DSP Programming with Mango DSP Hardware

Start-up Guide

Sept.2005

Contact Information

Mango DSP Ltd.

Postal: Har Hotzvim Industrial Park
P.O. Box 45116
Jerusalem, 91450 Israel

Tel: +972-2-588 5000
Fax: +972-2-532 8705

Detailed Revision History

Rev.	Date	Author	Approved	Description
A	14/11/04	Itay Chamiel		Preliminary Release
B	10/3/05	Itay Chamiel		
C	26/9/05	Itay Chamiel		Update for MangoX 1.0

Introduction

Thank you for choosing Mango DSP as your DSP solutions source. This guide will help you get up and running with DSP development and will help with many issues you may encounter as you begin.

About this Guide

This guide is intended for customers receiving development packages from Mango DSP. It will cover the basics of installing Mango hardware; explain the interaction of various system components; and describe proper operating and development procedures. It is also a sort of FAQ guide, covering various problems and their solutions.

First steps

The guide assumes that you have completed installation of the hardware and software drivers and libraries (see the "Mango ADK Installation" document for details on this process) and are ready to learn more about how to work with DSPs.

Conventions used in this manual

Mango DSP develops a wide range of hardware, and not all instructions in this manual pertain to all hardware types. The most important disctinction is between PCI cards such as the Seagull PMC/PCI, Phoenix, Harrier (ASB), etc, as opposed to the external Ethernet nodes such as the Raven-D, Raven-X, etc. Sections relevant to only one type of card will be marked **PCI Only** or **Ethernet Only**. (The Lark is considered a PCI system, as it is comprised of a host computer to which the DSPs are connected via PCI bus.)

All current Mango DSP hardware employs Texas Instruments DSPs. This guide therefore assumes that you are using TI's Code Composer Studio (CCS) as your development environment.

Please help!

Starting up with DSP programming in a new environment is not easy. We have attempted to explain the process as fully and as clearly as possible. If you believe this document contains errors or omissions please let us know so we can add it in the next revision.

1. Overview

The DSP is an independent processor with its own CPU, memory (both external SDRAM and internal ISRAM), real-time operating system (named DSP/BIOS) and input/output peripherals. The executable data, including the DSP/BIOS and all program code, data and libraries, are included in a single, COFF formatted file with an .out extension created by CCS upon building a project.

Host program development is done using a C/C++ environment suitable for your operating system. DSP development, on the other hand, requires a specialized development environment and special hardware for debugging. For this you will require Code Composer Studio (which operates under Windows only) and a cable, known as an emulator, that connects your PC (either through the USB port, printer parallel port, or to a PCI card) to the JTAG connector on your board.

(PCI Only) In a PCI based system, the DSP acts as a slave to the Host (PC). Such a system is normally designed as a program running under the host operating system (Microsoft Windows, Linux or VxWorks). During the initialization phase of the program, it will load the DSPs with the contents of the .out file, and start their execution. For the rest of the life of the program, the Host will communicate with the DSPs, sending or receiving data, as the application requires.
During the debug process it may sound sensible to load the DSP via the JTAG (using CCS) but this is not necessarily the case; it is actually much faster and more convenient to always have the Host load the DSP. This can cause slight difficulties with debugging using CCS, but these are easily worked around (see “Code Composer and External Loading”).

(Ethernet Only) In an external system there is no Host that loads the DSP; therefore the DSP can either be loaded using the JTAG (when debugging) or, on a production system, the .out file is stored on an on-board Flash chip and booted by the DSP on power-up. It is recommended that during the debug process the Flash remain clear, to prevent problems with CCS (see “Code Composer and External Loading”).

