DUAL PCM

FLEXIBLE - EDITION

Technical Manual

Introduction

This document explains the implementation and usage of "Dual PCM - FlexEd" (Flexible Edition), these are programming and hardware specifics about the driver and how to use and control it.

This driver was developed by MarkeyJester, with special thanks to the following people:

Natsumi ~	Thoroughly testing Dual PCM with her 68k driver, complete with hardware tests, and several suggestions from her which were implemented.
Ralakimus ~	Hardware testing.
Mike Pavone ~	For fixing requested issues with BlastEm which were of benefit to the sound driver, he was swift with his responses.
Sik ~	Supplying an improved frequency table, and a few suggestions for the manual.
SSRG ~	Feedback and decision influencing.

If anyone feels they have been neglected from the credits, do give me a shout, I may have simply forgotten.

Contents

What is it?

Dual PCM - FlexEd is a Z80 PCM playback driver for the SEGA Mega Drive/Genesis, it is an improved version built on top of it's predecessor "Dual PCM", this version is the "Flexible Edition" and it is capable of the following:

- 2 channel sample playback from 68k ROM (any location, any size, any time).
- PCM 8-bit 13,750Hz - 14,000Hz (approx).
- DMA halt quality loss prevention (stream/buffering).
- 68k YM2612 access time loss prevention (YM2612 queue).
- Automatic bank handling.
- Looping/replay functionality.
- Individual Pitch control.
- Individual Volume control.
- Individual Reverse playback.
- Individual PCM table filtering capabilities.

Sound drivers for the SEGA Mega Drive/Genesis suffer from a variety of quality issues, most commonly of which is "interruption loss", as a result, these drivers were design and used for percussion instruments only, percussion does not suffer from the loss as badly as other more dynamic sounding samples.

Dual PCM aims to change this by preventing interruption losses from occurring, this allows any samples to be played without any noticeable interruption quality issues, it also allows you to play 2 samples at once, so you can still play your percussion samples alongside the other non-percussion instruments/samples. Complete with pitch and volume changing capabilities, you can record a sample of a single note from an instrument, and play back that sample at any other note and volume of your desire.

This driver can be used in a variety of other ways too, thanks to its ability to play really long samples, complete with its looping functionality, you can in theory play background music on one channel, and SFX samples on another provided you have the ROM space to fit your sample music. The second sample channel can be used for SFX or vocal samples without interrupting the first sample channel if it's being used for percussion. There are many different ways you can use this driver, the above are just suggestions.

How you use the driver is entirely up to you, but you must program the 68k to control the driver to your liking. This document will give you plenty of information and examples which you can use and reference from.

With that in mind, thank you for choosing Dual PCM - Flexible Edition as your PCM playback driver.

Getting started

You can download Dual PCM one of the following ways:

1. A pre-assembled version http://mrjester.hapisan.com/05_DualPCM/Dual%20PCM%20-%20FlexEd.bin
2. The source code http://mrjester.hapisan.com/05_DualPCM/Dual%20PCM%20-%20FlexEd.asm

There are two ways you can install this; you can use the source code, or you can use the pre-assembled version.

1. The pre-assembled version will be easier to install, it is the simple case of including the binary file into your 68k ROM somewhere.
2. The source code version is a little messier, you will need to assemble the driver using a Z80 assembler of some kind, the benefit to using the source code version is that you have access to the labels, if you happen to have a 68k assembler which can read those labels, it also allows you the ability to make minor modifications to the driver, **Handling this is entirely down to you, if you are not sure, then please stick with the pre-assembled version.**

This document will display both the Z80 address and the label that represents that address, this will cover both the pre-assembled version and the source code version, for example $A00100 (VolumeControl), if you are using the pre-assembled version, then use $A00100, if you are using the source code version, then you can use either $A00100 or $A00000+VolumeControl.

This document will assume you are dealing with the driver as a pre-assembled file.

First, include Dual PCM into the 68k ROM somewhere:

```
Z80ROM:         incbin  "Dual PCM - FlexEd.bin"
Z80ROM_End:     even
```

Next, load Dual PCM into Z80 RAM $A00000:

```
                lea     (Z80ROM).l,a0                   ; load Z80 ROM data
                lea     ($A00000).l,a1                  ; load Z80 RAM space address
                move.w  #(Z80ROM_End-Z80ROM)-$01,d1     ; set repeat times
                move.w  #$0100,($A11100).l              ; request Z80 stop (ON)
                move.w  #$0100,($A11200).l              ; request Z80 reset (OFF)
                btst.b  #$00,($A11100).l                ; has the Z80 stopped yet?
                bne.s   *-$08                           ; if not, branch

        .LoadZ80:
                move.b  (a0)+,(a1)+                     ; copy Dual PCM to Z80 RAM
                dbf     d1,.LoadZ80                     ; repeat til done
                move.w  #$0000,($A11200).l              ; request Z80 reset (ON)
                moveq   #$7F,d1                         ; set repeat times
                dbf     d1,*                            ; no way of checking for reset, so a manual delay is necessary
                move.w  #$0000,($A11100).l              ; request Z80 stop (OFF)
                move.w  #$0100,($A11200).l              ; request Z80 reset (OFF)
```

Due to the nature of how Dual PCM works, the driver is always outputting audio data even when the channel is apparently not operational. As a result, you will need to create a block of silent PCM data in your source code somewhere, and then pass the address of this data to the driver before starting it.

Please see the below example of a block of silent PCM data:

```
                        align   $8000

                        dcb.b   $18*$10,$00                     ; End marker

SWF_MuteSample:         dcb.b   $8000-(($18*$10)*2),$80         ; a large block of silent PCM data
SWF_MuteSample_Rev:     dcb.b   $18*$10,$00                     ; End marker
```

This will create a block of $7D00 bytes of $80 PCM bytes, before and after this block are $180 bytes of $00 end marker bytes. This all mathematically adds up to $8000 bytes of data, likewise, there is an align to ensure this block of data is aligned to a Z80 window space. The "end marker" bytes will be explained later on, getting the driver installed and started is what's important.

Note: The alignment isn't necessary, nor is the block required to be so large, the driver will continue to work even without the carefully aligned data, however, given how frequently the mute sample will be used, it is recommend here. The mute sample size can be smaller, however, the smaller it is, the more often Dual PCM has to reset the address, every reset costs a fraction of time which you would rather occur least often possible.

The address of the sample now needs to be given to Dual PCM.

The addressing system of Dual PCM for samples is pretty standard practice for the Mega Drive. We'll start with an example address of our sample; $2CA04E, this has to be broken up into a bank address and a window address, to understand we will break the address into binary:

```
  2      C      A      0      4      E
0010   1100   1010   0000   0100   1110
```

The bank address and window address are denoted by "B" (Bank) and "W" (Window):

```
0010   1100   1010   0000   0100   1110
-BBB   BBBB   BWWW   WWWW   WWWW   WWWW
```

So the bank address is **01011001** ($59), and the window address is 010000001001110 ($404E). The window map address for the Z80 is $8000 - $FFFF, so you must add (logical OR) $8000 to the window address. $404E + $8000 = $C04E. The three bytes of our sample address are (in reverse order) $4E, $C0, and $59.

You will likely want to use a macro/assembly equation to do these calculations for you, so I have provided an example equation using the mute sample as a reference:

```
MuteSample:     dc.b    ((SWF_MuteSample)&$FF)
                dc.b    ((((SWF_MuteSample)>>$08)&$7F)|$80)
                dc.b    (((SWF_MuteSample)&$7F8000)>>$0F)

                dc.b    ((SWF_MuteSample_Rev)&$FF)
                dc.b    ((((SWF_MuteSample_Rev)>>$08)&$7F)|$80)
                dc.b    (((SWF_MuteSample_Rev)&$7F8000)>>$0F)
```

You may notice there are 6 bytes in the above equation example, the first three is for the "beginning" address of the sample, for reverse playback the driver needs to know where to start from the end, the "_Rev" suffix is short for "Reverse". Dual PCM will need these two addresses, one for forwards playing, one for backwards playing. Even if the sample is mute, the driver could be playing in reverse, this is a reassurance measure.

The Z80 address of the mute sample pointers is between $A00C62 and $A00C67, the first three bytes are the forwards pointer, the second three bytes are for the reverse pointer.

