MG3500 PROGRAMMING GETTING STARTED GUIDE

November 14, 2008

Document Release 1.0

Document Number: PN984

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

www.mobilygen.com

PREFACE

Welcome to the *MG3500 Programming Getting Started Guide.*

This document 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.

The following topics are covered in Preface:

- "Document Overview"
- "MG3500 Codec Documentation Set"
- "Revision History"
- "References"
- "Contact"

Document Overview

The following are the chapters in this guide:

- Chapter 1," Overview of Programming MG3500 Codec."
- Chapter 2," Codec Objects Definitions."
- Chapter 3," Lua Scripting Language."
- Chapter 4," From Lua to C."
- Chapter 5," Writing Scripts."
- Chapter 6," Setting up Codec Objects."
- Chapter 7," Sample Scripts."

MG3500 Codec Documentation Set

- MG3500 Mobiapp Reference Manual
- MG3500 Codec Firmware API Reference Manual

Revision History

This document was prepared by Mobilygen Corporation.

Revision	Documentation Release Date	Notes
Internal: 0.6 External: 1:0	September 21, 2008	First internal release Second external release

References

The following are references to books, documents, and Web sites for three subjects.

[1] Lerusalimschy, Roberto, "*Programming In Lua, Second Edition*, May 2006." Lua.org, December 2003, ISBN 8590379825.

[2] Lerusalimschy, Roberto; de Figueiredo, L. H.; Celes, W., "*Lua 5.1 Reference Manual.*" Lua.org, August 2006. ISBN 85-903798-3-3. http://www.lua.org/manual/5.1/.

[3] Mobilygen Corporations, "*En-ViE Codec Firmware API Functional Specification*, Version 1.0."

Contact

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

CONTENTS

1

Overview of Programming MG3500 Codec

Mobilygen G3500 provides a host reference application called *Mgapp,* which in turn has its own plugin called *Mobiapp*.

The following topics are covered in this chapter:

- “Overview of Mobiapp”
- “Overview of Mobiapp User Interfaces”
- “Overview of Mobiapp C Objects”

Overview of Mobiapp

Mobiapp is Mgapp's main application that controls the operations and the datapath of the Codec. Mobiapp is an instance of Mgapp in a form of a shell script that loads the Mobiapp shared object (.so) and passes the appropriate command-line arguments to the Mobiapp plugin's entry point. If used "as is," Mobiapp controls the operations of the MG3500 audio and video Codec while acting as a reference for a customer-developed application.

See-> *MG3500 Mobiapp Reference Manual* for detailed information about the Mobiapp plugin application.

Topics in this section are

- "Mobiapp Architecture"
- "Other Mobiapp Components"

Mobiapp Architecture

Mobiapp consists of two main components: Interface Component (Lua Interpreter and C Programs) as well as C Objects Component. By default, the Interface component represents the user interface files that implement the Lua interface. These interface files, also known as *Lua Binding*, are identified as UI_SRC in the Mobiapp Makefile and include all of the Lua shell functionality.

The C Object component, however, consists of a set of C functions that implement the Codec APIs and are indicated as the C*ore source* in the Mobiapp Makefile. These C functions implement commands and events handlings, bitstreams storage, and retrieval and other necessary functions.

See-> Section "Overview of Mobiapp User Interfaces," for a brief introduction to Lua and C programs to interface with Mobiapp.

See-> Section "Overview of Mobiapp C Objects," for brief descriptions of the two types of Codec's C objects: Proxies and Host.

See-> Section "Other Mobiapp Components," for references to the boot loader and Codec commands components of Mobiapp.

Other Mobiapp Components

In addition to the Mobiapp interfaces and C objects, two other components comprise Mobiapp: Boot loader, as well as Codec Command and Event Protocol control.

See-> *MG3500 Mobiapp Reference Manual* for information on Firmware loader as well as the commands that implement the Codec commands and event protocols.

Overview of Mobiapp User Interfaces

Mobiapp uses command-line shells as well as the Lua interpreter as its main interface to control the operations of the MG3500 Codec. Lua interpreter enables Mobiapp to fully test the entire Codec APIs without using a C compiler or additional tools.

Although the Lua interpreter acts as the user interface of the Mobiapp plugin, building other user interfaces in C programs that either access the Mobiapp's C programs objects directly or use the Lua scripting language through standard and non-interactive mechanisms is also possible.

Topics in this section are

- "Lua Scripting"
- "C Programs"

Lua Scripting

Mobiapp uses a Lua interpreter that has been extended with bindings to the C objects. These Lua bindings allow writing scripts to perform all or some of the Codec operations without using a compiler or tools other than a text editor. The Lua interpreter is generated both as an interactive console session and as a server that can execute Lua *chunks* from a remote host via a socket interface.

Scripts can be stored in files which can be loaded and executed from the command line prompt, by using the following command:

```
mobiapp –f <script path>
```

See-> Chapter 3," Lua Scripting Language," for more information about the Lua interpreter.

The *Lua Programming Language Manual* for complete information about Lua scripting language.

C Programs

Using the Lua as the interface to Mobiapp is optional since all of the states and functions can be achieved through C code to interact with the C objects. You may choose to write applications that directly use the C objects and completely bypass Lua. However, keep in mind that the standard use of the Mobiapp reference application is through the Lua interpreter and shell along with scripts that manipulate the C objects appropriately.

See-> Chapter 4," From Lua to C," for instructions to remove Lua from your system and write your own C codes to interface with Mobiapp.

Overview of Mobiapp C Objects

The MG3500 API is implemented by using a set of Codec's object types. These Codec objects are created, controlled, and deleted both by using commands sent from the host to the Codec and through events that are received by the host and sent by the Codec.

Mobiapp implements a set of host-side objects which are "proxies" for their corresponding Codec object. This means that when a host object is created, a matching Codec object associated with that host object will be created as well. Similarly, when an API call is made to a host object, a corresponding command will be sent to the matching Codec object.

Included in Mobiapp is a set of objects that are written in C. These C objects enable Mobiapp to both implement the Codec's API for each of the Codec's proxy Object types, or to provide functionality related to bitstreams and meta-data created by the Codec *Host-only* Objects.

Topics in this section are

- Codec Proxies Objects
- Host Objects

See-> *MG3500 Mobiapp Reference Manual* for detailed information about the Mobiapp C objects.

Codec Proxies Object Types

The objects that reside on the host are referred to as "Codec Proxy Objects," when they exist primarily to control a Codec object.

See-> Chapter 2," Codec Objects Definitions," for definitions of the C objects.

See-> *MG3500 Mobiapp Reference Manual* for detailed information on these objects.

The following is a list of all available Codec proxy object types along with brief descriptions of each object.

LIVE VIDEO PREPROCESSOR (LVPP)

LVPP Controls the Video Input (VIP) and Video Pre-Processor (VPP).

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qlvpp.h`
`C Object`	`lvppapi.h, lvppapi.c`
`Lua` bindings	`lvpplua.c`

AUDIO ENCODER

The audio encoder is a Codec peer object that implements the API of the Codec's audio encoder object.

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qaudenc.h`
`C Object`	`audencapi.h, audencapi.c`
`Lua` bindings	`audenclua.c`

AVC ENCODER

The AVC encoder is a Codec peer object that implements the API of the Codec's AVC encoder object. It controls the Audio and Video Encoder.

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qavcenc.h`
`C Object`	`avcencapi.h, avcencapi.c`
`Lua` bindings	`avcenclua.c`

AVC DECODER

The AVC decoder is a peer object for the Codec's Audio-Video Decoder.

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qavdec.h`
`C Object`	`avdecapi.h, avdecapi.c`
`Lua` bindings	`avdeclua.c`

STEREO AUDIO INPUT

The stereo audio input is a Codec peer object that implements the API of the Codec object.

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qsain.h`
`C Object`	`sainapi.h, sainapi.c`
`Lua` bindings	`sainlua.c`

HOST VIDEO OUTPUT

The host-video output object acts as both a proxy for the Codec HVOUT object as well as implementing an event handler for the HVOUT itself.

Mobiapp Object Type	LVPP Object Name
`Codec API`	`qhvout.h`
`C Object`	`hvoutapi.h, hvoutapi.c`
`Lua` bindings	`hvoutlua.c`

AUDIO VIDEO OUTPUT COMPOSITION

This object is a peer object for the Codec's Audio-Video Output Compositor.

