, 7 \\
& \quad \text{adds one } 8 \\
& \quad \text{tblr } memloAl, 9 \\
& \quad \text{adds one } 10 \\
& \quad \text{tblr } nextWcmaAl, 0, 11 \\
& \quad \text{adds one } 12 \\
& \quad \text{tblr } sdecayAl, 13 \\
& \quad \text{adds one } 14 \\
& \quad \text{sacl } bufdoneptrAl, 0, 15 \\
\end{align*}
\]

\[
\begin{align*}
\text{seenA1:} & \quad \text{tblr tmp } 1 \\
& \quad \text{zals tmp } 2 \\
& \quad \text{bz notseenA1 } 3 \\
& \quad \text{lack dmem\_err\_wcA1\_seen } 4 \\
& \quad \text{tblw wcma\_cntAl } 5 \\
& \quad \text{zals bufdoneptrAl } 6 \\
& \quad \text{b seenA1 } 7 \\
\end{align*}
\]

As we saw before, nextWcmaAl points to the beginning of the new buffer pointer. We've already read in we, so we increment nextWcmaAl by 1 in line S, and read in memhiA1. Now the value in AC is squirreled away into flaglocA1 for later use; we want to remember the address of memhi so that the flags contained in memhi can be read in later if need be.

In lines 8-14, we increment AC and read in the next four entries of the buffer pointer. In lines 15 and 16, we squirrel away the location of the bufdone flag for the new buffer pointer; this will be used when the buffer is done.

5.4.2.5. Testing wcA1_seen

As we leave the last block of code discussed in the previous section, AC contains bufdoneptr. We can use that AC value for a tblr to see whether the value at bufdoneptr is zero or non-zero from any previous use:

\[
\begin{align*}
\text{seenA1:} & \quad \text{tblr tmp } 1 \\
& \quad \text{zals tmp } 2 \\
& \quad \text{bz notseenA1 } 3 \\
& \quad \text{lack dmem\_err\_wcA1\_seen } 4 \\
& \quad \text{tblw wcma\_cntAl } 5 \\
& \quad \text{zals bufdoneptrAl } 6 \\
& \quad \text{b seenA1 } 7 \\
\end{align*}
\]

In some cases, it is an error for the value at bufdoneptr to be non-zero, because it means that the 68000 hasn't updated the current buffer pointer from the last time that it was used. In other cases, the value at bufdoneptr should be non-zero; for example, if we're looping several times through a silence buffer using the same buffer pointer.

In lines 1 and 2 we load into AC the value from DMEM pointed to by bufdoneptr. If that MEM location is 0, we branch in line 3 and go on our merry way. Otherwise, we set the DMEM
location given by dmem_err_wcAl_seen to the non-zero value given in the on-chip RAM location wcma_cntA1. Then we drop through.

Lines 5 and 6 are currently commented out. If you un-comment them, then you will loop through these seven lines until the DMEM location at buffdoneptr goes to 0.

5.4.2.6. Testing attackbit

Now we see if it's time to restart the attack for the current stream:

```
notseenAl:
  zals  memhiAl
  and  attackbit
  bz   nonewattackAl
  zac
  sach  attackAl,0
  sach  decayGlAl,0
  b    wasWcmaAl

nonewattackAl:
```

As you know, we just read in rnemhi. The on-chip RAM location attackbit contains a 1 in bit 15. After the and in line 2, we branch if the AC is 0, meaning that the attack is not to be reset.

To reset the attack, you set attackAl to be 0; in a one-pole of the form y[n]=x[n]-y[n-1], attackAl contains the y[n-1] term. It is also necessary to zero out decayGlAl; this has the effect of turning off any decay that may be happening or may have happened. Finally, we branch in line 7 to wasWcmaAl:, which ends the buffer pointer processing for this stream in this click.

Line 4 is commented out. We need a zero in lines 5 and 6, but we get it "for free" because the result of the and in line 2 is guaranteed to be 0 in the high half of AC. So we use sach's in lines 5 and 6 to get the 0 from AC hi.

. 5.4.2.7. Testing zero bit

It can happen, if you don't reset the attack, that you still should check the zero bit. One would hope that the 68000 will not set the zero bit on the same buffer pointer that it sets the attack bit; doing so would cause the 32010 to work much harder than necessary. Even if this happens, the zero bit will be caught later. So we have:

```
nonewattackAl:
  zals  memhiAl
  and  zero bit
  bz   wasWcmaAl
  sacl  zeroAl,0
  sach  gainAl,0
  b    wasWcmaAl
```

As in the previous section, we read memhiAl into AC, but this time we and it with zero bit. If the result is 0, then we branch to wasWcmaAl, and the buffer pointer processing is finished for this stream for this click.
Otherwise we store 1) a non-zero term from (pay attention now) ac \(\text{lo}\) into zeroAl, and 2) a zero into gainAl from ac hi. Remember that ac hi is guaranteed to be 0 after the and operation of line 2. The non-zero value in zeroAl means that the buffer pointer points to a zero buffer. (zeroAl will be reset to 0 sometime later in the code that alternates with reading in the buffer pointer.) And the zero in gainAl guarantees that the stream will equal 0 when the output sample is calculated. Recall from the section "Calculate an output sample" that we do

\[
\begin{align*}
\text{lt} & \quad \text{insampAll;} 1 \\
\text{mpy} & \quad \text{gainAl;} 2
\end{align*}
\]

Obviously, if gainAl is 0, then the result of the multiply will be 0 too.

This completes the processing of a new buffer pointer.

5.4.3. No New Buffer Pointer

We now turn to the code executed on those clicks when a new buffer pointer is not read in. This is the code that is executed most of the time, since we read in a new buffer pointer relatively rarely.

5.4.3.1. Store updated wc

Recall from Section 5.4.1 that we reached this point with the decremented value of wc still in the accumulator. The first order of business is to squirrel away that decremented value:

\[
\text{nowcmaAl: sacl wcAl,0}
\]

(In the code, the label is of the form nowcma:XX).

5.4.3.2. Increment MEM address

Now there is the mundane task of incrementing the concatenation of memhi and memlo. In other words, we need a double-precision addition:

\[
\begin{align*}
\text{zalh} & \quad \text{memhiAl} \\
\text{adds} & \quad \text{memloAl} \\
\text{adds} & \quad \text{eight} \\
\text{sach} & \quad \text{memhiAl,0} \\
\text{sacl} & \quad \text{memloAl,0}
\end{align*}
\]

Obviously, the first two lines load the AC with the double-precision address; we increment that number by 8, since there are 8 samples per click, then store the result. Note that the flag bits in memhi[7:15] may be changed as a result of this operation. This is why we cannot use the on-
chip RAM copy of memhi any more as a source for those flag bits.  

5.4.3.3. Get current flags

Instead, we go out to DMEM and read in memhi from there. Recall from Seeton 5.4.2.4 that flagloc points to memhi in the current buffer pointer. With these two instructions:

\[
\begin{align*}
\text{zals} & \quad \text{flaglocAl} \\
\text{tblr} & \quad \text{flagAl}
\end{align*}
\]

we read, into the on-chip RAM location flagAl, the current contents of memhi in DMEM. In the current implementation, we test two flag bits. This means, you see, that the 68000 can set those bits in memhi even after the buffer pointer has been read in.

5.4.3.4. Test abort bit

Note: This feature is to be changed in the next release of the 32010 playback software, so I do not recommend scratching your head about this section of the manual.

In the first three instructions, we load the flags into the AC, do an and with the on-chip RAM location abortbit, and test for 0. If the AC has zero, then we branch to line 18 and continue on our merry way.

If abortbit has been set, then we must do two things: 1) start the decay, and 2) reset wc so that a new buffer pointer will be read in as soon as reasonable. That is, we will read in a new buffer pointer as soon as the decay is done, unless wc is such that it will cause a new buffer pointer to be read in before the decay is done anyway. The quantity abortcnt from the on-chip RAM gives the value that we'll overwrite we with. abortcnt normally corresponds to the length of the

2 someone should calculate the maximum length of a buffer and figure out what the highest-order bit is that could be affected by incrementing memhi. Then we could limit flags in memhi to the "safe" bits and not worry about squirreling away an extra copy in on-chip RAM.
we
So, in line 4 we start with the current we, loaded into AC hi. If its sign bit is on, we know that we is very large, and we will proceed (by jumping to stampAl) to reset we to abortcnt.

If the sign bit of wc is off, then in line 6 we subtract (from ac hi) abortcnt itself. If we \( \leq \) abortcnt, then in line 7 we skip ahead to line 10. For wc \( \geq \) abortcnt, we stamp wc to be equal to abortcnt in lines 8 and 9.

Now line 10 gets tricky. By now, wc contains either its old value or else abortcnt, whichever is smaller. In line 11 we add 1 to that value, and store the result in sdecayAl. The result is that much later (see Section 5.6.4), when we calculate a new decay term, the decay is guaranteed to kick in.

Finally, in lines 13-17, we turn off abortbit in the copy in DMEM, so that we don't re-initialize the decay the next time we go through this code. To do this, we and flagAl (loaded into AO in line 13) with the on-chip RAM quantity not abortbits, which has the logical complement of abortbit. The result is stored in on-chip RAM in flagAl, and written out to DMEM in lines 16 and 17.

5.4.3.5. Test zero bit

After all that nonsense, this section will seem trivial.

\[
\begin{align*}
tstoAl: & \quad zals \quad flagAl & \quad 1 \\
& \quad and \quad zerobit & \quad 2 \\
& \quad sacl \quad zeroAl,0 & \quad 3 \\
& \quad bz \quad wasWcmaAl & \quad 4 \\
& \quad sach \quad gainAl,0 & \quad 5 \\
\end{align*}
\]

Recall from the previous section that the on-chip RAM location flagAl has been updated from DMEM. We and it with zerobit in line 2 and store the result from AC lo into zeroAl in line 3. If zeroAl is non-zero, it will signal that the current stream is reading from a silence buffer. For a non-zero AO in line 4, we zero out gainAl in line 5. Recall that AO hi is guaranteed to be 0 as a result of the and in line 2, and that a gainAl = 0 guarantees that the stream will be 0 as it is mixed and sent to the DASI.

We have now reached wasWcmaAl. This completes the code executed when a new buffer pointer is not read in. Congratulations on having made it this far!

5.5. Read gain terms

This building block is actually fairly straightforward. Once per click, we read in a new gain term for each of output channels A and B. It is the job of the on-board 68000 to turn the user inputs into exactly one gain term for exactly each channel. For example, if the user is twiddling a knob for overall loudness and another knob for channel balance, then those knobs must be melded into two gain terms.
So in instruction 1 we load the accumulator with the address in DMEM of the gain term for channel A, and in instruction 2 we read that term into the on-chip RAM location aGain. Instructions 3 and 4 repeat the process for the B channel.

5.6. Calculate stream gain

The stream gain is calculated by taking into account a number of elements:

a) is the current buffer a "silence" buffer? If so, set the gain to 0;

b) is the attack in progress? For convenience, we say that the attack is in progress during the fade-in as well as during the "steady-state" in the stream's amplitude envelope. If the attack is in progress, calculate the next value of the attack. Notice that a new attack value is calculated only once per click.

c) is the decay in progress? If so, calculate a new decay value.

The final stream gain is the product of the channel gain (see above) times either the attack or the decay, whichever is in progress. If this is a "silence" buffer, then the stream gain stays at 0.

That having been said, then, here is the code:

```
zals zeroAl 1
bnz LA1G2 2
zalh decayGlAl 3
bnz LA1Gl; 4
zalh attackGl; 5
lt attackAl; 6
mpy attackG2; 7
lta aGain; 8
sach attackAl,1; 9
mpy attackGl; 10
pac ; 11
sach gainAl,1; 12
wcAl; 13
sub decayAl,14
bhz LA1G2; 15
zals maxpos; 16
sub attackAl,17
scl decayAl,0; 18
zals decayGl; 19
scl decayGlAl,0; 20
b
LA1Gl: lt decayAl,22
mpy decayG2; 23
lta aGain; 24
sach decayAl; 25
zals maxpos; 26
sub decayAl; 27
scl tmp,0; 28
mpy tmp; 29
pac ; 30
sach gainAl,1; 31
```
5.6.1. Silence buffer?

Recall that during the buffer pointer readin, a stream's gain is set to 0 for a "silence" buffer. In instructions 1 and 2, we test zeroAl, which is non-zero when the current stream is reading silence. For non-zero zeroAl, we simply bypass lines 3 through 31 and go on our merry way.

5.6.2. Attack or decay?

decayGlAl in line 3 is a flag. When it is 0, we are in the attack or steady-state part of the stream's amplitude envelope. When it is non-zero, it a) indicates that the decay is happening; b) gives the time constant for the decay. So on non-zero decayGlAl, we jump down to LA1Gl, which is where the decay is processed.

5.6.3. Process attack

But first let's look at the attack processing in lines 5-21. The attack is implemented by a one-pole filter of the form

\[ y[n] = g \times x[n] + (1-g) \times y[n-1] \]

To get the attack (fadein), we use the step response of the one-pole. For the step response, \( x[n] \) simplifies to 1.0. During the initialization code, since \( g \) and \( x[n] \)=1.0 are constant for all time, we calculate the term \( \text{attackGl} = g \times \text{Ox3777} \), where \( g \) is the time constant supplied by the 68000 and Ox3777 is the maximum positive number inside the 32010. Likewise during initialization, we calculate \( (1-g) \) as a constant and store it away as attackG2. This means that the filter reduces to:

\[ \text{attackAl} = \text{attackGl} + \text{attackG2} \times \text{attackAl} \]

where attackAl corresponds to \( y[n] \) and \( y[n-1] \) in the previous formula.

