MG3500 SDK USER'S GUIDE

November 18, 2008

Document Release 1.0

Document Number: PN985

Mobilygen Corporation
2900 Lakeside Drive #100
Santa Clara, CA 95054

www.mobilygen.com

PREFACE

Welcome to the *MG3500 SDK User's Guide.*

This guide provides the instructions necessary to install the MG3500 SDK, to create boot images, and to run boot images from the board. Also included is how to connect the board to the controllers, devices, and interfaces along with instructions for setting up Audio and Video chips.

The following topics are covered in the Preface:

- "Document Overview"
- "MG3500 Documentation Set"
- "MG3500 Documentation Set Location"
- "Contact"

Document Overview

This guide contains following chapters:

Chapter 1, "Installing the MG3500 SDK"

Chapter 1 provides the necessary instructions to install the MG3500 SDK as well as system requirements, SDK contents, and SDK supported features.

Chapter 2, "Creating Boot Images"

Chapter 2 explains how to set up your development environment by first creating xmodem and NAND boot images. It also covers the instructions to build the distribution from the sources.

Chapter 3, "Running Images on the Board"

Chapter 3 describes how to run the xmodem and NAND images that were created above.

Chapter 4, "Connecting to the I²C Controller"

Chapter 4 describes the I²C protocol tools, supported drives, and instructions on how to connect to the I²C protocol.

Chapter 5, "Connecting to the SPI Controller"

Chapter 5 describes the SPI hardware block, tools, and instructions on how to connect to this serial peripheral interface.

Chapter 6, "Connecting to the GPIO Controller"

Chapter 6 describes GPIO hardware blocks, its controlling pins, and instructions on how to connect to this General Purpose Input/Output interface.

Chapter 7, "Connecting to Interfaces"

Chapter 7 describes PWM hardware blocks, USB host interface, and Watchdog timer interfaces.

Chapter 8, "Connecting to the Input/Output Multiplexer"

Chapter 8 describes the Input/Output Control (IOCRTL) host multiplexer along with its hardware blocks and specified macros.

Chapter 9, "Setting up Audio and Video Chips"

Chapter 9 describes how to set up High Definition Multimedia Interface (HDMI), Analog HDMI Dual Display Interface, as well as Source Selection Analog Anti-Aliasing video decoder.

MG3500 Documentation Set

The MG3500 SDK User's Guide is a companion document to the following manuals in the MG3500 documentation set:

- *MG3500 SDK User's Guide*—This document.
- *MG3500 SDK4 Release Notes*—Contains descriptions of the new Codec enhancements, Codec bug fixes, Mobiapp enhancements, Mobiapp bug fixes, Linux distribution enhancements, and Linux distribution bug fixes.
- *MG3500 Programming Getting Started Guide*—Provides an overview of the MG3500 Mobiapp application, an overview of the Lua scripting language that is embedded within MG3500 Mobiapp, how to interface with Mobiapp, and some sample scripts.
- *MG3500 Mobiapp Reference Manual*—Contains detailed information about the Mobiapp plugin application, including Mobiapp's components, C objects, Lua Interpreter, object types, and Codec API libraries.
- *MG3500 Codec Firmware API Reference Manual*—Describes the Application Programming Interface (API) for the Media Processor firmware and how the Media Processor responds to its API calls. It is the functional specification for the firmware and is a programming manual for the System Host CPU-based software.
- *MG3500 Sample Scripts*—Describes the supported sample Lua scripts that are delivered along with the SDK release. These scripts describe how to implement various use-cases that are supported by the H.264 codec. They are configurable with parameters, which can be set up either within the scripts themselves or through the " –e option" of Mobiapp.

MG3500 Documentation Set Location

MG3500 documentation set listed above are provide on the **MDS** site that is available to the user once the product is registered.

Revision History

This document was prepared by Mobilygen Corporation.

Revision	Documentation Release Date	Notes
Internal 0.1	October 5, 2008	First internal release.
Internal 0.2	October 8, 2008	Second internal release.
Internal 0.3	October 9, 2008	
Internal 0.4	October 10, 2008	
External 1.0	November 18, 2008	First external release.

Contact

You may contact us at **support@mobilygen.com.**

CONTENTS

1

Installing the MG3500 SDK

The MG3500 SDK provides the tools necessary for developing H.264 based products that use the Mobilygen Codec. The SDK provides APIs for interfaces on the devices as well as for using hardware Audio and Video codec features of the MG3500.

The following topics are covered in this chapter:

- "SDK Supported Features"
- "SDK Content"
- "System Requirements"
- "Installing the SDK"

SDK Supported Features

This release contains the following features:

Description	Feature
Toolchain	C, C++, C library
Boot from	NAND, xmodem, NOR, DDR
Storages	NAND, NOR, NFS, CIFS, USB mass storage
Devices upgrade from	Serial, Ethernet
Supported controllers	GPIO, SPI master, I2C master, PWM master, USB master
Supported I/O multiplexer	IOCTRL
Services	FTP, Telnet, HTTP, DHCP client
MGApp features	QHAL for Codec and SOC H.264 Single stream, Half-duplex encode/decode SD and HD File formats: QBX Media: file, Streaming over RTP, UDP Web Demo

SDK Content

The following comprises the contents of the MG3500 SDK release 4:

Content Description	File Name
Toolchain for MG3500	`toolchain.arm-merlin-linux-uclibc-SDK<release-number>r<build-number>.i686-linux.tar.bz2`
Binary version of MG3500 Linux distribution	`dist.merlin-SDK<release-number>r<build-number>.tar.bz2`
Source version of MG3500 Linux distribution	`dist.merlin-SDK<release-number>.src.tar.bz2`
Mobilygen build utilities	`utils.i686-linux-SDK<release-number>r<build-number>.tar.bz2`
MG3500 Codec release	`merlinsw_codec.qmm-merlin-SDK<release-number>r<build-number>.tar.bz2`
MGApp host reference SDK built for arm-merlin	`merlinsw_host.arm-merlin-SDK<release-number>r<build-number>.tar.bz2`
MGApp host reference SDK source	`merlinsw_host.src-SDK<release-number>r<build-number>.tar.bz2`
Configuration file customized for xmodem boot on the EVP	`config.evp-xmodem`
Configuration file customized for NAND boot on the EVP	`config.evp-nand`
Configuration file customized for DDR boot on the Merlin Bring up board	`config.merlinbup`
REG files to boot the chip clock/ddr	`mg3500-2ddr2_248MHz_128MB-soc.reg` `mg3500-2ddr2_264MHz_128MB-soc.reg`

System Requirements

The MG3500 Evaluation Platform Software Development Kit (SDK) software can be built on the following operating system and library:

Operating System: Linux i686 platform

GNU C Library: GLIBC 2.3.4 or above.

The software generated by the distribution and toolchain supports the following EVP:

Evaluation Platform:MDS EVP board

This release was tested through using a host running Linux distribution **Ubuntu 8.04** and an EVP board.

MG3500 Platform: merlin.

Packages for Distribution

The following packages must be installed in order to build the distribution.

- `ibc6-dev`
- `gcc`
- `g++`
- `make`
- `libncurses-dev`
- `zlib1g-dev`
- `libstdc++-dev`
- `patch`
- `fakeroot`
- `bison`
- `m4`
- `flex`
- `autoreconf`
- `automake`

Setting up Ubuntu for Bash Shell

Ubuntu defaults to the dash shell, which offers fewer features but is POSIX compliant. However since the installation scripts rely on the bash shell, the following link must be set:

```
sudo ln -sf /bin/bash /bin/sh
```

INSTALLING THE SDK

All the MG3500 SDK installation files are contained in a file called "**.run**" file. This file is a collection of tar files that comprise the toolchain, distribution sources, host code sources, etc. The installation files are located in the `/opt/mobilygen` directory.

Installing MG350 SDK is a two-step process:

1. Execute the **.run** file to get the files required for the next stage, including all the pre-built binaries for MG3500 SDK.
2. Install the components just un-tarred from the **.run** file, including the sources to build new binaries.

Note: Installing the toolchain for MG3500 requires root privileges.

This installation assumes a normal user, for example "mg3500" with a home directory of "`/home/mg3500"` for installing the MG3500 SDK software.

1. Change directory to your user directory.

```
$ cd /home/mg3500
```

2. Create a new directory where you want to install the SDK software.

```
$ mkdir <release_dir>
```

where <*release_dir*> is SDK<*n*> and "n" is the current SDK release number, for example "SDK4."

3. Change to this directory.

```
$ cd <release_dir>
```

4. Transfer the **release.merlin-SDK<M>.RC<m>.run** file to your local machine, where "<M>" stands for the major release number and "<m>" stands for the release candidate or minor number.

5 Login as "root" and then run the following commands:

```
$ chmod +x <path_to_release_file>/release.merlin-
  SDK<M>.RC<m>.run
$ <path_to_release_file>/release.merlin-SDK<M>.RC<m>.run
```

This will create a new directory <release_dir>/SDK<M>.RC<m>, which will be referred to as the <sdk_release_dir> in the instructions that follow.

6 Log out and login back again for your environment to be updated since the `/root/.profile` has been changed.

7 To look at the source code or to recompile it, run the following command:

Caution! You should not be root.

```
$ <sdk_release_dir>/install.sh --src
```

This will install the source code in the "`~/project`" directory.

8 To install the source code in a different directory, use the full path for that directory.

```
$ <sdk_release_dir>/install.sh --src --workdir <path to
install in>
```

9 To look at all the available options, type

```
$ <sdk_release_dir>/install.sh --help
```

Above, the user has installed the toolchain and the sources. The next chapter will discuss how to create boot images for the board.

2

Creating Boot Images

The next step after installing the MG3500 SDK software is to create flash images for booting the board. Building or modifying the distribution sources is also possible.

The following topics are covered in this chapter:

- "Creating xmodem Boot Images"
- "Creating NAND Boot Images"
- "Building the Distribution"

CREATING XMODEM BOOT IMAGES

If the board was never flashed before or the NAND was corrupted, xmodem images must be used to boot the system. To create the binaries that boot the system via the xmodem data transfer, follow the procedure covered in this section.

See-> Chapter 3, "Running Images on the Board," for instructions on how to run the xmodem images built by following these instructions.

