DiSPATCH
Host Support Software Manual

Revision 1.52
Updated 2 May 1997
Part Number: 880-3075-004

Vigra, a division of VisiCom Labs.

CONTENTS

1 Document Overview **1**

2 Software Overview **3**

2.1 Do you need to use Vigra's library and drivers? 3

3 Installation **5**

3.1 SunOS v4.1.x 6

3.1.1 Kernel Architecture 7

3.1.2 Installation Summary 7

3.1.3 Install the Source Files 8

3.1.4 Select the Mapping Function 8

3.1.5 Create the Kernel Configuration File 8

3.1.6 Add Driver Entries to "`conf.c`" 10

3.1.7 Files 10

3.1.8 Configure the Kernel 11

3.1.9 Build, Install, and Boot the New Kernel 11

3.1.10 Make the Device Handles in "`/dev`" 11

3.1.11 Test the Driver 12

3.2 Solaris 2.x 13

3.2.1 Installation Summary 13

3.2.2 Configure and Install the MMI Boards 14

3.2.3 Check Available Disk Space 14

3.2.4 Install the VIGRAmmi Package 14

3.2.5 Driver Configuration 15
3.2.6 Test the Installation 17
3.3 VxWorks 18
3.3.1 Building the VxWorks package 18
3.3.2 Configuration 19
3.3.3 The Included Object Files 20
3.3.4 Testing the Audio System 21
3.4 IRIX v4.0.x 23
3.4.1 Installation 23
3.4.2 Building the Driver 24
3.4.3 Install the Driver Binary 24
3.4.4 VME Vector Configuration 24
3.4.5 The Driver Configuration File 25
3.4.6 Rebuild the Kernel 26
3.4.7 Boot the New Kernel 26
3.4.8 Create Device Nodes 26
3.4.9 Test the Driver 27
3.5 IRIX v5.2 28
3.5.1 Installation 28
3.5.2 Select a Major Number 29
3.5.3 Configure “`mmidsp.sm`” for your MMI boards 29
3.5.4 Build and Install the Driver 31
3.5.5 Create the Device Handles 31
3.5.6 Rebuild the Kernel 32
3.5.7 Boot the New Kernel 32
3.5.8 Test the Driver 32
3.6 OS-9 33
3.6.1 Functional Overview 33
3.6.2 Installation 34

3.6.3 Extract DiSPATCH files 34
3.6.4 Configuration 35
3.6.5 Compile All Source Files 35
3.6.6 Edit the System Startup File 36
3.6.7 Reboot the System 36
3.6.8 Test the Package 36
3.7 HP-RT 37
3.7.1 Installation 37
3.7.2 Extract the DiSPATCH Files 38
3.7.3 The `HPRTroot` Environment Variable 38
3.7.4 Interrupt Level Allocation 38
3.7.5 Driver Configuration 39
3.7.6 Add the Driver 40
3.7.7 Build a New HP-RT Kernel 40
3.7.8 Build the Library and Applications 40
3.7.9 Reboot the System 41
3.7.10 Test the Package 41
3.8 HP-UX 9.0.x 42
3.8.1 Installation 42
3.8.2 Extract the DiSPATCH Files 43
3.8.3 Configure your new MMI board 43
3.8.4 Interrupt Level Allocation 43
3.8.5 Driver Configuration 44
3.8.6 Add the Driver to the Master File 45
3.8.7 Add the Driver to the driver description file 46
3.8.8 Build a New HP-UX Kernel 46
3.8.9 Install the New HP-UX Kernel 46
3.8.10 Build the Library and Applications 46
3.8.11 Reboot the System 47

3.8.12 Make the Device Handles in "`/dev`" 47
3.8.13 Test the Package 48
3.8.14 HP-UX System Administration Manager (SAM) 49
3.9 HP-UX 10.xx 49
3.9.1 Installation 49
3.9.2 Extract the DiSPATCH Files 50
3.9.3 Configure the MMI Board 50
3.9.4 HP-UX VME Settings 51
3.9.5 Driver Configuration 51
3.9.6 Build a New HP-UX Kernel 53
3.9.7 Back Up Your Existing Kernel 53
3.9.8 Install the New HP-UX Kernel 53
3.9.9 Build the Library and Applications 53
3.9.10 Reboot the System 54
3.9.11 Make the Device Handles in "`/dev`" 54
3.9.12 Test the Package 54
3.10 Motorola System V/88 55
3.10.1 Installation 55
3.10.2 Extract the DiSPATCH Files 56
3.10.3 Check over the definitions in `Makefile` 56
3.10.4 Build the Device Driver and Configuration Tools 57
3.10.5 Driver Configuration 58
3.10.6 Install the DiSPATCH Driver and Support Files 59
3.10.7 Add the Driver to the Kernel Using `/etc/sysgen` 59
3.10.8 Verify the Maximum Shared Memory Segment Size (SHMMAX). 60
3.10.9 Build a New Kernel 60
3.10.10 Reboot the System V/88 System 60
3.10.11 Build the Library and Applications 60
3.10.12 Test the Package 61

4 C Programming Library **63**

4.1 Library Overview . . . 63

4.1.1 Device Driver . . . 63

4.1.2 DSP Naming . . . 64

4.1.3 Shared Access . . . 64

4.1.4 Handles . . . 65

4.1.5 Source Code . . . 65

4.1.6 Include Files . . . 65

4.1.7 Completion Tokens . . . 66

4.2 Initialization and Control Functions . . . 67

4.2.1 `mmi_open` . . . 67

4.2.2 `mmi_close` . . . 69

4.2.3 `mmi_lib_initialize` . . . 70

4.2.4 `mmi_get_dsp` . . . 71

4.2.5 `mmi_get_ram_base` . . . 72

4.2.6 `mmi_get_model` . . . 73

4.2.7 `mmi_start_firmware` . . . 74

4.2.8 `mmi_boot_default` . . . 75

4.2.9 `mmi_diag_boot` . . . 76

4.2.10 `mmi_boot_file` . . . 78

4.2.11 `mmi_halt_dsp` . . . 79

4.2.12 `mmi_poll_messages` . . . 80

4.2.13 `mmi_malloc` . . . 81

4.2.14 `mmi_test_host_dram` . . . 82

4.2.15 `mmi_wait_response` . . . 83

4.2.16 `mmi_complete` . . . 84

4.2.17 `mmi_check_response` . . . 85

4.2.18 `mmi_get_dsp_filedes` . . . 86

4.2.19 `mmi_register_callback` . . . 87

4.2.20 mmi_delete_callback 88
4.2.21 mmi_init_cmd_codes 89
4.2.22 mmi_start_dribble 90
4.2.23 mmi_end_dribble 91
4.2.24 mmi_discard_resp 92
4.2.25 mmi_parse_format 93
4.2.26 mmi_samples_per_word 94
4.2.27 mmi_dsp_command 95
4.3 DiSPATCH Command Functions 96
4.3.1 mmi_load_dspmem 96
4.3.2 mmi_load_dspmem_file 97
4.3.3 mmi_register_counter 98
4.3.4 mmi_end_counter 99
4.3.5 mmi_reset_counter 100
4.3.6 mmi_read_counter 101
4.3.7 mmi_play_buf 102
4.3.8 mmi_play_subbuffers 103
4.3.9 mmi_record_buf 104
4.3.10 mmi_monitor_buf 105
4.3.11 mmi_play_file 106
4.3.12 mmi_record_file 107
4.3.13 mmi_set_play_format 108
4.3.14 mmi_set_record_format 109
4.3.15 mmi_set_monitor_format 110
4.3.16 mmi_set_play_gain 111
4.3.17 mmi_set_record_gain 112
4.3.18 mmi_play_ctrl 113
4.3.19 mmi_record_ctrl 114
4.3.20 mmi_play_position 115

4.3.21 mmi_record_position . . . 116
4.3.22 mmi_monitor_position . . . 117
4.3.23 mmi_abort_track . . . 118
4.3.24 mmi_abort_all_play . . . 119
4.3.25 mmi_abort_record . . . 120
4.3.26 mmi_abort_monitor . . . 121
4.3.27 mmi_led_ctrl . . . 122
4.3.28 mmi_set_input_gain . . . 123
4.3.29 mmi_set_output_gain . . . 124
4.3.30 mmi_set_mixer . . . 125
4.3.31 mmi_set_speed_change . . . 126
4.3.32 mmi_set_resample . . . 127
4.3.33 mmi_set_sidetone . . . 128
4.3.34 mmi_transform_buffer . . . 129
4.3.35 mmi_transform_file . . . 130
4.3.36 mmi_count_buffers . . . 131
4.3.37 mmi_clip_led . . . 132
4.3.38 mmi_enable_measurements . . . 133
4.3.39 mmi_disable_measurements . . . 134
4.3.40 mmi_input_peak . . . 135
4.3.41 mmi_input_bias . . . 136
4.3.42 mmi_signal_detect . . . 137
4.3.43 mmi_set_equalizer . . . 139
4.3.44 mmi_set_reverb . . . 140
4.3.45 mmi_select_input . . . 141
4.3.46 mmi_stereo_mode . . . 142
4.3.47 mmi_query_stereo . . . 143
4.3.48 mmi_filter_play . . . 144
4.3.49 mmi_filter_record . . . 145

4.3.50 mmi_load_play_fir 146
4.3.51 mmi_load_record_fir 147
4.3.52 mmi_set_srate 148
4.3.53 mmi_request_srate 149
4.3.54 mmi_set_pnm 150
4.3.55 mmi_play_tone 151
4.3.56 mmi_ramtone 152
4.3.57 mmi_end_tone 154
4.3.58 mmi_load_table 155
4.3.59 mmi_waveshape 156
4.3.60 mmi_fetch_error 157
4.3.61 mmi_enable_mail 158
4.3.62 mmi_disable_mail 159
4.3.63 mmi_probe_firmware 160
4.3.64 mmi_get_version 161
4.3.65 mmi_show_configuration 162
4.3.66 mmi_test_dsp_dram 163
4.3.67 mmi_start_loopback 164
4.3.68 mmi_end_loopback 165
4.3.69 mmi_issue_invalid_cmd 166
4.3.70 mmi_dsp_nop 167
4.3.71 mmi_query_load 168
4.3.72 mmi_analog_test 169
4.3.73 mmi_hammer_dsp 171
4.3.74 mmi_eprom_checksum 172
4.3.75 mmi_show_state 173

5 Example Applications 175

5.1 MMI-Test 175
5.1.1 Running MMI-Test 175

5.1.2 The Command Line 176
5.1.3 Selecting and Initializing an MMI Board 177
5.1.4 An Example 178
5.1.5 X-Windows Sliders 179
5.1.6 Command Reference 180
5.2 ToneShop and SampleTones 190
5.3 `simp_play.c` and `simp_record.c` 190
5.4 Play 190
5.4.1 Unix Usage 191
5.4.2 VxWorks Usage 192
5.5 Record 192

1. Document Overview

This manual describes the DiSPATCH Software Package provided by Vigra for use with the DiSPATCH DSP firmware. The DiSPATCH Software Package includes the following parts:

- A device driver tailored for each supported host platform.
- A C programming library.
- An interactive DiSPATCH test program.
- Several example application programs.
- A binary image of the DiSPATCH firmware program to upload to the DSP.

This manual describes the installation and use of these programs on the host system.

This manual does *not* describe the concepts or operation of DiSPATCH firmware itself. That information can be found in the *DiSPATCH Firmware User's Manual*, available from Vigra.

This manual is divided into three major parts:

1. System installation.
2. The C programming library.
3. Example applications (including MMI-Test).

Please be sure to read the files called "`README`" included with the software release. These files contains information that may not have been included in the latest release of the printed instructions.

2. Software Overview

The DiSPATCH system is a unique operating environment that allows the DSP to process audio streams and buffers on behalf of the host. The DiSPATCH system consists of two simple parts:

1. The DiSPATCH firmware running on each DSP.
2. The DiSPATCH host support software running on the VME host computer.

These two parts of DiSPATCH communicate over the VME bus, passing messages and sharing audio data.

2.1 Do you need to use Vigra's library and drivers?

Strictly speaking, the host programmer does not need to use *any* of the host support software provided by Vigra. The DiSPATCH DSP firmware is documented in detail in the *DiSPATCH User's Manual*. The firmware itself is completely independent of the host environment and does not depend on the library or driver for vital computations.

System programmers with unique system environments or special requirements may find it necessary to communicate with the DSP directly from their application, without using the provided library interface at all. For these applications, the source code to the library is provided as a valuable example for all of the functions provided by DiSPATCH.

However, for application programmers using a supported host operating system, the DiSPATCH programming library probably provides the most direct path to implementation, since most of the low-level initialization and communication functions are already coded. All functions provided by the DiSPATCH firmware are fully supported by the programming library.

The library provides an identical application interface to DiSPATCH on all supported platforms. The support software is fully interrupt-driven; the host will never use CPU time waiting or polling for responses. This allows maximum performance and flexibility on multitasking and real-time operating systems.

Any application linked with the library contains the binary images of the DiSPATCH firmware, making a self-contained application program. No external files are necessary to initialize or use the DSP.

3. INSTALLATION

At the time of this writing, host-support packages are available for these host platforms:

- SunOS v4.1.x (Sun Microsystems)
  - Sun Sparcstation using Performance Technologies SBS-915 (PTVME) SBus-to-VME adaptor.
  - Force CPU 2CE/16 VME Sun Sparc.
  - Force CPU 1E VME Sun Sparc.
  - Themis SPARC 10HS (SunOS 4.1.3B_HS)
  - Themis SPARC 10MP (SunOS 4.1.4 + Themis VME patches)
- Solaris v2.x, SPARC (Sun Microsystems)
  - Themis VME SPARC CPU (SunOS 5.5 + Themis VME driver)
  - Force VME SPARC CPU (SunOS 5.4 or 5.5 + Force VME driver)
- VxWorks v5.x (Wind River Systems)
  - Motorola MVME167 Single Board Computer (M68040).
  - Force CPU 2CE/16 VME Sun SPARC.
- IRIX v4.0.x (Silicon Graphics Inc.)
  - All SGI workstations equipped with a VMEbus.
- IRIX v5.2 (Silicon Graphics Inc.)
  - All SGI workstations equipped with a VMEbus.
- OS-9 v2.4 (Microware Systems Corp.)
  - Motorola MVME167 Single Board Computer (M68040).

- HP-RT v1.11 (Hewlett-Packard Co.)
  - HP 9000/742 target board.
- HP-UX v9.0x (Hewlett-Packard Co.)
  - HP 9000 VME CPU board. (742, 743 Anole)
- HP-UX v10.xx (Hewlett-Packard Co.)
  - HP 9000 VME CPU board. (742, 743 Anole)
- Motorola UNIX System V/88 Release R32V3.
  - Motorola MVME-187 Single Board Computer (M88100).

While all of these platforms share a common library interface, the device driver and installation procedure for these platforms varies significantly. Please read the section that relates to your system environment.

The distributed file "`README`" contains any information that is newer than the current revision of the DiSPATCH documentation. Please read it before proceeding with the installation.

3.1 SunOS v4.1.x

The "`dispatch/driver/sunos`" directory of the DiSPATCH distribution contains the "mmidsp" device driver for SunOS v4.1.x systems. The file "`README.install`" in that directory contains a text version of these installation instructions.

If you have never installed a device driver or recompiled the SunOS kernel, additional help from a nearby guru or Vigra may be necessary. These instructions are only a brief overview, and your system may have special requirements.

This DiSPATCH "mmidsp" device driver is responsible for the following:

- Mapping the VME board into user memory via `mmap()`.
- Receiving DiSPATCH messages from a single DSP on the MMI board via `read()`.

When the user opens an mmidsp device, the driver enables interrupts and begins handling responses from one DSP. Each incoming word is held in a queue until read by the user process via `read()`. Each DSP on the MMI board gets a separate device handle.

Note that the device driver does not initialize the DSP. It is expected to be initialized by the user process. The user process is also responsible for all writes to the DSP host port, which are done directly.

This driver is required by the DiSPATCH programming library.

These instructions assume the new kernel will be named “MMIDSP”. If you choose a different name, please adjust the examples accordingly.

3.1.1 Kernel Architecture

To use this device driver, your SunOS machine must be one of the “sun4c”, “sun4m”, or “sun4rf” architectures. If you are unsure of your architecture, run “`arch -k`” and verify that it reports one of the above.

Where these instructions refer to an architecture, such as in a filename, the generic name `sun4x` is used to represent `sun4m`, `sun4c`, or `sun4rf`. Please substitute the name that is appropriate to your target system.

3.1.2 Installation Summary

To install the mmidsp driver, the following steps must be performed as root. Each of these steps is explained in detail below.

1. Copy the driver source files to “`/usr/kvm/sys/vigra`”.
2. Verify that the driver is set for the correct mapping call.
3. Add one entry for each DSP to “`/usr/kvm/sys/sun4x/conf/MMIDSP`”.
4. Add the driver entry points to “`/usr/kvm/sys/sun/conf.c`”.
5. Add an entry for the driver to “`/usr/kvm/sys/sun4x/conf/files`”.
6. Run “`/etc/config`” to configure the new kernel.
7. Build, install, and boot the new kernel.
8. Make the device entries in the “`/dev`” directory.
9. Test the driver with “`mmi-test`”.

3.1.3 Install the Source Files

First, create a directory called "`/usr/kvm/sys/vigra`". Copy these two files from the driver distribution directory into the new directory:

```
vme-mmidsp.c
dsp_defs.h
```

3.1.4 Select the Mapping Function

Since most Sun Sparcstation machines normally have only SBus slots, third-party vendors have developed VME extensions to the hardware and kernel. These custom extensions determine how the mmidsp driver will map the memory on the VME board.

The mmidsp driver requires that these kernel extensions be installed, since it uses the "`xx_map()`" routines provided by the computer manufacturer. If you have not yet built a SunOS kernel to support your VME system, please do so before installing the mmidsp driver.

Edit "`vme-mmidsp.c`" and uncomment the `#define` appropriate for your VME system. For example, if you are using a SUN4/330, uncomment the line that reads:

```
 /* #define SUN4_330 */
```

3.1.5 Create the Kernel Configuration File

To add the "mmidsp" driver to your existing kernel configuration, copy your kernel's configuration file to "`/usr/kvm/sys/sun4x/conf/MMIDSP`".

This new configuration file will be modified to include the "mmidsp" driver.

Each MMI board installed in the VME bus requires the following entries in the configuration file:

- One entry for the DRAM on the board.
- One entry for *each DSP on the MMI board*.

The exact contents of the configuration entries will vary with board models and base addresses. Look at the files "`MMIDSP.210`" and "`MMIDSP.420`" for configuration examples.

First, place the following line before any MMI entries to tell the config program about the new device driver:

```
device-driver mmidsp
```

Then, make one entry for the DRAM on the board, at the base address of the board:

```
device mmidsp0 at vme32d32 ? csr 0x81000000
```

Now make one entry for each DSP on the board, at the DSP base address. You must also list the VME interrupt priority and vector to use for each DSP. Be sure to select a *unique* interrupt vector for each DSP. The DSP interrupt vectors must not be used by any other devices.

To compute the base address of each DSP, add one of the following constants to the VME base address of the board:

- MMI-420 and MMI-4210:

DSP A	`0x3FFE80`
DSP B	`0x3FFEC0`
DSP C	`0x3FFF00`
DSP D	`0x3FFF40`

- MMI-210 (1 meg):

DSP A	`0x0FFE40`
DSP B	`0x0FFE80`

- MMI-210 (4 meg):

DSP A	`0x3FFE40`
DSP B	`0x3FFE80`

- MMI-105 (1 meg):

DSP A	`0x0FFC40`

The interrupt priority is jumper-selectable, as described in the MMI Owner's Manual. The factory default setting is level 1.

A complete example configuration for an MMI-4210 (four DSPs) is as follows:

```
#
# MMI-4210 (four DSPs) installed at base address 0x81000000, priority 1:
#
```

```
device-driver mmidsp
device mmidsp0 at vme32d32 ? csr 0x81000000
device mmidsp1 at vme32d32 ? csr 0x813FFE80 priority 1 vector mmidsp_intr 0xd0
device mmidsp2 at vme32d32 ? csr 0x813FFEC0 priority 1 vector mmidsp_intr 0xd1
device mmidsp3 at vme32d32 ? csr 0x813FFF00 priority 1 vector mmidsp_intr 0xd2
device mmidsp4 at vme32d32 ? csr 0x813FFF40 priority 1 vector mmidsp_intr 0xd3
```

3.1.6 Add Driver Entries to "`conf.c`"

The following code must be added to the first part of "`/usr/sys/sun/conf.c`":

```
#include "mmidsp.h"
#if NMMIDSP > 0
extern int mmidsp_open(), mmidsp_close();
extern int mmidsp_read(), mmidsp_select();
extern int mmidsp_mmap();
#else
#define mmidsp_open             nodev
#define mmidsp_close            nodev
#define mmidsp_read             nodev
#define mmidsp_select           nodev
#define mmidsp_mmap             nodev
#endif
```

Later in the same file, this code must be inserted at the end of the `cdevsw[]` array:

```
    {
        mmidsp_open,    mmidsp_close,   mmidsp_read,    nodev,          /*xx*/
        nodev,          nodev,          mmidsp_select,  mmidsp_mmap,
        0,              0,
    },
```

Fill in and note the major number to the right of this entry when you edit `cdevsw[]`; this will be the major number for all mmidsp device nodes.

3.1.7 Files

To tell the kernel configuration about the location of the files "`vme-mmidsp.c`" and "`dsp_defs.h`", edit the file "`/usr/kvm/sys/sun4x/conf/files`". Some system configurations may place this file elsewhere.

Add a line to the end of “`files`”, indicating where to find the MMI driver source, relative to “`/usr/sys`”. If the files were copied to a “`/usr/sys/vigra`” directory, add this line:

```
vigra/vme-mmidsp.c              optional mmidsp device-driver
```

3.1.8 Configure the Kernel

As root, run:

```
/etc/config MMIDSP
```

from the “`/usr/kvm/sys/sun4x/conf`” directory to configure the new kernel.

3.1.9 Build, Install, and Boot the New Kernel

Change directory to “`/usr/kvm/sys/sun4x/MMIDSP`” and run “`/bin/make`” to build the new kernel. Fix any compilation errors before continuing.

After making a backup copy of the old kernel, install the new kernel in the root directory, like this:

```
cp /vmunix /vmunix.nommi
cp /usr/kvm/sys/sun4x/MMIDSP/vmunix /
```

Then reboot the machine to run the new kernel:

```
/etc/reboot
```

3.1.10 Make the Device Handles in “`/dev`”

Each MMI board has one or more DSPs, each of which needs its own driver handle, since they send messages independently. Only one mapping device handle is needed for each board, since all DSPs share a common DRAM address space.

The device handles for a two-DSP board (such as the MMI-210) are as follows:

```
/dev/mmidsp0_map
/dev/mmidsp0_A
/dev/mmidsp0_B
```

The first device ("`*_map`"), is the `mmap()` handler, and is used only for memory-mapping. There are no DSPs directly associated with this handle. Each MMI board has one `mmap()` handler, whose entry in the kernel configuration file has the base address of the board.

The remaining device handles ("`*_A`", "`*_B`",...) represent the DSPs on the board. There will be one such handle for each DSP on the MMI board. These entries in the config file use the base address of the DSP they represent.

The minor number for the device handles is simply the unit value for the corresponding entry in the "`config`" file. For example, the first entry (`mmidsp0`) will get the minor number 0. The second entry (`mmidsp1`) uses the minor number 1.

To create devices for the example configuration from Section 3.1.5, make the following device entries in "`/dev`" using mknod, where `'XX'` represents the major device number for the mmidsp device.

```
mknod mmidsp0_map c XX 0
mknod mmidsp0_A c XX 1
mknod mmidsp0_B c XX 2
mknod mmidsp0_C c XX 3
mknod mmidsp0_D c XX 4
```

Be sure to set the file permissions appropriately.

3.1.11 Test the Driver

Verify that the driver is working properly by running the "`mmi-test`" program in the "`dispatch/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation.

The first argument to the "`open_board`" command should be "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or "`MMI-4210`" for MMI-420/MMI-4210 models.

For example, type the following commands to turn DSP A's LED on and off on an MMI-4210:

```
% mmi-test

       *** MMI Test ***
```

```
Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4210 /dev/mmidsp0
MMI-4210 #1 is ready.

MMI-4210 #1 [DSP A]: boot

MMI-4210 #1 [DSP A]: led on

MMI-4210 #1 [DSP A]: led off

MMI-4210 #1 [DSP A]:
```

If the LED turned on and off correctly, the driver is operational. For more information on the `mmi-test` program, see Section 5.1.

3.2 Solaris 2.x

The *DiSPATCH Host Support Package for Solaris 2.x* supports Sun Solaris 2.4 (SunOS 5.4) and Solaris 2.5 (SunOS 5.5). Contact Vigra for information regarding using DiSPATCH on newer releases of Solaris. This DiSPATCH package supports SPARC-based VME hardware platforms only.

3.2.1 Installation Summary

To install the DiSPATCH support software and “`mmidsp`” device driver, the following steps must be performed as root. Each of these steps is explained in detail below.

1. Configure and install the MMI boards.
2. Check available disk space.
3. Install the “`VIGRAmmi`” package from the distribution media.
4. Run “`mmi-config`” and configure the driver.
5. Test the driver with “`mmi-test`”.