But wait, there's more! You know that the fractional multiply attackG2 \( \times \) attackAl will have to be left-shifted by 1 place. Under normal conditions, we would do the multiply, left-shift and store the product, load it into the AC, add in attackGl, and then store the entire sum into attackAl. This takes too many instructions. One way out is to store attackGl during initialization so that it is right-shifted one bit. That means we can pre-load the AC with the right-shifted attackAl, which happens in instruction 5 above. In instructions 6 and 7 we multiply attackAl times attackG2. Instruction 8 accumulates the product into what was already in AC, and in instruction 9 we left-shift the accumulated sum by 1 and overwrite the old attackAl.

Meanwhile, back in instruction 8 we also loaded the t register with the gain for the A channel. So in instruction 10 we multiply the new attackAl by that gain, and in instruction 12 we can store the current stream's gain term.
5.6.4. Start decay?

Now is the time to check the sdecay term from the buffer pointer to see if the decay should start here. In instructions 13-15, we do an unsigned compare by performing a subtract. If wcAl > sdecayAl, then we branch to LA1G2 and go on our merry way. If wcAl <= sdecayAl, then we execute lines 16-20.

The first order of business is to start the decay where the attack was. 3 Under normal conditions, the attack filter's step response would have reached something close to 1.0. In screw cases, however, you'd be starting the decay before the fadein was completed. So in line5 16-18, the decay filter is initialized. If the attack filter has in fact reached nearly 1.0, then the step response of the decay filter will start at 0. If the attack filter is still in the middle of the fadein, then the decay filter will be initialized to 1.0-attackAl.

During initialization, the constant term decayG1 is calculated. But the calculation of the decay term will use decayG1Al. During the attack and the steady-state, that quantity is kept at 0. Setting decayG1Al to the non-zero value decayG1 in lines 19 and 20 means that on the next click, we will start processing with the decay generator, as we already saw in line 4.

5.6.5. Process decay

The decay is handled in lines 22-31. We reach LA1G1 from line 4 only if decayG1Al in AC is non-zero. So if we jump to line 22, decayG1Al is still in the AC.

Lines 22 through 25 duplicate what we saw for the attack generator. In other words, the decay generator's step response is taken. In lines 26 through 28, we subtract that step response from 1.0 to get the actual decay term. In lines 29 through 31, we multiply that decay term by the channel gain (loaded into the t register in line 24) to get the stream gain.

This completes the code for calculating a stream gain, whether we're dealing with a silence buffer, a fadein, or a fadeout

5.7. Calculate click count

The 58000 initializes a 32-bit number at locations dmem_scnthi and dmem_scntlo in DMEM to some number, typically 0, at the start of time. During initialization, those values are read in to the on-chip RAM locations scnthi and scntlo. The concatenation of those two numbers is incremented by 1 during each click, and the result is written out to DMEM so the 58000 can report exactly where we are at any given time.

This is another double-precision add; there was already one earlier, when memhi and memlo were incremented.

| 31h | scnthi | 1 |
| 32s | sentlo | 2 |
| 33s | one | 3 |
| 34h | scnthi | 4 |
| 35l | sentlo | 5 |

3 If I remember correctly, it was Bernard who pointed out the necessity of this. At any rate I don't get credit for his one.
In lines 1 and 2 we load the AC with the double-precision number. It is incremented by 1 in line 3, and the result is written out in lines 4 and 5. The result is also copied to DMEM in lines 6-9.

5.8. Check hang flag

Recall from the buffer pointer handler that it is possible to "hang" the code in an infinite loop every time that a new buffer pointer is about to be read in. It is also possible to "hang" the code once per click. {Again, this code belongs under the rubric of debugging.)

Here is the code:

```
lack dmem_hang ; 1
tblr hang 2
zals hang ; 3
bnz bisamplel  ; 4
```

In lines 1 and 2, we read from DMEM the value in dmem_hang. In lines 3 and 4, we loop back to 1 if (and as long as) the value in dmem_hang stays non-zero. So if you want the code to hang in this manner, set dmem_hang to something non-zero. Clear dmem_hang to zero when you are done.

It is possible to have this happen automatically. Right at the end of the code you will find:

```
lack dmem_hang
nop dmem_hang tblw minus 0x7D79
```

If you replace the non-op with tblw minus (which assembles to 0x7D79 in the current scheme of things), then dmem_hang will be written to be non-zero after every click is processed. In effect, the code will hang every click.

6. Assembling the building blocks

In the actual code, each of building blocks given in Section 5 is duplicated as often as necessary. To be exact, for each stream we have the following:

1. Read eight samples
2. Handle the buffer pointer and other DMA overhead; check the flags in DMEM
3. Calculate a gain term, including attack, decay, and "silence"

Furthermore, for each channel, we must do this:
4. Read in the gain term.

5. Calculate 8 samples

6. Output 8 samples to the D/A.

Finally, once per loop we must do the following:

7. Check the hang flag

8. Update the click count

9. Provide debugging output

6.1. Timing considerations

The situation is complicated because 1) it takes a certain amount of time to read a sample from MEM, and 2) we cannot output a new sample to the D/A until the old sample has been taken. That is, the code cannot "get ahead" of the D/A. As we have seen, MEM reads and D/A writes are handled with "buzz loop" i/o. The building blocks of code have been distributed exceedingly carefully so that as little time as possible is spent in those buzz loops.

Each sample time (at 48000 kHz) corresponds to 104 instruction cycles on the 32010, given its 200 nsec cycle time. We have decided that it is safe, on the average, to consume perhaps as many as 95 of them. There is some slop here; we can take more than 104 instructions to do one sample as long as the next sample or so takes correspondingly less time.

6.2. The Overall Plan

Table 1 shows how this works. In this section, we will discuss each column from the table.

Ignore, for now, the first column.

The second column lists the samples 1 through 8. As you know, we output 8 stereo samples to the D/A per click. The sample "boundaries" are used to divide the 900 or so lines of code. The number (9) at the top of the column means that each time we write out a stereo sample to the D/A, it consumes a maximum of 9 instruction times.

The third column shows when the samples are read. Samples 1 through 3 in stream A1 are read while the first of 8 samples is being output, as the first line in the table shows, and so on. The numbers in parentheses after each entry show how many instructions times are required. The numbers vary slightly due to considerations of optimization, which will be discussed later.

The fourth column shows when the eight stereo output samples are calculated. You see, the first of eight samples is calculated while sample no. 7 is being output, and the whole thing is pipelined, so that sample no. 8 is being calculated while sample no. 2 is being output. The calculation of sample 2 is split across the output times for sample 7 and sample 8. It takes 6 instructions to calculate each mono sample, or 12 for a stereo sample.

Column 5 shows how the buffer pointer ("wcma") handling is distributed. A maximum of 43 instructions is needed for this.
In column 6, we see where the attack, decay, stream gain terms, and "silence" buffer terms are handled. For a normal audio stream, 24 instruction times are required. A "silence" buffer takes only 3 instruction times, as the footnote says.

The 7th column shows some miscellaneous functions. It takes 7 instruction times to check the hang flag, which happens right at the top of the code. The two channel gains are read during the output of sample 2. And we update the click count during the time of sample 8.

The eighth and final column shows the total maximum number of instructions required for each sample time.

This, then, is how the code operates. After initialization (to be discussed below), the TMS enters a loop which goes from the top of the table to the bottom of the table, then starts over again. Using this table, you should be able to find any of the building blocks for any given sample, channel, or stream.

If you must modify the code, I encourage you to update your copy of Table 1 frequently. A roadmap like this is really necessary.

6.3. Labels in the code

Some labels in the code were already discussed in the opening paragraphs to Section 5, and in the individual subsections of Section 6.

Returning to the first column in Table 1, I have found it convenient to group samples 1-8 into "bisamples"; the bisample numbers are given in this column. Thus we see that we read all eight samples for stream A1; and handle the buffer pointer for the same stream, all within the first bisample. The bisample boundaries are explicitly labeled in the code as bisample1:, bisample2:, and so on.

In the code, the comment at the beginning of each block of code includes a number in parentheses. This shows how many instruction times (abbreviated iX) the block consumes.

6.4. Streams A2 and B2

These streams are currently not implemented, and have not yet been debugged. Following the original specification of the playback code, space was allotted for them in Table 1. In many cases, the code is already in place, commented out. Here is one example:

```plaintext
;B2    out   biozMEMdone, PAS
;B2    zais mellItimes
;B2 L616:    bioz  L617
;B2      subs one
;B2      bgz  L616
;B2      lack dmem_err_RDMA
;B2      tbl...  one
;B2      zais mem_times
;B2      b      L616
;B2 L617: in insampB21, PA2
;B2      out   twobits, PA3
```

The comment at the beginning of the line flags you that this is code intended for stream B2.
None of the code commented out in this fashion has been debugged.

6.5. Optimizing bioz

For doing DMA to MEM, and for writing samples to the D/A, the BIO flag must be set to "see" different flags. As we have seen above, this is done by issuing a

```
out biozdasi_new_samp, PA5
```

instruction for doing D/A i/o, or a

```
out biozMEMdone, PA5
```

for doing MEM accesses. Since the appropriate value is stored in the register PA5, it is not necessary to issue a new out instruction every time a new MEM read is started. Thus, you will find in the code that some of those out instructions have been commented out.

When you move around the code building blocks, you must very carefully check to ensure that you have not disrupted the order in which the values in PA5 are set.

6.6. Optimizing It and lta

By using the same sort of reasoning, we can optimize the use of the on-chip T register, which figures in multiplications. To review briefly, the steps for a multiply-accumulate are

```
zac1 zacl1 zac etc
lt
mpy
pac, apac, etc
sach, sacl, etc
```

initialize accumulator
load something into T
multiply T register by something
move or accumulate P register to AC
store output

Recall further that an lta instruction, instead of the lt given above, both 1) loads something into T and 2) does the effect of an apac, so that the P register is accumulated into the AC. The lta instruction is used in the code whereever possible. Sometimes the quantity loaded into T is not used for many many lines.

Let us examine a quick example. Here is the calculation of sample 6 for channel A, taken from sample time 1 (cf. Table 1):

```
CALCA6
   calculate A6 sample (6ix)
   lt insampA16
   mpy gainA1
   pac
   lt insampA26
   mpy gainA2
   lta insampA17
```
Again, when you move around the building blocks of code, you must carefully check the It and lta instructions to make sure that the correct value is always in the T register.

6.7. Pipelining considerations

The code building blocks cannot be moved around arbitrarily; there are some restrictions, and these are noted very carefully in the code. For example, consider the code excerpt just given in the previous section. It must occur before sample 6 for stream A1 (insampA16) is read in from MEM.

It is also instructive to examine the pipelining implied by Table 1. To output a given sample actually takes two complete passes through the code. In the first pass, we read in a full set of eight samples. To clarify the discussion, let's start with sample 1, which is read in during four different sample times: sample 1 for stream A1 is read in during sample time no. 1; sample 1 for stream A2 is read in during sample 3, and so on (see Table 1). During sample time no. 7, after sample 1 for all four streams has been read in, we can calculate the output samples for channels A and B. During the next pass through the code, (stereo) sample 1 is output during sample time no. 1.

Initialization will be discussed in Section 6.8. Here it should be at least pointed out that during initialization, all of the RAM locations for storing samples (8 samples per stream times four streams) are set to 0. This means that during the very very first pass through the code, you get 8 samples of silence out of the D/A, at no extra charge. The very first sample from the actual buffer out there in MEM is not calculated until sample time no. 7 and not output until sample time no. 1 in the second click. The alternative is to read the first four samples’ worth of data during initialization, but that was a headache. To be sure, some of the building blocks could be invoked via subroutine calls, but I preferred to avoid the extra overhead.

6.8. The Initialization Code

This completes the discussion of the main code loop. Now let us turn to the initialization that precedes this loop; there is a lot of code here too. For initialization, we will walk through the code from top to bottom, following the arrangement in the source file.

---

4Remember that the first 8 samples out of the D/A are going to be 0 anyway. This is because the attack is guaranteed to be reset on the first buffer, which means that the gain for each stream is 0 during the first click.
6.8.1. TMS RESET

Recall from the earlier discussion of the MEM memory map that DMEM[0:2] are reserved. After the TMS RESET signal is de-asserted, the TMS hardware branches to location DMEM[0] (on page 7 of the source) to start the code. By convention, a (two-location) branch instruction is placed there. We branch to the label spotinit, which is the actual start of the initialization code.

In DMEM[2] we have a ret (return) instruction for the unlikely event that a spurious hardware interrupt should occur. This has never been observed to happen.

There has been considerable discussion of the possibility of having more that one TMS program in DMEM. Since the playback code discussed here occupies only about 1100 DMEM locations, and DMEM is 4K words long, there would be ample room for two or more other programs in DMEM. After asserting the TMS RESET signal, the 68000 could clobber the branch address in DMEM[1] to point to any of the putative programs in DMEM. This scheme has not been tried out yet.

6.8.2. Initialize the chip

The TI manual is unfortunately incomplete in describing the startup state of the TMS chip. We found this out empirically ...

Right after the label "spotinit:" on p. 10 of the code, we execute

```assembly
ldpk 0; set RAM page pointer to 0
dint ; disable interrupts
larp 0; zero out auxiliary register pointer
rovm ; disable, for now, overflow mode of AC
```

In particular, they don't tell you in the manual that ldpk 0 is needed. These four instructions are scattered among the instructions that are discussed in the next two sections.

6.8.3. Initialize constants (page II)

A large number of constants are stored in the on-chip RAM to optimize the code in the click loop. The values for many of them (like zero, one, two, etc.) are meant to be obvious from their names.

The quantity in maxpos is the maximum positive number, OxFF. I chose to read it in from a DMEM location (whose contents should please not be changed) rather than calculating it by hand.

