STH

SICS Thin Hypervisor
Reference Manual
Version 0.4

VIKTOR DO

April 2013

Contents

List of Figures

List of Tables

Acronyms

AP access permission

API application programming interface

OS operating system

RO read only

RW read write

STH SICS Thin Hypervisor

Preface

The SICS Thin Hypervisor (STH) is a thin, portable, high performance software layer designed to provide increased security in embedded systems by applying virtualization for its isolation properties. This reference manual provides a detailed description of the STH hypervisor and its API.

Chapter 1

Introduction to the STH Hypervisor

The SICS Thin Hypervisor (STH) software is a small portable hypervisor designed for the ARM architecture, with the main purpose to improve the security of embedded systems through the isolation properties that virtualization can provide. This is achieved by having a very thin software layer, the hypervisor, solely running in the processors privileged mode, managing and maintaining the resource allocation and access policies to the guests running on top of it. This allows the hypervisor to limit the power of the guest by virtualizing it to run in unprivileged mode, effectively creating an isolation between the different components. With the help of the hypervisor, one can run a commodity OS kernel such as Linux and its user applications together with trusted security services in different execution environments isolated from each other.

1.1 ARM Linux support

The SICS hypervisor uses paravirtualization to virtualize the guest system to use a hypercall interface that the hypervisor provides. All privileged instructions in the Linux kernel have been replaced with the corresponding hypercalls to maintain functionality, as the guest kernel has been demoted to run in unprivileged mode. The Linux user processes are unaffected and maintains binary compatibility.

The biggest modification to the Linux kernel is the memory management in the kernel. The page tables are now READ ONLY, and any modifications have to be done through the hypervisor where it can verify and make sure that any illegal mappings are not executed. In the following section, we will discuss the Linux memory configurations and settings.

1.1.1 Linux Hypervisor domain access

There are four virtual guest modes that the hypervisor can switch between and these are the following: *kernel*, *task*, *interrupt* and the *trusted* mode. By having different virtual guest modes, we can have different domain access configurations for each mode that suits our security needs. The guest OS user process applications are

run in virtual guest mode *task*, while the OS kernel is run in the virtual guest mode *kernel*. Most important, the trusted secure applications are configured to run in the virtual guest mode *trusted*. *Interrupt* mode is used to handle interrupt context in the guest and is part of the guest OS kernel. The hypervisor is responsible for switching address spaces and maintaining the virtual privilege level of the current mode. Table 1.1 shows how each virtual guest mode's memory configuration is setup.

Table 1.1: Domain access configuration for the hypervisor guests modes.
00 - No access,
01 - Client (Access checked against AP bit in the page table, shown in table 1.2)

Domain	**3**	**2**	**1**	**0**
MemRegions	**Trusted**	**USER**	**KERNEL**	**Hypervisor Device**
GuestMode				
GM_INTERRUPT	00	01	01	01
GM_TRUSTED	01	00	00	01
GM_KERNEL	00	01	01	01
GM_TASK	00	01	00	01

Table 1.2: Page table AP configuration

Region	**Domain**	**AP (User mode)**	**AP (Privileged mode)**
Hypervisor	0	No Access	Read/Write
Device	0	No Access	Read/Write
Kernel	1	Read/Write	Read/Write
Task	2	Read/Write	Read/Write
Trusted	3	Read/Write	Read/Write

If we look at the domain access permission for the virtual guest mode *task* in Table 1.1, the kernel memory area (domain 1) are set to no access. This effectively isolates the kernel from the applications. At the virtual guest mode *kernel*, the domain access permission to hypervisor (domain 0), kernel (domain 1), task (domain 2) are all set to client. This means that for these domains, accesses are checked against the access permission bit in the page table settings. Looking at the access permissions in table 1.2 for unprivileged mode, the access permissions for these domains are all set to read/write except for the hypervisor and device region. This protects the hypervisor software and the devices from illegal accesses when the processor is in the unprivileged mode[1].

As we can see on the configuration, the trusted domain (domain 3) is not ac-

[1]All guests and trusted applications runs in unprivileged mode, only the hypervisor runs in privileged mode

cessible from the *task* or the *kernel* mode. The only virtual guest mode that can access the trusted domain is *trusted* mode which only the hypervisor can switch to.

1.1.2 Memory Layout

The Linux/Hypervisor page table memory layout looks the following: Physical address 0-1MB is reserved for the hypervisor. Address 1-15MB can be reserved for the trusted applications that provide security services to the Linux kernel and shared memory. The Linux kernel then has from 16MB - end of RAM to claim for its own use. Approximately the first two megabytes of Linux physical memory are reserved for hardware/boot information, Linux OS text, data and paging data structures. Below is a rough overview on how the physical-virtual mappings looks like.

