Implementation of a universal framework for deployment, debugging and control of applications on DSP targets

Marko Krnjetin, Branislav Rankov, Miodrag Đukić, Vladimir Kovačević

***Abstract* — This paper presents an implementation of a universal framework for deployment, debugging and control of application on a DSP target, which can be used for fast development of SDK tools. Framework abstracts access to supported DSP targets allowing the same development tools to be used for all of them. It supports concurrent and remote access to the target. Adding support for new DSP targets is done by simply implementing predefined API. Paper also contains an example of an actual SDK based on the proposed framework.**

***Key Words* —Development environment, DSP Debugger, Framework for deployment, debugging and control of applications on DSP, GDB**

I. Introduction

Continuously growing demand for DSP based products increases dynamics of DSP application development. Besides the quality, achieving short TTM (Time to Market) becomes one of the main goals for a company producing such a product. For a DSP chip producer, this means that offering a high quality System Development Kit (SDK) becomes arbitrary. SDK includes evaluation boards, simulators and a set of software tools that help DSP application developer to write and test his application [1]. These software development tools must be considered part of a DSP product and need to be of high quality.

One part of the development tools allows user to transfer his processing algorithm to the new DSP platform and usually consists of compiler, assembler and linker. Once the application is built, it needs to be tested on an actual target DSP (or a simulator). Developer needs a way to deploy the application to the target, debug it and control it in the same way as it will be controlled on the end product.

The work described in this paper was partially supported by the Ministry of Science of the Republic of Serbia, project 12004, of year 2008.

Marko Krnjetin, RT-RK, Narodnog Fronta 23b, 21000 Novi Sad, Serbia (e-mail: marko.krnjetin@rt-sp.com)

Branislav Rankov, RT-RK, Narodnog Fronta 23b, 21000 Novi Sad, Serbia (e-mail: branislav.rankov@rt-sp.com)

Miodrag Đukić, Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia; (e-mail: miodrag.djukic@rt-sp.com)

Vladimir Kovačević, Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia; (e-mail: vladimir.kovacevic@rt-rk.com)

Tools offering this functionality are often custom made by each DSP manufacturer. Yet, a lot of the functionality they offer is similar. This offers ground for the idea about a generic framework developing tools for deployment, debugging and control of DSP application. In further text, these software tools will be simply referred to as 'Tools'.

This paper will focus on part of the framework offering communication with target.

There are numbers of frameworks offering parts of the functionality described above, but most of them are focused on targets complying with von Neumann architecture.

One example of a framework like this is GDB (GNU Project debugger). It supports debugging on a custom target via it's 'remote target' option. GDB defines an interface (called 'stub') that implements remote debugging protocol with GDB, but leaves target side part of the API to be customized for chosen platform. Once this interface is implemented, it is compiled and linked with any application that is to be debugged on target platform. When the application is deployed and started, GDB is connected with the application trough TCP or COM channel [2][3].

This approach is not very practical for resource limited, dedicated architectures like DSP. It also doesn't solve the two other requirements: deploying application (which, in case of DSP evaluation board also requires setting up board peripherals) and communicating with the application by emulating HOST control of the end product.

Compared to other processor targets, DSPs are specific in a number of characteristics: they have specialized architecture (separate code and multiple data memory zones - Harvard architecture, specialized addressing schemes and instruction sets), limited resources (memory and processor power), integrated peripherals (such as AD and DA converters, SPDIF decoders…) and real time operation [4].

For these reasons, existing generic debugging frameworks are usually not well suited for DSPs.

This paper will describe an implementation of a generic framework for delivering, debugging and controlling applications on DSP evaluation boards and simulators (in further text referred to as "DSP targets"). First part will describe some key aspects considered during the design process. Second part will describe the actual structure of the implemented framework. Final part will demonstrate an example of DSP SDK based on the proposed framework.

II. FRAMEWORK GOALS

While proposing this framework following aspects were kept in mind:

Universal interface towards all DSP targets: This interface should offer an abstraction of commands needed for deploying, debugging and controlling the applications running on a DSP target. At the same time it needs to allow each target to present it's specifics to the tools accessing it, in order for them to adapt to it if needed.

Having a universal interface towards different targets makes it possible to use the same set of developing tools for all targets supported by the framework. It also offers the ability to use the simulators in the same way as hardware DSP targets. Although this can be useful at any time when evaluation boards are not available, it is a really useful feature for DSP chip producer, as it allows him to start working on DSP applications even before the actual chip is out of production (simulators can be relatively easily generated from Verilog code [5]).