In this section we also initialize some variables with the constants that we have just defined. For example, setting the on-chip RAM copy of wc to 0 for all available streams will force a new buffer pointer to be read in during the code that follows. Note that some code has been included for streams A2 and B2 but is still commented out, as those streams have not yet been implemented.
The quantities dmem_rgarbage and dmem_wgarbage are dummy locations. When the very first buffer pointer is read in, then one of these dummy locations is written with a dummy "buffer done" flag. dmem_rgarbage is used for stream A1, and dmem_wgarbage is used for stream A2. Someone should figure out a better way to do this.

The quantity twobits contains 011. As we have seen, this initializes MEM addressing to auto-increment, and coincidentally starts a MEM read, the results of which are discarded.

Finally in this section, we initialize the values needed for the bioz control, and set up bit masks for the flags in memhi in the buffer pointer.

6.8.4. Initialize the D/As (page 11)

Sandwiched in the middle of this is a very important pair of lines:

; zero out D/A's
out zero,PA6
out zero,PA7 no need to buzz on bioz here

As soon as possible during initialization (viz., as soon as ldph is set and zero is defined) we send a zero out to the DASI's D/A registers. This guarantees that there will be no clicks or pops when the subsequent playback code starts up, because the playback code forces an attack to start at 0 for all streams.

It is still possible that some earlier playback code will have left a spurious value in PA6 or PA7, and therefore you might get a click when this initialization code sets those two registers to 0 (see Section 8). We have discussed and mapped out but not yet implemented a small secondary program which the 68000 could invoke after playback (especially an aborted playback) to stuff zeros into PA6 and PA7. Such a program would avoid all clicks when subsequent playback code is invoked.

6.8.5. Read in parameters (page 12)

Now we read in a few dozen parameters from DMEM into the on-chip memory. This is meant to be straightforward code, and since the function of each of these variables has been discussed earlier in this document, we will say nothing more here.

6.8.6. Set flags in DMEM (page 12)

During initialization, all flags whose names have the form _err_ must be initialized to 0. This is done next.

The flag dmern_err_wDMA is actually only used when the MEM initialization feature (to be discussed below) is invoked.

Currently, dmem_tmp is set to all ones. This serves to tell us that initialization has at least gotten this far, which is sometimes useful for debugging. Sometimes dmem_tmp is used for other purposes later in the code, mostly debugging, but they seem to have all disappeared in the
6.8.7. Initialize attack and decay gains (page 13)

The mathematics behind the calculation of the filter coefficients for attack and decay were discussed in Section 5.6.3, so this code will not be discussed in detail here. We should point out that the effect of

\[ \text{sach} \quad \quad \text{attackGl,0} \]

is to right-shift the quantity attackGl by 1 bit. The reasons for this are given in Section 5.6.3.

For debugging purposes, we write a copy of attackGl and friends to DMEM.

At the end of this block we invoke the sovm instruction to set overflow mode in the AC. Overflow mode remains set throughout the rest of the code, so that if we overflow by adding two channels of audio, the result will be clipping rather than a huge pop.

6.8.8. Read in first buffer pointer (page 13)

Here we will discuss the code for stream Al, and handle only m passing any differences for stream Bl.

The first step is to get the very first buffer point start address, which lives in dmem_first_wcmaAl:

```assembly
iswcAll: lack dmem._first_wcmaAl ; 1
    tblr nextWcmaAl ; 2
    zals nextWcmaAl ; 3
    bnz iswcA12 ; 4
    lack dmem._err_no_wcmaAl ; 5
    tblw minus ; 6
    b iswcAll ; 7

iswcA12: ...
```

In line 3, we test to ensure that the start address for the first buffer pointer is non-zero. Remember that the source code initializes that location in DMEM to 0, and the 68000 must overwrite that location with something else. For a start address = 0, we loop here (from line 7 back to line 1) until the start address turns non-zero. This is of course an error condition, so we set dmem.err_no_wcmaAl to all ones, which indicates that a zero address for the first buffer pointer was found.

The next two lines in the code:

```assembly
iswcA12: zals bufdoneptrAl
    tblw minus
```
match the opening gambit in the normal buffer pointer code (see Section 5.4.1), except that when we load bufdoneptrAl into the AC, what we're really loading is dmem_rgarbage. Remember that this was taken care of during initialization. It is thus dmem_garbage that is overwritten with all ones at this point. My goal here was to make the initialization code match the code in the main click loop as closely as possible, so that you could drop a copy of the main loop buffer pointer handler into this spot in the initialization. Now dmem_rgarbage being set to all ones means, for debugging purposes, that the initialization has gotten at least this far. Still, at some point, the two lines of code just given should be removed. (Note that dmem_wgarbage is used for stream Bl).

The buffer pointer initialization continues by reading in the first buffer pointer. Note that since attackAl was set to 0 earlier in the initialization, you are guaranteed to get an attack (fadein) on the very first buffer.

6.8.9. Initialize MEM (debugging) (page 14)

The next hunk of code, controlled by the compile-time switch INITMEM, is of interest only to those who will have to do heavy-duty debugging.

The basic idea is to fill some hunk of MEM with known values. The first variant is to fill MEM with ascending numbers, for testing the DMA code. The second variant is to fill MEM with maxpos (=Ox7FFF), which is useful for debugging the attack and decay one-pole filters. This code uses the dummy buffer pointer at PLAY2_sample_wcmal, which is discussed in more detail in Section 6.9.5. Suffice it to say for now that this dummy buffer pointer is included for free as part of the source code but is normally not used by the 68000. The dummy buffer pointer points at the 256 locations MEM[Ox2000:0x21FE].

To start off, as you'll recall from the discussion in Section 5.2, it will be necessary to set up the bioz flag correctly for DMA:

```c
out biozMEMdone, PA5
```

Furthermore, we use the two auxiliary registers as counters:

```c
lark ARO, 255
lark ARl, 0
```

AR0 will be a loop counter soon, and ARl will give us the numbers that step up by one.

Getting back to the buffer pointers, what we really want is to stuff the start address from the buffer pointer into PA0 and PA1 if the buffer pointer starts at PLAY2_sample_wcmal, then memhi for that buffer pointer is at the next location in MEM. We read in that location with this code and immediately write out memhi to PA1:

```c
lark PLAY2_sample_wcmal
adds one
>tblr tmp
out tmp, PA1
```
For debugging purposes, we write out an *extra* copy of `memhi` with the next two lines:

```plaintext
lack    PLAY2_sample_PA1
tblw    tmp
```

Then we get `memlo` (which is two locations after `PLAY2_sample_wcmal`) with this code:

```plaintext
lack    PLAY2_sample_wcmal
adds    two
tblr    tmp
out     tmp, PA0
lack    PLAY2_sample_PA0
tblw    tmp
```

and write out a debugging copy of `memlo` into `PLAY2_sample_PA0`.

Now we can enter a little loop that writes out the values to MEM:

```plaintext
DO1:
4tifdef   FILL  MEM_WITH_.ONES
  out     maxpos, PA2
4telse
  larp    ARl
  mar     *+
  sar     ARl, tmp
  out     tmp, PA2
4tendif
  zals    mem_times
D03:
  biez    DOS
  subs    one
  bgz     D03
  lack    dmem_err_wDMA
  tblw    minus
  zals    mem_times
  b       D03
DOS:
  larp    AR0
  banz    DO1
```

As we saw before, the loop is controlled by the counter in AR0; and the `banz DO1` in line 18 handles the loop. We have to explicitly select AR0 in line 17 because AR1 is used earlier in line 5.

Now if you want MEM filled with all ones, then line 2 is executed. Actually, writing out `maxpos` to PA2 in line 2 starts the write cycle.

If you want MEM filled with numbers that step up by 1, then lines 4 through 7 are what you need. In line 4 we select AR1, in line 5 we increment it by 1, and in lines 6 and 7 we write it out to MEM.

Lines 9 — 16 are the same sort of BIO loop that was encountered in the discussion of the code block for reading in samples. The only difference is that now we use `dmem_err_wDMA`, indicating that MEM is hung during a `write`.

As a further part of MEM initialization, we next set up a region guaranteed to contain only zeros. This is a dummy "silence" buffer also useful for debugging. This code operates like the code just discussed. The only differences are: 1) we work with the buffer pointer that starts at
PLAY2_zero_wcmal; 2) the buffer is only 16 samples long; 3) we copy memhi and memlo to PLAY2_zero_PA1 and PLAY2_zero_PA0, respectively.

6.8.10. Initialize internal sample storage (page 16)

Now we zero out the internal RAM locations that will eventually hold the input samples read in by DMA from MEM. This is actually quite simple:

```
lark ARO, 23  ; 1
lark ARI, insampAll ; 2
zac 3
zeroinO:larp ARI 4
sacl *,0,ARO 5
banz zeroinO 6
```

AR0 is the loop counter for the 24 locations. Normally, we would expect 32 locations: 8 samples per stream times four streams. However, as I mentioned in Section 4.2.2.7, I ran out of on-chip RAM locations; so I sacrificed the 8 locations for stream B2, since that stream is not implemented yet. So we're really zeroing out 8 samples per stream X 3 streams = 24 locations, and to make the banz work right, we load one less than that (=Q3) into ARO. ARI will be a register pointing to the location in RAM to be zeroed. We zero the AC in the third line, then start the loop. In line 4 we explicitly select ARI. In line 5 we write out to the location that ARI points to, then increment ARI, then select AR0 (the construct ",,0," in that line is merely a syntactic place holder). The banz in line 6 completes the loop.

In the source code, there follows a similar loop for zeroing out the 16 locations where the output samples are stored.

6.8.11. Set up debugging (page 16)

Now we have some simple blocks of initialization for debugging purposes. First we clean out the DMEM copies of the input samples for streams A1 and B1.

Then we write out to DMEM-some copies of various quantities in the on-chip RAM, especially the gain terms for the attack and decay filters.

6.8.12. Finish up initialization stage (page 17)

The next-to-last step in the initialization is to preload the T register with insampA15. In a way, this is a semantically meaningless step, as we know that insampA15 is zero (because we just initialized it to that value). Still, I found it important to keep this one line in the code to remind myself (cf. Table 1) that insampA15 really must be in the T register because sample 5 is calculated during the first part of the click loop.
6.8.13. The init_done flag

The very last part of initialization is to set the init_done flag in DMEM to be minus. I didn't mention it before, but that flag was explicitly set to 0 right at the beginning of initialization.

It was Bernard who suggested the dmem_init_done flag. After TMS RESET is de-asserted, the 68000 can time out after some interval if dmem_init_done isn't reset to 1; we assume that the TMS is hung, and print out an error message.

6.9. I/O for Debugging

I can't recommend this entire major section for anyone unless you really intend to debug this code yourself. We're talking nit and grit here.

6.9.1. The hang flag

The hang flag has already been discussed in Section 5.8. Let me mention again that you can set by hand the location dmem_hang. At the beginning of the next pass through the click loop, the code will hang in a small loop until you, by hand, set dmem_hang to 0 again.

This takes the place of the ability to set break points and single-step instructions, which is not supported by the TMS 32010. This was one of the features that I thought of as I was driving in to TDW after JAM woke me up at 5:30 the Friday a.m. before NAB.

6.9.2. Copies of buffer pointers

As we saw earlier in Section 4.2.1.6, there is a place for copies of the buffer pointers for streams A1 and B1. In the slot for sample 5 (page 31), we leisurely fill in those locations in DMEM. Note that this function is not shown in Table 1. Also, if you ever do implement streams A2 and B2, then some other place will have to be found for this debugging code, as it occupies instruction times that are reserved for the normal stream A2 functions.

Since the buffer pointers are copied out once per click, they contain the updated values of memhi, memlo, and we.

Actually, the copy in DMEM is slightly more informative that the original buffer pointer m DMEM. Here is the format for the copy:

Ver. 1.0 C 1986 TDW 35
6.9.3. Copies of output samples

In the time allotted for outputting sample 6 (in Table I), you will find code that copies the current samples going to the B output channel. Also, the current attack, decay, and gain terms for the streams AI and BI are written out to DMEM here.

6.9.4. Copies of input samples

In the time allotted for outputting sample 8, you will find a long string of tblw instructions for copying out the current 8 input samples for streams AI and BI to DMEM.

6.9.5. Dummy buffer pointers

On p. 9 of the code you will find three buffer pointers to very short buffers in MEM, and on p. 10 there are two pointers to dummy "silence" buffers in MEM. These regions of MEM can be initialized to known values using the code discussed in Section 6.8.9.

The first buffer pointer starts at location PLAY2_sample_wcmal. It lasts for three clicks, and points to a second buffer pointer at location PLAY2_sample_wcmaw2. The second buffer pointer lasts for one click and points to a third buffer pointer at location PLAY2_sample_wcma3. This third buffer pointer lasts for three clicks, and points back to the first buffer pointer. In the third buffer pointer, the decay is started. The buffers pointed to by these three buffer pointers are contiguous. That is, the last sample from the first buffer is followed in MEM by the first sample from the second buffer.

The first pointer to a silence buffer lasts two clicks (which is, as you'll recall, the minimum acceptable length for a silence buffer). The second pointer to a silence buffer lasts forever, but points back to the first buffer for a silence buffer, just in case. Note that both buffer pointers have zero bit set.

To use the dummy buffer pointer, set dmem_first_wcmaAI to be PLAY2_sample_wcmal, which is set to 52 in dmem.h. To use the dummy pointer to the silence buffer, set dmem_first_wcmaAI to PLAY2_zero_wcmaAI, which is 70.

Whenever I change the buffer pointer code of section 5.4, I cause the hang flag to be set (see Sections 5.4.2.2 and 6.9.1) and step through the clicks carefully, reading out the updated buffer pointers in the debugging copies.
6.9.6. What if you get the Sound of Silence?

