prplSecurity™ Framework

Application Note

prpl Foundation

July 2016

Security Working Group

Acknowledgements

prpl Foundation

Art Swift, President

Cesare Garlati, Chief Security Strategist

Eric Schultz, Community Manager

Benjamin DuPont, Software Engineer

Altran

David Jackson, Global Technical Director

Toon Peters, Solutions Manager

Matthias van Parys, Software Engineer

Frederik van Slycken, Technical Lead

Imagination Technologies

Majid Bemanian, Director of Marketing

David Lau, VP of Software Engineering

Intrinsic-ID

Marten van Hulst, Principal Hardware Designer

Olaf Heemskerk, Lead Software Engineer

Pierpaolo Bagnasco, Software Engineer

Roel Maes, Senior Security Architect

Pontifical Catholic University Rio Grande do Sul

Fabiano Hessel, Professors

Carlos Moratelli, Adjunct Professors

Sergio Johann Filho, Adjunct Professors

Marcelo Veiga Neves, Adjunct Professors

Contents

Introduction

This technical note describes how to build and run a secure application according to the principles set forth by the prpl Security Guidance for Critical Areas of Embedded Computing – see http://prpl.works/security-guidance. It demonstrates a real world implementation of the multi-domain security provided by the prplSecurity™ framework including: the prplHypervisor™, prplSecureInterVM™ communications, and prplPUF™ APIs.

Security through separation

Security through separation of duties is a classic, time-tested approach to protecting computer systems and the data contained therein. Security is the policy principle for protecting an asset. Separation was historically associated with "air-gapped" systems not interconnected by a network. In the context of this document, separation is a technical mechanism used to implement and maintain security. Separation may entail the use of different physical devices or other means, such as memory mapping. By separating and restricting the availability and use of assets, security is enforced according to prescribed policy. It is often said that the only secure system is one that is not connected to any other system – and even then an "air gapped" system might be compromised by non-traditional means (e.g. Stuxnet virus compromise on Iranian uranium enrichment centrifuges used in nuclear reactors). However, in a world where much value is ascribed to the interconnection of systems to create networks – so called Internet of Things (IoT), a physically and logically isolated system is not very interesting to most people. This application note focuses on systems that can retain their security attributes even when connected to open networks.

Virtualization is another technology that can achieve logical separation. Virtualized systems are used to simulate, isolate and control IT assets such as hardware platforms, operating systems, storage devices and resources for networking. Many cloud-based multi-tenant architectures are based on virtualization, and its use has surged in data centers of all kinds. Data centers typically use virtualization to consolidate physical systems by running more software workloads on fewer devices. But virtualization really has an old history, as "Virtual Machines" (VMs) were pioneered in the early 1960s by companies such as Bell Labs, General Electric and IBM Corporation. The prplSecurity™ framework uses virtualization to bring direct benefits for securing embedded connected systems.

Using Virtualization to Secure Embedded Systems

The goal of separation used for security purposes is to create and preserve a trusted operating environment for an embedded system. Separation is intended to prevent exploitable defects in one virtualized environment from propagating to adjacent virtual environments, or to the physical platform as a whole. Failures that occur in one environment are limited to that environment. Of course, when an adversary has a greater level of access, the challenge grows to fend off attacks. The greatest level of access is full physical access to the host system. Secure separation allows an embedded system to process sensitive data securely on behalf of client applications, and to continue doing so if one of the virtual environments is compromised. Separation also enables protection across and between all subsystems of a system-on-a-chip within a unified memory architecture. This means protection covers not only the CPU, but also graphics processors, audio and video processors, communications subsystems, and other subsystems of the chip.

The strategic goal of the prplSecurity™ framework for embedded systems – particularly using virtualization – is to achieve widespread, multiplatform enablement of trusted operating environments that are not limited to a single trusted computing domain, a single application environment, or to the CPU. Unlocking this interoperability is the Holy Grail of multi-domain security for embedded systems.

Hardware-Assisted Virtualization

Virtualization is a logical separation technology used to simulate, isolate and control IT assets such as hardware platforms, operating systems, storage devices and resources for networking. Virtualization allows the running of multiple systems and applications on a single platform. In the IT data center world, consolidation of infrastructure, easing manageability and reducing capital and operational costs are powerful benefits of virtualization. For embedded systems, virtualization is a clear path to "doing more with less" on millions or billions of globally deployed devices. While data center virtualization now separates the management of computing resources, storage and networking into distinct virtual domains that can be combined to create a complete computing environment, embedded virtualization will focus on sharing the subsystems of a single system-on-a-chip integrated circuit in separate VMs during the foreseeable future.

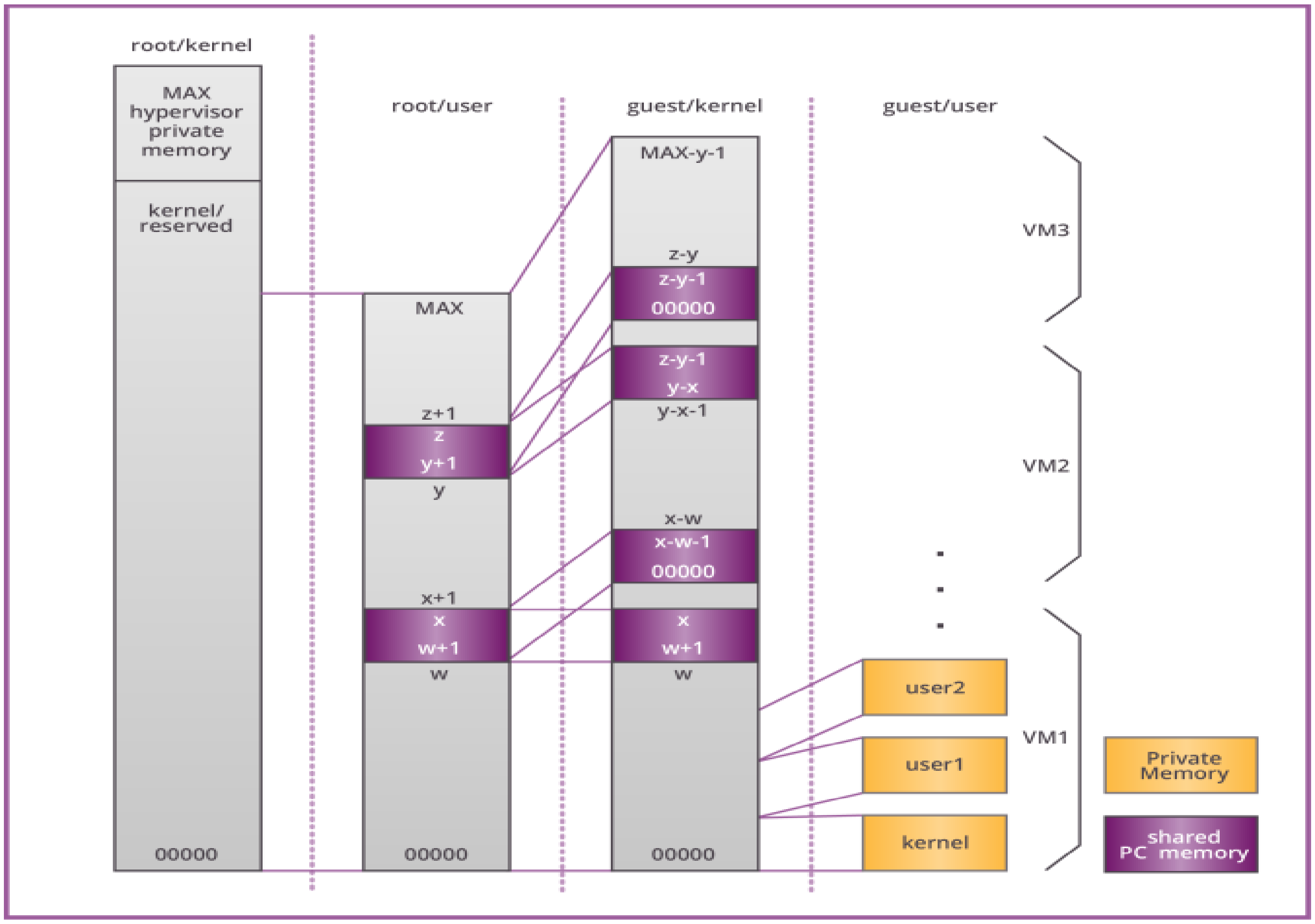
Virtualization also offers many benefits for embedded systems security. Separation is used to create and preserve a trusted operating environment. It is intended to prevent exploitable defects propagating beyond the environment in which the failure occurs. Separation allows an embedded system to process sensitive data securely on behalf of client applications, and to continue doing so if a peer VM is compromised. Separation also enables protection across and between all subsystems of a system-on-a-chip within a unified memory architecture. This means protection covers not only the CPU, but also graphics processors, audio and video processors, communications subsystems, and other hardware subsystems.

