STM Linux Device Driver Framework Specification Rev 0.1

Table of Contents

1.0 Terminology

SoC – System On a Chip
STM – System Trace Module (TI)/System Trace Macrocell(ARM)
STP – System Trace Protocol
DTC – Debug Test Controller
ETB – Embedded Trace Buffer
OST – Open System Trace Base Protocol
MIPI – Mobile Industry Processor Interface

2.0 Introduction

Multiple SC and IP vendors have developed STM modules based on the MIPI STP protocol, all with their own individual Linux device drivers. Each vendor's device driver has unique user space characteristics and module specific dependencies. It's the object of this driver to provide a common Linux device driver API/Framework that promotes portability of user mode code between different STM implementations.

This document describes a STM device driver architecture that presents common characteristics to user space applications, while at the same time accommodates multiple vendor specific STM implementations, multiple STM protocols (STP 1.0 and STP 2.0), and preserves the vendor specific tool chain investment.

The following STM Linux Driver White Paper link provides an introduction to STM and software message generation across multiple cores in a complex SoC, discusses device modules that generate STM messages autonomously, and provides a discussion on STM uses cases.

http://people.linaro.org/~lool/STMLinuxDriverWhitePaper_Rev0%202S.pdf

3.0 STM Roadmap

The STM roadmap applies to both application debug instrumentation and hardware messages generation (power and bus profiling).

Phase 1

Industry standard device driver APIs that export data out the pins of the device to both vendor specific and industry standard host tool chains. At the Linux user application interface promote portability between vendors.

Phase 2

Utilize Embedded Trace Buffers (ETBs) to capture STM data locally for remote data capture. Data will be transported back to host systems (customer implementation dependent) as a file for evaluation with vendor specific host tool chain.

Phase 3

Decode and evaluate ETB captured STM data locally and integrate results with standard debug tools such as perf and ftrace

Note that Phases 2 and 3 may have no impact on the STM driver, but we need to keep these goals in mind to make sure we don’t do something that prevents ETB utilization with the STM driver.

4.0 Requirements

High Level Requirements:

Vendor independent requirements:

1. User space applications require no special knowledge of STM or vendor specific module implementations
2. Provide common functionality across the standard Linux driver API functions (file operations) promoting application portability
3. Preserve the current vendor specific tool chain investments
4. User space applications can use the STM driver without re-compiling the application (through IO redirection)
5. User space applications can use the STM device through the standard driver file operation functions (open, write).
6. Support STM device enable/disable capabilities.
7. STM data is exported and captured external to the device (where a host DTC is required) or internal to the device by an ETB. ETB captured data

requires client transport to the host for processing and display. ETB capture is beyond the scope of this document.

8. Support a mechanism to provide common STM runtime and state information (number of open channels, number of bytes written on a channel,…)
9. Every message generated by the driver is terminated with a timestamp.
10. The Standard STM driver will wrap user message in OST if required for the host tool chain.
11. Provide a common method of transporting meta data (data needed for processing of messages). Both software and hardware generated messages may require meta data to provide additional context for processing.
12. Provide straight forward (easy to use) kernel logging support for printk, allowing the user to see their user space debug messages interleaved with kernel printk messages. Also provide an easy method for supporting Kprobe. Once an easy method to support Kprobe exists, exporting ftrace messages to the host tool chain may be fairly straight forward.

Vendor dependent requirements:

13. Mmap support- Provide message transport from user space with the smallest overhead possible, thus support exposing the transport hardware interface through direct mapping from kernel to user space. In this mode a vendor specific STM Library is also required to be integrated into the application. In this case the driver must maintain management and assignement of all channels used by the driver and that are mapped.
14. Debugfs - For every vendor there will be implementation specific details that need to be configured by the user. These generally are elements such as port configuration, master enables/disables, and master modes (lossless/lossy). If users utilize scripts to perform this configuration their application code will remain portable between the different device vendors. Normally driver configuration parameters and attributes should be exposed through sysfs. There are a couple of reasons why debugfs may be preferred in the STM case:
    - The Linux tracing infrastructure (ftace and perf) utilizes debugfs for their API needs, so one could make the argument that STM should follow their methodology for consistency across the tracing infrastructure.
    - STM is strictly for “debug” and “profiling” thus it’s use of debugfs is appropriate.

    An argument for not using debugfs:
    - Debugfs may not get mounted on production systems because of security holes left by other modules (although one could argue that these are bugs). The long term STM roadmap includes production use of logging and profiling tools in remote production systems. So

if by its very nature debugfs presents security hazards, it may not be a great idea in the long term to get tangled up with it.

Since the STM debugfs APIs by definition will contain vendor specific methods, to promote portability vendors should avoid exposing message transport through debugfs (debugfs/sysfs) and use the standard file operations or mmap with a vendor specific library.