BEFORE BUILDING THE IMAGES

On some Linux distributions (that is, **Ubuntu**), the pre-compiled version of "fakeroot" that is provided by Mobilygen will fail. If this happens, you may use the version of fakeroot that is installed locally on your machine.

The MG350 configurations are based on the EVP board, for example **Ubuntu.**

1. Login as a "normal user."
2. Create a directory where the binaries for the new images you create will reside, for example "`mg3500`." The new images are derived from existing images that were installed in Chapter 1," Installing the SDK."

   See-> The MG3500 SDK4 Release Notes for the specific `*.reg` files to use.
3. Change the directory to your home directory.
4. Create a new directory, referred to as "binaries directory" <bin_dir>.

```
$ mkdir -p binaries/SDK<M>.RC<m>
```

5. Change to this new directory and set the QUARC_ROOT env variable and copy the memory configuration file.

```
$ cd binaries/SDK<M>.RC<m>
$ export QUARC_ROOT='pwd'
$ cp <sdk_release_dir>/<memory_configuration_file>.reg ./
```

<sdk_release_dir> was the new directory created in Chapter 1," Installing the MG3500 SDK."

6 Run the following command to start a configuration utility.

```
$ distinstall.sh --config <sdk_release_dir>/config.evp-
xmodem <sdk_release_dir>/<distribution tarball>
```

Note: If you encounter any problem, see Section "The following files along with their default names (unless they were changed in the config utility) will be created:" below.

7 Select different options if you wish or you may simply accept the default configuration settings specified by the **config.evp-xmodem** file.

8 If the configuration file has been modified and saving the settings is desired, enter "yes" here. This will be saved in the **<bin_dir>/.config** file.

9 If the user decides to run **distinstall** again, removing the "--config" flag will be necessary. A copy will also be written back to the <sdk_reldir> directory.

Note: The **.config** file is the configuration file for the last configuration made and will be overwritten if you choose a different configuration such as "--config" the next time you run **distinstall.sh**.

10 The following files along with their default names (unless they were changed in the config utility) will be created:

File Name	Function	Description
xmodem-bootloader.bin	The bootloader.	This is the first file to upload using xmodem.
xmodem-kernel.img	The Kernel image.	This is the second file to upload using xmodem.
xmodem-initrd.img	The initrd image.	This is the third file to upload using xmodem.

See-> Section "Booting from xmodem Images," in Chapter 3, "Running Images on the Board," on how to use these xmodem images.

TROUBLESHOOTING

1 Check if fakeroot has been installed.

```
$ which fakeroot
/usr/bin/fakeroot
```

2 Add the following to the **distinstall.sh** command:

```
$ distinstall.sh --config <sdk_release_dir>/config.evp-
xmodem  <sdk_release_dir>/dist.merlin-SDK4r2430.tar.bz2 -
- --fakeroot /usr/bin/fakeroot
```

Caution! This must be the last argument passed. Everything after "--" will be passed to other utilities. Also, if the user is building from the source code, this will not be an issue since fakeroot will be built according to your current system configuration.

CREATING NAND BOOT IMAGES

To update a system that has already been flashed, create binaries to boot the system from a NAND flash device. The instructions for creating NAND images are covered in this section.

1 Login as a "normal" user.

2 Create a directory where the binaries for the new images created will reside. The new images are derived from existing images that were installed in Chapter 1," Installing the SDK."

See-> The MG3500 SDK4 Release Notes for the specific ***.reg** files to use.

3 Change directory to your home directory.

4 Create a new directory, referred to as "binaries directory."

```
$ mkdir -p binaries/SDK<M>.RC<m>
```

5 Go to your binaries directory (see above).

```
$ cd <bin_dir>
$ cp <sdk_release_dir>/<memory_configuration_file>.reg ./
```

<sdk_release_dir> is the directory where the release files are stored.

6 Run the following command to start a configuration utility.

```
$ distinstall.sh --config <sdk_release_dir>/config.evp-nand
<distribution tarball>  --install <additional tarball>
```

7 A configuration utility will start now.

8 Select different options if you wish or simply accept the default configuration settings specified by the **config.evp-nand** file.

Note: If you encounter any problem, see Section "The following files along with their default names (unless they were changed in the config utility) will be created:" below.

9 If modifying the configuration file and saving the settings are desired, enter "yes" here. This will be saved in the **<bin_dir>/.config** file.

10 If the user decides to run **distinstall** again, remove the "--config" flag. A copy will also be written back to the <sdk_reldir> directory.

Note: The **.config** file is the configuration file for the last configuration made and will be overwritten if you choose a different configuration such as "--config" the next time you run **distinstall.sh**.

11 The following files will be created. These default names can be changed by using the **config** utility.

File Name	Function	Description
nand-bootloader.bin	The bootloader.	This is the first file to upload using NAND: Flash this one first.
nand-initrd.img	The initrd image.	This is the second file to upload using NAND.
nand-kernel.img	The Kernel image.	This is the third file to upload using NAND.
nand-rootfs.tgz	An archive of the root file system.	This will be asked for by the update utility.

See-> Section "Booting from NAND Images," and also in Chapter 3, "Running Images on the Board," on how to use the NAND images.

TROUBLESHOOTING

1 On some Linux distributions (that is, **Ubuntu)**, the pre-compiled version of "fakeroot" that is provided by Mobilygen will fail. If this happens, you may use the version of fakeroot that is installed locally on your machine.

2 Check to see if fakeroot has been installed.

```
$ which fakeroot
/usr/bin/fakeroot
```

BUILDING THE DISTRIBUTION

This section provides the instructions to build or modify the distribution from the sources.

1 T build or modify the distribution sources, you must first install it using the **install.sh --src** command, as described in Chapter 1," Installing the MG3500 SDK."

2 Login as your normal user.

3 Go into your work directory (default is ~/project).

4 Make sure QUARC_ROOT is defined.

5 Run the following commands:

```
$ cd dist
$ make all DIST=merlin
```

This will build a file named **dist.merlin-rexported.tar.bz2** that you can use with the **distinstall.sh** command, as described in Chapter 1," Installing the MG3500 SDK."

6 A menu configuration tool will open. Select "Exit" and then "Save."

7 To build boot images for pre-defined configurations, you can do the following if you have created the **dist.merlin-rexported.tar.bz2** file. A menu configuration tool will not open in these example because a pre- define configuration will be used.

for NAND

```
$ make install DIST=merlin PKGCFG=evp-nand
```

for xmodem

```
$ make install DIST=merlin PKGCFG=evp-xmodem
```

PCKCFG indicates which pre-defined configuration to use. These configuration files can be found in `dist/configs/distributions/merlin/packages/dist`.

These commands will create bootable images; however, they do not include the codec reference application. As mentioned above, to do this you must use the **distinstall.sh** command along with the **dist.merlin-rexported.tar.bz2** file

3

Running Images on the Board

The next step after installing the SDK and creating boot images is to run these images on the board.

The following topics are covered in this chapter:

- "Booting from xmodem Images"
- "Booting from NAND Images"
- "Setting Switches for Non-EVP Boards"

BOOTING FROM XMODEM IMAGES

Booting from the board via xmodem is necessary if a board has never been flashed or if the flash becomes corrupted.

To run the xmodem binaries on the board, follow the procedure as described in this section.

The instructions that follow describe how to update the flash images by booting from the images that were downloaded to the board through using xmodem.

Note: This is referencing "xmodem -1k," where the 1k option is not available by default for a normal xmodem transf operation.

Connect through a serial cable and use these serial settings as follows:

1 Make sure the CONFIG_* pins are set to the xmodem boot by specifying the EVP board or using the pins on the chip as follow:

```
SW1-3=1 SW1-4=0 SW1-5=1 SW1-6=1
```

See-> Section "Setting Switches for Non-EVP Boards," for other available switch settings on the chip.

2 Power on the board.

3 On the Serial Console (115200 8N1) check to see if "C" appears.

4 Transfer the **xmodem-bootloader.bin** file by using xmodem 1k. When the transfer is done the following will appear:

```
Mboot start... bootdevice ID 0x00000003
```

5 A memory test will follow. If any error occurs, it will probably mean the board is defective.

6 After approximately 30 seconds that the test is done, the following will display:

```
Preparing to receive kernel image
CCCCC
```

7 Transfer the **xmodem-kernel.img** file by using xmodem 1k.

8 Next transfer the **xmodem-initrd.img** file.

9 After this transfer is done the kernel should boot and display messages on the serial console.

10 When prompted to login, enter "root" for user ID and "mobiroot" for password.

Booting from NAND Images

- "Installing NAND Images for System Not Flashed"
- "Upgrading NAND Images for System Already Flashed"

Installing NAND Images for Systems Not Flashed

If a board has never been flashed or the flash becomes corrupted, the board needs to be booted via xmodem.

See-> Section "Booting from xmodem Images," for instructions on how to boot from xmodem.

Note: This does not recover blocks that have been marked as "bad blocks."

USAGE

All NAND images must have the following names when using the **mtd_dev_init** script:

- nand-bootloader.bin
- nand-initrd.img
- nand-kernel.img
- nand-rootfs.tgz

1 After booting from xmodem, the images must be made accessible. This can be done via NFS or by plugging in a USB flash drive with the images present.

2 Plugin in the USB flash drive. This device should become visible in /media.

Or

Mount a remote filesystem via NFS

```
$ modprobe nfs
$ portmap
```

3 Mount the machine where your flash images are for example:

```
$ mount 192.168.0.1:<work_dir> /mnt
```

4 Change directory to where the images are present and run the following script.

Mobilygen will provide the latest partition information in the release notes.

```
$ /bin/mtd_dev_init --parts "<nand_partition_information>"
```

5 Reset the dip switches for NAND boot and reboot the board.

See-> Section “Setting Switches for Non-EVP Boards” for Non-EVP Boards,” for other available switch settings on the chip.

6 Reboot your system.

```
$ reboot
```

Upgrading NAND Images for Systems Already Flashed

To upgrade a system that has already been flashed, use the “upgrade” utility script.

1 During boot up, look for the following message:

```
.
.
serial8250.10: ttyS2 at MMIO 0x88201800 (irq = 16) is a 16550A
TCP cubic registered
Freeing init memory: 68K
initramfs script version 1.0.
Loading driver for board type: mg3500_evp.
Press ‘U’ to upgrade the board ... skipped.
```

2 Press "U" when prompted if not pressed already.
There will be two seconds of delay before the system will resume the normal boot process.