Shared Access: Multiple tools should be able to access the same target concurrently. It is a common scenario that tool for controlling the application is used in parallel with the debugger if a problem of interest is demonstrated on a specific board configuration.

Remote Access: Due to sometimes limited availability of evaluation boards or need to use them together with some other more expensive or hard to move equipment (during testing for example) it is useful to be able to access targets on a remote host machines.

Target Management: Framework should have an interface for managing and configuring available targets, possibly in a form of a GUI.

Portability: Framework should be implemented in a way that allows it to be ported to different host platforms.

III. FRAMEWORK STRUCTURE

The framework is implemented as a system of several components, main being:

- Target Definition Module – Used for adding support for a new DSP target into the framework
- Target Proxy Server – Central process containing representation of all available targets
- Development Tools API – Interface towards the tools wishing to access a target trough the framework

Implementation and functionality of each of these components will be described in more detail in this section. Structure of the framework is also illustrated in Picture 1.

A. Target Definition Module

In order for a specific DSP target to be supported by the framework, a Target Definition Module defining this target must be created. This module must implement a predefined Target Interface API, which contains methods describing usual functionality needed when working with a DSP target.

These methods can be split in four groups:

- Methods for auto - detection of the defining target. Used for automatic recognition and configuration of evaluation boards (not needed for simulators).
- Factory methods used for creating an instance of defining target interface.
- Methods for identifying the target (information about number of DSP cores, registers, memory zones and their ranges as well as configurable peripherals)
- Methods for deploying and controlling DSP application. (Writing and reading IIC or SPI buses or programming on-board flash).
- Debugging methods (standard debugging commands for setting breakpoints, stepping, resuming, reading/writing memory and registers and reading DSP's status). These API's are defined according to DSP specific needs (they specifying core, memory zone, etc).

How this API is implemented depends on the specific target.

For evaluation boards, implementation is usually going to use system driver to communicate with on-board JTAG adapter or some other interface towards DSP's debug port. It will translate Target Interface API calls into debug port understandable commands and interpret values of control registers (like DSP status) and present them as events.

In case of simulator targets, it will create an instance of simulator and offer control of the simulator execution.

Some methods in Target Interface API are not needed for hardware and some for simulator targets and can be left unimplemented.

