4Front Technologies

FM SYNTHESIZER INTERFACE

INTRODUCTION

The FM synthesizer uses FM modulation to perform tone generation. The FM synthesizer supports OPL3 mode (2 or 4 op + stereo mode) or the older OPL2 mode (Adlib mono mode). FM syn- thesis interface to the device driver is provided through ioctls. The following sections deal with the basic FM synthesis programming techniques.

FM Synthesizer Specifications:

- Yamaha YMF 262 OPL-3 Chip
- Adlib Compatible OPL-2 mode
- OPL-2 mode (Adlib) supports 9 channels of 2-operator FM tones or 6 channels of 2 operator FM tones + 5 percussion instruments
- OPL-3 mode supports 18 channels of 2-operator Stereo FM tones or 6 channels of 4-operator and 6 channels of 2-operator FM tones 5 percussion instrument channels

FM synthesis uses a modulator cell and a carrier cell. The modulator cell modulates the carrier cell.The FM synthesizer provides 36 cells comprising of 18 modulator cells and l8 carrier cells that result in 18 simultaneous channels being generated. In essence, the 18 channels can generate any 2 operator note.

There are basically 2 modes supported by the FM synthesizer: OPL-2 and OPL-3 modes. OPL-2 mode consists of nine 2-operator note channels with mono output. OPL-3 mode consists of eighteen 2-operator note channels with stereo output or six 4-operator and six 2-operator channels. Both OPL-2 and OPL-3 modes support 5 percussion instruments and when the rhythm mode is selected, the percussion instruments occupy 3 channels (l channel for bass drum, l/2 channel for the other 4 percussion instruments).

DATA STRUCTURES

The FM synthesizer uses three structures for FM tone generation. The data structure dm_fm_voice sets the voice parameters for the FM tone. The parameters do not change for a given type of voice. The second data structure used is dm_fm_note. This data structure sets the frequency and octave and sounds the tone when activated for a particular channel. The final data structure is dm_fm_param. This data structure controls the rhythm section as well as global parameters for the FM synthesizer.

FM Voice Data Structure

```
struct dm_fm_voice
{
     unsigned char op;          /*0 for modulator and 1 for carrier */
     unsigned char voice;       /*Channels of 2-op notes*/
     unsigned char am;          /*Tremolo or AM modulation effect flag - 1 bit*/
     unsigned char vibrato;     /*Vibrato effect flag - 1 bit*/
     unsigned char do_sustain;  /*Sustaining sound phase flag -1 1 bit*/
     unsigned char kbd_scale;   /*keyboard scaling flag - 1 bit*/
     unsigned char harmonic;    /*Harmonic or frequency multiplier - 4 bits*/
     unsigned char scale_level; /*Decreasing volume of higher notes - 2 bits*/
     unsigned char volume;      /*Volume of output - 6 bits*/
     unsigned char attack;      /*Attack phase level of the note - 4 bits*/
     unsigned char decay;       /*Decay phase level of the note - 4 bits*/
     unsigned char sustain;     /*Sustain phase level of the note - 4 bits*/
     unsigned char release;     /*Release phase level of the note - 4 bits*/
     unsigned char feedback;    /*Feedback from op 1 or op 2 - 3 bits*/
     unsigned char connection;  /*Serial or parallel operator connection - 1 bit*/
     unsigned char left;        /*Left channel audio output*/
     unsigned char right;       /*Right channel audio output*/
     unsigned char waveform;    /*Waveform select - 3 bits*/
};
```

- **op** - holds the type modulator or carrier operator. A value of 0 denotes a modulator cell and 1 denotes a carrier cell.
- **voice** - holds the voice or the channel number. There are 36 op cells that result in 18 channels that can produce a note simultaneously. A value of 0 through 17 specify a channel. In rhythm mode, channels 6, 7 and 8 cannot be used to generate melodic notes.
- **am** - is a flag (1-bit) that turns the AM modulation (tremolo) effect on or off. The rate for AM modulation is 3.7 Hz.
- **vibrato** - is a flag (1-bit) that turn the vibrato effect on or off. The rate is 6.4Hz.
- **do_sustain** - is a flag (1-bit) that turns the sustained sound on or off. If do_sustain is 1 then the sustaining sound is sound when the note is played. If do_sustain is 0 then the diminishing sound is selected when the note on.