When you are debugging, and nothing seems to be happening, then check dmem_init_done. If it is zero, you know you never reached the end of the initialization stage. You should therefore check dmem_err_wcAl_zero, dmem_err_wcBl_zero, dmem_err_no_wcmaAl, dmem_err_no_wcmaBl, and dmem_err_wDMA (if you are doing INITMEM); if any of them is non-zero, then go find the problem. Also, recall that if dmem_tmp is 0 (and dmem_init_done is 0), then you've not reached the stage in the initialization, mentioned before, where dmem_tmp is set to all ones.

Then check the copy of the buffer pointer. If the click loop is flying by, we should be constantly changing. The same is true of memhi and memlo.

Make sure that aGain and bGain are not 0.

Examine the output samples as copied into DMEM. If they are 0, and the gain terms are non-zero, then check the input samples as copied into DMEM.

If those are 0, then assert the hang flag. From the copy of the buffer pointer in DMEM, find the MEM addresses. Go see what's in MEM.

7. Using the assembler's output

For each and every variable in DMEM, I cannot encourage you strongly enough to include, in the declarations section starting on page 7 of the source, a statement of the form:

```c
>dmem_decayGl 0
>dmem_decayG2 0
```

In the first line, the ">=" construct sets the location counter to the value to the right of the equals sign. The next line fills the corresponding location in DMEM with an initialization value. Thus, in the source code we have initialized dmem_decayGl to have the value 0. For tables, you should do something like this:

```c
>dmem_inputAl 0 0 0 0 0 0 0 0
```

Here we have eight locations that will contain copies of the eight samples read in from MEM, for stream Al.

Not only are you thereby guaranteed to know the initial value of every location in DMEM. Remember that as the source code is loaded into DMEM (which is, after all, program memory), these initialization values will be explicitly set by the loader.
But furthermore, in the .list file produced by the assembler, you get a three-column output that looks like this:

```
  .=dmem_decayGl
  4000ee 0077 0000 0
dmem_decayG2
  4000f0 0078 0000 0
dmem_inputA1
  40010a 0085 0000 0
dmem_abortcnt, dmem_da_loop_times, dmem_mem_times, dmem_decayincr,
dmem_attackincr, dmem_first_wcmaA1, dmem_first_wcmaBl, dmem_scnthi,
dmem_scntlo, dmem_aGain, dmem_bGain, dmemHang (set this to 0!), dmem_wcmaHangBl (ditto!),
dmem_wcmaHangBl (ditto!).
```

The first column contains the D11EM address, in DDT format. The second column gives the address running (in hex) from 0 through 4K. The third column gives the value initially loaded into DMEM. The point of mentioning all this is that if you take the trouble to put a ".=\n" line into the declaration section of the source code, you automatically get, in the code listing, the DDT style address for that DMEM variable. Believe me, this will save you immense amounts of time when you are debugging.

The same listing will quickly tell you where to overwrite nops if you want to munge the code for debugging. For example, at the end of the listing for the playback code you will find:

```
  400a90 0548 7e17  lack dmem_hang
  400a92 0549 7f80  nop
```

When using DDT, 400a92 is the location at which you overwrite the default nop with the tblw minus instruction (currently assembled as 0x7D79) if you want to invoke the hang feature after every run through the click loop.

8. Starting and stopping the playback code

Before starting up the playback code, the following locations in DMEM should be initialized by the 68000: dmem_abortcnt, dmem_da_loop_times, dmem_mem_times, dmem_decayincr, dmem_attackincr, dmem_first_wcmaA1, dmem_first_wcmaBl, dmem_scnthi, dmem_scntlo, dmem_aGain, dmem_bGain, dmemHang (set this to 0!), dmem_wcmaHangBl (ditto!), dmem_wcmaHangBl (ditto!).

The following locations will be written out later by the 32010: dmem_init_done, dmem_err_da, dmem_err_rDMA, dmem_err_wDMA, dmem_err_no_wcmaA1, dmem_err_no_wcmaBl, dmem_err_wcA1_zero, dmem_err_wcBl_zero, dmem_err_wcA1_seen, dmem_err_wcBl_seen, dmem_rgarbage, dmem_wgarbage, dmem_tmp, dmem_err_wcA1_copy et seq., dmem_err_wcBl_copy et seq..

The following location should not be changed from the default value given in the source code: dmem_maxpos.

If the initialization of DMEM is finished, then the 68000 may de-assert the TMS' RESET signal.
As we have seen, this starts execution at DMEM[0], which is a two-location branch instruction.

To stop the TMS, there are two alternatives. One is that the 68000 asserts the TMS' RESET signal again. This will stop everything dead. The most recent values will be left hanging in the DASI's registers, which means that the D/As will output DC, which is equivalent to silence. Note, however, that the next playback may have a pop at the beginning, since we always reset those registers to 0 for the first buffer of each stream (see Section 6.8.4).

The other method of stopping the TMS is to corral all streams into reading from a "silence" buffer, possibly ramping down the last cue for a stream and then moving on to a silence buffer. Each stream should loop reading silence buffers until all streams have reached such a state. The 68000 can test the buffer done flags in the buffer pointer. When all streams have started reading silence buffers, then it is safe to assert the TMS' reset signal, as we know that the DASI's D/A register contains 0. This is the method that should normally be used.

9. Table 1.

<table>
<thead>
<tr>
<th>Bi-Sample No.</th>
<th>Output Sample No. (9)</th>
<th>Read Sample No.</th>
<th>Calculate Sample (12)</th>
<th>Check Wema or Incr Mem (43)</th>
<th>Calculate Attack/ Decay/Gain Term (24*)</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A11-A13 (22)</td>
<td>5</td>
<td>A1</td>
<td>Check hang (7)</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A15-A18 (44)</td>
<td>6-8</td>
<td>B1</td>
<td>Read gains (8)</td>
<td>97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B11-B14 (20)</td>
<td></td>
<td>B1</td>
<td>A1</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B15-B18 (44)</td>
<td></td>
<td>B1</td>
<td>77</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A21-A24 (20)</td>
<td></td>
<td>A2</td>
<td>A2</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A25-A28 (44)</td>
<td></td>
<td>B2</td>
<td>77</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B21-B24 (20)</td>
<td>1, 2A</td>
<td>B2</td>
<td>90</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B25-B28 (44)</td>
<td>2B-4</td>
<td></td>
<td>Click Cnt (13)</td>
<td>96</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Only 3 ix if the stream is a "silence" buffer.

10. Appendix A. The playback code (listing).
#include "dmem.h"
define INITMEM
#define DABIOZ bioz
/* DABIOZ covers only bioz's for D/A */
/* legal values of DABIOZ are "b" or "bioz" */
/* "b" is for debugging standalone only */
define FILL_MEM_WITH_ONES 1 /* use this for debugging */
/* attack, decay. Turn it off to get MEM filled */
/* with 1, 2, 3, ... for debugging readin code */

; to MAS:
; you must initialize
; PLAY2_bGain

; ut/u0/john/od/spot2.a

; playback code for soundDroid spotter, commercial release
; version. Reads in one audio stream for both A and B
; sides (of dasi). Does fadein or out as necessary, incorporates gain, to
; produce one stereo sample.
; Code in place for reading in other streams,
; but commented out.

; john 7/24/86 --- 8/6/86

; if the reads from MEM need to be spread out more, you *could* drop
; one in the middle of the do-wcma code, and duplicate it in the
; don't-wcma code
; plans for "zero" buffer:
; 1. mem reads, branching out of the middle of nwcmaA1
; 2. maybe confine buffers to certain areas of MEM, to avoid reading in
; MEMHI?
; create a DMEM location for flags. flags read in once per cycle.
; bits 0-4 are abort bits for each of four streams. test abort
; bits in non-read-wcma part of wcma check. attack, zero still in memhi
; collapse all dmem_err_wcA1_zero flags to one, since writing out addr?

; DMEM locations (referred to in TI manual as pma, program memory address).
; Locations [0:7] reserved so that tblr/w won't interfere with in/out.
; MAS's suggestions 7/24/86:
; 1) operate asynchronously, with flags as to when to readin wcma
; 2) add another wcma state: already read in, but not finished;
; to allow him more time

; initialization parameters
dmem_abortcnt = PLAY2_abortcut ; When you set abortbit in a wcma, wc for
; that stream will be set to this number.
; remove this eventually *********
dmem_da_loop_times = PLAY2_da_loop_times
; Can be initialized to some constant by 68000;
; left alone thereafter. We copy this once
; into 32010 parameter memory. Default is
; set in code below. Controls how often to loop before
; setting dmem_da_lus
dmem_mem_times = PLAY2_mem_times
; How often to loop when reading MEM before complaining.

mem_decayincr = PLAY2_decayincr

mem_attackincr = PLAY2_attackincr ; calculated by sattack on /ut/john/odd.
dmem_maxpos = PLAY2_maxpos; please make sure this contains 0x7FFF
dmem_first_wcmaA1 = PLAY2_first_wcmaA1

; Pointer to very first wcma for stream 1, A sample. See
; mem1A1 for required format. We restart attack for you.
; Default set to 0 in code below.
; If 68000 forgets to turn it on, dmem_no_wcmaA1 will be
; set, and 32010 will just spin its wheels.
dmem_first_wcmaB1 = PLAY2_first_wcmaB1

; run-time inputs to 32010 and outputs from 32010

dmem_scnthi = PLAY2_scnthi; Public copy of *click* counter, initialized

dmem_scntlo = PLAY2_scntlo; by 68000.
dmem_init_done = PLAY2_init_done; 0 while initiating, all 1s when init done.
dmem_aGain = PLAY2_aGain; Gain for sample A. Read on each click.
dmem_bGain = PLAY2_bGain; Ditto, sample B.
dmem_hang = PLAY2_hang; keep this at 0. 32010 will read this once per click.
; Whenever this is nonzero, we continue putting out
; the same sample to the D/A's. We initialize this to 0
; after dsp reset.

; error flags

dmem_err_da = PLAY2_err_da; Initialized to 0 by 32010.
; Asserted to 1 when spin longer than dmem_da_loop_times
; waiting for sample to be eaten by DASI.
; Must be cleared to 0 by 68000.
dmem_err_rDMA = PLAY2_err_rDMA
; Initialized to 0 by 32010. Set to 1s by 32010 when
; mem has hung after dmem_mem_times loops attempting to read.
; Must be cleared to 0 by 68000.
dmem_err_wDMA = PLAY2_err_wDMA; Reserved for A/D code, used here for debugging
; dmem_err_no_wcmaA1 = PLAY2_err_no_wcmaA1; set to 0 by 32010. Set to all 1s if
; dmem_first_wcmaA1 is 0. In this case, 32010 spins
; its wheels until dmem_first_wcmaA1 goes nonzero.
dmem_err_no_wcmaB1 = PLAY2_err_no_wcmaB1
; dmem_err_no_wcmaA2 = PLAY2_err_no_wcmaA2
; dmem_err_no_wcmaB2 = PLAY2_err_no_wcmaB2

dmem_err_wcA1_zero = PLAY2_err_wcA1_zero
; this means that wc in some wcma was zero
dmem_err_wcB1_zero = PLAY2_err_wcB1_zero
dmem_err_wcA1_seen = PLAY2_err_wcA1_seen
dmem_err_wcB1_seen = PLAY2_err_wcB1_seen

; miscellaneous

dmem_rgarbage = PLAY2_rgarbage; 32010 sometimes writes garbage here. Ignore me

dmem_wgarbage = PLAY2_wgarbage ; ditto
dmem_tmp = PLAY2_tmp; ditto

; wcmas
52 thru 82 have dummy default wcmas; can be consolidated.
88 thru 99 are reserved for the following:
dmem_err_wcA1_copy = PLAY2_err_wcA1_copy
; Al wcma copied to dmem on every click, for debugging

; debugging info
dmem_attackG1 = PLAY2_attackG1
; dmem's copies of these coefficients, for debugging only
dmem_attackG2 = PLAY2_attackG2
dmem_decayG1 = PLAY2_decayG1
dmem_decayG2 = PLAY2_decayG2
dmem_output1 = PLAY2_output1 ; B side only
dmem_output2 = PLAY2_output2
dmem_output3 = PLAY2_output3
dmem_output4 = PLAY2_output4
dmem_output5 = PLAY2_output5
dmem_output6 = PLAY2_output6
dmem_output7 = PLAY2_output7
dmem_output8 = PLAY2_output8
dmem_inputA1 = PLAY2_inputA1 ; A side only
dmem_inputA2 = PLAY2_inputA2
dmem_inputA3 = PLAY2_inputA3
dmem_inputA4 = PLAY2_inputA4
dmem_inputA5 = PLAY2_inputA5
dmem_inputA6 = PLAY2_inputA6
dmem_inputA7 = PLAY2_inputA7
dmem_inputA8 = PLAY2_inputA8
dmem_inputB1 = PLAY2_inputB1 ; B side only
dmem_inputB2 = PLAY2_inputB2
dmem_inputB3 = PLAY2_inputB3
dmem_inputB4 = PLAY2_inputB4
dmem_inputB5 = PLAY2_inputB5
dmem_inputB6 = PLAY2_inputB6
dmem_inputB7 = PLAY2_inputB7
dmem_inputB8 = PLAY2_inputB8
dmem_attackA1 = PLAY2_attackA1
dmem_decayA1 = PLAY2_decayA1
dmem_gainA1 = PLAY2_gainA1
dmem_attackB1 = PLAY2_attackB1
dmem_decayB1 = PLAY2_decayB1
dmem_gainB1 = PLAY2_gainB1
dmem_attackbit = PLAY2_attackbit ; will have a 1 in the high-order bit
dmem_zeroBit = PLAY2_zeroBit
dmem_abortBit = PLAY2_abortBit

dmem_notAbortBits = PLAY2_notAbortBits ; all 1s except for abortbits
dmem_decayA1 = PLAY2_decayA1

dmem_decayB1 = PLAY2_decayB1
dmem_wcmaHangA1 = PLAY2_wcmaHangA1
dmem_wcmaHangB1 = PLAY2_wcmaHangB1
dmem_wcma_cntA1 = PLAY2_wcma_cntA1
dmem_wcma_cntB1 = PLAY2_wcma_cntB1
; 160 to 255 reserved for mas to munge wcmas

; dma (data memory address) locations (on-chip)

 error and control flags
_a_loop_times = 0 ; How many times to loop waiting for biozdasi_new_samp
; before setting dmem_err_da.
iozdasi_new_samp=1; Contains 02, for bioz flag.
_em_times=2; How many times to loop waiting for biozMEMdone before
; setting dmem_err_rDMA.
biozMEMdone=3; Contains 0.
; biozaudio=XXX; Contains 04, for bioz flag.
hang=5; current contents of dmem_hand. Read once per clicque.