Going back to our Z80 loading routine, we will make a modification to it:

```
                lea     (Z80ROM).l,a0                   ; load Z80 ROM data
                lea     ($A00000).l,a1                  ; load Z80 RAM space address
                move.w  #(Z80ROM_End-Z80ROM)-$01,d1     ; set repeat times
                move.w  #$0100,($A11100).l              ; request Z80 stop (ON)
                move.w  #$0100,($A11200).l              ; request Z80 reset (OFF)
                btst.b  #$00,($A11100).l                ; has the Z80 stopped yet?
                bne.s   *-$08                           ; if not, branch

        .LoadZ80:
                move.b  (a0)+,(a1)+                     ; copy Dual PCM to Z80 RAM
                dbf     d1,.LoadZ80                     ; repeat til done
                lea     (MuteSample).l,a0               ; load mute sample address
                lea     ($A00C62).l,a1                  ; load Z80 RAM space where the pointer is to be stored
                move.b  (a0)+,(a1)+                     ; copy pointer over into Z80
                move.b  (a0)+,(a1)+                     ; ''
                move.b  (a0)+,(a1)+                     ; ''
                move.b  (a0)+,(a1)+                     ; copy "reverse" pointer over into Z80
                move.b  (a0)+,(a1)+                     ; ''
                move.b  (a0)+,(a1)+                     ; ''
                move.w  #$0000,($A11200).l              ; request Z80 reset (ON)
                moveq   #$7F,d1                         ; set repeat times
                dbf     d1,*                            ; no way of checking for reset, so a manual delay is necessary
                move.w  #$0000,($A11100).l              ; request Z80 stop (OFF)
                move.w  #$0100,($A11200).l              ; request Z80 reset (OFF)
```

The highlighted instructions have been added, these will copy the pointer address of the mute sample into the driver after it has been loaded, but before it has been started up. Now Dual PCM knows where the mute sample is and will remain silent on start-up.

The driver is now installed and ready to be used.

Sample format

In the previous chapter we established that Dual PCM requires two pointers, one for the beginning of the sample, and one for the end of the sample. Despite what you might think, the end pointer is not for calculating the size of the sample, it is strictly there for reverse playback starting point and nothing more.

The actual size of the sample is (and will always be) unknown to Dual PCM, and instead, Dual PCM detects the end of a sample by finding an "End Marker" while processing the sample.

End Markers

Byte $00 signifies the end of a sample, so long as Dual PCM does not read $00 while processing a sample, it will for all intent and purposes continue reading through the banks playing back the data as audio. These $00 bytes are referred to as "End Markers".

An "End Marker" must exist at the end of a sample, though it must also exist before the beginning of a sample too, just in-case the sample happens to be playing in reverse. A single byte of $00 will not be enough to signify the end however, due to pitch control capabilities, the end needs to be filled with $180 bytes of $00's.

The size $180 will be explained in a later chapter, you have to understand how the pitch control mechanism works before these end markers will make complete sense. For now, ensure that all samples included into the ROM have these end marker blocks before and after the file include. Here is a generic directive you can use to create the block:

```
dcb.b	$18*$10,$00
```

PCM format

The sample format is 8-bit PCM mono, with a frequency of approximately 13,750Hz to 14,000Hz. However, a byte of $00 cannot exist in the PCM sample because of the end markers, bytes $01 to $FF will be processed as audio like normal. So the format is therefore technically 255-Byte PCM mono.

You must ensure that all $00 bytes in your PCM samples are converted to $01. The download link below provides a tool for converting ".wav" files to the correct format.

http://mrjester.hapisan.com/05_DualPCM/ConvPCM.zip

I recommend creating a folder named "Samples" and unzipping the program (and the text document) into this "Samples" folder, you can then put your samples into the folder with the program.

Using this program is very simple, just pass one or more ".wav" files onto the program (you can drag and drop the files onto the program icon for convenience). The files can be 8-bit, 16-bit, mono, stereo, and any frequency. The tool will convert it to the correct frequency and format for you (including taking into account the $00 end markers), and will spit them out as ".swf" files in a folder named "incswf".

Note, this program does **not overwrite the original samples**, it leaves the original high quality versions intact and does not delete them. You may want to keep the original high quality samples for future use (you never know when you'll want the high quality versions).

Including the samples

Here is an example of including sample data, Sonic 1's drum samples are used in this example, though you may include whatever you want:

```
                        align   $8000

                        dcb.b   $18*$10,$00

SWF_MuteSample:         dcb.b   $8000-(($18*$10)*2),$80
SWF_MuteSample_Rev:     dcb.b   $18*$10,$00

SWF_S1_Kick:            incbin  "Samples\incswf\Sonic 1 Kick.swf"
SWF_S1_Kick_Rev:        dcb.b   $18*$10,$00

SWF_S1_Snare:           incbin  "Samples\incswf\Sonic 1 Snare.swf"
SWF_S1_Snare_Rev:       dcb.b   $18*$10,$00

SWF_S1_Timpani:         incbin  "Samples\incswf\Sonic 1 Timpani.swf"
SWF_S1_Timpani_Rev:     dcb.b   $18*$10,$00
```

Notice how a single end marker acts as an end marker for both the end of the previous sample, and the start of the next sample. I have included the samples nearby the mute sample, though it should not matter where you include the samples so long as an end marker exists before and after the sample in question.

The directory for these samples is up to you, my samples are in a folder named "Samples", this folder contains all of my ".wav" samples and the ConvPCM program I've linked above. I simply drop the files onto the program and it creates the folder "incswf" and puts the converted data inside that folder. So all of the converted samples I want to include are in "Samples\incswf\", and the original high quality unconverted ".wav" files are still in the "Samples" folder preserved in-case I need them for something else at a later date. I recommend setting it up this way for your own convenience.

If you want to follow along you can download the Sonic 1 drum samples in .wav format here:

http://mrjester.hapisan.com/05_DualPCM/Sonic%201%20Kick.wav
http://mrjester.hapisan.com/05_DualPCM/Sonic%201%20Snare.wav
http://mrjester.hapisan.com/05_DualPCM/Sonic%201%20Timpani.wav

Playing a sample

Now that we have samples included and the driver is setup, we can start requesting samples to be played.

Dual PCM requires four pointers:

1. Start sample *forwards*
2. Start sample *reverse*
3. Loop sample *forwards*
4. Loop sample *reverse*

Dual PCM will process the "Start sample" first, once an end marker has been reached, it will move onto the "Loop sample" and play that next, from here on out reaching an end marker will cause Dual PCM to replay the "Loop sample" over and over. You can think of the "Start sample" as an attack sample, and the "Loop sample" as a sustain sample.

For something as simple as a drum sample, you would likely set the "Start sample" to point to the drum sample, and set the "Loop sample" to point to your mute sample block, thus ensuring the drum is played once and then the driver goes back to being mute.

Below is a generic macro to help construct the four pointers:

```
dcz80           macro   Sample, SampleRev, SampleLoop, SampleLoopRev
                dc.b    ((Sample)&$FF)
                dc.b    ((((Sample)>>$08)&$7F)|$80)
                dc.b    (((Sample)&$7F8000)>>$0F)
                dc.b    (((SampleRev)-1)&$FF)
                dc.b    (((((SampleRev)-1)>>$08)&$7F)|$80)
                dc.b    ((((SampleRev)-1)&$7F8000)>>$0F)
                dc.b    ((SampleLoop)&$FF)
                dc.b    ((((SampleLoop)>>$08)&$7F)|$80)
                dc.b    (((SampleLoop)&$7F8000)>>$0F)
                dc.b    (((SampleLoopRev)-1)&$FF)
                dc.b    (((((SampleLoopRev)-1)>>$08)&$7F)|$80)
                dc.b    ((((SampleLoopRev)-1)&$7F8000)>>$0F)
                endm
```

The pointer format is the same described in the "Getting started" chapter for the mute sample, this is simply a chain of four. In the previous chapter I show including Sonic 1's drum set as an example, we will create pointers for these:

```
Sonic1Kick:     dcz80   SWF_S1_Kick,            SWF_S1_Kick_Rev,        SWF_MuteSample,         SWF_MuteSample_Rev
Sonic1Snare:    dcz80   SWF_S1_Snare,           SWF_S1_Snare_Rev,       SWF_MuteSample,         SWF_MuteSample_Rev
Sonic1Timpani:  dcz80   SWF_S1_Timpani,         SWF_S1_Timpani_Rev,     SWF_MuteSample,         SWF_MuteSample_Rev
```

The first two pointers art the "Start sample" pointers, one for forwards, one for reverse playback, the second two pointers are the "Loop sample" pointers, again, one for forwards, one for reverse. Notice in these instances the "Loop sample" pointers point to our mute sample, thus, ensuring no looping.

Now, Dual PCM has two virtual channels, PCM1 and PCM2, the details of these virtual channels will be explained in the next chapter, let's get a sample played on one of these channels first.

To write a sample to PCM1:

- The pointers address is at $A00C69 (PCM1_Sample).
- The request flag is at $A0064E (PCM1_NewRET).

To write a sample to PCM2:

- The pointers address is at $A00C75 (PCM2_Sample).
- The request flag is at $A00651 (PCM2_NewRET).

The process should be simple, stop the Z80, write the sample pointers to the PCM channel, then set the request flag, and finally start the Z80. Below is an example of a sample being written to PCM1:

```
		lea	(Sonic1Kick).l,a0			; load sample pointers
		lea	($A00C69).l,a1				; load PCM1 pointers
		lea	($A0064E).l,a2				; load PCM1 request switch
		move.w	#$0100,($A11100).l			; request Z80 stop (ON)
		btst.b	#$00,($A11100).l			; has the Z80 stopped yet?
		bne.s	*-$08					; if not, branch
		move.b	(a0)+,(a1)+				; set address of sample
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; set address of reverse sample
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; set address of loop sample
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; set address of loop reverse sample
		move.b	(a0)+,(a1)+				; ''
		move.b	(a0)+,(a1)+				; ''
		move.b	#%11011010,(a2)				; set request
		move.w	#$0000,($A11100).l			; request Z80 stop (OFF)
```

PCM2's method will be similar, though using the addresses $A00C75 and $A00651. Notice the request flag has been set in this example, this is necessary to trigger the driver to read the "Start sample" first.