- **harmonic** - is a 3 bit field (values 0-7) which represents the harmonic or the multiplication factor that needs to be applied to the frequency (fnum). The following table specifies the multiplication factor with respect to harmonic number.

- **kbd_scale** is a flag (1-bit) that turns on the keyboard scale rate. If kbd_scale is 1 then the attack/decay rates become faster as the pitch (fnum + octave) increases.

- **scale_level** - is a 2 bit field (values 0-3) which produces a gradual decrease in note output level towards higher pitches (octave+fnum). The following table shows the scale level and the corresponding attenuation.

- **volume** - is a 6 bit field (values 0-63) which represents the total output volume of the op. Maximum attenuation is 47.25 dB. Attenuating the output from a modulator cell will change the frequency spectrum produced by the carrier cell.

- **attack** - is a 4-bit field (values 0-15) that sets the attack rate or the rising time of the sound.
- **decay** - is a 4-bit field (values 0-15) that sets the decay rate or the diminishing time after the attack.
- **sustain** - is a 4-bit field (values 0-15) that set the sustain level. For continuing sounds the sustain level gives the point of change where the attenuated sounds in the decay mode changes to a sound having a constant level. For diminishing sounds, the sustain level gives the point where the decay mode changes to a release mode.
- **release** - is a 4-bit field (values 0-15) that set the release level. For continuing sounds, the release level defines the rate at which sound disappears after key_off. For diminishing sounds, the release level indicates the attenuation after the sustain level is reached.
- **feedback** - is a 3 bit field (values 0-7) which determines the modulation factor for self-feedback. This is applicable only to the modulator cell.

- **connection** - is a flag (1-bit) that describes the connection of the modulator and carrier. When connection is 0 the carrier is chained to the modulator in a serial connection and produces true FM tones. If connection is 1, the carrier and modulator are connected in parallel to produce two simultaneous tones.
- **waveform** - is a 3 bit field (values 0-7) which specify the shape of the waveform..
- **left** - is a flag (1 bit) field which enables (set to 1) the left output channel or disables (set to 0) the left output channel.
- **right** - is a flag (1 bit) field which enables (set to 1) the right output channel or disables (set to 0) the right output channel.

FM Note Data Structure

The next data structure consists of parameters such as octave, channel and frequency. These parameters vary compared to the dm_fm_voice characteristics.

```
struct dm_fm_note
{
   unsigned char voice;         /*18 channels of 2-op notes*/
   unsigned char octave;        /*Octave number of the note - 3 bits*/
   unsigned int fnum;           /*Frequency of the note - 10 bits*/
   unsigned char key_on;        /*Output sound flag - 1 bit*/
};
```

- **voice** - holds the voice or the channel number. There are 36 op cells that result in 18 channels that can produce a note simultaneously. A value of 0 through 17 specify a channel. In rhythm mode, channels 6, 7 and 8 cannot be used to generate melodic notes.
- **octave** - is a 3 bit value (values 0-7) which represents the octave number of the note.
- **fnum** - is a 10 bit value (values 0-1023) which represents the frequency of the note. The following table shows the frequencies of all notes for octave=4 [Table]
- **key_on** - is a flag (1-bit) that voices the notes (i.e sound is produced). When key_on is 1 sound is produced. If key_on is 0 no sound is produced. In order to sound a note the key_on should make a 0 to 1 transition. Hence you need to set it to 0 and then set it to 1 in the FM synthesizer.

