Abstract — This paper outline realisation of DSP Simulator used in a time of developing application software VOIP transcoder board. The main purpose of DSP Simulator was early test application software while hardware was not ready. Together with Softswitch simulator it helped sw/hw integration and earlier delivery project to the end customers.

Keywords — Host - main processor on a board, Command - message issued by Host processor, Event - message issued by DSP processor or Simulator

I. INTRODUCTION

From the time of the first use of microprocessors and microcontrollers in embedded systems, software has been blamed for product being late to market. This stigma results from software being developed after hardware is fabricated. During the past few years, development of different types of simulator has advanced to a point where the concurrent development of software and hardware can be contemplated using simulation environments. The workflow in a traditional embedded project is shown in Figure 1

Attempts at concurrent hardware and software development have resulted in only a small portion of software bugs being found early. Software still waits for hardware prototypes before significant progress is made on integration. So, hw/sw integration problems are still found late in development and solved in software

A. Co-design and Co-simulation

The process of concurrently developing hardware and software encompasses two main areas of study: co-design and co-simulation. In the context of this article, the term co-design refers to the process of translating the requirements for a desired system level functional behavior into a partition of hardware and software designs which, taken together, provide and maintain the desired system behavior. The term co-simulation is simulation of a model of the hardware running the software, simultaneously providing visibility (and control, to a lesser degree) of the hardware model while allowing software execution to be controlled and observed at all levels of detail necessary for understanding the behavior of the system. The co-simulation environment should provide a user interface that is consistent with the current state-of-the-art for both the hardware simulators and the software emulators used by the development team.

In this document is presented one Co-Simulator solution which simulate DSP complex located on VOIP transcoder board

II. DSP DRIVERTER

A. Realisation

The DSP driver presents a set of API functions on host processor which allow application clients to communicate with DSP processors. Application clients are: Call-Control task, Diagnostic task, Provisioning task, Debug menu task etc. The DSP driver is realized using two tasks (threads) Tx and Rx. Tx task is responsible for accepting requests from application clients and pack requests in the message structures. These messages structures are written into the DSP internal memory. DSP memory is usually visible from the host processor via Motorola expansion bus (MEXI) or Intell PCI bus. It is dual-port memory. Rx task is driven by ISR which trigger DSP. The DSP processor(s) trigger ISR on a host processor whenever message from DSP(s) is available in DSP memory. ISR sends Real Time message to Rx task that there is an DSP message waiting to be read. Rx task read messages (events) and swap them from DSP memory to Host memory. Rx task forward messages to the application task assigned in the header of message via callback function.

B. Host to DSP complex (or Simulator) communication

The Host processor and DSP/Simulator communicate via shared accessible memory area located on DSP/Simulator side. This memory is divided on two segments. First segment of memory is called Command Queue and second segment of memory is called Event Queue. The Command Queue is written to by the Host processor and read by the DSP/Simulator. The Event Queue is written by the DSP/Simulator and read by the Host. For a Command Queue we can define following six parameters: Start address, Size, ReadIndex, WriteIndex, Uncongested Length and ErrorCounts. All parameters are configurable and can be defined in a header file. All queue parameters are positioned relative to queue Start address. For an Event Queue we can define following six parameters: Start address, Size, ReadIndex, WriteIndex, MultipleRate and ErrorCount.
Whenever the DSP driver wants to write a command in the Command Queue, it checks available space. If there is enough space in the Command Queue, the DSP driver will directly write the command into DSP/Simulator Command Queue. If there is not enough space the DSP driver will buffer command. The DSP/Simulator sends an Uncongested Event whenever Command Queue has increased free space from less to more of $N$ free bytes. The Rx Task of the DSP driver receives this event and invokes the Tx task to transfer commands to the DSP/Simulator Command Queue.

The DSP driver sequentially reads all messages (events) from the Simulator Event queue written by DSP/Simulator. The DSP driver increments the Read Index in a circular fashion after reading a whole event from the Event Queue by the size of event. The DSP/Simulator increments the Write Index in a circular fashion after writing a whole event to the Event Queue by the size of event. The Simulator simulates ISR and sends an Real Time message to Rx task whenever the Event Queue has events and period specified in Interrupt rate parameter has expired. Event queue on the Simulator side should be capable of holding a maximum of $N$ events. $N$ is application dependent. (~ 300 events of average size) and $N$ can be tuned during integration phase.