You must give Dual PCM time to set itself up first though, Dual PCM has to initialise itself once you've loaded the driver into the Z80 and started it. You cannot make requests immediately, Dual PCM will overwrite your requests with the mute sample while it's initialising itself. Once Dual PCM is ready, then you can make your requests as and when you see fit. But you must let Dual PCM do its thing when it first starts.

Once the sample address is written, it will remain inside the driver until you replace it with a different sample address. The sample that is already written can be triggered to play again by setting **only the request flag**:

```
		lea	($A0064E).l,a2				; load PCM1 request switch
		move.w	#$0100,($A11100).l			; request Z80 stop (ON)
		btst.b	#$00,($A11100).l			; has the Z80 stopped yet?
		bne.s	*-$08					; if not, branch
		move.b	#%11011010,(a2)				; set request
		move.w	#$0000,($A11100).l			; request Z80 stop (OFF)
```

If you wish to play a different sample, a new sample address needs to be written as per the previous example.

You do not need to worry about banking.

Dual PCM has an auto-banking mechanism, meaning once it has reached the end of an $8000 byte bank, it will automatically switch to the next bank to continue playing the sample. This means you don't need to break your sample up into little pieces, you don't necessarily need to align the samples in your ROM, you really do not need to worry about any aspect of the bank sections at all. My advice is to simply forget that they are there, and treat the sample includes as you would if you were including any normal data into the 68k ROM.

The sample can exist anywhere in the first 8MB range of the 68k's memory. Most ROMs for the system are 4MB in size, though if you do expand the ROM size further and provide a cartridge to allow /DTAK, then Dual PCM will at least be able to access sample data up to the 8MB mark.

Looping sample example

Now that you know how to play a sample, we will look at looping a sample.

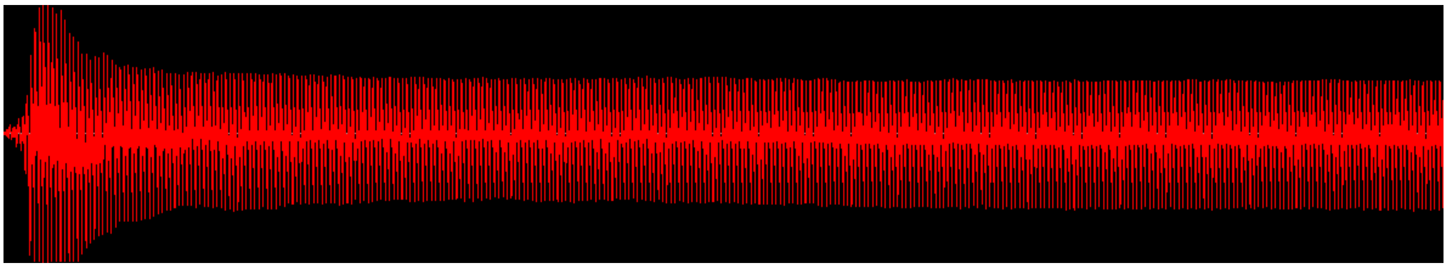
Please insert the following equates somewhere in your source, they will be incredibly useful for you to control the exact position of a sample to loop to:

```
Sec     =       14000   ; Hz per second
Mil     =       1000    ; milliseconds per second
```

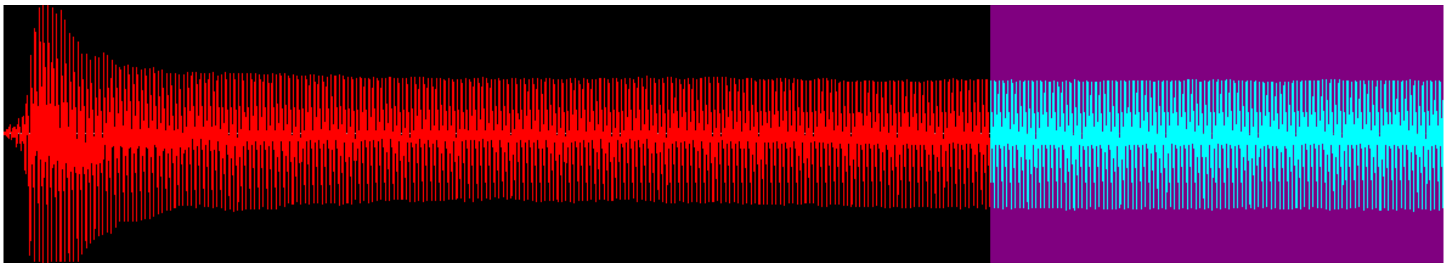
I have a Saxophone ".wav" sample we can use for this demonstration, you can download it here:

http://mrjester.hapisan.com/05_DualPCM/Saxophone.wav

This is a recording of a Saxophone being played at note C4, below is a an audio graph of the Saxophone:



As you can see, this sample has an "attack" (the beginning bulky part that shoots up suddenly and then drops back down), followed by a "sustain" (the long block in the middle that goes on and on). So what we would like to do for this sample, is have the attack play, and then the sustain part we can loop over and over again (The purple section):



So first, we'll convert and include the sample into the 68k ROM (see the previous two chapters for more details):

```
SWF_Saxophone:          incbin  "Samples\incswf\Saxophone.swf"
SWF_Saxophone_Rev:      dcb.b   $18*$10,$00
```

Now the actual pointers for these will be:

```
Saxophone:      dcz80   SWF_Saxophone,  SWF_Saxophone_Rev,      SWF_Saxophone+((1010*Sec)/Mil),  SWF_Saxophone_Rev
```

This is the common "Start forwards, Start reverse, Loop forwards, Loop reverse", but notice that the "Loop forwards" pointer is somewhat different. The equates you included earlier are being used in this equation +((1010*Sec)/Mil), this is calculated to add 1.010 seconds to the start location of the sample (this is approximately where the purple section starts).

Now when the Saxophone sample is played, you'll hear the attack part, but then it'll sustain pretty much forever, never-endingly playing the sustain part. This will save you a lot of ROM space as you can loop the sustain parts continuously. I believe a lot of SNES games have a similar method for saving on ROM space.

DMA interrupt loss prevention

Dual PCM has a streaming/buffer mechanism for preventing quality loss by interruptions.

DMA Transfer

The VDP has a DMA controller, the principle is to give the VDP direct access to 68k memory, and then make transfers from 68k memory to VDP Video RAM. For the standard manual method, the 68k does not have direct access to VDP Video RAM, the transfers are done one word at a time through the FIFO, and it requires the VDP to empty the FIFO as the 68k fills it up, thus, the 68k often has to wait. For this reason, a DMA transfer is faster and often preferred.

The issue

The issue is, in order for the VDP to access 68k memory, the 68k has to be temporarily halted until the transfer is complete, the 68k is unable to access its own memory and thus unable to proceed with its instructions.

Dual PCM has the Z80 accessing sample data through the window into 68k ROM, the bytes are collected through a process known as "cycle stealing", that is to say the byte is being collected through the 68k's clock cycle. But, since the 68k is halted by the VDP for the DMA transfer, and the VDP is accessing 68k memory, the Z80 is unable obtain the byte. The Z80 therefore ends up locked, waiting for a response from the 68k, the 68k is halted and waiting for the VDP to finish its transfer and resume the 68k.

Since the Z80 is now technically halted, it is no longer transferring the PCM data smoothly and on time, thus, the audio becomes interrupted and choppy. This (along with several unnecessary Z80 stops) is the main cause as to why so many games have poor sample quality.

Streaming/buffer

Normal drivers will read the PCM from the window and directly play it out to the DAC port, this is direct playback. Dual PCM however will read the PCM from the window but store it inside a small buffer space, it will then copy from the buffer to the DAC port later.

The reading however is much faster than the playback itself, this is to allow Dual PCM to fill up the buffer while it's playing, for every $10 bytes played, $18 bytes are read, once the buffer is full, Dual PCM will then read PCM data at the same speed as it is playing back, to keep the buffer 100% full.

DMA transfers usually occur during V-blank, so once V-blank occurs, the VDP should trigger the Z80 to interrupt into its V-blank routine, once in this routine, Dual PCM will continue the playback, but it will avoid any reading from the window to prevent itself from being halted, since the buffer holds audio data ahead of time, Dual PCM can continue playing what it's streamed already, and hopefully, the DMA transfers will finish before the buffer is emptied, once finished, it will return back to the main routine and start streaming and filling up the buffer once more for the next V-blank.

Because of this buffer, audio playback will always be approximately 1 frame behind the request. But where other drivers will produce choppy playback, Dual PCM will not.

Virtual channels and V-blank

Dual PCM has two virtual channels named PCM1 and PCM2, these virtual channels act as individual and independent playback mechanisms, they can read two different samples independently of one another, even if the samples are located in different bank/window locations, they can read the samples at different speeds from each other to simulate different pitches as well as reverse playback, they can reduce the volume of the samples individually, and they can loop back at different positions if needed.