FM Parameter Data Structure

The following is a description of the data structure used for the rhythm section and global FM parameters.

```
struct dm_fm_param
{
    unsigned char am_depth;     /*AM modulation depth for AM modulation effect*/
    unsigned char vib_depth;    /*Vibrato depth for Vibrato effect*/
    unsigned char kbd_split;    /*split keyboard for kbd_scaling*/
    unsigned char rhythm;       /*turn on rhythm mode*/
    unsigned char bass;         /*bass  - occupies channel 7(modulator & carrier)*/
    unsigned char snare;        /*snare - occupies modulator of channel 8*/
    unsigned char tomtom;       /*tom-tom - occupies modulator of channel 9*/
    unsigned char cymbal;       /*cymbal - occupies carrier of channel 9*/
    unsigned char hihat;        /*hihat - occupies carrier of channel 8*/
};
```

- **am_depth** This is a flag (1 bit) field which determines the Amplitude Modulation (tremolo) depth. The attenuation is 4.8dB when am_depth=1 and 1dB when am_depth=0.
- **vib_depth** This is a flag (1 bit) field which determines the Vibrato depth of the op cell. The attenuation factor is 14% when vib_depth=1 and 7% when vib_depth=0.
- **kbd_split** This is a flag (1 bit) field which determines the split method to select the key scale number. This field is used to select the kbd_scale value. Depending on the pitch of the note (octave + fnum), a key scale number between 0 and 15 is generated. This key scale number is applied to the attack/decay/sustain/release rates depending whether kbd_scale is 1 or 0. If kbd_split=1 then the key scale number depends on the Most Significant Bit (MSB) of the frequency. If kbd_split=0 then the key_scale number depends on the the 2nd MSB of the frequency.
- **rhythm** This is flag (1 bit) field. When rhythm=1, the channels 6,7 and 8 are used to generate percussion instruments such as bass, snare, tomtom, hihat and cymbals. Hence regular FM notes or ops cannot be played on these channels.
- **bass** This is a flag (1 bit) field which turns on or off the bass drum percussion instrument. The bass drum requires a modulator and a carrier cell that occupy channel 6. Both ops require note settings (Attack/Decay/Sustain/Release/Octave/Fnum etc) but the key_on field should be set to 0. Only when bass=1 is the bass drum sound produced.
- **snare** This is a flag (1 bit) field which turns on or off the snare drum percussion instrument. The snare drum requires a modulator cell occupying channel 7. The modulator cell needs to be set with the note settings that simulate a snare drum but the key_on field must be 0. Snare drum sound is produced when snare=1.
- **tomtom** This is a flag (1 bit) field which turns on of off the tomtom drums. The tomtom drum requires a carrier cell occupying channel 8. The carrier cell needs to be set with note settings that resemble the tomtom drum but the key_on field must be 0. Tomtom drum sound is produced when tomtom=1.
- **cymbal** This is a flag (1 bit) field which turns on or off the cymbal. The cymbal instrument requires a modulator cell occupying channel 8. The modulator cell must be set with note settings that resemble a cymbal. As with the previous percussion instrument, key_on should be 0. Only when cymbal=1, is the cymbal sound produced.
- **hihat** This is a flag (1 bit) field which turns on or off the hihat. The hihat instrument requires a carrier cell occupying channel 7. The carrier cell must be set with note settings that resemble a hihat. As with the previous percussion instrument, key_on should be 0. Only when hihat=1, is the hihat sound produced.

FM SYNTHESIZER IOCTLS

In this section we will examine the ioctls that are used to generate FM tones. Before you can issue the ioctls you must first obtain the file descriptor using an open call on the /dev/sbpfm0 device described in section 1. In the following examples we will use the file descriptor fmfd for the FM synthesizer device.

- **FM_IOCTL_RESET**
  Parameters: None
  This ioctl is used to reset the FM synthesizer After opening the FM device /dev/dmfm0 it is advisable to issue a reset ioctl.
  This ioctl is used as follows:
  `ioctl(fmdev, FM_IOCTL_RESET, NULL);`
- **FM_IOCTL_SET_MODE**
  Parameters: OPL2 or OPL3
  This ioctl is used to set the mode of the FM synthesizer. The parameters for this ioctl are OPL2 or OPL3. OPL2 sets the FM synthesizer in OPL2 or ADLIB compatible mode. In this mode only 9 channels of 2 op voices with mono output are permitted. In OPL3 mode, there are 18 channels of 2 op with stereo output or 6 channels of 4 ops and 6 channels of 2 ops with stereo output.
  The ioctl to set the FM synthesizer in OPL3 mode is as follows:
  `ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3);`