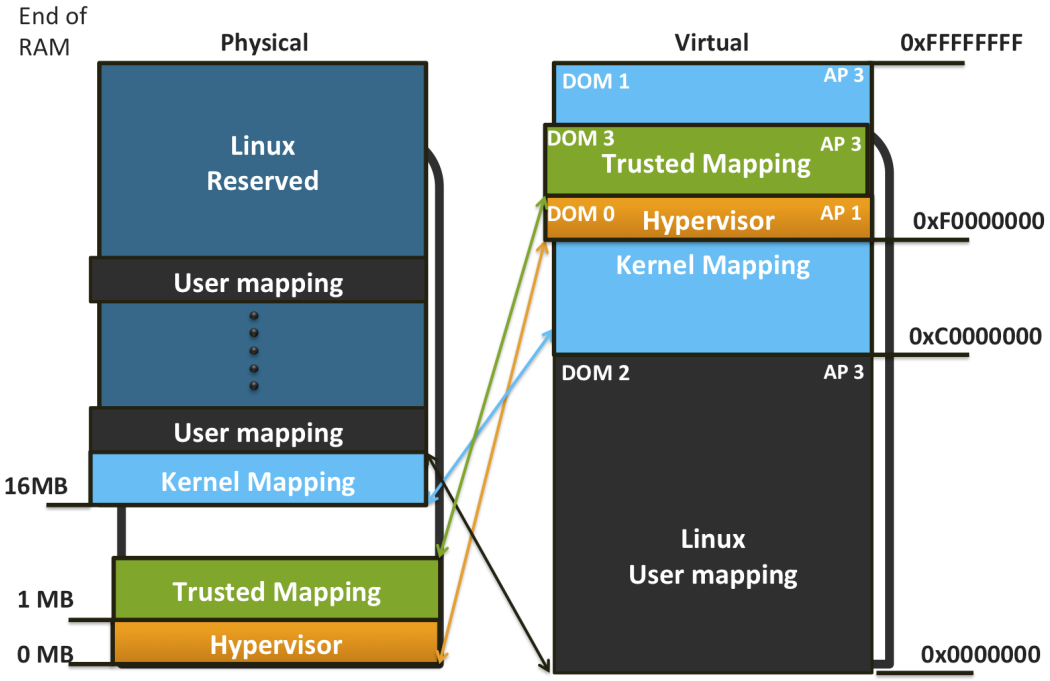

Figure 1.1: Linux/Hypervisor memory layout

The page tables have been setup so that the virtual mappings look according to Table 1.2. AP 1 specifies only privileged read/write and AP3 specifies both user and privileged read/write. With the help of the hypervisor, the domain access is restricted according to the current virtual guest mode according to Table 1.1.

1.1.3 Page table protection

In the Linux kernel, when the system is being initialized, the kernel maintains a set of page tables for its own use called the master kernel *Page Global Directory*.

After system initialization, the master page table are never directly used by any process or kernel thread. They are used as a reference table for every user process in the system, as each process has its own page global directory in where the kernel entries are copied. Whenever a process switch occurs, page tables for user space are switched as well in order to install a new address space.

In order to protect the hypervisor and security critical applications, all the kernel page tables have been set to RO. In order to map/unmap or modify the page tables, hypercalls have to be used where the hypervisor can check the page attributes for illegal values. The guest kernel can only map virtual and physical addresses that belong to the kernel. Any attempted mapping to hypervisor or trusted address space will generate an error.

Chapter 2

Platform support

The STH is a highly flexible and portable hypervisor that provides a minimal layer of hardware abstraction code on which various OS and platforms can be built upon it. This allows for easy migration to new platforms, currently STH supports the following:

CPU

`ARMv5 926EJ-S`
`ARMv7 Cortex A8`

Platforms

`BeagleBoard-XM`	:	Real HW support
`Beaglebone`	:	Real HW support
`Integrator CP`	:	OVP, Qemu simulation
`NovaThor U8500`	:	Real HW support

2.1 BeagleBoard-XM

The Linux kernel 2.6.34.3 have been paravirtualized to run on top of the STH Hypervisor. For development, we have used U-Boot as our bootloader to load and test our binary which includes the hypervisor, Linux kernel (with ramdisk) and trusted applications. The image is located in the mmc SD card, which works for development purposes, however take account that the hypervisor environment in the future will be booted from the ROM with a secure bootloader from T2Data.

Currently, there is no graphical support and all communication with the kernel is done through the UART. A bash shell console is available, in where you can navigate, explore and execute programs in the filesystem (initramdisk).