SAIN

The stereo audio input is a Codec peer object that implements the API of the Codec object.

Mobiapp Host Object Types

While most host objects are proxies for Codec objects, some host objects implement functionality on their own and do not have a peer Codec object. These objects are typically related to bitstream management and include the readstreamer and writestreamer objects.

READSTREAMER

The readstreamer is a host-only object that is responsible for reading bitstreams from the Codec, formatting them appropriately, and then storing or transmitting them to the ultimate destination.

- C Object—`readstreamer.c`
- Lua bindings—`readstreamerlua.c`

WRITESTREAMER

The writestreamer object handles bitstream writer.

- C Object—`writestreamer.h, writestreamer.c`
- Lua bindings—`writestreamerlua.c`

H4V BITSTREAM WRITER

The H4V bitstream writer is capable of writing an H.264 elementary video stream when receiving a AVC QBOX stream from the MG3500.

- C Object—`h4vwriter.c`

ADTS BITSTREAM WRITER

The ADTS bitstream writer is capable of writing ADTS compliant streams when used with the AAC-LC audio encoder on the MG3500

- C Object—`adtswriter.c`

AUDIO ELEMENTARY STREAM BITSTREAM WRITER

The audio elementary stream bitstream writer is designed to write bitstreams from audio Codecs whose payloads are entirely self-describing.

- C Object—`aeswriter.c`

NETWRITER

The network bitstream writer is capable of writing raw QBOX or video elementary streams across IP networks using either UDP or TCP sockets. Unlike RTP which has a control and data socket pair, the netwriter features only a single data socket.

2

Codec Objects Definitions

The MG3500 API is implemented using a set of Codec object types that provide a scalable interface. Codec Objects are comprised of Codec Proxy Objects, including System Control object as well as Codec Host-only Objects.

The following topics are covered in this chapter:

- “System Control Object Type Definition”
- “AVC Encoder (avcenc) Definition”
- “Audio Video Synchronizer (avsn) Definition”
- “Audio Video Output Compositor (avoc) Definition”
- “Read And Write Streamers (rs/ws) Definition”

SYSTEM CONTROL OBJECT TYPE DEFINITION

The system control object (`sysctl0`) is responsible for overall system configuration. This object is created and made available to the Mobiapp application immediately after booting the media engine. All other objects used in a system are created by users with reference to the `sysctl0` object. The Codec System Control object is a Class Factory.

See-> *Factory Method Pattern* for more information about the Class Factory type.

SYNTAX

The following structure shows the method used in defining the `sysctl0` object:

```
sysctl0:<classname>Create([args])
```

Where,

classname	is the class to which an object belong, for example LVPP,
args	is the arguments that specifies the object.

SYSTEM CONTROL CREATING READ/WRITE STREAMERS

The exceptions to the rule described above are the read and write streamers. The following syntax shows how they are created:

```
rs = sysctl0:createReadStreamer ([args])
ws = sysctl0:createWriteStreamer([args])
```

BROWSING OBJECT USING FOR LOOP

The system control object (`sysctl0`)is a global variable that is automatically created when Mobiapp is first booted.

EXAMPLE

```
print(sysctl0)
```
This prints the address of the table.

```
for k, v in pairs(sysctl0) do print(k,v) end
```
This example shows all the elements of the system control object.

ObjectHandles Table

One important table is the “objectHandles.” Every time the System Control object creates an object, a key is added with the object name.

EXAMPLE

```
sysctl0:avsnCreate{name = “avsn0”}
```

When `sysctl0` creates an `avsn` object, users can find a reference to that object by iterating through the objectHandles table. This is shown below:

```
for k, v in pairs(sysctl0.objectHandles) do
print(k,v)
end
```

The `for` loop shown below lists the methods of the `avsn` object.

```
for k, v in pairs(sysctl0.objectHandles.avsn0) do
print(k,v)
end
```

AVC Encoder (avcenc) Definition

The `avcenc` object enables encoding pre-processed video input frames. This object can receive an input from the `lvpp` or the `nvpp`. The `avcenc` object supports a wide variety of tools which allow the bitstreams generated to conform to the various profiles and levels specified by the H.264 standard. Additionally, `avcenc` allows the user to tune parameters to get the best possible video quality at the lowest bit-rate for a specific video application. The `avcenc` has a similar construct to ObjectHandles table.

See-> Section “ObjectHandles Table” for syntax and an example for setting up ObjectHandles tables.

Example

```
avcenc0 = sysctl0:avcencCreate({
                name = "avcenc0",
                ws = ws,
                maxwidth = HSIZE,
                maxheight = VSIZE16
                numreferenceframes = NUM_AVC_REFERENCES,
                outputbuffersize = ENC_OUTPUT_BUFFER
                });
```

The following defines the variables used in the above example:

name	is a name assigned by the user to a particular instance of this object, for example "`avcenc0.`"
ws	associates the encoder object with an instance of the writestreamer. The host notifies this writestreamer whenever there is a bitstream from this encoder instance that needs to be written to a file or streamed out.
maxwidth	specifies the maximum width of the video frames that will be input to the encoder. This number can be up to 1920 frames.
maxheight	specifies the maximum height (modulo 16) of the video frames that will be input to the encoder.
numreferenceframes	specifies the number of reference frames used by the encoder. This number depends on the encoding profile. A minimum of four reference frames must be used when encoding a baseline profile stream. Main and high profile streams need a larger number of reference frames.

`outputbuffersize`

specifies in bytes the amount of memory allocated for storing compressed bitstreams generated by the encoder. A number commensurate with the encoding bit-rate must be selected.

`maxwidth`, *`maxheight`*, and *`numreferenceframes`* determine the amount of memory allocated to store video frames for an instance of the encoder.

Note: Make sure sufficient amount of memory is allocated for the system to run smoothly.

The video encoder uses several methods which effectively perform the same function as the `Param()` method used by the other objects.

METHODS

`setVideoEncParam()`— general configuration.

`setVideoEncRegionParam()`— configures parameters specific to video regions.

`setVideoEncRateControlParam()`— configures parameters specific to encoder rate control.

Note: The parameters for the different functions are effected using a single activate command.

EXAMPLE 1

```
avcenc0:setVideoEncParam({
      param=Q_AVCENC_CMP_BITSTREAM_OUTPUT,
      value=BITSTREAM_OUTPUT});
avcenc0:activateVideoEncCfg();
```

See-> Section “Built-in Lua Libraries,” in Chapter 3," Lua Scripting Language," for more information about `avcenc` functions.

The sample Lua scripts distributed with the SDK release use a single function that encompasses all the initialization required for the video encoder.

EXAMPLE 2

The example below shows the function used to configure `avcenc` for 720p60 encoding:

```
avcenc_720p60(avcenc0, PROFILE, AVC_LEVEL, VBITRATE,
gopstruct, FRAME_STRUCTURE)
```

See-> The *MG3500 Codec Firmware API Reference Manual* for a list of the configuration parameters that may be used with the encoder.

Audio Video Synchronizer (avsn) Definition

The `avsn` object allows synchronizing audio and video. This object is required in preview and decode modes. If independent audio and video outputs are required, separate `avsn` objects may be created and bound to the appropriate objects upstream.

When bound to an `avoc` video or audio channel, the `avsn` must then be in the "running" state, otherwise failure to do so will lock the pipeline. Conversely, when unbound from an `avsn` video or audio channel, the `avsn` state must be set to the "idle" state.

A quick solution to resolve these types of issues is to use the `avsn:query()` method. Underflows are an indication of a problem downstream.

Audio Video Output Compositor (avoc) Definition

The `avoc` object provides the ability to present video and audio frames. This object can be used to display a single channel output, switch between channels or display a composed frame based on the video channel created and bound. Additionally, this object configures video output timing.

Note: Only video channel are guarantied to be glitch less.

The following steps list general guidelines for configuring the `avoc` object:

1. Configure the display window and audio parameters, if required.
2. Add and activate audio channels as required by the application.
3. Configure the timing for the display unit.

Read And Write Streamers (rs/ws) Definition

The read and write streamer objects are host-only objects. They read from and write to host memory when the Codec is decoding or encoding respectively.

Note: An instance of an encoder or decoder object must be associated with a `writestreamer` or `readstreamer` object.

The purpose of the streamers is to dispatch data to bitstream readers or writers which are registered in the system. It is, then, possible to pipe the data to multiple destinations such as a file and an RTP session at the same time.