2. Familiarizing yourself with the Environment

(PCI Only) First, let's ensure that all hardware is installed correctly. Under Windows 2000/XP, open Device Manager (quick shortcut: Start>Run>"devmgmt.msc"). There should be a category named Digital Signal Processing Devices. Under this, there should be several devices named "Mango Digital Signal Processing Device". The number of these devices should be equal to the total number of DSPs installed in your system (an exception to this rule is the ASB card: it should display as three devices for each card in your system).
Also, under "Memory technology driver" (Win2000) or "PCMCIA and Flash memory devices" (WinXP) there should be a device named "Mango Memory Device". This driver enables access to shared memory.
Under Linux, the process is somewhat different. You have to watch the output of the module loading process, and look for lines that say "Found device *n*: TI C64" (or, for the ASB card, "Intel 21555"). The number of devices found should match the expected value as above. To access this module, go to /usr/Mango/MangoBIOS/Drivers/Linux/mango and 'make'. Then, type './load' to load the driver. To make this happen automatically at bootup, add a call to this script from the RC scripts (either in /etc/rc.d/init.d or /etc/init.d, depending on distribution). This script also loads the memory driver named "memarea".

What is shared memory?

This shared memory is required for two-way communications with the DSP. When the DSP needs to read or write data to or from the Host memory, it must access a known, fixed, physical address. Since all current operating systems move memory segments around, often swapping them out to disk, a certain amount of memory must be set aside such that the operating system completely ignores it. On Windows this is performed by adding the MAXMEM parameter to one of the lines in the boot.ini file; on Linux, a 'mem' boot parameter is added. The size of this memory depends on your application; it is recommended to allocate as little as possible, as having less RAM for your operating system will affect performance.

Where should shared memory start?

Start up Windows without any MAXMEM parameter. On the "My Computer" icon, right click and select "Properties". The amount of RAM available to Windows will be displayed. This is equal to the amount of on-board RAM, minus any RAM shared with peripherals such as on-board VGA chipsets. Divide this number by 1024 and round down to the nearest integer. Subtract by the amount of MB you wish to allocate. This is the number you should set as "Shared Memory Start" under Mango Memory Settings in the Mango control applet. Also set the size in "Shared Memory Size". Exit the applet, ensure that in boot.ini the MAXMEM parameter is equal to the Shared Memory Start parameter, and reboot.
For Linux: Normally you can assume that your available RAM is equal to the amount of RAM physically installed. However, if you suspect that a peripheral (such as an onboard VGA chip) is using some RAM, perform the following: use the 'free' command to see how much RAM is normally available to the system (look at the "Mem" entry, not "Swap"). Divide by 1024 (if the value is in KB), and round down to the nearest whole. This is the amount of RAM you're starting off with.
From the available RAM, subtract the amount of MB you wish to set aside. This value must be placed in the kernel boot command line. If you're using LILO to boot the kernel, add "mem=*n*M" (for

example, for 96 MB use "mem=96M") to the "append" variable in /etc/lilo.conf. If GRUB is used, put this parameter in /boot/grub/menu.lst.
The next step is to inform the driver as to the shared memory's size. Edit the file /usr/Mango/MangoBIOS/Drivers/Linux/mango/allocator.c, change the MEMSIZE define at the top of the file to the size of the shared memory in MB, and rebuild the driver by typing "make".

Directory Structure

When installing the Mango ADK, a Mango directory is created on your system. On Windows it is normally located under C:\Mango, but this is user definable during installation. On Linux it is /usr/Mango.
The following subdirectories will be created under Mango:

- **binHost** – contains files necessary for Host development. You will notice that library files in this directory have a suffix in their name. This suffix indicates which Windows run-time library was used to compile the library. DS – Debug Single-Threaded, DM – Debug Multi-Threaded, DMDll – Debug Multi-Threaded Dll, RS – Release Single-Threaded, RM – Release Multi-Threaded, RMDll – Release Multi-Threaded Dll. Note that in Linux this naming convention is not used, the libraries are compiled in release mode.
    - **include** – contains header files for usage of the libs. The names of many files end with "Exp". This indicates an Exported header file intended for users, as opposed to internal header files used for compiling the libraries.
    - **ext** – any third party libraries are placed in this directory. At the time of this writing, this can include GPL multimedia libraries including SDL, Xvid and Mpeg4IP.
- **binDsp** – contains files necessary for DSP development. Libraries ending with "_Debug" have been compiled in Debug mode. Others are in Release mode.
    - **include** – contains header files for usage of the libs.
    - **ext** – third party libraries (such as codecs) are placed here.
    - **Other directories** – Occasionally, a certain library must be compiled differently for different cards (for example, FPGA Stream). In this case, the library will be located under a directory with the card's name. The include file will be located under include\card_name.
- **binShared** – these header files are shared between DSP and Host programs.
- **MangoBIOS** – this library contains documentation, code examples and/or source code for the libraries, if they have been included in your package.
- **Projects** – if you have been given the source code of a software project, it will be located here.