![Figure 2.0 Message written undivided in the command queue](image)

![Figure 3.0 Message written divided in the command queue](image)

III. DSP SIMULATOR (CO-SIMULATOR)

A. Requirements for the Simulator

In the absence of the DSP hardware (and firmware) it is necessary that Simulator simulate complete behavior of DSP complex and support following requirements:

- accept all possible commands
- generate the appropriate responses, along with a simulated Interrupt line. This should extend beyond a simple *CmdCompleted* for each command sent to the Simulator
- check correctness of generic parameters (header parameters) i.e. relationship between *dspId* and *Message Id*, correctness value of channel number etc. and in the case of error to generate *DSPError*
- check correctness of specific parameters for every command i.e. *Tx Gain* is greater than allowed value, packetisation period is not multiple of 5msec etc
- activate timer at the right moment to generate required response i.e. *CmdCompleted* after duration period
- observe the Command Queue and generate an Uncongested Event at the appropriate moment
- maintain data relating to Announcements and Tones which are configured, maintain channels which are busy etc
- simulate RTP Session Manager – activate timer and after receiving command *EndCall* generate *Stats* depending on a call duration
- change a state dependence of the command from Host i.e. Power-Up to Ready, Ready to Reset, Reset to Power-Up
- allow changing the configuration of the Simulator using CLI
- simulate Detection tones and Detection fax (accept request through CLI)
- simulate failure situation in the system i.e. “No Frame Pulse” - using CLI
- be independent of used OS and be independent of used host processor
- run on the target board i.e. in a case that DSP hardware doesn’t exist on the target hardware
- simulate real situation and do not disturb application software

B. Initialization of DSP Simulator

Upon Power-up/Reset of the PC or target board, the DSP Simulator remains in diagnostic state, until is ordered by the Host (*Tx task*) to exit this state. Since the DSP Simulator has exited the diagnostic state, it informs the host (*Rx Task*) that the DSP application has started.

While DSP Simulator application is running, the host can restart the Simulator application issuing an *Application Re-start* command. The DSP Simulator responds with *Application Started* event.
The top-level function `InitialiseSimulator()` appears in the initialization of `root` part of the host application. This function creates the Simulator task. The Simulator task has higher priority than all application tasks (greater than Call-Control, Diagnostics …). Through the process of initialization, it invokes two functions: `QUE_Init()` and `QUE_Reset()`. The `QUE_Init()` function configures the queue object and initializes all variables to the proper state. The `QUE_Reset()` function equalizes Read Index and Write Index with queue Start Address. At that moment, Simulator is created and initiated to the proper state. The DSP Simulator can be logically in two states: **PowerUpDiags** and **Ready** state.

Initialization of DSP Simulator Task corresponds to the physical reset of the DSP hardware. **PowerUpDiags** state concurs with the real state of DSP firmware when Host processor can run internal and external diagnostics of DSP firmware. The DSP Simulator stays in that state until the Host processor writes a certain value defined as `EndDiag` on a memory location referred as `DDCR`. On a real hardware, it is DSP Diagnostic Command Register. Until then the DSP Simulator task polls the value in the `DDCR` and compares it with `EndDiag`.

Since `DDCR` value is changed, the DSP Simulator goes to the **Ready** state. The DSP Simulator is capable to simulate DSP Reset (which uses application to physically reset DSP). After triggering that line (get command from Host) DSP Simulator will go immediately into the **PowerUpDiags** state. The “Reset state” doesn’t exist in the Simulator application and it is only a transient process from **Ready** to **PowerUpDiags** state.