Hardware-assisted virtualization creates a virtual machine on a platform. The virtual machine simulates a real computer and its operating system in order to host the operation of a guest operating system and applications. The crux of this scheme that enables everything to work is strict control of shared memory from the point-of-view of guest system components that are accessing the actual underlying hardware platform. The CPU, graphics processor, and all other devices on the platform must have a consistent view of this memory and its accessibility from any particular guest VM.

The creation, allocation of resources and access rights to VMs, and management of guest virtual machines is the job of the hypervisor. A hypervisor may be monolithic, or may consist of a minimal kernel ("microkernel") and associated user layer. In embedded systems, we generally prefer a microkernel approach as the minimal complexity of a small piece of trusted software is easier to inspect for defects that may reduce security.

A critical function of the hypervisor is to allocate memory for all resources on the platform. If errors creep into memory management, there is a risk of device or application failures. As for security, the occurrence of memory management errors creates significant opportunity for malicious exploits of the embedded system. Effective memory management is the key to securing virtualized embedded systems.

Translating Virtual Memory Allocations to Physical Memory



The figure above shows the different views on memory as seen by the hypervisor and guest VMs in translating virtual memory addresses to physical address in a four layer memory management scheme. The hypervisor microkernel operates at the root/kernel (highest) privilege level and controls all physical memory. The microkernel provides a very limited set of features: threads, memory management, inter-process communications and scheduling. Non-core hypervisor utilities provide memory allocation, driver paravirtualization, and other functionality. Each virtual machine starts in an apparent physical memory space allocated by the hypervisor. The guest VM level memory management unit and other memory management functions of the guest OS kernel are oblivious to the actual physical location of data in memory. User space allocations are managed normally by the guest OS kernel.

As mentioned in the introduction, one of the attractions to using hardware-based virtualization for embedded systems is that guest operating systems can run unmodified or with minimal porting requirements – usually related to device drivers and physical I/O. Also, CPU virtualization typically offers faster performance than other modes of virtualization. From a security perspective, the guest operating system is left unmodified, so it is harder to detect that it is operating in a virtual machine – and it is less susceptible to attacks even if the guest detects that it is operating in a VM. Also, hardware virtualized peripherals are not as dependent on the correct operation of the hypervisor to assure separation. The main caveat for hardware-based virtualization is that the hypervisor-managed second-level memory mapping is managed correctly to ensure each guest is separated from each other.

prplHypervisor™

The prplHypervisor™ is the industry-first open source hypervisor specifically designed to provide security through separation for the billions of embedded connected devices that power the Internet of Things. The MIPS M5150 version of the prplHypervisor™ implements MIPS VZ extensions to provide a lightweight isolation layer for Microchip Technology's PIC32MZ microcontrollers. In addition to real-time hardware virtualization, the prplHypervisor™ provides additional security services including prplPUF™ authentication and prplSecureInterVM™ communications.

The prplHypervisor™ features minimal attack surface - less than 7,000 lines of code, limited footprint – 30KB flash, 4K RAM/VM, up to eight isolated domains, and strong temporal isolation EDF algorithm for real-time VCPUS. Preliminary performance tests show negligible overhead for context switching and interVM communications. prplHypervisor™, prplPUF™, and prplSecureInterVM™ technologies are part of the prplSecurity™ open source framework and are released under prpl Foundation permissive license – see http://prplfoundation.org/ip-policy.

MIPS Virtualization Module (VZ)

Virtualization defines a set of extensions to the MIPS32 Architecture for efficient implementation of virtualized systems.

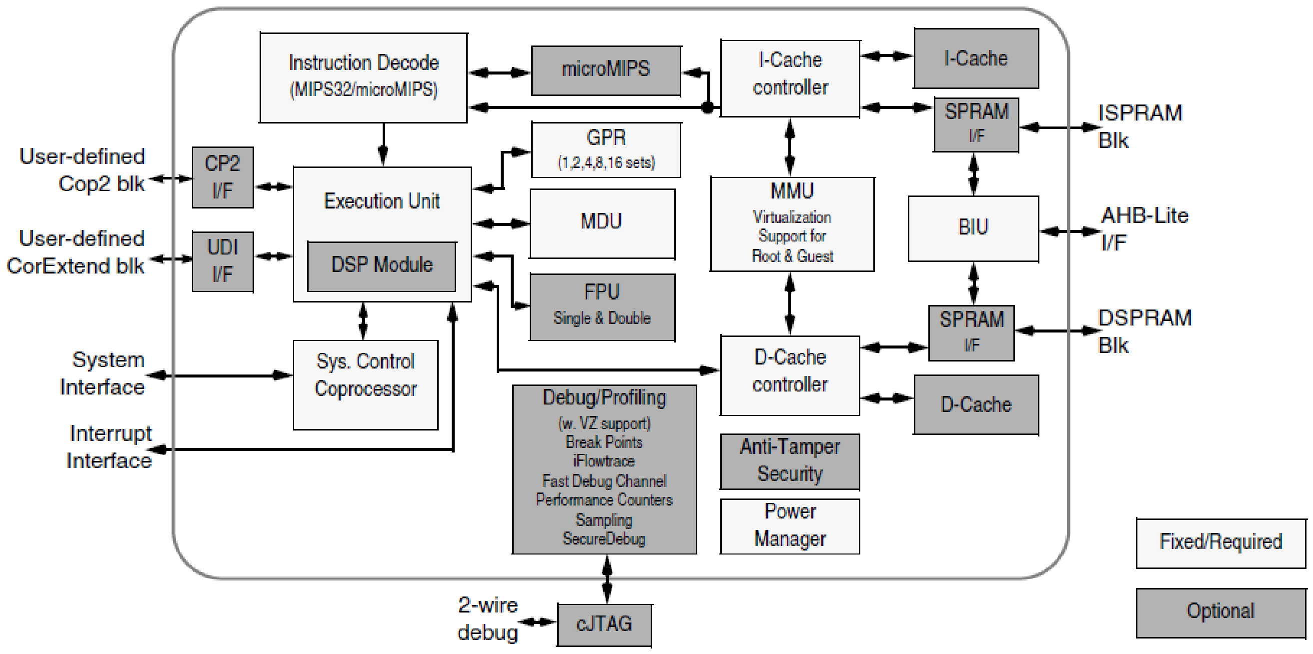
Virtualization is enabled by software—the key element is a control program known as a Virtual Machine Monitor (VMM) or hypervisor. The hypervisor is in full control of machine resources at all times. When an operating system (OS) kernel runs within a virtual machine (VM), it becomes a guest of the hypervisor. All operations performed by a guest must be explicitly permitted by the hypervisor. To ensure that it remains in control, the hypervisor always runs at a higher level of privilege than a guest operating system kernel.