- **FM_IOCTL_SET_VOICE**
  Parameters: `&(struct dm_fm_voice)voice`
  This ioctl sets the voice parameters for the modulator and carrier ops. The parameter voice is of type struct dm_fm_voice. The ioctl us used as follows:
  `ioctl(fmfd, FM_IOCTL_SET_VOICE, &voice);`

- **FM_IOCTL_PLAY_NOTE**
  Paramters: `&(struct dm_fm_note)note`
  This ioctl is used to voice a particular FM channel which has been preset with the FM voice characteristics. The parameter note is of type struct dm_fm_note.
  The ioctl is used as follows:
  ```
  note.key_on = 0;
  ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
  note.key_on = 1;
  ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
  ```
- **FM_IOCTL_SET_PARAMS**
  Paramters: `&(struct dm_fm_params)param`
  This ioctl is used to set global FM paramters as well as control the percussion instruments. The parameter param is of type struct dm_fm_params.
  The ioctl is used as follows:
  `ioctl(fmfd, FM_IOCTL_SET_PARAMS, &param);`
- **FM_IOCTL_SET_OPL**
  Paramters: `(char) conn_type;`
  This ioctl is used to set the connection type for the 4 op mode. The parameter conn_type is a byte defining the connection. If you want the synthesizer in 2 -op mode then the conn_type = 0x0. If you want the synthesizer in 4-op mode with six 4-op channels then conn_type = 0x3F.

PROGRAMMING THE FM SYNTHESIZER

In this section we will write a simple program to play random notes on the FM synthesizer. This example will demonstrate the ability of the FM synthesizer.

```
#include <stdio.h>
#include <fnctl.h>
#include <math.h>
#include <sys/dm.h>

#define VOICES 18;
#define RAND(bits)  (random() & (1<<(bits)) -1))
main()
{
        int fmfd;
        struct dm_fm_voice modulator, carrier;
        struct dm_fm_note note;
        struct dm_fm_params param;
        int channel_num;

/* First we open the FM device using an open call */
        fmfd = open("/dev/dmfm0", O_WRONLY);
        if (fmfd < 0)
           perror("open");

/* Now we reset the FM synthesizer using the RESET ioctl */
        if (ioctl(fmfd, FM_IOCTL_RESET) == -1)
           perror("reset");

/* Now set the FM synthesizer in OPL3 mode */
        if (ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3) == -1)
           perror("mode");

        while (1) {
/* set global parameters but do not turn on percussion section */
            param.am_depth = RAND(1);
            param.vib_depth = RAND(1);
            param.kbd_split = RAND(1);
            param.rhythm = 0;
            param.bass = 0;
            param.snare = 0;
            param.hihat = 0;
            param.cymbal = 0;
            param.tomtom = 0;
/* send the param structure to the FM synthesizer */
            ioctl(fmfd, FM_IOCTL_SET_PARAMS, &param);

/* Play the note on all channels at the same time */
            for (channel_num = 0; channel_num < VOICES; channel_num++)
            {
/*
 * Now fill in the modulator cell structure using randomly gener
 * ated values and masking off the bits. Look at the definition
 * of RAND(bits)
 */
                modulator.voice = channel_num;
```

```
                modulator.op = 0;
                modulator.am = RAND(1);
                modulator.vibrato = RAND(1);
                modulator.do_sustain = RAND(1);
                modulator.kbd_scale = RAND(1);
                modulator.connection = 0;
                modulator.attack = RAND(4);
                modulator.decay = RAND(4);
                modulator.sustain = RAND(4);
                modulator.release = RAND(4);
                modulator.octave = RAND(3);
                modulator.volume = RAND(6);
                modulator.scale_level = RAND(2);
                modulator.feedback = RAND(3);
                modulator.waveform = RAND(3);
                modulator.left = RAND(1);
                modulator.right = RAND(1);
/* Send the modulator structure to the FM synth */
                if (ioctl(fmfd, FM_IOCTL_SET_VOICE, &modulator) == -1)
                   perror("modulator");
/*
 * Now fill in the carrier cell structure using randomly gener
 * ated values and masking off the bits. Look at the definition
 * of RAND(bits)
 */
                carrier.voice = channel_num;
                carrier.op = 1;
                carrier.am = RAND(1);
                carrier.vibrato = RAND(1);
                carrier.do_sustain = RAND(1);
                carrier.kbd_scale = RAND(1);
                carrier.connection = 0;
                carrier.attack = RAND(4);
                carrier.decay = RAND(4);
                carrier.sustain = RAND(4);
                carrier.release = RAND(4);
                carriers.harmonic = RAND(4);
                carrier.volume = RAND(6);
                carrier.scale_level = RAND(2);
                carrier.feedback = RAND(3);
                carrier.waveform = RAND(3);
                carrier.left = RAND(1);
                carrier.right = RAND(1);
/* Send the carrier structure to the FM synth */
                if (ioctl(fmfd, FM_IOCTL_SET_VOICE, &carrier) == -1)
                   perror("carrier");
/*
 * Now fill in the note  structure with random octaves and frequencies.
 * Before sounding the FM tone turn the note off and then key_on the note.
 */
                note.voice = channel_num;
                note.octave = RAND(3);
                note.fnum =  RAND(10);
                note.key_on = 0;
                if (ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note) == -1)
                   perror("note");
                note.key_on = 1;
                if (ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note) == -1)
                   perror("note");
/* sleep between notes */
                usleep(100000);
            } /*for loop*/
        } /*while loop */
}
```