2.1.1 Security

The Linux OS together with it's user processes are executed entirely in the non privileged mode of the CPU. The hypervisor offers isolation between the Linux

kernel and it's user processes, just like in the normal case with a unmodified Linux kernel. Main difference is that we now have a trusted application running isolated barebone outside of the Linux address space. From the Linux OS point of view, the trusted application is invisible and there is no way of accessing this address space. Communication can only occur through the hypervisor.

2.1.2 Device mappings

Only the most important peripherals have been mapped to the Linux kernel such as the UART, interrupt controller and timer. Currently, the hypervisor does not allow any new mappings of devices as it poses as a potential security threat. Having devices that can perform DMA operations on arbitrarily addresses can compromise the security of the whole system.

This also protects us from a range of attacks, such as creating a mapping to the mmc card, rewrite the boot loader and bypass the loading of the hypervisor.

2.2 Beaglebone

The Linux kernel have not been paravirtualized to the beaglebone yet, due to lack of support for 2.6 vanilla kernel and the lack of a simulation model. For this platform, only the minimal guest works.

Currently, the minimal guest does not do anything useful besides printing output to the serial port.

2.3 Integrator CP

The platform Integrator CP have been tested in OVP (Open Virtual Platform) and Qemu, and has the same Linux support as the BeagleBoard-Xm. It has not been tested on real HW.

2.4 NovaThor u8500

This platform has been used in a research project and is only for SICS development purposes. It has limited compatibility and the build is not expected to work.

Chapter 3

Getting Started with the Hypervisor

In this chapter, we introduce how to get started with the STH Hypervisor, where to get the source code, compile and run it.

3.1 Getting the source

The current STH Hypervisor code is available as a compressed file under the download section at the git repository bitbucket.

It can also be obtained with the following command:

```
$ git clone git@bitbucket.org:sicssec/sth.git
```

or the https adddress from your user account in bitbucket

Obtain it with the following command:

```
$ git clone https://youruseraccount@bitbucket.org/sicssec/sth.git
```

When checked out, you can update your tree to the latest with

```
$ git pull
```

With these two commands, you can get and keep up to date with the git tree.

In the download section of the STH repository, there is also the bootloader compressed containing MLO (X-loader), uImage.bin (U-boot) and uEnv.txt (Linux boot params) for the Beagleboard XM, used under development.

3.2 Hypervisor source Tree

The hypervisor source tree is divided into different folders, listed in Table 3.1

Table 3.1: Hypervisor source tree

Directory	Description
bin	Binary of utils
core	Core of the Hypervisor
doc	Hypervisor source documentation
drivers	Low level HW drivers
guests	Guest binaries
library	Minimal library that hypervisor uses
simulation	OVP simulation platforms
templates	Configuration templates for HW and SW
test	Unit test code
utils	Utilities needed to build hypervisor

3.3 Building the Hypervisor

In order to build the Hypervisor, execute the `Makefile` located at the root of the source tree. You need a ARM cross-compiler in order to compile the hypervisor to the target system. More information on suggested compiler and tools can be found in Appendix A.1

The make will generate a uImage format file named {platform-specific}.fw.img in the folder `core\build`, which can be loaded with U-Boot in order to start up the system.

3.4 Configuring the Hypervisor

The hypervisor can be built for different platforms and this is specified in the `target` file in the root source tree.

Listing 3.1: Target config file

```
# -*- Makefile -*-
# Target configuration
#---------------------------

#PLATFORM = ovp_arm9
#PLATFORM = ovp_integratorCP
#PLATFORM = u8500_ref
PLATFORM  = beagleboard
#PLATFORM = beaglebone

#SOFTWARE = minimal
#SOFTWARE = trusted linux
SOFTWARE  = linux

#Enable this if you want to compile for OVP
#SIMULATION_OVP = 1
```

At least one of the `PLATFORM` and `SOFTWARE` variables needs to be defined in order to build successfully. It is important to perform a Make clean before switching target platform or software, as old lingering object files will lead to bugs and errors.

Some specific platforms can also be compiled for simulation on OVP, to do this, remove the comment from `SIMULATION_OVP`.

Listing 3.1 will build the guest linux for the beagleboard platform.

3.4.1 Run on HW

Rename the image file named {platform-specific}.fw.img in the folder `core\build` to uImage and copy it into /boot of the *ext* partition of the SD card.

As the hypervisor communicates through the UART port of the board, in order to see any output, you need to connect to the serial port. The tool minicom can be used, the port is usual ttyUSB0 for the Beaglebone and BeagleBoard.

3.4.2 Run on Simulation

Qemu can be used to simulate the BeagleBoard-xm. Unfortunately, there is no model for the Beaglebone. Instructions on simulation can be found in Appendix A.2

3.5 Status of build

As of commit from *2013-05-31* in the git tree, only the target Beagleboard-xm and Integrator CP platform works with the software linux and trusted linux. The minimal build works on all targets except ovp_arm9 and u8500. Other builds may not work as they currently are mainly used for development.