The two virtual channels have their two sample bytes fused together and fed into the YM2612's single DAC 8-bit channel, this simulates the illusion of two channels, though realistically only a single channel of the YM2612 is being used. The YM2612's DAC channel is a hardware substitution for FM6, thus, FM6 must be sacrificed in order for Dual PCM to use the DAC channel. Since the 8-bits of the channel correspond to a monophonic output, stereo is not plausible, likewise, as the two virtual channels are sharing a single YM2612 channel, the channels cannot independently control the panning.

When to write to them

You can write to them technically whenever you want, though be warned, writing the sample address more than once per frame may cause Dual PCM to misread the address, it could be halfway reading the old address while you're writing the new address.

These channels have a "request flag" which must be set, this will prevent the above from being an issue, the flag is checked first, if it is set, Dual PCM will **de-set the flag first** before attempting to read the sample address, that way, if you make the request just as it de-sets the flag, the new address is already in by then so it won't matter, if Dual PCM is mid-way reading the address, the request has been set again so Dual PCM will re-read the address on the next pass to fix itself.

This will mean for a brief moment however (for $18 bytes of the sample), the audio may be wrong, this is quite insignificant and not likely notice-able.

If you wish to avoid it anyway, then it is best to write the information to the virtual channels once per frame, the recommended time to do this is during V-blank (explained below). You can write to either channel at different times, this is strictly regarding writing to the same aspect of the same channel twice in a frame.

V-blank

Games with 68k tracker drivers tend to write to the sound chips once per frame, and usually after V-blank has ran and the DMA transfers have done. This is also the recommended time to write to Dual PCM, be it for the sample or for YM2612 queue data. We will touch up on this later.

Controls in sync

When writing a sample address, if the sample is required to play at say a different pitch or different volume, while you can write the sample address, the pitch, the volume, etc at different times, it is recommended to try and write them at near enough the same time. Dual PCM isn't perfect, so the pitch change or volume change can often be out of sync with when the new sample was requested.

Pitch control

Dual PCM can control the pitch of the samples with precision.

Most drivers use the "delay" method to control the pitch, meaning they deliberately slow down the speed at which they give the DAC the bytes. Because there are two PCM channels and the pitches have to be controlled independently, a different mechanism is in place.

General explanation

The pitch is controlled through a word frequency. The format of this word is $QQFF, where QQ is the quotient, and FF is the fraction.

Normal pitch is $0100 (basically 1.00). If you increase this frequency you will increase the speed the channel will play the samples at, this affects the pitch. Setting the pitch to $0200 will play the samples at twice the speed and the pitch will be higher, you can have specific and precise pitches in-between, for example $0180 will be halfway between the two.

The pitch can go as high as $1000, this is quite a high pitch. Do not exceed this amount.

The pitch can also be slowed down by reducing the frequency closer to $0000, for example $0080 will now play the samples at half the pitch, $003E is even slower, $0014 even slower, and $0001 is the absolute slowest. Setting the pitch to $0000 will cause the channel to remain in a paused state,

For **reversed playback**, you can set the pitch into negative ($FFFF and below), for example, negative $0100 is $FF00, and this will cause the sample to play at normal pitch but be playing backwards instead of forwards. $FF80 will be playing in reverse but at half the pitch (since it's negative $0080), so the pitch will be lowered by half.

The pitch can go as high for reverse playback as it can for forwards playback, the highest being $F000 (negative $1000). Do not drop below $F000.

Thorough explanation

The pitch control works through stretching (repeating) and squashing (skipping).

The samples are always played by Dual PCM at a frequency between 13,750Hz and 14,000Hz. When the samples are loaded, the pitch controls how often to re-read the same byte again, or how often to skip over bytes.

For example:

```
80 84 88 8C 8E 8F 8D 85 80 7C 72 70 6E 73 78 7F
```

With a pitch of $0200 will skip every second byte, this means the sample stored inside Dual PCM only has half of the sample, it's as if the sample has been "re-sampled":

```
80 88 8E 8D 80 72 ... etc
```

A pitch of $0080 will cause Dual PCM to read the same byte twice before advancing, thus stretching out the sample:

```
80 80 84 84 88 88 8C 8C 8E ... etc
```

This ensures that very precise and complex frequencies can be processed thanks to the fractions. It also allows both channels to be independently controlled, since the playback speed is constant, one channel can have its pitch altered without affecting the other channel.

End Markers

Going back to end markers, due to the skipping method required to increase the pitch of a sample, the end marker can be accidentally skipped over, this is the reason why the end markers are a block of $00's instead of a single byte $00.

Using our example, but with an end marker of only one byte:

```
80 84 88 8C 8E 00 80 84 88 8C ... etc
```

If we pretend the pitch is $0200, then the bytes will be read in the following order:

```
80 88 8E 80 88 ... etc
```

As you can see, the byte 00 was entirely skipped, and the driver continued playing onwards, potentially playing the next sample it is not supposed to play. Another reason is that Dual PCM does not check every byte it reads, it just checks one byte out of $18 to save on CPU time.

The blocks of $00's need to be large enough to account for any pitch, and any position Dual PCM could be reading at. Since the pitch can go as high as $1000, the quotient $10 is the number of bytes it can skip to. Dual PCM also reads $18 bytes per loop, of which only one is checked.

So this is $10 multiplied by $18, which is a total of $180 bytes of $00 end markers in a block.

This is why all end markers you have seen in examples above have had the following equation:

```
		dcb.b	$18*$10,$00
```

This creates the $180 block of $00's.

As mentioned before, thanks to reverse playback being an option, these end marker blocks must exist before the sample as well as after.

Setting the pitch

To write a pitch to one of the virtual channels the pitch's quotient and fraction must be written to separate addresses.

For PCM1:

- Pitch quotient to be written to $A005DD (PCM1_PitchHigh+1).
- Pitch fraction to be written to $A005E8 (PCM1_PitchLow+1).
- Request flag to be set $A005D2 (PCM1_ChangePitch).

For PCM2:

- Pitch quotient to be written to $A0060E (PCM2_PitchHigh+1).
- Pitch fraction to be written to $A00619 (PCM2_PitchLow+1).
- Request flag to be set $A00603 (PCM2_ChangePitch).

Here is an example of setting PCM1's pitch, assuming d0 has the pitch quotient $QQ and d1 has the pitch fraction $FF:

```
move.w  #$0100,($A11100).l             ; request Z80 stop (ON)
btst.b  #$00,($A11100).l               ; has the Z80 stopped yet?
bne.s   *-$08                          ; if not, branch
move.b  d0,($A005DD).l                 ; set pitch quotient
move.b  d1,($A005E8).l                 ; set pitch fraction
move.b  #%11010010,($A005D2).l         ; set request
move.w  #$0000,($A11100).l             ; request Z80 stop (OFF)
```

By default, the pitch is $0100, normal forwards.

You can change this pitch as often as you want, it is recommended only to change it once per frame, but you can constantly change the pitch in real-time while the sample is currently playing.

This means you can perform pitch bends and vibrato affects/LFO/Modulation onto the sample. This also includes reverse playback too, you can midway force the sample into reverse and then forwards again. The precision of pitch control also means slight detuning effects can be applied.

Non-percussion samples

The driver is designed to play back more than just drum samples, it allows you the ability to play-back virtually any sound you wish within a good margin of reason.

If you can get a hold of a recorded sample of an instrument, say a Saxophone. Then try to get the recording in note C4, this particular note is best used with the following frequency values supplied below:

```
;       dc.w      C      C#     D      Eb     E      F      F#     G      G#     A      Bb     B

        dc.w    $0010,$0011,$0012,$0013,$0014,$0015,$0017,$0018,$0019,$001B,$001D,$001E   ; Octave 0
        dc.w    $0020,$0022,$0024,$0026,$0028,$002B,$002D,$0030,$0033,$0036,$0039,$003C   ; Octave 1
        dc.w    $0040,$0044,$0048,$004C,$0051,$0055,$005B,$0060,$0066,$006C,$0072,$0079   ; Octave 2
        dc.w    $0080,$0088,$0090,$0098,$00A1,$00AB,$00B5,$00C0,$00CB,$00D7,$00E4,$00F2   ; Octave 3
        dc.w    $0100,$010F,$011F,$0130,$0143,$0156,$016A,$0180,$0196,$01AF,$01C8,$01E3   ; Octave 4
        dc.w    $0200,$021E,$023F,$0261,$0285,$02AB,$02D4,$02FF,$032D,$035D,$0390,$03C7   ; Octave 5
        dc.w    $0400,$043D,$047D,$04C2,$050A,$0557,$05A8,$05FE,$0659,$06BA,$0721,$078D   ; Octave 6
        dc.w    $0800,$087A,$08FB,$0983,$0A14,$0AAE,$0B50,$0BFD,$0CB3,$0D74,$0E41,$0F1A   ; Octave 7
```

These values have been created based on an exponential equation supplied by Sik, though feel free to modify them if you require something different.

If you want to play your Saxophone sample at say... note F#5, you would use frequency $02D4.
If you want to play your Saxophone sample at say... note B1, you would use frequency $003C.

For reverse playback, the same frequencies can be used, just make sure you negate them before you give them to Dual PCM, frequency $FD2C (negative $02D4) will play the Saxophone sample in reverse at note F#5. Note that reverse playing back the Saxophone sample will cause it to reverse the "attack" part of the sample, this reverse playback is more for non-looping samples, like perhaps vocals, percussion or other less loopy types.