The hypervisor is responsible for managing access to sensitive resources, maintaining the expected behavior for each VM, and sharing resources between multiple VMs. In a traditional operating system, the kernel (or supervisor) typically runs at a higher level of privilege than user applications. The kernel provides a protected virtual-memory environment for each user application, inter-process communications, IO device sharing and transparent context switching. The hypervisor performs the same basic functions in a virtualized system, except that the hypervisor's clients are full operating systems rather than user applications.

The virtual machine execution environment created and managed by the hypervisor consists of the full Instruction Set Architecture, including all Privileged Resource Architecture facilities, plus any device-specific or board-specific peripherals and associated registers. It appears to each guest operating system as if it is running on a real machine with full and exclusive control.

The Virtualization Module enables full virtualization, and is intended to allow VM scheduling to take place while meeting real-time requirements, and to minimize costs of context switching between VMs.

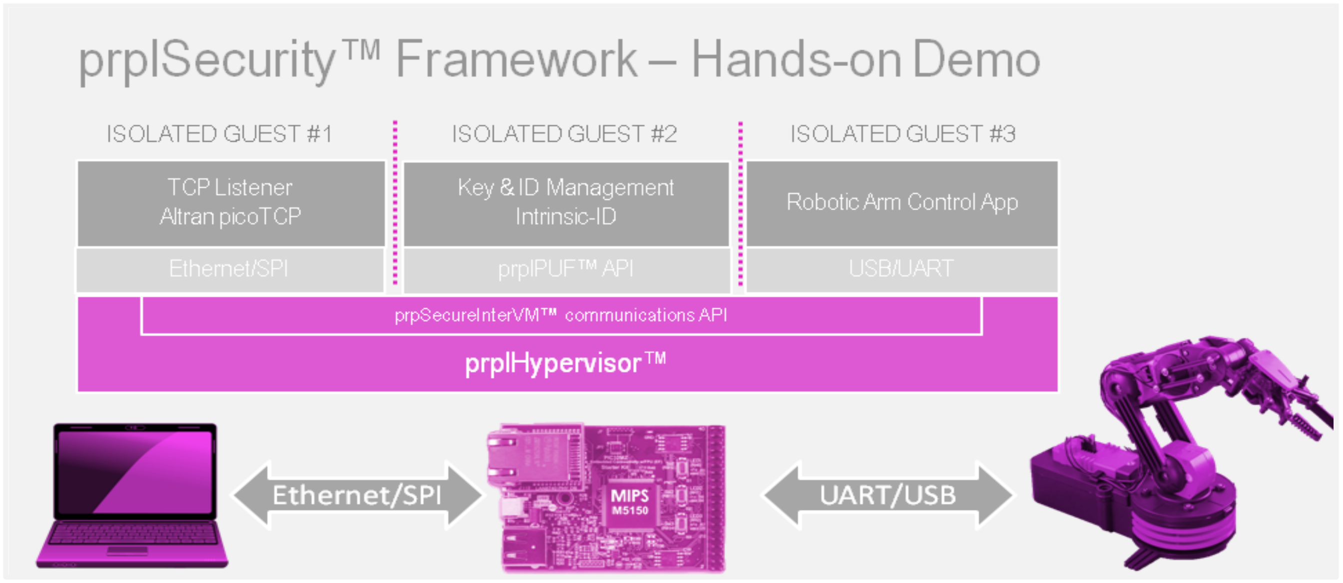
MIPS 32® M5150 Core Block Diagram



Application Concept

This note describes the application of the prplSecurity™ framework to a real word scenario: an embedded application that controls the movement of a robotic arm connected to the Internet.

The system architecture is comprised of three separate bare-metal applications, each running in its own virtual machine. The prplHypervisor™ enforces hardware-level separation of CPU and memory and secure access to I/O peripherals. The prplSecureInterVM™ API provides secure communications across the three security domains. The Intrinsic-ID implementation of the prplPUF™ API authenticates incoming requests. A virtualized implementation of Altran's picoTCP provides a complete TCP/IP stack optimized for resource constraint devices.



Virtual machine #1 connects to the Internet via Ethernet – or to a laptop in case of a disconnected demo, listens for incoming connections for host 192.168.0.2 port 80, authenticates the incoming requests via the prplPUF™ services provided by VM #2, and eventually relies valid commands to VM#3 for execution.

Virtual machine #2 provides a virtualized implementation of the prplPUF™ API. During the initial enrollment phase it wraps a predefined secret to create a unique repeatable key. At every boot it recreates the same unique key by fingerprinting the hardware itself so that no keys are ever stored in the system. It then provides authentication services to VM #1 via secure inter-VM communications.

Virtual machine #3 connects to the robotic arm via USB, listens for incoming authenticated commands from VM #1, and generates a predefined sequence of real-time instructions to operate the five motors and the light of the robotic arm. Command '1' starts/restarts the sequence from the predefined home position. Command '2' immediately turns off the light to acknowledge the receipt of the pause command and stops the sequence when the arm reaches the home position. Any other command is ignored.

Important: 1) the three virtual machines are completely separated at the hardware level 2) can only communicate via secure inter-VM communications 3) the authentication key is not stored in the system.

How to flash the prpl bootloader

This is a onetime operation necessary to replace the Microchip bootloader with the prplSecurity™ bootloader. Once the prpl bootloader is written in flash memory there is no need to repeat these steps. The original Microchip bootloader can be restored at any time using Microchip MPLAB IDE or IPE.

Important: once installed, the prpl bootloader connects to J11 [USB to UART] – not J3 [USB Debug].

Important: this onetime operation requires the Microchip MPLAB IDE or IPE software installed in your development system – see http://microchip.wikidot.com/ipe:installation. The Microchip software is only available for 32bit platforms. If you run Linux 64bit you need to make sure you have the 32bit compatibility libraries installed.