3. Starting up Code Composer Studio

The first step, if you have not done so, is to install the device drivers for your JTAG. (TI's XDS560 is already installed with CCS, but any other emulator will require installation).
Load Code Composer Setup and close any dialog box that opens. You should have a list of available boards on the right hand side, and a list of the current setup on the left. Assuming the current setup does not suit your hardware, delete it.

If you can't see your board on the right, you may need to manually add it; this is the case for some Spectrum Digital emulators. Click "Install a Device Driver" and find the file. The first place to look should be \TI\drivers. In our case (Spectrum Digital USB Emulator) we needed to perform this step twice, for sdgo6400.dvr and for sdgo6400_11.dvr (for silicon 1.0x and 1.1, respectively). Please contact the manufacturer of your JTAG if you have difficulties with this step. Note that all DSPs relevant to this discussion are of the C64 family; this includes the DM642.

The TI C6415 DSP has two distinct silicon versions. 1.0x is a legacy version that is no longer in production, but some older C6415-based Mango cards did use it: Look for the text D101 or D103 printed on the DSP chips. If instead you find D11, or you are using a C6412 or DM642 based card, your version is 1.1.

For these two versions, there are two driver versions for most emulators. Select the version that fits your hardware and drag it to the left pane.

Now, select your board and enter its properties. The first tab "Board Name & Data File" usually doesn't need changes. If you are using an XDS560 with a silicon version of 1.0x, choose "Auto-generate board data file with extra configuration file" and as the configuration file choose \TI\cc\bin\560withc64x.cfg. Other changes may be required for other JTAG types.

The next tab "Board Properties" usually shouldn't be changed. If you are using an XDS560 with a silicon version of 1.0x, change TCLK to "Legacy". If you are using the Spectrum Digital USB Emulator, make sure the port number is 510. Other changes may be required for other JTAG types.

Under "Processor Configuration" you will tell CCS how many DSPs there are on the board. You must configure the exact correct amount of DSPs. It is highly recommended to name the first DSP "CPU_0" rather than the default "CPU_1". This is because your Host application will refer to the DSPs counting from zero.

If for any reason you wish to use less than all DSPs on the board, you must install a *bypass* instead of each DSP you wish to skip. If you do this, remember to set the value **38** when prompted for the number of bits in the instruction register.

Under the "Startup GEL file(s)" you should give all DSPs the appropriate GEL file for your board. This file should be included in the hardware support package you received with your card, and should be named <card_name>.gel. It is recommended to copy it to \TI\cc\gel. After setting one DSP, you can easily cut and paste the text string to all the others.

You can now finish and close the Setup window. Answer ‘Yes’ when asked to save the configuration, and to start CCS. CCS should now load and have all DSPs available.

4. (PCI Only) Host Program Operation

The initialization process of every Host-based project requires that the DSPs be freshly loaded with their program code and run, thus ensuring that they are in a known state. This section describes this process. Please refer to the example programs supplied with your hardware for details of exact implementation.

Step 1 – init: The first step is to open access to the DSPs, which is comprised of opening MangoBIOS, receiving information about the number of devices in the system, and opening card handles.

You may have noticed that all function calls beyond <card_name>_Open are of the form:

```
handle.function(&handle, parameters);
```

This is because each card type may have somewhat different function implementations. To hide these differences, each card handle is filled with pointers to the correct functions by the card's Open function.