Chapter 4

Data Types

4.1 Hypervisor data types

There are many data structures belonging to the hypervisor, and we will concentrate on the most important ones. For more details, the reader is asked to take a look into the source code. This section focus on providing a clear overview, helping to understand the rough structure of the hypervisor.

4.1.1 virtual_machine

A guest OS is described by the `virtual_machine` data structure, and it manages the state and information of the guest, and it's configurations.

Table 4.1: **virtual_machine struct**

Type	Name	Description
`uint32_t:`	`id`	ID of the virtual machine
`uint32_t:`	`current_guest_mode`	Current guest execution mode
`uint32_t:`	`interrupted_mode`	Guest mode that interrupted the VM
`guest_info:`	`guest_info`	Memory layout and systemcalls of the guest
`uint32_t:`	`*exception_vector`	Address to the guest exception vector
`hyper_mode_state:`	`mode_states[]`	Array containing the different guest modes
`hyper_mode_state:`	`*current_mode_state`	Current guest mode that is operating
`hc_config:`	`*config`	Active guest configuration file
`virtual_machine:`	`*next`	Pointer to next virtual machine if available

The hypervisor manages the virtual machine by changing the `current_mode_state` depending on which part of the guest is running. The structure `current_mode_state` is set to one of the following guest modes.

`HC_GM_TRUSTED`	:	Trusted application context
`HC_GM_KERNEL`	:	Guest OS kernel context
`HC_GM_TASK`	:	User processes context
`HC_GM_INTERRUPT`	:	Guest OS kernel interrupt context

Each guest mode have their own context with specific domain memory access permissions. To see more details on the configuration, refer to Table 1.1.

4.1.2 hyper_mode_state

Table 4.2: **hyper_mode_state**

Type	Name	Description
context:	ctx	Structure containing context registers
hc_guest_mode:	*mode_config	Contains the domain access permission of the mode.
uint32_t:	rpc_for	Specifies the guest mode that called an rpc
uint32_t:	rpc_to	Specifies the target guest mode the rpc is aimed at

The hyper_mode_state structure contains information about the context of the mode state, domain access permissions and rpc information that the mode is handling.

4.1.3 hc_config

Table 4.3: **hc_config**

Type	Name	Description
addr_t:	guest_entry_point	Entry address of guest
hc_guest_mode:	*mode_config	Contains the domain access permission of the mode.
hc_rpc_handler:	*rpc_handlers	Contains entry address, sp and mode of the RPC.

The hc_config structure contains the configuration of the guest. The hypervisor uses this structure to setup the different mode states and guest specific configurations.

4.2 Hypercall data types

The following data types are used with hypercalls.

4.2.1 boot_info

The boot_info struct contains information about the guest OS that it passes to the hypervisor. The hypervisor also passes CPU hardware information through the cpu_id, cpu_mmf and cpu_cr registers. Used by the HYPERCALL_GUEST_INIT hypercall.

Table 4.4: **boot_info struct**

Type	Name	Description
guest_info:	guest	struct containing OS kernel information
uint32_t:	cpu_id	CPU ID version
uint32_t:	cpu_mmf	CPU memory model feature
uint32_t:	cpu_cr	CPU control register read by the hypervisor

guest_info struct contains guest specific information about the memory layout and systemcalls.

Table 4.5: **guest_info struct**

Type	Name	Description
uint32_t:	nr_syscalls	Number of syscalls in guest OS
uint32_t:	page_offset	Start address of page_offset
uint32_t:	phys_offset	Start address of phys_offset
uint32_t:	vmalloc_end	Start address of vmalloc_end
uint32_t:	guest_size	Start address of guest_size

These structs are currently used to dynamically setup the guest in the initialization phase through a hypercall.

Chapter 5

Hypercalls

In a virtualized environment, the guest all run in the CPU's user mode and have no access to privileged instructions and limited access to the hardware. In order to maintain functionality, the STH Hypervisor offers a hypercall interface to the guest. It's important to differ from hypercalls and systemcalls. Hypercalls are used by the guest OS kernel, while systemcalls are used by the guest OS user processes. The main difference is that systemcalls are forwarded to the guest OS systemcall table and are handled by the guest OS while hypercalls are handled by the hypervisor.

A hypercall is an explicit request to the hypervisor made via a software interrupt and each hypercall takes zero or more input parameters and also provides zero or more output parameters through the virtual hardware registers. Unless otherwise specified, all the virtual hardware registers will be preserved by the hypercall.