3.2.2 Configure and Install the MMI Boards

For best results, the Solaris device driver requires that the MMI audio boards be installed into the system before the software is loaded. If the software is installed before the MMI hardware is in place, it will be necessary to reboot the machine and possibly reconfigure the device driver.

Before installing the MMI VME cards, note the model number, VME base address and interrupt priority level for each board. Make sure there are no address conflicts between boards in the system. The MMI boards utilize the IACK line on the VME bus, so be certain that any empty VME slots are correctly jumpered to pass IACK through.

3.2.3 Check Available Disk Space

The Solaris DiSPATCH software package (VIGRAmmi) requires approximately 35 megabytes of disk space. Before installing the package, verify that there is adequate free space on the target filesystem.

3.2.4 Install the VIGRAmmi Package

The software is distributed in standard Solaris “package” format. To install the software, become root and use the “pkgadd” utility provided with Solaris.

To load the DiSPATCH software from tape media, use a command similar to the following:

```
pkgadd -d /dev/rmt/0
```

If you have acquired DiSPATCH software in a single-file format, such as from Vigra’s FTP site, use this command to install from the file:

```
pkgadd -d dispatch-1.80-Solaris.pkg
```

It may also be necessary to uncompress the file prior to installation.

To install from CD-ROM, load the CD-ROM into the system, verify that it is mounted, and run pkgadd:

```
pkgadd -d /cdrom
```

Solaris will find one available package, as shown below.

```
The following packages are available:
  1  VIGRAmmi     DiSPATCH MMI Audio Support Software
                  (sparc) 1.80

Select package(s) you wish to process (or 'all' to process
all packages). (default: all) [?,??,q]:
```

Select the `VIGRAmmi` package to install the DiSPATCH device driver and host support software.

To install the package in a directory other than the default "`/opt`", use the `-R` option with "`pkgadd`". See the "`pkgadd`" documentation for details.

3.2.5 Driver Configuration

After the package installation has finished, it is necessary to configure the device driver to match your installed audio hardware. Solaris can not auto-detect VME devices, so you must create a file describing your system configuration. The "`mmi-config`" program is provided to assist you in making this file.

As root, run the interactive script "`/opt/VIGRAmmi/mmi-config`".[1] This program will prompt for board information and create the configuration file "`mmidsp.conf`" for the kernel.

For each installed board, you will need to provide the following information:

- The starting VME interrupt vector.
- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Assigning the VME Interrupt Vectors

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`mmi-config`" program. The user must specify

[1]The pathname will be different if you specified a base directory other than the default "`/opt`" during package installation.

the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xD0` where possible.

Selecting the board to be configured

A numbered list of MMI audio board models will be listed to the screen. To add configuration information for a board, select the model name that corresponds to your board. To end the configuration, enter “`0`”.

Entering the base VME Address of the board

Each MMI board occupies a VME address range specified by the board’s jumper settings. The address entered into the program must match the settings on the board. This address should be entered in hexadecimal form (i.e. `0x81000000`).

Entering the VME Interrupt Priority Level of the board

Each DiSPATCH board is set to a VME Interrupt Priority Level based on the board jumper settings. The priority level specified must match the settings on the board. The value should be in the range of 1 to 7; the factory default setting is level 1. Consult the MMI Owner’s Manual for information regarding changing this setting.

Confirming your selections

Once the settings for a board are entered, they will be displayed for verification. If the information needs to be changed, enter ‘`N`’, and enter the correct information. If the information is correct, enter ‘`Y`’, and the board’s configuration will be added to the driver information file.

After “`mmi-config`” is finished, it will copy the `mmidsp` device driver and configuration file to the kernel driver directory and load the driver into the kernel. This should generate several messages to the system console describing the new devices. If any errors are reported, stop and verify the jumper settings and backplane configuration.

If the boards were not installed as described to “`mmi-config`”, power down the system, install the MMI boards, and reboot using the “`-r`” option.

If MMI boards are added, removed or reconfigured at a later date, simply run the "`mmi-config`" program again and provide the new configuration details.

3.2.6 Test the Installation

Verify that the driver is working properly by running the "`mmi-test`" program in the "`/opt/VIGRAmmi/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation.

The first argument to the "`open_board`" command should be the board model name: "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or "`MMI-4211`" for that model.

For example, type the following commands to turn DSP A's LED on and off on an MMI-4211:

```
% mmi-test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4210 #1 is ready.

MMI-4210 #1 [DSP A]: boot

MMI-4210 #1 [DSP A]: led on

MMI-4210 #1 [DSP A]: led off

MMI-4210 #1 [DSP A]:
```

If the front-panel LED turned on and off correctly, the driver is operational. For more information on the `mmi-test` program, see Section 5.1.

3.3 VxWorks

The DiSPATCH support software for VxWorks is supplied in both source and binary object (.o) files. There is no literal "device driver" under VxWorks because modules can attach interrupts to standard C functions.

The VxWorks DiSPATCH host support software provides calls to emulate the mapping and opening of the DSP under Unix. This allows the library to have very little special code for VxWorks platforms. Most VxWorks DiSPATCH applications should compile and run under supported Unix systems as well.

The distribution media contains three directories pertaining to VxWorks:

"`dispatch/vxworks/common`" This directory contains files that are shared by all supported VxWorks targets.

"`dispatch/vxworks/mv167`" This directory contains source and object files that pertain only to the Motorola MVME–167 (68040) board.

"`dispatch/vxworks/sun2ce`" These object and source files pertain only to the Force SPARC CPU-2CE.

3.3.1 Building the VxWorks package

All source code for the DiSPATCH library and VxWorks support is provided with the host support software. The pre-built modules for each supported platform are also included, but they may need to be modified to suit your system configuration (see Section 3.3.2).

To compile the VxWorks support software, begin by changing the current working directory to the source directory pertaining to your target board (i.e. "`dispatch/vxworks/mv167`" for the MVME–167). Then run "`make`" to build the object modules for that target.

Note that the DiSPATCH Makefiles use several of the same environment variables as the standard VxWorks development environment. These variables must be set as per the directions given in the VxWorks documentation. Specifically, the following variables must be set to reflect your VxWorks configuration:

VX_VW_BASE: The base path of the VxWorks distribution. For example, "`/usr/vw`".

VX_HOST_TYPE: The Wind Rivers type name for your host CPU. For example, "`sun4`".

See the comments near the top of the Makefiles for details concerning how the program paths are constructed, especially if your environment uses a non-standard VxWorks arrangement.

The VPATH feature

The included Makefiles for VxWorks make frequent use of the VPATH control variable. If your "make" program does not support this feature, please configure, build and install the free GNU "make" program, included with the software package in the directory "dispatch/make-3.68".

3.3.2 Configuration

Before the library can be built and loaded into VxWorks, it must be configured for the boards installed in your system. The configuration information is contained in a structure near the top of "dispatch/lib/sys_vxworks.c".

The definition of INTERRUPT_LEVEL must reflect the interrupt priority level of all MMI boards installed in the system. The factory default jumper setting is IRQ level 1.

The structure mmi_vx_config[] contains one entry for each DSP present in the system. MMI boards have from one to four DSPs per board, so each board may require multiple entries in the configuration structure. An example configuration for one MMI-4211 and one MMI-210-4 is shown below:

```
static struct board_desc mmi_vx_config[] =
{
  /* MMI-4211 */
  { "/dev/mmidsp0_A", 0x02000000, 0xe0 },
  { "/dev/mmidsp0_B", 0x02000000, 0xe1 },
  { "/dev/mmidsp0_C", 0x02000000, 0xe2 },
  { "/dev/mmidsp0_D", 0x02000000, 0xe3 },

  /* MMI-210-4 */
  { "/dev/mmidsp1_A", 0x02400000, 0xe4 },
  { "/dev/mmidsp1_B", 0x02400000, 0xe5 },
}
```

Each configuration entry in this structure has three parts:

Device filename: Applications that use the DiSPATCH library refer to each DSP by a symbolic name. This is equivalent to the "device node" filename under Unix. A standard device name for a DiSPATCH DSP is "`/dev/mmidsp0_B`". This specifies the second DSP (B) on the first "mmidsp" board installed (0).

Each DSP must have a unique name. The actual name is arbitrary, but using the standard naming structure from Unix will allow for application compatibility.

Board base address: This is the physical VME base address for the MMI board, as set by the on-board jumpers. All DSPs on one MMI board share a common base address.

It is important that the MMI board be configured to an address that is accessible by the VxWorks kernel as an A32/D32 VMEbus region. By default, the VxWorks kernel contains only one small VME window. The default ranges are listed below:

Force CPU-2CE	`0x00000000 - 0x00FFFFFF`
MVME–167	`0x02000000 - 0x02FFFFFF`

The jumpers on the MMI board must be configured to locate the board within the allowed address range. Alternatively, the default VME mappings in the kernel can be modified and recompiled to allow other address ranges. Consult the VxWorks documentation for details on rebuilding the kernel.

VME interrupt vector: Each DSP must be assigned a unique interrupt vector number that is not shared with any other VME devices.

When the configuration structure has been edited to reflect your system environment, the VxWorks program files must be recompiled.

To rebuild the code, go into the directory pertaining to your VxWorks target CPU board and run the command "`make`" (see Section 3.3.1). If the build is successful, there will be no errors or warnings and the directory will contain the new object files.

3.3.3 The Included Object Files

All of the DiSPATCH host application support is contained in the module "`VX-libmmi.o`". This file contains the DiSPATCH library, VxWorks interface, and firmware binaries for all MMI boards. This module must be loaded before any DiSPATCH applications can be run.

Several other modules are provided which are not part of the library, but may prove useful to the application designer:

“`VX-test.o`” This module provides the external function `mmi_test()`. Calling this function from the shell will run the interactive MMI-Test program. See Section 3.3.4.

“`VX-play.o`” This module supplies the external function `play()`. This is an example function that uses the DiSPATCH library to play an audio file. Several global variables control the operation of the `play()` function. Call the routine `play_usage()` for a list of the relevant variables and their current settings.

“`VX-beep.o`” This is very simple module to generate a short tone on the MMI board. The external routine is called `beep()`. The function `beep()` is usually invoked directly from the VxWorks shell as a diagnostic function. An example invocation is:

```
beep ("MMI-4211", "/dev/mmidsp0", 0)
```

The arguments are, in order:

Model name: A string name for the MMI model (i.e. “`MMI-4211`”).

Device name: A string device name specifying the MMI board to use (i.e. “`/dev/mmidsp0`”).

Channel: An integer value indicating which DSP on the board to use (i.e. “`0`” for DSP A).

3.3.4 Testing the Audio System

After DiSPATCH has been installed in the VxWorks system, some simple diagnostics should be run to verify that the board is correctly installed and the system is properly configured.

Call “`mmi_show_config()`”

Begin by calling the diagnostic function `mmi_show_config()` from the VxWorks shell. This routine will list all the DSPs currently configured into the DiSPATCH library. If this list does not match the intended configuration, verify that the contents of the “`sys_vxworks.c`” configuration structure is correct. If any entries in the "**Virtual Addr**" column show "`None`" or the "**Probe**" column lists "`Fail!`", then the system is not configured properly and will not work.

Note that the Force Sun-2CE does not probe the VMEbus reliably and may show "`OK`" when the board is not actually installed. This is a known VxWorks problem that may be resolved in future releases.

Run "`mmi_test()`"

An interactive interface to DiSPATCH is provided in the module "`VX-test.o`". This module provides the external routine `mmi_test()`. Call this function from the shell to invoke the test program.

By executing a few commands, you can initialize the MMI board, boot a DSP, execute a few DiSPATCH commands, and close the board. An example dialogue to blink a front-panel LED is shown below. Be sure to use the correct model type name for your installation ("MMI-4211" is used in the example).

```
-> ld < VX-libmmi.o
value = 16260912 = 0xf81f30
-> ld < VX-test.o
value = 16360892 = 0xf9a5bc
-> mmi_test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4210 #1 is ready.

MMI-4210 #1 [DSP A]: boot

MMI-4210 #1 [DSP A]: led on

MMI-4210 #1 [DSP A]: led off

MMI-4210 #1 [DSP A]:
```

3.4 IRIX v4.0.x

The “`dispatch/driver/sgi`” directory of the DiSPATCH distribution the device driver for SGI systems running IRIX v4.0.x.

This device driver is responsible for the following:

- Mapping the VME board into user memory via mmap().
- Receiving DiSPATCH messages from a single DSP on the MMI board via `read()`.

When the user opens an `mmidsp` device, the driver enables interrupts and begins handling responses from one DSP. Each incoming word is held in a queue until read by the user process via `read()`. Each DSP on the MMI board gets a separate device handle.

Note that the device driver does *not* initialize the DSP. It is expected to be initialized by the user process. The user process also writes to the DSP host port directly.

This driver is required by the DiSPATCH programming library.

3.4.1 Installation

To install the mmidsp driver, the following steps must be performed as root. Each of these steps is explained in detail below.

1. Build the driver binary “`mmidsp.o`” or use the one provided.
2. Copy the driver binary into “`/usr/sysgen/boot`”.
3. Edit the “`system.mmidsp`” file and append it to “`/usr/sysgen/system`”.
4. Copy the “`mmidsp`” file into “`/usr/sysgen/master.d`”.
5. Rebuild the kernel.
6. Boot the new kernel.
7. Create the device nodes.
8. Test the driver with “`mmi-test`”.

3.4.2 Building the Driver

A compiled binary for the device driver is provided with the source code. If possible, this binary should be used as-is. The filename is “`mmidsp.o`” in the “`dispatch/driver/sgi`” directory.

However, if it is necessary to recompile the device driver the source files are included as “`sgi-mmidsp.c`” and “`dsp-defs.h`”. The Makefile will rebuild the binary if necessary.

3.4.3 Install the Driver Binary

The binary file named “`mmidsp.o`” from Section 3.4.2 must be copied into the directory “`/usr/sysgen/boot`” for inclusion into the kernel.

3.4.4 VME Vector Configuration

The file “`/usr/sysgen/system`” contains the kernel configuration information for VME devices. This file describes the VME address, interrupt priority, and interrupt vector of the MMI board.

Example entries are included in “`mmi210.system`” and “`mmi420.system`” for the MMI-210-1 and MMI-4211/MMI-4210 audio boards, respectively. Use these files as guidelines for your own configuration.

You must add one VECTOR line to “`/usr/sysgen/system`” for each DSP present on the system. Each installed MMI-210 has two DSPs, while an MMI-420 has four.

An example VECTOR entry is shown below:

```
VECTOR: module=mmidsp vector=0xd0 ipl=1 unit=0 base=0xDB000000 base2=0xDB0FFE80
```

module: This name must be “mmidsp” because this is the name of the driver.

vector: This can be any available (unused) VME interrupt vector. Vectors 0xd0 through 0xdf are reserved by SGI for customer boards. Each VECTOR line MUST have its own interrupt vector value. No interrupt vector sharing is allowed.

ipl: This value should reflect the Interrupt Priority Level jumper setting on the MMI board. The boards are shipped with a default setting of 1.

unit: The first mmidsp entry must be unit number zero, and each following entry should increment this value. Every entry must have a unique unit number, and there should be no skipped values.

base: This is the kernel address of the MMI board. It is computed based on the VME physical address. All DSPs on one MMI board share the same “base” value. This VME physical address is jumper-selectable. See the MMI User’s Manual for details on the jumper settings. Read the information in “`/usr/sysgen/system`” for details on computing the SGI kernel base address from the VME physical address.

base2: This is the Host Port address of each DSP. Each DSP (and VECTOR entry) has a unique base2 address. The DSP base address (base2) is the VME base address plus a constant offset. The offset of each DSP is listed below. See the “Hardware Definitions” chapter of the DiSPATCH User’s Manual for details.

- MMI-420 and MMI-4210:

DSP A	`0x3FFE80`
DSP B	`0x3FFEC0`
DSP C	`0x3FFF00`
DSP D	`0x3FFF40`

- MMI-210 (1 meg):

DSP A	`0x0FFE40`
DSP B	`0x0FFE80`

- MMI-210 (4 meg):

DSP A	`0x3FFE40`
DSP B	`0x3FFE80`

- MMI-105 (1 meg):

DSP A	`0x0FFC40`

3.4.5 The Driver Configuration File

The device driver configuration file is provided as “`mmidsp`”. This file defines the device major number to be “61” by default. If this major number is already in use on your system, set the value in this file to be one that is not used. Do not edit anything else in this file.

Copy the “`mmidsp`” file into the “`/usr/sysgen/master.d`” directory.

3.4.6 Rebuild the Kernel

The kernel must now be recompiled to include the new mmidsp driver. As root, execute the following command:

```
/etc/autoconfig -f
```

This recompiles and installs the new kernel.

3.4.7 Boot the New Kernel

Reboot the system to begin using the new kernel. This can be done with the "`reboot`" command or "`init 6`".

3.4.8 Create Device Nodes

To access the driver, there must be device descriptor files in the "`/dev`" directory. The major number for these device files is the one assigned in "`/usr/sysgen/master.d/mmidsp`". The minor number for each device is the same as the unit number defined on the VECTOR line of "`/usr/sysgen/system`".

The device file names are of the format "`mmidsp%d_%c`", where the number indicates which MMI board, and the letter (A, B, C, or D) indicates which DSP on the board. For example, "`mmidsp0_A`" represents DSP A on the first MMI board. This device would have unit number zero. "`mmidsp0_B`" is DSP B on the first board, and has unit number 1. The second MMI board (if any) would have names beginning with "`mmidsp1_`".

For example, to create device entries for the example configuration given in "`mmi420.system`" and "`mmidsp`", make the following device entries in "`/dev`" using mknod. The major number here is 61, as defined in "`mmidsp`".

```
mknod /dev/mmidsp0_A c 61 0
mknod /dev/mmidsp0_B c 61 1
mknod /dev/mmidsp0_C c 61 2
mknod /dev/mmidsp0_D c 61 3
```

In addition, there must be an "`mmidsp0_map`" device. This device is used by the library to map the board into user space. On SGI systems, this device is identical to that named "`mmidsp0_A`". To create it, use the following link command:

```
ln /dev/mmidsp0_A /dev/mmidsp0_map
```

All told, there should be one device entry for each DSP, and one "`*_map`" entry for each MMI board. Be sure to set the file permissions appropriately for your environment. For unrestricted access to the audio boards, use this command:

```
chmod 666 /dev/mmidsp*
```

3.4.9 Test the Driver

Verify that the driver is working properly by running the "`mmi-test`" program in the "`dispatch/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation.

The first argument to the "`open_board`" command should be "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or MMI-4210 for MMI-420/MMI-4210 models.

For example, type the following commands to turn DSP A's LED on and off on an MMI-4210:

```
% mmi-test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4210 /dev/mmidsp0
MMI-4210 #1 is ready.

MMI-4210 #1 [DSP A]: boot

MMI-4210 #1 [DSP A]: led on

MMI-4210 #1 [DSP A]: led off

MMI-4210 #1 [DSP A]:
```

3.5 IRIX v5.2

The “`dispatch/driver/irix-5`” directory of the DiSPATCH distribution contains the “MMIDSP” device driver for SGI systems running IRIX Release 5.2 and compatible versions. Most filenames mentioned below are in this directory.

This device driver is responsible for the following:

- Mapping the VME board into user memory via `mmap()`.
- Receiving DiSPATCH messages from each DSP on the MMI board via `read()`.

When the user opens an `mmidsp` device, the driver enables interrupts and begins handling responses from one DSP. Each incoming word is held in a queue until read by the user process via `read()`. Each DSP on the MMI board gets a separate device handle.

Note that the device driver does *not* initialize the DSP. It is expected to be initialized by the user process. The user process also writes to the DSP host port directly.

This driver is required by the DiSPATCH programming library.

3.5.1 Installation

To install the mmidsp driver, the following steps must be performed as root. Each of these steps is explained in detail below.

1. Select an unused major number.
2. Edit “`mmidsp.sm`” to reflect your system hardware configuration.
3. Build and install the driver using `make`.
4. Create the device handles.
5. Rebuild the kernel.
6. Boot the new kernel.
7. Test the driver with “`mmi-test`”.

3.5.2 Select a Major Number

The mmidsp device driver requires a unique character device major number. Find an unused major number by checking the "/dev" directory. The preconfigured number, 61, is not used by standard Irix 5.2 system and is probably available for use by mmidsp.

If you select a major number other than 61, edit the definition of MAJOR in "dispatch/driver/irix-5/Makefile" to reflect your chosen number. Also edit "master.mmidsp" to use the same number.

3.5.3 Configure "mmidsp.sm" for your MMI boards

The file "/dispatch/driver/irix-5/mmidsp.sm" contains the kernel configuration information for MMI boards. It describes the VME address, interrupt priority, and interrupt vector of each board and must match the hardware settings.

You must have one VECTOR line in "mmidsp.sm" for each DSP present on the system. Each installed MMI-210 has two DSPs, while an MMI-4211 has four. After the DSP entries, you must have one VECTOR line for the RAM space on each board. For example, an MMI-4211 will have one VECTOR for each of:

- DSP A
- DSP B
- DSP C
- DSP D
- RAM Space (4 Megabytes)

Example entries are included in "mmidsp.sm" for the MMI-4211 and MMI-210-4 audio boards. Use these listings as guidelines for your own configuration.

An example VECTOR entry for a DSP is shown below. Note that the entire entry must be on **one continuous line** in the file.

```
VECTOR: bustype=VME module=mmidsp adapter=0 ipl=1 ctlr=0 iospace=(A32NP,
  0x153ffe80,0x40) probe_space=(A32NP,0x153ffe80,1)
```

bustype: This will always be "VME", since all MMI boards plug into the VME bus.

module: This name must be “mmidsp” because this is the name of the driver.

adapter: This describes which VMEbus the kernel is to use. Most systems have only one VMEbus, so this value is usually 0.

ipl: This value should reflect the Interrupt Priority Level jumper setting on the MMI board. The boards are shipped with a default setting of 1. The VECTOR entry for the RAM space contains no `ipl` field, since it doesn’t use interrupts.

ctlr: The first VECTOR entry must be unit number zero, and each following entry should increment this value. Every entry must have a unique ctlr number, and there should be no skipped values. Note that the DSP entries must come before the RAM entry, as shown in the example configurations.

iospace: This declares the VME memory region (space, address, and length) of the RAM or DSP entry. Use `A32NP` for the VME space (first value) for all VECTOR lines.

The addresses are determined by the jumper settings, and fixed offsets. See the MMI User’s Manual for details on the jumper settings.

For the RAM VECTOR entry, the address and size reflect the physical VME base address and total size of the board, respectively. For example, any four-megabyte board would have one VECTOR with an `iospace` defined like this: “`iospace=(A32NP,0x15000000,0x400000)`”.

For each DSP VECTOR entry, the definition of `iospace` declares the VME base address and size of each DSP host port. The size (third value) is always `0x40`, and the DSP base address (second value) is the VME base address plus a constant offset. The offset of each DSP is listed below. See the “Hardware Definitions” chapter of the DiSPATCH User’s Manual for details.

- MMI-420 and MMI-4210:

DSP A	`0x3FFE80`
DSP B	`0x3FFEC0`
DSP C	`0x3FFF00`
DSP D	`0x3FFF40`

- MMI-210 (1 meg):

DSP A	`0x0FFE40`
DSP B	`0x0FFE80`

- MMI-210 (4 meg):

DSP A	`0x3FFE40`
DSP B	`0x3FFE80`

- MMI-105 (1 meg):
  DSP A `0x0FFC40`

probe_space: This specifies the VME region to be probed during boot. The first two values (space and address) should be the same as those in `iospace` definition. The third value, length, should always be `1`.

3.5.4 Build and Install the Driver

As root, execute `make` in the "`dispatch/driver/irix-5`" directory to compile the driver code and copy all configuration files into the standard system directories There should be no warnings or errors during the execution of `make`.

Alternatively, you may study the `Makefile` and perform the tasks manually as root.

3.5.5 Create the Device Handles

There must be one device file in "`/dev`" for each VECTOR entry in "`mmidsp.sm`". For single-board installations, the makefile provides two targets to help create these entries.

As root, execute "`make devs-4`" if you have one quad-DSP board installed, or "`make devs-2`" if you have an MMI-210 model installed.

Be sure to set the file permissions appropriately for your environment. For unrestricted access to the audio boards, use this command:

```
chmod 666 /dev/mmidsp*
```

Multiple Boards Installed in One System

If you have more than one board in a single system, you must create additional device handles by hand. Continue as shown in the makefile, creating one device node for each VECTOR defined in "`mmidsp.sm`". The minor number for each device is the same as the ctlr number defined on the VECTOR line.

The device file names are of the format "`mmidsp%d_%c`", where the number indicates which MMI board, and the letter (A, B, C, or D) indicates which DSP on the board. For example, "`mmidsp0_A`" represents DSP A on the first MMI board. This device would have ctlr number zero. "`mmidsp0_B`" is DSP B on the first board, and has ctlr number 1. The second MMI board (if any) would have names beginning with "mmidsp1_".

All told, there should be one device entry for each DSP, and one "*_map" entry for each MMI board.

3.5.6 Rebuild the Kernel

The kernel must now be recompiled to include the new mmidsp driver. As root, execute the following command:

```
/etc/autoconfig -f
```

This recompiles and installs the new kernel.

3.5.7 Boot the New Kernel

Reboot the system to begin using the new kernel. This can be done with the "reboot" command or "init 6".