3 The following will display:

```
Press 'U' to upgrade the board ... U
Upgrading board:
```

4 Select one of the following available options:

- **abort**—Resume the boot process.
- **nfs**—Get new images via NFS.
- **ftp**—Get new images via FTP.
- **http**—Get new images via HTTP.
- **nand**—Get new images from data partition on NAND.
  Images must be copied while system is up and re-partition is not allowed.
- **uart**—Get new images via serial connection.
- **shell**—Drop into shell.
- **USB—**Only a single USB device can be plugged in.

5 After selecting an upgrade method follow the prompts. The following is an example of an upgrade via NFS:

```
Press 'U' to upgrade the board ... U
Upgrading board:
Supported upgraded modes are: abort ftp http nfs uart usb
nand shell
Upgrade from: nfs
Server name or IP: qu062
Path to images : /export/home/jhane/project/merlinevp/dist
Please enter the name of the files on the server:
Upgrade bootloader(Y/N): Y
Enter bootloader filename(nand-bootloader.bin):
Upgrade kernel(Y/N): Y
Enter kernel filename(nand-kernel.img):
Upgrade initrd(Y/N): Y
Enter initrd filename(nand-initrd.img):
Upgrade rootfs(Y/N): Y
Enter rootfs filename(nand-rootfs.tgz):
Re-partition MTD device(Y/N) ? : n

You requested an upgrade from nfs using the following files:
Bootloader: nand-bootloader.bin
Kernel: nand-kernel.img
Initrd: nand-initrd.img
RootFS: nand-rootfs.tgz
Those images will be downloaded from qu062.
Is this correct(Y/N) ? Y

WARNING: All data in selected sections will be deleted!!
Do you want to continue (Y/N) ?

The release notes will provide the current NAND partition
sizes and indicate if these have changed since the previous
release. However, to verify current partition sizes of your
board cut-n-paste the following at the prompt:

cat /proc/mtd | grep mtd  |
while read line
do
partid=`echo $line |cut -d" " -f1`
size="0x`echo $line |cut -d" " -f2`"
echo "$partid $((${size} / 0x400))k"
done

Compare the output of this with the partition sizes provided
in the release note to see if you need to repartition your
NAND.

If you need to repartition, answer ‘Y’ to the prompt and cut-
n-paste the new definition from the release notes.
```

Setting Switches for Non-EVP Boards

This section describes the available options for setting up the switches on the chip in addition to the ones available for setting up xmodem and 8-bit NAND devices.

Additional xmodem Switch Settings

Set switches by using the pin numbers on the chip based on the desired operation:

S31-3	S31-4	S31-5	S31-6	Boot mode
1	0	1	1	Load from UARTDBG using xmodem.
1	1	0	0	Boot disabled.
1	1	0	1	Boot from DDR.

Additional NAND Switch Settings

Set switches by using the pin numbers on the chip based on the desired NAND operation:

S31-3	S31-4	S31-5	S31-6	Boot mode
0	0	0	0	Load from large 8-bits NAND <= 1Gbits on CS1.
0	0	0	1	Load from large 8-bits NAND > 1Gbits on CS1.
0	0	1	0	Load from large 16-bits NAND <= 1Gbits on CS1.
0	0	1	1	Load from large 16-bits NAND > 1Gbits on CS1.
0	1	0	0	Load from large 8-bits NAND <= 256Mb on CS1.
0	1	0	1	Load from large 8-bits NAND > 256Mb on CS1.

4

Connecting to the I²C Controller

Mobilygen MG3500 uses Inter-Integrated Circuit I²C to connect low-speed peripherals to a motherboard or an embedded system.

The following topics are covered in this chapter:

- "I²C Protocol Tools"
- "Supported Drivers"
- "Configuring Buses"

I²C Protocol Tools

The I²C protocol enables IC to IC communication via

- Two dedicated I²C hardware blocks
- A GPIO-based, "bit-banging" interface

Two I²C Dedicated Hardware Blocks

The two dedicated I²C hardware blocks are the most efficient way for MG3500 to communicate with other chips, enabling them to achieve the highest transfer speeds. They can use DMA to relieve the CPU. They can act both as a bus master or slave.

The two dedicated I²C blocks are called “I²C_0,” and “I²C_1.”

I²C_0 is directly connected to MG3500 pins SCL0 and SDA0 and therefore controls only one I²C bus. However, I²C_1 has a multiplexer which allows it to control two different busses. Pins for those two busses are respectively VID01_SCL, VID01_SDA and VID23_SCL, VID23_SDA.

Thus the total number of dedicated busses supported by the two dedicated I²C hardware blocks is three.

Note: You cannot use I²C_1 and SPI_1 at the same time.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," for more information about using the correct pins with the I²C hardware blocks.

A GPIO-Based Bit-banging Interface

The GPIO-based “bit-banging" interface is an alternative to communicating with other chips. This method is more demanding of CPU cycles, which results in its inability to achieve speeds higher than approximately 50Kbps. In this case, the MG3500 can only act as a bus master. Using GPIO-based I²C interface can be a fallback solution if you have timing issues with the dedicated blocks or need more I²C buses.

However, the GPIO-based interface allows you to control more than three busses. By default, the kernel is compiled to provide access from user-space to those three dedicated mentioned above, and finally accessing the three I²C-over-GPIO busses.

Those I²C-over-GPIO busses are actually using the same pins used by the three dedicated busses (unless you change this in the board driver), requiring a multiplexer witch from a dedicated bus to a I²C-over-GPIO bus. This switch can be done easily at runtime from user-space.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," to learn how to switch from a dedicated bus to a I²C-over-GPIO bus.

SUPPORTED DRIVERS

KERNEL DRIVERS

The drivers needed for using the dedicated busses are

Driver Name	Description
`i2c-core`	Core driver for the I²C subsystem of the Linux kernel.
`i2c-dev`	The driver that exports the I²C busses to user-space, thus creating device nodes in /dev.
`dwapbi2c`	The driver that interacts directly with the hardware blocks.
`i2c-mux`	The driver that controls the multiplexing of busses for I²C_1.

The drivers needed for using I²C-over-GPIO busses are:

Driver Name	Description
`i2c-core`	The core driver for the I²C subsystem of the Linux kernel;
`i2c-dev`	The driver that exports the I²C busses to user-space, thus creating device nodes in `/dev`.
`dwapbgpio`	The driver that controls the GPIOs.
`gpioi2c`	The bit-banging I²C driver.

The user may also need to load the driver where all the busses and devices of the MG3500 are registered, as are called the "board driver" or "board file", and "`board_mg3500evp`" for EV. This is also where the user needs to look if adding or removing I²C buses registration is desired, which will require recompiling the Linux kernel.

By default, all drivers mentioned previously are loaded at kernel startup except "`gpioI²C`" so that the user may load manually as needed, by running:

```
modprobe gpioi2c
```

Note: loading `gpioI²C` will not activate the I²C-over-GPIO busses since they will be present but ineffective.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," to learn how to I²C-over-GPIO busses, which results in disabling the corresponding dedicated bus.

Device Nodes

The `i2c-dev` driver will create device nodes in `/dev` one for each bus, along with aliases for easier usage.

Typically, the user will find those device nodes, assuming all the drivers have been loaded:

Device Node	Description
`i2c` and `i2c-0`	Both correspond to the bus controlled by I2C_0.
`i2c-1`	Corresponds to the bus(es) controlled by I2C_1, but does not set the multiplexer. **Note:** You should not use this node.
`i2c-2` and `i2c-v01`	Correspond to the first bus controlled by I2C_1. It will automatically set the multiplexer properly.
`i2c-3` and `i2c-v23`	Correspond to the second controlled by I2C_1. It will automatically set the multiplexer properly;
`i2c-4` and `i2c-gpio-0`	Correspond to the GPIO-based bus that will use the same pins as i2c-0.
`i2c-5` and `i2c-gpio-v01`	Correspond to the GPIO-based bus that will use the same pins as i2c-v01.
`i2c-6` and `i2c-gpio-v23`	Correspond to the GPIO-based bus that will use the same pins as i2c-v23.

CONFIGURING BUSES

I2C_0 and I2C_1 allow changing the bus settings at runtime.

This can be done through virtual files, respectively `/sys/devices/I2C_0/config` and `/sys/devices/I2C_1/config`.

To display current settings, enter

```
cat /sys/devices/I2C_*/config
```

This will typically produce:

```
dwapbi2c 3:1.1: I2C_0 Bus Configuration (clk=108000000Hz)

 (M) Master/Slave Mode:      MASTER
 (S) I2C Speed:              STANDARD (100kbit/s)
 (A) Slave Address:          0xaa (transmitting on 0xab)
 (T) Transfer Timeout:       3000msecs
 (R) Master Restart Feature: Enabled
 (D) Try Use DMA:            NO
 (t) SDA Setup Delay:        100 periods (925ns)

echo '<letter> <value>' in this file to change value.
dwapbi2c 3:1.1: I2C_1 Bus Configuration (clk=108000000Hz)

 (M) Master/Slave Mode:      MASTER
 (S) I2C Speed:              STANDARD (100kbit/s)
 (A) Slave Address:          0xac (transmitting on 0xad)
 (T) Transfer Timeout:       3000msecs
 (R) Master Restart Feature: Enabled
 (D) Try Use DMA:            NO
 (t) SDA Setup Delay:        100 periods (925ns)

echo '<letter> <value>' in this file to change value.
```

CHANGING SPEED SETTING SAMPLE

To change a setting, for example to set the speed to fast, enter

```
echo S FAST > /sys/devices/I2C_0/config
```

Description of the acceptable values

- Master/Slave Mode can be set to Master or Slave.
- Speed can be set to `LOW` (10Kbit/s), `STANDARD` (100Kbit/s), `FAST` (400Kbit/s), or `HIGH` (3.4Mbit/s).
- Slave Address accepts any non-reserved I²C addresses (like 0 or 1). This is the address the block will answer to if slave mode is enabled.
- Transfer timeout is the delay after which any transfer will be aborted. This is especially useful when the block is acting as a slave and you do not want to wait indefinitely for a remote master to poll you. This can be set to any value in milliseconds, or to `INFINITE` if you do not want a timeout. The wait will still be interruptible by a signal, for example by issuing a CTRL+C.
- Master Restart Feature can be Enabled or Disabled. This controls wether or not the I²C restart feature can be used.
- Trying to use DMA will tell the driver to setup DMA. This can fail especially if all the DMA channels (4 channels on the MG3500) are already in use. Acceptable values for this field are `YES` and `NO`.
- SDA Setup Delay, "t," determines how long the block should wait after it sets a value for SDA before releasing the SCL line.
  The I²C-Bus Specification from Philips requires a minimum data set-up time of 250ns for standard mode and 100ns for fast mode. In addition, Note 4 in the Philips specification states that when a device stretches the LOW period of the SCL signal, it must output the next data bit to the SDA line 1250ns before the SCL line is released. The SDA Setup Delay register enforces the timing requirement.