Note: When no bitstream writers are opened, disabling the bitstream output from the `avc` encoder is important. Otherwise data will queue up without being consumed.

3

Lua Scripting Language

Lua is a powerful, fast, light-weight, and embeddable scripting language. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Mobiapp uses the Lua interpreter that has been extended with bindings to the C objects.

The following topics are covered in this chapter:

- "Introduction to Lua"
- "Learning Lua Scripting"
- "Lua Built-ins"

INTRODUCTION TO LUA

Lua is an object oriented, extensible, and embeddable scripting language. A Lua interpreter can be extended by bindings to the C objects. The Lua bindings allow for much or all of the MG3500 Codec operation to be scripted without the use of a compiler or tools other than a text editor. Lua is dynamically typed, runs by interpreting byte code for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

See-> The *Lua Programming Language Manual* for complete information about Lua scripting language.

LUA AND MOBIAPP

The Lua interpreter can be considered to be the "user interface" of the Mobiapp plugin. It is, however, possible to build other user interfaces that either access the C objects directly or that invoke the Lua interpreter through standard, non-interactive mechanisms.

Using Lua in a product is completely optional as all states and functions can be achieved through C code which interacts with the C objects. You may completely bypass Lua, however, only when you make use of C objects and write applications that make direct use of the object. However, keep in mind that the standard use of the reference application is through the Lua interpreter and shell, along with scripts that manipulate the C objects appropriately.

LUA ADVANTAGES

Lua is an superior prototyping language because

- No compilation is required; just change and run.
- No memory allocation (garbage collector).
- Built-in associative arrays and hash tables.
- Comprehensive back trace (no core dump to analyze).
- Fast and lightweight scripting language.

Learning Lua Scripting

This section describes the semantics of the Lua scripting language, including Lua variables, functions, control structure, tables, and objects.

Variables and functions

In Lua, any string that is not a reserved keyword is considered to be an identifier. When an identifier is not initialized, it is set by default to “null.”

```
-- prints "Type of myVar is null"
print("Type of myVar is ", type(myVar))
myVar = "Hello world!"

-- prints "Type of myVar is string"
print("Type of myVar is ", type(myVar))
myVar = true

-- prints "Type of myVar is boolean"
print("Type of myVar is ", type(myVar))
myVar = 123.99

-- prints "Type of myVar is 123.99"
print("Type of myVar is ", type(myVar))

-- prints "Type print is function"
print("Type print is ", type(print))
```

The print function takes positional parameters and prints them regardless of their type. The `printf` function (not built in) can be created as follow.

```
function printf(...)
print(string.format(unpack(arg)))
end

foo=printf
printf("myVar is set to %f and the type is %s\n", myVar,
type(myVar))

foo("hello world\n")
```

Functions are first class values which means that they can be stored in variables as illustrated above with the “foo” identifier which contains the address of `printf`.

Functions can return more than one value as shown below:

```
--[[ Formal parameters are declared between ()
   Functions can return one or more return value
--]]

function func(arg1, arg2, arg3)
print(arg1, arg2, arg3)
return arg1, arg2, arg3
end

v1, v2, v3 = func(1,2,3)
print(v1,v2,v3) -- prints 1  2  3
```

Control Structures

The most common control structures:

```
-- Generic form of an if statement

if cond1 and (cond2 or cond3) then
elseif cond4 then
else
end

-- Generic for loop
for i=min,max do
...
end
-- Generic while loop
while(cond) do
...
end
```

Tables

Tables are one of the most important constructs in Lua. Everything in Lua is part of a table.

For example, the global name space (called _G), where the variables and functions listed above are stored, is a table.

The “constructor expression” for a table is {}.

```
a = {}; print(type(a)) -- prints 'table'
```

Use the following to see the list for an in the global namespace:

```
for i, v in pairs(_G) print(i, v)
```

Where *i* is an identifier, and *v* its a value.

Tables can be used as "standard arrays," "associative arrays," or a combination of both.

```
array = {1,2,3,4}
assoc = (key1 = 1, k2 = "string", key3 = true)
comb = {1,2, key1 = "string", 4)
```

There are two main table iterators in Lua: "ipair*"* and "pairs." Ipairs iterates through the indexed elements of a table while pairs iterates through all elements.

```
> for k, v in ipairs(comb) do print(k,v) end
1       1
2       2
3       4
> for k, v in pairs(comb) do print(k,v) end
1       1
2       2
3       4
key1    string
```

The first index is by convention 1. Hence to print the first element, you may type `comb[1]` and to print the first key `comb.key` or `comb["key"].`

Note: Tables are an important concepts since not only they are used to implement objects but they pass actual parameters to the MG3500 API.

Objects

Objects are in their simplest form tables for which some keys are references to functions.

```
object = {self = 1}
function object:method1(options)
   self.member = options
end

function object.method2(self, options)
   self.member = options
end
```

In this case, the object is a "singleton."

See-> The *Lua Programming Language Manual* for more advanced topics on classes and objects.

There is technically no difference between the first and the second methods: The use of colon, ":" is a more elegant way to make the self (the reference to the object table in this particular case) implicit.

Throughout the code script example, you will often see

```
sysctl0:configure({param=Q_SYS_CFG_VIN0_CONTROL,
value=0x00048249});
```

where

`sysctl0`	is the reference to an object,
configure	is one of its method,
{…}	is a table containing its actual arguments.

```
sysctl0.configure(sysctl0, {param=PARAMNAME, value=VALUE});
```

is equivalent to the first form.

Mobilygen is using tables as a convention to pass parameters. This has the advantage of removing the constraint of using positional parameters, make this call equivalent to the first one.

```
sysctl0:configure({value=0x00048249},
param=Q_SYS_CFG_VIN0_CONTROL});
```

Note: Lua does not require statements to end with a semi-colon, "; ".

Lua Built-ins

Built-in Lua Libraries

Included with Lua comes is a set of built in libraries, including the following most important ones:

1. IO—Input/Output facilities.
2. OS—Basic OS facilities.
3. Table—Table manipulation.
4. String—String operators and manipulation.
5. Math—Math library.

See-> The *Lua 5.1 Reference Manual* for detailed information about the Lua scripting language.

Built-in Lua Scripts

The **Mobiapp** application includes several Lua scripts that are executed automatically at the startup. The Lua scripts are located in <host>/luascripts/builtin and are executed according to their numerical sort order given that each script is named with a leading number such as 40board, 60libavcenc, etc.

See-> Chapter 4," From Lua to C," to transition from the Lua scripting language to writing your own C codes.

See-> Chapter 5," Writing Scripts," to learn how to learn about the steps for writing a Lua script.

4

From Lua to C

Lua is a lightweight script language, an extension to an existing program in C that allows configuring programs and modifying run-time behavior. Being an extension language, Lua can only be embedded in a host client, called the host. This host program can then invoke functions to execute a piece of the Lua code, read and write Lua variables, and register C functions that are called upon by Lua code. By using the C functions, Lua can be enhanced to cope with a wide range of different domains, thereby creating customized programming languages that share a syntactical framework.

The following topics are covered in this chapter:

- "Overview of Lua to C"
- "The Lua Application Program Interface"
- "Mobiapp Interaction via C Programs"
- "Lua Objects Bindings"
- "Lua to C Sample Script"
- "Built-in Scripts"

Overview of Lua to C

Lua is the ideal scripting language that enables rapidly prototyping of your Mobiapp system since it can be dynamically typed, interpreted from bytecodes, and is able to manage memory automatically with garbage collection.

Alternatively, you may use C programs to call Mobiapp by translating Lua calls to C or C++. Understanding the Lua Application Program Interface is important since it will provide the information necessary to transition from Lua to C. It also explains how Lua interfaces with the Mobiapp plugin.

The Lua Application Program Interface

This section describes the C API for Lua that are basically a set of C functions available to the host program that communicates with Lua.

Topics covered in this section are

- "Lua Stacks"
- "Lua Chunks"
- "Lua Macros"
- "Lua States"

Lua Stacks

Lua is intended to be embedded into other applications, which results in providing a robust, easy to use "C APIs." The Lua API makes extensive use of a global "stack" which is used to pass parameters to and from Lua and C functions. Lua, then, provides functions to push and pop most simple C data types (integers, floats, etc.) to and from the stack, as well as functions for manipulating tables through the stack. Arranging data between C and Lua functions is also done using the Lua stack.