3.5.8 Test the Driver

Verify that the driver is working properly by running the "mmi-test" program in the "dispatch/apps" directory. Use the "led" command to test the firmware, driver, and library operation.

The first argument to the "open_board" command should be "MMI-210-1" for 1-meg MMI-210 boards, "MMI-210-4" for 4-meg MMI-210 boards, or "MMI-4211" for the MMI-4211 model.

For example, type the following commands to turn DSP A's LED on and off on an MMI-4211:

```
% mmi-test

        *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $
```

```
MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4211 #1 is ready.

MMI-4211 #1 [DSP A]: boot

MMI-4211 #1 [DSP A]: led on

MMI-4211 #1 [DSP A]: led off

MMI-4211 #1 [DSP A]:
```

3.6 OS-9

DiSPATCH supports version 2.4 of the OS-9 operating system from Microware Systems Corp. The source code and makefiles require that the Ultra-C compiler (v1.1.2 or newer) be installed on the development system as the default compiler.

3.6.1 Functional Overview

Each DSP in the DiSPATCH system has two controlling processes:

- A system-state daemon process to receive and process messages.
- The user-state application process to process audio data.

System-State daemon

Before any applications can use the DiSPATCH library, one system-state daemon must be running for each DSP to be used. Each daemon handles the VME interrupts and incoming messages for one DSP. These processes are usually executed from the system startup file, and run in the background at all times. They remain dormant until the DSPs become active.

The system-state process is managed by the program “`os9_mmidsp`”, located in the “`dispatch/apps`” directory. This program is built by the included makefiles and made system-state automatically. For this reason, “`make`” must be run by the super-user, or the system-state attribute will not be set.

The "os9_mmidsp" process takes the place of the "device driver" used on Unix-like platforms. All incoming messages are handled by the process, while the application and library perform all DSP control operations.

User-state application

The user-state application is responsible for initializing and controlling the operation of the DiSPATCH DSPs. Most applications should be written to use the high-level control functions provided by the DiSPATCH Programming Library. Each application connects to a "os9_mmidsp" process to receive messages from the DSP.

3.6.2 Installation

To install the DiSPATCH package, the following steps must be performed as superuser. Each of these steps is explained in detail below.

1. Extract the files from the distribution media.
2. Configure the source files by running "make configure".
3. Build the library and applications by running "make".
4. Add the "os9_mmidsp" processes to the system startup file.
5. Reboot the OS-9 system.
6. Test the DiSPATCH system with the provided applications.

3.6.3 Extract DiSPATCH files

The DiSPATCH files are shipped in OS-9 "fsave" format. To extract the files, change to a directory (using "chd") in which you have write permission. Execute the following command to automatically extract the tree from the default tape device to the current directory:

```
frestore -s
```

This will create the top-level directory called "`dispatch`" and place all files within it. To specify an alternate tape device or to control other "`frestore`" options, please consult the Microware OS-9 manuals.

At this time, please read the file named "`dispatch/README`" for any special information not included in the latest version of the documentation.

3.6.4 Configuration

Before building the DiSPATCH source files, two files must be created to describe the hardware configuration of your system. These two files are the daemon startup script ("`apps/mmi_startup`") and the library hardware configuration file ("`lib/sys_os9_config.c`").

Both of these files are created automatically by the interactive "`configure`" program in the "`dispatch/config`" directory. To build and run this program, run "`make configure`" from the top-level "`dispatch`" directory.

If the program builds successfully, it will run and prompt for board information. For each installed board, you will need to know the following:

- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`configure`" program. The user must specify the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xE0` where possible.

Note that the MMI board **must** be located in an un-cached region of VME space. It is not presently possible to use any MMI board with kernel caching, since both the DSP and the host CPU will modify DRAM. Consult the OS-9 documentation for information on how to designate some or all of VME space as **un-cached**.

3.6.5 Compile All Source Files

As the super-user, run "`make`" from the top level "`dispatch`" directory. This will compile the DiSPATCH library and all provided applications. Note that this may take as long as several hours.

There should be no warnings or errors reported during the compilation. If any are encountered, there is likely to be a problem with the installation, and the error should be corrected before proceeding.

3.6.6 Edit the System Startup File

Before any DiSPATCH applications can be run, there must be one "`os9_mmidsp`" process running for each installed DSP. The "`configuration`" program creates a script file called "`mmi_startup`" in the "`apps`" directory that loads the "`os9_mmidsp`" module and spawns the necessary processes.

The system "`startup`" file for OS-9 should be used to automatically load the daemon processes by executing "`mmi_startup`". This will ensure that the necessary processes are always present after booting.

3.6.7 Reboot the System

Reboot the OS-9 system and verify that the "`startup`" script correctly executes "`mmi_startup`", and that all the daemon processes start correctly.

3.6.8 Test the Package

Verify that the DiSPATCH system is working properly by running the "`mmi_test`" program in the "`dispatch/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation. See Section 5.1 for more information on using the "`mmi_test`" program.

The first argument to the "`open_board`" command should be "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or "`MMI-4211`" for MMI-420/MMI-4211 models.

For example, type the following commands to turn the green LED on and off on an MMI-210-4:

```
$ mmi_test

       *** MMI Test ***

Interactive DiSPATCH test program.
```

```
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-210-4 /dev/mmidsp0
MMI-4210 #1 is ready.

MMI-4210 #1 [DSP A]: boot

MMI-4210 #1 [DSP A]: led on

MMI-4210 #1 [DSP A]: led off

MMI-4210 #1 [DSP A]:
```

3.7 HP-RT

The directory “`dispatch/driver/hp-rt`” contains DiSPATCH support for version 1.11 of the HP-RT operating system from Hewlett-Packard Co.

Before attempting to add the DiSPATCH device driver to your HP-RT system, please verify that you can build and boot a new HP-RT kernel without the driver. These instructions assume a working knowledge of HP-RT system administration. Consult the HP documents titled *HP-RT System Administration Tasks* (B3127-90005) and *Driver Writing in the HP-RT Environment* (B3127-90006) for additional information.

3.7.1 Installation

To install the DiSPATCH package, the following steps must be performed. Each of these steps is explained in detail below.

1. Extract the files from the distribution media.
2. Verify the `HPRTroot` environment variable setting.
3. Allocate interrupt levels to HP-RT.
4. Configure the DiSPATCH device driver.
5. Add the driver to the HP-RT kernel configuration.

6. Build a new HP-RT kernel.
7. Build the library and sample applications.
8. Reboot the HP-RT system using the new kernel.
9. Test the DiSPATCH system with the provided applications.

3.7.2 Extract the DiSPATCH Files

The DiSPATCH files are shipped in "`tar`" format. To extract the distribution, `cd` to a directory where you have write permission, and run "`tar -xv`". It may also be necessary to explicitly specify the name of your local tape device on the command line.

This will create the "`dispatch`" directory tree and all required source files.

3.7.3 The `HPRTroot` Environment Variable

The DiSPATCH device driver "`Makefile`" uses the current value of the `HPRTroot` environment variable. It is important that this variable be set correctly before building the DiSPATCH device driver.

This variable must point to the location of the HP-RT file system's root directory on the HP-UX host system (usually "`/HP-RT`"). No default is assumed by the DiSPATCH Makefiles.

3.7.4 Interrupt Level Allocation

Each of the seven VMEbus interrupt levels can be assigned to either the HP-RT target system or the HP-UX host system. An interrupt level can not be assigned to both systems at once.

Before installing and configuring the MMI boards, you much choose one or more interrupt levels and assign them to the HP-RT system. The factory-default interrupt level for MMI boards is 1. This can be changed via the on-board jumper settings. Consult the MMI owner's manual for available jumper configurations.

HP-UX Configuration

Verify that the interrupts are allocated correctly in the file "`/etc/vme/vme.CFG`" on the HP-UX host system. The MMI interrupt request line should be assigned to the `hp742rt` processor.

If any changes are made to "`vme.CFG`", run "`/etc/vme_config`" and reboot the HP-UX system to make the changes take effect.

HP-RT Configuration

On the HP-RT system, the file "`$HPRTroot/usr/include/machine/sysdev.h`" defines the VME interrupt allocation.

Find the definition of `VME_INT_x_CPU`, where `x` is the interupt level used by the MMI board(s). Make sure that this symbol is defined to be the HP-RT CPU number so that it can claim that interrupt level. For example, if the MMI boards are using VME interrupt level 1, and the HP-RT CPU number is 1, this line should appear in the "`sysdev.h`" file:

```
#define         VME_INT_1_CPU   1
```

Later in that same file, change the definition of `HI_VME_x` from `0` to `0xFF` to allow multiple interrupt vectors on that level. For example, this line should be used for interrupt level 1:

```
#define HI_VME_1    0xFF /* Highest return id for VME level 1 interrupts */
```

Also, make sure that the value of `VME_IACK_MODE_VALUE` does *not* include `VME_FAST_IACK_x`. This will instruct the HP-RT kernel to use a second-level interrupt vector table as required.

Consult the *HP-RT System Administration Tasks* manual for more information on interrupt allocation under HP-RT.

3.7.5 Driver Configuration

Before building the DiSPATCH device driver, two files must be created to describe the hardware configuration of your system. These two files are the MMIDSP kernel data structures ("`mmidsp_info.c`") and the kernel configuration file ("`mmidsp.cfg`").

Both of these files are created automatically by running the supplied interactive "`configure`" program. To build and run this program, type "`make configure`" from within the "`dispatch/driver/hp-rt`" directory.

If the program builds sucessfully, it will run and prompt for board information. For each installed board, you will need to provide the following information:

- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`configure`" program. The user must specify the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xE0` where possible.

3.7.6 Add the Driver

To make the DiSPATCH (mmidsp) driver part of the HP-RT kernel, edit the file "`$HPRTroot/etc/conf/cfg/CONFIG.TBL`" and add these lines to the end:

```
##########################################################################
# mmidsp device driver for Vigra DSP audio boards.
##########################################################################
I:mmidsp.cfg
```

3.7.7 Build a New HP-RT Kernel

As the super-user, go to the "`dispatch/driver/hp-rt`" directory and run "`make kernel`". This will compile the DiSPATCH driver and build a new HP-RT kernel.

If any errors are encountered during compilation, there is likely to be a problem with the installation, and the error should be corrected before proceeding.

3.7.8 Build the Library and Applications

The DiSPATCH library and sample applications are provided in C source form. They must be compiled on the HP-UX host system for the HP-RT target.

To build the provided library and applications, go to the top-level “dispatch” distribution directory on the HP-UX host and run:

```
make all
```

No warnings or errors should be reported during the compilation. If any errors are detected, they should be resolved before continuing with the installation.

3.7.9 Reboot the System

Reboot the HP-RT system and verify that the new kernel is executing correctly. During startup, the kernel should print a Vigra copyright message and state the number of DiSPATCH DSPs configured into the system, as shown below.

```
Vigra MMIDSP driver installed.  Copyright (C) 1994, Vigra.
HP-RT kernel configured for 4 Vigra audio DSPs.
```

3.7.10 Test the Package

Verify that the DiSPATCH system is working properly by running the “mmi-test” program in the “dispatch/apps” directory. Use the “led” command to test the firmware, driver, and library operation. See Section 5.1 for more information on using the “mmi-test” program.

The first argument to the “open_board” command should be “MMI-210-1” for 1-meg MMI-210 boards, “MMI-210-4” for 4-meg MMI-210 boards, or “MMI-4211” for MMI-420/MMI-4211 models.

For example, type the following commands to turn the red LED on and off on an MMI-4211:

```
$ ./mmi-test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
```

```
MMI-4211 #1 is ready.

MMI-4211 #1 [DSP A]: boot

MMI-4211 #1 [DSP A]: led on

MMI-4211 #1 [DSP A]: led off

MMI-4211 #1 [DSP A]:
```

3.8 HP-UX 9.0.x

The directory “`dispatch/driver/hp-ux9`” contains DiSPATCH support for version 9.0.x of the HP-UX operating system from Hewlett-Packard Co.

Before attempting to add the DiSPATCH device driver to your HP-UX system, please verify that you can build and boot a new HP-UX kernel without the driver. These instructions assume a working knowledge of HP-UX system administration and kernel manipulation. Consult the Hewlett-Packard system administration manuals for additional information.

3.8.1 Installation

To install the DiSPATCH package, the following steps must be performed. Each of these steps is explained in detail below.

1. Extract the files from the distribution media.
2. Configure the MMI board jumpers.
3. Allocate interrupt levels to HP-UX.
4. Configure the DiSPATCH device driver.
5. Add the driver to the HP-UX “master” file.
6. Add the driver to the HP-UX “dfile”.
7. Build a new HP-UX kernel.
8. Install the new HP-UX kernel.

9. Build the library and sample applications.
10. Reboot the HP-UX system using the new kernel.
11. Make the device handles.
12. Test the DiSPATCH system with the provided applications.

3.8.2 Extract the DiSPATCH Files

The DiSPATCH files are shipped in "`tar`" format. To extract the distribution, `cd` to a directory where you have write permission, and run "`tar -xv`". It may also be necessary to explicitly specify the name of your local tape device on the command line.

This will create the "`dispatch`" directory tree and all required source files. The top-level directory name will also contain the current DiSPATCH release number, as in "`dispatch-1.60`". Please read the file named "RELEASE" for any supplemental instructions.

3.8.3 Configure your new MMI board

To install the DiSPATCH device driver, you will need to know the settings of each MMI board installed in your HP-UX system. These values are:

1. The VME base address.
2. The VME interrupt priority level.

The VME base address and interrupt priority level are set via jumpers on the MMI board. Please choose settings that do not conflict with other boards or system hardware.

3.8.4 Interrupt Level Allocation

Each of the seven VMEbus interrupt levels can be assigned to one HP processor board. An interrupt level can not be assigned to multiple systems at once.

Before installing and configuring the MMI boards, you much choose one or more interrupt levels and assign them to the HP-UX system. The factory-default interrupt level for MMI boards is 1. This can be changed via the on-board jumper settings. Consult the MMI owner's manual for available jumper configurations.

Verify that the interrupts are allocated correctly in "`/etc/vme/vme.CFG`" on the HP-UX host system. If you have multiple CPU boards in the same VMEbus, the MMI interrupt request line should be assigned to the HP-UX processor only.

If any changes are made to "`vme.CFG`", use "`/etc/vme_config`" and reboot the HP-UX system to make the changes take effect.

If an HP-RT target system shares the same VMEbus as the HP-UX host, verify that the interrupt assignment does not conflict with those defined in "`$HPRTroot/usr/include/machine/sysdev.h`". Consult the *HP-RT System Administration Tasks* manual for more information on interrupt allocation under HP-RT.

3.8.5 Driver Configuration

Before building the DiSPATCH device driver, the file "`mmidspinfo.c`" must be created to describe the hardware configuration of your system.

This file is created automatically by running the interactive "`configure`" program. To build and run this program, type "`make config`" from within the "`dispatch/driver/hp-ux9`" directory.

If the program builds successfully, it will run and prompt for board information. For each installed board, you will need to provide the following information:

- The starting VME interrupt vector.
- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Assigning the VME Interrupt Vectors

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`configure`" program. The user must specify the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xE0` where possible.

Alternatively, the VME interrupt vectors can be automatically assigned by HP-UX at boot time. To request automatic vector assignment, enter `auto` as the starting interrupt vector.

Selecting the board to be configured

A list of MMI audio board models to choose from is displayed. To add configuration information for a board, select the model name that corresponds to your board. To end the configuration, enter “`0`”.

Entering the base VME Address of the board

Each MMI board occupies a VME address range specified by the board’s jumper settings. The address entered into the program must match the settings on the board. This address should be entered in hexadecimal form (i.e. `0x81000000`).

Entering the VME Interrupt Priority Level of the board

Each DiSPATCH board is set to a VME Interrupt Priority Level based on the board jumper settings. The priority level specified must match the settings on the board. The value should be in the range of 1 to 7; the factory default setting is level 1. Consult the MMI Owner’s Manual for information regarding changing this setting.

Confirming your selections

Once the settings for a board are entered, they will be displayed for verification. If the information needs to be changed, enter `N`, and enter the correct information. If the information is correct, enter `Y`, and the board’s configuration will be added to the driver information file.

3.8.6 Add the Driver to the Master File

To make the DiSPATCH (mmidsp) driver part of the HP-UX kernel, edit the file “`/etc/master`” and add this line to the section reserved for “Third Party and User Drivers”:

```
mmidsp mmidsp 1 1FC -1 43
```

The last number in the line specifies the major number to be assigned to the DiSPATCH device driver. The major number is a unique identifier used to access the device driver. No other drivers may share this number. On most systems, 43 is available for use. You may choose any unused number other than 43 and create the device nodes accordingly.

3.8.7 Add the Driver to the driver description file

Locate the `dfile` used to build the current HP-UX kernel. This is usually named `/etc/conf/dfile`. If this is not the `dfile` used to build your HP-UX system, then please edit “`dispatch/driver/hp-ux9/Makefile`” to reflect the correct filename.

Edit “`dfile`” and add the name `mmidsp` to the end of the list. Also, verify that the driver `vme2` is included in the list. If it is not, add it as well.

3.8.8 Build a New HP-UX Kernel

Go to the “`dispatch/driver/hp-ux9`” directory and run “`make kernel`”. This will compile the DiSPATCH driver and build a new HP-UX kernel.

If any errors are encountered during compilation, there is likely to be a problem with the installation, and the error should be corrected before proceeding.

3.8.9 Install the New HP-UX Kernel

Make a backup of your old (working) HP-UX kernel and copy the new HP-UX kernel file into the root directory:

```
cp -p /hp-ux /hp-ux.nommi
cp -p ./hp-ux /hp-ux
```

3.8.10 Build the Library and Applications

The DiSPATCH library and sample applications are provided in C source form. They must be compiled on the HP-UX host system before use.

To build the provided library and applications, go to the top-level “`dispatch`” distribution directory and run:

```
make all
```

No warnings or errors should be reported during the compilation. If any errors are detected, they should be resolved before continuing with the installation.

3.8.11 Reboot the System

Reboot the HP-UX system and verify that the new kernel is executing correctly. During startup, the kernel should print a Vigra copyright message and state the number of DiSPATCH devices configured into the system, as shown below.

```
(c)Copyright 1995,1996 Vigra a division of VisiCom Laboratories Inc.
mmidsp driver: DiSPATCH Release 1.60, 14-Mar-96
    Unit  0: /dev/mmidsp0_map @ A32 0x81000000
    Unit  1: /dev/mmidsp0_A   @ A32 0x813ffe80 IntVec=0x01 IRQ Level 1
    Unit  2: /dev/mmidsp0_B   @ A32 0x813ffec0 IntVec=0x02 IRQ Level 1
    Unit  3: /dev/mmidsp0_C   @ A32 0x813fff00 IntVec=0x03 IRQ Level 1
    Unit  4: /dev/mmidsp0_D   @ A32 0x813fff40 IntVec=0x04 IRQ Level 1
    Vigra MMI DSP Driver Attached, 5 Devices Supported.
```

3.8.12 Make the Device Handles in "/dev"

Each MMI board has one or more DSPs, each of which needs its own driver handle, since they send messages independently. Only one mapping device handle is needed for each board, since all DSPs share a common DRAM address space.

The comments in the file "dispatch/driver/hp-ux9/mmidspinfo.c" contain the information needed to properly create the device files. All the DiSPATCH devices are considered character devices.

Execute the mknod command once for each DiSPATCH board, and once for each DSP on a board. The device names are "mmidsp" followed by the board number (starting with 0), followed by "_map" (if the device is the mapper device), or followed by "_A" for the first DSP, "_B" for the second DSP, and so on. The minor number is the unit number shown in the comments of the driver information file.

For example, if the driver information file contains the following:

```
  /* MMI-4211 at base address 0x81000000 */
  { 0x81000000, 4,   -1 }, /*  0 /dev/mmidsp0_map */
  { 0x813FFE80, 0x00, 1 }, /*  1 /dev/mmidsp0_A */
  { 0x813FFEC0, 0x00, 1 }, /*  2 /dev/mmidsp0_B */
  { 0x813FFF00, 0x00, 1 }, /*  3 /dev/mmidsp0_C */
  { 0x813FFF40, 0x00, 1 }, /*  4 /dev/mmidsp0_D */
```

Then the commands to execute would be:

```
mknod /dev/mmidsp0_map c 43 0
mknod /dev/mmidsp0_A c 43 1
mknod /dev/mmidsp0_B c 43 2
mknod /dev/mmidsp0_C c 43 3
mknod /dev/mmidsp0_D c 43 4
```

Note: If the major number selected for the DiSPATCH driver is not 43, be sure to use the major number that was specified in the "/etc/master" file.

Be sure to set the file permissions appropriately.

3.8.13 Test the Package

Verify that the DiSPATCH system is working properly by running the "mmi-test" program in the "dispatch/apps" directory. Use the "led" command to test the firmware, driver, and library operation. See Section 5.1 for more information on using the "mmi-test" program.

The first argument to the "open_board" command should be "MMI-210-1" for 1-meg MMI-210 boards, "MMI-210-4" for 4-meg MMI-210 boards, or "MMI-4211" for MMI-420/MMI-4211 models.

For example, type the following commands to turn the red LED on and off on an MMI-4211:

```
$ ./mmi-test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4211 #1 is ready.
```

```
MMI-4211 #1 [DSP A]: boot

MMI-4211 #1 [DSP A]: led on

MMI-4211 #1 [DSP A]: led off

MMI-4211 #1 [DSP A]:
```

3.8.14 HP-UX System Administration Manager (SAM)

If you use the HP-UX System Administration Manager (SAM), you can update it to include a description of the DiSPATCH mmidsp driver by adding a line to the SAM driver description file "`/usr/sam/lib/kc/drivers.tx`".

The line to be added is:

```
mmidsp::Card:Out:VisiCom DiSPATCH Driver
```

The new line should be added just after the line:

```
vme2::Card:Out:VME Expander Driver7.
```

3.9 HP-UX 10.xx

This section describes the DiSPATCH software installation procedure for version 10 of the HP-UX operating system from Hewlett-Packard Co.

Before attempting to add the DiSPATCH device driver to your HP-UX 10.xx system, please verify that you can build and boot a new HP-UX kernel without the driver. These instructions assume a working knowledge of HP-UX system administration and kernel manipulation. Consult the Hewlett-Packard system administration manuals for additional information.

3.9.1 Installation

To install the DiSPATCH package, the following steps must be performed. Each of these steps is explained in detail below.

1. Extract the files from the distribution media.

2. Configure the MMI board jumpers and install the board.
3. Configure HP-UX VME settings.
4. Configure the DiSPATCH device driver.
5. Build a new HP-UX kernel.
6. Back up your existing HP-UX kernel.
7. Install the new HP-UX kernel.
8. Build the library and sample applications.
9. Reboot the HP-UX system using the new kernel.
10. Make the device handles.
11. Test the DiSPATCH system with the provided applications.

3.9.2 Extract the DiSPATCH Files

The DiSPATCH files are shipped in "`tar`" format, usually on 4mm DAT tape. To extract the distribution, `cd` to a directory where you have write permission, and run "`tar -xv`". It may also be necessary to explicitly specify the name of your local tape device on the command line.

This will create the "`dispatch`" directory tree and all required source files. The top-level directory name will also contain the current DiSPATCH release number, as in "`dispatch-1.87`". Please read the file named "RELEASE" for any supplemental instructions.

3.9.3 Configure the MMI Board

To install the DiSPATCH device driver, you will need to know the settings of each MMI board installed in your HP-UX system. These values are:

1. The VME base address.
2. The VME interrupt priority level.

The VME base address and interrupt priority level are set via jumpers on the MMI board. Choose settings that do not conflict with other boards or system hardware.

3.9.4 HP-UX VME Settings

HP-UX requires that the system administrator configure the VME bus to select interrupts and address regions for each VME board. Each of the seven VMEbus interrupt levels can be assigned to one HP processor board. An interrupt level can not be assigned to multiple systems at once.

Before installing and configuring the MMI boards, you much choose one or more interrupt levels and assign them to the HP-UX system. The factory-default interrupt level for MMI boards is 1. This can be changed via the on-board jumper settings. Consult the MMI owner's manual for available jumper configurations.

Verify that the interrupts and memory regions are allocated correctly by using "`vme_config(1M)`" on the HP-UX host system. If you have multiple CPU boards in the same VMEbus, the MMI interrupt request line should be assigned to the HP-UX processor only. For more information on VME configuration and the configuration file format, see the *HP-UX 10.20 VME Services Guide*.

If an HP-RT target system shares the same VMEbus as the HP-UX host, verify that the interrupt assignment does not conflict with those defined in "`$HPRTroot/usr/include/machine/sysdev.h`". Consult the *HP-RT System Administration Tasks* manual for more information on interrupt allocation under HP-RT.

3.9.5 Driver Configuration

Before building the new kernel device driver, the DiSPATCH configuration program must be run to interactively describe the MMI hardware configuration of your system. To build and run the "configure" program, execute "`make config`" from within the "`dispatch/driver/hp-ux10`" directory.

If the program builds successfully, it will run and prompt for board information. For each installed board, you will need to provide the following information:

- The starting VME interrupt vector.
- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Assigning the VME Interrupt Vectors

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`configure`" program. The user must specify the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xE0` where possible.

Alternatively, the VME interrupt vectors can be automatically assigned by HP-UX at boot time. To request automatic vector assignment, enter the word `auto` as the starting interrupt vector.