After opening the card handle, the following steps must be performed for each of the DSPs in the system:

Step 2 – Warm Reset: This step is accomplished by the "reset" command. It halts the DSP and resets it to a known state. The Program Counter is reset to address 0.

Step 3 – Init Emif: The EMIF (External Memory Interface) must be properly configured to enable the DSP to access its external SDRAM memory. Such initialization can be done by the DSP itself, but often program code and data must be loaded into external memory before runtime. Therefore, the EMIF registers must be initialized by an external source, before loading any code. When using CCS, the EMIF registers are initialized by the commands in the GEL file. Since this process does not involve CCS at all, the Host must initialize the EMIF with a set of predetermined values, accomplished by the "init_emif" command.

Step 4 – Load Program: This step loads the DSP with program code from an .out file. Simply use the "load_from_file" command, with the file name as its parameter.

Step 5 – Run: Finally, release the DSP from its reset state and allow it to start running. This is done by the h2d_interrupt command.

The following steps are application dependent, but are needed in most cases:

Step 6 – Wait: Before the Host can assume that the DSP is ready, it must receive an indication from the DSP that it has completed its own initialization. This is usually done by setting aside a single word of memory at a known address on the DSP, and designating it as a "mailbox". The contents of this mailbox are set to zero by the Host before the DSP runs. The DSP, on the other hand, will place a nonzero value there when it has completed its init code. The Host will poll that memory location until it sees the correct value.

Step 7 – Setup communications: If the application uses PCI Streams (see the "Host-DSP Communications" section) this is the point where they should be initialized. On the DSP, it is best to open the streams in the init code (see Step 6 above).

Final step – Initial handshakes and communication: The rest of the process is completely application dependent. Usually the Host will send some initial parameters to the DSP, for example, letting it know which DSP it is in the system and what its function will be. From then on, data will be streamed to and from the DSP, as the application requires.

Notes:

1. Between steps 2 and 5 – that is, after Reset and before Run – the DSP is in a state in which, if CCS is running, it may hang until the DSP leaves this state. Therefore, treat this process as atomic and do not halt it during debugging. You cannot set software breakpoints in CCS after loading and before running; see the "Golden Rules" section below.
2. The h2d_interrupt command serves two distinct purposes. When the DSP has been warm reset, this command releases it from this state and begins program execution. On the other hand, while the program is already running, this command is used to send a Host Interrupt to the DSP, which can be caught using an Interrupt Service Routine in the DSP code. The "Transfers" demo is a simple example of this.

5. Code Composer and External Loading

There are several issues that must be taken into account when the DSP is being loaded by external means (Flash or PCI).

Issue 1: Code Composer will not start if DSP has been loaded externally

If the DSP has been loaded from Flash or PCI at any time since its last power-up, CCS will fail to start up. For this reason, always start CCS before running your Host application for debugging.
If you haven't followed this rule and reached a situation where CCS fails, the workaround is as follows:
(Ethernet Only) Ensure that the Flash is clear of any executable code (applications normally include a Web interface that allows clearing the Flash) and then push the board's Reset button or power cycle it.
If for any reason you can't or don't want to clear the Flash, you can try the following procedure, which may take several tries:

1. Wait for CCS to display its failure message.
2. Press the board's Reset button and click CCS's "Retry" button at the same time.
3. Repeat step 2 until successful.

(PCI Only) The simple but annoying solution is to power down the entire system and restart. Load CCS, and only when CCS is running, start your Host program.
Alternatively, some boards include a Reset button that can be used. Try it, it may or may not work.
The best solution, which requires a one-time effort, is to write a small application that brings all DSPs to a warm-reset state. This application only needs to go over all the DSPs on the card and run the "reset" function on them, and then exit (that is, run only steps 1 and 2 as described in the Host Program Operation section). After this application exits, CCS will load successfully.