To call a Lua function, arguments are pushed onto the stack, and then the "lua_call" is used to call the actual function. When writing C functions that can be to be directly called from Lua, the arguments are popped from the stack.

The Lua stack is somewhat different from a traditional stack in that it can be indexed directly. Negative indices indicate offsets from the top of the stack (for example, -1 is the last element), while positive indices indicate offsets from the bottom.

Lua Chunks

A Lua “Chunk” is basically a sequence of statements that refers to each piece of the code that Lua executes. An example is a file or a single line in an interactive mode.

A chunk may be as simple as a single statement, such as in the "hello world" example, or it may be composed of a mix of statements and function definitions, which are actually assignments such as the factorial example. A chunk may be as large as you wish. Because Lua is used also as a data-description language, chunks with several megabytes are not uncommon.

Lua Macros

All API functions and related types and constants are declared in the header file `lua.h`.

Reference to the term "function" can be considered as a “Lua macro facility” in the API instead. The macro facility is similar to the C pre-processor although it works on an already pre-digested token stream and is not a separate program through which Lua code is passed. All such macros use each of its arguments exactly once (except for the first argument, which is always a Lua state), and so do not result in hidden side-effects.

The Lua macros have been defined by Mobilygen to simplify binding creation. Their definition can be found in the `luadef.h` header file in the Mobiapp plugin source tree.

Lua States

The Lua library does not contain any global variables. The entire state of the Lua interpreter (global variables, stack, etc.) is stored in a dynamically allocated structure of type “`lua_State`.” A pointer to this state must be passed as the first argument to every function in the library, except to `lua_open`, which in turn creates a Lua state from scratch.

CALLING API FUNCTIONS

Before calling any API function, first create a state by calling `lua_open`

```
lua_State *lua_open (void);
```

RELEASING LUA STATE

To release a state created with `lua_open`, call `lua_close`

```
void lua_close (lua_State *L);
```

This function destroys all objects in the given Lua state by calling the corresponding garbage-collection methods, if any, while freeing up all dynamic memory used by that state. Calling this function on many platforms may not be needed since all resources are naturally released when the host program ends. However, programs that take a long time to run, such as a daemon or a web server, might need to release states as soon as they are not needed to avoid growing too large.

MOBIAPP INTERACTION VIA C PROGRAMS

Lua is a scripting language that enables full testing of the entire codec APIs without the use of a C compiler or tools.With Lua only an editor is necessary.

Topics covered in this section are

- "Lua Interpreter Plugin"
- "Mobiapp Lua Interface Files"

LUA INTERPRETER PLUGIN

The mobiapp plugin is not sufficient to build an application since an application framework or user interface must be added first. Mobilygen supplies a second plugin called "`ui_lua`" which implements a Lua interpreter with bindings to the mobiapp plugin, allowing the codec APIs to be fully scriptable. The name of the Lua plugin `ui_lua` refers to the Lua interpreter as the user interface for Mobiapp.

The `ui_lua` plugin is fully removable and optional from mgapp simply skipping the loading of the plugin.

MOBIAPP LUA INTERFACE FILES

All types of the MG3500 Codec objects, such as sysctl and avcenc, are implemented using three different types of files:

- C program files—For example, `avcencapi.c`.
- Header fils—For example, `avcenc.h`.
- Lua binding files—For example, `avcenclua.c`.

See-> Section "Overview of Mobiapp C Objects," in Chapter 1," Overview of Programming MG3500 Codec."

See-> Chapter 2," Codec Objects Definitions," for detailed information on the Codec objects.

Lua Objects Bindings

The `ui_lua` plugin provides a set of Lua bindings that connect the C object APIs to Lua functions so that each C API is made available as a Lua function call. Since the Lua bindings are object-oriented, the bindings create and manage Lua objects. Each Lua object contains in its private data a pointer to an underlying C object which is hidden from a scripting perspective. The Lua essentially "wraps" the C object and makes that C object directly scriptable.

The purpose of the Lua binding file, then, is to register C functions with Lua and translate arguments from the Lua domain to the C domain via the Lua stack.

Topics covered in this section are

- "C Functions Definition"
- "Registering C Functions"

C Functions Definition

Lua can be extended with functions written in C. These functions must be of type `lua_CFunction`, defined as

```
typedef int (*lua_CFunction) (lua_State *L);
```

A C function receives a Lua state and returns an integer, which is the number of values it wants to return to Lua.

Registering C Functions

The following macro shows how registering a C function to Lua is done:

```
#define lua_register(L,n,f) \
               (lua_pushstring(L, n), \
                lua_pushcfunction(L, f), \
                lua_settable(L, LUA_GLOBALSINDEX))
     /* lua_State *L;    */
     /* const char *n;   */
     /* lua_CFunction f; */
```

In this example, the function receives the name it will have in Lua and a pointer to that function. Thus, the C function foo above may be registered in Lua as average by calling

```
lua_register(L, "average", foo);
```

The signature of `AVCEncoderAPI_Create` is identical to the Lua method and any reference to Lua objects is translated to their corresponding C object using the `CommonAPI_Gelua_CFunctiontCodecObjectId()API`.

Lua to C Sample Script

Typically, the C implementation simply translates the C API to an opcode or payload which is then sent to the MG3500 Codec via the QHAL driver. This is illustrated in the example that follows:

```
int AVCEncoderAPI_BindVideoInput(AVCENC_HANDLE h, void * vpp)
{
    COMMAND cmd;
int rval;
    AVCENC_HANDLE_PRIV *this = (AVCENC_HANDLE_PRIV *)h;
    STRUCT_Q_AVCENC_CMD_BIND_VIDEO_INPUT *pArgs =
               (STRUCT_Q_AVCENC_CMD_BIND_VIDEO_INPUT
*)cmd.arguments;

int ctrlObjID = CommonAPI_GetCodecObjectId(vpp);

    CommonAPI_ClearCommand(&cmd);

    // build command
    cmd.controlObjectId = this->cd.id;
    cmd.opcode = Q_AVCENC_CMD_BIND_VIDEO_INPUT;

    pArgs->ctrlObjID = ctrlObjID;

    // Send request to the main thread
    rval = MWare_BlockingSendCommand(this->hmw, &cmd);

    return rval;
}
```

The Lua binding for the function listed is the example shown below.

```
/// \brief AVCEncoderAPI.bindVideoInput
///
/// usage: filename
///
/// AVCEncoderAPI.bindVideoInput
///
int cliAVCEncoderAPI_BindVideoInput(lua_State *L)
{
    AVCENC_HANDLE h;
    void * src;

    LUA_GET_SELF(h);
    src = getHandle(L, "src");

    AVCEncoderAPI_BindVideoInput(h, src);

    return 0;
}
```

Because C functions (e.g. `AVCEncoderAPI_BindVideoInput`) can not be called directly, a stub is defined. By convention, stub functions start with **cli**<*function name*> as shown in the example above. Stubs are registered with Lua at creation time.

Built-in Scripts

The `ui_lua` plugin contains several "built-in" Lua scripts. These scripts are included in the plugin at compile-time and are automatically executed when the scripts are loaded. The built-in scripts include the board abstraction library that are used to control video input and output as well as various utilities to help debug and initialize the board.

These higher-level functions that enable encapsulating complex Codec configuration routines that are known as built-ins are written exclusively in Lua.

Included in Mobiapp at compile time as Lua chunks, these higher-level functions are compiled and executed at startup time by the Lua interpreter and then embedded in Mobiapp. At the present time, translating the built-in libraries to C and so making them accessible from a C program is necessary.

Built-in Scripts Location

Built-ins can be found under the following location:

```
<install dir>/merlinsw/host/luascripts/builtin
```

See-> Chapter 5," Writing Scripts." for learning about the basic steps for writing scripts.

See-> Chapter 7," Sample Scripts." for sample scripts that illustrate the fundamental parts required to configure and operate the MG3500 Codec.

5

Writing Scripts

Writing scripts requires understanding some general rules to help the user create their own scripts.