1) Install the following software packages on Linux 64 bit:

```
~$: sudo dpkg --add-architecture i386
~$: sudo apt-get update
~$: sudo apt-get install gcc-multilib
~$: sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
~$: sudo apt-get install libexpat1-dev:i386
~$: sudo apt-get install libX11-dev:i386
~$: sudo apt-get install libXext-dev:i386
```

2) Connect your development system to J3 [USB Debug] and use the MPLABX IPE software to flash the Microchip_UART.hex file to the board – this file is located in the prplHypervisor/bin directory.

Upon successful completion of this step, the red led LED1 blinks for 5 seconds to indicate that the prpl bootloader is ready to upload a new hex file from J11 [USB to UART].

Note: you can reboot the development board at any time by pressing SW1.

How to build the application

1) Install the MIPS-MTI toolchain – see http://community.imgtec.com/developers/mips/tools/codescape-mips-sdk/download-codescape-mips-sdk-essentials. Note: the Microchip toolchain doesn't support MIPS VZ extensions and can't be used for the development of hardware virtualized applications.

2) Install the srecord package. The srecord package is a collection of tools for manipulating EPROM load files. It reads and writes numerous EPROM file formats and can perform many different manipulations.

```
~$: sudo apt-get update

~$: sudo apt-get install srecord
```

3) Optional: uninstall the ModemManager package. The ModemManager software package scans for modems on individual serial ports when they are plugged in – i.e. ttyACMx ports. This keeps the port busy for several seconds. To avoid this inconvenience uninstall this package from your system.

```
~$: sudo apt-get remove modemmanager.
```

4) Clone the prplHypervisor™ repository and switch to the demo-july-2016 branch:

```
~$: git clone https://github.com/prplfoundation/prpl-hypervisor

~$: cd prpl-hypervisor

~/prplHypervisor$: git checkout demo-july-2016
```

Note: the above steps are only required if you plan to contribute to mainline development. If you just want to build a local instance you can download a snapshot of the branch at:
http://github.com/prplfoundation/prpl-hypervisor/archive/demo-july-2016.zip

5) Download the picoTCP library and expand its folder at the same level of the prplHypervisor folder:
http://github.com/tass-belgium/picotcp/releases/download/prpl-v0.1/libpicotcp.tgz

6) Use the "make" command to compile and build hypervisor and virtual machines. This process will generate the firmware.hex file:

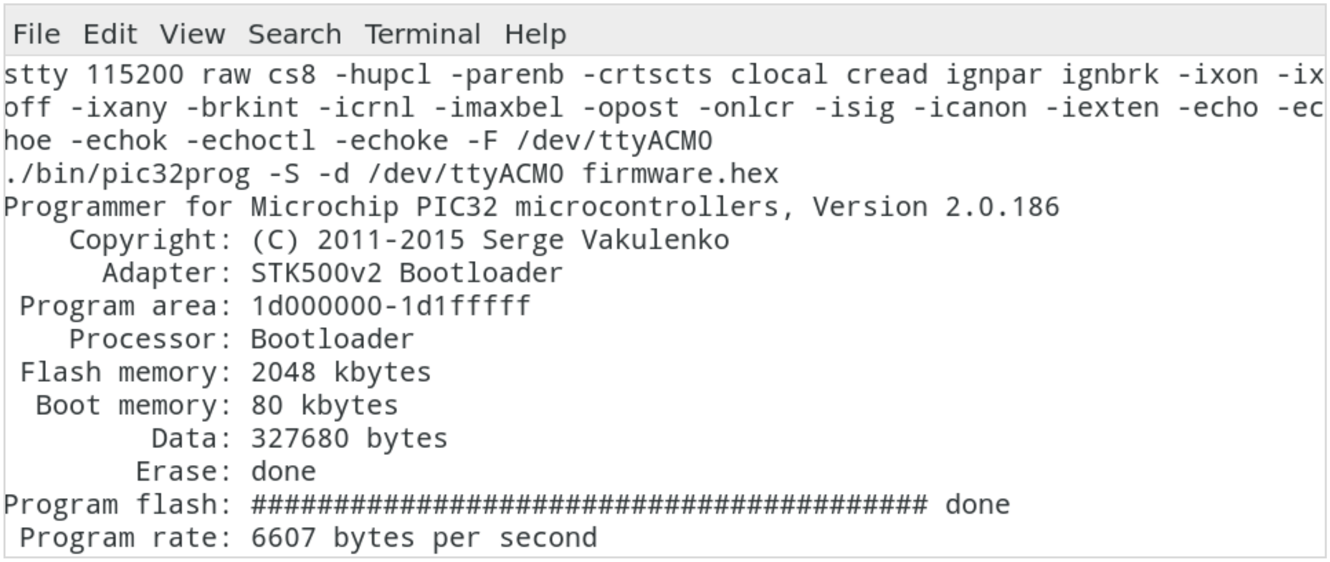
```
~/prplHypervisor$: make
```

7) Upload the firmware.hex file to the board using the "make load" command and the prpl bootloader.

```
~/prplHypervisor$: make load
```

Note: make sure the board is connected via J11, powered via J4, and the red led LED1 is blinking – if not press SW1 to reboot.

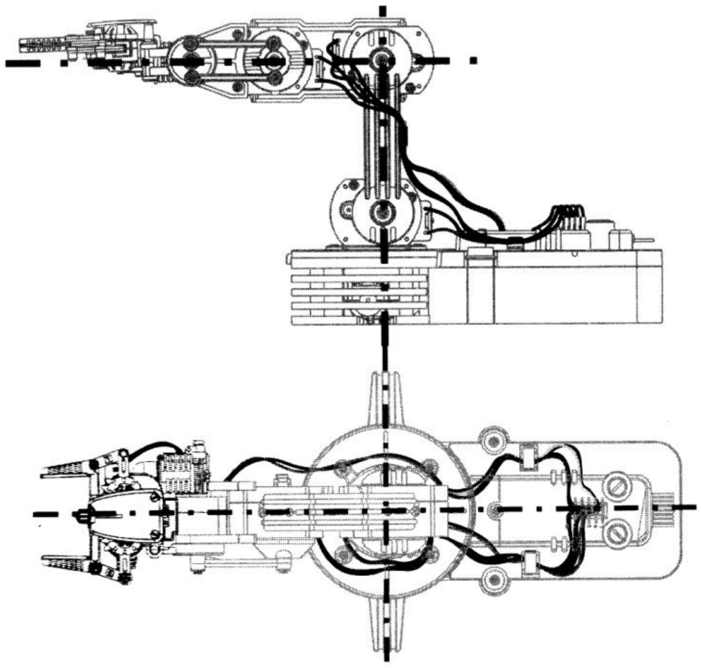
Below is the output of the "make load" command



```
stty 115200 raw cs8 -hupcl -parenb -crtscts clocal cread ignpar ignbrk -ixon -ix
off -ixany -brkint -icrnl -imaxbel -opost -onlcr -isig -icanon -iexten -echo -ec
hoe -echok -echoctl -echoke -F /dev/ttyACM0
./bin/pic32prog -S -d /dev/ttyACM0 firmware.hex
Programmer for Microchip PIC32 microcontrollers, Version 2.0.186
    Copyright: (C) 2011-2015 Serge Vakulenko
      Adapter: STK500v2 Bootloader
 Program area: 1d000000-1d1fffff
    Processor: Bootloader
 Flash memory: 2048 kbytes
  Boot memory: 80 kbytes
         Data: 327680 bytes
        Erase: done
Program flash: ######################################### done
 Program rate: 6607 bytes per second
```

How to run the application

1) Connect the robotic arm to the laptop USB port. Switch on the robotic arm. Use the OWI Windows application or the Linux utility to position the motors in the home position as indicated below. Power off the arm.