ADDITIONAL NOTES ON FM PROGRAMMING

FM synthesis requires many parameter fields to be set and sometimes it is simpler to use "patches" to simulate various instruments. The Sound Blaster Instrument (SBI) format provides a uniform approach to programming the FM synthesizer. The SBI format only handles sound characteristics. The program has to provide the frequency and octave values. Sounding of the FM tone occurs when a key_on it is set on a particular voice channel. The following is a description of the SBI file format:

Note: The names in parenthesis denote the dm_fm_note structure parameter.

OFFSET (hex) Description

```
00 - 03     File ID - 4 Bit ASCII String "SBI" ending with 0x1A

04 - 23     Instrument Name - Null terminated ASCII string

24          Modulator Sound Characteristics
            Bit 7: AM Modulation (am)
            Bit 6: Vibrato (vibrato)
            Bit 5: Sustaining Sound (do_sustain)
            Bit 4: Envelop Scaling (kbd_scale)
            Bits 3-0: Frequency Multiplier (harmonic)

25          Carrier Sound Characteristics
            Bit 7: AM Modulation (am)
            Bit 6: Vibrato (vibrato)
            Bit 5: Sustaining Sound (do_sustain)
            Bit 4: Envelop Scaling (kbd_scale)
            Bits 3-0: Frequency Multiplier (harmonic)

26          Modulator Scaling/Output Level
            Bits 7-6: Level Scaling (scale_level)
            Bits 5-0: Output Level (volume)
```

```
27          Carrier Scaling/Output Level
            Bits 7-6: Level Scaling (scale_level)
            Bits 5-0: Output Level (volume)

28          Modulator Attack/Decay
            Bits 7-4: Attack Rate (attack)
            Bits 3-0: Decay Rate (decay)

29          Carrier Attack/Decay
            Bits 7-4: Attack Rate (attack)
            Bits 3-0: Decay Rate (decay)

2A          Modulator Sustain/Release
            Bits 7-4: Sustain Level (sustain)
            Bits 3-0: Release Rate (release)

2B          Carrier Sustain/Release
            Bits 7-4: Sustain Level (sustain)
            Bits 3-0: Release Rate (release)

2C          Modulator Wave Select
            Bits 7-2: All bits clear (0)
            Bits 1-0: Wave Select (waveform)

2D          Carrier Wave Select
            Bits 7-2: All bits clear (0)
            Bits 1-0: Wave Select (waveform)

2E          Feedback/Connection
            Bits 7-4: All bits clear (0)
            Bits 3-1: Modulator Feedback (feedback)
            Bit 0: Connection (connection)

2F - 33     Reserved for future use
```

```
The above description requires you to declare the following structures:
        struct dm_fm_voice modulator;
        struct dm_fm_voice carrier;
        struct dm_fm_note note;

Now stuff the corresponding values from the SBI file into the respective structures. From the above description, the programmer needs to provide are t

       modulator.voice = carrier.voice = x     (where x is a channel number between 0 and 17)
       modulator.left = carrier.left = x       (where x is the left audio output flag 0 or 1)
       modulator.right = carrier.right = x     (where x is the right audio output flag 0 or 1)
       modulator.op = 0     (modulator's op number is 0)
       carrier.op = 1       (carrier's op is 1)

       note.voice = x       (where x is a channel number from 0-17)
       note.fnum =  x       (where x is a frequency number from 0-1023)
       note.octave = x      (where x is an octave number from 0-7)
       note.key_on = 1      (the note must be keyed off and then keyed on)
```