The following topics are covered in this chapter:

- “Steps for Writing Scripts”
- “Prepare to Write Scripts”
- “Initialize the Board”
- “Set up Objects”
- “Bind Objects”
- “Start Objects”

Steps for Writing Scripts

Writing a script consists of the following steps:

1. "Prepare to Write Scripts"
2. "Initialize the Board"
3. "Set up Objects"
4. "Set up Objects"
5. "Bind Objects"
6. "Start Objects"

Prepare to Write Scripts

Objects follow a producer-consumer model. Every object has an input port and an output port and accepts an input type to produce an output. An object may be a producer and a consumer depending on where it is located in the data path. The relationship between a producer and consumer can be one-to-one (`avcenc`), one-to-many (`lvpp`), or many-to- one (`avsn`).

Example of a one-to-one model would be the producer object `lvpp` bound to consumer object avcenc. Example of a one-to-many model would be the producer object avcenc bound to consumer objects `hvout` and `avoc`. This also shows that the avcenc object is a consumer and a producer.

Initialize the Board

Board I/Os configuration is encapsulated into the board object. The board object is defined as part of the built-in scripts that are compiled in Mobiapp, called "`40board.lua`."

This object detects the board types, configures peripherals, sets up clocks, and programs registers that control audio and video input timing and other objects.

The implementation of board objects is specific to the hardware that is used, in this case the MG3500 Evaluation Platform as the target hardware. Users may refer to the `40board.lua` file to create their own.

Configuring System Parameters Example

```
board.setVideoOutput({port=0, format='720p',mode='hdmi'});
board.setVideoInput({port=0, format='720p',mode='hdmi'});
```

The above methods configure the “`hdmi`” receiver and transmitter on the MG3500 EVP to format and output the video input and output to 720p resolution.

Caution! The board object is a singleton which is available in the global Lua namespace. Make note of the usage of “a.” as opposed to “a:” between board and the method being called, for example `board.setVideoOutput versus avcenc0:query()`.

Tip: To avoid recompiling Mobiapp every time a change is made to the board object, you may specify the new board object on the Mobiapp command line. When Mobiapp starts, it will execute the default board detection and map the I/O configuration functions to the entry points in the board object. At this point, I/Os are not configured.

Mobiapp will then load and execute the board file specified on the command line. Board detection will be performed one more time and the new code will be mapped to the entries.

```
mobiapp -f myboard.lua -f myscript.lua
```

Caution! This method will still work even if an error message displays when a new board type introduced as the first board is detected and fails, which will result in Mobiapp aborting any execution. Notice that if a new board file is added to the Linux distribution (`dist/linux/mobilygen/drivers/boards`), a corresponding entry needs to be added to the board object.

Set up Objects

After the board is initialized and the system is configured, for the next step objects need to be created.

The following is an example of how an object is created.

SETTING UP OBJECTS EXAMPLE

```
lvpp0 = sysctl0:lvppCreate{
                            name            ="lvpp0",
                            slot            =0,
                            port            =0,
                            maxWidth        =1280,
                            maxHeight       =720,
                            numVidFrame     =11,
                            numPMEDSFrame   =11,
                            maxPMEDSFactor  =2
                           }
```

See-> Chapter 2," Codec Objects Definitions," for descriptions of the two types of objects provided by Codec, Codec Proxy objects and Host-only objects.

See-> Chapter 6," Setting up Codec Objects," for detailed instructions for setting up the objects.

BIND OBJECTS

The preview( ) function below describes the binding of objects and the order in which they must be started.

```
function preview()
    avsn0:bindVideoInput({src=lvpp0});
    avsn0:start();
    lvpp0:start();
end
```

To simplify the configuration process, functions common to an object are compiled into libraries. These libraries are provided with the reference code and must be made available to the Lua environment to execute the scripts.

LOADING LIBRARIES EXAMPLE

The following sample script will show several examples of libraries being loaded. Notice that the MG3500 uses single-buffered and double-buffered configuration parameters.

In firmware single-buffered parameters use a "CFG" tag and double-buffered parameters use a "CMP" tag.

```
Q_SYS_CFG_VIN0_CONTROL single buffered configuration parameter
Q_LVPP_CMP_START_VLINE double-buffered configuration parameter
```

Both types of parameters may be issued at any time during system operation but double-buffered parameters must be activated before they become effective. In the Lua space, each object has an activate method associated with that object that conforms the following:

```
<classname>:activateCfg()
```

Note: An `avsn` object must be started before the `lvpp` object.

Start Objects

After objects are created, configured, and bounded, they must be started. A data path is initiated by starting upstream objects first and working downstream. For example, in "preview" the `lvpp` object is bound to an `avsn` object.

Starting Objects Example

This script starts some objects and uses functions to start and stop other objects as required. The statements below show the start of continuously running objects.

```
avoc0:start();
ws0:start()
rs0:start();
```

Functions

After these objects are started, functions may be used to create and destroy data paths that are required for specific tasks. The functions that are used in the script are shown below. Functions that start a task bind objects and start them in the order required, for example "preview." Others that stop a task, for example `estop()`, shows the sequence of steps that stop objects before they are unbound.

- Preview
- Stop
- "Record to file"

- “Estop”
- “Play from File”
- “Stop Decoder ”

PREVIEW

```
function preview()
        avsn0:bindVideoInput({src=lvpp0});
        avsn0:start();
        lvpp0:start();
end
```

STOP

```
function stop()
        lvpp0:flush();
        avsn0:stop();
        avsn0:unbindVideoInput();
end
```

RECORD TO FILE

```
filenameh4v = "";
filenameqbox = "";

function record(fileName)
  avcenc0:bindVideoInput({src=lvpp0});
  filenameh4v = string.format("%s.h4v", fileName);
  filenameqbox = string.format("%s.qbox", fileName);
  ws0:open({type="h4v”, file=filenameh4v, sid =
     {videoStreamId}});
  ws0:open({type="file", file=filenameqbox, sid =
     {videoStreamId}});
  avcenc0:record();
  lvpp0:start();
 end
```

ESTOP

```
function estop()
    lvpp0:flush();
    waitForEvents({objid=avcenc0, timeout=1000000, any=0,
    events={Q_AVCENC_EV_BITSTREAM_FLUSHED}});
    avcenc0:waitForLastBlock();
    ws0:close({type="h4v", file=filenameh4v});
    ws0:close({type="file", file=filenameqbox});
    avcenc0:unbindVideoInput();
end
```

PLAY FROM FILE

```
decodeFile = "";

function play(FileName)
    decodeFile = FileName;
    avsn0:bindVideoInput({src=avdec0});
    avoc0:setVideoChannelSource({ch=vch0, src=avsn0});
    avoc0:activateVideoChannelCfg();
    startTime = os.time();
    avsn0:start();
    avdec0:start();
    bsr0 = rs0:open({type="qbox", file=FileName,
    srcId={videoStreamId, audioStreamId});
    waitForEvents({objid=avdec0, timeout=1000000, any=0,
    events={Q_AVDEC_EV_VIDEO_END_OF_STREAM}});
    endTime = os.time();
    totalTime = os.difftime(endTime, startTime);
print(string.format("Decode Time = %d seconds\n",
totalTime));
end
```

STOP DECODER

```
function dstop()
    rs0:close({type="qbox", file=decodeFile});
    avdec0:stop();
    avsn0:stop();
waitForEvents({objid=avdec0, timeout=1000000, any=0,
events={Q_AVDEC_EV_VIDEO_END_OF_STREAM}});
    avsn0:unbindVideoInput();
    avsn0:activateCfg();
end
```

6

Setting up Codec Objects

Writing scripts requires initialing the board, configuring the system, and then setting up the Codec objects.

The following topics are covered in this chapter:

OBJECTS CREATION

The system control object `sysctl0` enables the creation and configuration of the Codec objects. This object is created and made available to the application after booting the media engine.

CREATING AN OBJECT SYNTAX

The following syntax shows the construction methods for creating an object:

```
systl0:<classname>Create([arguments])
```

CREATING READ/WRITE STREAMERS SYNTAX

The exceptions to the rule described above are the read and write streamers. The following syntax:

```
rs = sysctl0:createReadStreamer ([arguments])
ws = sysctl0:createWriteStreamer([arguments])
```

Objects Configuration

Once objects are created, they need to be configured and then be activated to support the functionality required of them.