   Important: the robotic arm as no servomotors or feedback mechanisms. The control application running in VM #3 has no way to detect the initial position of the arm. If the home position is not correct, the sequence will likely overextend the arm's motors and potentially result in permanent mechanical damage of the gearboxes.

   Note: alternatively you could use the wired controller instead of the USB interface to perform step #1.

2) Connect the robotic arm to the board USB Type A receptacle J5. Switch on the robotic arm.

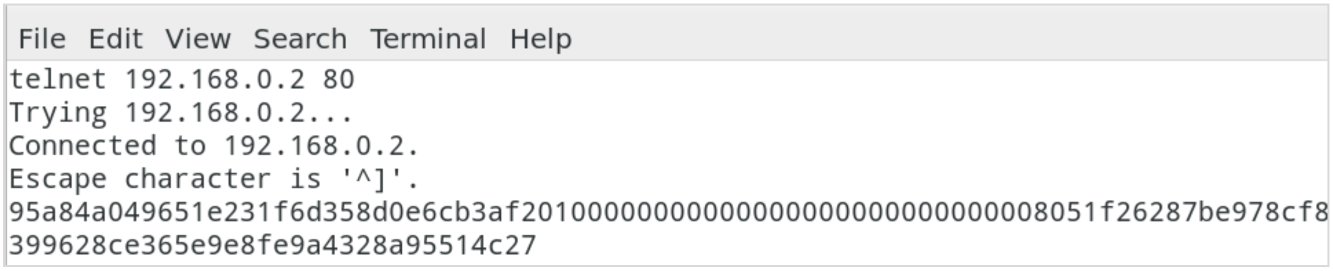
3) Assign a static IP address to the laptop in the range 192.168.0/24 – i.e. 192.168.0.1. Do not assign address 192.168.0.2 as we use it for the board. Connected the board to the laptop via Ethernet PHY daughter board.

4) Connect the board USB Type micro-AB receptacle J4 to a 5V power source – i.e. laptop USB port. The red led LED1 will blink for 5 seconds. The system is now listening for incoming connections on the Ethernet port.

5) Ping host 192.168.0.2 to test Ethernet connectivity and verify that VM #1 is up and running:

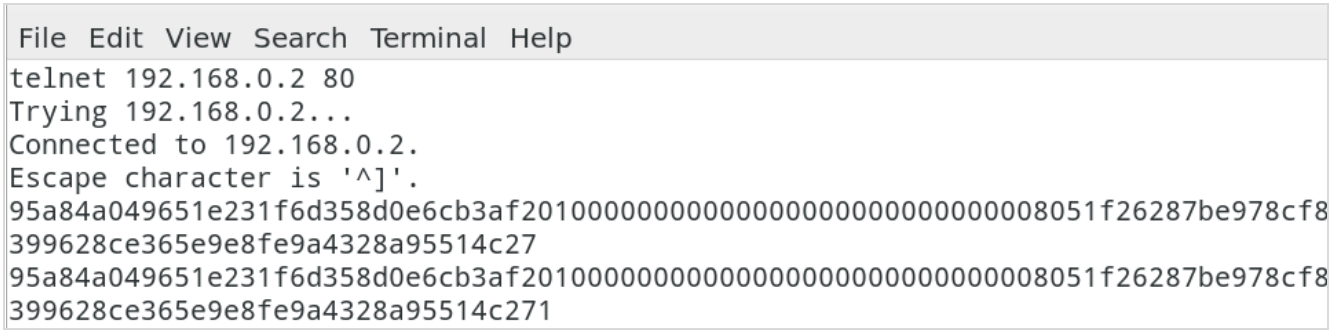
```
~$: ping 192.168.0.2
```

6) Open a telnet session for host 192.168.0.2 port 80. VM #1 will reply by sending the 56bit key in ASCII format. Note: this is only for demo purpose. Real word applications should report the wrapped key only during the initial enrollment phase. The key is unique for each device and will be different from the one shown here.



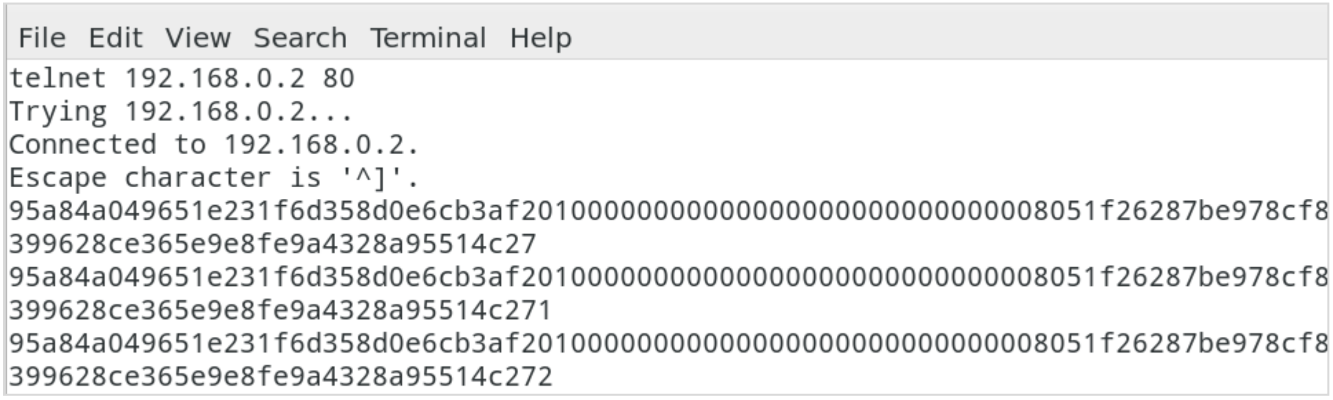
```
telnet 192.168.0.2 80
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c27
```

7) Send the command ‘1’ prefixed with the ASCII representation of the key – i.e. copy & paste the character sequence received at step 6. The robotic arm will start moving in a continuous loop according to the predefined sequence. Note: this is only for demo purpose. Real word applications should not send the key in clear but rather implement TLS or similar encryption mechanism.