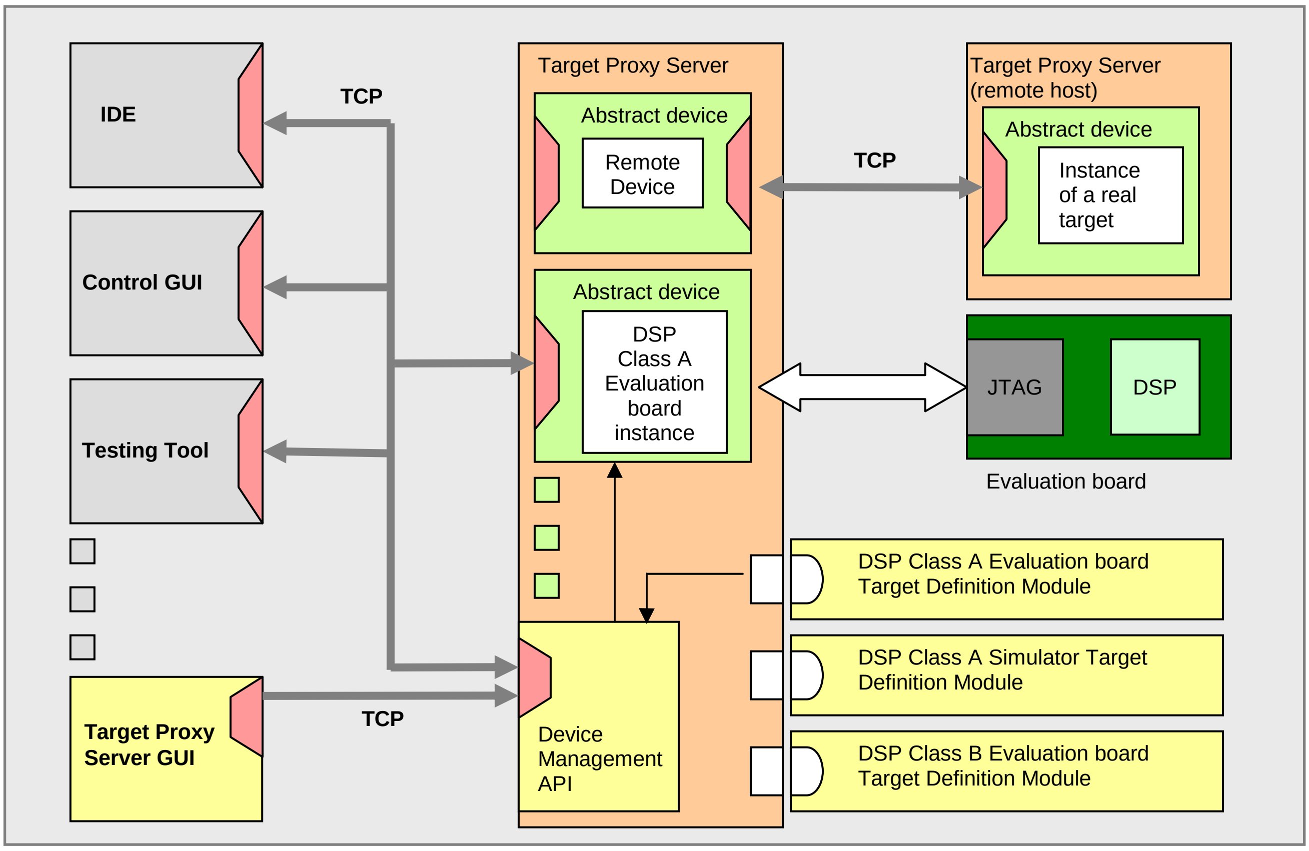
B. Target Proxy Server

The core of the framework is a Target Proxy Server application (TPS). This application is intended to be run on local host as a server and act as a proxy between software tools and the DSP target. Each supported target (connected evaluation board or created simulator instance) has it's representation in TPS. Tools that need to access any existing target, do that trough TPS, by connecting to it as clients.

Inter - process communication between tools and the TPS is done trough TCP channel. In this communication TPS is acting as a server listening on a predefined port. TCP was chosen as a communication channel because it is easily ported to different host platforms and is making the system ready for remote target access scenarios.

Each available target is represented in TPS trough an instance Abstract DSP Device (or just 'Device'). Abstract DSP Device is a bridge between client tools wishing to use the target and implementation of Target Interface API in sense that it performs following tasks:

- Manages all tools accessing target it represents, synchronizing them at the same time
- Receives commands sent by client tools and translates them into proper calls to the Target Interface API methods. It also sends the responses back to the calling tool.
- Monitors the DSP target's status (using methods from Target Interface API) and notifies all interested tools of any status changes. These


IDE
Control GUI
Testing Tool
Target Proxy Server GUI
TCP
TCP
Target Proxy Server
Abstract device
Remote Device
Abstract device
DSP Class A Evaluation board instance
Device Management API
Target Proxy Server (remote host)
Abstract device
Instance of a real target
TCP
JTAG
DSP
Evaluation board
DSP Class A Evaluation board Target Definition Module
DSP Class A Simulator Target Definition Module
DSP Class B Evaluation board Target Definition Module

Picture 1. Shows the strucure of the proposed framework. In the middle is TPS, with three Target Definiton Modules and two device instances. First device is a remote target and the second represents a localy connected evaluation board.

status changes include target getting suspended or resumed and board getting unplugged.

Device representations are created by user or automatically. On startup, TPS loads all installed Target Definition Modules and builds a list of supported targets. It hooks to each module's auto detection API to be notified when a new evaluation board is connected. When this happens it uses the factory API to create an instance of appropriate target interface implementing class. Finally, it creates a device representation for it. Simulators have to be created either manually by user or semi - automatically by tools (e.g. IDE when debug session is started on simulator).

Beside of presenting Target Interface API to the client tools, Target Proxy Server exposes additional API for device management:

- Listing existing and supported targets
- Creating and deleting device instances
- Configuring connected hardware boards
- Configuring TPS application itself (e.g. logging options).

C. *Development Tools API*

Tools can be integrated into the framework trough supplied Target Client Libraries. Each tool needing to use a DSP target must include these libraries and use them to access the target. These libraries offer the application a way to:

- connect to TPS, and query for available Abstract DSP Devices
- choose a Device of interest and connect to it
- access the selected device's Target Interface API
- register as an event listener for specific target

Calls to the methods defined in Target Client Libraries are transferred to commands understandable to the Abstract DSP Device interpreter and sent to it trough a TCP communication channel. Two channels are defined, one for issuing commands and getting responses and one (optional) for getting device's event notifications.