Some hypercalls executes security critical operations, and will be restricted to callers that have the appropriate access. The parameters sent with these hypercalls will also be controlled to make sure that they execute the intended correct operation. The STH Hypervisor only allows hypercalls if it origins from the virtual kernel space. Thus, user applications can not execute hypercalls.

5.1 HYPERCALL_GUEST_INIT

`hypercall_guest_init(boot_info *info)`

Parameters:

`info`	-	Pointer to a **boot_info** structure containing boot information

Output:

`info`	-	Hypervisor writes CPU information to `cpu_id`, `cpu_mmf` and `cpu_cr` in the **boot_info** struct.

This hypercall is used early in the OS kernel boot-up phase in order to to pass all the necessary kernel information that the hypervisor needs to know for it to function properly. The hypervisor also provides the OS kernel with CPU hardware information through the struct. All the necessary information that the guest OS need to pass are described in the `boot_info` struct. It also clear out all the page table mappings below the kernel image, and above it from the first memory bank (size of the guest) to `VM_ALLOC_END`. The hypervisor also makes sure that hypervisor or trusted mappings are not cleared.

As this hypercall passes a pointer from user space, the address of the pointer is checked that it reside from kernel space. This hypercall can only be performed once in the boot-up phase, and any subsequent tries will result in an error.

5.2 HYPERCALL_INTERRUPT_SET

`hypercall_interrupt_set(uint32_t interrupt, uin32_t op)`

Parameters:

`interrupt`	-	Specifies the mask in the `CPSR` for the specific interrupt. 0x80 for `IRQ` or 0x40 for `FIQ`
`op`	-	Operation for the chosen interrupt, disable (0), enable (1) or restore (2). If restore is chosen, the first param `interrupt` will be the mask to restore from.

The HYPERCALL_INTERRUPT_SET hypercall disables or enables the IRQ or FIQ in the guest system. If the operation restore is chosen, the interrupt flags will be restored to the first param interrupt. The hypervisor masks the parameters so that it only allows the guest to modify the IRQ and FIQ flag in the CPSR register.

5.3 HYPERCALL_END_INTERRUPT

```
end_interrupt()
```

Parameters:

none

The END_INTERRUPT hypercall is used by the guest OS at the end of an interrupt in order to restore the interrupted context.

Can only be called from virtual interrupt mode.

5.4 HYPERCALL_CACHE_OP

```
hypercall_cache_op(enum hyp_cache_op op, addr_t va,
uint32_t size)
```

Parameters:

op	-	Specifies the cache operation that is to be executed
va	-	Virtual address of the target (0 if not used)
size	-	Size of the intended operation (0 if not used)

The HYPERCALL_CACHE_OP hypercall is used by the guest OS to control the TLB, cache and instruction caches. It executes clean, flush and invalidate operations depending on the specified enum parameter op. List of the available operations in the enum structure hyp_cache_op are listed below. Each operation takes different amount of arguments which are specified in the list below. Arguments that are not used are disregarded by the hypervisor.

FLUSH_ALL	:	Clean and invalidate all TLB, instruction and data caches. (op=0, 0, 0)
FLUSH_D_CACHE_AREA	:	Flush data cache on an specified address range (op=1, va, size)
INVAL_D_CACHE_MVA	:	Invalidate data cache on specified virtual address (op=2, va, 0)
FLUSH_I_CACHE_ALL	:	Flush all instruction caches (op=3,0,0)
FLUSH_I_CACHE_MVA	:	Flush instruction cache on specified virtual address (op=4,va,0)
INVAL_ALL_BRANCH	:	Invalidate entire branch prediction array (op=5,0,0)
INVAL_TLB_ALL	:	Invalidate all TLB entries (op=6,0,0)
INVAL_TLB_MVA	:	Invalidate TLB on specified virtual address (op=7,va,0)
INVAL_TLB_ASID	:	Invalidate TLB on specified ASID (op=8,va,0). Special case, argument

5.5 HYPERCALL_CREATE_SECTION

```
hypercall_create_section(addr_t va, addr_t pa,
uint32_t page_attr)
```

Parameters:

va	-	Creates a section page table for the specified virtual address (va)
pa	-	Map the virtual address (va) to the specified physical address pa.
page_attr	-	Page attributes for the current mapping. Can only set for its own domain.

This hypercall creates a page table section, mapping the specified virtual address to the specified physical address. The virtual address that is to be mapped must not belong to the hypervisor or any trusted services, and the physical address must belong to the guest OS. Standard configuration is set so that the guest OS owns from the 16MB to the end of physical memory. 0-1MB belongs to hypervisor, while 1-15MB belong to trusted services (Might change in future). Trying to map illegal addresses will generate an error.