; parameters
attackincr=6; The constant, standard value for attack increment.
decayincr=7; The standard value for decay increment.
attackGl=8; Gain terms for one-pole creating attack; filled
during initialization. The attack
attackG2=9; filter is attackGl + attackG2 * attackAl.
decayGl=10
decayG2=11
scnthi=12; *Click* counter, pronounced "scount" or "scent";
scntlo=13; initialized to whatever by 68000.
decayGlAl=14
decayGlB1=15
; decayGlA2=16
; decayGlB2=17

; here is the wcma for stream 1 of sample A. The term wcma is derived from
; PDP-10 usage, in which a 36-bit word had word count in the left half and
; memory address in the right. If you change the order here, you must change
; the code at readwcAl and friends
wcAl=18; No. of clicks left to read in current buffer.
; Decrement on each click. FEATURE: the 32010
; will behave unpredictably if you specify wc of 0.
; Therefore, we trap wc=0, loop, and set a flag.
memhiAl=19
; Well, this is really:
; MEM hi memory address (for PA1) in bits 0:6.
; bit 15: (zerobit) 1 means this buffer has only 0s.
; Restriction: such a buffer must be wc>=2 long.
; bit 14: (attackbit) restart attack.
; bit 13: (abortbit) start decay, set wc to ****
; bits 12-7 ignored here
; bits 0-6 MEM hi address (for PA1)
memloAl=20; MEM lo memory address (for PAO).
nextWcmaAl=21; bits 0-7, points to next wcma.
sdecayAl=22; Click no. where to start downramping decay. When
; wcAl is decremented *past* sdecayAl, decay kicks in.
; Actually, please tell us click no. + 1.

; Here are the rest of the parameters for stream A.
bufdoneptrAl=23;
flaglocAl=24;
zeroA1=25;
flagA1 = 26;
; Attack and decay terms are calculated once per click:
decayAl=27; current value of decay generator. Decay goes from 0x7FFF
; to 0. Right when decay reaches 0, wc should also go to 0.
attackAl=28; current value of attack generator. Attack goes from 0 to
; something close to 037777 and sticks there.
Now we duplicate the above for the other three streams.

; Here is B1
wcBl = 30
memhiBl = 31
memloBl = 32
nextWcmaBl = 33
sdecayBl = 34
bufdoneptrBl = 35
flaglocBl = 36
zeroBl = 37
flagBl = 38
decayBl = 39
attackBl = 40
gainBl = 41
; Here is A2.
wcA2 = 27
memhiA2 = 28
memloA2 = 29
sdecayA2 = 31
bufdoneptrA2 = 32
nextwcmaA2 = 33
bufdoneptrA2 = 33
decayA2 = 34
attackA2 = 35
gainA2 = 36
; Here is B2
wcB2 = 48
memhiB2 = 49
memloB2 = 50
sdecayB2 = 51
bufdoneptrB2 = 52
nextwcmaB2 = 53
decayB2 = 55
attackB2 = 56
gainB2 = 57

; during each eight-sample click, we read in eight samples (in
; two groups of four each) for
; each of four streams. The samples are stored here until they
; can be used.

insampAll = 59 ; storage for eight samples for stream A1
insampA12 = 60
insampA13 = 61
insampA14 = 63
insampA15 = 64
insampA16 = 65
insampA17 = 66
insampA18 = 67
insampA21 = 68 ; storage for eight samples for stream A2
insampA22 = 69
insampA23 = 70
insampA24 = 71
'insampA25 = 72
' sampA26 = 73
insampA27=74
   insampA28=75
   insampB11=76 ; storage for eight samples for stream B1
   insampB12=77
   insampB13=78
   insampB14=79
   insampB15=80
   insampB16=81
   insampB17=82
   insampB18=83
   insampB21=84 ; storage for eight samples for stream B2
   insampB22=85
   insampB23=86
   insampB24=87
   insampB25=88
   insampB26=89
   insampB27=90
   insampB28=91
   wcma_cntA1=84
   wcma_cntB1=85

; locations for combining all four streams into two samples
aGain = 92 ; gains for A, B channels, as read in from DMEM.
bGain = 93 ; Gains range from 0 to 0x7FFF
outsampA1=94 ; (insampA1n*gainA1 + insampA2n*gainA2), ready to ship to D/A
outsampA2=95
outsampA2=96
outsampB2 =97
outsampA3=98
outsampB3 =99
outsampA4=100
outsampB4 =101
outsampA5=102
outsampB5 =103
outsampA6=104
outsampB6 =105
outsampA7=106
outsampB7 =107
outsampA8=108
outsampB8 =109

; miscellaneous constants
tmp=110
maxpos=111 ; will hold maximum positive number, it says here
zero =112 ; that's right, 0
one=113 ; 01
two=114
three=115
four =116
five =117
six=118
seven=119 ; contains 7
eight=120 ; contains 8
minus=121 ; all one's
vobits=122 ; 11b
sportcnt=123 ; replace this with an lack eventually *****
attackbit=124 ; will have a 1 in the high-order bit
verobit=125
_abortbit=126
notabortbits=127 ; all 1's except for abortbit
; remember that the maximum useable is 127 unless you change ldpk

.b =0
.b spotinit
ret

; here are inputs that MAS must set
.b dmem_first_wcmaAl
PLAY2_zero_wcmaAl ; ******** debugging only: 0 wcma
PLAY2_sample_wcmaAl ; ******** has fake samples
0 0

; here are some inputs that MAS should set or leave alone
.b dmem_abortcnt ; *********** debugging value only 1414
.b dmem_da_loop_times
1000
.b dmem_mem_times
1000
.b dmem_aGain
07FFh ; largest possible positive number
.b dmem_bGain
07FFh
.b dmem_decayincr
5fff; 8 samples debugging only 145dh ; attack, decay last 64 clicks
.b dmem_attackincr
5fff; 8 samples 145dh

; MAS will read these on occasion
.b dmem_init_done
0
.b dmem_scnthi
0
.b dmem_scntlo
0

; here we group the error flags that MAS should read often
.b dmem_err_da
0
.b dmem_err_no_wcmaAl
0 0 ; B1
0 0 ; A2
0 0 ; B2
.b dmem_err_rDMA
0
.b dmem_err_wDMA ; debugging only
0
.b dmem_err_wcAl_zero
0 0
.="dmem_err_wca1_seen"
0

; the following must not be changed by anyone ever
.="dmem_maxpos"
0?FFPh ; is maximum positive number

; the following are misc. debugging flags just for john
.="dmem_hang"
0
.="dmem_wcmahangA1"
0
.="dmem_wcmahangB1"
0
.="dmem_rgarbage"
0
.="dmem_wgarbage"
0
.="dmem_attackG1"
0
.="dmem_attackG2"
0
.="dmem_decayG1"
0
.="dmem_decayG2"
0
.="dmem_inputA1"
0
0
0
0
0
0
0
0
0
0
0
0
.="dmem_inputB1"
0
0
0
0
0
0
0
0
0
0
0
0
.="dmem_output1"
0
0
0
0
0
0
0
0
0
0
0
.="dmem_attackA1"
0
Here is a dummy set of circularly linked wcmas. It reads 48 words starting at MEM[2000] (as typed to DDT). Attacks occur on first wcma. Note that these addresses are picked up by INITMEM block during initialization.

Here are the wcma addresses:

1. PLAY2_sample_wcma1
   - Flags: memhi: 8000 zero, 4000 attack, 2000 abort
   - Memlo: Achtung that address is embedded later
   - Bufdone: for initing read buf to all 1's

2. PLAY2_sample_wcma2
   - Flags: memhi: 8000 zero, 4000 attack, 2000 abort

3. PLAY2_sample_wcma3
   - Flags: memhi: 8000 zero, 4000 attack, 2000 abort

These addresses are used to initialize the wcma structures.
4128 ; memlo
PLAY2_sample_wcmal ; next ; TMS will hang when go to .=sampl
 1 ; sdecay
 0 ; bufdone

; these two wcmas point to 0 buffers.
.=PLAY2_zero_wcmal
 2 ; wc
 8000h ; zero buffer
 4352 ; memlo
PLAY2_zero_wcmal2 ; next
 0 ; sdecay
 0 ; bufdone
.=PLAY2_zero_wcmal2
0FFFFh ; wc
8000h ; zero buffer
4352 ; memlo
PLAY2_zero_wcmal ; next ; TMS will hang when go to .=zero_
 0 ; sdecay
 0 ; bufdone

.=dmem_err_wcAl_copy
 88 ; wc
 0 ; memhi
 0 ; memlo
 0 ; sdecay
 0 ; bufdoneptr
 0 ; nextwcma
 0 ; flags
 0 ; zero buf?
.=dmem_wcma_cntAl
0
.=dmem_err_wcBl_copy
 0 ; wc
 0 ; memhi
 0 ; memlo
 0 ; sdecay
 0 ; bufdoneptr
 0 ; nextwcma
 0 ; flags
 0 ; zero buf?
.=dmem_wcma_cntBl
0

.=256
spotinit:
; standard startup:
ldpk 0 ; spot code can be separated into
dint ; init and execute halves if needed.
; then the 68000 can clobber DMEM[1]
; and do either part separately.
initialize some constants
zac
sacl zero,0
lack  

dmem_init_done  
tblw  

zero  

; zero out D/A's  
out  

zero, PA6  
out  

zero, PA7  

; no need to buzz on bioz here  
; continue standard startup and ...  
larp  

0  

rovm  

; continue initializing constants  
lack  

1  

sacl  

one, 0  

lack  

2  

sacl  

two, 0  

lack  

3  

sacl  

three, 0  

lack  

4  

sacl  

four, 0  

lack  

5  

sacl  

five, 0  

lack  

6  

sacl  

six, 0  

lack  

7  

sacl  

seven, 0  

lack  

8  

sacl  

eight, 0  

lack  

dmem_maxpos  
tblr  

maxpos  
zac  

biozMEMdone, 0  

sacl  

flagA1, 0  

sacl  

zeroA1, 0  

sacl  

attackA1, 0  

sacl  

decayA1, 0  

sacl  

flagB1, 0  

sacl  

zeroB1, 0  

sacl  

attackB1, 0  

sacl  

decayB1, 0  

; sacl  

attackA2, 0  

; sacl  

decayA2, 0  

; sacl  

attackB2, 0  

; sacl  

decayB2, 0  

sacl  

gainA1, 0  

sacl  

gainA2, 0  

sacl  

gainB1, 0  

sacl  

gainB2, 0  

sacl  

decayG1A1, 0  

sacl  

decayG1B1, 0  

; sacl  

decayG1A2, 0  

; sacl  

decayG1B2, 0  

; sacl  

wcA1, 0  ; this will force loading a wcma during first pass  

sacl  

wcB1, 0  

; sacl  

wcA2, 0  

; sacl  

wcB2, 0  

sub  

one  

sacl  

minus, 0  

lack  

dmem_rg客商age
sac1 bufdoneptrA1,0
lack dmem_wgarbage
sac1 bufdoneptrB1,0
lack 11b
sac1 twobits,0
out twobits,PA3

lack 4

sac1 biozaudio,0
lack 2
sac1 biozdsasi_new_samp,0
lac one,15
sac1 zero_bit,0
lac one,14
sac1 attack_bit,0
lac one,13
sac1 abort_bit,0
zals abortbit
xor minus
sac1 notabortbits,0

; read in parameters
lack dmem_abortcnt
tblr abortcnt
lack dmem_da_loop_times
tblr da_loop_times ; Initialize da_loop_times.
lack dmem_scntlo
tblr scntlo ; and click counter
lack dmem_scnthi
tblr scnthi
lack dmem_mem_times
tblr mem_times
lack dmem_attackincr
tblr attackincr
lack dmem_decayincr
tblr decayincr
lack dmem_aGain
tblr aGain
lack dmem_bGain
tblr bGain
lack dmem_wcma_cntAl
tblr wcma_cntAl
lack dmem_wcma_cntBl
tblr wcma_cntBl

; initialize flags in dmem:
lack dmem_tmp
tblw minus
lack dmem_err_da
tblw zero
lack dmem_err_rDMA
tblw zero
lack dmem_err_wDMA ; debugging only
tblw zero
lack dmem_err_no_wcmaAl
tblw zero
lack dmem_err_no_wcmaA1
tblw zero
lack dmem_err_no_wcmaA2
;
tblw zero
;
lack dmem_err_no_wcmaB2
;
tblw zero

; initialize attack filter gains
lt attackincr
mpy maxpos
pac
sach attackG1,0 ; *not* sach ...1 --- see documentation
zals maxpos
subs attackincr
sacl attackG2,0
lack dmem_attackG1
tblw attackG1
lack dmem_attackG2
tblw attackG2