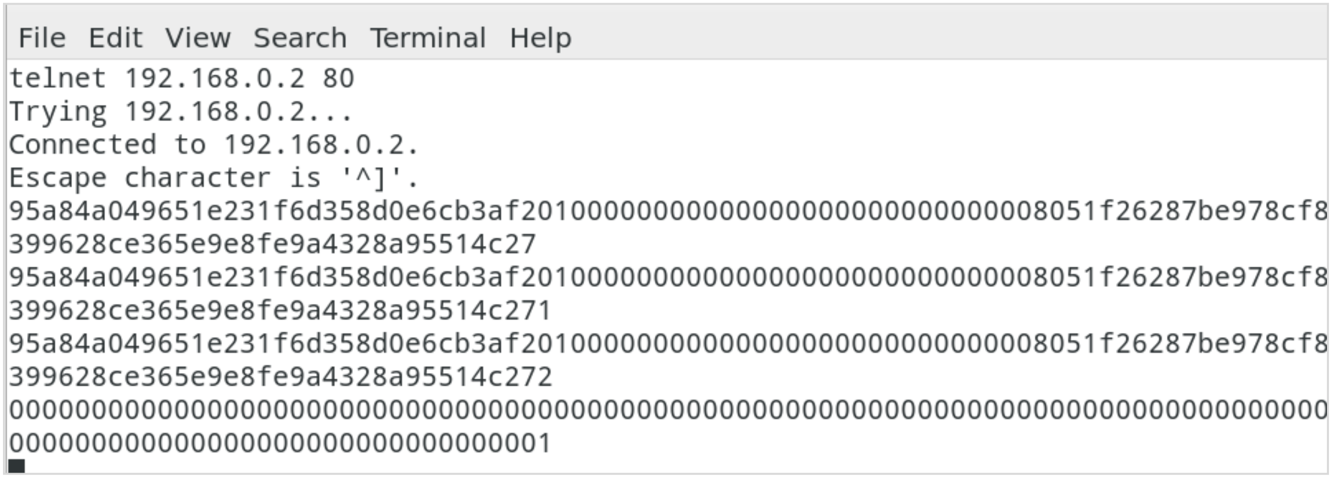
```
telnet 192.168.0.2 80
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c27
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c271
```

8) Send the command ‘2’ prefixed with the ASCII representation of the key. The robotic arm light will immediately turn off to acknowledge the receipt of the stop command. The sequence will pause as soon as the arm reaches the home position. Note: valid commands are ‘1’ for start and ‘2’ for stop. Any other command is ignored.



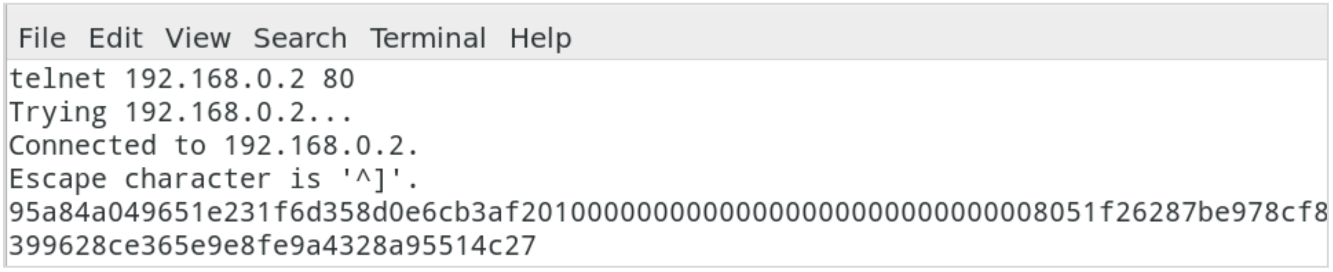
```
telnet 192.168.0.2 80
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c27
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c271
95a84a049651e231f6d358d0e6cb3af20100000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c272
```

9) Send the command ‘1’ prefixed with an invalid key – i.e. a series of ‘0’. The system will ignore the request and the arm will not move.



```
telnet 192.168.0.2 80
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
95a84a049651e231f6d358d0e6cb3af2010000000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c27
95a84a049651e231f6d358d0e6cb3af2010000000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c271
95a84a049651e231f6d358d0e6cb3af2010000000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c272
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000001
```

10) Press SW1 to reboot the board. Open a new telnet session as at step #6 and observe that the system consistently regenerate the same unique key.



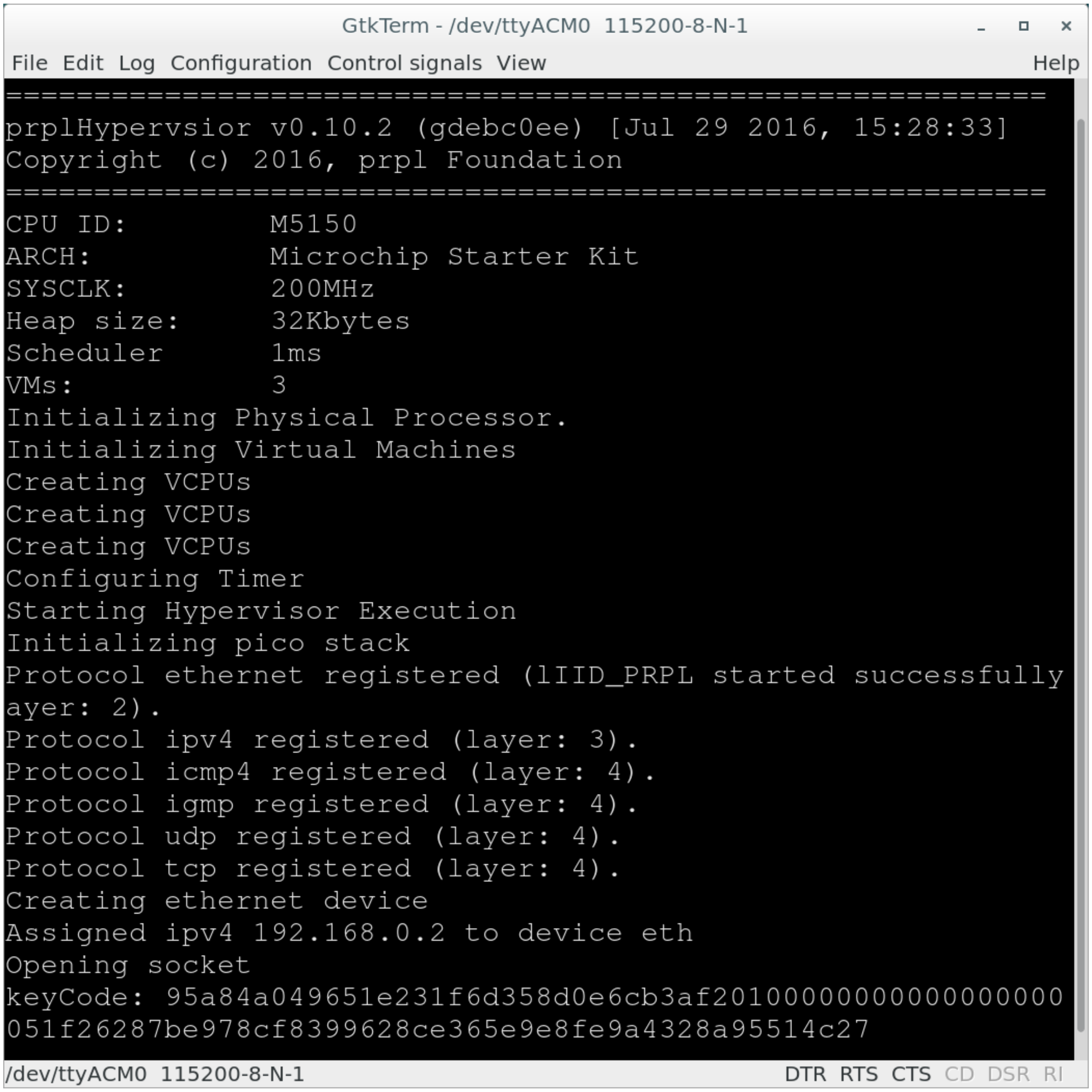
```
telnet 192.168.0.2 80
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
95a84a049651e231f6d358d0e6cb3af2010000000000000000000000000000008051f26287be978cf8
399628ce365e9e8fe9a4328a95514c27
```

11) Repeat the steps above with an identical board to verify that the random generated key is different and unique for each board.