Programming The FM Synthesizer Using SBI Files

The following code explains the mechanism to read an SBI format file and play the note at frquency 800 in octave number 5.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include "dm.h"

struct dm_fm_voice op0, op1;  /* the voice struct to hold the SBI file */
struct dm_fm_note note;       /* the note struct to make the sound */
int fmfd, fd;                 /* fmfd - FM dev handle; fd - SBI file */
char instrument_buf[16];      /* buffer to hold the SBI data from file */

main (argc, argv)
int argc;
char **argv;
{
        fmfd = open("/dev/dmfm0", O_WRONLY);   /* open the FM device */
        ioctl(fmfd,FM_IOCTL_RESET);            /* reset the FM device */
        ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3);  /* set mode to OPL3 */
        set_params();                          /* set global FM params */
        fd = open(argv[1], O_RDONLY);          /* open the SBI file */

/* now verify that it is truely an SBI instrument file by reading the
 * hearder
 */
        if (!verify_sbi(fd)) {
                printf("file is not in SBI format\n");
                exit (0);
        }
        get_instrument(fd);                    /* fill the voice structs */
        play_instrument();                     /* play the sound */
}

/* check for "SBI" + 0x1A (\032) in first four bytes of file */
int verify_sbi(fd)
int fd;
{
    char idbuf[5];    /* get id */
```

```
    lseek(fd, 0, SEEK_SET);
    if (read(fd, idbuf, 4) != 4)
        return(FALSE);    /* compare to standard id */
    idbuf[4] = (char)0;
    if (strcmp(idbuf, "SBI\032") != 0)
        return(FALSE);    return(TRUE);
}

get_instrument(fd)
int fd;
{
    lseek(fd, 0x24, SEEK_SET);
    read(fd, instrument_buf, 16);

/* Modulator Characteristics */
        if (instrument_buf[0] & (1<<7))
                op0.vibrato = 1;
        else
                op0.vibrato = 0;
        if (instrument_buf[0] & (1<<6))
                op0.am = 1;
        else
                op0.am = 0;
        if (instrument_buf[0] & (1<<5))
                op0.kbd_scale = 1;
        else
                op0.kbd_scale = 0;
        if (instrument_buf[0] & (1<<4))
                op0.do_sustain = 1;
        else
          op0.do_sustain = 0;
        op0.harmonic = instrument_buf[0] & 0x0F;

/* Carrier Characteristics */
        if (instrument_buf[1] & (1<<7))
                op1.vibrato = 1;
        else
                op1.vibrato = 0;
        if (instrument_buf[1] & (1<<6))
                op1.am = 1;
        else
                op1.am = 0;
        if (instrument_buf[1] & (1<<5))
                op1.kbd_scale = 1;
        else
                op1.kbd_scale = 0;
        if (instrument_buf[1] & (1<<4))
                op1.do_sustain = 1;
        else
    op1.do_sustain = 0;
        op1.harmonic = instrument_buf[1] & 0x0F;