; initialize decay filter gains
lt decayincr
mpy maxpos
pac
sach decayG1,0 ; *not* sach ...1 --- see documentation
zals maxpos
subs decayincr
sacl decayG2,0
lack dmem_decayG1
tblw decayG1
lack dmem_decayG2
tblw decayG2
sovm

; make sure there's a wcma out there for A1

iswcA1: lack dmem_first_wcmaA1
tblr nextWcmaA1 ; since wcA1 (above) = 0, first time
zals nextWcmaA1
bnz iswcA12 ; code will decipher wcma address from
lack dmem_err_no_wcmaA1 ; here
 tblw minus
 b iswcA1 ; spin wheels until a wcma arrives
 ; in order to permit "pipelining" in 8-sample click loop, we'll
 ; read in the entire first wcma here
iswcA12: zals bufdoneptrA1 ; * set done flag and
 tblw minus ; * (bufdoneptrA1 contains address)
 ; [ previous two only for debugging, pull out xgarbage entirely ]
readWCa1: zals nextWcmaA1 ; * get a new wcma;
 tblr wcA1 ; * (nextWcmaA1 contains address)
zals wcA1
bnz wcA1OK ; * trap on wc=0?
lack dmem_err_wcA1_zero ; *
tblw nextWcmaA1 ; *
 b readWCa1 ; *
wC1A1OK: zals nextWcmaA1 ; *
 adds one ; *
tblr memhiA1 ; *
sacl  flaglocA1,0 ; *
adds  one ; *
tblr  memloA1 ; *
adds  one ; *
tblr  nextWcmaA1,0 ; *
adds  one ; *
tblr  sdecayA1 ; *
adds  one ; *
sacl  bufdoneptrA1,0 ; *

; note that you always get an attack on the first wcma

; make sure there's a wcma out there
iswCB1l: lack  dmem_first_wcmaBl
  tblr  nextWcmaBl ; since wcBl (above) = 0, first time
  zals  nextWcmaBl
  bnz  iswCB12 ; code will decipher wcma address from
  lack  dmem_err_no_wcmaBl ; here
  tblw  minus
  b   iswCB1l ; spin wheels until a wcma arrives
; in order to permit "pipelining" in 8-sample click loop, we'll
; read in the entire first wcma here
iswCB12: zals  bufdoneptrBl ; * set done flag and
  tblw  minus ; * (bufdoneptrBl contains address)
  ; [ previous two only for debugging, pull out xgarbage entirely ]
readWcBl1: zals  nextWcmaBl ; * get a new wcma;
  tblr  wcBl ; * (nextWcmaBl contains address)
  zals  wcBl
  bnz  wcBl10K ; * yes, set flag
  lack  dmem_err_wcBl_zero ; *
  tblw  nextWcmaBl ; *
  b   readWcBl1 ; *
wcBl10K: zals  nextWcmaBl ; *
  adds  one ; *
  tblr  memhiBl ; *
sacl  flaglocBl,0 ; *
  adds  one ; *
  tblr  memloBl ; *
  adds  one ; *
  tblr  nextWcmaBl,0 ; *
  adds  one ; *
  tblr  sdecayBl ; *
  adds  one ; *
sacl  bufdoneptrBl,0 ; *

; note that you always get an attack on the first wcma

#endif INITMEM
; init MEM[0x2000:0x21FE (DDT style)] for debugging ***************
lark  AR0, 255
lark  AR1, 0
out  biozMEMdone, PA5
lack  PLAY2_done, wcma
adds  one
tblr  tmp ; read memhi for first sample buffer
out  tmp, PA1
lack  PLAY2 PA0
tblw  tmp
lack PLAY2_sample_wcmal
adds two
	tblr tmp ; read memlo for ditto
out tmp, PA0
lack PLAY2_sample_PA1
tblw tmp

D01: 
#ifdef FILL_MEM_WITH_ONES
	out maxpos, PA2 ; for debugging attack/decay
#else
	larp ARL
	mar ++
	sar ARL, tmp ; for debugging readin
	out tmp, PA2
#endif
zals mem_times

D03: 
bioz D05 ; WHILE NOT MEM done
subs one
bgz D03 ; loop
lack dmem_err_wDMA ; mem hung, therefore,
tblw minus ; flag hung
zals mem_times
b D03

D05: 
larp AR0
banz D01

; now set up a fake silence buffer
lack PLAY2_zero_wcmal
adds one
	tblr tmp ; read memhi for first zero buffer
out tmp, PA1
lack PLAY2_zero_PA0
tblw tmp
lack PLAY2_zero_wcmal
adds two
	tblr tmp ; read memlo for ditto
out tmp, PA0
lack PLAY2_zero_PA1
tblw tmp
larp AR0
lark AR0, 15

D08: 
out zero,PA2
zals mem_times

D10: 
bioz D11 ; WHILE NOT MEM done
subs one
bgz D10 ; loop
lack dmem_err_wDMA ; mem hung, therefore,
tblw minus ; flag hung
zals mem_times
b D10

D11: banz D08

#endif

; zero out internal storage of input. We *know* that the
; gain terms are all zero until the first pass through the
; click code, so we leave the inputs and outputs all zero
; for that first pass. The alternative is to read in here
; 8 inputs for all four streams and to advance the
; gain terms for all four streams here as well, then calculate
; here the 4 initial output samples (5 through 8 get calculated
; in the first part of the click loop).
lark AR0, 23  ; 8 samples * 3 streams - 1 for banz
lark AR1, insampAll
zac zeroin0:larp AR1
sacl *+,0,AR0
banz zeroin0

; zero out output samples too.
lark AR0, 15  ; 8 samples * (2 for stereo) - 1 for banz
lark AR1, outsampA1
zac zeroout0:larp AR1
sacl *+,0,AR0
banz zeroout0

; debugging only
lack dmem_inputA1
tblw insampA1
lack dmem_inputA2
tblw insampA2
lack dmem_inputA3
tblw insampA3
lack dmem_inputA4
tblw insampA4
lack dmem_inputA5
tblw insampA5
lack dmem_inputA6
tblw insampA6
lack dmem_inputA7
tblw insampA7
lack dmem_inputA8
tblw insampA8
lack dmem_inputB1
tblw insampB1
lack dmem_inputB2
tblw insampB2
lack dmem_inputB3
tblw insampB3
lack dmem_inputB4
tblw insampB4
lack dmem_inputB5
tblw insampB5
lack dmem_inputB6
tblw insampB6
lack dmem_inputB7
tblw insampB7
lack dmem_inputB8
tblw insampB8
; debugging
lack dmem_attackA1
tblw attackA1
lack dmem_decayA1
tblw decayA1
lack dmem_gainA1
tblw gainA1
lack dmem_attackB1
tblw attackB1
lack dmem_decayB1
tblw decayB1
lack dmem_gainB1
tblw gainB1

; start "pipelining" internal to click loop
lt insampA15

; flag init done
lack dmem_init_done
tblw minus

; click loop starts here
; ;* means: this code happens only once per clique
; ************* first sample ***************

; check hang flag (7 ix); *
bisamplel: lack dmem hang ; *
tblr hang ; *
zals hang ; *
bnz bisamplel ; *

; set up reading All:Al8, start read All (6)
; the should be at least 10 machine cycles between
; each invocation of out twobits (see diagnostics documentation)
out memloA1, PA0
out memhiA1, PA1

startAll: out twobits, PA3

samplel: ; output sample 1 (9 ix if bioz fall through)
out biozdasi_new samp, PA5
zals da_loop_times

L11: DABIOZ L12
subs one
bgz L11
lack dmem_err_da
tblw one
zals da_loop_times
b L11

L12: out outsampA1, PA6
out outsampB1, PA7

; read All, start read Al2 (9)
out biozMEMdone, PA5
zals mem_times

;0: bioz readAll
subs one
```assembly
bgz L00
lack dmem_err_rDMA
tblw one
zals mem_times
b L00
readAll: in insampA11, PA2
startA2: out twobits, PA3

; CALCA5
; calculate A5 sample (6 ix).
; This must happen before read A15, below
lt insampA15 ; already in place from init or from sample 8:
mpy gainA1
pac
lt insampA25
mpy gainA2
lta insampA16
sach outsampA5, 1

; CALCA6
; calculate A6 sample (6 ix)
; This must happen before read A16, below
lt insampA16
mpy gainA1
pac
lt insampA26
mpy gainA2
lta insampA17
sach outsampA6, 1

; read A12, start read A13 (7)
; out biozMEMdone, PA5
zals mem_times
L04: bioz readA12
subs one
bgz L04
lack dmem_err_rDMA
tblw two
zals mem_times
b L04
readA12: in insampA12, PA2
startA3: out twobits, PA3

; take care of wc, wcma, memhi, memlo for A1 (***** ix max).
; The readWcA1/noWcmaA1 branches are exactly the same length.
; This block must happen after out memhiA1, memloA1, above
zals wcA1 ; * decrement wcA1
subs one ; *
; read in wcma for A1?
bnz noWcmaA1 ; *
zals bufdoneptrA1 ; * set done flag and
tblw minus ; * (bufdoneptrA1 contains address)
readWcA1:
lack dmem_wcmaHangA1
nop
co: lack dmem_wcmaHangA1 ; tblw nextWcmaA1 = 0x7D15
```
tblr  tmp
zals  tmp
bnz  fco
zals nextWcmaAl
; * get a new wcma;
tblr wcAl
; * (nextWcmaAl contains address)
zals wcAl
bnz wcAlOK
; * trap on wc=0?
lack dmem_err_wcAl_zero
; *
tblw nextWcmaAl
; *
b readWcAl
; *
wcAlOK: zals wcma_cntAl
; *
adds one
; *
sacl wcma_cntAl,0
; *
zals nextWcmaAl
; *
adds one
; *
tblr memhiAl
; *
sacl flaglocAl,0
; *
adds one
; *
tblr memloAl
; *
adds one
; *
tblr nextWcmaAl,0
; *
adds one
; *
tblr sdecayAl
; *
adds one
; *
sacl bufdoneptrAl,0
; *
seenAl: tblr tmp
zals tmp
bz notseenAl
; lack dmem_err_wcAl_seen
tblw wcma_cntAl
; zals bufdoneptrAl
; b seenAl
notseenAl:
; zals memhiAl
and attackbit
; *
bz nonewattackAl
; * and maybe
; zac nonewattackAl
; * reset attack
; sach attackAl,0
; * (AC hi has zero from and)
sach decayGlAl,0
; * and turn off decay.
; b wasWcmaAl
; *
nonewattackAl:
; zals memhiAl
and zerobit
; *
bz wasWcmaAl
; *
sacl zeroAl,0
; *
sach gainAl,0
; *
; b wasWcmaAl
; *
; store decremented wcAl (****** ix count included in above)
ncWcmaAl: sacl wcAl,0
; *
; increment MEM address for Al. This is why this wcma handling
; must happen after out memhiAl, memloAl above.
zalh memhiAl
; * increment mem address
adds memloAl
; *
adds eight
; *
sach memhiAl,0
; *
sacl memloAl,0
zals flagloAl
	* Get current flags.
tblr flagAl
zals flagAl
and abortbit
	* Shall we abort this stream?
bz tst0Al
zalh wcAl
	* Yes.
blz stompAl
	* If the sign bit in wcAl is still set,
	* we *know* we have a huge wcAl. Stomp it.
subh abortcnt
blz abortAl
stompAl: zals abortcnt
	* wc>abortcnt. Stomp wc.
sacl wcAl
	*** ; following five lines duplicated from below, just above LAlGl:
abortAl: zals wcAl
	* this cute kludge will start up the
adds one
	* decay at LAlGl, below
sacl sdecayAl,0
	; zals maxpos
	* Start decay where attack was and
; subs attackAl
	;*
sacl decayAl,0
	;*
zals decayGl
	* signal that decay is active.
sacl decayGlAl,0
	;*
zals flagAl
	* turn off abort bit
and notabortbits
	;*
sacl flagAl,0
zals flagloAl
	;*
tblw flagAl

+st0Al: zals flagAl
	* Is this buffer (also) all zeros?
and zerobit
	;*
sacl zeroAl,0
bz wasWcmaAl
	;*
sach gainAl,0
	* (ac hi is 0 from and)
	* A 0 in gainAl guarantees that at CALCA1,
	* CALCA2 etc the contribution from stream Al
	* will be 0.

wasWcmaAl:
	; read Al3, start Al4 (9)
out biozMEMdone, PA5
zals mem_times

L14: bioz readA13
subs one
bgz L14
lack dmem_err_rDMA
tblw three
zals mem_times
b L14

readA13: in insampA13, PA2
startA14: out twobits, PA3


; ***************
; output sample # 2 (9)
sample2: out biozdasi_new_samp, PA5
zals da_loop_times

L21: DABIOZ L22
subs one
bgz L21
lack       dmem_err_da
tblw       two
zals       da_loop_times
b          L21
L22:       out       outsampA2, PA6
            out       outsampB2, PA7
            ; read A14, start A15 (7)
            out       biozMEMdone, PA5
            zals       mem_times
L17:       bioz       readA14
            subs       one
            bz         L17
            lack       dmem_err_rDMA
            tblw       four
            zals       mem_times
            b          L17
readA14:   in         insampA14, PA2
startA15:  out         twobits, PA3
            ; CALCA7
            ; calculate A7 sample (6)
            ; This must happen before read A17, below
            lt         insampA17
            mpy        gainA1    ; insampA17 in T register from sample6
            pac
            lt         insampA27
            mpy        gainA2
            lta        insampA18
            sach       outsampA7, 1
            ; CALCA8
            ; calculate A8 sample (6)
            ; This must happen before read A18, below
            lt         insampA18
            mpy        gainA1
            pac
            lt         insampA28
            mpy        gainA2
            lta        insampB15
            sach       outsampA8, 1
            ; read A15, start A16 (9)
            ; out       biozMEMdone, PA5
            zals       mem_times
L24:       bioz       readA15
            subs       one
            bz         L24
            lack       dmem_err_rDMA
            tblw       five
            zals       mem_times
            b          L24
readA15:   in         insampA15, PA2
startA16:  out         twobits, PA3
            _CALCB5
; calculate B5 sample (6 ix)
lt  insampB15
mpy  gainB1
pac
lt  insampB25
mpy  gainB2
lt  insampB16 ;  lta  insampB16
sach  outsampB5, 1