Optional: connect an ENC28J60 based SPI / Ethernet adapter to test virtualized SPI I/O.

PIC32 extension pin	PIC32 I/O pin	enc28j60 pin	function
15	RB3	CS	Chip Select
17	3V3	VCC	3.3V
19	RF5	SI	Data Master -> Slave
21	RF4	SO	Data Slave -> Master
23	RD1	SCK	Serial clock
25	GND	GND	Ground

Optional: connect a UART terminal to J11 to see the console output for hypervisor and virtual machines – 115200/8/N/1.


GtkTerm - /dev/ttyACM0 115200-8-N-1
File Edit Log Configuration Control signals View Help
/dev/ttyACM0 115200-8-N-1
DTR RTS CTS CD DSR RI

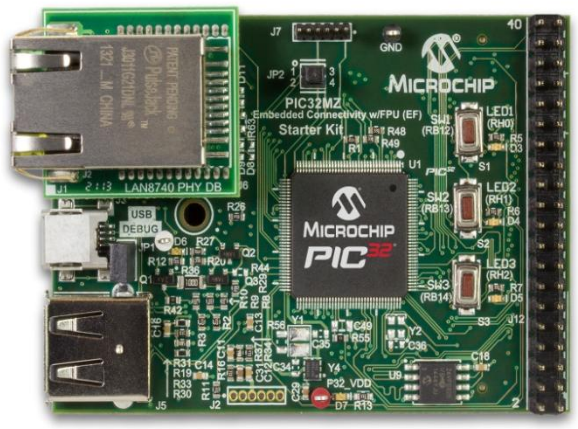
```
===========================================================
prplHypervsior v0.10.2 (gdebc0ee) [Jul 29 2016, 15:28:33]
Copyright (c) 2016, prpl Foundation
===========================================================
CPU ID:        M5150
ARCH:          Microchip Starter Kit
SYSCLK:        200MHz
Heap size:     32Kbytes
Scheduler      1ms
VMs:           3
Initializing Physical Processor.
Initializing Virtual Machines
Creating VCPUs
Creating VCPUs
Creating VCPUs
Configuring Timer
Starting Hypervisor Execution
Initializing pico stack
Protocol ethernet registered (lIID_PRPL started successfully
ayer: 2).
Protocol ipv4 registered (layer: 3).
Protocol icmp4 registered (layer: 4).
Protocol igmp registered (layer: 4).
Protocol udp registered (layer: 4).
Protocol tcp registered (layer: 4).
Creating ethernet device
Assigned ipv4 192.168.0.2 to device eth
Opening socket
keyCode: 95a84a049651e231f6d358d0e6cb3af20100000000000000000
051f26287be978cf8399628ce365e9e8fe9a4328a95514c27
```

Hardware Requirements

- Laptop or desktop computer running Linux or Windows with at least one USB port and one Ethernet port available. Optional: RS232 port or additional USB port to connect the prplHypervisor™ console.

- PIC32MZ-EF Embedded Connectivity Starter Kit board and cables.

  http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dm320007

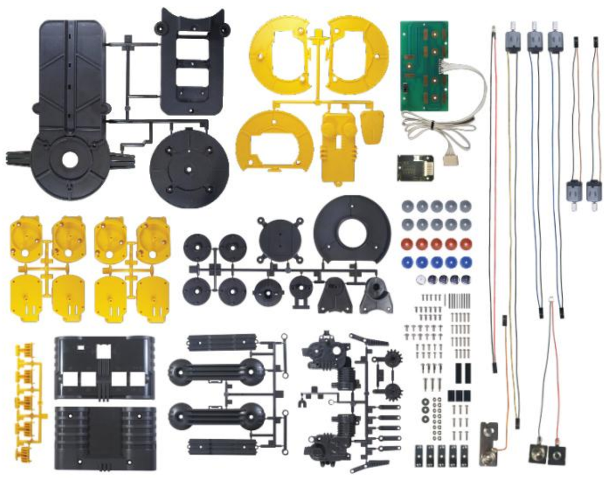


- Optional: ENC28J60 based SPI / Ethernet adapter and 6 female-to-female jumper wires.

  http://www.newegg.com/Product/Product.aspx?Item=9SIA5W02EF6845

- OWI-535 Robotic Arm Kit with USB interface – 4 "D" batteries not included.

  http://www.owirobot.com/robotic-arm-edge-1/



- USB Interface for Robotic Arm Edge.

  http://www.owirobot.com/products/USB-Interface-for-Robotic-Arm-Edge.html

Software Requirements

- prplHypervisor™ demo-july-2016
- Virtualized picoTCP library
- MIPS-MTI toolchain
- srecord software package
- Ubuntu 14.04 LTS or Debian 8 or Windows 7 and above
- Microchip MPLAB IDE or IPE – only required to flash the prpl bootloader

prplHypervisor™ API

The folder prplHypervisor/bare-metal-apps/apps includes a few sample applications – i.e. uart.c, blink.c, ping.c, and others. The hypervisor is currently configured to create 3 bare-metal applications (virtual machines). Each virtual machine has 64Kb of SRAM and 128Kb of flash available. These parameters are defined in the config.h file and can be changed according to requirements. VM applications are defined in the prplHypervisor/Makefile in the APP_LIST section. The following example generates a new virtualized system for three bare-metal applications: uart.c, arm-control.c, and blink.c.

```
 #List of bare-metal applications

#APP_LIST = tcp-listener iidprpl arm-control

APP_LIST = uart arm-control blink
```

Timers

By default the hypervisor scheduler tick is 10ms. Each VM code runs for 10ms before being preemptively switched. During this time the VM receives timer interrupts every 1ms. The function irq_timer() handles these interrupts at the VM level.

UART access

printf() and getchar() default to UART2 (J11 connector). The serial_select() function allows switching to UART6 (PIN 12 TX and PIN 11 RX of the 40 pin connector):

int32_t **serial_select**(uint32_t serial_number)

Valid values for serial_number are UART2 and UART6.