Selecting the board to be configured

A list of MMI audio board models to choose from is displayed. To add configuration information for a board, select the model name that corresponds to your board. To end the configuration, enter "`0`".

Entering the base VME Address of the board

Each MMI board occupies a VME address range specified by the board's jumper settings. The address entered into the program must match the settings on the board. This address should be entered in hexadecimal form (i.e. `0x81000000`).

Entering the VME Interrupt Priority Level of the board

Each DiSPATCH board is set to a VME Interrupt Priority Level based on the board jumper settings. The priority level specified must match the settings on the board. The value should be in the range of 1 to 7; the factory default setting is level 1. Consult the MMI Owner's Manual for information regarding changing this setting.

Confirming your selections

Once the settings for a board are entered, they will be displayed for verification. If the information needs to be changed, enter `N` to start over. If the information is correct, enter `Y`, and the board's configuration will be added to the driver information file.

3.9.6 Build a New HP-UX Kernel

Go to the "`dispatch/driver/hp-ux10`" directory and run "`make kernel`". This will compile the DiSPATCH driver and build a new HP-UX kernel. The kernel and customized files are built in this directory, and no system files are modified at this time.

If any errors are encountered during compilation, there is likely to be a problem with the installation, and the error should be corrected before proceeding. There may be a warning about "potentially obsolete features" during the link phase of the kernel build. This message can be safely ignored. A successful kernel build produces the file "`vmunix_test`".

3.9.7 Back Up Your Existing Kernel

Before installing the DiSPATCH kernel for the first time, make a back-up copy of your existing files. You can run "`make backup_kernel`" to copy your existing "`/stand/system`" and "`/stand/vmunix`" to "`system.no-mmidsp`" and "`vmunix.no-mmidsp`", respectively. If this is not appropriate for your system, please back up your kernel files manually.

3.9.8 Install the New HP-UX Kernel

After making a reliable backup of your existing (working) HP-UX kernel, copy the new HP-UX kernel file into place with "`make install_kernel`".

3.9.9 Build the Library and Applications

The DiSPATCH library and sample applications are provided in C source form. They must be compiled on the HP-UX host system before use.

To build the provided library and applications, go to the top-level "`dispatch`" distribution directory and run:

```
make all
```

No warnings or errors should be reported during the compilation. If any errors are detected, they should be resolved before continuing with the installation.

3.9.10 Reboot the System

Reboot the HP-UX system and verify that the new kernel is executing correctly. During startup, the kernel should print a Vigra copyright message and state the number of DiSPATCH devices configured into the system, as shown below.

```
DiSPATCH Release 1.87, 1997-05-02
Unit  0: /dev/mmidsp0_map @ A32 0x81000000
Unit  1: /dev/mmidsp0_A   @ A32 0x813ffe80 IntVec=0x01 IRQ Level 1
Unit  2: /dev/mmidsp0_B   @ A32 0x813ffec0 IntVec=0x02 IRQ Level 1
Unit  3: /dev/mmidsp0_C   @ A32 0x813fff00 IntVec=0x03 IRQ Level 1
Unit  4: /dev/mmidsp0_D   @ A32 0x813fff40 IntVec=0x04 IRQ Level 1
Vigra MMI DSP Driver Attached, 5 Devices Supported.
```

3.9.11 Make the Device Handles in "`/dev`"

Each MMI board has one or more DSPs, each of which needs its own driver handle, since they send messages independently. In addition, one mapping device handle is needed for each MMI board, since all DSPs share a common DRAM address space.

To create these device nodes, run the script "`make_devs`" as `root` from the "`dispatch/driver/hp-ux10`" driver directory. This should automatically find the assigned device major number and create the configured devices in `/dev`.

Be sure to set the file permissions for "`/dev/mmidsp*`" appropriately. Only users with read and write permission on these device handles can access the audio boards.

3.9.12 Test the Package

Verify that the DiSPATCH system is working properly by running the "`mmi-test`" program in the "`dispatch/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation. See Section 5.1 for more information on using the "`mmi-test`" program.

The first argument to the "`open_board`" command should be "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or "`MMI-4211`" for MMI-420/MMI-4211 models.

For example, type the following commands to turn the red LED on and off on an MMI-4211:

```
$ ./mmi-test
```

```
       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4211 #1 is ready.

MMI-4211 #1 [DSP A]: boot

MMI-4211 #1 [DSP A]: led on

MMI-4211 #1 [DSP A]: led off

MMI-4211 #1 [DSP A]:
```

3.10 Motorola System V/88

The directory "`dispatch/driver/sysv88`" contains DiSPATCH support for Motorola UNIX System V/88 Release R32V3.

Before attempting to add the DiSPATCH device driver to your System V/88 system, please verify that you can build and boot a new kernel without this driver. These instructions assume a working knowledge of System V/88 system administration. Consult the Motorola document titled *System V/88 and System V/68 Device Driver Writer's Guide* for additional information concerning the use of `sysgen` and building a Unix kernel.

Before beginning DiSPATCH installation, **make certain that you have a complete and reliable system backup!** As with any kernel modifications, an error in installation could damage existing files or render the entire system unbootable. You may also wish to make a copy of your existing "`/unix`" to keep on hand as an alternate kernel with no DiSPATCH driver.

3.10.1 Installation

To install the DiSPATCH package, the following steps must be performed. Each of these steps is explained in detail below.

1. Extract the files from the distribution media.
2. Verify that the definitions in the driver `Makefile` are correct.
3. Build the device driver and configuration tools.
4. Configure the DiSPATCH device driver.
5. Install the DiSPATCH driver support files.
6. Add the driver to the System V/88 kernel configuration using sysgen.
7. Verify that the maximum shared memory segment size is adequate.
8. Build a new System V/88 kernel.
9. Reboot the System V/88 system using the new kernel.
10. Build the library and sample applications.
11. Test the DiSPATCH system with the provided applications.

3.10.2 Extract the DiSPATCH Files

The DiSPATCH files are shipped in “`tar`” format. To extract the distribution, `cd` to a directory where you have write permission, and run “`tar -xv`”. It may also be necessary to explicitly specify the name of your local tape device on the command line.

This will create the “`dispatch`” directory tree and all required source files. The name of the “`dispatch`” directory may also contain a version number, for example: “`dispatch-1.45`”.

3.10.3 Check over the definitions in `Makefile`

The first part of the `Makefile` defines several symbols that may need editing for your system. These are:

`MAJOR_NUMBER`
: The major device number for all MMI DSP device nodes. Make sure that this is set to a major number not currently in use on your system (0–255).

`KERN_SRC`
: This is the base directory for the Motorola kernel build tree.

`KERN_LIB_DIR`
: This directory contains all the binary libraries that make up the kernel. The compiled MMI driver library will be copied here.

`SYSGEN_DESC_DIR`
: This directory holds device descriptions used by `/etc/sysgen`. The new MMI driver description will be placed here.

`RC_FILE`
: The MMI startup file needs to run each time the System V/88 system is booted. By copying the startup script to this file, it will be done automatically during the "`/etc/rc`" phase of startup. Make sure that the number in this filename does not conflict with any other startup files on your system.

By default, the `Makefile` will use the default C compiler (`cc`) to build all binaries. This is assumed to be the Green Hills C compiler. If your default compiler is not the Green Hills C compiler, then you may need to edit the definitions of `CC` and `CFLAGS` as necessary.

The default values provided in the Makefile are suitable for a standard installation of System V/88.

3.10.4 Build the Device Driver and Configuration Tools

To build the device driver and configuration tools, change to the directory "`dispatch/driver/sysv88k`" and run `make`.

The `Makefile` will build three binaries:

`lib.mmidsp`
: This is the kernel device driver archive for System V/88, necessary to support DSP interrupts and messages. The driver binary does not contain any configuration information that is specific to your system configuration.

`configure`
: This program will be used next to interactively construct the system-specific configuration files for your installation.

`mmi_map`
: This program creates a shared memory segment for each MMI board, so that many applications can share a board with the kernel. It will be run during system startup, usually during `/etc/rc2` operations.

The compiler should not report any errors or warnings during compilation. If any were encountered during the build of these files, stop here and investigate the cause.

3.10.5 Driver Configuration

The program "`configure`" will help you create three files that describe your system configuration. These files are:

`mmiconf.desc`
: This file is a "device description" entry for the Unix `sysgen` kernel-building system. It will be installed into the `sysgen/descriptions` directory of your kernel source tree.

`mmiconf.mkdev`
: This is a shell script to create the character device nodes in the `/dev` directory. It will be run automatically as part of the installation procedure.

`mmiconf.start`
: This is a shell script to create the necessary shared memory regions at startup. It will be copied to `/etc/rc2.d/S95mmi_start` during installation, which will allow it to run during each future boot process.

To create these files, execute the command "`make conf`" from within the "`dispatch/driver/sysv88k`" directory. This program will ask questions and prompt for board information. For each installed board, you will need to provide the following information:

- The Vigra MMI model name.
- The board's VME base address, as configured by the jumpers.
- The board's VME interrupt priority level.

Each DSP installed in the system requires a unique VME interrupt vector. These will be assigned sequentially by the "`configure`" program. The user must specify the first VME vector to be allocated to the MMI boards. Please be certain that all allocated vectors are unique to the MMI boards and not shared with any other internal or VME systems. Vigra recommends a starting interrupt vector of `0xE0` where possible.

When configuration is complete, the three "`mmiconf.*`" files will be created in the current directory. You may examine or edit them if necessary. You may also re-run `configure` to create new files if there were any errors in entering your system configuration.

3.10.6 Install the DiSPATCH Driver and Support Files

The installation step must be performed by the super-user since it copies files into protected directories. Please verify that you are `root` before proceeding.

To perform the driver installation, execute the command "`make install`" from the "`dispatch/driver/sysv88k`" directory. This will perform the following steps:

- Copy "`mmiconf.desc`" to the sysgen description directory.
- Copy "`lib.mmidsp`" to the kernel library directory.
- Copy the program "`mmi_map`" to `/etc`.
- Copy "`mmiconf.start`" to a startup file in `/etc/rc`.
- Run "`mmiconf.mkdev`" to create device nodes in `/dev`.

You may verify the exact commands and filenames used during installation by running "`make -n install`" or reading the `Makefile` itself.

3.10.7 Add the Driver to the Kernel Using `/etc/sysgen`

After all the necessary configuration files have been copied into place, you must run "`/etc/sysgen`" to select the new `mmidsp` driver and include it in the kernel.

The new device description line should appear near the end of the device list and looks like this:

```
Vigra MMI Audio DSP Board                                    mmidsp
```

Mark this selection with a "*" (by pressing $\boxed{\texttt{s}}$), and then press $\boxed{\texttt{o}}$ to select the individual parts of the driver.

You will then be presented with a list of devices and one driver. **You must select ALL the entries on this `mmidsp` screen.** When all lines are selected and marked with a "*", you may exit the `mmidsp` screen, but do not exit `sysgen` yet.

3.10.8 Verify the Maximum Shared Memory Segment Size (`SHMMAX`).

The default sysgen configuration for System V/88 sets the maximum shared memory segment size to 512 kbytes. This is not big enough to map an MMI audio board, so this maximum must be raised before DiSPATCH can function.

The configuration parameter `SHMMAX` defines the maximum shared memory segment size. This parameter is found on the "Shared Memory Parameters" screen of the kernel configuration in `sysgen`. Set the value of this field to **four megabytes**, entered as "`(4096*1024)`".

3.10.9 Build a New Kernel

Press `q` repeatedly to exit `sysgen` and answer "yes" to all three of the questions that follow. This will save your configuration changes, rebuild the Unix kernel, and install the new kernel at the next reboot.

3.10.10 Reboot the System V/88 System

If no errors are reported during kernel compilation, reboot the machine by running "`init 0`" or something similar. When the system reboots, you should see a series of kernel messages preceeded by "`mmidsp:`" indicating your MMI configuration. They will look similar to these:

```
mmidsp: Vigra MMI Audio DSP driver installed. (C) 1994, Vigra.
mmidsp: DSP 0 is at 0x813FFE80 using ivect 0xD0, level 1.
mmidsp: DSP 1 is at 0x813FFEC0 using ivect 0xD1, level 1.
mmidsp: DSP 2 is at 0x813FFF00 using ivect 0xD2, level 1.
mmidsp: DSP 3 is at 0x813FFF40 using ivect 0xD3, level 1.
```

If any messages report that a warning that a DSP was "not found", then there may be a configuration or installation error. Check the kernel configuration files and address jumper settings on each card.

3.10.11 Build the Library and Applications

The DiSPATCH library and sample applications are provided in C source form. They must be compiled on the System V/88 machine before they can be run.

To build the provided library and applications, go to the top-level "`dispatch`" distribution directory and run:

```
make all
```

No warnings or errors should be reported during the compilation. If any errors are detected, they should be resolved before continuing with the installation.

3.10.12 Test the Package

Verify that the DiSPATCH system is working properly by running the "`mmi-test`" program in the "`dispatch/apps`" directory. Use the "`led`" command to test the firmware, driver, and library operation. See Section 5.1 for more information on using the "`mmi-test`" program.

The first argument to the "`open_board`" command should be "`MMI-210-1`" for 1-meg MMI-210 boards, "`MMI-210-4`" for 4-meg MMI-210 boards, or "`MMI-4211`" for MMI-420/MMI-4211 models.

For example, type the following commands to turn the red LED on and off on an MMI-4211:

```
$ ./mmi-test

       *** MMI Test ***

Interactive DiSPATCH test program.
Copyright (C) 1993 Vigra, Inc.
$Revision: 1.23 $

MMI Test: open_board MMI-4211 /dev/mmidsp0
MMI-4211 #1 is ready.

MMI-4211 #1 [DSP A]: boot

MMI-4211 #1 [DSP A]: led on

MMI-4211 #1 [DSP A]: led off

MMI-4211 #1 [DSP A]:
```

4. C Programming Library

The **DiSPATCH Programming Library** provides the software developer with a complete set of high-level function calls to manipulate the DiSPATCH firmware from the host application. In conjunction with the DiSPATCH device driver, the library routines handle all DSP initialization and bootstrap operations, insulating the audio programmer from the many of the details of VME interfacing.

Vigra's MMI VME audio product line offers a wide variety of specific board features, and each board has unique capabilities and interface procedures. The programming library provides a *uniform* interface to all supported Vigra VME audio boards allowing a single application to use any model, as well ensure compatibility with future Vigra DiSPATCH VME boards.

4.1 Library Overview

The primary function of the library is to provide a software interface to the DiSPATCH firmware and VME hardware. The library does *not* perform any audio signal processing. Instead, the library directs the DSP which processes the audio streams in real time on behalf of the host. The library manages the transfer of audio data between the host and the MMI board memory.

The library treats each audio DSP as an independent processing agent. Different DSPs on the same MMI board can be used to perform completely unrelated tasks. In most cases, they can even be controlled by separate system processes.

The library is fully re-entrant and dynamically allocates all resources to support a virtually unlimited number of DSPs.

4.1.1 Device Driver

The DiSPATCH device driver is responsible for receiving messages from the DSP and placing them in a queue to be read by the library. The driver does not interpret or process the incoming data at all. These simple requirements allow the device driver to be small, efficient, and very portable. The driver is interrupt driven for minimum latency.

On systems (such as VxWorks) that allow interrupts to be received by non-kernel processes, the device driver is implemented as an integral part of the library. This interrupt-driven routine receives the DSP messages and writes them to a data stream that is read by the library.

4.1.2 DSP Naming

The library refers to DSPs only by a symbolic name. On Unix-like operating systems, this name is the filename for the special device driver node, usually in the "`/dev`" directory. The library requires that the driver handles are named as described in Chapter 3. The host application gives the library a basename for the board (i.e. "`/dev/mmidsp0`"), and the library builds the appropriate DSP device names from it.

Under the Microware OS-9 operating system, the DSP symbolic names are identical to those described above. The Unix "`/dev/`" prefix is optional under OS-9, but it is silently ignored by the library to allow complete application compatibility between platforms.

This means that neither the DiSPATCH library or host application needs to know the physical address of the VME board. Only the kernel configuration file needs to contain this information.[1] However, the host application *does* need to know the device filename and the board model to properly initialize the board.

4.1.3 Shared Access

Two or more different processes or threads can share an MMI board if, and only if, they use separate DSPs. Under no conditions can two processes share a single DSP, because only one process can receive messages from each individual DSP.

A single process can utilize many different DSPs, on any variety of supported MMI boards. The library will simultaneously process messages from all active DSPs.

[1] In the case of VxWorks, the hardware configuration information is compiled as part of the interrupt service routine and support functions in "`sys_vxworks.c`".

4.1.4 Handles

The library provides two special structures to contain run-time information about the DSP board. The first is of a type named `mmi_board_t`. This is a pointer to a structure that contains various state information about a single MMI board. The library creates and returns this structure to the application upon board initialization.

After obtaining the `mmi_board_t` board handle with `mmi_open()`, the application can then call `mmi_get_dsp()` to create a handle for a single DSP. This handle is of type `dsp_t` and is passed as an argument to almost all library routines.

4.1.5 Source Code

Complete source code to the DiSPATCH Programming Library is provided on the distribution media. This source code can be an invaluable aid in tracking bugs or enhancing the software for specialized applications. However, Vigra can not provide technical support for customer-modified versions of the library. Please study the code carefully before making any changes, and always retain a copy of the original library source code.

4.1.6 Include Files

The DiSPATCH Programming Library includes a header file named “`dispatch.h`” which must be included by any application that uses the library. This file defines necessary data structures used by the library and several constants that may be useful to the application.

Also, the “`dispatch.h`” file declares C function prototypes for all external library functions. For ANSI C compilers, these prototypes include argument declarations, while non-ANSI compilers will receive only forward function declarations without argument lists.

Because “`dispatch.h`” includes “`firm_defs.h`”, that file must also be accessible when compiling DiSPATCH applications. The include file “`dsp_defs.h`” is necessary for recompiling the library or device drivers, but should never need to be included in application code.

All DiSPATCH include files are wrapped with preprocessor directives to ensure that they are included only once during compilation.

4.1.7 Completion Tokens

Some DiSPATCH library functions start DSP tasks, such as playback, that take a significant amount of time to complete. These functions return a unique positive integer “completion token” that can be saved by the calling application. The DSP will send a completion message to the library when the task completes, and the library will store this notification internally.

The DiSPATCH application can wait for a completion message by calling `mmi_complete()` with the given completion token. This function will not return until the message is received from the DSP. Because `mmi_complete()` uses the `select()` system call, very little host CPU time is consumed during the wait.

Alternatively, the application can poll for the response while performing other tasks. In this case, the application can check if the response has been received (poll) by calling `mmi_check_response()`. This function will return `0` only after the specified response has been received from the DSP. The host application should then call `mmi_complete()` to delete the message from the library’s internal list.

4.2 Initialization and Control Functions

The functions described in this section deal with the initialization and control of the library itself. In general, these commands do not execute DiSPATCH commands on the DSP, but are performed by the host itself.

4.2.1 `mmi_open`

Prototype:

```
mmi_board_t mmi_open (char *model, char *basename,
                      long unsigned int vme_addr);
```

Arguments:

`model`	String model name.
`basename`	Device driver basename.
`vme_addr`	VME address offset.

Returns: A pointer to an allocated and initialized MMI board handle, or `NULL` if there was an error during initialization.

Description: This function opens and prepares an MMI board for library use. This is usually the first DiSPATCH function called by an application during the initialization process.

The arguments specify which MMI board to open and initialize. The `model` argument is a string name representing Vigra's model name for the board. The name is case-insensitive but must otherwise match one of the supported board names. At the time of this writing, the following models are recognized and supported by DiSPATCH:

Model Name
MMI-105-1
MMI-105-4
MMI-105-8
MMI-210-1
MMI-210-4
MMI-4210
MMI-4211
MMI-420

The last argument, `vme_addr`, can be used to specify an application-dependent address offset for VME mapping. Under all presently-supported operating systems, *this parameter should always be set to zero*, since the VME base address of the board is known by the kernel.

The `mmi_open()` library function performs these functions:

1. Initialize the library state if this is the first call to `mmi_open`.
2. Validate the specified model name and look up the corresponding board functions.
3. Allocate an initialize a new `mmi_board_t` structure.
4. Initialize the status information for each DSP on the board.
5. Write the Command ID codes into the command structures in DRAM.

4.2.2 `mmi_close`

Prototype: `int mmi_close (mmi_board_t mmi);`

Arguments:

`mmi`	An MMI board handle returned by `mmi_open()`.

Returns: Zero on success, or `-1` on failure.

Description: This function deallocates and unmaps and MMI board structure created by `mmi_open()`. Any active DSPs on the closed board will be halted. All memory allocated by the library for the specified board will be returned to the system by `mmi_close()`.

It is important to properly close all MMI boards before exiting, especially on systems like VxWorks that do not automatically return system resources upon task completion.

After calling `mmi_close()`, the MMI board pointer is totally deallocated and should not be used again by the calling application. If necessary, the board can be re-opened by calling `mmi_open()`.

4.2.3 `mmi_lib_initialize`

Prototype: `int mmi_lib_initialize (void);`

Returns: Zero on success, or `-1` on failure.

Description: This routine resets the operating state of the DiSPATCH Programming Library. Most applications will never need to call this function directly, since it is automatically executed the first time `mmi_open()` is called.

On operating systems such as VxWorks that do not reset global variable for each process or task, it may be necessary to call this function manually when an application is not terminated normally. If all applications properly call `mmi_close()` before exiting, then it is never necessary to manually reset the library.

Note that manually initializing the library does not reclaim any memory that may still be allocated. Under VxWorks, if a task terminates before releasing library resources, then they can not be recovered except by resetting the operating system. Unix-like systems always automatically de-allocate these resources for terminated tasks.

4.2.4 `mmi_get_dsp`

Prototype: `dsp_t mmi_get_dsp (mmi_board_t mmi, int channel);`

Arguments:

`mmi`	Board handle created by `mmi_open()`.
`channel`	DSP channel number to retrieve.

Returns: A handle (type `dsp_t`) for the specified DSP, or `NULL` on failure.

Description: Most of the functions in the DiSPATCH Programming Library take an argument of type `dsp_t` as their first argument. This pointer is the handle for a single DSP on an initialized MMI board. The `mmi_get_dsp()` function creates the handle for a given DSP.

The first argument is a handle for the whole MMI board containing the desired DSP. This handle is returned from a successful call to `mmi_open()`.

The `channel` argument specifies which of the available DSPs to use. The first DSP is `0`, the second is `1`, and so on.

This function will return `NULL` if the `mmi` argument is `NULL` or the specified channel number is invalid.

4.2.5 `mmi_get_ram_base`

Prototype: `unsigned short *mmi_get_ram_base (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: A pointer to the start of DRAM, or `NULL` if `dsp` is `NULL`.

Description: This function returns a pointer to the start of mapped DRAM. This is the starting address of the user-accessible memory on the MMI board. Any data on the board can be directly accessed by the user application be reading or writing to memory starting at this address.

The number of 16-bit words available for user data buffers is stored in the `databuf_size` field of an initialized `mmi_board` structure. The total size of on-board DRAM (including space reserved for the library) is found in the `ram_size` field.

4.2.6 mmi_get_model

Prototype: `char *mmi_get_model (mmi_board_t mmi);`

Arguments:

`mmi`	Board handle created by `mmi_open()`.

Returns: The string model name of the board.

Description: This function is used to retrieve the model name of an open MMI board. It returns the name string given to `mmi_open()`.

This function will return `NULL` if the `mmi` argument is `NULL`.

4.2.7 `mmi_start_firmware`

Prototype: `int mmi_start_firmware (dsp_t dsp, char *basename);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`basename`	Substitute boot image during development. Use `NULL`.

Returns: Zero on success, or `-1` on failure.

Description: This powerful function completely initializes a single DSP for DiSPATCH operation. This procedure involves resetting the DSP, uploading the P, X, and Y memory images, and waiting for the appropriate boot messages.

When this function returns successfully, the DSP is ready for DiSPATCH operation.

The `basename` argument is used only during firmware development and should normally be passed as `NULL`. If a string basename is supplied, the library will load and boot the specified binary image from disk. When `basename` is `NULL`, the library will use the default DiSPATCH firmware image linked with into library.

4.2.8 `mmi_boot_default`

Prototype: `int mmi_boot_default (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function performs the first stage of the DiSPATCH boot procedure by resetting the DSP hardware and uploading the default P memory image. The X and Y memory regions are not initialized by this function.

Applications should normally use `mmi_start_firmware()`, instead of this function, to perform *all* stages of the boot procedure.

4.2.9 `mmi_diag_boot`

Prototype: `int mmi_diag_boot (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This diagnostic routine performs a standard DiSPATCH DSP initialization and boot procedure, while printing verbose status information to `stdout`. Since each step is announced and then performed discretely, this function is very useful for rapidly determining which part of the procedure is failing.

At the time of this writing, `mmi_diag_boot()` of DiSPATCH Package version 1.30 produces the following output for correct initialization:

```
                ** Diagnostic Boot **
1:        Un-reset the DSP and select VMEbus as the boot source.
2:        Making sure the DSP is ready for first word.
3:        Send the program word #1 to the host port.
4:        Waiting for the DSP to get word #1.
5:        Send the program word #2 to the host port.
6:        Waiting for the DSP to get word #2.
7:        Send the program word #3 to the host port.
8:        Waiting for the DSP to get word #3.
9:        Sending the remaining 8363 program words to host port.
10:       Checking if the DSP accepted the last program word.
11:       Activating firmware.  LED should light 1/2 brightness.
12:       Opening DSP device driver.
13:       Waiting for Stage 1 Boot Progress message.
14:       Uploaded X memory image.
15:       Uploaded Y memory image.
16:       Waiting for Stage 2 Boot Progress message.
17:       Probing running firmware.
 ** Initialization complete. **
```

The exact number and format of the boot stages may change in future versions, but the complete output should always end with "** Initialization complete. **".