The `setParam()` method is used for configuration. Use of the `setParam()` method is shown below:

```
<object ref.>:setParam{
                    param=<cfg_param>,
                    value=<value>
                   }
```

Configuring a Single Object Syntax

In addition to creating objects, `sysctl` enables system configuration, video inputs, clocks for a variety of hardware blocks, outputs, etc. The `configure()` method of the `sysctl0` objects is used for configuring objects. The syntax is shown below:

```
sysctl0:configure {
               param = Q_SYS_CFG_PLL3_CLOCK_HZ,
               vale = 2048000
               }
```

Note: Notice the omission of the parenthesis. This is a valid Lua construct when using a table to pass on actual parameter using a key or value pair. In the above example, PLL3 is programmed to generate a 2.048MHz clock.

See-> The *MG3500 Codec Firmware API Reference Manual* for a list of parameters that can be used with this method.

Configuring Multiple Objects Syntax: configure() Method

Most methods have a single purpose, for example `configure()` writes a “value” to “param.” The following example configures two control registers, VIN0 and VIN1.

```
sysctl0:configure({
                param = Q_SYS_CFG_VIN0_CONTROL,
                value = 0x00048249
                });
sysctl0:configure({
               param = Q_SYS_CFG_VIN1_CONTROL,
               value = 0x00000000
               });
```

Configuring Multiple Objects Syntax: setVPortMode () Method

Other functions take multiple arguments. The `setVPortMode()` method of the `LVPP` class sets up the video input ports and internal muxers to route video as required by the system. An example of this method used in a multi-channel system is shown below. Arguments “syncmode” and “syncheight” are not required in systems that have a single video input to a VIP (video input processor). This is shown in the following example:

```
sysctl0:setVPortMode{
                    port = 0,
                    live = 1,
                    interlaced = 1,
                    bottomfieldfirst = 0,
                    videocontrol = 0x80101080,
                    interleave = 1,
                    syncmode = Q_SYS_CFP_VPORT_SYNC_MODE_MOBI,
                    syncheight = 525
                     }
```

See-> The *MG3500 Codec Firmware API Reference Manual* for more information on the system control object.

The following are descriptions of the parameters in the above example:

port

Refers to the VIP module used. Valid arguments are 0 and 1. Port 0 can only take in a live input (abstracted by an LVPP objects). Port 1 can take in a live OR non-live video input (abstracted by an `nvpp` object). Non-live video ports can be used for scaling (input frames read from memory) and can operate faster than real time.

If the system requires just one live video input it MUST use port 0. Only port 1 can be used as non-live input so a system that encodes video frames stored in memory, for example using `hvcap(),` initializing port 1 is sufficient. An example of non-live port initialization is shown below.:

```
sysctl0:setVPortMode{
                port=1,
                live=0
                }
```

live	Indicates the nature of the video input to the video input module. Valid arguments are 0 and 1. A value of 1 indicates live video input and 0 indicates non-live video input. When connected to live input, a VIN can operate only at the video input clock speed. A VIP consists of a video input module and video processing module which operate on independent clocks. For example, when connected to an NTSC source VIN will be run at 27Mhz. VIN can operate up to frequencies of 125MHz.
interlaced	when set to 1 indicates that the video source uses interlaced scans.
bottomfieldfirst	when set to 1 indicates that the video source sends the bottom field first.
videocontrol	The register controls several parameters including sync source and width of video input, 8-bit and 16-bit. **See->** The *MG3500 Codec Firmware API Reference Manual* for more information on `Q_SYS_VPORT_MODE`.
interleave	This indicates the number of interleaved channels in the video input. Multi-channel designs can have up to four channels (byte) interleaved in a single input video stream.
syncmode	
syncheight	

Objects Activation

Most configuration parameters are double-buffered. This means that after the appropriate values have been passed on to the parameters, an “activate” command must be issued to the particular instance of the object so that these values take effect. This is done by using the `activateCfg()`method, as shown below:

```
<object ref>:activateCfg()
```

Setting up Live Video Pre-Processor (lvpp) Object

The live video pre-processor `lvpp` object enables capturing and pre-processing of live input video data. This object handles cropping, scaling, filtering, and all other functions associated with the video input module.

Creating the lvpp Object

The `lvpp` object is created with reference to the system control object. The following example shows how to create `lvpp`:

```
lvpp0 = sysctl0:lvppCreate{
                 name="lvpp”,
                 slot=0,
                 port=0,
                 maxWidth=720,
                 maxHeight=480,
                 maxPMEDSFactor=2,
                 numVidFrame=6
                         }
```

The `lvpp` object allocates memory for captured frames and for setting up the `lvpp` for pre-processing. In Lua to C conversion, all arguments except name must be passed on to the C API layer.

See-> The *MG3500 Codec Firmware API Reference Manual* for information refer `Q_SYS_CMD_LVPP_CREATE`.

name	Is an arbitrary name assigned to this object, used for control in Lua space. As mentioned earlier, a table indexed by object name can be accessed to retrieve object references. **Note:** Same names can be re-used however, only the last reference of the object will be available from the sysctl0.objectHandles table.
slot	relates to the number of interleaved channels in a video input stream. In a multichannel system each interleaved channel will have a different slot number as illustrated in Figure 1 Slot Allocation1.

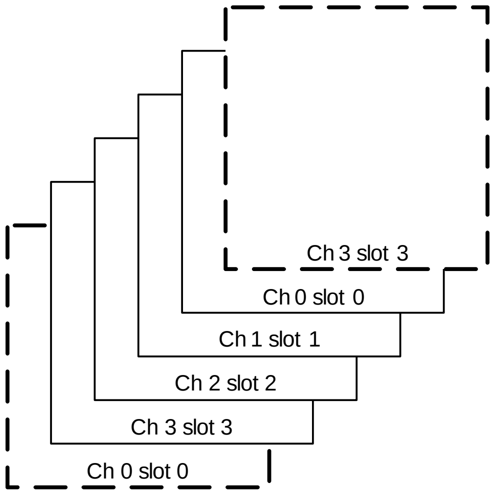

Figure 1 **Slot Allocation**

port	Specifies the video input port to be used. Refer to section MG3500 Codec System Control (`sysctl`) Object for more information.
maxwidth	Specifies the maximum width of the input frames. Memory is allocated appropriately.
maxheight	Specifies the maximum height of the input frames. Memory is allocated appropriately.
maxPMEDSFactor	Specifies the maximum Pre-Motion Estimation (PME) Decimation factor. **See->** The *MG3500 Codec Firmware API Reference Manual.*
numvidframe	Number of buffers to allocate for a particular instance of the `lvpp` object. Encoding profile, core clock and GOP structure will have an impact on the numbers of buffers required.

Because of the complexity of `lvpp`, core configuration functions have been defined in the `libvpp.lua`.

```
Setup_VPP(lvpp0, HSIZE, VSIZE, 1, 0, 2, 4)
```

See-> Section, “Built-in Scripts,” in Chapter 4," From Lua to C."

After an `lvpp` object is created, it must then be configured correctly to process the incoming video. This includes cropping crop(), scaling size(), temporal filtering filterTemporalMotion(), etc.

See-> The *MG3500 Codec Firmware API Reference Manual* for detailed descriptions of the functions supported by the `lvpp`, and for a list of the configuration parameters.

Configuring and Activating lvpp Object

An example of configuring an activating the `lvpp` object is shown below.

```
lvpp0:setParam{
            param=Q_LVPP_CMP_FIELD_OFFSET,
            value=313
            }
lvpp0:setParam{
            param=Q_LVPP_CMP_START_VLINE,
            value=24
            }
lvpp0:setParam{
            param=Q_LVPP_CMP_START_HPIXEL,
            value=0
            }
lvpp0:setParam{
            param=Q_LVPP_CMP_PIXEL_AR_X,
            value=16
            }
lvpp0:setParam{
            param=Q_LVPP_CMP_PIXEL_AR_Y,
            value=15
            }
lvpp0:setParam{
            param=Q_LVPP_CMP_DEST_PROG,
            value=0
            }
lvpp0:activateCfg()
```

Setting up Non-Real Time Video Pre-Processor (nvpp) Object

The nvpp object is similar to the nvpp and uses a similar API. The nvpp object requires the use of a VIP module (always port 1).

Creating nvpp Object

When port 1 is used with an nvpp object, a live video input cannot be connected to this object. Instead, input frames to this object are read from memory.

```
sysctl0:setVPortMode{
                port = 1,
                live = 0
                    }
nvpp0 = sysctl0:nvppCreate {
                name = "nvpp0",
                maxWidth = 720,
                maxHeight = 480,
                maxPMEDSFactor = 1
                      }
```

Configuring nvpp Object

The functions and configuration parameters associated with the nvpp are similar to those used with the lvpp.