Volume control

This part of the driver is more complicated, so understand the below is a description of the volume control and how it works, for details of using volume control, please see the next chapter. I do recommend at least glossing over this page for your information.

Many methods were tried and practiced with Dual PCM to find the right volume control method, after experiments and discussions with my testers, we have come to the conclusion that there is no single best way to go about this. We have three main methods all of which are similar in practice, some have benefits in one way but downfalls in another, of which other methods will have the reverse of.

The main idea is the same throughout all three methods. Dual PCM has a volume look up table (VLUT), one for each virtual channel. This is a block of $100 bytes, these bytes will swap in substitution with the actual samples bytes to simulate a volume difference. Effectively speaking, the samples are being re-sampled with a different volume in real-time right before they are played.

These two $100 byte tables are actually "buffers", so you can swap out the volume table for a different table which contains completely different volume values. But it's the swapping out that's different between the methods.

<u>Method 1 - Dual PCM calculates the table</u>

Dual PCM has the ability to real-time calculate the volume table based on the volume you set, it will manually create and write all $100 bytes of the table.

- Pros - It's entirely and internally within the Z80, no 68k control is required other than just giving it the volume byte.
- Cons - It's very slow, even with heavy optimisations carefully thought through. This can lead to the samples popping every time you change the volume.

The routine for this method is **comment out but still available in the source code**, this means though that in order to use this method, you'll have to use the source code instead of the pre-assembled version, the pre-assembled version was not assembled using this method.

You will be on your own to re-implement it, but the code can be found at "SwitchVolume:" In theory, you should be able to comment this back in (making sure you erase the other method's code after it, or else there might not be enough Z80 RAM space), and it should function as intended, but this original method was replaced with one of the other methods quite a long time ago, so I am unaware of whether or not anything will break. As I mentioned, you're on your own if you use this method.

The routine writes linear volume values, but it can be easily modified to do this in a logarithmic fashion, I suggest using a swap table to swap the linear value with a logarithmic version before letting the routine calculate the table.

<u>Method 2 - 68k calculates the table</u>

This method, instead of changing the volume value of Dual PCM, you ignore that, and simple write the new volume table directly to Dual PCM's volume table buffers (VLUT), PCM1's volume table is at $A00A00 (PCM_Volume1), and PCM2's volume table is at $A00B00 (PCM_Volume1). The first byte **must be 00** though, due to the end markers, even if Dual PCM detects a 00 end marker, it is possible for it to have the potential to accidentally process it.

You may get the 68k to calculate this table in real time and write it to the table, it is recommended to calculate the table into a 68k RAM buffer before copying that buffer to the Z80 volume table just to reduce the amount of time the Z80 may be halted for. The quicker you can write the table, the less of a chance there is of the sample quality dropping.

- Pros - Saves a lot of Z80 time, thus reducing the possibility of popping.
- Cons - Requires you to waste 68k CPU time, the popping might also not be entirely gone as the Z80 is still halted to load the new table.

This method can be used on either the pre-assembled or the source version, just do not change the volume byte Dual PCM's virtual channels.

Method 3 - Z80 loads new table through window

This method is the method currently setup for the pre-assembled version, and the source version.

All tables are pre-generated in a bank which is stored on the 68k ROM. You change the volume byte, and the Z80 will fetch the new volume table from 68k ROM and copy it directly to the volume table buffer (VLUT).

This method may be the fastest, but also the most expensive in terms of ROM size, but it does allow you to change the volume without popping or quality loss.

- Pros - No popping or quality loss (so long as the volume is not changed too frequently), no 68k time required.
- Cons - Costs 68k ROM $8000 bytes for the pre-generated volume tables.

This method has been chosen as it helps to *almost* eliminate any popping or quality loss when changing the volume.

The mechanism involved for loading the new table tries to copy the table over whilst still flushing PCM audio, which is why the quality isn't damaged. The downfall is, the volume table can only be updated during the loop that's responsible for filling up the streaming/buffers.

If Dual PCM is not filling up those buffers, then eventually the playback mechanism is going to run out of bytes to play. Because this is only done once, it seems just barely able to change the table and still fill up the buffer just enough to cover most DMA transfers. However, if you do not let Dual PCM fill the buffer up before changing the volume again, you will eventually empty the buffer fully. This will cause quality loss.

So it's recommended to spread out your volume changes out across several frames, for example, if you want to do volume fading, we recommend instead of changing the volume by 1 every frame, you instead change the volume by 4, but every 4 frames. The 3 frames no volume is changing, Dual PCM will be able to refill it's buffers.

This is not mentioned as a "con" for this method, because the other methods have quality issues regardless of how frequently you change the table.

Setting the volume

Here we will setup the volume bank first, and then we will change the volume of a sample. This will describe using "Method 3" of the three methods explained in the previous chapter, if you are not familiar, then please go back and have a look.

Method 3 involves accessing a bank of volume values, I have provided a volume bank for you in the following link:

http://mrjester.hapisan.com/05_DualPCM/Volumes.bin

This table is designed with a type of "logarithm", the FM of the YM2612 is logarithmic in terms of volume so this table should ensure the volume fading "sounds" linear and in-line with FM.

Installing the volume bank

Include the bank anywhere in the 68k ROM, but you must ensure this is aligned to the next $8000 position in ROM so the Z80 can access it easily (the Z80 has no auto-banking for the volume bank, only the samples):

```
                        align   $8000
PCM_Volumes:            incbin  "Volumes.bin"
                        dcb.b   $18*$10,$00

SWF_MuteSample:         dcb.b   $8000-(($18*$10)*2),$80
SWF_MuteSample_Rev:     dcb.b   $18*$10,$00

SWF_S1_Kick:            incbin  "Samples\incswf\Sonic 1 Kick.swf"
SWF_S1_Kick_Rev:        dcb.b   $18*$10,$00

SWF_S1_Snare:           incbin  "Samples\incswf\Sonic 1 Snare.swf"
SWF_S1_Snare_Rev:       dcb.b   $18*$10,$00

SWF_S1_Timpani:         incbin  "Samples\incswf\Sonic 1 Timpani.swf"
SWF_S1_Timpani_Rev:     dcb.b   $18*$10,$00
```

As you can see (highlighted yellow), I have included mine amongst the sample data, more specifically I have included just before the "mute sample" we created, since I've aligned that to $8000 and the bank itself is $8000, it makes it pretty much the perfect place to put it.

Next up, is to tell Dual PCM exactly where the volume bank is, this can be done when you first load the sound driver, writing to the address $A01A38 (SV_VolumeBank):

```
        lea     (Z80ROM).l,a0                   ; load Z80 ROM data
        lea     ($A00000).l,a1                  ; load Z80 RAM space address
        move.w  #(Z80ROM_End-Z80ROM)-$01,d1     ; set repeat times
        move.w  #$0100,($A11100).l              ; request Z80 stop (ON)
        move.w  #$0100,($A11200).l              ; request Z80 reset (OFF)
        btst.b  #$00,($A11100).l                ; has the Z80 stopped yet?
        bne.s   *-$08                           ; if not, branch

.LoadZ80:
        move.b  (a0)+,(a1)+                     ; copy Dual PCM to Z80 RAM
        dbf     d1,.LoadZ80                     ; repeat til done
        lea     (MuteSample).l,a0               ; load mute sample address
        lea     ($A00C62).l,a1                  ; load Z80 RAM space where the pointer is to be stored
        move.b  (a0)+,(a1)+                     ; copy pointer over into Z80
        move.b  (a0)+,(a1)+                     ; ''
        move.b  (a0)+,(a1)+                     ; ''
        move.b  (a0)+,(a1)+                     ; copy "reverse" pointer over into Z80
        move.b  (a0)+,(a1)+                     ; ''
        move.b  (a0)+,(a1)+                     ; ''
        lea     ($A01A38).l,a1                  ; load volume bank address write routine
        move.b  #$74|((PCM_Volumes>>$0F)&1),(a1)+; write "ld  (hl),?" instructions
        move.b  #$74|((PCM_Volumes>>$10)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$11)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$12)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$13)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$14)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$15)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$16)&1),(a1)+; ''
        move.b  #$74|((PCM_Volumes>>$17)&1),(a1)+; ''
        move.w  #$0000,($A11200).l              ; request Z80 reset (ON)
```

```
		moveq	#$7F,d1				; set repeat times
		dbf	d1,*				; no way of checking for reset, so a manual delay is necessary
		move.w	#$0000,($A11100).l		; request Z80 stop (OFF)
		move.w	#$0100,($A11200).l		; request Z80 reset (OFF)
```

The highlighted yellow code has been added to our loading routine, the equation is to write a series of "ld (hl),?" instructions into the Z80 which act as a bank switch setting mechanism, the equation will correctly set "ld (hl),?" to the correct register required for setting the bank so it points to our volume bank.

The volume bank is now installed and we can start making volume changes.

Changing the volume

To change the volume for PCM1:

- Write the volume byte to $A00642 (PCM1_Volume+1).
- Set the request at $A00634 (PCM_ChangeVolume).