Note: If an I²C-over-GPIO bus is being used, changing its configuration at runtime is not possible. For this, the board driver needs to be modified which will require recompiling the kernel.

Using I²C Tool: I²CRW

This SDK provides two different tools to perform I²C transactions.

The first one and probably the most important one is `i2crw`.

Invoking the `i2crw` tool without specifying any arguments will result in display the following help:

```
ic2rw [...options...] <bus Dev Path> <slave Chip Address>
[<subduers>] [<byte1> <byte2> ...]

<busDevPath> is the device node path, e.g. /dev/i2c-0 (mandatory)
<slaveChipAddress> is the target chip address, e.g. 0x98 (mandatory
if acting as master)
<subAddress> is an optional sub-address, e.g. 0xab
<byteX> are optional data bytes, e.g. 0xab, 198...
```

AVAILABLE OPTIONS

Option Name	Description
`blk <size>, count <size>`	Read a block of data of size <*size*>, otherwise only one byte is read.
`no-subaddr`	No sub-address is needed or provided.
`subaddr <8|16>`	Size of the sub-address in bits. The default is 8.
`-16bits`	<*byte1*>, <*byte2*> ... are actually 16bit words.
`sccb`	Target chip uses the SCCB protocol (a variant of I²C) In this protocol the read operation is done in two transactions: "register addr." is first sent alone, then a "restart cond." occurs, after which time the value can be read.
`slave-read <size>`	Read <*size*> bytes from the I²C slave device. This request may stall until we are contacted by a master. **Note:** You cannot specify a chip address to talk to with this command.
`slave-write`	Specify bytes to be sent when I²C device is slave and is contacted by a master on the bus. **Note:** You cannot specify a chip address to talk to with this command.
`check`	Use when reading data to check that the data read match <*byte1*>, <*byte2*>, etc. Works only for master reading. Program will return the number of mismatches, 0 on success.
`quiet`	Do not display any messages
`raw`	Received bytes will be printed directly to STDOUT. This implies "--quiet."

USING I²C TOOL: I²CPROG

The `i2cprog` tool can be used to perform "batch" operations, taking a file with a series of read or write operations to perform. Calling tool without specifying any argument will result in displaying help.

ARGUMENTS

```
[-d|--dev <i2c_bus>] [-v|--verbose] [-h|--help] <script>
```

Argument Name	Description
`-d|--dev <i2c_bus>`	Name of the I²C bus device to open.
`-v|--verbose`	Program will describe what it is doing.
`-h|--help`	Display this help.
`<script>`	Script to run.

SCRIPT SYNTAX

A script is describing a list of Read and Write operations to perform on the I²C bus.

The first command of a script must be "A" to set the device address on I²C bus.

Addresses are sub-addresses and values are 8-bit integer values (0x means hex, O means octal, nothing means decimal.

If a script line contains "#," the remaining of the line is ignored.

Operation	Description
`A addr`	Set I²C device address on the bus. You can change the address any time in a script.
`D Delay`	<waiting for description from Ralph.>
`C subaddr value1 ... valueN [string]:`	Read I²C and compare result to <*value1*> ..., if not equal display <*string*> (if not NULL) and exit.
`R subaddr size [string]:`	Read <*size*> values on I²C, display <*string*>, and values read (if <string> is not NULL).
`W subaddr value1 ... valueN:`	Write N values at sub address <*addr*>. The address should be automatically incriminated by device.
`Q [string]:`	Quit and display <string>. EOF will be interpreted as a Q command with string=NULL

RETURN CODES

Code	Description
0	Script has been run successfully.
>0	Check the command failed at the line indicated by the positive return value.
-1	A system error happened, for example a file I/O, memory allocation, etc.
-2	An I²C library error occurred.
-3	A parse error occurred.

The `i2cexp` tool allows programming the I²C expander chips on the EVP.

```
ic2exp <options>  : i2c port expander utility
```

OPTIONS

Option	Description
`bus <i2c_bus>`	Specify I²C bus to open, The default is /dev/i2c-0.
`dev <0xNN>`	Specify device address on I²C bus.
`verbose`	Do not display any messages.
`set <bit>`	Set <*bit*> in port expander.
`clear <bit>`	Clear <*bit*> in port expander.
`readbit <bit>`	Print the value of a single bit.
`read`	Read all bits in a port expander. Outputs 0x<7..0> 0x<15..8>.
`raw-write 0x<7..0> 0x<15..8>`	Overwrite all bits with new data. Use this with caution!

USING I²C FROM YOUR OWN PROGRAM

To use I²C directly from own program, a library named `libi2c` may be used.

Refer to the documented header file of this library to learn about its usage. This header is typically located in `dist/libi2c/1/i2c.h` and if the "dist" source tree has been rebuilt and installed, it will be copied into `dist/merlin/i686-linux/arm-merlin-linux-uclibc/usr/include`.

The library itself will be available when the "dist" source tree is rebuilt and installed inside `dist/merlin/i686-linux/arm-merlin-linux-uclibc/usr/lib`.

SLAVE MODE AND DYNAMIC BEHAVIOR

When acting as a slave, the driver for the dedicated blocks which is provided in this SDK will not allow you to forge dynamic responses to a master request.

For example, if you want to make believe that you had a register or value map and that you want a remote master to be able to read and write the register values in this map, you will not be able to do it unless you write your own driver.

In slave mode, the provided driver will only allow you to wait for a master to make a request, and the data that you are supposed to return in the response (if any) must be passed “in advance” to the driver, i.e. before it begins to wait for a request.

USING DWAPB I²C FROM YOUR OWN KERNEL DRIVER

The `dwapbi2c` driver allows you to configure the buses from within the kernel by using a minimal API.

This can be useful if you are writing your own driver on top of `dwapbi2c`. You will still need to use the I²C subsystem of the Linux kernel to request transfers. This API allows you only to control the driver specifics.

This API is described in file `dist/linux/mobilygen/include/linux/i2c/dwapbi2c.h`.

5

Connecting to the SPI Controller

Mobilygen MG3500 uses Serial Peripheral Interface (SPI) to enable serial exchange of data between two devices: a master and a slave.

The following topics are covered in this chapter:

- "SPI Hardware Blocks"
- "SPI Tools"
- "Using SPI"

SPI Hardware Blocks

The Serial Peripheral Interface (SPI) Bus for MG3500 provides three hardware SPI blocks:

- SPI_0
- SPI_1
- SPI_2

SPI_0 and SPI_1 are both master SPI blocks, while SPI_2 is a slave block. Contrary to I²C blocks, the fact that a SPI block is a master or slave cannot be changed by software. Both blocks can use DMA to relieve the CPU.

SPI_0 and SPI_2 share the same pins as the MG3500

These pins are also shared with the bitstream interface. Of the three interfaces, only one interface at-a-time can be chosen. The multiplexer that defines which of the interfaces can connect to the pins can be controlled from user-space at runtime.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," for information about using the multiplexer.

SPI_1 shares its pins with I2C_1

This means that the user cannot use them at the same time.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," for information about how to change the multiplexer that controls which interface is connected to the pins.

As with I²C, the user can use GPIOs to have more SPI busses. The "bit-banging" SPI-over-GPIO driver is not as efficient as the dedicated hardware blocks, meaning that it will be more CPU intensive and will not run as fast. This driver is not enabled by default, so you will have to reconfigure and recompile the kernel to use it. Also, you will have to register the additional SPI-over-GPIO busses inside the board driver. See the following for additional details.

KERNEL DRIVERS

The following are the drivers that need to use SPI:

Driver Name	Description
`dwapbssi`	Controller driver that interacts directly with the hardware blocks.
`at25`	Client driver specifically designed to talk to EEPROM chips. It will create a virtual file that you can use to access target EEPROM contents directly. This file is typically reside in `/sys/devices/SPI_X/spiX.Y/eeprom`, where X is the bus number, and Y is the chip select number.
`spidev`	Client driver that creates device nodes in `/dev` that you can use to access the buses.

`dwapbssi` and `at25` should be loaded by default at board startup because they are needed on EVP to read board name inside the on board EEPROM.

Note: `spidev` is not loaded by default. If accessing any `/dev/spidevX.Y` device node is needed, run

```
modprobe spidev
```

Also, using a SPI-over-GPIO bus is needed, enabling the compilation of the following drivers in the Linux kernel configuration will be needed:

- Device Drivers -> SPI -> Bit-banging SPI master
- Device Drivers -> Mobilygen GPIO drivers -> Support for SPI over GPIO (`gpiospi`)

The drivers to be loaded will be named respectively "`spi_bitbang`" and "`gpiospi`." As with I²C-over-GPIO, you also need to load the "`dwapbgpio` driver," but this one is loaded by default at boot time.

CONFIGURING BUSES

The `dwapbssi` driver allows changing most of the configuration for buses and devices on these buses at runtime. This is done via virtual files.