4.2.10 `mmi_boot_file`

Prototype: `int mmi_boot_file (dsp_t dsp, char *filename);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`filename`	The filename of the DiSPATCH P memory image.

Returns: Zero on success, or `-1` on failure.

Description: This function performs the first stage of the DiSPATCH boot procedure by resetting the DSP hardware and uploading the specified P memory image. The X and Y memory regions are not initialized by this function.

The `filename` argument specifies a data file containing a raw DiSPATCH image to be uploaded to the DSP P memory space. To use the default firmware image built into the library, call `mmi_boot_default()` instead.

Applications should normally use `mmi_start_firmware()`, instead of this function, to perform *all* stages of the boot procedure.

4.2.11 `mmi_halt_dsp`

Prototype: `int mmi_halt_dsp (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function asserts the hardware reset for the specified DSP. This freezes all DSP operation and immediately halts any audio output.

After a hardware reset, the DSP must be re-initialized before DiSPATCH operation can be resumed. This can be accomplished with a call to `mmi_start_firmware()`. The DSP is then completely reset to the default startup state.

4.2.12 `mmi_poll_messages`

Prototype: `int mmi_poll_messages (void);`

Returns: Zero on success, or `-1` on failure.

Description: This routine services any messages that may be waiting in active DSPs. After all waiting messages have been read from the device driver and processed by the library, the routine returns. Any messages received from the DSP are stored in the library queues. If no messages are waiting to be read, then this function returns immediately.

This function can be used when a process is expecting a message from the DSP but does not wish to wait indefinitely for it by calling `mmi_wait_response()`. Instead, the process can call `mmi_poll_messages()` at convenient intervals, and then check for specific received messages with `mmi_check_response()`.

4.2.13 mmi_malloc

Prototype: `void *mmi_malloc (int size);`

Arguments:

`size`	Number of bytes to allocate.

Returns: Memory pointer on success, or `NULL` on failure.

Description: This is a very simple function to allocate a block of memory and test for failure. If the system fails to provide the requested memory, then the library will print an error message to `stderr` and return `NULL`.

This function is used within the library itself and made externally available only for convenience. It is not necessary for the application programmer to use this function to allocate memory.

4.2.14 `mmi_test_host_dram`

Prototype: `int mmi_test_host_dram (mmi_board_t mmi, int verbose);`

Arguments:

`mmi`	An MMI board handle returned by `mmi_open()`.
`verbose`	Non-zero value prints verbose progress messages.

Returns: Zero on success, or `-1` on RAM failure.

Description: This routine tests the host's access to the shared DRAM on an MMI board. No DSP is involved in this procedure.

If the `verbose` argument is non-zero, then the library will print status messages to `stdout` during the test. If `verbose` is zero, no messages will be printed; only the return value will indicate the test results.

The DRAM test is completely destructive, but the Command ID codes are automatically restored after the test. Any command arguments or audio data buffers will be lost during the test.

4.2.15 `mmi_wait_response`

Prototype: `int mmi_wait_response (dsp_t dsp, int type, int data);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`type`	DiSPATCH message type code.
`data`	Data portion of the expected message.

Returns: Zero on success, or `-1` on error.

Description: This function will wait indefinitely for a specific message from the DSP. The message is specified by two 24-bit values, a `type` word and a `data` word. The DiSPATCH message types are defined in "`dispatch.h`" and documented in the *DiSPATCH Firmware User's Manual*.

This function will first check the list of previously received and unclaimed DiSPATCH messages, and then wait for new messages if the expected message is not found. When the message is received from the DSP, it will be deleted from the list and `mmi_wait_response()` will return.

4.2.16 `mmi_complete`

Prototype: `int mmi_complete (dsp_t dsp, int token);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`token`	A completion token to wait for.

Returns: Zero on success, or `-1` on error.

Description: This function waits for any queued DiSPATCH command, such as `PLAY_BUFFER`, to be completed. Functions that result in completion messages, such as `mmi_play_buffer()`, return a completion token to the caller, as described in Section 4.1.7. By calling `mmi_complete()` with the given completion token, the application can pause until the response is received from the DSP.

This function is identical to calling `mmi_wait_response()` with a `MSGTYPE_COMPLETION` message type.

4.2.17 `mmi_check_response`

Prototype: `int mmi_check_response (dsp_t dsp, int msg_type, int msg_data);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`msg_type`	DiSPATCH message type code.
`msg_data`	Message data to check for.

Returns: Zero if the message is found, or `-1` if it has not been received.

Description: While `mmi_wait_response()` waits indefinitely for the specified message to be received from the DSP, this function will simply check the pending messages at the time it is called and return immediately.

A return value of zero indicates that the specified message was received from the DSP and is waiting in the library message list. The application should then call `mmi_wait_response()` to remove the message from the list.

4.2.18 `mmi_get_dsp_filedes`

Prototype: `int mmi_get_dsp_filedes (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: DSP file descriptor on success, or `-1` on failure.

Description: Some applications may not be well-suited to the library's standard message receiving functions, such as `mmi_wait_response()` or `mmi_complete()`. These programs may need to wait upon other file descriptors and call `select()` from inside the user application.

The function `mmi_get_dsp_filedes()` provides a means to retrieve the file descriptor for a given DSP. This descriptor can be used by the application for calls to `select()`, but should never be read from outside of the library.

4.2.19 `mmi_register_callback`

Prototype: `int mmi_register_callback (dsp_t dsp, int type, int data, func_t *func, void *arg1,`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`type`	Message type to link to callback.
`data`	Message data word to link to callback.
`func`	User function to call when message is received.
`arg1`	
`arg2`	
`arg3`	Optional user-defined arguments to be passed to callback.

Returns: Zero on success, or `-1` on failure.

Description: This function requests that a user function be called when a specific message arrives from the DSP. This response interface can be used instead of the `mmi_wait_response()` or `mmi_complete()` for increased efficiency.

When a DSP message is received that matches the callback, the specified user function will be called by the library as follows:

```
func (dsp, type, data, arg1, arg2, arg3);
```

The return value of the user function is ignored. Note that the DSP message will **not** be stored in the library queue. This means that the application is not required to delete it with a call to `mmi_wait_response()`. Messages that do not match any callbacks will be handled normally.

Each DSP message may only have **one** callback associated with it. After the first call to `mmi_register_callback()`, any calls with the same `type` and `data` will **replace** the current callback. Note that the callback assignments for one DSP are independent from those of any other DSPs. Each callback remains in place until the application calls `mmi_delete_callback()` to remove it.

Note that the callback mechanism is still in development at Release 1.50, and the interface is subject to change.

4.2.20 `mmi_delete_callback`

Prototype: `int mmi_delete_callback (dsp_t dsp, int type, int data);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`type`	Message type to unlink from callback.
`data`	Message data word to unlink from callback.

Returns: Zero on success, or `-1` on failure.

Description: This function removes a callback associated with a single DSP message. Messages matching the specified `type` and `data` will revert to being handled by the library queue mechanism.

A return value of zero indicates that the callback was found and removed, and `-1` is returned if no matching callback exists for this DSP.

Note that the callback mechanism is still in development at Release 1.50, and the interface is subject to change.

4.2.21 `mmi_init_cmd_codes`

Prototype: `int mmi_init_cmd_codes (mmi_board_t mmi);`

Arguments:

`mmi`	An MMI board handle returned by `mmi_open()`.

Returns: Zero on success, or `-1` on failure.

Description: One or more of each available DiSPATCH command is allocated a space in shared DRAM for its command code and arguments. This function initializes the Command ID field of each of these command blocks.

This function is called automatically during board initialization with `mmi_open()`, and also after any destructive RAM test procedure. If an application modifies or destroys the Command ID fields, then it must call `mmi_init_cmd_codes()` to restore the contents before any commands can be executed.

4.2.22 `mmi_start_dribble`

Prototype: `int mmi_start_dribble (char *filename);`

Arguments:

`filename`	Output file for messages, or "-" for `stdout`.

Returns: Zero on success, or `-1` on failure.

Description: This debugging function prints a message line to the specified output file when any DiSPATCH message is received from a DSP. An example message is show below:

```
[DSP A]  ACK:    0x1FDD9A   LED_CTRL
```

This shows the DSP that sent the message, the message type, the data word, and the command that is referenced at that address, if any.

This is strictly a debugging function for use during development. Since every DSP message generates a dribble message, the command response time and library processing overhead is increased. To disable dribble messages, call `mmi_end_dribble()`.

4.2.23 `mmi_end_dribble`

Prototype: `int mmi_end_dribble (void);`

Returns: Zero on success, or `-1` on failure.

Description: This disables the debugging message output started by calling `mmi_start_dribble()`. This function closes the output file, unless `stdout` was specified.

4.2.24 `mmi_discard_resp`

Prototype: `int mmi_discard_resp (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function can be used to discard all unclaimed responses from a given DSP. The library collects all responses into an internal list until they are each found and deleted by `mmi_wait_response()`. If the application does not regularly delete each response by calling `mmi_wait_response()`, it may be necessary to occasionally flush the list by calling `mmi_discard_resp()`.

The library will issue a warning if the response list grows to a large size without acknowledgement. A long list of unclaimed responses may also make the library less efficient, so the application should flush unwanted responses when convenient.

4.2.25 `mmi_parse_format`

Prototype: `int mmi_parse_format (char *name);`

Arguments:

`name`	A string format name to be parsed.

Returns: DiSPATCH format code on success, or `-1` for unrecognized format names.

Description: This is a simple function to return a DiSPATCH format code for a given format name string. The case of the format string is ignored. Since this is a common requirement for text option parsing, it is provided by the library. No MMI or DSP handles are required or used. At the present time, the following format names are understood by the function:

- PCM-16
- PCM16
- PCM-8
- PCM8
- VQ
- ULAW
- ALAW
- ADPCM
- BETA
- TONEGEN

4.2.26 `mmi_samples_per_word`

Prototype: `int mmi_samples_per_word (int format_code);`

Arguments:

`format_code`	An audio data format code.

Returns: Number of samples per word on success, or `-1` for invalid format codes.

Description: This function can be used to easily determine the number of audio samples in one 16-bit word for a given audio data format. This is especially useful to applications when computing buffer sizes, or the desired length (in words) for audio recording.

The return value indicates the number of samples packed into each 16-bit word, or `-1` if the specified audio format code was invalid or has no relevant size (like `TONEGEN`).

4.2.27 mmi_dsp_command

Prototype: `int mmi_dsp_command (dsp_t dsp, void *cmd_addr);`

Arguments:

dsp	A DSP handle returned by mmi_get_dsp().
cmd_addr	A pointer to the (virtual) start address of the Command Block.

Returns: Zero on success, or -1 on failure.

Description: This function provides a low-level interface to the command execution mechanism of DiSPATCH. The caller must provide the cmd_addr, a pointer to the start of the Command Block which must reside entirely on the MMI board.

Most applications will never need to call mmi_dsp_command() directly, since all DiSPATCH commands have a specific library function that implicitly executes the command internally. This function is made external only for special customizations and command prototyping.

4.3 DiSPATCH Command Functions

4.3.1 `mmi_load_dspmem`

Prototype:

```
int mmi_load_dspmem (dsp_t dsp, int space, unsigned long *buffer,
                     int addr, int wordcount);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`space`	Specifies which one of the three DSP memory spaces to load into.
`buffer`	The local address of the source data on the host.
`addr`	The DSP private RAM starting address to load into.
`wordcount`	The number of 24-bit DSP words to copy to the DSP.

Returns: Zero on success, or `-1` on failure.

Description: This function loads values from host memory into the DSPs private static RAM. Since this RAM is not directly accessible by the host, the library copies the data into shared DRAM, and then executes a `LOAD_x_MEM` command to request that the DSP transfer it into private memory.

The `space` argument must be one of `X_MEMORY`, `Y_MEMORY`, or `P_MEMORY`, as defined in “`dispatch.h`”.

The `buffer` argument points to the source data to be copied from the host. The data in the buffer is expected to be 32-bit raw integer values. The lower 24 bits of each source word will be loaded into DSP memory.

The `addr` argument specifies the DSP private RAM start address to load the data into. This is a 16-bit address ranging from `$0000` to `$FFFF`. The previous contents of the private memory will be overwritten. The `wordcount` parameter is the number of words to copy to the DSP.

Commands: `LOAD_X`
`LOAD_Y`
`LOAD_P`

4.3.2 mmi_load_dspmem_file

Prototype:

```
int mmi_load_dspmem_file (dsp_t dsp, int space, char *filename,
                                     int addr);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`space`	Specifies which one of the three DSP memory spaces to load into.
`filename`	The path and filename of the raw data file to read from.
`addr`	The DSP private RAM starting address to load into.

Returns: Zero on success, or `-1` on failure.

Description: This function loads values from a data file into the DSPs private static RAM. Since this RAM is not directly accessible by the host, the library copies the data into shared DRAM, and then executes a `LOAD_x_MEM` command to request that the DSP transfer it into private memory.

The `space` argument must be one of `X_MEMORY`, `Y_MEMORY`, or `P_MEMORY`, as defined in “`dispatch.h`”.

The `filename` parameter specifies a data file to supply the raw values to send to the DSP. The entire contents of the file, up to a maximum of 64K, will be transferred to the DSPs private memory. The file data is expected to be in raw binary format, with three bytes for each 24-bit DSP word. The lowest 8 bits must be in the first byte, the middle 8 bits in the second byte, and the high 8 bits in the third byte of each word.

The `addr` argument specifies the DSP private RAM start address to load the data into. This is a 16-bit address ranging from `$0000` to `$FFFF`. The previous contents of the private memory will be overwritten.

Commands: `LOAD_X`
`LOAD_Y`
`LOAD_P`

4.3.3 `mmi_register_counter`

Prototype: `int mmi_register_counter (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This routine enables the Play Counter, as described in the *DiSPATCH Firmware User's Manual*. The play counter keeps an updated count of the number of outgoing audio samples. The counter value is reset to zero by the `mmi_register_counter()` routine. After enabling the Play Counter, the current count can be read by calling `mmi_read_counter`.

Commands: `REGISTER_COUNTER`

4.3.4 `mmi_end_counter`

Prototype: `int mmi_end_counter (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: The Play Counter should be disabled when it is not being used, since it places an additional processing burden on the DSP. The `mmi_end_counter()` function tells the DSP to turn off the counter.

It is allowable to call `mmi_end_counter()` when the counter is not running.

Commands: `END_COUNTER`

4.3.5 `mmi_reset_counter`

Prototype: `int mmi_reset_counter (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: As described in Section 7.5 of the *DiSPATCH Firmware User's Manual*, The play counter begins at zero and increments until wrap-around. The application can call `mmi_reset_counter()` at any time to reset the value of the counter to zero.

Commands: `RESET_COUNTER`

4.3.6 `mmi_read_counter`

Prototype: `int mmi_read_counter (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Counter value on success, or `-1` on failure.

Description: This function fetches the current value of the play counter from DRAM and returns it as a positive 31-bit integer value.

If a counter is not presently enabled on this DSP, then this function will return `-1`.

4.3.7 `mmi_play_buf`

Prototype: `int mmi_play_buf (dsp_t dsp, int track, int start, int size);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The track number to use for this playback request.
`start`	The DRAM start address of the buffer.
`size`	The number of 16-bit words to be played.

Returns: DSP completion token on success, or `-1` on failure.

Description: This function instructs the DSP to play samples from a specified region of DRAM. The application is assumed to have already filled that memory with audio samples of the appropriate format for playback on this track.

The `start` parameter specifies the DRAM offset of the start of the buffer. Like all DRAM addresses, this address specifies a word offset from the beginning of DRAM. All audio buffers must start and end on 16-bit word boundaries.

The `size` parameter specifies the number of 16-bit words to be played. Depending on the data format in effect on the playback track, this may or may not be equal to the number audio samples in the buffer. For example, when using the PCM-16 audio format, each word represents one sample. However, when using the PCM-8 audio format, each word contains two 8-bit samples. The same `count` will play twice as long for the PCM-8 format than for the PCM-16 format. See `mmi_play_format()` for details on changing the audio data format for playback.

The return value of this function is a unique token for this playback request. See Section 4.1.7 for details. Token values are always positive integer values. A value of `-1` will be returned by the `mmi_play_buf()` function in the event of an error.

Commands: PLAY_BUFFER

4.3.8 `mmi_play_subbuffers`

Prototype:

```
int mmi_play_subbuffers (dsp_t dsp, int track, int start,
                         int size, int subcount,
                         int spacing);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The starting track number for the play requests.
`start`	The DRAM start address of the buffers.
`size`	The number of words in each sub-buffer.
`subcount`	The number of sub-buffers to mix.
`spacing`	Word offset between first sample of each sub-buffer.

Returns: Completion token on success, or `-1` on failure.

Description: This function starts playback of one or more sub-buffers to be mixed in parallel by the DSP. A complete description of sub-buffer mixing is given in Section 6.1 of the *DiSPATCH Firmware User's Manual*.

This function passes the given arguments directly to the DSP and returns the completion token for the command. See Section 4.1.7 for more information about the returned completion token. In the event of failure, a value of `-1` is returned.

Commands: PLAY_SUBBUFS

4.3.9 `mmi_record_buf`

Prototype: `int mmi_record_buf (dsp_t dsp, int start, int size);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`start`	The DRAM start address of the buffer.
`size`	The number of 16-bit words to be recorded.

Returns: DSP completion token on success, or `-1` on failure.

Description: This function instructs the DSP to record samples into a specified region of DRAM. Any data previously in this DRAM buffer will be overwritten by the DSP.

The `start` parameter specifies the DRAM offset of the start of the buffer. Like all DRAM addresses, this address specifies a word offset from the beginning of DRAM. All audio buffers must start and end on 16-bit word boundaries.

The `size` parameter specifies the number of 16-bit words to be recorded. The format of the incoming audio data depends on the recording data format presently in effect on the DSP. See `mmi_record_format()` to change the data format for recorded audio.

The return value of this function is a unique token for this record request. See Section 4.1.7 for details. Token values are always positive integer values. A value of `-1` will be returned by the `mmi_record_buf()` function in the event of an error.

Commands: RECORD_BUFFER

4.3.10 `mmi_monitor_buf`

Prototype: `int mmi_monitor_buf (dsp_t dsp, int start, int size);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`start`	The DRAM start address of the buffer.
`size`	The number of 16-bit words to be monitored.

Returns: Completion token on success, or `-1` on failure.

Description: This function is similar to `mmi_record_buf()`, but records the *outgoing* audio signal instead of the input signal. This can be used to make a copy of the playback audio after all processing has been performed by DiSPATCH. For more information on the playback monitor feature, see Section 7.4 of the *DiSPATCH Firmware User's Manual*.

The `start` parameter specifies the DRAM offset of the start of the buffer. Like all DRAM addresses, this address specifies a word offset from the beginning of DRAM. All audio buffers must start and end on 16-bit word boundaries.

The `size` parameter specifies the number of 16-bit words to be monitored. The format of the resulting audio data depends on the monitor data format presently in effect on the DSP. Use `mmi_monitor_format()` to change the data format for monitored audio.

The return value of this function is a unique token for this monitor request. See Section 4.1.7 for details. Token values are always positive integer values. A value of `-1` will be returned by the `mmi_record_buf()` function in the event of an error.

Commands: `MONITOR_BUFFER`

4.3.11 `mmi_play_file`

Prototype:

```
int mmi_play_file (dsp_t dsp, int track, char *filename,
                   int bufsize, int bufstart);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The playback track number to use.
`filename`	The audio data filename to play from, or “-” for `stdin`.
`bufsize`	The size of the data buffer to use for playback.
`bufstart`	The DRAM start offset for the playback data buffer.

Returns: Zero on success, or `-1` on failure.

Description: This function uses double-buffering to play an arbitrary-length audio file on a given DSP. The `track` argument specifies which of the play track is to be used for all play requests generated by this function. The track is assumed to be prepared for playback before calling `mmi_play_file()`. The sample rate and audio data format must be set correctly before calling this function.

The `filename` specifies a system file name to provide the audio data. The special filename of “-” indicates that the audio data is provided via `stdin` instead of a data file.

The `bufsize` argument determines the size of the memory region (in words) to use for the double-buffering. The `bufstart` argument specifies where this region will begin in DRAM. The memory region will be divided into two equal halves by the library for double-buffering. It is important to select a region that will not be used by other DSPs or simultaneous activities during playback.

This function returns after the file has finished playing.

Commands: PLAY_BUFFER

4.3.12 `mmi_record_file`

Prototype:

```
int mmi_record_file (dsp_t dsp, char *filename, int length,
                     int bufsize, int bufstart);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`filename`	The audio data filename to record to, or "-" for `stdout`.
`length`	The length of the recording, in words.
`bufsize`	The size of the data buffer to use for recording.
`bufstart`	The DRAM start offset for the recording data buffer.

Returns: Zero on success, or `-1` on failure.

Description: This function uses double-buffering to record an arbitrary-length audio file from a given DSP. The recording sample rate and audio data format must be set correctly before calling this function.

The `filename` specifies a system file name to receive the audio data. Any previous contents of the destination file will be overwritten. The special filename of "-" indicates that the audio data is to be written to `stdout` instead of a data file.

The `length` argument determines how long the recording will be. The length is always specified as a number of 16-bit words, regardless of the audio data format. The application should take into account the sample rate, stereo mode, and audio data format when determining the word length of the recording.

The `bufsize` argument determines the size of the memory region (in words) to use for the double-buffering. The `bufstart` argument specifies where this region will begin in DRAM. The memory region will be divided into two equal halves by the library for double-buffering. It is important to select a region that will not be used by other DSPs or simultaneous activities during recording.

This function returns after the file has finished recording.

Commands: `RECORD_BUFFER`

4.3.13 `mmi_set_play_format`

Prototype: `int mmi_set_play_format (dsp_t dsp, int track, int format);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The playback track number to use the new format.
`format`	A DiSPATCH audio data format code.

Returns: Zero on success, or `-1` on failure.

Description: This routine selects the audio data format for use on a given playback track and DSP. The new format takes effect immediately.

The format code must be one of those defined in "`dispatch.h`" and documented in the *DiSPATCH Firmware User's Manual*.

Commands: `SET_PLAY_FORMAT`

4.3.14 `mmi_set_record_format`

Prototype: `int mmi_set_record_format (dsp_t dsp, int format);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`format`	A DiSPATCH audio data format code.

Returns: Zero on success, or `-1` on failure.

Description: This routine selects the data format for audio recorded by the specified DSP. The new format takes effect immediately.

The format code must be one of those defined in "`dispatch.h`" and documented in the *DiSPATCH Firmware User's Manual*.

Commands: `SET_RECORD_FORMAT`

4.3.15 `mmi_set_monitor_format`

Prototype: `int mmi_set_monitor_format (dsp_t dsp, int format);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`format`	A DiSPATCH audio data format code.

Returns: Zero on success, or `-1` on failure.

Description: This routine selects the data format for audio monitored by the specified DSP. The new format takes effect immediately.

The format code must be one of those defined in “`dispatch.h`” and documented in the *DiSPATCH Firmware User’s Manual*.

Commands: `SET_MONITOR_FORMAT`

4.3.16 `mmi_set_play_gain`

Prototype:

```
int mmi_set_play_gain (dsp_t dsp, int track, double l_gain,
                                double r_gain);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	Playback track number, or `MASTER_PLAY_GAIN`.
`l_gain`	The new digital gain value for the left channel.
`r_gain`	The new digital gain value for the right channel.

Returns: Zero on success, or `-1` on failure.

Description: The DSP can digitally scale the volume level of each playback track in real-time. In addition, a global “master” play gain can be used to digitally scale the entire outgoing audio signal.

The gain levels for the track and master gains are set by this function. The floating-point gain parameters, `l_gain` and `r_gain` can range from 0.0 to 2.0, representing the gain scale from total silence (0% gain) to twice normal volume (200% gain). Values outside this range will be rejected.

The track number must be a valid track number, or the defined constant `MASTER_PLAY_GAIN`.

Note: On monophonic systems, the left and right play gains must be identical. Only stereo boards allow different gains for the left and right channels.

Since this gain adjustment is performed on the audio data *before* it is played by the digital-to-analog converter, the analog output gain should first be adjusted to provide the desired output signal level. (See `mmi_set_input_gain()`.)

Commands: `SET_PLAY_GAIN`

4.3.17 `mmi_set_record_gain`

Prototype: `int mmi_set_record_gain (dsp_t dsp, double l_gain, double r_gain);`

Arguments:

`l_gain`	The new digital gain value for the left channel.
`r_gain`	The new digital gain value for the right channel.

Returns: Zero on success, or `-1` on failure.

Description: The DSP can digitally scale the volume level of the incoming recorded audio in real-time.

The digital recording gain level is set by this function. The floating-point gain parameters, `l_gain` and `r_gain` can range from 0.0 to 2.0, representing the gain scale from total silence (0% gain) to twice normal volume (200% gain). Values outside this range will be rejected.