To change the volume for PCM2:

- Write the volume byte to $A00647 (PCM2_Volume+1).
- Set the request at $A00634 (PCM_ChangeVolume).

Notice that they both share the same request flag. This is why if you intend on changing the volume for both channels on the same frame, it is recommended to change them at the same time. However, there has been no major harm with changing the volume for each channel at different times on the same frame, at least, no notice-able issues I've experienced. If you do experience audio quality issues, then this may be why, so keep it as a precaution.

The volume byte that's being written to the channel can be from $00 to $7F.

Now, assuming you have not modified the volume bank in any way, $00 should be the loudest volume (normal volume), and $7F should be the quietest volume. Writing $80 will mute the channel, forcing Dual PCM to flood the volume table with mute values, so the audio playing cannot be heard, this does not mean the channel isn't playing though.

Writing other values from $81 to $FF is not recommended, and should be known to be "undefined", I suspect what might happen is that volume will get louder again as you reach $FF, but the wave positions will be in reverse, or it's possible the values being read are from the Z80 ROM portion, and can result in a crash when reading from the $7F00 region, but don't quote me on that one.

Summary; $00 = Loudest, $7F = Quietest, $80 = Mute.

Writing a volume value of $40 for example will result in the volume sounding half loud, here is an example of writing a volume value to PCM1 (assuming you have the volume in d0):

```
		move.w	#$0100,($A11100).l		; request Z80 stop (ON)
		btst.b	#$00,($A11100).l		; has the Z80 stopped yet?
		bne.s	*-$08				; if not, branch
		move.b	d0,($A00642).l			; set volume
		move.b	#%11010010,($A00634).l		; set request
		move.w	#$0000,($A11100).l		; request Z80 stop (OFF)
```

YM2612 queue

Dual PCM is a PCM playback driver, ordinarily the FM and PSG aspects are meant to be ignored and handled by the 68k. The problem is, the YM2612 cannot be accessed by the 68k directly, only the Z80 can access it.

Often the 68k will use the Z80's memory space to access the YM2612, but in order to do that, it must stop the Z80. As a PCM playback driver, this is not very good for Dual PCM, for the longer Dual PCM is stopped for, the more damage and chopping this will do to the sample playback.

When writing data to the YM2612, a register address must be set, then the data to be written must be set. There must be a delay between the setting of the address and the writing of the data as the YM2612 may be busy and not gotten around to updating the address on time. Ordinarily you would check to see if the YM2612's busy bit is set before writing anything to it. This wait costs a huge portion of time for both the 68k and the Z80.

However, as long as the amount of data being written isn't too frequent, you can get away with the 68k accessing the YM2612 without damaging the quality much.

The YM2612 queue is optional, you can choose not to use it and that is 100% fine. There is only one rule to remember, **you must set the YM2612 address back to $2A (the DAC port)**, Dual PCM makes the assumption that the YM2612 is already set there, so after you write your FM data, set the address back to $2A, and then start the Z80 again. Also keep in mind that the PCM playback is a frame behind, so it's best to delay your FM writing for a single frame, I suggest delaying the FM and PSG channels for a single frame on start-up.

Explaining the queue

This is a large buffer in Z80 RAM, a list if you will, this list contains the YM2612 register/address, the port to write to, and the data to write to it. The idea is, you stop the Z80, write the register/data to this list (NOT the YM2612), and then start the Z80 again. Since the 68k doesn't need to wait for the YM2612 to finish being busy, it can write FM information much more quickly, this is also good for Dual PCM since it does not need to be stopped for very long.

Dual PCM will transfer the YM2612 data into the YM2612 itself in-between PCM writes, thus allowing the data to get to the YM2612 without stopping PCM playback.

So that's the queue.

Using the queue

There are technically two queues, one at $A01000 (YM_Buffer1) and one at $A01400 (YM_Buffer2). This is a double buffer mechanism, while you're writing the data to one queue (using the 68k), the Z80 will be reading from the other queue. The next frame, the positions will switch, the Z80 will be reading from the queue you just wrote to, while you write to the queue the Z80 previously read from.

Which buffer you use must be defined by the switch, the switch is located at $A00C68 (YM_Buffer).

- If the byte is $00, the Z80 is reading $A01000 (YM_Buffer1), the 68k is writing $A01400 (YM_Buffer2).
- If the byte is $FF, the Z80 is reading $A01400 (YM_Buffer2), the 68k is writing $A01000 (YM_Buffer1).

You must write to the correct list, if you do not, you may overwrite your own requests from the last frame.

Dual PCM will be the one responsible for switching the buffer, though, it is possible for Dual PCM to be too busy and not fully finish processing a list. In which case, the switch will not be changed, if Dual PCM has not switched the buffer, then it is strongly advised not to write to **any** of the queues, and simply postpone your tracker from doing anything until the next frame. This generally only happens if the list is quite large and Dual PCM cannot process all of it quickly enough, so I suspect you'll only receive this issue when you initialise the start of a music track.

Here is some example code you can use as a reference:

```
		lea	($A00C68).l,a0		; load buffer ID address
		move.w	#$0100,($A11100).l	; request Z80 stop (ON)
		btst.b	#$00,($A11100).l	; has the Z80 stopped yet?
		bne.s	*-$08			; if not, branch
		move.b	(a0),d0			; load buffer ID
		move.w	#$0000,($A11100).l	; request Z80 stop (OFF)
		cmp.b	(LastBuffer).w,d0	; did the 68k write to this buffer last frame?
		bne.s	.Continue		; if not, branch
		rts				; return (postpone)

	.Continue:
		move.b	d0,(LastBuffer).w	; store last buffer ID for next frame
		lea	($A01000).l,a4		; set the cue address to buffer 1
		tst.b	d0			; is the Z80 accessing buffer 1?
		bne.s	.WriteBuffer1		; if not, branch
		lea	($A01400).l,a4		; set the cue address to buffer 2

	.WriteBuffer1:

		; Continue sound driver here...
```

As you can see, this routine will load the correct queue address into a4 depending on the buffer flag the Z80 has currently set. Before this though, you can see it stores the buffer flag and then checks on the next frame to see if the flag has changed, if the flag has not changed then the Z80 is not ready and has not cleared out the queue fully yet, so there is no choice but to return and do nothing sound related.

"LastBuffer" represents a byte of 68k RAM.

Writing to the queue

Now that the correct queue is loaded into a4, we can start writing data to it.

Every three bytes in the queue is a single entry. Let's pretend you want to set the address to $B4, and write the value $10 to it in the YM2612. We'll pretend we're writing to the second port ($A04002 and $A04003).

Normally it would look something like this (assuming d0 has the register/address ($B4) and d1 has the data ($10)):

```
		move.w	#$0100,($A11100).l	; request Z80 stop (ON)
		btst.b	#$00,($A11100).l	; has the Z80 stopped yet?
		bne.s	*-$08			; if not, branch

	.WaitYM1:
		move.b	($A04000).l,d2		; is the YM2612 busy?
		bmi.s	.WaitYM1		; if so, branch and wait
		move.b	d0,($A04002).l		; set the address
		nop				; give the YM2612 time to...
		nop				; ...signify it's busy

	.WaitYM2:
		move.b	($A04000).l,d2		; is the YM2612 busy?
		bmi.s	.WaitYM1		; if so, branch and wait
		move.b	d1,($A04003).l		; write the data to the addess
		move.w	#$0000,($A11100).l	; request Z80 stop (OFF)
```

Using the queue would instead look like this:

```
		move.w	#$0100,($A11100).l	; request Z80 stop (ON)
		btst.b	#$00,($A11100).l	; has the Z80 stopped yet?
		bne.s	*-$08			; if not, branch
		move.b	#$02,(a4)+		; write YM2612 port address
		move.b	d1,(a4)+		; write YM2612 data
		move.b	d0,(a4)+		; write YM2612 address
		st.b	(a4)			; set end of list marker
		move.w	#$0000,($A11100).l	; request Z80 stop (OFF)
```

Already you should be able to see the significant difference in size and speed for the 68k alone.

A few things to note about this code, the port address has been written first, this can be either 00 or 02 depending on which port you want to write to, if this is FF, this will signify to Dual PCM that it's the end of the queue (this is the reason why there is an "st.b" instruction setting FF on the next slot ahead of time).