See-> The *MG3500 Codec Firmware API Reference Manual* for a list of configuration parameters.

Setting up Stereo Audio Input (sain) Object

The `sain` object is responsible for configuring the audio port for audio capture (oversampling, sampling rate etc.). Like the video port the audio input module also has two physical interfaces which are abstracted as ports in software. Unlike the video ports, the audio ports are identical. Audio signals connected to the AUD0 interface are mapped to port 0 and signals connected to AUD1 are mapped to port 1.

Creating sain Object

```
sain0 = sysctl0:sainCreate{
                      name="sain0",
                      port=0
                      }

Sample configuration and activation of the sain object is
shown below.

sain0:setParam{
            param=Q_SAIN_CMP_AUD_SERIAL_MODE,
            value=Q_SAIN_CFP_AUD_SERIAL_MODE_I2S
            }
sain0:setParam{
            param=Q_SAIN_CMP_AUD_MASTER_CLOCK,
            value=Q_SAIN_CFP_AUD_MASTER_CLOCK_256FS
}
sain0:setParam{
            param=Q_SAIN_CMP_AUD_SAMPLE_RATE,
            value=SAMPLERATE
            }
sain0:setParam{
            param=Q_SAIN_CMP_AUD_SAMPLE_SIZE,
            value=16
            }
sain0:setParam{
            param=Q_SAIN_CMP_AUD_CODEC,
            value=SAIN_AUDIO_CODEC
            }
sain0:activateCfg()
```

When creating the `sain` object, it is important to associate the object to the type of audio encoding used in the system. The audio encoding type will determine the number of audio samples in a frame.

Setting up Audio Video Decoder (avdec) Object

The `avdec` object allows decoding a video-only, audio-only, or an audio-video stream. This is created as a single object that is initialized with the audio and video decoder type.

The `avdec` object controls all parameters associated with the H.264, MPEG2, MJPEG video decoder, audio decoder, and the de-multiplexer.

Creating avdec Object

The `avdec` object controls all parameters associated with the H.264, MPEG2, MJPEG video decoder, audio decoder, and the de-multiplexer.

Creation of an AV decoder object is shown below. All arguments except name are passed on to the firmware API layer.

```
avdec0 = sysctl0:avdecCreate({
                   name="avdec0",
                   atype=Q_AVDEC_CFP_AUDIO_CODEC_QMA,
                   vtype=Q_AVDEC_CFP_VIDEO_CODEC_AVC,
                   audioBufferSize=100000,
                   videoBufferSize=8000000,
                   maxWidth=1280,
                   maxHeight=720
                          });
```

Configuring avdec Object

The `avdec` object also has a configuration method. The following example shows the configuration of the audio sample rate and sample size for the output.

```
avdec0:setParam({
            param = Q_AVDEC_CMP_AUDIO_SAMPLE_RATE,
            value=24000});
avdec0:setParam({
            param = Q_AVDEC_CMP_AUDIO_SAMPLE_SIZE,
            value=16});
avdec0:activateCfg();
```

See-> The *MG3500 Codec Firmware API Reference Manual* for more information on the AV decoder object.

Setting up Audio Video Encoder (avenc) Object

The `avenc` object controls all parameters associated with the encoder. Creation of an avcenc object is shown below. All arguments other than “name” and “ws” are passed on to the firmware API layer. Creation of an encoder object must be associated with a write streamer even if it does not write to a file.

Creating AVENC Object

The `avenc` object controls all parameters associated with the encoder. Creation of an avcenc object is shown below. All arguments other than “name” and “ws” are passed on to the firmware API layer. Creation of an encoder object must be associated with a write streamer even if it does not write to a file.

```
avcenc0 = sysctl0:avcencCreate({
                            name = “avcenc0”,
                            ws = ws0,
                            maxwidth = 1280,
                            mightiest = 720,
                            numreferenceframes = 10
                            });
```

Configuring avcenc Object

The following function sets up an instance of the encoder, avcenc`0`, to encode a high-profile stream at 9Mbps using an IBBrBP GOP structure. This function and others used to configure the encoder can be found in libavcenc.lua.

- avcenc_720p60—`avcenc`, profile, level, bitrate_qp, gop_structure, picture_coding.
- avcenc_720p60—`avcenc0`, "high", 40, 9000000, `Q_AVCENC_CFP_GOP_STRUCTURE_IBBRBP`, 0

The bitstreams generated by the encoder in the MG3500 use a native format called the “qbox.” In this format, packets containing compressed video use a “streamid = 2,” where as packets containing compressed audio use a “streamed = 1.”

The 60-bit `libavcenc.lua` library contains functions used by the AVC encoder object.

The following commands assign and activate the ID's that should be used with the stream generated by the encoder. In this case the stream will contain video only.

```
videoStreamId = 2;

avcenc0:setStreamID({streamid=videoStreamId});
avcenc0:activateCfg();
```

See-> The *MG3500 Codec Firmware API Reference Manual* for information on qbox and on the AVC encoder object.

Setting up Audio Video Synchronizer (avsn) Object

The `avsn` object is used to synchronize audio and video. Audio is master and video frames are dropped or repeated to achieve synchronization.

Creating avsn Object

Creation of a simple object is shown below:

```
avsn0 = sysctl0:avsnCreate({name="avsn0”});
```

See-> The *MG3500 Codec Firmware API Reference Manual* for information about the AVSN object.

Setting up Audio Video Output Compositor (avoc) Object

The `avoc` object controls all parameters associated with the compositor that handles the display of video and audio frames.

```
avoc0 = sysctl0:avocCreate({name="avoc0"});
```

The following example shows how `avoc` is configured for 720P 60 output.

```
avoc0:setOutput720p60();

avoc0:setParam({param=Q_AVOC_CMP_VID_0_DISPLAY_WIDTH,
            value=1280});
avoc0:setParam({param=Q_AVOC_CMP_VID_0_DISPLAY_HEIGHT,
            value=720});

avoc0:activateCfg();
```

See-> The *MG3500 Codec Firmware API Reference Manual* for information on `avoc.`

Creating avoc Object

The `avoc` object controls all parameters associated with the compositor that handles the display of video and audio frames.

```
avoc0 = sysctl0:avocCreate({name="avoc0"});
```

Configuring avoc Object

The following configures the `avoc` object for 720P 60 output:

```
avoc0:setOutput720p60();
avoc0:setParam(
      {param=Q_AVOC_CMP_VID_0_DISPLAY_WIDTH,value=1280});
avoc0:setParam(
      {param=Q_AVOC_CMP_VID_0_DISPLAY_HEIGHT, value=720});
avoc0:activateCfg();
```

See-> The *MG3500 Codec Firmware API Reference Manual* for information about the `avoc` object.

Setting up Read/Write Streamer (rs/ws) Objects

The read and write streamer objects are host-only objects. They read from and write to host memory when the Codec is decoding or encoding respectively. An instance of an encoder or decoder object "must" be associated with a writestreamer/readstreamer.

```
rs0 = sysctl0:createReadStreamer({name="rs0"});

ws0 = sysctl0:createWriteStreamer({name="ws0"});
```

Note: Creating a "Bitstream Writer" object is optional, in which case the encoded bitstream is not saved to a storage device.

7

Sample Scripts

This chapter walks through a few sample scripts in order to illustrate what is necessary to configure operate the MG3500 Codec.

The following topics are covered in this chapter:

- “Encoding Pipeline Sample Scripts”
- “Composing Audio Video Channel Sample Script”
- “Creating and Initializing avoc Sample Script”
- “Adding and Configuring vc to avoc Sample Script”

Encoding Pipeline Sample Scripts

The `avencode.lua` script consists of a single channel AV encode and display. This example shows how to setup a basic single encode pipeline.