5.6 HYPERCALL_NEW_PGD

```
hypercall_new_pgd(addr_t pgd)
```

Parameters:

`pgd`	-	Address of the empty allocated space for the 16k page global directory

This hypercall is used to create a new page global directory. The guest kernel allocates 16k of space for the page table and the hypervisor makes this address space READ ONLY for the guest, cleans all the user page entries, and copies the rest of the kernel, IO and hypervisor page entries. Each user process has its own address space and own set of page tables and the kernel can switch between those with the `HYPERCALL_SWITCH_MM` hypercall.

The address of the page global directory can not reside in the hypervisor or trusted address space, and must be in the kernel address space. (Plan to incorporate a list of used page tables so that the hypervisor can keep track of them).

5.7 HYPERCALL_FREE_PGD

```
hypercall_free_pgd(addr_t pgd)
```

Parameters:

`pgd`	-	Address of the allocated 16k page global directory that is to be freed.

This hypercall is used to free an existing page global directory. The hypervisor changes back the page table address space to be READ/WRITE so that the guest kernel can reclaim the allocated space.

The address of the page global directory can not reside in the hypervisor or trusted address space, and must be in the kernel address space. (Plan to incorporate a list of used page tables so that the hypervisor can keep track of them).

5.8 HYPERCALL_SWITCH_MM

`hypercall_switch_mm(addr_t table_base, uint32_t context_id)`

Parameters:

`table_base`	-	Sets the physical translation table address. Resides in the guest address space and is read only.
`context_id`	-	Sets the context ID register.

This hypercall switches the current used page table base address and sets the context ID register. As each user process have its own set of page tables in user space, the address space of the page tables are protected by the hypervisor by having READ ONLY access in user space.

Table base address must belong to kernel address space and be READ ONLY.

5.9 HYPERCALL_SET_PMD

`hypercall_set_pmd(addr_t pmd, uint32_t pval)`

Parameters:

`pmd`	-	Address of the page middle directory
`pval`	-	Sets the page attribute

This hypercall sets the page middle directory entry according to the parameter `val`. In the ARM architecture this refers to the first level page table.

The address of the page middle directory can not reside in the hypervisor or trusted address space, and must be in the kernel address space. The physical address in the page table attribute (`val`) can also not be set to the hypervisor or trusted physical address space. Trying to do so will generate an error.

5.10 HYPERCALL_SET_PTE

```
hypercall_set_pte(addr_t *va, uint32_t linux_pte,
uint32_t phys_pte)
```

Parameters:

`va`	-	Pointer address to the Linux page table entry
`linux_pte`	-	Linux page table entry
`phys_pte`	-	Hardware page table entry

This hypercall sets the Linux page table entry at the specified address `va` and the HW pte at `va` - 0x800. Because the Linux kernel needs some bits that are not provided on the ARM architecture (such as "dirty" and "accessed" bits), the Linux kernel uses an extra set of Linux page table entries to describe these bits. More information on the layout of the page tables in ARM Linux, refer to subsection B.1.

The address of the page table entry can not reside in the hypervisor or trusted address space, and must be in the kernel address space. The physical address in the pte attribute (`val`) can also not be set to the hypervisor or trusted physical address space. Trying to do so will result in an error.

5.11 HYPERCALL_SET_TLS_ID

```
hypercall_set_tls(uint32_t id)
```

Parameters:

`id`	-	Thread ID

This hypercall is used to set the thread local storage (`TLS`) hw register. Used by the OS kernel to efficiently store per-thread local data for management purposes. TLS register is read only in user mode.

5.12 HYPERCALL_SET_CTX_ID

```
hypercall_set_ctx_id(uint32_t id)
```

Parameters:

`id`	-	Context ID

This hypercall is used to set the context ID hw register. Used by trace and debug logic in the OS kernel to identify the current process that is running.

5.13 HYPERCALL_RESTORE_REGS

```
hypercall_restore_regs(uint32_t *regs)
```

Parameters:

`regs`	-	Pointer to a struct containing cpu hw registers

This hypercall is used to restore the context information to the specified `regs` struct. Contains 17 registers from r0-r15 and the CPSR register.

The CPSR register will always be masked to run in the CPU usr mode and the address of the pointer is checked that it reside from kernel space.

5.14 HYPERCALL_RESTORE_LINUX_REGS

```
hypercall_restore_linux_regs(uint32_t return_value,
BOOL syscall)
```

Parameters:

`return_value`		Contains the return value if the parameter syscall is "TRUE"
`syscall`	-	Specifies if its a restore from a system call

This hypercall is used to restore the Linux user context. All the context information is stored inside the Linux kernel and can be quickly accessed by looking at the kernel stack pointer. The hypervisor uses this information to restore the user context. The parameter `syscall` is used to specify if a user process is returning from a systemcall. If it's set to "TRUE", the first parameter `return_value` will be returned to the restored context. If it's set to "FALSE", no return value is returned and a normal restore context is performed.