/* Modulator Scale/Volume Level */
        op0.scale_level = instrument_buf[2] >>6;
        op0.volume = instrument_buf[2] & 0x3f;

/* Carrier Scale/Volume Level */
        op1.scale_level  = instrument_buf[3] >>6;
        op1.volume = instrument_buf[3] & 0x3f;

/* Modulator Attack/Decay */
        op0.attack = instrument_buf[4] >> 4;
        op0.decay = instrument_buf[4] & 0xF;

/* Carrier Attack/Decay */
        op1.attack = instrument_buf[5] >> 4;
        op1.decay = instrument_buf[5] & 0xF;

/* Modulator Sustain/Release */
        op0.sustain = instrument_buf[6] >> 4;
        op0.release = instrument_buf[6] & 0xF;

/* Carrier Sustain/Release */
        op1.sustain = instrument_buf[7] >> 4;
        op1.release = instrument_buf[7] & 0xF;

/* Modulator Waveform */
        op0.waveform = instrument_buf[8] & 0x03;

/* Carruer Waveform */
        op1.waveform = instrument_buf[9] & 0x03;

/* Modulator Feedback/Connection*/
        op0.connection = instrument_buf[0xA]  & 0x01;
        op1.connection = op0.connection;
        op0.feedback = (instrument_buf[0xA] >> 1) & 0x07;
        op1.feedback = op0.feedback;

/* byte 0xB - 20  Reserved */
}

play_instrument()
{
/*
 * Set the FM channel to channel 0. Fill in the rest of the fields and
 * Issue an ioctl to set the modulator parameters
 */
                op0.op = 0;
                op0.voice = 0;
                op0.left = 1;
                op0.right = 1;
                ioctl(fmfd, FM_IOCTL_SET_VOICE, &op0);
/*
 * Set the FM channel to channel 0. Fill in the rest of the fields and
```

```
 * Issue an ioctl to set the carrier parameters
 */
                op1.op = 1;
                op1.voice = 0;
                op1.left = 1;
                op1.right = 1;
                op1.volume = 63;
                ioctl(fmfd, FM_IOCTL_SET_VOICE, &op1);
/*
 * Fill in the note structure and first key_off the note and then key_on.
 */
                note.voice = 0;             /* select channel 0 */
                note.octave = 5;
                note.fnum = 800;
                note.key_on = 0;                /*Key off*/
                ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
                note.key_on = 1;                /*Key on*/
                ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note)
}

set_params()
{
struct dm_fm_params p;
        p.am_depth = 0;
        p.vib_depth = 0
        p.kbd_split = 0;
        p.rhythm = 0;
        p.bass = 0;
        p.snare = 0;
        p.tomtom = 0;
        p.cymbal = 0;
        p.hihat = 0;
        ioctl (fmfd, FM_IOCTL_SET_PARAMS, &p);
}
```

FM Synthesizer in 4 - Operator Mode

In th 4-op mode, the FM synthesizer uses 4 ops which actually comprize of two 2-op channels. From 18 2-op channels, we can get six 4-op channels, with 5 channels for percussion (as described above) and three 2-op channels used for FM voices. The diagram below describes how 18 2-op channels are organized:

4-Operator Schematic for Modulator/Carriers

The ioctl FM_IOCTL_SET_OPL is used to set the 4-op connection mask. This ioctl requires a 6 bit mask. If the mask is 0 then all the FM voice channels default to 2 op mode thus yielding 18 channels or 15 voice plus 5 percussion channels. If the mask is set to 0x3F then the FM synthesizer is configured for six 4 op voice channels plus six 2 op voice channels. The six 2 op voice channels can be configured for 5 percussion channels a 3 voice channels or 6 voice channels. In the case of the 4 -op channels the above diagram shows which two op channels go into building the six 4 op channels.

With 4 ops, the following diagram shows how the ops can be connected. The connection bits from the first two ops is designated as C0 and the connection bits from the remaining two are designated as C1.

Connection Possibilities with 4 operators