; CALCB6
; calculate B6 sample (6 ix)
lt  insampB16
mpy  gainB1
pac
lt  insampB26
mpy  gainB2
lt  insampB17 ;  lta  insampB17
sach  outsampB6, 1

; read A16, start read A17 (7 ix)
out  biozMEMdone, PA5
zals  mem_times
L27:  bioz  readA16
subs  one
bgz  L27
lack  dmem_err_rDMA
tblw  six
zals  mem_times
b  L27
readA16:  in  insampA16, PA2
startA17:  out  twobits, PA3

; CALCB7
; calculate B7 sample (6)
lt  insampB17
mpy  gainB1
pac
lt  insampB27
mpy  gainB2
lt  insampB18 ;  lta  insampB18
sach  outsampB7, 1

; read A17, start A18 (7 ix)
out  biozMEMdone, PA5
zals  mem_times
L210:  bioz  readA17
subs  one
bgz  L210
lack  dmem_err_rDMA
tblw  seven
zals  mem_times
b  L210
readA17:  in  insampA17, PA2
startA18:  out  twobits, PA3
\[\text{; read in new aGain, bGain (8 ix)}\]
\[\text{tldr aGain} ; *\]
\[\text{tldr bGain} ; *\]
\[\text{; read in new bGain} ; *\]
\[\text{lack dmem_bGain} ; *\]
\[\text{tblr bGain} ; *\]

\[\text{; read A18 (5 ix)}\]
\[\text{out biozMEMdone, PA5}\]
\[\text{zals mem_times}\]
\[\text{L213: bioz readA18}\]
\[\text{subs one}\]
\[\text{bgz L213}\]
\[\text{lack dmem_err_rDMA}\]
\[\text{tblw eight}\]
\[\text{zals mem_times}\]
\[\text{b L213}\]
\[\text{readA18: in insampA18, PA2}\]

\[\text{; CALC8}\]
\[\text{; calculate B8 sample (6)}\]
\[\text{lt insampB18}\]
\[\text{mpy gainB1}\]
\[\text{pac}\]
\[\text{lt insampB28}\]
\[\text{mpy gainB2}\]
\[\text{apac} \quad \text{; use lta for next sample?}\]
\[\text{lt attackA1} \quad \text{; lta attackA1}\]
\[\text{sach outsampB8, l}\]

\[\text{; } \]
\[\text{; } \]

\[\text{bisample2:}\]
\[\text{; set up reading B11:B18, start read B11 (6)}\]
\[\text{out memloB1, PA0}\]
\[\text{out memhiB1, PA1}\]
\[\text{startB11: out twobits, PA3}\]

\[\text{; output sample 3 (9 ix if bioz fall through)}\]
\[\text{sample3: out biozdasi_newsamp, PA5}\]
\[\text{zals da_loop_times}\]
\[\text{L31: DABIOZ L32}\]
\[\text{subs one}\]
\[\text{bgz L31}\]
\[\text{lack dmem_err_da}\]
\[\text{tblw three}\]
\[\text{zals da_loop_times}\]
\[\text{b L31}\]
\[\text{L32: out outsampA3, PA6}\]
\[\text{out outsampB3, PA7}\]

\[\text{; NB this and other reads can be folded into wcmaB1, gain calculation for A1, B1 if need to spread out reads within this bisample}\]
; read B11, start read B12 (7)
out  biozMEMdone, PA5
zals  mem_times
L216:  bioz  readB11
subs  one
bgz  L216
lack  dmem_err_RDMA
tblw  one
zals  mem_times
b  L216
readB11:  in  insampB11, PA2
startB12:  out  twobits, PA3

; take care of wc, wcma, memhi, memlo for B1 (****** ix max).
; The readWcB1/noWcmaB1 branches are exactly the same length.
; This block must happen after out memhiB1, memloB1, above
zals  wcB1  ; * decrement wcB1
subs  one  ; *
; read in wcma for B1?
bnz  noWcmaB1  ; *
zals  bufdoneptrB1  ; * set done flag and
tblw  minus  ; *(bufdoneptrB1 contains address)
readWCb1:
    lack  dmem_wcmaHandleB1
    nop
fooB1:  lack  dmem_wcmaHandleB1
    tblr  tmp
    zals  tmp
    bnz  fooB1
    zals  nextWcmaB1  ; * get a new wcma;
    tblr  wcB1  ; *(nextWcmaB1 contains address)
    zals  wcB1  ; * trap on wc=0?
    bnz  wcB10K  ; * yes, set flag
    lack  dmem_err_wcB1_zero  ; *
    tblw  nextWcmaB1  ; *
    b  readWCb1  ; *

wcB10K:  zals  wcma_cntB1  ; *
    adds  one  ; *
    sacl  wcma_cntB1,0  ; *
    zals  nextWcmaB1  ; *
    adds  one  ; *
    tblr  memhiB1  ; *
    sacl  flaglocB1,0  ; *
    adds  one  ; *
    tblr  memloB1  ; *
    adds  one  ; *
    tblr  nextWcmaB1,0  ; *
    adds  one  ; *
    tblr  sdecayB1  ; *
    adds  one  ; *
    sacl  bufdoneptrB1,0  ; *
seenB1:  tblr  tmp
    zals  tmp
    bz  notseenB1
    lack  dmem_err_wcB1_seen
tblw  wcma_cntBl
    zals  bufdoneptrBl
    b    seenBl

notseenBl:
    zals  memhiBl  ; *
    and  attackbit  ; *
    bz    nonewattackBl  ; * and maybe
    ; zac
    sach  attackBl,0  ; * (AC hi has zero from and)
    sach  decayG1Bl,0  ; * and turn off decay.
    b    wasWcmaBl

nonewattackBl:
    zals  memhiBl  ; *
    and  zeroBit  ; *
    bz    wasWcmaBl  ; *
    sacl  zeroBl,0  ; *
    sach  gainBl,0  ; *
    b    wasWcmaBl  ; *
    ; store decremented wcBl (****** ix count included in above)

noWcmaBl:  sacl  wcBl,0
    ; increment MEM address for Bl. This is why this wcma handling
    ; must happen after out memhiBl, memloBl above.
    zalh  memhiBl  ; * increment mem address
    adds  memloBl  ; *
    adds  eight  ; *
    sach  memhiBl,0  ; *
    sacl  memloBl,0  ; *
    zals  flaglocBl  ; * Get current flags.
    tblr  flagBl  ; *
    zals  flagBl  ; *
    and  abortbit  ; * Shall we abort this stream?
    bz    tst0Bl
    zalh  wcBl  ; * Yes.
    blz    stompBl  ; * If the sign bit in wcBl is still set,
    ;  we *know* we have a huge wcBl. Stomp it.
    subh  abortcnt  ; *
    blz    abortBl  ; *

stompBl:  zals  abortcnt  ; * wc>abortcnt. Stomp wc.
    sacl  wcBl,0  ; *

; *** ; following five lines duplicated from below, just above LB1Gl:

abortBl:  zals  wcBl  ; * this cute kludge will start up the
    adds  one  ; * decay at LB1Gl, below
    sacl  sdecayBl,0
    ; zals  maxpos  ; * Start decay where attack was and
    ; subs  attackBl  ; *
    ; sacl  decayBl,0  ; *
    ; zals  decayGl  ; * signal that decay is active.
    ; sacl  decayG1Bl,0  ; *
    ; zals  flagBl  ; * turn off abort bit
    and  notabortbits  ; *
    sacl  flagBl,0  ; *
    zals  flaglocBl  ; *
    tblw  flagBl  ; *

tst0Bl:  zals  flagBl  ; * Is this buffer (also) all zeros?
    and  zeroBit  ; *
    sacl  zeroBl,0  ; *
bz wasWcmaBl
sach gainBl,0

; * (ac hi is 0 from and)
; * A 0 in gainBl guarantees that at CALCBl,
; * CALCA2 etc the contribution from stream Bl
; * will be 0.

wasWcmaBl:

; read Bl2, start read Bl3 (7)
out blozMEMdone, PA5
zals mem_times
L219: bioz readBl2
subs one
bgz L219
lack dmem_err_rDMA
tblw two
zals mem_times
b L219
readBl2: in insampBl2, PA2
startBl3: out twobits, PA3

; * get new gain term for Al (24 *x max)
zals zeroAl
bnz LA1G2 ; * Leave gain at 0.
zalh decayGlAl ; * (NB: shifted)
bnz LA1G1
zalh attackGl ; * Using attack.
l attackAl ; * (attackAl already in T register).
mpy attackG2 ; * Get new attack term.
lta aGain
sach attackAl,1
mpy attackAl ; * Scale it by gain.
pac
sach gainAl,1
zals WCAl ; * Start decay now? (WCAl must not = 0!)
subs sdecayAl
bgz LA1G2
; if you change from here to LA1G1, change above at nowcmaAl to match
zals maxpos ; * Yes. Start decay where attack was and
subs attackAl
sach decayAl,0
zals decayGl
scl decayGl,0 ; * signal that decay is active.
scl decayGlAl,0
b LA1G2

LA1G1: lt decayAl ; * OK, we'll use decay.
mpy decayG2 ; * Get new decay term.
lta aGain ; * (decayGlAl still in AC hl)
sach decayAl,1
zals maxpos
subs decayAl
scl tmp,0
mpy tmp ; * Scale it by gain.
pac
sach gainAl,1

LA1G2:  

**********
; read B13, start B14 (9)
out biozMEMdone, PA5
zals mem_times
L34: bioz readB13
subs one
bgz L34
lack dmem_err_rDMA
tblw three
zals mem_times
b L34
readB13: in insampB13, PA2
startB14: out twobits, PA3

; output sample # 4 (9)
sample4: out biozdasi_new_samp, PA5
zals da_loop_times
L41: DABIOZ L42
subs one
bgz L41
lack dmem_err_da
tblw four
zals da_loop_times
b L41
L42: out outsampA4, PA6
out outsampB4, PA7

; read B14, start B15 (7)
out biozMEMdone, PA5
zals mem_times
L37: bioz readB14
subs one
bgz L37
lack dmem_err_rDMA
tblw four
zals mem_times
b L37
readB14: in insampB14, PA2
startB15: out twobits, PA3

; read B15, start B16 (9)
; out biozMEMdone, PA5
zals mem_times
L44: bioz readB15
subs one
bgz L44
lack dmem_err_rDMA
tblw five
zals mem_times
b L44
readB15: in insampB15, PA2
startB16: out twobits, PA3

; read B16, start read B17 (7 ix if bioz fall through)
out biozMEMdone, PA5
zals mem_times
Aug 12 11:27 1986  play2.a Page 28

L47:  bioz  readB16
       subs  one
       bgz  L47
       lack  dmem_err_rDMA
       tblw  six
       zals  mem_times
       b  L47
readB16:  in  insampB16, PA2
startB17:  out  twobits, PA3

; * get new gain term for Bl (24 ix max)
   zals  zeroBl  ; *
   bnz  LBlG2  ; * Leave gain at 0.
   zalh  decayG1Bl  ; *(NB: shifted)
   bnz  LBlG1  ; *
   zalh  attackG1  ; * Using attack.
   lt  attackBl  ; * (***) can this be folded in somewhere?)
   mpy  attackG2  ; * Get new attack term.
   lta  bGain  ; *
   sach  attackBl,l  ; *
   mpy  attackBl  ; * Scale it by gain.
   pac  ; *
   sach  gainBl,l  ; *
   zals  wcBl  ; * Start decay now? (wcBl must not = 0!)
   subs  sdecayBl  ; *
   bgz  LBlG2  ; *
   ; if you change from here to LBlG1, change above at nowcmaBl to match
   zals  maxpos  ; * Yes. Start decay where attack was and
   subs  attackBl  ; *
   sacl  decayBl,0  ; *
   zals  decayG1  ; * signal that decay is active.
   sacl  decayG1Bl,0  ; *
   b  LBlG2  ; *
LBlG1:  lt  decayBl  ; * OK, we'll use decay.
   mpy  decayG2  ; * Get new decay term.
   lta  bGain  ; *(decayG1Bl still in AC hi)
   sach  decayBl,l  ; *
   zals  maxpos  ; *
   subs  decayBl  ; *
   sacl  tmp,0  ; *
   mpy  tmp  ; * Scale it by gain.
   pac  ; *
   sach  gainBl,l  ; *
LBlG2:  

; read Bl7, start Bl8  (7 ix if bioz fall through)
; out  biozMEMdone, PA5
   zals  mem_times
L410:  bioz  readB17
       subs  one
       bgz  L410
       lack  dmem_err_rDMA
       tblw  seven
       zals  mem_times
       b  L410
__adB17:  in  insampB17, PA2
startB18: out twobits, PA3

; read B18 (5 ix if bioz falls through)
; out biozMEMdone, PA5
zals mem_times
L413: bioz readB18
subs one
bgz L413
lack dmem_err_rDMA
tblw eight
zals mem_times
b L413
readB18: in insampB18, PA2

; ******************************************

bisample3:

; output sample 5 (9 ix if bioz fall through)
sample5: out biozdas1_new_samp, PA5
zals da_loop_times
L51: DABIOZ L52
subs one
bgz L51
lack dmem_err_da
tblw five
zals da_loop_times
b L51
L52: out outsampA5, PA6
out outsampB5, PA7