The hypervisor makes sure the correct virtual guest mode is set when restoring context. Virtual kernel mode when in kernel space, and virtual user mode when running user processes.

5.15 HYPERCALL_RPC

`hypercall_rpc(uint32_t rpc_op, uint32_t rpc_arg)`

Parameters:

`rpc_op`	-	Specifies the rpc operation that is to be executed
`arg`	-	Contains a struct with RPC information (*currently, discarded)

This hypercall is used by the guest to communicate between different address spaces, such as executing a trusted crypto service that is isolated from the rest of the system. It is up to the trusted service on how the rpc_arg should look like, and how it is interpreted.For early testing, we have support for the following operations in our trusted services, according to the parameter `rpc_op` below.

`Init`	:	Initialize trusted service. A random number should be provided for seed. (rpc_op=0, (int)seed)
`Encrypt`	:	Encrypts a local string for test purposes. (rpc_op=1, 0)
`Decrypt`	:	Decrypt a local string for test purposes. (rpc_op=2, 0)
`Read`	:	Read a local string for test purposes. (rpc_op=3, 0)

Currently, the structure of the RPC information exchange have not been defined yet as global platform support is being incorporated into the RPC calls.

5.16 HYPERCALL_END_RPC

`hypercall_end_rpc()`

Parameters:

`none`	-	

This hypercall is used to end an existing RPC call and restore the previous context.

Appendix A

Tools

A.1 Compiler

We are using the Codesourcery EABI cross-compiler for the ARM architecture and this version are known to work together with the hypervisor software. For Linux, we are using the GNU EABI.

Linux:	`Codesourcery arm-2012.03.57-arm-none-linux-gnueabi.bin`
Hypervisor:	`Codesourcery arm-2012.03-56-arm-none-eabi.bin`

Available here:
www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition

A.2 Simulation

In order to simulate the platform, we are using the Imperas simulation environment from OVP (Open virtual platforms) and QEMU to simulate the BeagleBoard XM platform.

OVP:	`Imperas 20120906.0.Linux32.exe`
Qemu:	`QEMU emulator version 1.4.0`

Available here:
http://www.ovpworld.org/
http://wiki.qemu.org/Download

More information on Beagleboard on Qemu:
https://wiki.linaro.org/Resources/HowTo/Qemu-beagleboard

The latest Qemu-system-arm version have support for the beagleboard and can be installed through the command:

```
$ apt-get install qemu qemu-system
```

A.2.1 Linaro Image tools

In ubuntu, run the following commands to install the Linaro image tools

```
$ sudo add-apt-repository ppa:linaro-maintainers/tools
$ sudo apt-get update
$ sudo apt-get install linaro-image-tools
```

Then download the nano image and omap3 hardware pack:

```
$ wget http://releases.linaro.org/platform/linaro-n/nano/11.08/nano-n-tar-20110823-1.
tar.gz
$ wget http://releases.linaro.org/platform/linaro-n/nano/11.08/hwpack_linaro-omap3_
20110823-0_armel_supported.tar.gz
```

Generate the image for Qemu, we make the image 128MB to suit our needs instead of the normal 4GB SD image.

```
$ linaro-media-create --image_file beagle_sd.img --image-size 128M --dev beagle
--binary nano-n-tar-20110823-1.tar.gz --hwpack hwpack_linaro-omap3_20110823-0_armel
_supported.tar.gz
```

Change ownership of the image file to yourself

```
$ sudo chown viktor:viktor beagle_sd.img
```

Now we have a bootable image that we can run in Qemu. To run the image in Qemu, run this command

```
$ qemu-system-arm -M beaglexm -drive if=sd,cache=writeback,file=./beagle.img
-clock unix -serial stdio -device usb-kbd -device usb-mouse -usb -device usb-net
,netdev=mynet -netdev user,id=mynet
```

In order to modify the SD image contents you can mount it on the host using a loopback mount. First find the boot partition inside the image file by:

```
$ fdisk -ul beagle_sd.img
```

Take the “start” number of beagle_sd.img1 (63) and enter it into the mount command multiplied by block size (512):

Device	Boot	Start	End	Blocks	Id	System
beagle_sd.img1:	*	63	106494	53216	c	W95 FAT32 (LBA)
beagle_sd.img2:		106496	262143	77824	83	Linux

```
$ sudo mount -o loop,offset=$[63*512] beagle_sd.img /mnt/
```

This is the boot partition in where the bootloader should be located. The source code for U-boot can be found in A.4 and needs to be compiled for the beagleboard xm. It is also available at the download section in the bitbucket repository named bootloader-beagle-xm20130517.tar.gz.