Issue 2: (PCI Only) Code Composer displays strange error messages or hangs

Code Composer doesn't like the external tampering that inevitably occurs when the Host resets, loads and runs the DSP. Software breakpoints are destroyed and the CPU may no longer be in the state that CCS thinks it is. Also, once CCS is running, it may hang if the Host leaves a DSP in warm-reset state without running it. To eliminate all these problems, follow these rules:

The Golden Rules for keeping CCS happy during PCI development

Rule 1: Load CCS before performing any host access to the DSP. (For multi-DSP boards, the Parallel Debug Manager should appear. For single-DSP boards, the CCS application window should appear.) If you have already run your host program, warm reset all DSPs and then load CCS.
Rule 2: Before running the Host, all open debug windows **must** be in the following state:

- DSP is **running**. You may need to load a program first (doesn't matter which).
- **No software breakpoints** are set. You can click "Remove All Breakpoints" to quickly accomplish this.

Rule 3: Once CCS is running, it may hang if a DSP is left in warm-reset state. Therefore, treat your host application's DSP-load process (for each DSP) as atomic and don't stop it in the middle.
Rule 4: You cannot place breakpoints before the DSP is running; this is a direct result of rules 2 and 3. This means that you can't debug your DSP-side initialization routines. If you do need to debug them, load the DSP via CCS rather than by the host.

Rule 5: Once the host is running, in CCS, select File>Load Symbols>Load Symbols Only and open your .out file. This **must** be the same .out file loaded by the host! After performing this step, you may place breakpoints and debug normally.

Here are a few more tips:

- Remember to remove all breakpoints and Run before restarting the host application.
- The “load symbols” process must be done only once, but must be redone after rebuilding a new .out file. For this you can use File>Reload Symbols>All.
- An occasional dialog box that complains about a problem with breakpoints can be dismissed with “cancel”. (This dialog usually shows up when CCS is shutting down.)

6. SIO Data Communications

One of the challenges in the development of a DSP project is maintaining proper data communications between various system components. An example would be a Host having to maintain data and command transfers with many DSPs, or a DSP sending or receiving data to an external peripheral through an FPGA. In all these cases there are issues such as setting up fast data transfers, handshaking between the source and destination, ensuring that no data is lost, and overall system efficiency. The SIO mechanism aims to create a fast, efficient and robust data transfer mechanism that hides its implementation from the user and solves all of the above problems. Our experience has shown that projects using SIO Streams rather than other methods (such as mailboxes and direct transfers) are more reliable and easier to maintain, and that using SIO is preferable despite an initial learning curve.

At present, SIO Streams are used for DSP-Host communication (using the PCI Stream library) and for DSP-FPGA communication (using the FPGA Stream library). These will be described individually, but first we will explain the theory behind SIO and the Issue-Reclaim Model.

Within the DSP, SIO is a standard module within DSP/BIOS. Functions such as SIO_create, SIO_issue and SIO_reclaim are documented in the help system. What they actually do is call a device driver to handle the data transfers. The FPGA Stream or PCI Stream libraries serve as device drivers under the SIO module. They are “installed” in your project by adding an entry, in the CDB file, under Input/Output>Device Drivers>User-Defined Devices, and DIO – Class Driver. Simply copy the relevant entries from an example program.

On the other end of the communications, things work differently: See the next sections dealing with PCI Stream and FPGA Stream for details.

The Issue-Reclaim Model

Understanding the Issue-Reclaim model is the key to understanding how the SIO Stream works. In this model, data transfers are comprised of distinct buffers that are issued to the SIO system, and later reclaimed. From the time that a buffer is issued and until it is successfully reclaimed, it may not be accessed by the application.
The behavior is somewhat different depending on whether this is an input or output channel.

In an **input** channel, after opening the stream, one or more buffers are issued. Issuing more than one buffer implicitly and automatically enables a multi-buffering system (2 will create a double-buffer, 3 will create a triple buffer, and so on). Once the buffers are issued, the input loop begins. The loop starts with a “reclaim” command (which will block until data is available), data processing of the received buffer, and finally an “issue” to give the buffer back to the stream. Note that the received buffer may be any one of the buffers originally given (they will be returned cyclically, in the order they were first issued); you shouldn’t be concerned with this as long as you always process and issue back the same buffer you received by the “reclaim” command. Also note that if you’ve issued more than one buffer, the other buffer(s) will be receiving data in the background while you are processing the current buffer. If you attempt to reclaim and there is no data ready to be received, the current task will become blocked (or, the reclaim function will return after a timeout, depending on your parameters).