To view current configuration, do

```
cat /sys/devices/SPI_X/config
```

Where X is the bus number. This will print for example, for SPI_0:

```
dwapbssi 4:1.0: SPI_0 MASTER Bus Configuration
(ssi_clk=120000000Hz)
Theorical Max. Transfer Speed:   60000000Hz

 (S) Speed:                      (Default per-chip
configuration.)
      Transfer Timeout:           (Automatic in master
mode.)
 (X) Disabled:                   NO
 (I) Ignore RXFIFO Over. Intr.: NO

0.0 SLAVE Device (chipselect 0):
 (M) SPI Mode:                   3
 (C) Client Driver/medullas:    at25
 (S) Speed:                      3000000Hz
 (D) Use TX DMA:                 YES
 (d) Use RX DMA:                 YES
0.1 SLAVE Device (chipselect 1):
 (M) SPI Mode:                   3
 (C) Client Driver/modalias:    spidev
 (S) Speed:                      10000Hz
 (D) Use TX DMA:                 YES
 (d) Use RX DMA:                 YES

echo '<item> [<chipselect>] <value>' in this file to
change setting.
```

Here is a description of each field along with acceptable values:

Field Name	Description
Speed	The transfer speed in Hz that should be used for every transfer. This overrides the transfer speed that is individually set for each chip on the bus. Note that this speed is not guaranteed, meaning that if the hardware block cannot achieve this speed a warning will be displayed and the closest achievable speed will be used. The driver also sets restriction on the speed depending on wether or nor you DMA is used. When DMA is not used, speeds higher than 3MHz will not be allowed.
Transfer	Timeout is only relevant for SPI_2, which is a slave block. This defines how long the driver will wait for a remote master to request for a transaction. This can be set to any value in milliseconds or to `INFINITE` if you do not want a timeout. Be careful that the thread where the transfer (and possibly the wait) will take place will NOT be the same as the process that issued the transfer. This means that if you set the timeout to `INFINITE`, a signal like CTRL+C that was issued from user-space on the program requesting the transfer will not break the wait since the transfer has been running asynchronously into a kernel work queue. However, you can abort a transfer/wait by using the driver API or from user-space by running: `echo abort > /sys/devices/SPI_2/config`
Disabled	Use “X” to change the "Disabled" state of the driver. The driver will disable itself when it is requested to use DMA but it cannot initialize DMA, or when it cannot get any free channel. The acceptable values are `YES` and `NO`.
Ignore RXFIFO Overflow Interrupt	This option is useful when for example the driver is a slave (that is, using SPI_2) and that receiving data as fast as possible is desired. The RX FIFO overflow interrupt will be ignored, indicating that the transfer will go on regardless of the errors. In this case, implementing an error correcting code to correct potentially erroneous bytes will be necessary.
SPI Mode	Determines what is the type of serial transfer used by the remote chip: 0: Serial clock toggles in middle of first data bit, inactive state of serial clock is low; 1: Serial clock toggles at start of first data bit, inactive state of serial clock is low; 2: Serial clock toggles in middle of first data bit, inactive state of serial clock is high; 3: Serial clock toggles at start of first data bit, inactive state of serial clock is high, and the chip select is not de-asserted between each data byte.
Client Driver/ Modalias	The setting defines which driver sits between SPI and the controller driver, `dwapbssi`. Since the SPI subsystem of the Linux kernel does not allow multiple client drivers, this allows witching back and forth from a driver to another.

Speed

The setting for a slave device defines the speed that should be used for this particular slave chip. This value is overridden by the global Speed configuration setting. All the notes for the global Speed setting apply also to this per-chip Speed setting. This per-chip setting has the same "letter" as the top level speed setting. Make sure you understand correctly how to specify a per-chip speed setting and not the global one:

```
# This will set the global speed and override every other
speed setting:
echo S 10000 > /sys/devices/SPI_0/config

# This will set the speed for slave chip at chipselect 1:
echo S 1 10000 > /sys/devices/SPI_0/config
```

Use TX DMA

The acceptable values are YES and NO. When set to YES, the driver will request a DMA channel for a particular direction: TX is for data sent by MG3500, whereas RX is for data received. For best performance, always make sure that if you choose to request only one direction as DMA, the other direction is not needed by the transfers performed. Therefore, avoid requesting a DMA channel for TX or RX only when you have data going in both directions. Otherwise, you may see warnings messages about the speed being limited by the driver.

Use RX DMA

Note: : If you an SPI-over-GPIO bus is used, changing its configuration at runtime will not be possible. In this case, modifying the board driver to do that and recompiling the kernel will be necessary.

SPI Tools

Two utilities are provided to communicate using the SPI protocol:

- `spirw`
- `spiprog`

spirw Tool

The `spirw` allows performing single transactions, while enabling testing an EEPROM. When no parameters are provided, the program will display this help:

```
*** SPIDEV/AT25 Utility Program ***

Sample usage:

spirw /dev/spidev1.0 --count 4
 will read 4 bytes from /dev/spidev1.0

spirw /dev/spidev0.1 0xaa 0xbb
 will write 2 bytes to /dev/spidev0.1 and read 2 bytes in the same
 transaction

spirw /dev/spidev0.1 --count 4 0xaa 0xbb
 will write 2 bytes to /dev/spidev0.1 and read 4 read bytes in the
 same transaction (null padding added)
 --count cannot be less than number of bytes to write.)

spirw /sys/bus/amba/devices/SPI_0/spi0.0/eeprom --eeprom-test -
 size 32768 --count 64 --offset 0x100
 will test writing/reading eeprom file with random data, comparing
 results
```

Other options (not for EEPROM test):

Field Name	Description
`16bits`	Input data is 16 bits wide.
`speed <KHz>`	This will specify a custom transfer speed in KHz.
`infile <file>`	Input data file.
`outfile <file>`	Output data file.

SPIPROG TOOL

The second supported SPI tool is `spiprog`. Similar to `i2cprog`, it takes a file as parameter with a series of transactions that you want to perform, as shown below:

```
--- SPI Communication Program Usage ---
[-d | --dev /dev/spidevX.Y] [-v | --verbose] [-h | --help] <script>

Where:
-d | --dev /dev/spidevX.Y : SPI device to communicate with
-v | --verbose             : enable verbose mode
-h | --help                : display this help
<script>                   : script to run
```

SCRIPT SYNTAX

A script is a list of read and write operations to perform on the SPI device. A string must be started and terminated by "or."

Syntax	Description
`#`	Comment.
`D "/dev/spidevX.Y"`	Change device to communicate with, only at the beginning of the script.
`W <bitCount> byte1 ... byteN`	Write something to the device.
`R <bitCount>`	Read from the device.
`A <bitCount> byte1 ... byteN`	Write and read simultaneously.
`Q "Exit Message"`	Print the message and exit.

RETURN CODES

Syntax	Description
`0`	Script has been run successfully.
`>0`	Check command failed at the line indicated by the positive return value.
`-1`	A system error happened, such as a file I/O, memory allocation, etc.
`-2`	A libspi error has occurred.
`-3`	A parse error has occurred.

USING SPI

USING SPI FROM YOUR OWN PROGRAM

If the user wishes to use SPI directly from own program, the user can use a library named "`libspi`." This also enables repeating the functions that the library has in the user's own program. The reason for this ability is that the library now consists of direct calling of the `ioctl` function on device nodes created by the `spidev` driver.

Refer to the documented header file of this library to learn about its usage. This header is typically located in `dist/libspi/1/spi.h`, and if the "dist" source tree has been rebuilt and installed, it will be copied into `dist/merlin/i686-linux/arm-merlin-linux-uclibc/usr/include`.

The library itself will be available when the "dist" source tree is re-built and installed, inside `dist/merlin/i686-linux/arm-merlin-linux-uclibc/usr/lib`.

USING DWAPBSSI FROM YOUR OWN KERNEL DRIVER

The `dwapbssi` driver allows configuring the buses and aborting a running transfer from within the kernel through using an API.

This API is described in file `dist/linux/mobilygen/include/linux/spi/dwapbssi.h`.

This can be useful if the user is writing own driver on top of `dwapbssi`. This API is not intended to replace the SPI subsystem of the Linux kernel; however, it will allow controlling the driver specifics. The user should still request transfers by using the API described in `include/linux/spi/spi.h` and `drivers/spi/spi.c` in the kernel source tree.

6

Connecting to the GPIO Controller

Mobilygen MG3500 uses the General Purpose Input/Output (GPIO) interface to communicate with the peripherals devices.

The following topics are covered in this chapter:

- "GPIO Hardware Blocks"
- "Controlling GPIO Pins from User Space"
- "Controlling GPIO Pins from your Kernel Driver"

GPIO Hardware Blocks

MG3500 supports three hardware GPIO blocks:

- GPIO_0
- GPIO_1
- GPIO_2

Each block is also called a “bank,” since each of the three blocks can actually control several GPIO pins:

- bank GPIO_0 can control 8 GPIO pins, numbered 0 to 7
- bank GPIO_1 can control 32 GPIO pins, numbered 8 to 39
- bank GPIO_2 can control 32 GPIO pins, numbered 40 to 71.

Note: Only the GPIOs from bank 0 have actual dedicated pins on the MG3500 package. Other GPIOs are multiplexed with other peripherals and need to be switched to GPIO mode before they can be used.

See-> Chapter 8," Connecting to the Input/Output Multiplexer," to learn about the `ioctrl` multiplexer.

Only one module needs to be loaded to be able to use those banks: `dwapbgpio`. It should be loaded by default.

CONTROLLING GPIO PINS FROM USER SPACE

Three ways to control the GPIO pins are available:

- Using entries in `/proc/driver/gpio/native`
- Using entries in `/sys/class/native`
- Using device nodes in `/dev/gpio/native`

Note: For SDK2, replace "native" with "merlin."'

Each way has its own benefits and particularities, as are described below to help selecting the one that suits your application the best.

/PROC DIRECTORIES

You will find those files in `/proc/driver/gpio/native`:

```
00    10    20    30    40    50    60    70
01    11    21    31    41    51    61    71
02    12    22    32    42    52    62    bank0
03    3     23    33    43    53    63    bank1
04    14    24    34    44    54    64    bank2
05    15    25    35    45    55    65    status0
06    16    26    36    46    56    66    status1
07    17    27    37    47    57    67    status2
08    18    28    38    48    58    68
09    19    29    39    49    59    69
```

Each numbered file (00 through 71) controls the GPIO pin with the same number.

To get current GPIO pin value, regardless of whether the MG3500 is driving it or not, simply `cat` the file. It will print out 1 or 0 depending on wether or not the current value is driven or it is read high or low, respectively.

- To make the MG3500 drive the pin high or low, echo respectively 1 or 0 into the file.
- To set the pin in tri-state mode, i.e. to stop driving the pin, echo 'z' into the file.
- To have the pin invert briefly its state, i.e. to strobe/pulse, echo 'p x' into the file, where x is the delay in microseconds that the pulse should last.