If you wish to write a stream of values to the queue, it is recommend that you set everything up that you need, stop the Z80, write each slot one by one, then perform the "st.b" last, and then start the Z80 again. If you were to stop and start the Z80 for every individual YM2612 write, this will not only cost Dual PCM time, but it will also cost the 68k time too, here is a basic (and crude) example:

```
		move.w	#$0100,($A11100).l	; request Z80 stop (ON)
		btst.b	#$00,($A11100).l	; has the Z80 stopped yet?
		bne.s	*-$08			; if not, branch

	; FM 1 panning left
		move.b	#$00,(a4)+		; write YM2612 port address
		move.b	#$B4,(a4)+		; write YM2612 data
		move.b	#$80,(a4)+		; write YM2612 address
	; FM 2 panning left
		move.b	#$00,(a4)+		; write YM2612 port address
		move.b	#$B5,(a4)+		; write YM2612 data
		move.b	#$80,(a4)+		; write YM2612 address
	; FM 3 panning left
		move.b	#$00,(a4)+		; write YM2612 port address
		move.b	#$B6,(a4)+		; write YM2612 data
		move.b	#$80,(a4)+		; write YM2612 address
	; FM 4 panning right
		move.b	#$02,(a4)+		; write YM2612 port address
		move.b	#$B4,(a4)+		; write YM2612 data
		move.b	#$40,(a4)+		; write YM2612 address
	; FM 5 panning right
		move.b	#$02,(a4)+		; write YM2612 port address
		move.b	#$B5,(a4)+		; write YM2612 data
		move.b	#$40,(a4)+		; write YM2612 address
	; FM 6/DAC panning right
		move.b	#$02,(a4)+		; write YM2612 port address
		move.b	#$B6,(a4)+		; write YM2612 data
		move.b	#$40,(a4)+		; write YM2612 address

		st.b	(a4)			; set end of list marker
		move.w	#$0000,($A11100).l	; request Z80 stop (OFF)
```

No doubt you will have your own optimal ways to do this. Just remember, keep the Z80 halting as brief as possible, as always make sure the queue ends with an FF.

You can postpone the FF marker writing until the whole driver is finished writing what it needs to write. **As long as you don't forget to do it**, Dual PCM will not stop processing the YM2612 data until it reaches an FF.

The data will not reach the YM2612 until the next frame, but since the PCM playback is approximately delayed by a frame, the FM and PCM *should* be in sync, so you won't need to delay the FM channels, they are automatically delayed by these queues, thus, only the PSG channels (if you have any) will need a frame delay.

Additional control

There are other effects you can perform using Dual PCM, these are unintended or unplanned effects and are simply by-products, but they might be useful to you if you can use them effectively. What follows are also a few suggestions, feel free to ignore them.

Suggestions for use

Here are a few suggestions on how you might use Dual PCM.

1. Use one PCM channel for percussion in music, and use the other PCM channel for SFX samples.
2. Use both PCM channels for music, one for percussion, one for instrumental, and maybe allow SFX samples to interrupt the PCM percussion channel.
3. Use one PCM channel for looping large background music samples, and use the other for PCM SFX samples (this can be costly, so it depends on how much ROM space you have).
4. Use the PCM channels to play back majority of the music in the form of complex samples, and use the YM2612 to play drums (using multiply FM channels together you should be able to get realistic sounding percussion).

There are a few more suggestions I could through, but many are very specific, and some just outright silly, but do-able.

Accessing sample 8MB+

Due to the nature of the auto-banking mechanism, only the first 8MB of 68k memory is accessible, this should normally be enough as most ROMs stand at 4MB without /DTAK, though should you happen to want to access beyond 8MB, you will need to make some modifications to the driver, I would suggest using the MSB of the sample pointer's window address for the remaining 9th bit, but you will need to reprogram certain aspects of Dual PCM to get that 9th bit to be updated correctly. Including the auto-bank switching to include it when incrementing/decrementing through the banks.

This is entirely in your ballpark though, you're on your own.

PCM table filtering

As mentioned in the first chapter, the driver has some PCM filtering capabilities. What is it?

The volume table as explained in the volume chapters is a look up table (VLUT), while it is specifically designed for volume control, nothing stops you from putting *other* values into the table, while I haven't experimented much, it might be possible to get the samples to sound different in a variety of ways by having the table constructed a specific way.

For example, you could reduce the bit-length of samples to something lower like 4-bit, you could ensure large spikes of waves are smoothed out, I have no clear idea as to how these would sound, as I said, this is experimental. These buffer tables were designed strictly for volume control.

If you are calculating the tables yourself on the 68k side, you can perhaps generate these different effects *with* the volume control as well.

If you are using the volume bank method, you can always change the volume bank address in Dual PCM (similar to how it's set when you first initialise the driver).

Delay time manipulation effects

One thing I *did* experiment with was tampering with the delay time mechanisms throughout the driver. These needed to be fine tuned such that they would keep the PCM playback smooth regardless of routine being ran. While I am certain I have not got these 100%, I am also confident these can be abused to get varying effects. We had moderate success at creating certain underwater or weather sounding effects on the samples, as well as some mild form of "time stretching" (where the sample is played slower, but retains its pitch).

The following addresses can be manipulated, I have also supplied their default value that is expected to be there:

$A00143 (Z80_DelayYM1+1)	$07
$A0016A (Z80_DelayYM2+1)	$07
$A0018F (Z80_DelayYM3+1)	$07
$A00198 (Z80_DelayYM4+1)	$07
$A001B4 (Z80_DelayEnd1+1)	$0C
$A001CD (Z80_DelayEnd2+1)	$0C
$A000B4 (Z80_VBlank1+1)	$0C
$A0009B (Z80_VBlank2+1)	$0C
$A00688 (Z80_Flush1+1)	$0C
$A006A2 (Z80_Flush2+1)	$0C

As an example, we tried the following values to achieve a distorted water filtering effect:

$A00143 (Z80_DelayYM1+1)	$07
$A0016A (Z80_DelayYM2+1)	$07
$A0018F (Z80_DelayYM3+1)	$14
$A00198 (Z80_DelayYM4+1)	$0D
$A001B4 (Z80_DelayEnd1+1)	$0C
$A001CD (Z80_DelayEnd2+1)	$0B
$A000B4 (Z80_VBlank1+1)	$15
$A0009B (Z80_VBlank2+1)	$15
$A00688 (Z80_Flush1+1)	$15
$A006A2 (Z80_Flush2+1)	$15

Time stretching can be achieved to some moderate degree by tampering with $A00109 (YM_FlushTimer+2), normally this value is set to $02, increasing this value can simulate time stretching, however, this is not the intended use of this timer and is more of a by-product, likewise, affects the whole drive, so both PCM channels will be affected.

Feel free to experiment and see what sticks. Though do be warned, tampering with these values can cause the driver to crash if the values are too absurd.

Repeat/tracker manipulation

The "Loop pointer" of a sample in the virtual channels *could* be used to repeat samples in the form of a PCM tracker.

This will require careful timing on the 68k side however, if done correctly, you can swap out the loop pointer to point to a different sample, and the channel won't start playing that sample until it has finished playing the last sample it was given. As long as you do not set the request flag, and only write the "Loop start" address, it will never start playing until it reaches an end marker and attempts to loop.

Getting the timing right can be difficult, though you might be able to read the channel's bank and sample **current** address and work it out that way. I won't give details, so you'll have to look inside the source code and work it out yourself, though I suspect you may as well make the necessary modifications for Dual PCM to perform this function properly instead of relying on careful 68k timing.

Important notes

Here are a few important notes to remember.

Delays

Due to the streaming/buffer mechanism, PCM audio will always be played approximately at a 1 frame delay. If you are using the YM2612 queue, then the FM audio will also be delayed by 1 frame.

Please make sure you take this delay into account when operating other audio aspects, such as the PSG, (and the FM if you aren't using the YM2612 queue).

50Hz

Dual PCM has only enough time and buffer space to cover about $1800 - $1E00 bytes of DMA transfer during V-blank, this is **almost** enough to cover 60Hz system's. 50Hz has a longer V-blank period, thus much more data can be transferred during V-blank. If your ROM is intended for 50Hz systems only for the purpose of being able to utilise the larger DMA transfer amount, then Dual PCM will not be able to cover you entirely, while the PCM quality won't be as bad as other drivers, it still won't be as smooth as intended.

Volume

If you are using the bank table reading method (default behaviour for the raw binary version), then constantly changing the volume table every frame will result in quality loss, CPU time is required to load the new table, this time is taken out of the streaming mechanism, so the buffer is not being filled while the table is being copied over.

There is enough buffer time for 1 volume table update without affecting the quality, **but only 1 frame**. A second volume change will require about 4 frames of delay before Dual PCM can handle another volume change.

Changing the volume frequency won't stop the audio from playing or do any major damage, but it will cause some noticeable chopping in the audio.

If you are fading the audio, consider changing the volume in larger increments spread across several frames, i.e. instead of every frame changing the volume by 1, do every 4 frames changing the volume by 4. Volume fading is the only time I can currently think of where you would want volume changes occurring so quickly, for in-music volume changes, even simulating single channel echoes shouldn't require this frequent of a change, so you should be able to get away with it.

Doing both channels at once might require 8 frames, it's best to experiment and find the right amount that suits you.

Pitch

It is currently recommended not to set the pitch above $1000 or below $F000 (-$1000). Due to the nature of how pitch control has been constructed in code, the Z80 stack pointer register (sp) is being used as an adding device.

The pitch as mentioned before is broken into quotient and fraction, a pitch of $02C4 is actually 2.C4 where the 2 is the quotient, and C4 is the fraction. Because of the nature of the instruction used to load the PCM, the address is automatically incremented by 1, so the quotient has 1 subtracted from it to counter this, meaning Dual PCM's actual internal pitch is $01.C4.

The quotient and fraction are split into separate registers, the quotient is put inside the stack pointer while the fraction is put inside the b register. After giving Dual PCM $02C4, Dual PCM's registers contain the following:

```
sp = $0001
b = $C4
```

Now because the stack has the quotient, if an interrupt should occur, the Z80 will push its return address onto the stack, so given the pitch to be anywhere from $F000 to $1000, this means the data can be pushed anywhere between addresses $FFF0 to $000F (before $000F backwards).