Since this gain adjustment is performed on the audio data *after* it has been digitized by the analog-to-digital converter, the analog input gain should first be adjusted to match the input signal. (See `mmi_set_input_gain()`.)

Note: On monophonic systems, the left and right record gains must be identical. Only stereo boards allow different gains for the left and right channels.

Commands: `SET_RECORD_GAIN`

4.3.18 `mmi_play_ctrl`

Prototype: `int mmi_play_ctrl (dsp_t dsp, int run);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`run`	New run state: `0` = pause, `1` = run.

Returns: Zero on success, or `-1` on failure.

Description: This function instructs the DSP to pause or resume audio playback. When playback is paused, all play requests are frozen and there is no audio output from the DSP. When playback is then resumed, audio output activity continues from where it was paused.

Commands: `PAUSE_PLAY`
`RESUME_PLAY`

4.3.19 `mmi_record_ctrl`

Prototype: `int mmi_record_ctrl (dsp_t dsp, int run);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`run`	New run state: `0` = pause, `1` = run.

Returns: Zero on success, or `-1` on failure.

Description: This function instructs the DSP to pause or resume audio recording. When recording is paused, all record requests are frozen, and the sidetone signal is silenced. When recording is then resumed, recording activity continues from where it was paused.

Commands: `PAUSE_RECORD`
`RESUME_RECORD`

4.3.20 `mmi_play_position`

Prototype:

```
int mmi_play_position (dsp_t dsp, int track, int *addr,
                                int *count);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`*addr`	The current DRAM position of the active play request.
`*count`	The remaining number of words in the active play request.

Returns: Zero on success, or `-1` on failure.

Description: This function can be used to approximate the instantaneous status of the playback request currently being processed by the DSP. This allows the host to check where in DRAM the outgoing audio data is being read from, and how many words are remaining in the current play buffer.

The `addr` argument must be a valid pointer to an integer variable. The `mmi_play_position()` function will store the current DRAM offset into this variable. The `count` argument must be a valid pointer to an integer variable. This variable will indicate the number of 16-bit sample words remaining to be played from the DRAM buffer.

Since this is an instantaneous status report, the playback may have progressed before the host has processed the given play position. For this reason, the reported position should be considered an approximation of the current playback status. Note, however, that the status is always accurate when playback is paused.

Commands: PLAY_POS

4.3.21 mmi_record_position

Prototype: `int mmi_record_position (dsp_t dsp, int *addr, int *count);`

Arguments:

dsp	A DSP handle returned by mmi_get_dsp().
*addr	The current DRAM position of the active record request.
*count	The remaining number of words in the active record request.

Returns: Zero on success, or -1 on failure.

Description: This function can be used to approximate the instantaneous status of the record request currently being processed by the DSP. This allows the host to check where in DRAM the incoming audio data is being placed, and how many words are remaining in the current record buffer.

The addr argument must be a valid pointer to an integer variable. The mmi_record_position() function will store the current DRAM offset into this variable. The count argument must be a valid pointer to an integer variable. This variable will indicate the number of 16-bit sample words remaining to be placed into the DRAM buffer.

Since this is an instantaneous status report, the recording may have progressed before the host has processed the given record position. For this reason, the reported position should be considered an approximation of the current recording status. Note, however, that the status is always accurate when recording is paused.

Commands: RECORD_POS

4.3.22 `mmi_monitor_position`

Prototype: `int mmi_monitor_position (dsp_t dsp, int *addr, int *count);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`*addr`	The current DRAM position of the active monitor request.
`*count`	The remaining number of words in the active monitor request.

Returns: Zero on success, or `-1` on failure.

Description: This function can be used to approximate the instantaneous status of the playback monitor request currently being processed by the DSP. This allows the host to check where in DRAM the monitored audio data is being written to, and how many words are remaining in the current monitor buffer.

The `addr` argument must be a valid pointer to an integer variable. The `mmi_monitor_position()` function will store the current DRAM offset into this variable. The `count` argument must be a valid pointer to an integer variable. This variable will indicate the number of 16-bit sample words remaining to be stored into the DRAM buffer.

Since this is an instantaneous status report, the playback and monitoring may have progressed before the host has processed the given monitor position. For this reason, the reported position should be considered an approximation of the current monitor status. Note, however, that the status is always accurate when playback is paused.

Commands: `MONITOR_POS`

4.3.23 `mmi_abort_track`

Prototype: `int mmi_abort_track (dsp_t dsp, int track);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	Playback track number to abort.

Returns: Zero on success, or `-1` on failure.

Description: This aborts all pending playback requests on the specified track. No completion messages are sent for the aborted track. Pending record requests and playback requests on other tracks are unaffected by this function.

Commands: `ABORT_TRACK`

4.3.24 `mmi_abort_all_play`

Prototype: `int mmi_abort_all_play (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This aborts all pending playback requests on all tracks. No completion messages are sent for the aborted play requests.

Commands: ABORT_ALL_PLAY

4.3.25 `mmi_abort_record`

Prototype: `int mmi_abort_record (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function aborts all pending record requests. No completion messages are sent for the aborted record requests.

Commands: `ABORT_RECORD`

4.3.26 `mmi_abort_monitor`

Prototype: `int mmi_abort_monitor (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function aborts all pending monitor requests. No completion messages are sent for the aborted monitor requests.

Commands: `ABORT_MONITOR`

4.3.27 `mmi_led_ctrl`

Prototype: `int mmi_led_ctrl (dsp_t dsp, int led_id, int state);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`led_id`	Which LED to control, counting from 0.
`state`	The new state for the specified LED: `1` = on, `0` = off.

Returns: Zero on success, or `-1` on failure.

Description: This function asks the DSP to change the state of one front-panel LED indicator.

On the MMI-420, each DSP controls one front-panel LED. The `led_id` parameter must always be zero for this model.

On the MMI-210 and MMI-105 boards, there are two LEDs. Both of the front-panel LEDs must be controlled by DSP A. DSP B should not be used to set the LED states. The green LED is specified by `led_id` zero, while the red LED is led_id one.

Commands: LED_CTRL

4.3.28 `mmi_set_input_gain`

Prototype: `int mmi_set_input_gain (dsp_t dsp, int gain);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`gain`	A gain value between 0 and 255, inclusive.

Returns: Zero on success, or `-1` on failure.

Description: This function sets the analog input gain level for the analog-to-digital hardware. The gain level may be changed at any time, and the new gain level takes effect immediately.

The actual analog gain depends on the MMI model in use, but all boards use the same range of 0 to 255, with higher numbers resulting in more signal gain.

Commands: `SET_INPUT_GAIN`

4.3.29 `mmi_set_output_gain`

Prototype: `int mmi_set_output_gain (dsp_t dsp, int gain);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`gain`	A gain value between 0 and 255, inclusive.

Returns: Zero on success, or `-1` on failure.

Description: This function sets the analog output gain level for the digital-to-analog hardware. The gain level may be changed at any time, and the new gain level takes effect immediately.

The actual analog gain depends on the MMI model in use, but all boards use the same range of 0 to 255, with higher numbers resulting in more signal gain.

Commands: `SET_OUTPUT_GAIN`

4.3.30 `mmi_set_mixer`

Prototype: `int mmi_set_mixer (dsp_t dsp, int setting);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`setting`	The new analog mixer setting.

Returns: Zero on success, or `-1` on failure.

Description: Some MMI models have multiple analog output ports and a programmable analog router. This router can direct the audio output of one or more DSPs to a given analog audio port on the front panel.

For example, a mixer may be configured to direct the combined audio output of DSP A and DSP B to the audio output jack labeled OUTPUT D. At the same time, the output from DSP C could be directed to OUTPUT C and OUTPUT A.

The `mmi_set_mixer()` function configures the programmable mixer on boards with such hardware. The `dsp` argument selects a single DSP to control. The routing of the audio from this DSP is specified by the bits in `setting`. Each bit corresponds to one available analog output port, starting with the lowest bit, as documented in the *DiSPATCH Firmware User's Manual*.

For example, to send the signal to OUTPUT B only, the value of `setting` would be `0x2`. To send to OUTPUT A and OUTPUT C, pass a value of `0x5`, and a value of `0xF` selects all of the first four output ports.

Commands: `ROUTE_OUTPUT`

4.3.31 `mmi_set_speed_change`

Prototype: `int mmi_set_speed_change (dsp_t dsp, int percent);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`percent`	The new relative playback speed, in percent.

Returns: Zero on success, or `-1` on failure.

Description: This controls the real-time pitch-corrected speed-change algorithm performed by the DSP. The algorithm is capable of slowing the speed of playback to 50% of normal speed, or accellerating it to 200% of normal speed.

The `percent` argument specifies the new speed factor, relative to normal speed. A `percent` value of `75` instructs the DSP to play all audio at 75% of normal speed, slowing it down. The `percent` parameter must be between `50` and 200, inclusive.

The speed-change algorithm is automatically disabled by the DSP when the `percent` value is exactly `100`.

Commands: `SPEED_CHANGE`

4.3.32 `mmi_set_resample`

Prototype: `int mmi_set_resample (dsp_t dsp, int multiplier);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`multiplier`	Resampling factor (`1` or `4`).

Returns: Zero on success, or `-1` on failure.

Description: This controls the real-time multi-rate filter used to lower the effective audio sample rate for playback and recording. This filter provides the transparent emulation of sample rates that are exactly one quarter of the hardware settings.

The `multiplier` argument specifies the new sample rate factor. A value of `4` will divide the hardware sample rate by four, while a `multiplier` of `1` will disable the multi-rate filter and return to the normal (hardware) sample rate.

See the `RESAMPLE` command documentation in the *DiSPATCH Firmware User's Manual* for important usage details and restrictions.

Commands: `RESAMPLE`

4.3.33 `mmi_set_sidetone`

Prototype: `int mmi_set_sidetone (dsp_t dsp, double sidetone);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`sidetone`	New sidetone gain level from 0.0 to 2.0, inclusive.

Returns: Zero on success, or `-1` on failure.

Description: This function enables or disables the audio sidetone feature and sets the sidetone gain level. When sidetone gain is set to be greater than 0.0, some amount of the incoming signal is directly routed to the digital output. This pass-through signal has very low latency and is well suited for headset voice feedback.

The `sidetone` gain value can range from 0.0 to 2.0 (inclusive), representing the range between silence (no sidetone), to a signal amplification of 200%. A `sidetone` value of 1.0 passes the incoming signal through unchanged. This loopback setting is useful for adjusting the analog input and output gain levels for optimum performance.

When the gain is set to 0.0, the sidetone feature is automatically disabled by the DSP to conserve processing bandwidth.

Commands: `SET_SIDETONE`

4.3.34 mmi_transform_buffer

Prototype:

```
int mmi_transform_buffer (dsp_t dsp, int src_fmt, int src_addr,
                          int src_count, int dest_fmt,
                          int dest_addr, int subcount, int spacing);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`src_fmt`	The format code of the source data.
`src_addr`	The DRAM address offset of the source buffer.
`src_count`	The number of data words in the source buffer.
`dest_fmt`	The format code for the resulting data.
`dest_addr`	The DRAM start address of the destination buffer.
`subcount`	The number of sub-buffers in the input buffer.
`spacing`	Word offset between first sample of each sub-buffer.

Returns: Number of resulting data words, or `-1` on failure.

Description: This function asks the DSP to convert a block of audio data in DRAM from one data format into a different format. Multiple audio buffers can be mixed in parallel to produce one converted block of data. All audio processing is performed in-RAM, with no analog audio input or output. The conversion runs as fast as possible, and this function does not return until the entire buffer has been converted.

The transform arguments and operation are described in detail in the *DiSPATCH Firmware User's Manual*, under `TRANSFORM`.

The DSP will suspend all normal play and record processing while the conversion is in progress. Also, note that the source and destination buffer regions should not overlap in memory.

Commands: `TRANSFORM`

4.3.35 `mmi_transform_file`

Prototype:

```
int mmi_transform_file (dsp_t dsp, int src_fmt, char *src_filename,
                        int dest_fmt, char *dest_filename);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`src_fmt`	Audio data format for the source file.
`src_filename`	Filename of audio source data to be converted.
`dest_fmt`	Audio data format for the destination file.
`dest_filename`	Filename for converted output data.

Returns: Zero on success, or `-1` on failure.

Description: This is a convenient function to convert a given audio file from one DiSPATCH format to another. The library opens both files, reads one block at a time into DRAM, calls `mmi_transform_buffer()`, and writes it to the output file.

The entire file is converted and there is no limit on file length. Note that the library uses a region of DRAM starting at address zero, so any previous contents in that region are destroyed. The size of this transfer region (presently 240,000 words) is defined in “`lib_local.h`” as `TRANSFORM_BLOCKSIZE`.

Commands: `TRANSFORM`

4.3.36 mmi_count_buffers

Prototype:

```
int mmi_count_buffers (dsp_t dsp, int *play, int *rec,
                       int *monitor);
```

Arguments:

dsp	A DSP handle returned by `mmi_get_dsp()`.
*play	The number of pending play requests.
*rec	The number of pending record requests.
*monitor	The number of pending monitor requests.

Returns: Zero on success, or `-1` on failure.

Description: This function reports the number of requests that are currently queued for processing. The counts include the buffer currently in progress.

The value of the `play` variable will be set to the number of playback buffers. The value of `record` and `monitor` will reflect the number of pending record and monitor requests, respectively.

Commands: `COUNT_BUFFERS`

4.3.37 `mmi_clip_led`

Prototype: `int mmi_clip_led (dsp_t dsp, int ceiling);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`ceiling`	Peak sample value, or zero to disable.

Returns: Zero on success, or `-1` on failure.

Description: This function enables or disables the clipping LED indicator. The clipping LED can be used to help set the audio input level to maximize the dynamic range while avoiding distortion from clipping.

The value of the `ceiling` argument sets the minimum 16-bit sample value required to light the front-panel LED for `dsp`. A `ceiling` value of zero will disable the clipping indicator and return the LED to manual control.

See the `CLIP_LED` command documentation in the *DiSPATCH Firmware User's Manual* for usage restrictions and suggested `ceiling` values.

Commands: `CLIP_LED`

4.3.38 `mmi_enable_measurements`

Prototype: `int mmi_enable_measurements (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function enables continuous input peak and bias measurements on the DSP. This processing is disabled by default. Before any calls to `mmi_input_peak()` or `mmi_input_bias()` can be made, these measurements must be enabled by a call to `mmi_enable_measurements()`.

When input measurements are no longer needed, they should be disabled with `mmi_disable_measurements()` for efficiency. Any extraneous calls to `mmi_disable_measurements()` while measurements are disabled will be safely ignored.

Note that no measurements are computed while recording is paused.

Commands: `ENABLE_MEASURE`

4.3.39 `mmi_disable_measurements`

Prototype: `int mmi_disable_measurements (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function should be called to inhibit the measurement computations if input measurements have been enabled and they are no longer needed. Measurements can later be re-enabled by another call to `mmi_enable_measurements()`. Any extraneous calls to `mmi_disable_measurements()` while measurements are disabled will be safely ignored.

Commands: `DISABLE_MEASURE`

4.3.40 `mmi_input_peak`

Prototype: `int mmi_input_peak (dsp_t dsp, double *left, double *right);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`left`	Peak input level of the left channel (`0.0` to `1.0`).
`right`	Peak input level of the right channel (`0.0` to `1.0`).

Returns: Zero on success, or `-1` on failure.

Description: This function provides the peak input signal level for each of the left and right audio channels. During mono operation, both of these values will be equal. The floating-point signal levels range from `0.0` (representing silence), to `1.0` (fully saturated).

The input peak level represents the single highest amplitude sample since the last call to `mmi_input_peak()`. This information can be used to warn the user of potential clipping, or to simulate a VU-meter.

This function is only accepted after input measurements have been enabled with `mmi_enable_measurements()`.

Commands: `INPUT_PEAK`

4.3.41 `mmi_input_bias`

Prototype: `int mmi_input_bias (dsp_t dsp, int *left, int *right, int *samples);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`left`	Input bias of the left channel (`0.0` to `1.0`).
`right`	Input bias of the right channel (`0.0` to `1.0`).
`samples`	Number of samples used in the bias computation.

Returns: Zero on success, or `-1` on failure.

Description: This function returns the input bias level for each of the left and right audio channels. During mono operation, both of these values will be equal.

The reported bias levels range from `-32768` to `32767`. The ideal bias level for silent input is `0`. The bias levels are an arithmetic mean value of a large number of sequential input samples. The number of input samples used in the bias computation is returned in the `samples` variable.

This function is only accepted after input measurements have been enabled with `mmi_enable_measurements()`.

Commands: `INPUT_BIAS`

4.3.42 mmi_signal_detect

Prototype:

```
int mmi_signal_detect (dsp_t dsp, int detector,
                       double voice_thresh, double energy_thresh,
                       int up_notify, int down_notify,
                       func_t *callback, void *data);
```

Arguments:

dsp	A DSP handle returned by `mmi_get_dsp()`.
detector	Which detector to configure (0 through 3).
voice_thresh	The required ratio of voice-band to total energy.
energy_thresh	The required amount of total energy.
up_notify	The number of energy periods to trigger an UP transition.
down_notify	The number of energy periods to trigger a DOWN transition.
callback	A function to be called when a transition is detected, or `NULL`.
data	Optional user-defined argument to be passed to callback.

Returns: Zero on success, or `-1` on failure.

Description: This function configures one of the available signal detectors in DiSPATCH. The application is free to change the parameters of any detector at any time during operation.

There are several independent signal detectors available on each DSP. The number of available detectors is defined as the constant `NUM_DETECTORS` in "`dispatch.h`". This is currently 4. Each detector can have unique parameters and callbacks.

For a detailed description of the four signal detection control parameters, consult Section 8.2 of the *DiSPATCH Firmware User's Manual*. To disable a signal detector, set both the `voice_thresh` and the `energy_thresh` to zero.

The `callback` parameter is an optional application-defined callback function to be called whenever DiSPATCH detects a signal state transition. This function will be called by the library as follows:

```
callback (detector, state, address, data);
```

The three arguments passed to the function are:

`detector`: This is the number of the detector that has reported a signal transition.

`state`: This is the new state of the detector. A value of `1` indicates the signal is now present, while `0` indicates not-present.

`address`: This is the address offset in shared DRAM where the transition took place, if recording is in progress. This value is undefined if no recording is in progress.

`data`: Arbitrary data pointer to be used by the application. This is usually a pointer to a user-defined data structure with information concerning the signal detector or DSP.

The return value of the callback function, if any, is ignored. If the callback function address is specified as `NULL`, then no function will be called on transition.

Commands: SIGNAL_DETECT

4.3.43 `mmi_set_equalizer`

Prototype: `int mmi_set_equalizer (dsp_t dsp, int enable, int level[10]);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`enable`	Enable (1) or disable (0) the equalizer feature.
`level`	An array of ten integers, representing the gain for each band.

Returns: Zero on success, or `-1` on failure.

Description: This function controls the real-time frequency equalizer. The operation and restrictions of this feature are documented in the *DiSPATCH Firmware User's Manual*, under the `EQUALIZER` command.

The `enable` flag controls whether the equalizer is enabled or disabled. Because the equalizer is computationally intensive, it should be disabled when not in use.

The `level` argument is an array of exactly ten integers, each ranging from -6553 to 32767. These represent the gain setting of each of the ten frequency bands, starting with the lowest frequency (31 Hz).

Commands: `EQUALIZER`

4.3.44 `mmi_set_reverb`

Prototype: `int mmi_set_reverb (dsp_t dsp, double gain, int delay);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`gain`	Feedback gain setting, from 0.0 to 2.0.
`delay`	Length of the digital delay line (0 to 2046).

Returns: Zero on success, or `-1` on failure.

Description: This function controls the crude and simple "reverb" function in the DiSPATCH play processor. The `gain` parameter selects how much of the outgoing signal is placed in the delay loop, and the `delay` argument specifies how long (in samples) the feedback delay will be.

Refer to the REVERB command description in the *DiSPATCH Firmware User's Manual* for more information on the reverb feature.

Commands: REVERB

4.3.45 `mmi_select_input`

Prototype: `int mmi_select_input (dsp_t dsp, int input);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`input`	Audio input source; `SOURCE_MIC` for microphone or `SOURCE_LINE` for Line-In.

Returns: Zero on success, or `-1` on failure.

Description: On audio boards with selectable analog input sources, this function selects which audio source will be used during recording. The available sources are represented by the `SOURCE_MIC` and `SOURCE_LINE` constants defined in "`dispatch.h`", representing microphone input and line-level input, respectively.

Commands: `SELECT_INPUT`

4.3.46 `mmi_stereo_mode`

Prototype: `int mmi_stereo_mode (dsp_t dsp, int enable);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`enable`	`1` for stereo mode, `0` for mono.

Returns: Zero on success, or `-1` on failure.

Description: Some MMI boards support a sample-interleaved stereo mode of operation. On those boards, stereo can be enabled or disabled with this function.

When stereo is enabled, the left and right audio channels will receive alternating samples, beginning with the left channel. All audio buffers are assumed to be in sample-interleaved format when stereo mode is enabled.

Because DiSPATCH does not yet support full algorithmic processing of stereo data, *only PCM sample formats are supported in stereo mode*. The allowed formats include PCM-16, PCM-8, μ-Law, and A-Law. All other formats are restricted to monophonic operation.

Note that stereo mode effectively *doubles* the number of samples for a given time period, and likewise doubles the required computational demands on the DSP. Be aware that stereo operation at high sample rates will leave little remaining bandwidth for other DiSPATCH features.

On the MMI-210, stereo operation requires that *only one* DSP is operational. The other DSP must be left in the **reset** state and must not be initialized or used.

Commands: `STEREO_MODE`

4.3.47 `mmi_query_stereo`

Prototype: `int mmi_query_stereo (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero for mono, one for stereo, or `-1` on error.

Description: This command can be used to determine if the DSP is currently operating in stereophonic mode. If DiSPATCH is operating in mono (the default), a zero is returned. If stereo is engaged, a one is returned.

QUERY_STATE

4.3.48 `mmi_filter_play`

Prototype: `int mmi_filter_play (dsp_t dsp, int enable);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`enable`	Enable (1), or disable (0) the playback FIR filter.

Returns: Zero on success, or `-1` on failure.

Description: This function enables or disables the playback FIR filter. The filter is a simple user-definable finite impulse response (FIR) filter that is applied to all outgoing audio data. The FIR filter coefficients and number of taps must be declared with `mmi_load_play_fir()` before enabling the filter.

The play filter can not be used when stereo mode is enabled.

Commands: PLAY_FIR

4.3.49 `mmi_filter_record`

Prototype: `int mmi_filter_record (dsp_t dsp, int enable);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`enable`	Enable (1), or disable (0) the recording FIR filter.

Returns: Zero on success, or `-1` on failure.

Description: This function enables or disables the recording FIR filter. The filter is a simple user-definable FIR filter that is applied to the incoming audio data stream. The FIR filter coefficients and number of taps must be declared with `mmi_load_record_fir()` before enabling the filter.

The record filter can not be used when stereo mode is enabled.

Commands: `RECORD_FIR`

4.3.50 `mmi_load_play_fir`

Prototype: `int mmi_load_play_fir (dsp_t dsp, char *coeff_file);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`coeff_file`	Name of the filter coefficient data file.

Returns: Zero on success, or `-1` on failure.

Description: The playback FIR filter requires that a set of filter coefficients be loaded by the application. These coefficients directly control the characteristics of the FIR filter, and are usually generated by a separate filter design program. The play and record FIR filters have independent control parameters and coefficients.

The named data file is expected to contain the raw 24-bit values for the filter coefficients, in high/middle/low byte order. The entire contents of the data file will be read into DRAM and transferred to the DSP. The number of filter coefficients (taps) is determined from the length of the data file (i.e. length in bytes divided by three).

The FIR filter may be have from 1 to 256 taps. Excessively big data files will cause an error (`-1`) to be returned.

Commands: PLAY_FIR

4.3.51 `mmi_load_record_fir`

Prototype: `int mmi_load_record_fir (dsp_t dsp, char *coeff_file);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`coeff_file`	Name of the filter coefficient data file.

Returns: Zero on success, or `-1` on failure.

Description: The record FIR filter requires that a set of filter coefficients be loaded by the application. These coefficients directly control the characteristics of the FIR filter, and are usually generated by a separate filter design program. The play and record FIR filters have independent control parameters and coefficients.

The named data file is expected to contain the raw 24-bit values for the filter coefficients, in high/middle/low byte order. The entire contents of the data file will be read into DRAM and transferred to the DSP. The number of filter coefficients (taps) is determined from the length of the data file (i.e. length in bytes divided by three).

The FIR filter may be have from 1 to 256 taps. Excessively big data files will cause an error (`-1`) to be returned.

Commands: `RECORD_FIR`

4.3.52 `mmi_set_srate`

Prototype: `int mmi_set_srate (dsp_t dsp, int play, int record);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`play`	The new playback sample rate (in Hertz).
`record`	The new recording sample rate (in Hertz).

Returns: Zero on success, or `-1` on failure.

Description: This function sets the playback and recording sample rates. Since each MMI board has unique provisions and restrictions for sample rates, this function will validate the specified sample rates and return `-1` if the values are not permitted for the given model. If the sample rates are acceptable, they will be passed to the DSP and put into use. See hardware description sections of the *DiSPATCH Firmware User's Manual* for more information about the allowable sample rates for a specific MMI model.

A special sample rate value of `-1` can be passed to `mmi_set_srate` as a wild-card for either the `play` or the `record` argument. The library will then select an acceptable sample rate to pair with the specified rate. For example, to select an 8000 Hz playback rate without regard for the record rate, use 8000 for `play`, and `-1` for `record`.

Commands: SET_SRATE

4.3.53 `mmi_request_srate`

Prototype: `int mmi_request_srate (dsp_t dsp, int play, int record);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`play`	The sample rate for playback (in Hertz).
`record`	The sample rate for recording (in Hertz).

Returns: Zero on success, or `-1` on failure.

Description: This function asks the DSP to select the specified sample rates. Applications should call the `mmi_set_srate()` function instead, which will ensure that the requested sample rates are valid for the MMI model in use.

Commands: SET_SRATE

4.3.54 `mmi_set_pnm`

Prototype: `int mmi_set_pnm (dsp_t dsp, int p, int n, int m);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`p`	A 16-bit value to be sent to the programmable frequency synthesizer.
`n`	A 16-bit value to be sent to the programmable frequency synthesizer.
`m`	A 16-bit value to be sent to the programmable frequency synthesizer.

Returns: Zero on success, or `-1` on failure.

Description: This function is rendered **obsolete** by the `mmi_set_srate()` function, which allows the DSP to compute and use precise P, N, and M values for a given frequency.

This function is left in the library for experimental use or non-standard frequency generation hardware. The three arguments are passed directly to the DSP, which then sends them to the programmable frequency synthesizer (PFS) upon execution of the `SET_PNM` command.

Commands: `SET_PNM`

4.3.55 `mmi_play_tone`

Prototype:

```
int mmi_play_tone (dsp_t dsp, int track, int srate,
                   mmi_tone_t tone);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The playback track number to generate the tone.
`srate`	The current playback sample rate (in Hertz).
`tone`	The structure describing the tone to be generated.