Files `bank0`, `bank1`, `bank2` allow controlling all or some of the pins of the given bank at the same time (bulk operations).

To get the value of all the pins of the bank, simply `cat` the file. It will display a 32 bit hexadecimal integer, for example 0x0ac7ffff.

Note: Even if the bank is not 32 bit wide, for example for bank 0 which is 8 pins = 8 bit wide, it will always print a 32 bit hexadecimal integer and unused high order bits will always be 0.

To set the value of all OR some of the pins in the bank, echo "<mask> <data>" into the bank file, where <mask> is the 32 bit mask to define which pins are to be affected by the operation and <data> is the 32 bit new value for those pins.

Files `status0`, `status1`, `status2` enable displaying an overview of the status of all the pins for each bank. The only operation available on these status files is `cat`. The first line is a legend indicating how to read the table:

- <enabled> ("e" for enabled or "d" for disabled) indicates that the pin is currently in GPIO mode as defined by the `ioctrl` driver.
- Chapter 8," Connecting to the Input/Output Multiplexer," for more information about the pin multiplexers and how to change them.

Note: Changing a GPIO pin value, such as direction will have no effect on the output if the GPIO mode is disabled (even though the status will change.)

- <dir> is the direction. If it is "i," then the pin is in tri-state mode. It will be o if the pin is being driven by the MG3500.
- <val> is the current value, either the value read in tri-state mode or the value driven.
- <int-enabled> indicates whether the interrupts for the pin are enabled or not.
- <active> indicates whether an interrupt is currently active.
- <owner> is the current owner of the pin in the kernel. Typically, this will be the name of a driver which has requested to exclusively use the pin.
  This will not prevent you from modifying the pin status, however, you must be aware that doing so may alter the functioning of the driver which uses the pin.

Here is a sample output for `cat status2`, when the driver `gpioi2c` has been loaded:

```
bank2: <enabled> <dir>,<val> <int-enabled>,<active> <owner>
00,gpio040: d i,0 0,0
01,gpio041: d i,0 0,0
02,gpio042: d i,1 0,0
03,gpio043: d i,0 0,0
04,gpio044: d i,1 0,0
05,gpio045: d i,1 0,0
06,gpio046: d i,1 0,0
07,gpio047: d i,1 0,0
08,gpio048: d i,1 0,0
09,gpio049: d i,1 0,0
10,gpio050: e i,1 0,0
11,gpio051: e i,1 0,0
12,gpio052: d i,0 0,0
13,gpio053: d i,1 0,0
14,gpio054: d i,0 0,0 gpioi2c.2.SCL
15,gpio055: d i,1 0,0 gpioi2c.2.SDA
16,gpio056: d i,1 0,0
17,gpio057: d i,1 0,0
18,gpio058: d i,1 0,0
19,gpio059: d i,1 0,0
20,gpio060: d i,1 0,0
21,gpio061: d i,1 0,0 gpioi2c.0.SCL
22,gpio062: d i,1 0,0 gpioi2c.0.SDA
23,gpio063: d i,1 0,0
24,gpio064: e i,1 0,0
25,gpio065: d i,0 0,0
26,gpio066: d i,1 0,0 gpioi2c.1.SCL
27,gpio067: d i,1 0,0 gpioi2c.1.SDA
28,gpio068: d i,1 0,0
29,gpio069: d i,1 0,0
30,gpio070: d i,1 0,0
31,gpio071: d i,1 0,0
```

Note: Please note that the `/proc` GPIO interface does not allow you to configure the interrupts for a particular pin, only display some information about it. See `/sys` or `/dev` interface for that.

/SYS DIRECTORIES

The directory path is `/sys/class/native`, where directories named `gpioXXX` where `XXX` is the GPIO pin number (0 to 71) will be found. Those contain a file named `control` that provide the exact same functionality as the file `/proc/driver/gpio/native/XX`.

Additionally, you will also find dDirectories named `gpioXXXint` where XXX is the GPIO pin number (0 to 71). Those contain a file named `control` that can be used to control the interrupts associated with the GPIO pin:

- `cat` it to display detailed information about the current interrupt settings.
- echo 1/0 into the control file to enable/disable the interrupts for the pin.
- echo "x" to clear an active interrupt.
- echo "c<*letter*>" to configure the interrupt mode, where <*letter*> is one letter as described below:
  - **e** sets type to edge-triggered;
  - **l** sets type to level-triggered;
  - **b** sets type to both (not supported on the MG3500);
  - **p** sets the polarity to low level/falling edge;
  - **P** sets the polarity to high level/rising edge;
  - **q** sets the polarity to both;
  - **d** disables the debounce feature;
  - **D** enables the debounce feature.
- Directories named `bankX` where X is the bank number (0, 1 or 2). Those contain a two relevant files:
  - One named `control` that provides the exact same functionality as the file `/proc/driver/gpio/native/bankX`.
  - One named `status` that provides the exact same functionality as the file `/proc/driver/gpio/native/statusX`.

/DEV DIRECTORIES

The "`ioctl`" keyword used below refers to the standard `ioctl` function of the C library and is not the same as the "`ioctrl`" driver used to control the pin multiplexers for the MG3500.

One device node for each GPIO pin and one for each GPIO pin interrupt is created.

Nodes named `/dev/gpio/native/gpioXXX` where `X` is the GPIO pin number (0 - 71) allow the following file operations:

Operation	Description
read	This will return the current value of the GPIO pin. If the user tries to read more than one byte from the device node, say Y bytes, the GPIO pin will be polled Y times, and the read buffer will contain a series of Y zeros or ones. Each poll can be delayed by a certain amount of time. This amount of time can be setup through an ioctl on the device node.
write	This will set the driven value. The write buffer should contain a series of zero or non-zero bytes defining the consecutive values to drive. Each value setting can be delayed by a certain amount of time. This amount of time can be setup through an `ioctl` on the "bank" device node. look for GPIOBANK_RWTIMING below.
ioctl	The supported `ioctl` operations, declared in `dist/linux/mobilygen/include/gpio-core.h` are the following: • GPIO_READ: Return value of the `ioctl` will be the current pin value. • GPIO_SET: Pin will be driven high. • GPIO_CLEAR: Pin will be driven low. • GPIO_INPUT: Pin will be set to tri-state (input) mode.

GPIO Sample Code

```
int fd = open("/dev/gpio/native/gpio000", O_RDWR);

/* Read GPIO value. */
char v;
int c;
read(fd, &c, 1);
/* Alternatively: */
v = ioctl(fd, GPIO_READ);

/* Write GPIO value, tie it down. */
v = 0;
write(fd, &c, 1);
/* Alternatively: */
ioctl(fd, GPIO_CLEAR);

/* Write GPIO value, tie it high. */
v = 1;
write(fd, &c, 1);
/* Alternatively: */
ioctl(fd, GPIO_SET);

/* Set to tristate. */
ioctl(fd, GPIO_INPUT);

close(fd);
```

Nodes named `/dev/gpio/native/gpioXXXint` where `X` is the GPIO pin number (0 - 71) allow the following file operations:

Operation	Description
read	This will typically wait for Y interrupts to occur, where Y is the read buffer size. There will be no wait if • The count of interrupt that occurred since the last call to read is more than Y. • The device node was opened with the `O_NONBLOCK` flag, in which case `-EAGAIN` error code will be returned.
write	This will enable interrupts, specifying that they should be disabled after Y of them occurred, where Y is the write buffer size.
poll	This will typically wait for at least one interrupt to occur and then return, but it will not enable them prior to waiting. If an interrupt has occurred since the last read operation, it will not wait and return immediately. This poll operation should be used prior to the read operation to check that at least one interrupt has actually occurred since last read.

ioctl

The supported `ioctl` operations, declared in `dist/linux/mobilygen/include/gpio-core.h`, are the following:

`GPIOINT_ENABLE`: interrupts will be enabled.

`GPIOINT_DISABLE`: interrupts will be disabled.

`GPIOINT_IS_ACTIVE`: returns 1 if an interrupt is currently active.

`GPIOINT_CLEAR`: clears an active interrupt.

`GPIOINT_TYPE`: allows to configure the type of the interrupt. ioctl argument can be:

1: type is level-triggered.

2: type is edge-triggered.

3: type is level and edge-triggered.

`GPIOINT_POLARITY`: this allows changing the polarity of the interrupt. ioctl argument can be:

1: polarity is low-level/falling-edge;

2: polarity is high-level/rising-edge;

3: polarity is both levels or edges.

`GPIOINT_DEBOUNCE_ON`: enables/disables the debounce feature of the GPIO controller. The `ioctl` argument should be 0 to disable and 1 to enable.

POLLING GPIO SAMPLE CODE

```
// A simple utility that polls on a specified file
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>

int main(int argc, char** argv)
{
        struct pollfd pfd;
        int res;

        if (argc != 2) {
                printf("Usage: poll_util <filename>\n");
                return 1;
        }
        pfd.fd = open(argv[1], O_RDONLY);
        if (pfd.fd < 0) {
                printf("Error while trying to open file %s\n",
argv[1]);
                return 1;
        }
        pfd.events = POLLIN;
        if (poll(&pfd, 1, -1) < 0) {
                printf("Error returned by poll()\n");
                return 1;
        }

        close(pfd.fd);
}
```

Nodes named `/dev/gpio/native/bankX` where `X` is the bank number allow the following operations:

Operation	Description
read	This will read the value for the whole bank. If the read buffer size is more than the bank size, then the bank will be read several consecutive times. These consecutive reads can be delayed by a certain amount of time. See below how to configure this amount of time with `ioctl`.
write	This will write the whole bank with the write buffer value. If the write buffer size is larger than the bank size, several writes will be issued consecutively. These consecutive writes can be delayed by a certain amount of time. See below on how to configure this amount of time with `ioctl`. You have to set the write mask prior to issuing a write using an `ioctl`.

ioctl

The supported `ioctl` operations, declared in `dist/linux/mobilygen/include/gpio-core.h` are the following:

- `GPIOBANK_SETGRP`: Sets up the write mask. The argument of the ioctl must be a pointer to an unsigned long integer containing the actual mask.
- `GPIOBANK_GETGRP`: Gets the write mask. The argument of the ioctl must be a pointer to an unsigned long integer which is to receive the current mask.
- `GPIOBANK_INPUT`: Sets up all the pins of the bank to input mode.
- `GPIOBANK_WRITE`: An alternative way of writing to the bank. The argument of the `ioctl` must be a pointer to an unsigned long integer containing the new bank value. The mask defined with `GPIOBANK_SETGRP` will be used.
- `GPIOBANK_READ`: An alternative way of reading the bank. The argument of the ioctl must be a pointer to an unsigned long integer which is to receive the bank value.
- `GPIOBANK_WRISATOMIC`: Returns 1 if read/write operations on the bank are atomic, i.e. if all the pins will change at the same time when you issue a write. If a read operation, then it will capture all the pin values at the same time.
- `GPIOBANK_RWTIMING`: Sets up the delay between consecutive read/write operations as mentioned above. This is expressed in microseconds. The argument of the `ioctl` is the new value to set.