Lower Level Requirements:

15. When a device node is opened by a new process, a unique STM channel will be assigned to that process id. If this rule is not enforced it breaks STM's message interleaving which provides process/thread safety. This also has the added advantage of eliminating the tendency of vendors to overload the STM channel concept. A user can create additional device nodes that follow the same rule (each combination of device node and process is assigned a unique STM channel for message transport).
16. Since STM channels are assigned dynamically a mechanism must be provided to associate the STM device node (and Linux process id?) to the message's STM channel (See Meta Data requirement)
17. Some channels will be reserved for internal use by the driver (this needs some additional definition).

Vendor Specific Libraries:

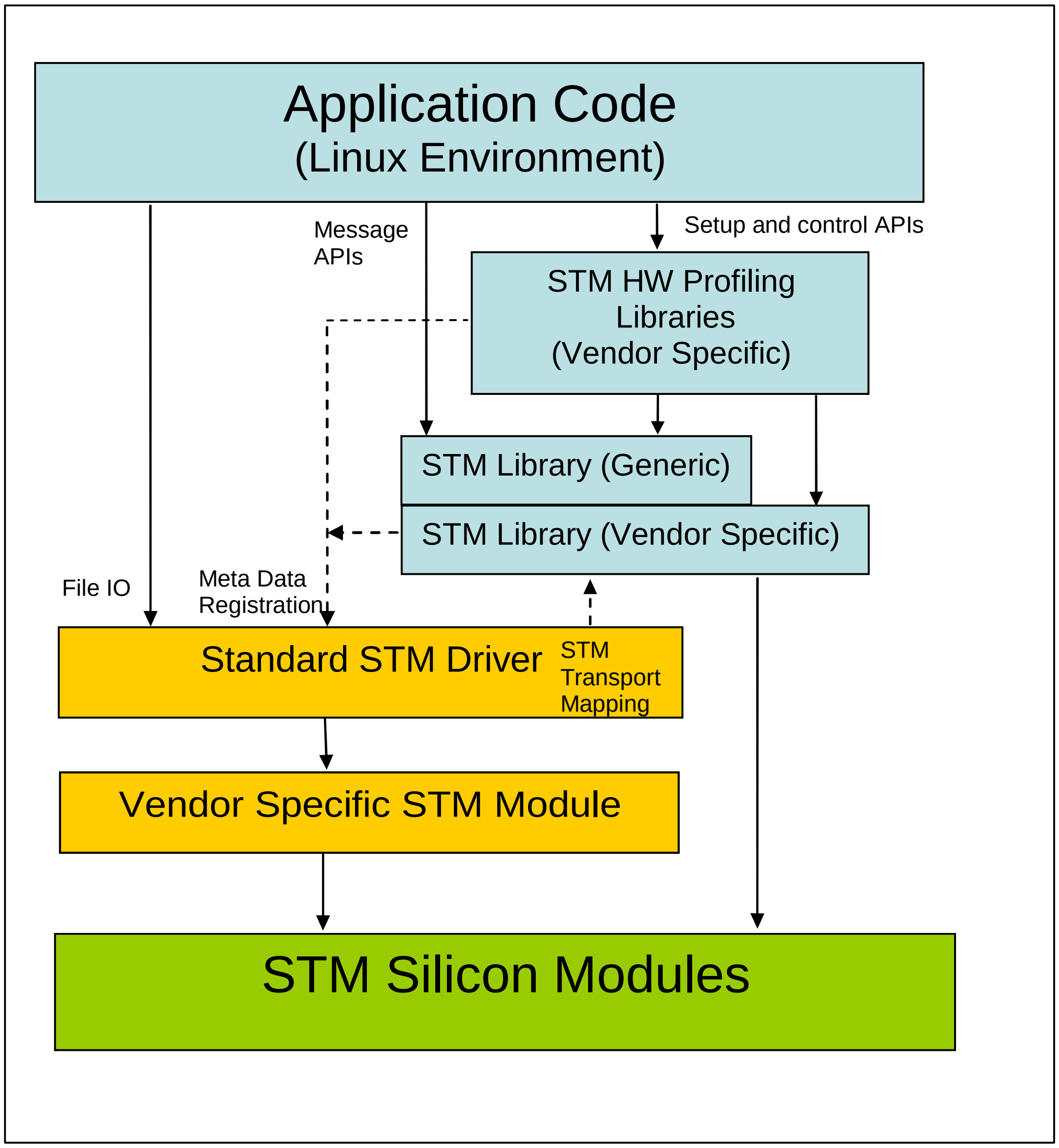
It would be very beneficial if user mode vendor specific libraries also utilized a standard API so that higher level libraries for specific debug applications (LTTng is a very good example) could be portable between different vendor specific libraries that utilized direct mapped transport (through mmap). For this reason the STM driver must at a minimum manage STM channel allocation to user space, allowing those user space applications that utilize direct mapped libraries to coexist with standard usage of the driver (through file operations) from user space.

5.0 Architecture

The following diagram shows the (proposed) architecture:

- Standard Linux stacked driver where a registration function will be exported from the Standard STM driver to register Vendor Specific STM Drivers.
- Standard STM driver that supports generic portable file operations.
- Vendor Specific STM Module provides HW dependent support.
- Standard STM driver provides uniform meta data registration and generation for all user mode components.

– STM Channel mapping to user space for vendor specific library support.



End of Document