### C. DSP Events

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChannelSetup</td>
<td>Information on Channel Setup</td>
</tr>
<tr>
<td>ToneDetected</td>
<td>Indicates Tone has been detected.</td>
</tr>
<tr>
<td>ABCDDetected</td>
<td>Pattern which has persisted for Short or Long Duration.</td>
</tr>
<tr>
<td>RTPDetected</td>
<td>RTP Named Event, which was detected.</td>
</tr>
<tr>
<td>SupportedCodecs</td>
<td>Lists the number of Codecs supported on a specific core.</td>
</tr>
<tr>
<td>DefaultParams</td>
<td>The values of the default parameters on a DSP.</td>
</tr>
<tr>
<td>PacketLoss</td>
<td>Event indicating Channels on Core that exceed threshold</td>
</tr>
<tr>
<td>Transmission Delay</td>
<td>Event indicating Channels on Core that exceed threshold</td>
</tr>
<tr>
<td>JitterBufferSize</td>
<td>Event with new value of jitter buffer, if modified.</td>
</tr>
<tr>
<td>Stats</td>
<td>Contains statistics relating to a specific channel/call</td>
</tr>
<tr>
<td>DSPInstrumentInfo</td>
<td>Debug Event listing DSP resource usage, counters, etc.</td>
</tr>
<tr>
<td>DSPDebugInfo</td>
<td>Debug Event listing DSP addresses, Channels, Frame Rate,</td>
</tr>
<tr>
<td>DSPTraceData</td>
<td>Segment of Data from a long the DSP (i.e. Len, Data)</td>
</tr>
<tr>
<td>DSPStatus</td>
<td>Returns status of DSP (Num and which channels are active)</td>
</tr>
<tr>
<td>CmdCompleted</td>
<td>Indicates execution complete on identified command.</td>
</tr>
<tr>
<td>CmdStarted</td>
<td>Indicates execution started on identified command.</td>
</tr>
<tr>
<td>CmdInterrupted</td>
<td>Indicates interrupt of currently executing command.</td>
</tr>
<tr>
<td>ApplicationRunning</td>
<td>Indicates initialization of application code complete.</td>
</tr>
<tr>
<td>Uncongested</td>
<td>Used to inform host that the DSP is no longer congested.</td>
</tr>
<tr>
<td>VersionNumber</td>
<td>Returns version number and version string of DSP firmware.</td>
</tr>
<tr>
<td>DSPError</td>
<td>Indicates DSP Error has occurred.</td>
</tr>
</tbody>
</table>

The Figure 5.0 outlines Events formed by the DSP firmware application. For example, the State machine of Host Call-Control task is completely dependant from `CmdCompleted` Event. Performance monitoring task is triggered by Stats event. After finishing the call session the DSP processor sends statistics (Stats event) regarding the previous call (received packets, transmitted packets…). All parts of Host application software can be tested simulating real DSP events.

### D. Ready State

The Ready state is operational state of DSP Simulator. The behavior of Simulator in the Ready state depends from the commands that it gets from the host processor. The Figure 6.0 outlines the major blocks for considering It shows that events typically occur as result of issued commands from the host processor (except for some...
asynchronous events), thus the Command Queue is the starting point for the flow.

1. Four Timer objects act in the DSP Simulator:
   - First timer triggers Simulator to sweep Command Queue content into a local buffer, and update the information relating to the queue (i.e. Write Pointer.)
   - Second timer triggers Simulator to do same thing as first timer but for Event queue and uses the same rate (timing) as first timer. It is up to designer to decide if he wants to merge two timers in one
   - Third timer measures the time duration required for Tone/Signaling generated commands. It is used in a same time for ComCompleted event.
   - Fourth timer measure duration of calls and helps to Session Manager for generating RTP statistics

2. For each command in the command queue, the DSP ID is examined and compared with appropriate Message Id. The Channel ID is examined and the MessageId is used to direct the contents of the message to update the tables relating to the defined resource

3. Having updated the tables, based on the characteristics field, the simulator can decide to generate an event (e.g. CmdStarted, CmdComplete)
4. Each event is copied in the event queue and queue info has to be updated (Event queue Write Pointer is updated)
5. If command require an event to occur at some subsequent time the event will be added to a list and timer will subsequently cause output to the event buffer

IV CONCLUSION

The good Co-Simulator can provide interface necessary for Host software testing procedures and fully tested Host software can help in resolving hardware issues. Without reliable and enough tested Host software, hardware issues can remain and masked or even software can be blamed for imperfect hardware

LITERATURE