CONTROLLING GPIO PINS FROM YOUR KERNEL DRIVER

If the user wishes to control the GPIO pins from within own driver, using the API described in `dist/linux/mobilygen/include/gpio-core.h` will be needed.

If a GPIO pin is used as a client, a "request," meaning that other client drivers will not be able to request it as well will be needed.

However, this mutual exclusion mechanism is optional meaning that performing operations on the pins directly, without requesting them is also possible. The users, however, may write their own drivers carefully if they wish. The GPIO framework provided by MG3500 is very versatile and does not force the user to use any of its numerous features.

The best practice for a client driver is actually to register with the GPIO framework, declaring which GPIO pins you are interested in. When all the pins become available, that is when the controller driver is ready, then your client probe function will be called. When the controller driver is removed and the pins become unavailable, the remove function of your client driver will be called.

For a usage example of this registration mechanism and for an example usage of the API, see the `gpioi2c`, `gpiospi` and `gpio-client-sample` drivers, located in `dist/linux/mobilygen/drivers/mobigpio`.

A very basic usage example of the client API is also showed below. This will set the value of GPIO 64 to 1, and change the direction as well to output if need be.

CLIENT USAGE API EXAMPLE

```
/* This variable is used to store the GPIO index within the bank
where it is located. */
unsigned gpio_index;

/* This will hold the GPIO driver handle. */
gpio_driver_handle gpio_drv_h;

/* Find the driver associated with this GPIO. */
gpio_drv_h = gpio_driver_find(NULL, 64, &gpio_index);

/* Driver found? */
if (likely(gpio_drv_h)) {

/* GPIO driver was found/is available, change the direction/value.
*/
gpio_direction_output(gpio_drv_h, gpio_index, 1);

/* Release the handle. */
gpio_driver_put(gpio_drv_h);
```

7

Connecting to Interfaces

MG3500 supports Pulse Width Modulation (PWM) for controlling analog circuits, Universal Serial Bus (USB) to connect to a host interface, and a Watchdog Timer (WDT) as a hardware timing device.

The following topics are covered in this chapter:

- "PWM Hardware Blocks"
- "USB Host Interface"
- "Watchdog Timer (WDT)"

PWM Hardware Blocks

Three hardware PWM blocks reside on MG3500

- PWM_0
- PWM_1
- PWM_2

These hardware blocks are controlled by only one driver, `cadpwm`.

Note: Since the `cadpwm` driver is not loaded by default, the user needs to "modprobe" it.

Controlling from User-space

A couple of virtual files reside in `/sys/devices/PWM_X`, where `X` is the block number that allows configuring the PWM blocks from user-space:

- `enabled` (Readable/Writable): Enables or disables the block. When a block is disabled, the output signal on the pin will be flat. echo 1 or 0 in this file depending on wether you want to enable or disable the block.
- `frequency` (R/W): Frequency of the square signal that is output by the block.
- `frequency_min` (Read Only), `frequency_max` (RO): Boundaries for the achievable frequency.
- `dutycycle` (R/W): this is the duty cycle expressed in “per mil”:

```
# cat /sys/devices/PWM_0/dutycycle
500
/1000
```

- `dutycyle_min` (RO), `dutycyle_max` (RO): Boundaries for the achievable duty cycle.
- `period` (RW): An alternative way of setting the frequency. This is the number of input clock periods one output period should last.
- `hightime` (RW): An alternative way of setting the duty cycle. This is the number of input clock periods the output signal should stay high.

Controlling from within the Kernel

In some cases, the user may wish to configure the PWM blocks from own driver.

To do so, check the API available and documented in the following file:

```
dist/linux/mobilygen/include/linux/cadpwm.h.
```

USB Host Interface

Mobilygen's Universal Serial Bus (USB) provides USB developers with the functionality of converting Digital Video into H.264 compliant products that interface devices to a host computer.

To set up the USB interface do the steps in this section:

1 Load the modules.

```
modprobe dwc_otg
modprobe usb_storage
modprobe sd_mod
```

2 Plug in a USB flash drive or a USB hard disk.

A new entry `/dev/sda1` should show up in the `/dev` directory.

3 In most cases, `udev` will detect the device and mount it under the `/media` directory. If this does not happened, you will need to mount the disk manual. For example:

```
mount /dev/sda1 /mnt
```

Watchdog Timer (WDT)

A watchdog timer is a hardware timing device that initiates a system reset if the system fails to service the Watchdog services, bringing back the system from a hung state to a normal operation.

Watchdog Driver

This driver implements two watchdog operations: hardware and software.

The hardware watchdog monitors the system it will reset in case it had previously stopped responding, or if there is a problem with the software watchdog.

The software watchdog is handled by the user-space applications to handle application hangs.

How watchdog software operation works

1. Open `/dev/watchdog`.
2. Use the `ioctl` interface to set the timeout.
   Both second and millisecond resolution available
3. Specify desired settings.
4. "Pat the dog" via the `ioctl` interface.
   The application MUST do this to keep the watchdog from triggering.
5. Close `/dev/watchdog`.

If the software watchdog expires, the driver will attempt to kill the process that started the watchdog timer. If the driver is unable to kill the process the system will be reset.

Caution! Extreme caution should be used when using the software watchdog because of the obvious effect of resetting the system.

Loading Watchdog Driver

The watchdog driver is automatically loaded. However if the driver needs to be loaded or unloaded manually, the name of the driver to use is “dw_wdt.”

The default timeout of the driver is 17 seconds but this can be modified in two ways. First, a module param can be passed to the driver when it is loaded:

```
modparam dw_wdt hw_default_heartbeat=XX
```

where XX is the number of seconds.

The second method is to use the procfs interface:

```
echo XX > /proc/driver/watchdog/expires
```

where XX is the number of seconds.

To see what the current timeout is do the following:

```
cat /proc/driver/watchdog/expires
```

Note: Due to the hardware design exact time-outs are not possible. The driver will program the timeout to be as close as possible to the requested value without going over.

To see what the current status of the watchdog timer is

```
cat /proc/driver/watchdog/status
```

This will indicate the timeout for the hardware watchdog timer if enabled; otherwise, it will indicate disabled. Also it will show the PID of any processes that have registered for the software watchdog.

8

Connecting to the Input/Output Multiplexer

The `ioctrl` driver allows controlling all the multiplexers of the MG3500 pins.

The following topics are covered in this chapter:

- "Input/Output Control (IOCTRL): Host Multiplexer"
- "IOCTRL Hardware Blocks"
- "Using Macros"
- "Custom Settings"

Input/Output Control (IOCTRL): Host Multiplexer

The `ioctrl` driver allows controlling all the multiplexers for the MG3500 pins.

Although some of these multiplexers are automatically set up when you use the peripheral itself (for example with `I2C_1` and the `i2c-mux` driver that does it for you), a couple of multiplexers need to be set up manually before they can be used.

A system of macros are available to ease changing the multiplexers to a particular peripheral.

If you do not see any macros to perform the operations you need, you can still change settings on a per-pin basis.

Note: The `/sys` entries are only writable by root. Static setting can be set at startup in the board driver or settings can be changes in drivers via the API.

IOCTRL Hardware Blocks

When the `ioctrl` driver is in the master mode, it supports three interfaces

- Host Master
- NAND/NOR

The chip select signals for each of these interfaces allows signalling the start and end of an interface transaction. A single transaction owns the common set of IO pads/pins for the duration of the transaction. If two or more interfaces request the common set of IO pads/pins, no arbitration is performed by the hardware.

The external chip select signals are configured by the HostCSx fields of the HostMuxControl register.

For each output signal to the various modules, all signals will always be connected and active except for the chip select signals. These will be disabled if the module is not the active module selected. Inactive chip selects will be driven to the inactive state based upon the name of the module signal.

Using Macros

The file `/sys/bus/platform/drivers/ioctrl/macro` allows displaying all of the available macros, which pins they affect, and whether they are currently active or not.

To activate a macro, simply echo the macro name into this file. If successful, nothing will be displayed but you can check that everything was correctly set by doing `cat` on the file.

The macros that you will be likely to need the most are described below.

Switching Between SPI_0 and SPI_2 Macro

To enable the SPI master bus, SPI_0, do

```
echo SPI_0 > /sys/bus/platform/drivers/ioctrl/macro
```

To enable the SPI slave bus, SPI_2, do

```
echo SPI_2 > /sys/bus/platform/drivers/ioctrl/macro
```

Switching Between SPI_1 and I2C_1 Macro

To enable both chip selects of SPI_1 (V01 and V23), do

```
echo SPI_1 > /sys/bus/platform/drivers/ioctrl/macro
```

To enable chipselect V01 of SPI_1, do

```
echo SPI_1.V01 > /sys/bus/platform/drivers/ioctrl/macro
```

To enable chipselect V23 of SPI_1, do

```
echo SPI_1.V23 > /sys/bus/platform/drivers/ioctrl/macro
```

To enable bus V01 or V23 of I2C_1, you do not need to activate a macro manually by using `/dev/i2c-v01` or `/dev/i2c-v23`. This will automatically activate the proper multiplexers and disable respectively chipselect V01 of SPI_1, or chipselect V23 of SPI_1.

Replacing I²C-over-GPIO Busses Macro

To enable the I²C-over-GPIO replacement for I2C_0, do

```
echo I2C_0.GPIO > /sys/bus/platform/drivers/ioctrl/macro
```

To enable the I²C-over-GPIO replacement for I2C_1, bus V01, do

```
echo I2C_1.V01.GPIO > /sys/bus/platform/drivers/ioctrl/macro
```

This will disable SPI_1, chipselect V01.