In an **output** channel, we will divide the behavior into two sub-groups, depending on whether this is a single-burst channel (like a command output channel) or a streaming channel in which data must be output continuously.

Single-burst: This form of behavior is used in cases such as a Host sending operating commands to the DSP. In such a case, the host issues the command, usually a small buffer with a user-defined structure containing commands and parameters. The DSP will contain a listener task that will create its command input stream, issue one buffer, and then reclaim. The reclaim command will block until a command is received from the host. Once a command is received, the DSP acts on it, possibly sends an acknowledgment to the host via a separate stream, and then issues the command buffer back to the stream. The Host, after issuing the initial command, immediately attempts to reclaim it, thereby waiting for the DSP to finish processing the command. Once it has received the buffer back, it can check the acknowledgment stream for the DSP's answer.

Streaming output: For a continuous output data channel, multiple buffers are recommended. Let us denote the number of buffers as N (N=2 for double buffering, etc). The sender should allocate N buffers, stored in an array bufs[1..N] and then act upon the following pseudocode:

```
while(processing)
     if this is iteration (1..N)
          buf = bufs[iteration number]
     else
          reclaim a buffer from stream
          buf = buffer returned from "reclaim"

     process data and create output in "buf"
     issue buffer "buf"
end
```

As you can see, during the first N iterations, one of the created buffers is inserted into the stream. After all buffers are inserted, they are cyclically reclaimed, refilled, and issued back into the stream. With such a loop, for N>1, data transfers will occur in the background from one buffer while data processing is occuring in another.

Notes:

- Over the life of a single channel, the amount of issues and reclaims must be equal. If you wish to delete a stream, you must first reclaim all buffers. (If the buffers may not be available immediately, use "SIO_flush" to force them to be available. Then reclaim as necessary.)
- It is illegal to attempt a reclaim if there are no currently outstanding (issued) buffers.

(PCI Only) Host-DSP Communications – PCI Stream

On the DSP side, the PCI Stream is a library that must be included in your project and installed in the CDB file (copy the relevant entries under I/O Devices from an example project). On the Host side, PCI Stream is used as a regular lib named "h2d_PCI_Stream" and accessed using functions with the PCI_Stream prefix (PCI_Stream_issue, PCI_Stream_reclaim, etc). On both sides, waiting for data is performed via interrupts rather than polling for maximum efficiency.

DSP-FPGA Communications – FPGA Stream

As with the PCI Stream, FPGA Stream is a library that you include and then install using the CDB. This stream protocol is used with Mango hardware for streaming various kinds of data through the FPGA, depending on the application. Different applications demand different FPGA configurations with different amounts of input and output streams that can be opened. The FPGA is configured such that the registers it displays to the DSP conform to a certain specification that enables the library to query the FPGA as to its capabilities. After this initial query the library knows how many input and output channels may be opened, which interrupt lines are used by each of them, etc. The lib then reserves all the resources that it needs (EDMA channels, interrupt lines, memory, etc).

In addition to opening and closing streams, there are often additional command and status registers within the FPGA, that must be accessed directly by the user application: For example, a FPGA dealing with video will also have a video-enable register. Details for these registers are different for each application.

Note: There are several FPGA Stream libraries due to slight differences between the various Mango cards that require it. The lib you should use will be located under \Mango\binDsp\<card_name>\.

7. (PCI Only) MangoX

MangoX is a new multimedia framework that eases the development of audio/video applications on Mango hardware. It replaces the legacy library named MangoAV. The package supports all Mango PCI cards and is based on the MangoBIOS, PCI Stream and MangoAV_HW foundations.