Returns: Zero on success, or `-1` on failure.

Description: This function controls tone generation. The *DiSPATCH Tone Generation Manual* describes the operation and parameters of tone generation, as well as the library interface to this feature.

The `track` used to generate the tone must be previously configured (via `mmi_set_play_format()`) for the `TONEGEN` audio data format. The `srate` argument must reflect the current playback sample rate in Hertz. Note that stereo operation does not permit tone generation.

The `tone` argument is a pointer to a `struct mmi_tone`. The fields of this structure fully specify a tone to be generated. This structure is defined by "`dispatch.h`". See the *DiSPATCH Tone Generation Manual* for details on the member fields of this structure.

Commands: `TONEGEN`

4.3.56 `mmi_ramtone`

Prototype:

```
int mmi_ramtone (dsp_t dsp, int track, int format,
                 int address, int count);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The playback track number to provide the tone parameters.
`format`	A DiSPATCH audio data format code for generated data.
`address`	The DRAM start address of the destination buffer.
`count`	The number of audio samples to generate.

Returns: Number of words generated, or `-1` on failure.

Description: This function generates a tone into DRAM. Instead of playing the specified audio tone, the sample data is written to a specified region of shared memory. The host can then read it from DRAM or play this block of tone data at a later time (using `PLAY_BUFFER`).

The *DiSPATCH Tone Generation Manual* describes the operation and parameters of tone generation, as well as the library interface to this feature.

The `track` passed to `mmi_ramtone()` must be previously configured via `mmi_play_tone()` for the desired tone characteristics. If that track is not configured for the `TONEGEN` audio data format, then the tone parameters will not have an impact on any other playback.

The audio data `format` specified to `mmi_ramtone()` selects which audio data representation will be used for the data written to DRAM.

The `address` and `count` parameters specify the region of DRAM to be used for tone data. Note that the `count` is specified in *samples*, not output words. Upon success, the `mmi_ramtone()` function returns the number of generated words. This may or may not equal the number of samples, depending on the data format.

If the length of the tone, as specified in the tone structure, is less than the requested `count` of samples, then the tone will finish early, and the return value will indicate that fewer words were generated than requested. This allows the host to repeatedly call `mmi_ramtone()` until a tone of unknown length is finished playing. Put another way, the number of samples generated by a call to `mmi_ramtone()` is the **smaller of** the remaining tone length and the specified `count`.

All active playback and recording are **paused** while the tone generation is in progress. This function will not return until the requested tone buffer is complete. This is similar to the behavior of the `mmi_transform()` function.

Commands: `RAMTONE`

4.3.57 `mmi_end_tone`

Prototype: `int mmi_end_tone (dsp_t dsp, int track);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`track`	The track number of the tone.

Returns: Zero on success, or `-1` on failure.

Description: This function terminates any synthesized tone that may be playing on the specified track. The tone is terminated immediately upon execution, and no completion token is returned for the tone.

This function simply changes the length of the tone to zero, effectively terminating it.

Commands: `TONEGEN`

4.3.58 `mmi_load_table`

Prototype: `int mmi_load_table (dsp_t dsp, int table, int addr);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`table`	The wavetable number.
`addr`	The DRAM start address of the new 256 table values.

Returns: Zero on success, or `-1` on failure.

Description: This function copies a block of 256 sample values (each 16-bit) from DRAM into the DSPs internal wavetables, overwriting the previous table contents.

The format of the wavetable data is described in the *DiSPATCH Firmware User's Manual*, in the `LOAD_TABLE` command description.

Commands: `LOAD_TABLE`

4.3.59 `mmi_waveshape`

Prototype: `int mmi_waveshape (dsp_t dsp, int table);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`table`	Which table to use for wave mapping.

Returns: Zero on success, or `-1` on failure.

Description: This function controls the waveshaping post-processing play module. This is presently an experimental and unsupported feature for remapping waveform sample values.

The table number may be any of the available wavetables, or `-1` to disable the waveshaping module.

Commands: `WAVESHAPE`

4.3.60 `mmi_fetch_error`

Prototype: `int mmi_fetch_error (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This routine will execute the `FETCH_ERROR` DiSPATCH command and print the current DSP error status. The `mmi_dsp_command()` function calls this routine automatically whenever a DSP command returns an with an error condition. User applications can choose to explicitly read the error status by calling this function directly.

If a second error condition results from attempting to execute the `FETCH_ERROR` command, then the `mmi_fetch_error()` function will return `-1` instead of recursing. If the `FETCH_ERROR` command fails, then there is an unrecoverable condition in the DiSPATCH protocol, most likely caused by corrupted Command IDs in upper DRAM.

Commands: `FETCH_ERROR`

4.3.61 `mmi_enable_mail`

Prototype: `int mmi_enable_mail (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: The message mailbox is a simple text message passing system used during the development and debugging of the DiSPATCH DSP firmware. Host applications should not enable mail by calling this function.

Mail messages are out-of-band data received from the DSP during DiSPATCH operation. By default, mail messages will not be sent to the host unless requested.

Commands: `ENABLE_MAIL`

4.3.62 `mmi_disable_mail`

Prototype: `int mmi_disable_mail (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This routine will tell the DSP to inhibit any mailbox messages that may have been enabled by `mmi_enable_mail()`. This is the default operating state for each DSP upon initialization.

Commands: `DISABLE_MAIL`

4.3.63 `mmi_probe_firmware`

Prototype: `int mmi_probe_firmware (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero if the specified DSP is correctly running the DiSPATCH firmware, or `-1` if it is not.

Description: This function performs a quick test to verify that the DSP is booted and running the DiSPATCH firmware program. If the library has not booted the specified DSP, then this function will immediately return `-1`.

If the DSP state indicates that the firmware should be running, then the `LSHIFT` command will be executed and the results validated. If the result is correct, then a zero will be returned to the application.

Commands: `LSHIFT`

4.3.64 `mmi_get_version`

Prototype: `int mmi_get_version (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: DiSPATCH firmware version number on success, or `-1` on failure.

Description: This function retrieves the DiSPATCH firmware revision number from the DSP. The major and minor version numbers are returned from the function as an integer value computed as follows:

$$(\text{Major} * 100) + \text{Minor}$$

For example, on a DSP running DiSPATCH version number 3.20, the `mmi_get_version()` function would return `320`.

Commands: `VERSION`

4.3.65 `mmi_show_configuration`

Prototype: `int mmi_show_configuration (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function displays a long list of DiSPATCH firmware configuration settings. These values are of little value to the application programmer and are subject to change. At present, there is no provision to return any of the configuration values; the library prints them to standard output directly.

Commands: `CONFIGURATION`

4.3.66 `mmi_test_dsp_dram`

Prototype: `int mmi_test_dsp_dram (dsp_t dsp, int verbose);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`verbose`	Request verbose test detail messages.

Returns: Zero on success, or `-1` on RAM failure.

Description: This functions asks the DSP to perform a thorough test of all shared DRAM. If any part of the DRAM does not test correctly, a failure is reported and `-1` is returned.

After the DSP has found DRAM to be operating correctly, a special tag pattern is placed in memory to be verified by the host. This ensures that both the DSP and host processor agree on the RAM addressing.

When the tag pattern has been verified, the `mmi_test_dsp_dram()` function replaces all the Command IDs that were erased by the test procedure and then returns a value of `0`.

If the `verbose` argument is non-zero, then the library will print status messages to `stderr` while the test is in progress. In the event of failure, the routine will print the failed address. The library will not print at anything if the `verbose` argument is zero.

Note that the RAM test is destructive, so any audio data or command arguments in DRAM will be destroyed during the test. This can result in severe audio anomalies if playback is in progress when the test is executed.

Commands: `TEST_DRAM`

4.3.67 `mmi_start_loopback`

Prototype: `int mmi_start_loopback (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function puts the DSP into internal loopback mode. In this mode, all incoming digital audio is immediately copied to the output channel. Normal play and record processing is suspended during loopback.

A similar effect can be achieved by enabling the audio sidetone feature with `mmi_set_sidetone()`. This is the recommended method for testing the analog audio path.

Commands: `LOOPBACK`

4.3.68 `mmi_end_loopback`

Prototype: `int mmi_end_loopback (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function returns the DSP to the normal operating state from loopback mode. Record and playback processing is resumed immediately.

Commands: `END_LOOPBACK`

4.3.69 `mmi_issue_invalid_cmd`

Prototype: `int mmi_issue_invalid_cmd (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function intentionally executes a command with an invalid Command ID. The DSP will not recognize the command, resulting in an "Unrecognized Command Code". This function is useful for testing the error handling and recovery mechanisms in the library and applications.

Commands: Unrecognized: `$BADD`

4.3.70 `mmi_dsp_nop`

Prototype: `int mmi_dsp_nop (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function executes the special `NOP` DiSPATCH command, which does absolutely nothing. It is primarily useful for measuring the timing of the command and message protocol.

Commands: `NOP`

4.3.71 `mmi_query_load`

Prototype:

```
int mmi_query_load (dsp_t dsp, int *play, int *play_max,
                    int *record, int *record_max);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`*play`	The highest play load value since the last check.
`*play_max`	The maximum allowable play load for real-time operation.
`*record`	The highest record load value since the last check.
`*record_max`	The maximum allowable record load for real-time operation.

Returns: Zero on success, or `-1` on failure.

Description: The DSP keeps a close count of how many cycles are consumed during playback and recording operations. This function retrieves the current load measurements from the DSP.

The `play_max` and `record_max` arguments are pointers to integer variables. The `mmi_query_load()` function will place the maximum allowable load values into these two variables.

The `play` and `record` arguments are pointers to integer variables. The `mmi_query_load()` function will place the highest recorded load since the last reading into these variables.

If the `play` or `record` values exceed the `play_max` or `record_max`, respectively, then the DSP failed to process audio in the required time. When the DSP becomes overburdened, then the audio processing will fail to run in real-time and anomalies may result. DiSPATCH recovers automatically when the load returns to less than 100%.

For details on the computation and interpretation of the DiSPATCH load measurements, see the *DiSPATCH Firmware User's Manual* under the `QUERY_LOAD` command.

Commands: `QUERY_LOAD`

4.3.72 `mmi_analog_test`

Prototype:

```
int mmi_analog_test (dsp_t dsp, int speed, int *srate,
                     int *energy, int *impurity);
```

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.
`speed`	The approximate clock speed of the DSP, in megahertz.
`*srate`	Measured sample rate approximation (in Hertz).
`*energy`	Average RMS input signal energy.
`*impurity`	Average signal impurity (harmonic distortion + noise).

Returns: Zero on success, or `-1` on failure.

Description: This function initiates an analog audio quality test and reports the results. Before conducting the test, the signal output of the DSP must be directly connected to the analog input of the same DSP, and the gains must be set near unity.

The calling application must specify the clock speed of the DSP, since the timing calculations take this into account. At the time of this writing, the following speeds are recognized: 20, 27, and 33.

During the test, the DSP will synthesize a full-scale sinewave for several seconds and measure the incoming signal for amplitude and distortion. At the same time, the sample rate of the analog converters is measured. The test results are stored in the variables pointed to by `srate`, `energy`, and `impurity`. Note that the function makes *no* judgement of the quality of the signal, but only passes the values to the calling application. A good test signal will have a strong amplitude (high `energy`), and minimal distortion (low `impurity`).

The measured amplitude (`energy`) is returned as a linear value ranging from `0x0000` (no signal) to `0xFFFF` (full signal). Since the test signal is a sinewave, a full strength signal can approach 70% of full RMS energy, approximately `0xB300`.

The measured `impurity` is represented as a linear value ranging from `0x0000` (perfect signal) to `0xFFFF` (all noise). Since neither the analog converters nor the band-pass test filter is perfect, the signal will always contain some impurity.

A large impurity rating (greater than three percent of full scale) may indicate that the analog signal is being distorted, perhaps by excessive gain settings or incorrect connections. A small energy rating (less than twenty percent of full scale) may indicate that the loopback connection is not in place, or that the analog gains are set to low.

The measured sample rate is only an approximation, and may deviate by several Hertz.

This self-test is only valid for monophonic operation, and will give invalid results if stereo mode is enabled.

Commands: `ANALOG_TEST`

4.3.73 `mmi_hammer_dsp`

Prototype: `int mmi_hammer_dsp (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This simply executes the `HAMMER` command, which causes the DSP to loop for 162,024,000 clock cycles before returning. On a 27 MHz DSP, this function should take approximately six seconds to execute. By measuring this delay, the host can verify the clock speed of the DSP running DiSPATCH.

Commands: `HAMMER`

4.3.74 `mmi_eprom_checksum`

Prototype: `int mmi_eprom_checksum (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: EPROM checksum on success, or `-1` on failure.

Description: For MMI boards equipped with an EPROM, this function will return a checksum of the data. Note that DiSPATCH does not use the EPROM at all, but this function is retained for diagnostic purposes.

The EPROM checksum is a 24-bit positive value, representing the lowest 24 bits of the sum of all accessible bytes. A value of `-1` is returned on failure.

Commands: `EPROM_CHECKSUM`

4.3.75 `mmi_show_state`

Prototype: `int mmi_show_state (dsp_t dsp);`

Arguments:

`dsp`	A DSP handle returned by `mmi_get_dsp()`.

Returns: Zero on success, or `-1` on failure.

Description: This function prints a list of internal DiSPATCH state flags to `stdout`. These are used primarily during development and are subject to change.

Commands: `QUERY_STATE`

5. Example Applications

The DiSPATCH Software Package includes several example applications. These programs may be adapted to fit customer needs, or parts may be extracted and modified to accelerate application development.

Complete source code is included for all DiSPATCH example programs in the "`dispatch/apps`" distribution directory. As with the programming library, Vigra can not provide technical support for customer-modified versions of the example applications. Please study the code carefully before making any changes, and always retain a copy of the original source code.

5.1 MMI-Test

The MMI-Test program is a powerful test and application prototyping tool for DiSPATCH. MMI-Test provides a text-based command-line interface to all DiSPATCH features and functions. The program can be used interactively or directed by text script files.

The first part of this section explains how one might use MMI-Test, and the second part provides a more detailed reference for each command supported by the program.

Under most operating systems, the MMI-Test program uses the GNU Readline command-line interface, providing convenient and programmable editing features that closely resemble that of the Bash shell and GNU Emacs editor.

5.1.1 Running MMI-Test

The MMI-Test program is executed by simply running "`mmi-test`". The program uses no command-line arguments. An MMI audio processing board need not be installed to run the program, but most commands will not run until a board has been installed and selected.

Note that under OS-9, filenames are not allowed to contain the dash character (‘`-`’), so the program is called “`mmi_test`” on this platform, instead of “`mmi-test`”.

5.1.2 The Command Line

The command line processor is provided by GNU’s Readline and History libraries, and includes a number of features such as macros, intelligent terminal support, automatic completion, undo, keymap configuration, cut and paste, and command history. For information on how to fully use this powerful command line interface, please read the GNU Readline Library and GNU History Library documentation by Brian Fox. Full source code to these libraries is provided with MMI-Test.

In brief, command lines may be edited by use of standard keys such as delete, backspace, and cursor motion. A complete history of executed commands is kept and automatically saved between sessions. The keystrokes `Control`-`P` and `Control`-`N` will move backwards and forwards through previously-entered commands.

The screen can be cleared and redrawn at any time by hitting `Control`-`L`. To cancel a command line, press `Control`-`C`.

To exit the MMI-Test program, enter the `quit` command, or press `Control`-`D` from the command line.

Automatic Completion

Command words and filenames may be automatically completed by pressing the `TAB` key. This will finish the start of a command word, or beep if the command is not uniquely specified. A second press of the `TAB` key will show all the possible completions for the partial command or filename. Completion makes it easy to quickly enter commands, even long commands, because the user need only type the first part, and then press `TAB`.

5.1.3 Selecting and Initializing an MMI Board

Before executing any firmware commands, the user must select an MMI board to act upon. This is accomplished by use of the `open_board` command. This command requires a Vigra MMI model name and device handle as parameters. These specifies which board you wish to access, and what MMI model it is. For example, this command would open the first MMI-4210 on the system:

```
MMI Test: open_board MMI-4210 /dev/mmidsp0
```

The model name must be one of the recognized names shown on page 67. Case is ignored.

The device name is determined by the operating system configuration. The first installed MMI board is conventionally named "`/dev/mmidsp0`", and the second is named "`/dev/mmidsp1`", etc.

The `open_board` command will map the VME board into virtual space, if necessary, and then set up the status information in the interface routines. On-board DRAM will be initialized for firmware operation. An error will be returned if the board could not be mapped or accessed. This usually means that the VME address jumpers on the board are not configured correctly, or the model was specified incorrectly.

After mapping and initializing the board, most commands will become accessible to the user. The command-line prompt will change from "MMI Test:" to "MMI-4210 #1 [DSP A]:" or something similar. This prompt indicates the currently selected MMI model and DSP channel. It is updated when a different board or DSP is selected. All MMI-Test commands are directed to the selected board and DSP, as shown in the prompt.

More than one MMI board can be open at the same time. Each board will be assigned a different board number, starting from number one. The board number of the active board is shown in the prompt. The user can switch between open boards by calling `board_select` with the board number.

Selecting the DSP

Some MMI boards have more than one Digital Signal Processor (DSP). Each of the DSPs is totally independent and can be manipulated by itself. The MMI-Test program acts on one DSP at a time, referred to as the "selected DSP".

The user can select a different DSP at any time. Any unselected DSPs continue

DiSPATCH operation, but MMI-Test commands are directed only to the selected DSP. The user selects a DSP by using the `dsp_select` command. For example:

```
MMI-210-1 #1 [DSP A]: dsp_select b

MMI-210-1 #1 [DSP B]:
```

This command selects the second DSP on the board. The next prompt shows the new active DSP.

Booting the firmware

Before any DiSPATCH commands can be executed, the user must initialize the DSP and upload the DiSPATCH firmware. This can be easily accomplished by the use of the `boot` command, which performs all necessary DiSPATCH initialization.

5.1.4 An Example

An example session of MMI-Test is described below. Consult the MMI-Test command reference section for details on each command.

First, the user runs the test program from the Unix prompt:

```
hobbes% mmi-test
```

Then, the user must select a board model and device name:

```
MMI Test: open_board MMI-4211 /dev/mmidsp0
```

This specifies that an MMI-4211 has been configured with the file name "`/dev/mmidsp0`". The program will attempt to map and initialize the board, reporting any errors that are encountered. If no errors are detected, the board will be selected and the prompt will change.

```
MMI-4211 #1 [DSP A]: dsp_select B
```

This selects the second DSP for use, again reflected in the prompt.

```
MMI-4211 #1 [DSP B]: boot
```

This resets and boots the default DiSPATCH firmware binary that is part of the library.

```
MMI-4211 #1 [DSP B]: version
Firmware version 3.31

MMI-4211 #1 [DSP B]: test_dsp_dram
Running firmware DRAM test... Ok.
Running residue check... Ok.
DSP DRAM test passed.

MMI-4211 #1 [DSP B]: test_host_dram
Testing Host 32-bit DRAM access... done.
Testing Host 16-bit DRAM access... done.
Testing Host  8-bit DRAM access... done.
mmi: Host DRAM test passed.
```

These command check the firmware version number, and then do an exhaustive test of DSP and Host access to the on-board Dual-Port RAM.

```
MMI-4211 #1 [DSP B]: srate 8000

MMI-4211 #1 [DSP B]: input_gain 100

MMI-4211 #1 [DSP B]: speed_change 150

MMI-4211 #1 [DSP B]: led on

MMI-4211 #1 [DSP B]: quit
```

These commands set some audio parameters, turn on the front-panel LED, and then exit the MMI-Test program.

5.1.5 X-Windows Sliders

Several commands accept the keyword “slider” in place of the normal numeric argument. If MMI-Test is running under the X-Windows display system, this will cause a ScrollBar window to appear on the screen. This slider will then dynamically control the named parameter. Changes made to the slider position will immediately change the associated parameter, allowing for rapid selection and adjustment of the various audio controls.

For this function to work, the MMI-Test program must be able to call the external program "`slideout`", supplied by Vigra in the distribution directory "`dispatch/apps/slideout`". If the user can not execute this program, or X-Windows is not running, the slider option will have no effect.

5.1.6 Command Reference

This section describes each of the available commands and their parameters. Individual parameter values are represented by bracketed names, <like this>. Parameters that are optional are enclosed in square brackets, <**[**like this**]**>. All other parameters are required.

`?`
: Synonym for `help`.

`abort_all_play`
: Abort all pending play commands on this DSP.

`abort_monitor`
: Abort all pending monitor commands on this DSP.

`abort_play`
: Synonym for `abort_all_play`.

`abort_record`
: Abort all pending record commands on this DSP.

`abort_track` <***track***>
: Abort pending play commands for only one playback track.

`adaptive_fir` <***gain* | *Slider***> <***taps***>
: Set the adaptive FIR filter parameters. Experimental.

`analog_test` <***DSP Clock (MHz)***>
: Perform a self-test of the analog audio subsection. Requires that a loopback cable be installed from the channel output to channel output.

`audio_loop`
: Record and immediately play back buffers continuously. This creates a digital delay based on the current buffer size.

`bias`
: Synonym for `input_bias`.

`blinks` **[<*count*>]**
: Execute LED on/off commands as fast as possible. A count of –1 will blink the LED continuously.

`board_select` <***board number***>
: Select an MMI board by number. The board must have previously been opened with `open_board`.

`boot` **[<*firmware filename*>]**
: Boot the DiSPATCH firmware. If the optional firmware filename is provided, then the P, X, and Y binary images will be obtained from external files instead of the library default images.

`brief_analog_test` <***DSP Clock (MHz)***>
: Performs tha same test as `analog_test`, but prints the results in a more compact format.

`buffer_size` <***words | ?***>
: Set the MMI-Test audio buffer size. This buffer size is implicitly used by all double-buffered commands, like `play_file` and `record_file`.

`configuration`
: Show the DiSPATCH firmware configuration values.

`count_buffers`
: Print the number of pending record, play, and monitor buffers.

`counter` **[<*toggle*>]**
: Enable or disable the playback sample counter.

`debug`
: Special volatile debug function.

`diag_boot`
: Verbose diagnostic version of `boot`. This command lists each of the steps in the boot sequence as they are executed.

`discard`
: Discard all pending responses from the library's internal list.

`display_tone`
: Synonym for `tone_display`.

`do` <***filename***>
: Execute commands from a script file instead of the interactive command line. Control will return to the command line when the script finishes

`dribble` <***filename or -*** >
: This logs all DiSPATCH DSP messages to a file. The special filename of "-" will print them to the terminal running MMI-Test.

`dsp_select` <***new dsp channel***>
: Select a DSP channel as the active channel. MMI-Test will direct all new commands to the active channel. The new dsp channel may be specified by letter (starting from "A"), or by number (starting from 0).

`echo` **[**<***string***>**]**
: This prints its argument. This is useful for printing messages from script files.

`eprom_checksum`
: Compute and display the EPROM checksum for this DSP.

`equalizer` **[**<***toggle***>**]**
: Enable, disable, and adjust the 10-band spectrum EQ. The <toggle> argument enables or disables the equalizer. The first time the slider is enabled, the level-adjust sliders will appear. the levels. This command requires that graphical sliders be available, since 10 numeric arguments would get cumbersome.

`erase_dram`
: Zero the entire data buffer region of DRAM.

`exit`
: Synonym for `quit`.

`fetch_error`
: Retrieve the current error code from the DSP.

`halt`
: Halt the DSP. This puts the DSP into a hardware reset state, which requires that DiSPATCH be rebooted later on this DSP.