; set up reading A21:A28, start read A21 (6)
; out biozMEMdone, PA5
;A2 out memloA2, PA0
;A2 out memhiA2, PA1
;A2 out twobits, PA3

;A2 ; read A21, start read A22 (7)
;A2 ; out biozMEMdone, PA5
;A2 zals mem_times
;A2 L416: bioz L417
;A2 subs one
;A2 bgz L416
;A2 lack dmem_err_rDMA
;A2 tblw one
;A2 zals mem_times
;A2 b L416
;A2 L417: in insampA21, PA2
;A2 out twobits, PA3

;A2 ; take care of wc, wcma, memhi, memlo for A2 (43 max).
wasWcmaA2:

;A2 ; read A22, start read A23 (7)
A2 ; out biozMEMdone, PA5
;A2 zals mem_times
;A2 L419:       bioz    L420
A2    subs     one
;A2    bgz     L419
;A2    lack     dmem_err_DMA
;A2    tblw     two
;A2    zals     mem_times
;A2    b        L419
;A2 L420:       in      insampA22, PA2
;A2    out      twobits, PA3

;A2    ; * get new gain term for A2 (21 ix max)
;A2    ; **** check lta/lt once this is installed

; ************

; samples out, for debug
lack    dmem_output1
tblw    outsampB1
lack    dmem_output2
tblw    outsampB2
lack    dmem_output3
tblw    outsampB3
lack    dmem_output4
tblw    outsampB4
lack    dmem_output5
tblw    outsampB5
lack    dmem_output6
tblw    outsampB6
lack    dmem_output7
tblw    outsampB7
lack    dmem_output8
tblw    outsampB8

; debugging
lack    dmem_attackB1
tblw    attackB1
lack    dmem_decayB1
tblw    decayB1
lack    dmem_gainB1
tblw    gainB1
lack    dmem_attackA1
tblw    attackA1
lack    dmem_decayA1
tblw    decayA1
lack    dmem_gainA1
tblw    gainA1

; output sample # 6 (9)
sample6:  out    bioz3asi_new_samp, PA5
zals    da_loop_times

L61:     DABIOZ   L62
subs     one
bgz      L61
lack    dmem_err_da
tblw    four
zals    da_loop_times
b L61
32: out outsampA6, PA6
    out outsampB6, PA7

;A2 ; read A23, start A24 (9)
;A2 out biozMEMdone, PA5
;A2 zals mem_times
;A2 L54: bioz L55
;A2 subs one
;A2 bgz L54
;A2 lack dmem_err_rDMA
;A2 tblw three
;A2 zals mem_times
;A2 b L54
;A2 L55: in insampA23, PA2
;A2 out twobits, PA3

; while we're waiting for MEM, copy WCMA (debugging only)
    lack dmem_err_wcAl_copy ; *
    tblw wcAl ; *
    adds one ; *
    tblw memhiAl ; *
    adds one ; *
    tblw memloAl ; *
    adds one ; *
    tblw sdecayAl ; *
    adds one ; *
    tblw bufdoneptrAl ; *
    adds one ; *
    tblw nextWcmaAl ; *
    adds one ; *
    tblw flagAl ; *
    adds one ; *
    tblw zeroAl ; *

; Bl too
    lack dmem_err_wcBl_copy ; *
    tblw wcBl ; *
    adds one ; *
    tblw memhiBl ; *
    adds one ; *
    tblw memloBl ; *
    adds one ; *
    tblw sdecayBl ; *
    adds one ; *
    tblw bufdoneptrBl ; *
    adds one ; *
    tblw nextWcmaBl ; *
    adds one ; *
    tblw flagBl ; *
    adds one ; *
    tblw zeroBl ; *

;A2 ; read A24, start A25 (7)
;A2 ; out biozMEMdone, PA5
2 zals mem_times
;A2 L57: bioz L58
;A2 subs one
;A2 bgz L57
;A2 lack dmem_err_rDMA
;A2 tblw four
;A2 zals mem_times
;A2 b L57
;A2 L58: in insampA24, PA2
;A2 out twobits, PA3

;A2 ; read A25, start A26 (9)
;A2 out biozMEMdone, PA5
;A2 zals mem_times
;A2 L64: bioz L65
;A2 subs one
;A2 bgz L64
;A2 lack dmem_err_rDMA
;A2 tblw five
;A2 zals mem_times
;A2 b L64
;A2 L65: in insampA25, PA2
;A2 out twobits, PA3

; * get new gain term for A2 (21 ix max)
; **** check lta/lt once this is installed

;A2 ; read A26, start read A27 (7 ix if bioz fall through)
;A2 ; out biozMEMdone, PA5
;A2 zals mem_times
;A2 L67: bioz L68
;A2 subs one
;A2 bgz L67
;A2 lack dmem_err_rDMA
;A2 tblw six
;A2 zals mem_times
;A2 b L67
;A2 L68: in insampA26, PA2
;A2 out twobits, PA3

;A2 ; read A27, start A28 (7 ix if bioz fall through)
;A2 ; out biozMEMdone, PA5
;A2 zals mem_times
;A2 L610: bioz L611
;A2 subs one
;A2 bgz L610
;A2 lack dmem_err_rDMA
;A2 tblw seven
;A2 zals mem_times
;A2 b L610
;A2 L611: in insampA27, PA2
;A2 out twobits, PA3

;A2 ; read A28 (5 ix if bioz falls through)
;A2 ; out biozMEMdone, PA5
;A2 zals mem_times
;A2 L613: bioz L614
;A2 subs one
Aug 12 11:27 1986  play2.a Page 33

;A2  bgz   L613

\2  lack   dmem_err_rDMA

;A2  tblw  eight

;A2  zals  mem_times

;A2  b     L613

;A2 L614:   in   insampA28, PA2

;  ************************************************

bisample4:

; output sample 7 (9 ix if bioz fall through)

sample7: out   biozdsai_new_samp, PA5

zals   da_loop_times

L71:   DABIOZ L72

subs   one

bgz    L71

lack   dmem_err_da

tblw   one

zals   da_loop_times

b      L71

L72:   out   outsampA7, PA6

out    outsampB7, PA7

; set up reading B21:B28, start read B21 (6)

; out   biozMEMdone, PA5

;B2    out   memloB2, PA0

;B2    out   memhiB2, PA1

;B2    out   twobits, PA3

; CALCAL

; calculate A1 sample (7 ix).

  lt   insampAl   ; lta in previous cycle? ***********

  mpy  gainAl

  pac

  lt   insampA2l

  mpy  gainA2

  lta  insampB1l

  sach   outsampAl, 1

;B2   ; read B21, start read B22 (7)

;B2   ; out   biozMEMdone, PA5

;B2    zals   mem_times

;B2 L616:   bioz   L617

;B2    subs   one

;B2    bgz    L616

;B2    lack   dmem_err_rDMA

;B2    tblw   one

;B2    zals   mem_times

;B2    b      L616

;B2 L617:   in   insampB21, PA2

;B2    out   twobits, PA3

; CALCBl

; calculate B1 sample (6 ix)

; This must happen after read B21, above
lt insampB11
mpy gainB1
pac

lt insampB21
mpy gainB2
lt insampA12 ; lta insampA2
sach outsampB1, 1

; take care of wc, wcma, memhi, memlo for B2 (43 max).

;B2

; debugging only
lack dmem_decayG1A1
tblw decayG1A1
lack dmem_decayG1B1
tblw decayG1B1
lack dmem_attackbit
tblw attackbit
lack dmem_zero bit
tblw zeroBit
lack dmem_abort bit
tblw abortbit
lack dmem_inputA1
tblw insampA1
lack dmem_inputA2
tblw insampA2
lack dmem_inputA3
tblw insampA3
lack dmem_inputA4
tblw insampA4
lack dmem_inputA5
tblw insampA5
lack dmem_inputA6
tblw insampA6
lack dmem_inputA7
tblw insampA7
lack dmem_inputA8
tblw insampA8
lack dmem_inputB1
tblw insampB1
lack dmem_inputB2
tblw insampB2
lack dmem_inputB3
tblw insampB3
lack dmem_inputB4
tblw insampB4
lack dmem_inputB5
tblw insampB5
lack dmem_inputB6
tblw insampB6
lack dmem_inputB7
tblw insampB7
lack dmem_inputB8
tblw insampB8
lack dmem_wcma_cntA1
tblw wcma_cntA1
lack  dmem_wcma_cntB1
  tblw  wcma_cntB1

;  ************

;  output sample # 8 (9)
sample8:  out  biozdazi_new_samp, PA5
zals  da_loop_times
L81:  DABIOZ  L82
  subs  one
  bgz  L81
  lack  dmem_err_da
  tblw  two
  zals  da_loop_times
  b  L81
L82:  out  outsampA8, PA6
  out  outsampB8, PA7

;  calculate new click count  ;  * (13 ix)
zalh  scnthi  ;  * increment click counter,
adds  scntlo  ;  *
adds  one  ;  *
sach  scnthi  ;  *
sacl  scntlo  ;  *
lack  dmem_scntlo  ;  * store scntlo
  tblw  scntlo  ;  *
lack  dmem_scnthi  ;  * store scnthi
  tblw  scnthi  ;  *

;  CALCA2
;  ;  * (13 ix)
  lt  insampA12
  mpy  gainA1
  pac
  lt  insampA22
  mpy  gainA2
  lta  insampB12
  sach  outsampA2, 1

;B2  ;  read B22, start read B23 (7)
;B2  ;  out  biozMEMdone, PA5
;B2  zals  mem_times
;B2  L619:  bioz  L620
;B2  subs  one
;B2  bgz  L619
;B2  lack  dmem_err_RDMA
;B2  tblw  two
;B2  zals  mem_times
;B2  b  L619
;B2  L620:  in  insampB22, PA2
;B2  out  twobits, PA3

;  CALCB2
;  ;  calculate B2 sample (6 ix)
;  ;  This must happen before after [sic?] B22, above ?????????????
  lt  insampB12
mpy gainB1
pac
lt insampB22
; mpy gainB2
lt insampA13 ; lta insampA13
sach outsampB2, 1

; CALCA3
; calculate A3 sample (6)
lt insampA13
mpy gainAl ; insampA13 in T register from sample6
pac
lt insampA23
mpy gainA2
lta insampB13
sach outsampA3, 1

;B2 ; read B23, start B24 (9)
;B2 out biozMEMdone, PA5
;B2 zals mem_times
;B2 L74: bioz L75
;B2 subs one
;B2 bqz L74
;B2 lack dmem_err_DMA
;B2 tblw three
;B2 zals mem_times
;B2 b L74
;B2 L75: in insampB23, PA2 ; ************ debugging
;B2 out twobits, PA3

; CALCB3
; calculate B3 sample (6)
; This must happen after read B23, above
; lt insampB13
mpy gainB1
pac
; lt insampB23
; mpy gainB2
; lt insampA14 ; lta insampA14
sach outsampB3, 1

; CALCA4
; calculate A4 sample (6)
; lt insampA14
mpy gainAl
pac
lt insampA24
mpy gainA2
lta insampB14
sach outsampA4, 1

;B2 ; read B24, start B25 (7)
;B2 ; out biozMEMdone, PA5
;B2 zals mem_times
;B2 L77: bioz L78
;B2 subs one
Aug 12 11:27 1986 play2.a Page 37

;B2  bgz    L77
;B2  lack   dmem_err_rDMA
;B2  tblw   four
;B2  zals   mem_times
;B2  b      L77
;B2 L78:   in    insampB24, PA2
;B2 out    twobits, PA3

; CALCB4
    ; calculate B4 sample (6)
    ; This must happen before after B24, above
    lt   insampB14
    mpy  gainB1
    pac
    lt   insampB24
    mpy  gainB2
    lta  insampA15    ; insampA5 stays in place until bisamplel
    apac
    lt   insampA15    ; use lta for next sample? *******
sach  outsampB4, 1

;B2  ; read B25, start B26 (9)
;B2  out    biozMEMdone, PA5
;B2  zals   mem_times
;B2 L84:   bioz   L85
;B2 subs   one
;B2 bgz    L84
;B2 lack   dmem_err_rDMA
;B2 tblw   five
;B2 zals   mem_times
;B2 b      L84
;B2 L85:   in    insampB25, PA2
;B2 out    twobits, PA3

;B2  ; read B26, start read B27 (7 ix if bioz fall through)
;B2  ; out    biozMEMdone, PA5
;B2  zals   mem_times
;B2 L87:   bioz   L88
;B2 subs   one
;B2 bgz    L87
;B2 lack   dmem_err_rDMA
;B2 tblw   six
;B2 zals   mem_times
;B2 b      L87
;B2 L88:   in    insampB26, PA2
;B2 out    twobits, PA3

;B2  ; read B27, start B28 (7 ix if bioz fall through)
;B2  ; out    biozMEMdone, PA5
;B2  zals   mem_times
;B2 L810:  bioz   L811
;B2 subs   one
;B2 bgz    L810
;B2 lack   dmem_err_rDMA
;B2 tblw   seven
;B2 zals   mem_times
Aug 12 11:27, 1986 play2.a Page 38

;B2     b     L810
.B2     L811:   in   insampB27, PA2
     ;B2     out   twobits, PA3

;B2     ; read B28  (5 ix if bioz falls through)
;B2     ; out    biozMEMdone, PA5
;B2     zals    mem_times
;B2     L813:   bioz  L814
;B2     subs    one
;B2     bgz     L813
;B2     lack    dmem_err_rDMA
;B2     tblw    eight
;B2     zals    mem_times
;B2     b       L813
;B2     L814:   in   insampB28, PA2

; debugging ****
lack    dmem_hang
nop       ;    tblw  minus = 0x7D79

; end debugging ************
b    bisample1