After we have copied the bootloader into the boot partition unmount the image.

```
$ sudo umount /mnt/
```

Then the hypervisor should be put into partition beagle_sd.img2. We mount the partition with loop offset of 106496.

```
$ sudo mount -o loop,offset=$[106496*512] beagle_sd.img /mnt/
```

Then we copy the hypervisor into /boot folder. U-boot expects a uImage format of the binary which can be found in the core/build/{platform-specific}.fw.img. This should be renamed to uImage so that u-boot can find it. Then we can unmount.

```
$ sudo umount /mnt/
```

After copying our bootloader into the partition beagle_sd.img1 and our hypervisor binary into the \boot folder in partition beagle_sd.im2, we can run the simulation with the shorter command:

```
$ qemu-system-arm -M beaglexm -m 512 -sd beagle_sd.img -nographic
```

To exit the simulation, press Ctrl a - x.

A.3 Programming Environment

If you want to use Eclipse CDT as your IDE, the version below is known to work together with GDB server where you can set breakpoints and follow the execution directly on the source code. We encountered some problems with newer releases such as Juno and Indigo

IDE: `Eclipse-cpp-helios-SR2-linux-gtk.tar.gz`

Available here: `http://www.eclipse.org/downloads/packages/release/`

`helios/sr2`

A.4 Bootloader

For development, U-boot is used to load the hypervisor binary into the platform. T2Data will provide the secure bootloader.

Bootloader: `U-Boot SPL 2013.01-00336-g28786eb`

Available here: `http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=summary`

Appendix B

Linux Page tables

B.1 ARM Page table layout

Hardware wise, ARM has a two level page table structure, where the first level has 4096 entries, and each second level has 256 entries. Each entry is one 32-bit word. There also aren't any "accessed" and "dirty" bits in the second level entry. Linux on the other hand has a three level page table structure (from the x86 architecture), which in the ARM port is wrapped to fit a two level page table structure using only the PGD/PMD (page global directory and page middle directory refer to the same level one page) and PTE (page table entry, level two page). Linux however also expects one PTE table per page, and at least a "dirty" bit.

Linux have solved this problem by having 2048 entries in the first level, each of which is 8 bytes (i.e, two hardware pointers to the second level). The second level contains two hardware PTE tables arranged contiguously, followed by Linux PTE's. Therefore, the PTE level ends up with 512 entries leading to the following layout:

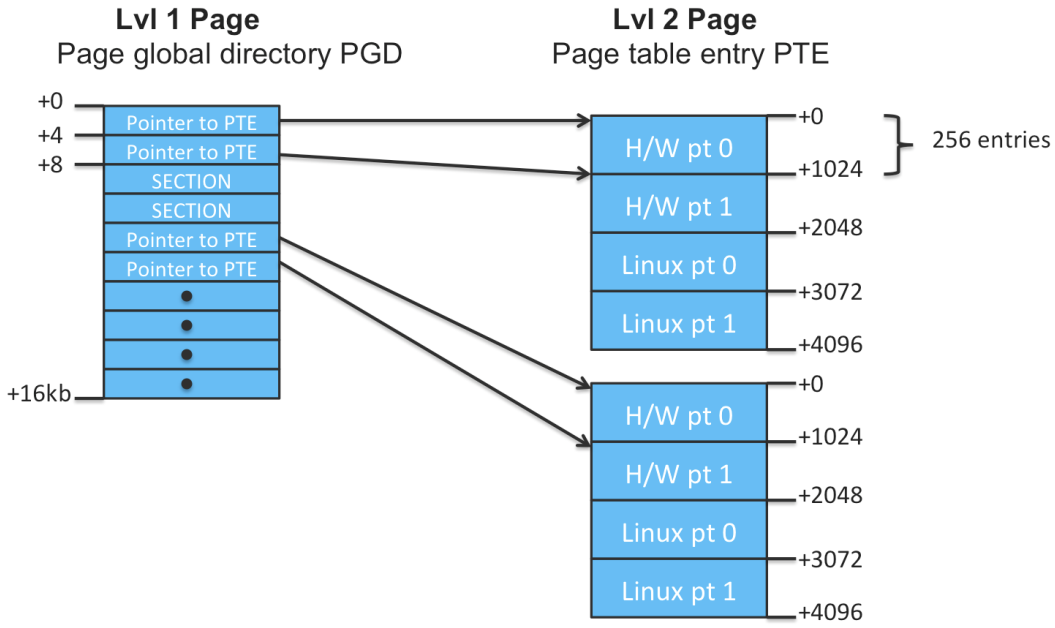

Figure B.1: Linux Page table layout