To enable the I²C-over-GPIO replacement for I2C_1, bus V23, do

```
echo I2C_1.V23.GPIO > /sys/bus/platform/drivers/ioctrl/macro
```

This will disable SPI_1, chipselect V23.

Custom Settings

The traditional control interface of the `ioctrl` driver includes in the following files:
`/sys/bus/platform/drivers/ioctrl/gpio`

This allows you to display (using `cat`) and change the pins that are connected to the GPIO banks or the peripherals and that whether they should be set to high or low.

gpio	Function	Pull Up/Down
0 gpio	(immutable)	U
1 gpio	(immutable)	U
2 gpio	(immutable)	U
3 gpio	(immutable)	U
4 gpio	(immutable)	U
5 gpio	(immutable)	U
6 gpio	(immutable)	U
7 gpio	(immutable)	U
8	primary/alt	U
9	primary/alt	U
10	primary/alt	U

The second column shows whether the pin is connected to the GPIO bank or to the peripheral(s). Except for the first eight pins which are immutable and will always be GPIO pins, change this setting:

- To connect the pin to its GPIO bank, `echo "function XX 1"` to this file, where `XX` is the GPIO number (0 - 71).
- To connect the pin to the peripheral(s), `echo "function XX 0"` to this file.

Whether the pin is pulled-up (U), pulled-down (D) or tri-state (Z).

- To enable the pull-up tie, `echo "pullup XX"` to this file, where `XX` is the GPIO number.
- To enable the pull-down tie, `echo "pulldown XX"` to this file.
- To disable pull-up and pull-down, `echo "tri-state XX"` to this file.

`/sys/bus/platform/drivers/ioctrl/function`

When a pin is not connected to its GPIO bank, this allows displaying (using `cat`) and changing which peripheral function is active (primary or alternative function):

Function	State
dbg_uart	0
i2c_cfg	0
v23_mosi	0
v23_mclk	0
v01_mosi	0
v01_mclk	0
spi_mosi	0
spi_mclk	0
spi_mss0	0
spi_mss1	0
spi_master	1

To activate a peripheral function, do an `echo "<function> <state>"` into the file, where `<function>` is one of the string in the first column and `<state>` is 0 or 1.

For a description of each function, see table below.

Function	Effect
dbg_uart	0: Connects ARM_DBG_UART to the DBG UART Port. (Default) 1: Connect QMM_DBG_UART to the DBG UART Port.
i2c_cfg	0: I2C_1 connects to the V01_* port 1: I2C_1 connects to the V23_* port
v23_mosi	0: V23_MOSI signal is active (Default) 1: V23_SDA signal is active.
v23_mclk	0: V23_MCLK signal is active (Default) 1: V23_SCL signal is active.
v01_mosi	0: V01_MOSI signal is active (Default) 1: V01_SDA signal is active.
v01_mclk	0: V01_MCLK signal is active (Default) 1: V01_SCL signal is active.
spi_mosi	0: SPI_MOSI signal is active (Default) 1: BS_DATA signal is active.
spi_mclk	0: SPI_MCLK signal is active (Default) 1: BS_CLK signal is active.
spi_mss0	0: SPI_MSS0 signal is active (Default) 1: BS_ENABLE signal is active.
spi_mss1	0: SPI_MSS0 signal is active (Default) 1: BS_REQ signal is active.
spi_master	0: the SPI interface is Slave (Default) 1: the SPI interface is Master.

```
echo I2C_1.V23.GPIO > /sys/bus/platform/drivers/ioctrl/macro
```

/sys/bus/platform/drivers/ioctrl/drivestrength

This allows setting up the drive strength for the output signal of a group of pins. This is the intensity of the current with which the group of pin is driven.

Group	Drive's Strength
serial	2
spi	1
v01spi	1
v23spi	1
video0	1
video1	1
video2	1
video3	1
host	1
ethernet	1
audio	2
sdmmc	2

To set the drive strength, `echo "<group> <step>"` into this file, where <*group*> is one of the pin group from the first column and <*step*> is the desired drive strength step. There is a 2mA interval between each step. Valid steps are 1, 2, 3, 4, 5, and 6, where 1 is the slowest and 6 is the fastest.

Note: Increasing the drive's strength can solve malformed signal issues but it will increase the power consumption of the MG3500.

`/sys/bus/platform/drivers/ioctrl/slewrate`

This will allow changing the slew rate for a group of pins.

Group	slewrate
serial	2
spi	1
v01spi	1
v23spi	1
video0	1
video1	1
video2	1
video3	1
host	1
ethernet	1
audio	2
sdmmc	2

The user may set the slew rate, `echo "<group> <step>"` into this file, where <*group*> is one of the pin group from the first column and <*step*> is the desired slew rate step. Valid steps are 0 (slowest), 1, 2 and 3 (fastest rate).

9

Setting up Audio and Video Chips

MG3500 SDK allows setting up Audio and Video Chips.

The following topics are covered in this chapter:

- "Setting up Video Peripherals"
- "High Definition Multimedia Interface (HDMI): AD9889 Transmitter"
- "Analog HDMI Dual Display Interface: AD9880 Receiver"
- "Source Selection Analog Anti-Aliasing: SAA7115 Video Decoder"

SETTING UP VIDEO PERIPHERALS

On the EVP, the user may set up the video chips using a set of programs called "`setCHIPNAME,"` for example "`setSAA7114`."

SYNTAX

```
setCHIPNAME [ <dev> <params> [ --verbose | --quiet | ] | --help ]
```

`verbose`, `quiet` and `help` are common to all of them.

`verbose` or `-v`: Set to make program more verbose.

`quiet` or `-q`: If set the program will not print anything (even errors).

`help` or `-h`: Display this help.

Note: Always specify the I²C bus (--bus) and address (--dev) of a chip when issuing a command since there can be more than one chip of this type on the board.

For example, to set the first `SAA7114` chip to NTSC by using the `video` connector, enter

```
setSAA7114 --bus /dev/i2c-2 --dev 0x40 --ntsc_svideo
```

Multiple arguments can be specified. For example, the following command is valid:

```
setAD9889 --bus /dev/i2c-3 --dev 0x7A --format 720p --input\
YUV --output RGB --style 2 --656
```

High Definition Multimedia Interface (HDMI): AD9889 Transmitter

This chip is an HD transmitter and offers an HD output port. It is located on the `/dev/i2c-3` bus, with address 0x7A.

Configuration Parameter: ad9889_param

Argument	Definition
`bus <i2c_bus>`	Specify I2C bus to open (default `/dev/i2c`).
`dev <0xNN>`	Specify device address on I2C bus.
`format <mode>`	Specify the mode used among 480i, 480p, 720p, 1080i, 1080p.
`style <num>`	1,2 or 3.
`input <video>`	RGB, YUV or YUV13.
`output <video>`	RGB, YUV or YUV13.
`656`	internal sync.
`noDE`	enable DE signal regeneration.
`writeaddr <0xNN>`	Defines sub-address for data write.
`writedata <0xNN> [<0xNN>...<0xNN>]`	Write data, range specifies multi-byte.
`readaddr <0xNN>`	Defines sub-address for data read (one byte read by default).
`readsize <NN>`	Define to read more than one byte.

Analog HDMI Dual Display Interface: AD9880 Receiver

The HD receiver features a VGA port for component input and an HDMI input. You can reach it through the `/dev/i2c-3` bus, at address 0x98.

Configuration Parameter: ad9880_params

Argument	Definition
`bus <i2c_bus>`	Specify I²C bus to open (default `/dev/i2c`).
`dev <0xNN>`	Specify device address on I²C bus.
`format <mode>`	Specify the mode used among 480i, 480p, 720p, 1080i, 1080p.
`input <video>`	RGB, YUV or YUV13.
`output <video>`	RGB, YUV or YUV13.
`656`	internal sync.
`invpwr`	invert the polarity of the power-down pin (default is high)
`brightness <offset>`	Brightness level (0 to 100, default is 50).
`analog <channel>`	Force the chip to use a specific analog channel.
`digital`	Force the chip to use the digital input.
`writeaddr <0xNN>`	Defines sub-address for data write.
`writedata <0xNN> [<0xNN>...<0xNN>]`	Write data, range specifies multi-byte.
`readaddr <0xNN>`	Defines sub-address for data read (one byte read by default).
`readsize <NN>`	Define to read more than one byte.

Source Selection Analog Anti-Aliasing: SAA7115 Video Decoder

The EVP contains the `SAA7115` Philips TV decoder chips. Each having a `svideo` and a composite input port. Both chips are located on the `/dev/i2c-2` bus, on address 0x40 (port 0) and 0x42 (port 1).

You can also use "--writeaddr" and "--writedata" to write directly to the registers if the predefined modes and options aren't specific enough.

```
vhelp or -vh: Display verbose help.
```

CONFIGURATION PARAMETER: SAA7115_PARAMS

Argument	Definition
`bus <i2c_bus>`	Specify I²C bus to open (default `/dev/i2c`).
`dev <0xNN>`	Specify device address on I²C bus (default is 0x42.
Predefined configurations	
`ntsc_svideo`	Set to default NTSC S-Video config.
`pal_svideo`	Set to default PAL S-Video config.
`ntsc_composite`	Set to default NTSC Composite config.
`pal_composite`	Set to default PAL Composite config.
`nochroma_ntsc`	Disable chroma.
`sync`	Change sync setup.
Note for filters: "on" indicates active; "off" indicates bypass/deactive	
`antialias_filter <on|off>`	Anti-alias filter enable
`chroma_trap_filter <on|off>`	Chrominance trap filter enable
`luma_comb_filter <on|off>`	Luminance comb filter enable
`chroma_vertical_filter <on|off>`	Chrominance vertical filter enable
`adaptive_luma_comb_filter <on|off>`	Adaptive luminance filter enable
`adaptive_chroma_comb_filter <on|off>`	Adaptive chrominance comb filter enable
`writeaddr <0xNN>`	Defines sub-address for data write.
`writedata <0xNN> [<0xNN>...<0xNN>]`	Write data, range specifies multi-byte.
`readaddr <0xNN>`	Defines sub-address for data read (one byte read by default).
`readsize <NN>`	Define to read more than one byte.