This script is available with every release under

`<install dir>/merlinsw/host/luascripts/supported`

Basic Encoding Pipeline Sample Script

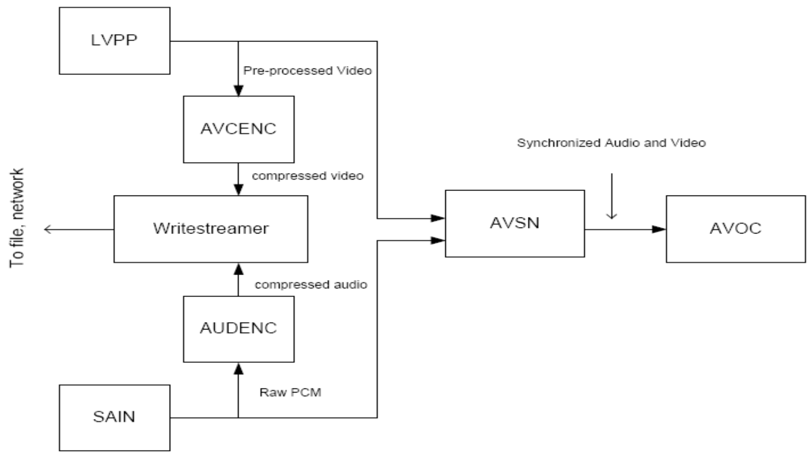

Figure 1 - **Single Channel AV Encoding and Display**

Multi-channel Encoding Pipeline Sample Script

The `dvr.lua` script shows how to leverage the first encode pipeline to create a multi-channel encode pipeline based on the same structure.

This script is available with every release under

`<install dir>/merlinsw/host/luascripts/supported`

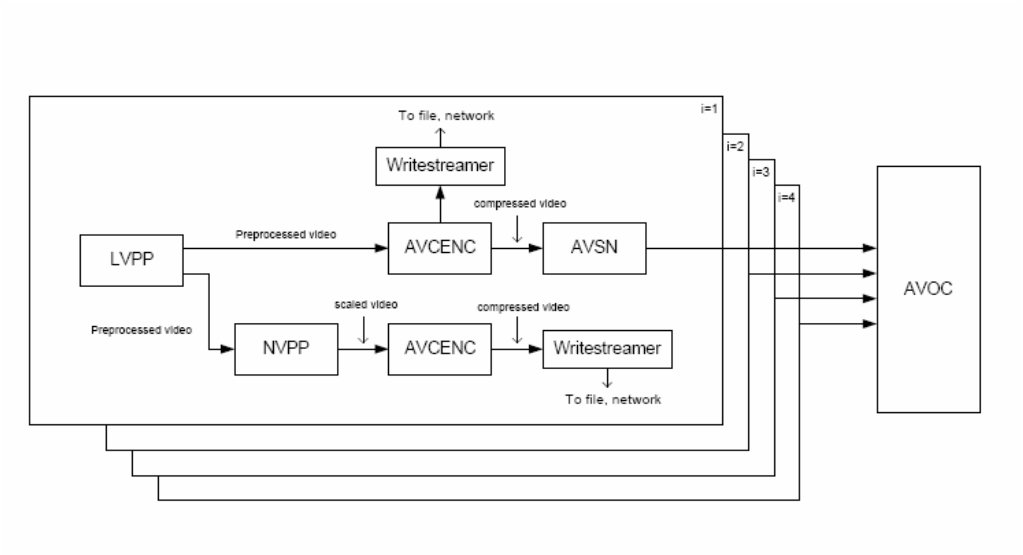

Figure 2 - **Multi-Channel Encoding Pipeline**

Composing Audio Video Channel Sample Script

In the composition mode, video channels bound to Audio Video Synchronizer `avsn` is composed and displayed. More video channels may exist but will not be presented if their source (normally an `avsn`) is not bound. This concept is useful when switching between "presentation grids." For example, five video channels can be created to show four encoders in preview mode and one decoder output. Switching to a grid of 1 x 2 consists in setting the source of the unwanted video channels to 0.

Note: An `avsn` may be connected to more than one video channel at once. It is however important to remember to maintain the appropriate `avsn` state based on the number of connections.

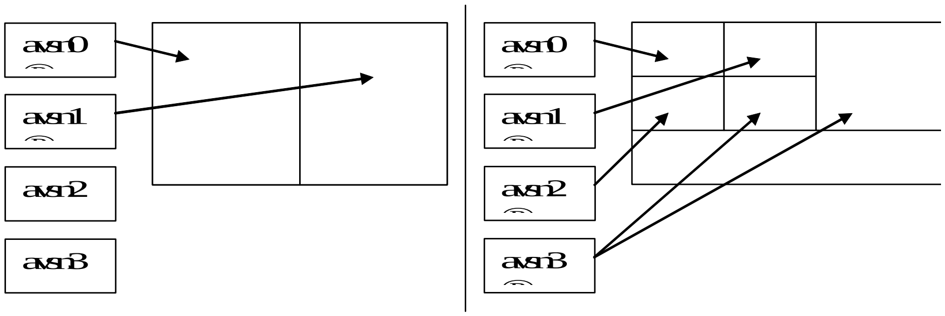

Figure 3 - **Audio Video Output Composition**

Creating and Initializing avoc Sample Script

The example below shows the creation and initialization of the avoc object.

Note: Unlike the avcenc object, avoc has separate activate methods for each method it supports.

The snippet below uses the `setParam()` method and is activated using `activateCfg()`. This configures the display size and audio output.

```
avoc0 = sysctl0:avocCreate({name="avoc0"});

avoc0:setParam({param=Q_AVOC_CMP_VID_0_DISPLAY_WIDTH,
                            value=HSIZE});
avoc0:setParam({param=Q_AVOC_CMP_VID_0_DISPLAY_HEIGHT,
                            value=VSIZE});

avoc0:setParam({param=Q_AVOC_CMP_AUD_SERIAL_MODE,

value=Q_AVOC_CFP_AUD_SERIAL_MODE_I2S});
avoc0:setParam({param=Q_AVOC_CMP_AUD_MASTER_CLOCK,

value=Q_AVOC_CFP_AUD_MASTER_CLOCK_256FS});
avoc0:setParam({param=Q_AVOC_CMP_AUD_SAMPLE_RATE,
                            value=SAMPLERATE});
avoc0:setParam({param=Q_AVOC_CMP_AUD_SAMPLE_SIZE,
value=16});

avoc0:activateCfg();
```

Adding and Configuring vc to avoc Sample Script

The example below adds and configures a video channel to the avoc object. This uses the `setVideoChannelParam()` method for configuration and `activateVideoChannelCfg()` for activation.

```
-- Create Video Channel-channel is an ID

vch0 = avoc0:addVideoChannel();

-- Define the source area. In most cases, this correspond to the full
video frame -- size

avoc0:setVideoChannelParam{
          ch = vch0,
          param = Q_AVOC_CMP_VIDEO_CHANNEL_SRC_WIDTH,
          value = HSIZE}

avoc0:setVideoChannelParam{
          ch = vch0,
          param = Q_AVOC_CMP_VIDEO_CHANNEL_SRC_ HEIGHT,
          value = VSIZE}

avoc0:setVideoChannelParam{
          ch = vch0,
          param = Q_AVOC_CMP_VIDEO_CHANNEL_SRC_OFFSET_X,
          value = SRC_OFFSET_X}

 avoc0:setVideoChannelParam{
         ch = vch0,
         param = Q_AVOC_CMP_VIDEO_CHANNEL_SRC_OFFSET_Y,
         value = SRC_OFFSET_Y}
```

```
-- Define the destination area with respect to the entire display size.
-- The offset is the position of the channel on the composed frame.

avoc0:setVideoChannelParam{
        ch = vch0,
        param = Q_AVOC_CMP_VIDEO_CHANNEL_DST_WIDTH,
        value = HSIZE}

avoc0:setVideoChannelParam{
        ch = vch0,
        param = Q_AVOC_CMP_VIDEO_CHANNEL_DST_ HEIGHT,
        value = HSIZE}

-- If the whole frame is not to be rendered, specify the

avoc0:setVideoChannelParam{
        ch = vch0,
        param = Q_AVOC_CMP_VIDEO_CHANNEL_DST_OFFSET_X,
        value = HSIZE}

avoc0:setVideoChannelParam{
        ch = vch0,
        param = Q Q_AVOC_CMP_VIDEO_CHANNEL_DST_OFFSET_Y,
        value = HSIZE}
        avoc0:activateVideoChannelCfg();

avoc0:setVideoChannelSource({ch=vch0, src=avsn0});

-- Remember to start the avsn
        avsn0:start()
```

Setting up Video Output Timing

Video output timing is setup using built-in functions. The example below configures all timing and other related parameters for a 720p60 display.

```
avoc0:setOutput720p60();
```

See-> The *MG3500 Codec Firmware API Reference Manual* for a list of configuration parameters.