Note that for the tools, all DSP targets are presented as Abstract DSP devices, which allows the same tools to be used for different DSP targets. To achieve this, tools must be able to accommodate for a different targets specifics, using the data obtained trough target identification API (part of Target interface API).

D. *Target Proxy Server GUI*

In order to let user have insight in activities of the TPS, manage and configure connected boards and simulators directly, a GUI module is added. TPS GUI is not integrated with TPS application but is rather implemented as a separate application which is connected with the TPS as another TCP client. This allowed the GUI to be written using Eclipse RCP framework, making it easily portable across different platforms.

E. *Remote Abstract DSP Device*

To support using DSP targets defined on remote hosts a Remote Abstract DSP Device is defined. On a TPS running on remote host, a regular device (real or simulator) is created and configured as public, marking it ready to be used by remote hosts. On local host, using TPS GUI, a new device whose type is set to "remote device" is created. This will allow user to specify remote host address and device name. Once this device is created, all commands sent to it will be delegated to the actual device on remote TPS.

IV. Real world example

Cirrus Logic's DSP SDK is adapted to use this framework. Cirrus Logic produces several types of DSP processors with one to four DSP cores. These DSP cores contain separate memory zone for program memory and two parallel data memory zones labeled X and Y.

Several types of targets are defined trough Target Definition Module:

- DSP evaluation board target – This type of target is using USB driver towards on-board microcontroller (MCU) to access board's peripherals and DSP's debug port.
- Full simulator target – logical DSP simulator generated from Verilog code including DSP core and all peripherals.
- Single core simulator target – logical DSP core simulator (no peripherals) – useful for verifying algorithms.
- Virtual machine – a functional simulator without peripherals, speed oriented, useful for full application functionality testing.

Software tools adapted to this framework are Eclipse based IDE (CLIDE), DSP Composer (application for creation of DSP application from existing modules, deployment and control), DSP Condenser (final deployment, control and monitoring) and BBT (Black box testing Tool - platform used for automated testing of DSP system).

Some of the tools are Java and some C++ based, so Target Client Libraries for both languages are used.

Integration of a tool into the framework can be best demonstrated on CLIDE application. Being based on Eclipse framework, basic IDE functionality needed for debugging is based on Eclipse's models. This includes: source file management, debug session launching support, debug target presentation (target, threads and stack frame status), breakpoint management and expression management [6].

Following adaptations are made:

- Debug session launch dialog is customized to present user with options to select a device to which debug session is to be launched. These options are based on available targets as reported by TPS. The launch window also offers user some target specific configuration (like input and output files for simulator or start PC value) and choice of deploying the application or just attaching debugger to already running one. Dialog also checks if the selected project's target chip matches selected launch target. At this point CLIDE is accessing TPS's API for configuring and managing devices using Target Client Libraries. In case the selected target is a new simulator, CLIDE will create a new simulator device in TPS.
- Eclipse's debug model was implemented to use Target Client Libraries to send commands to TPS device and respond to it's status changes. This implementation is dynamically configured to target's number of cores.
- Memory and register viewers are custom made to be dynamically adaptable for different target configurations. Once debug session is launched, target specific information defined in Target Definition Module is collected and viewers are configured accordingly. Register viewer gets the information about available registers, their sizes and IDs, while memory viewer gets information about existing memory zones and their sizes for each DSP core.

The framework allows multiple tools to work with the same DSP target concurrently. This scenario usually includes DSP Composer and CLIDE, where the former if used to bring the application to a desired state and later to catch it and start debug it once it is there.

Literatura

[1] V. Kovacević, M. Popović, *Sistemska programska podrška u realnom vremenu*, Fakultet tehnickih nauka, Novi Sad, 2002.

[2] John Gilmore, Stan Shebs, GBD Internals – *A guide to the internals of GNU debugger*, Cygnus Solutions, 2004, ch. 20.

[3] Richard Stallman, Roland Pesch, Stan Shebs – *Debugging with GDB: the gnu Source-Level Debugger,* Ninth edition

[4] V. Kovačević, M. Popović, M. Temerinac, N Teslić, *Arhitekture I algoritmi digitalnih signal procesora I,* Fakultet tehničkih nauka, Novi Sad, 2005.

[5] Wilson Snyder, Verilator 3.804 Manual, [Online source] http://www.veripool.org

[6] Darin Wright, Bjorn Freeman-Benson, *How to write an Eclipse debugger*, 2004, http://www.eclipse.org/articles/Article-Debugger/how-to.html