MangoX allows the user to define a *graph* of filters. By picking and connecting source, processing and destination filters, you can create a wide variety of different processing solutions, all using a simple API and without programming the DSP at all. Or, if you wish to add your own processing capability, all that is needed is to add another filter to the DSP project, without modifying existing code. There is also a capability to add custom commands from the Host to the DSP. MangoX is versatile, extensible, efficient and easy to use.

Filters included with the package include data transfer filters (which transfer data between the Host and DSP using PCI Streams), video and audio input and output filters, and video and audio codecs, including MPEG-4, MPEG-2 and H.263 video, and MP3 audio.

For example, a graph can use the following filters in order: FilterVideoIn, FilterMpeg4VEnc, FilterD2H_PCI. Upon submitting this graph to the DSP, the DSP will start receiving video from the video input, encoding it in MPEG-4 format, and sending the compressed frames to the Host.

The Host API enables easy access to the library's features by allowing the user to initialize filters and connect them to form a graph. This completed graph is submitted to the DSP using a simple API call, after which the DSP will begin to stream data accordingly. After opening the graph, all that is required is to send and/or receive buffers. Note that any of the more complex operating procedures (such as the need to load the DSP, perhaps configure the FPGA, set up PCI Streams) are entirely hidden and taken care of by the library. Internally, the library uses SIO for all data communications; it is designed from the ground up to allow for easy portability among various operating systems and hardware configurations.

The DSP side is supplied with the entire framework's source code, although lower level libraries are supplied in object form only. This includes the board support library for your particular card; PCI Stream; MangoAV_HW; and the included codecs.

Please see the MangoX User Manual for more details.

8. (Ethernet) RTP/RTSP Server and MangoAV_Net_RTP

Mango's networked cards are supplied with built in firmware that enables them to acts as IP nodes serving audio and video using standard RTP/RTSP protocols. Video is served in Mpeg-4 format and audio is in Mpeg-1/2 Layer 2. The client may be any standard RTP/RTSP client such as Apple Quicktime. A node is also capable of simultaneously acting as a client, receiving a stream from a remote node and playing the decoded output to its video and audio outputs. In such a way, two-way video/audio communications are possible using two boards.

The **DSP side** can be supplied to the customer as a closed or open solution. In closed form, the DSP's Flash memory is preloaded with the RTP Application, thus enabling a system that works out of the box and requires no DSP-side development. The system includes a Web-based interface that allows access to its configuration, eliminating the need for any special tools to configure the node.

In open form, the DSP application may be further developed; the RTP Server application is supplied in source form as an example project, which you may expand or rewrite as you wish. Note that lower level libraries, if included, will normally be supplied as object code only. This includes the Board Support Package for your particular card; FPGA Stream (if applicable); MangoAV_HW; the RTP/RTSP library; the webserver; Flash utilities; Networking Development Kit (supplied by TI); and any included codecs.

The **Host side** is a standard RTP/RTSP based client; you may use off-the-shelf products such as Apple Quicktime to receive and display the video. You can also custom build your own client. To simplify this process, we supply **MangoAV_Net_RTP**, a host side library that abstracts the details of creating a network connection, passing parameters and receiving data.

MangoAV_NET_RTP can co-exist with MangoX, enabling you to use both networked and PCI boards by the same application. For example, your application can receive compressed video from a remote network node, then pass the data to a Seagull PMC for decoding and display on a video monitor.

Please see the MangoAV_Net_RTP Reference manual for further information.

9. MangoAV_HW

The MangoAV Hardware support library is a DSP side library responsible for abstracting access to audio, video and other peripherals. The API is identical regardless of which card you are using, although the internal implementation may be completely different. Note that not all functionality is supported for all cards; for example the Phoenix (PC-104) board does not have a video output.

The important functions of this library include opening of access to video/audio inputs and outputs, and sending and receiving data from them. The library can also enable access to extra peripherals such as TTL inputs/outputs, if your card has them.

Please see the MangoAV_HW User's Manual for further information on this library.