`hammer`
: Execute a large fixed loop on the DSP. This can be useful for DSP execution speed measurements.

`help` **[<*command*>]**
: Display help text. If no <command> is specified, every available command and a brief description of each will be displayed. If a specific command is specified, then the description and usage for that command will be displayed.

`history` **[<*how many*>]**
: Show MMI-Test command history. This is used to review previously-used commands. If a count is not specified, all recorded command lines will be displayed.

`input_bias` **[<*buffer count*>]**
: Measure the DC input bias for this channel.

`input_gain` <***gain value | Slider***>
: Set the analog input amplifier gain on models that are equipped with appropriate hardware.

`input_peak`
: Measure and print the peak input signal level.

`input_select`
: Synonym for `select_input`.

`invalid_cmd`
: Issue an intentionally invalid DSP command for testing the error recovery mechanisms.

`led` **[<*toggle*>] [<*which led*>]**
: Set the front-panel LED state. On boards with more than one LED, the <which led> argument specifies which is to be set. The first available LED is number zero, the default.

`listen`
: Wait indefinately for DSP messages. This is most useful when the dribble file is enabled and a response is expected. MMI-Test must be interrupted to end this command.

`list` **[<*filespec*>]**
: List files in the specified directory. This is similar to the common "`ls -l`" command under Unix.

`load_dspmem` <***X|Y|P***> <***DSP addr***> <***file***>
: Upload a binary image from a system file to DSP private P, X, or Y memory.

`load_wave` <***filename***> <***table code***>
: Load a waveform file into a DSP wavetable. The file must be exactly 256 samples long (512 bytes).

`loopback` **[**<***toggle***>**]**
: Enable or disable the DSP direct digital audio loopback.

`ls`
: Synonym for `list`.

`measurements` **[**<***toggle***>**]**
: Enable or disable input signal measurements (peak and bias).

`mail` **[**<***toggle***>**]**
: Enable or disable DiSPATCH debugging mail messages.

`monitor_format` <***format***>
: Set the audio data format for monitor data.

`monitor_pos`
: Print the current monitor buffer position.

`monitor` **[**<***offset***>**]** **[**<***count***>**]**
: Start playback monitoring to copy processed playback audio to DRAM.

`nops` **[**<***how many***>**]**
: This repeatedly executes the `NOP` command as fast as possible. The default cycle count is 10,000. This is useful for timing the communication mechanism and overhead.

`open_board` <***model***> <***driver basename***>
: Open and initialize an MMI board. The model name must be a recognized Vigra MMI model name string, such as "MMI-4211". The driver basename specifies the configured system name for that board, such as "/dev/mmidsp0".

`output_gain` <***gain value | Slider***>
: Set the analog output amplifier gain, on boards equipped with appropriate hardware.

`pause_play`
: Pause all playback operation.

`pause_record`
: Pause all recording.

`pause` **[<*seconds*>]**
: Wait for the specified number of seconds.

`play_file` <***filename***> **[<*track*>]**
: Play an audio file using the specified playback track.

`play_filter` <**[*toggle*]**> <***coefficient file***>
: Playback FIR filter control. The first option enables or disables the FIR filter. If a coefficient file is specified, it will be loaded into the FIR filter table on the DSP.

`play_format` <***format***> **[<*track*>]**
: Set audio data format for playback on the specified track.

`play_gain` <***digital gain | Slider***> **[<*track | Master*>]**
: Set the digital playback gain (in percent) for one track or specify “Master” to set the master playback gain.

`play_pos` **[<*track*>]**
: Print the current play position.

`play`
: Synonym for play_file.

`probe`
: Verify that the DiSPATCH firmware is initialized and running.

`process` <***input file***> <***output file***>
: Play the input file while using monitor to store the processed audio into the output file.

`query_load`
: Read, display, and reset the worst-case bandwidth utilization measurement. A load greater than 100% indicates that the DSP may have failed to meet demands in real-time.

`quiet` <***toggle***>
: Toggle verbose mode. When verbose mode is enabled, all status messages will be supressed.

`quit`
: Exit the MMI-Test program.

`ram_copy` <***source***> <***dest***> **[<*count*>] [<*src start*>] [<*dest start*>]**
Copy data from one open MMI board to another. Both boards must have been previously opened with `open_board` and assigned a board number. The <source> and <dest> arguments are specified as board numbers.

`ram_load` <***filename***> **[<*offset*>] [<*count*>]**
Load data into DRAM from a file.

`ram_monitor`
Synonym for `monitor`.

`ram_play` **[<*offset*>] [<*count*>] [<*track*>]**
Play data already present in DRAM. This command returns immediately after initiating the playback.

`ram_record` **[<*offset*>] [<*count*>]**
Record audio to a buffer in DRAM. This command returns immediately after starting the recording.

`ram_save` <***filename***> **[<*offset*>] [<*count*>]**
Copy the contents of a DRAM region into a file.

`ram_subbuf_play` <***buf count***> <***buf spacing***> **[<*start*>] [<*size*>] [<*track*>]**
Play DRAM region using subbufers.

`ram_transform` <***src fmt***> <***dest fmt***> <***start***> <***size***> **[<*dest*>] [<*subbufs*>] [<*subbuf spacing*>]**
Convert the audio format of a region of data. The audio source data region and destination region should not overlap. This command will not return until the format conversion is complete.

`record_file` <***filename***> **[<*length*>]**
Record incoming audio to a file using double-buffering. The <length> is specified in seconds.

`record_filter` <**[*toggle*]**> <***coefficient file***>
Recording FIR filter control. The first option enables or disables the recording FIR filter. If a coefficient file is specified, it will be loaded into the FIR filter table on the DSP.

`record_format` <***format***>
Set the audio data format for recording.

`record_gain` <***digital gain | Slider***>
: Set the digital recording gain (in percent).

`record_pos`
: Print the current record buffer position.

`record`
: Synonym for `record_file`.

`repeat` <***command***>
: Repeat an MMI-Test command indefinately, or until MMI-Test is interrupted.

`reset_counter`
: Reset the playback sample counter value to zero.

`resume_play`
: Resume paused playback.

`resume_record`
: Resume paused recording.

`reverb` <***gain | Slider***> <***delay***>
: Set the digital reverb parameters or request slider controls.

`route` <***A, B, C, D***>
: Select audio signal output routing (using analog mixer). The audio output from the current DSP will be directed to all of the specified output ports.

`select_input` <***Microphone | Line***>
: Select the audio input source. Abbreviations are accepted.

`set_pnm` <***P***> <***N***> <***M***>
: Set programmable frequency synthesizer P, N, and M values manually.

`show_state`
: Show internal DiSPATCH state flags.

`sidetone` <***gain value | Slider***>
: Set the sidetone gain level, in percent.

`signal_detect` <***detector***> <***voice | Slider***> <***energy***> <***up***> <***down***>
: Configure the parameters for a signal detector.

`sparcplay` <***filename***>
: Play a μ-Law audio file directly to "`/dev/audio`". This only applies to Sun Sparcstations.

`speed_change` <***Slider | 50 - 200***>
: Set speed change percentage.

`srate` <***Slider | play freq***> **[**<***rec freq***>**]**
: Set the sampling rate for playback and recording.

`stereo_mode` <**[*toggle*]**>
: Enable or disable stereo operation.

`swatch` <***detector***>
: Start a “`swatch`” window to watch the status of a signal detector. This requires that the “`swatch`” program be installed and executable.

`test_dsp_dram`
: Test DSP access to DRAM.

`test_host_dram`
: Test host access to DRAM.

`test_tone` **[**<***freq***>**]** **[**<***length***>**]** **[**<***amplitude***>**]**
: Generate a test tone.

`time` <***command***>
: Time the execution of an MMI-Test command.

`tone_am_enable` **[**<***toggle***>**]**
: Enable or disable amplitude modulation.

`tone_am_frequency` <***Hertz***>
: Set AM frequency.

`tone_am_table` <***table***> **[**<***table start***>**]**
: Set wavetable for AM oscillator.

`tone_attack_time` <***seconds***>
: Set envelope attack time.

`tone_count` <***count | -1***>
: Set the repeat count of the tone. Use –1 for endless repetition.

`tone_display`
: Display the current tone parameters.

`tone_envelope_enable` **[**<***toggle***>**]**
: Enable or disable envelope.

`tone_fm_delta` <***delta freq* | *SI***>
: Set FM frequency delta (depth).

`tone_fm_enable` **[**<***toggle***>**]**
: Enable or disable frequency modulation.

`tone_fm_frequency` <***mod freq* | *SI***>
: Set frequency modulation rate.

`tone_fm_table` <***FM Table***> **[**<***table start***>**]**
: Set wavetable for FM oscillator.

`tone_frequency` <***frequency***>
: Set the main (carrier) tone frequency.

`tone_release_time` <***seconds***>
: Set envelope release time.

`tone_sustain_time` <***seconds***>
: Set the sustain length of a tone.

`tone_times` <***attack***> <***sustain***> <***release***>
: Set all three envelope times.

`tone_track` <***track***>
: Set the output playback track for tone generation.

`tone_wavetable` <***table code***>
: Set the wavetable for the main oscillator.

`transform_file` <***src fmt***> <***dest fmt***> <***src filename***> <***dest filename***>
: Convert the data format of an audio file.

`upload_p` <***binary P image filename***>
: Boot a binary program image from a file, but do not initialize X and Y memory.

`version`
: Display DiSPATCH firmware version number.

`wait` **[**<***prompt***>**]**
: Wait for the user to hit Enter.

`watch_counter`
: Show the running play counter until interrupted.

`waveshape` **[**<***table***>**]**
: Select the waveshaping table, or –1 to disable waveshaping.

5.2 ToneShop and SampleTones

ToneShop is an interactive tone editor for testing and prototyping tone synthesis parameters. This program is documented in detail in the *DiSPATCH Tone Generation Manual*. The SampleTones program also provides several examples of common models for tone synthesis.

5.3 `simp_play.c` and `simp_record.c`

These two files provide simple examples of playback and recording using the DiSPATCH library. Many features have been left out of these programs for maximum simplicity.

The command-line usage for `simp_play` is as follows:

```
simp_play 32000 myaudio.in
```

The first argument is the sample rate to use for playback. The second argument is the name of the file to supply the audio data for playback. If the filename argument is omitted or it is “-”, then the audio data will be read from “`stdin`”. If no arguments are provided, the sample rate will be 32000 Hz.

The command-line usage for `simp_record` is as follows:

```
simp_record 32000 30 myaudio.out
```

The first argument is the sample rate to use for recording. The second argument is the length of the recording in seconds. The last argument is the name of the output file for the recorded data, or “-” to write it to “`stdout`”. The default arguments are “32000”, “60”, and “-”, respectively.

The code for both the `simp_play` and `simp_record` programs were written to provide clear examples to the reader. The MMI model and device name are hard-coded into the program for simplicity, but can be easily edited for a different configuration. See the comments in the code for customization details.

5.4 Play

While `simp_play` is a simple example of playback, the `play` program provides added flexibility and access to more of the advanced features of DiSPATCH.

5.4.1 Unix Usage

The usage of the `play` program is different under VxWorks and Unix systems. On Unix-derived systems, the options are passed via the command line when executing the play program. The input file for play is also specified on the command line. An example invocation is shown below.

```
% play -model MMI-4211 -dev /dev/mmidsp0 -chan 0 -srate 44100 myaudio.pcm16
```

For details on the available options for the `play` program, execute "`play -help`" to display the option list. Version 1.12 of the `play` program displays the following information:

```
% play -help
Usage: play [options] input_file
  -help               Display this usage information.
  -model <name>       Specify the MMI model (i.e. MMI-4211).
  -device <name>      Specify the device driver (i.e. /dev/mmidsp0).
  -channel <chan>     Select the DSP channel (0, 1, 2, or 3).
  -srate <Hz>         The sample rate for playback (in Hertz).
  -frequency <Hz>     Same as -srate.
  -stereo             Enable stereo mode (default is mono).
  -gain <level>       Set the analog output gain level (0 - 255).
  -format <type>      Specify the audio data format type (i.e. pcm16).
  -quiet              Do not print any status information.
```

All command-line options can be unambiguously abbreviated. Any unspecified options will use the following default values:

Option	Default
model	"MMI-4211"
device	"/dev/mmidsp0"
channel	0
srate	44100
stereo	no
gain	143
format	"PCM16"
verbose	yes
filename	"-"

5.4.2 VxWorks Usage

The VxWorks command shell does not allow more than 10 command arguments, so a different option interface is provided. Since the variables used by `play` are directly accessible from the command shell, all the options can be changed by simply assigning new values to these global variables. After the control variables have been set as needed, the function `play()` can be called from the shell (or another function). The function `play_usage()` will display the available option variables and their current values. An example display from `play_usage()` is shown below.

```
-> play_usage

The MMI DiSPATCH play() routine uses these global variables:
  char *play_model    -  The MMI model name ("MMI-4211").
  char *play_device   -  The device driver name ("/dev/mmidsp0").
  char *play_format   -  Specify the audio data format type ("pcm16").
  char *play_file     -  The input filename ("-").
  int   play_channel  -  The DSP channel to use (0).
  int   play_srate    -  The sample rate (Hz) for playback (44100).
  int   play_stereo   -  1 = Stereo mode enabled, 0 = mono (0).
  int   play_gain     -  The analog output gain level (143).
  int   play_verbose  -  1 = Print status information, 0 = quiet (1).
```

To change the gain setting, for example, simply set the new value from the command line before calling `play()`:

```
-> play_gain = 50
_play_gain = 0x3b2654: value = 50 = 0x32 = '2'
->
```

5.5 Record

The `record` program is similar in operation to the `play` program. Instead of playing audio from a system file, it records incoming audio to a new file. Any existing file of the same name as the output file will be replaced. The Unix options for `record` are as follows:

```
% record -help
```

```
Usage: record [options] output_file
  -help             Display this usage information.
  -model <name>     Specify the MMI model (i.e. MMI-4211).
  -device <name>    Specify the device driver (i.e. /dev/mmidsp0).
  -channel <chan>   Select the DSP channel (0, 1, 2, or 3).
  -srate <Hz>       The sample rate for recording (in Hertz).
  -frequency <Hz>   Same as -srate.
  -stereo           Enable stereo mode (default is mono).
  -gain <level>     Set the analog output gain level (0 - 255).
  -format <type>    Specify the audio data format type (i.e. pcm16).
  -quiet            Do not print any status information.
  -source <name>    Input source ("line" or "mic").
  -length <secs>    Recording length in seconds.
```

Under VxWorks, the `record` options are controlled via global control variables, as with `play`. The control variables are:

```
-> record_usage

The MMI DiSPATCH record() routine uses these global variables:
  char *record_model    -  The MMI model name ("MMI-4211").
  char *record_device   -  The device driver name ("/dev/mmidsp0").
  char *record_format   -  Specify the audio data format type ("pcm16").
  char *record_file     -  The output filename ("-").
  char *record_source   -  Audio input source ("line").
  int   record_channel  -  The DSP channel to use (0).
  int   record_srate    -  The sample rate (Hz) for recording (44100).
  int   record_stereo   -  1 = Stereo mode enabled, 0 = mono (0).
  int   record_gain     -  The analog output gain level (255).
  int   record_verbose  -  1 = Print status information, 0 = quiet (1).
  int   record_length   -  Recording length in seconds (300).
```

INDEX

`dsp_t` type, 65, 71, 73
`mmi_board_t` type, 65
`mmi_complete`, 66
`play` example program, 190
`record` example program, 192
`simp_play`, 190
`simp_record`, 190
`select()` operation, 86
“`dispatch.h`”, 65
“`firm_defs.h`”, 65

Aborting
  monitoring, 121
  playback, 118, 119
  recording, 120
Address of DRAM start, 72
Address, physical, 64
Allocating memory, 81
Analog input gain, 123
Analog input select, 141
Analog output gain, 124
Analog signal routing, 125
Analog test, 169
ANSI C provisions, 65
Audio data formats, 93
Audio data packing, 94
Audio diagnostics, 169
Audio loopback, 164

Base address of DRAM, 72
Binary firmware images, 4
Board address, 64
Board deallocation, 69
Board model name, 73
Board names, 67
Board structures, 65
Booting DiSPATCH, 74
Buffer playback, 102
Buffer position, 115

C header files, 65
C programming library, 63
Callbacks, 87, 88
Checking for responses, 80
Checksum, EPROM, 172
Clipping LED, 132
Closing boards, 69
Command codes, 89
Command line, 176
Commands, executing, 95
Completion tokens, 66, 83–85, 102
Configuration, firmware, 162
Control functions, 67
Counting buffers, 131
Customer modifications, 65

Data format
  monitoring, 110
  playback, 108
  recording, 109
Data formats, 93
Deallocating boards, 69
Destructive DRAM test, 82
Device driver, 63
Device filenames, 64
Diagnostic DiSPATCH boot, 76
Digital loopback, 164
Discarding messages, 92

DiSPATCH Programming Library, 63
DiSPATCH Software Package, 1
Distortion test, 169
DRAM base address, 72
DRAM test
  DSP, 163
  host, 82
DRAM tone generation, 152
Dribble file, 90, 91
DSP bandwidth and load, 168
DSP clock speed, 171
DSP file descriptor, 86
DSP filenames, 64
DSP Handle, 71
DSP memory, loading, 96, 97
DSP reset, 79

Enabling mail messages, 158
EPROM checksum, 172
Equalizer, 139
Error codes, 157
Example applications, 175
Executing commands, 95

Faders, 179
Feedback, 128
Fetching error codes, 157
File descriptor, 86
File playback, 106
File recording, 107
Filenames for DSPs, 64
FIR filter, 144–147
Firmware configuration, 162
Firmware images, 4
Firmware initialization, 74
Firmware probing, 160
Firmware state, 173
Firmware version, 161
Flushing responses, 92
Format
  monitoring, 110
  playback, 108
  recording, 109
Format codes, 93
Format conversion, 129, 130
Format names, 93
Format sizes, 94
Frequency equalizer, 139
Front-panel LED, 132
Front-panel LEDs, 122

Gain level
  playback, 111
  recording, 112
Generating tones to DRAM, 152
GNU Readline, 176
Graphical sliders, 179

Hardware reset, 79
Header files, 65
Host-support packages, 5
HP-RT installation, 37
HP-UX installation, 42, 49

Impurity measurement, 169
Include files, 65
Initialization, 67
  DiSPATCH firmware, 74
  VxWorks, 70
Initializing command codes, 89
Initializing library state, 70
Input gain, 123
Input selection, 141
Interactive test, 175
Interrupts, 3
Interrupts vs. polled, 66, 85
Invalid command, 166
IRIX 5.2 installation, 28
IRIX installation, 23
Issue invalid command, 166

LED, 132
LED indicators, 122
Library, 63
Library state, resetting, 70
Load check, 168
Loading DSP memory, 96, 97
Loopback mode, 164, 165

Mail messages, 158, 159
Memory allocation, 81
Memory test, 82, 163
Message list, 66, 85, 92
Messages, printing, 90
Mixer, 125
MMI board address, 64
MMI model names, 67
MMI structures, 65
MMI-Test, 175
MMI-Test, example, 178
mmi_abort_all_play, 119
mmi_abort_monitor, 121
mmi_abort_record, 120
mmi_abort_track, 118
mmi_analog_test, 169
mmi_boot_default, 75
mmi_boot_file, 78
mmi_check_response, 85
mmi_clip_led, 132
mmi_close, 69
mmi_complete, 84
mmi_count_buffers, 131
mmi_delete_callback, 88
mmi_diag_boot, 76
mmi_disable_mail, 159
mmi_disable_measurements, 134
mmi_discard_resp, 92
mmi_dsp_command, 95
mmi_dsp_nop, 167
mmi_enable_mail, 158
mmi_enable_measurements, 133
mmi_end_counter, 99
mmi_end_dribble, 91
mmi_end_loopback, 165
mmi_end_tone, 154
mmi_eprom_checksum, 172
mmi_fetch_error, 157
mmi_filter_play, 144
mmi_filter_record, 145
mmi_get_dsp, 71
mmi_get_dsp_filedes, 86
mmi_get_model, 73
mmi_get_ram_base, 72
mmi_get_version, 161
mmi_halt_dsp, 79
mmi_hammer_dsp, 171
mmi_init_cmd_codes, 89
mmi_input_bias, 136
mmi_input_peak, 135
mmi_issue_invalid_cmd, 166
mmi_led_ctrl, 122
mmi_lib_initialize, 70
mmi_load_dspmem, 96
mmi_load_dspmem_file, 97
mmi_load_play_fir, 146
mmi_load_record_fir, 147
mmi_load_table, 155
mmi_malloc, 81
mmi_monitor_buf, 105
mmi_monitor_position, 117
mmi_open, 67
mmi_parse_format, 93
mmi_play_buf, 102
mmi_play_ctrl, 113
mmi_play_file, 106
mmi_play_position, 115
mmi_play_subbuffers, 103
mmi_play_tone, 151
mmi_poll_messages, 80
mmi_probe_firmware, 160
mmi_query_load, 168

mmi_query_stereo, 143
mmi_ramtone, 152
mmi_read_counter, 101
mmi_record_buf, 104
mmi_record_ctrl, 114
mmi_record_file, 107
mmi_record_position, 116
mmi_register_callback, 87
mmi_register_counter, 98
mmi_request_srate, 149
mmi_reset_counter, 100
mmi_samples_per_word, 94
mmi_select_input, 141
mmi_set_equalizer, 139
mmi_set_input_gain, 123
mmi_set_mixer, 125
mmi_set_monitor_format, 110
mmi_set_output_gain, 124
mmi_set_play_format, 108
mmi_set_play_gain, 111
mmi_set_pnm, 150
mmi_set_record_format, 109
mmi_set_record_gain, 112
mmi_set_resample, 127
mmi_set_reverb, 140
mmi_set_sidetone, 128
mmi_set_speed_change, 126
mmi_set_srate, 148
mmi_show_configuration, 162
mmi_show_state, 173
mmi_signal_detect, 137
mmi_start_dribble, 90
mmi_start_firmware, 74
mmi_start_loopback, 164
mmi_stereo_mode, 142
mmi_test_dsp_dram, 163
mmi_test_host_dram, 82
mmi_transform_buffer, 129
mmi_transform_file, 130
mmi_wait_response, 83
mmi_waveshape, 156
Model name string, 73
Model names, 67
Modified programs, 65
Monitor Position, 117
Monitoring
  aborting, 121
Monitoring playback, 105
Monophonic mode, 142
Motorola System V/88 installation, 55
Multi-thread access, 64
Multirate filter, 127

Names of MMI models, 67
NOP, 167

Object files (VxWorks), 20
Opening an MMI board, 67
OS-9 installation, 33
Out-of-band data, 158
Output gain, 124
Output ports, 125

Parsing format names, 93
Pausing playback, 113
Pausing recording, 114
PFS control, 150
Physical board address, 64
Playback
  aborting, 118, 119
  filter, 144, 146
  gain, 111, 124
  load, 168
  monitoring, 105
  pausing, 113
  position, 115
Playback counter
  reading, 101
  registering, 98
  resetting, 100
  terminating, 99

Playing a file, 106
Playing DRAM buffers, 102
Playing sub-buffers, 103
PNM values, 150
Polled operation, 66, 80, 85
Private DSP memory, 96, 97
Probing firmware, 160
Programming library, 63

RAM base address, 72
Re-entrant library code, 63
Reading a counter, 101
Readline, 176
README file, 1
Record Level, 132
Recording
  aborting, 120
  buffers, 104
  filter, 145, 147
  gain, 112, 123
  input source, 141
  load, 168
  pausing, 114
  position, 116
  to a file, 107
Registering a counter, 98
Registering callbacks, 87, 88
Resampling, 127
Resetting a counter, 100
Resetting DiSPATCH, 79
Resetting library state, 70
Responses, polling, 66
Restoring command codes, 89
Reverb, 140
Router, 125

Sample rate, 127
Sample rate, setting, 148
Sample-interleaving, 142
Samples per word, 94
SampleTones, 190
Sampling frequency, 148, 149
Self-test, 169
Setting the sample rate, 148
SGI installation, 23, 28
Shared access, 64
Show state, 173
Sidetone, 128
Signal detection, 137
Signal impurity, 169
Signal Mixer, 125
Signal quality measurement, 169
Silence detection, 137
Simultaneous use, 64
Slider support, 179
Software reset, 79
Solaris installation, 13
Source code, 65, 175
Source select, 141
Spectrum equalizer, 139
Speed change, 126
Starting DiSPATCH, 74
Stereo mode, 142, 143
Sub-buffer playback, 103
SunOS installation, 6
Synthesis of tones, 151
System load, 168
System V/88 installation, 55

Technical support, 65
Terminating a counter, 99
Testing DRAM
  DSP, 163
  host, 82
Time compress/expand, 126
Tokens
  completion, 66, 83–85
  waiting for, 66, 83–85
Tone Generation, 154
Tone generation, 151, 152

ToneShop, 190
Tracing messages, 90
Transform buffer, 129
Transform file, 130
Types, DiSPATCH, 65

Unclaimed responses, 92
Unix System V/88 installation, 55

Version number, 161
Voice detection, 137
VxWorks initialization, 70
VxWorks installation, 18
VxWorks object files, 20

Waveform synthesis, 151
Waveshaping, 156
Wavetable, loading, 155
Wind Rivers VxWorks installation, 18

X-Windows sliders, 179