$0000 - $000F = The Z80 RAM address.

$FFF0 - $FFFF = The end of the 68k access window.

The code at $0000 - $000F (the beginning of the driver) is no longer needed, so this becoming overwritten is not an issue, likewise, the stack attempting to write data to the access window into 68k ROM will have no effect.

For this reason **do not attempt to reset the Z80 at any time**, if you must reset the Z80 for whatever reason (say you wish to reset the YM2612 quicker this way), then please ensure that you reload the first $10 bytes of Dual PCM into the first $10 bytes of Z80 RAM ($A00000 - $A0000F), these instructions have a high chance of being overwritten by interrupts, thus, they need to be replenished.

Pitch is wrong?

The driver has been calibrated to match hardware, if you are using an emulator, and the emulator plays the sample at about a note higher in pitch than it should, then I can say the emulator is wrong. The original frequency table I supplied for each note was calculated based on what was heard on real hardware, and the replacement table by Sik exponentially calculated is a near match, so the table will not at fault.

The Z80 accessing the sample data through the 68k window is the main cause of these pitch discrepancies. Since reading a byte through the window relies on cycle stealing, the time it takes to read the byte is slower than reading it from its own RAM, and this slowdown is significant enough to affect the pitch. The sound driver has been adjusted correctly to account for this, so the correct pitch is always played.

The emulator you are using has to have the Z80 <-> 68k access timings correct. Many emulators do not have this correct; the Gens series are wrong, as well as Regen, all presenting the sample pitch a note higher than it should be. Kega Fusion is an odd bag for this in particular, the timings are not correct, however, it seems out of sheer dumb luck that the pitch happens to be correct, I can only assume Steve Snake put in hackish measures to account for this? You'd have to ask him about that, but Kega Fusion tends to output at near enough the correct pitch. Other emulators I cannot vouch for.

For BlastEm, as of version 0.6.2, the timing issues have been corrected, if not perfectly, then certainly good enough, I recommend this emulator.

YM2612 Access & Queue

If you do NOT use the YM2612 queue, the PCM will drop in quality and may become choppy, this does however depend on how frequent you access the YM2612 manually. Sonic 1 (of which this driver was tested on), I found that the quality was not that damaged by the YM2612 writes, the music itself seemed quite passive with YM2612 access while the music was already playing, the initialisation of the music as you request the play did cause a large amount of data to be written to the YM2612, but since this is the start of the music, PCM quality damage was hardly noticeable since it is part of the music.

SFX however, since they need to init every time they are requested, doing something as simple as collecting a ring would chop the PCM audio up.

So depending on your implementation and how quickly you can push the YM2612 data in you can achieve good PCM quality without the aid of the queue system. What's important is to keep the Z80 running, don't leave it stopped for too long.

Most importantly, **you must make sure you set the address of the YM2612 back to $2A (the DAC port)**, Dual PCM assumes the YM2612 is set to this address automatically and so does not bother setting it again to save on time. If you do not write $2A back in, Dual PCM will start writing PCM data to the last address/register you set, and you will end up with undesirable effects.

If you *do* use the YM2612 queue, PCM quality will be higher as the YM2612 data writing will be much quicker, it may be as quick as writing 3 bytes into Z80 RAM which is much faster than waiting for the YM2612 to finish being busy.

However, the queue is only a finite size, $400 bytes per queue have been reserved. See the **YM2612 queue** chapter for more details about the structure, but three bytes per entry, you have $155 YM2612 registers you can write to per frame, this was more than enough for Sonic 1's SMPS driver, though if you require more, you may have to modify the length of these queues, or not use the queue system at all.

68k tracker drivers in many games tend to be done once per frame, and often after the V-blank routine has ran, Sonic 1 for example will run its sound driver during V-blank, but after all transfers are made. This is the perfect time to be sending YM data or PCM requests. One thing to note, once you interrupt into V-blank with the 68k, please **do NOT attempt to write to the YM2612 queue immediately**, the Z80 needs time to interrupt into V-blank itself, and the 68k may be quicker. The Z80 is responsible for switching the buffer flag, so if you read the buffer flag and determine the queue to use *before* the Z80 has changed the flag, you will find you are writing to the same buffer that Dual PCM is about to read from, this queue has not been processed yet, so whatever you put in there the previous frame has not yet been passed to the YM2612.

It is strongly recommended that you deal with other things during V-blank first (such as any transfers you might need to do), *then* you can deal with the YM2612 and Dual PCM later. I am clearly unable to know your 68k side setup, so take this as a recommendation rather than an order.

Sample Popping

"My sample keeps popping in the same spot every time".

Dual PCM is not invincible, it still relies on the limitations of the hardware, the auto-banking mechanism might be what's responsible though.

Since Dual PCM needs to temporarily stop playing PCM audio in order to switch banks, this can lead to a single chop of the audio (the popping noise you hear), since bank switching doesn't occur very often, a single switch can often be fast or brief enough that one would never notice. Some samples though it might be more obvious.

My only recommendation is to reposition where in the ROM the sample is stored, if you need to align your samples to the next $8000 then so be it. Samples larger than $8000 bytes though, there is no fix for, since Dual PCM needs to switch the bank somewhere in the sample, all you can do is decide on your own the best place in the sample to do this.

There is no way to stop the popping for looping samples, since to re-loop the sample means to re-switch the bank again, Dual PCM *has* to stop processing sample data to switch the bank, there is honestly no way to get around this without sacrificing playback buffer time.

End marker size

Do all samples have to have end marker blocks as large as $180 bytes?

No, if you only intend to play the sample at normal pitch of $0100, then only $18 bytes will be required, a pitch of $0200 and only $30 bytes would be required, the higher the pitch you intend to play the sample at, the larger these blocks have to be.

It is recommended that you ensure all samples use the largest block size, but not essential. If you experience issues with samples not stopping, then your blocks are either not large enough, or the pitch change occurred at a different time to the sample request, this can for a brief moment allow Dual PCM to overflow whatever small block you've put in.

Also keep in mind that if you are sharing an end marker of the end of one sample, with the beginning of another sample, ensure that this end marker is large enough for both samples.

It's not essential, but be careful.

Address summary

The following is a list of addresses compiled together in a short list (and default values they should contain if any).

TBS = To Be Set.
RO = Read Only.

Address	Label	Description	Default value
$A00C62	MuteSample	Starting mute sample.	TBS
$A00C69	PCM1_Sample	PCM1's sample location/bank address	TBS
$A0064E	PCM1_NewRET	PCM1's sample request instruction	TBS
$A00C75	PCM2_Sample	PCM2's sample location/bank address	TBS
$A00651	PCM2_NewRET	PCM2's sample request instruction	TBS
$A005DD	PCM1_PitchHigh+1	PCM1's Pitch quotient (QQ)	$01
$A005E8	PCM1_PitchLow+1	PCM1's Pitch fraction (FF)	$00
$A005D2	PCM1_ChangePitch	PCM1's Pitch request instruction	TBS
$A0060E	PCM2_PitchHigh+1	PCM2's Pitch quotient (QQ)	$01
$A00619	PCM2_PitchLow+1	PCM2's Pitch fraction (FF)	$00
$A00603	PCM2_ChangePitch	PCM2's Pitch request instruction	TBS
$A01A38	SV_VolumeBank	Bank switch instructions addressed to the Volume bank.	TBS
$A00634	PCM_ChangeVolume	Change volume request instruction	TBS
$A00642	PCM1_Volume+1	PCM1's volume amount	$00
$A00647	PCM2_Volume+1	PCM2's volume amount	$00
$A00C68	YM_Buffer	Which YM2612 queue the Z80 is processing	RO
$A01000	YM_Buffer1	YM2612 Queue buffer 1 ($00/$FF)	TBS
$A01400	YM_Buffer2	YM2612 Queue buffer 2 ($FF/$00)	TBS
$A00143	Z80_DelayYM1+1	PCM synchronisation delay start 1 for YM2612 queue	$07
$A0016A	Z80_DelayYM2+1	PCM synchronisation delay start 2 for YM2612 queue	$07
$A0018F	Z80_DelayYM3+1	PCM synchronisation delay loop 1 for YM2612 queue	$07
$A00198	Z80_DelayYM4+1	PCM synchronisation delay loop 2 for YM2612 queue	$07
$A001B4	Z80_DelayEnd1+1	PCM synchronisation delay end 1 for YM2612 queue	$0C
$A001CD	Z80_DelayEnd2+1	PCM synchronisation delay end 2 for YM2612 queue	$0C
$A000B4	Z80_VBlank1+1	PCM synchronisation delay part 1 for V-blank idling	$0C
$A0009B	Z80_VBlank2+1	PCM synchronisation delay part 2 for V-blank idling	$0C
$A00688	Z80_Flush1+1	PCM synchronisation delay part 1 for flush loop	$0C
$A006A2	Z80_Flush2+1	PCM synchronisation delay part 2 for flush loop	$0C
$A00109	YM_FlushTimer+2	V-blank's default delay if YM queue is short	$02