Secure Inter-VM Communications

The prplHypervisor™ implements the prplSecureInterVM™ API for secure inter-VM communications. The current implementation of the API provides the following two send / receive functions:

- int32_t **ReceiveMessage**(uint32_t *source, char* buffer, uint32_t bufsz, uint32_t block);
- uint32_t **SendMessage**(uint32_t target, char* buffer, uint32_t size);

The hypervisor assigns a unique ID to each VM during initialization. The VM ID is assigned according to the APP_LIST section of the Makefile. The first VM in the list is assigned ID 1, the second ID 2 and so on. The VM ID is the source/target parameter used in the above API. Each VM can discover its ID by calling hyp_get_guest_id().

The ReiceveMessage() hypercall returns greater than 0 if a message was received. The data is copied to the buffer limited to bufsz bytes. Zero means no data received. The SendMessage() hypercall returns the number of bytes sent. Zero or less means error.

Errors codes:

- **MESSAGE_VCPU_NOT_FOUND** (destination not found)
- **MESSAGE_FULL** (destination queue full)
- **MESSAGE_TOO_BIG** (message to big)
- **MESSAGE_VCPU_NOT_INIT** (the target VPCU is not ready for communication)

ReceiveMessage() blocks until a message is received. Specify block=0 to execute in non-blocking mode.

The bare-metal application Ping-Pong is an example of inter-VM communications. VM #1 (ping) send a message to VM #2 (pong). Upon receipt of the message, VM#2 sends the message back to VM #1. To build the Ping-Pong example, modify the APP_LIST section of the prplHypervisor/Makefile as follows:

```
APP_LIST =  ping pong
```

prplHypervisor™ API

The current implementation of the execution environment includes the following Unix style functions:

- int8_t *strcpy(int8_t *dst, const int8_t *src);
- int8_t *strncpy(int8_t *s1, int8_t *s2, int32_t n);
- int8_t *strcat(int8_t *dst, const int8_t *src);
- int8_t *strncat(int8_t *s1, int8_t *s2, int32_t n);
- int32_t strcmp(const int8_t *s1, const int8_t *s2);
- int32_t strncmp(int8_t *s1, int8_t *s2, int32_t n);
- int8_t *strstr(const int8_t *string, const int8_t *find);
- int32_t strlen(const int8_t *s);
- int8_t *strchr(const int8_t *s, int32_t c);
- int8_t *strpbrk(int8_t *str, int8_t *set);
- int8_t *strsep(int8_t **pp, int8_t *delim);
- int8_t *strtok(int8_t *s, const int8_t *delim);
- void *memcpy(void *dst, const void *src, uint32_t n);
- void *memmove(void *dst, const void *src, uint32_t n);
- int32_t memcmp(const void *cs, const void *ct, uint32_t n);
- void *memset(void *s, int32_t c, uint32_t n);
- int32_t strtol(const int8_t *s, int8_t **end, int32_t base);
- int32_t atoi(const int8_t *s);
- int8_t *itoa(int32_t i, int8_t *s, int32_t base);
- int32_t puts(const int8_t *str);
- int8_t *gets(int8_t *s);
- int32_t abs(int32_t n);
- int32_t random(void);
- void srand(uint32_t seed);
- int printf(const int8_t *fmt, ...);
- int sprintf(int8_t *out, const int8_t *fmt, ...);
- void udelay(uint32_t usec);
- void putchar(int32_t value);
- uint32_t getchar(void);

prplPUF™ API

The IID_PRPL library is the key management module that generates and securely stores a unique hardware bound cryptographic key based on SRAM PUF.

With this key other keys or data can be securely stored. Instead of storing the key in tamper resistant Non-Volatile Memory (NVM, typically secure EEPROM) or even hard-wiring it into the encryption core, the IID_PRPL component reconstructs the key on the fly/startup. In order to reconstruct the key, IID_PRPL needs an activation code and device specific SRAM values. The activation code is generated during the enrollment phase, which can be performed by using IID_PRPL API. When the device key is not needed anymore it can be removed from memory. When it is needed at a later stage it can be reconstructed again.

Terminology: A KeyCode is a data package that is encrypted and authenticated with keys derived from the PUF. It is created by calling the IID_PRPL_WrapKey() function, which performs a wrapping operation. The data embedded in the KeyCode can only be reconstructed (unwrapped) by calling the IID_PRPL_UnwrapKey() function, and by using the same device and application as was used during the execution of the IID_PRPL_WrapKey() function.

Order of operations

1. After power-up or reset, IID_PRPL starts in an uninitialized state. It must first be initialized by calling IID_PRPL_Init() function.
2. In order to use other IID_PRPL functions, a device has to be enrolled. This can be achieved by calling IID_PRPL_Enroll(), which generates a device-specific activation code.
3. Once a device is enrolled, IID_PRPL_Start() is called to enable IID_PRPL's key management functions.
4. If key management functions are not required anymore, IID_PRPL_Stop() should be called in order to remove the internal device key from memory.

Notes

- IID_PRPL does not allocate any memory. It is the responsibility of the caller to allocate all required buffers with the correct sizes.
- The SRAM used by IID_PRPL must not vary in location and size.
- Access to the SRAM needs to be limited to IID_PRPL only in order to ensure the security benefits.
- After enrollment, the activation code must be stored by the caller (e.g. in Non Volatile Memory).
- If the activation code is corrupted, previously generated key codes can no longer be unwrapped.
- Re-enrollment of a device is possible, however previously generated key codes can no longer be unwrapped.

Code example

One of the functions of PUF library, IID_PRPL_GetAuthenticationResponse(), can be used to uniquely identify a specific device. In order to be able to uniquely identify a device, an authentication key needs to be used. The authentication key can be generated either by an authentication server or by the device. In the latter case, this key must be transferred (securely) to the before mentioned server. The authentication key should also be stored on the device by wrapping it into a KeyCode with the KEY_PROP_AUTHENTICATION property set.

Every time a device needs to be identified, the authentication server should generate a new random challenge. This challenge is then transferred to the device. On the device, the challenge, together with the stored authentication KeyCode, should be fed into the IID_PRPL_GetAuthenticationResponse() function, which computes a response. This response should then be transferred to the authentication server.

Finally on the authentication server the device response can be verified by comparing it to the response generated on the server by using the same cryptographic function. Intrinsic-ID provides a separate server library that implements the verification function.

VM #2 uses IID_PRPL library's functions. It initializes the library, and it enrolls the device if it has not been enrolled, i.e. if the activation code is not stored in Flash.

At a later stage, IID PRPL VM reconstructs the device internal key in order to enable IID_PRPL key management's functions.

If the startup succeeds, IID PRPL VM starts handling messages received from other VMs that want to use PUF's functionality (by means of PUF library/API).

Limitations of the open source version of the library

The IID_PRPL library is intended only for demo and evaluation purposes. It must not be used in production environments. Commercial applications should use Intrinsic-ID proprietary implementation - Quiddikey.

- IID_PRPL library maximum data/keys size is limited to 88 bytes
- IID_PRPL library authentication functionality is limited to 64 bytes
- IID_PRPL library KeyCodes generated by different devices are not guaranteed to be entirely different
- IID_PRPL library does not perform comprehensive checks on inputs/outputs parameters
- IID_PRPL library is not reset-proof: if the initialization fails the device has to be restarted

prplPUF™ API

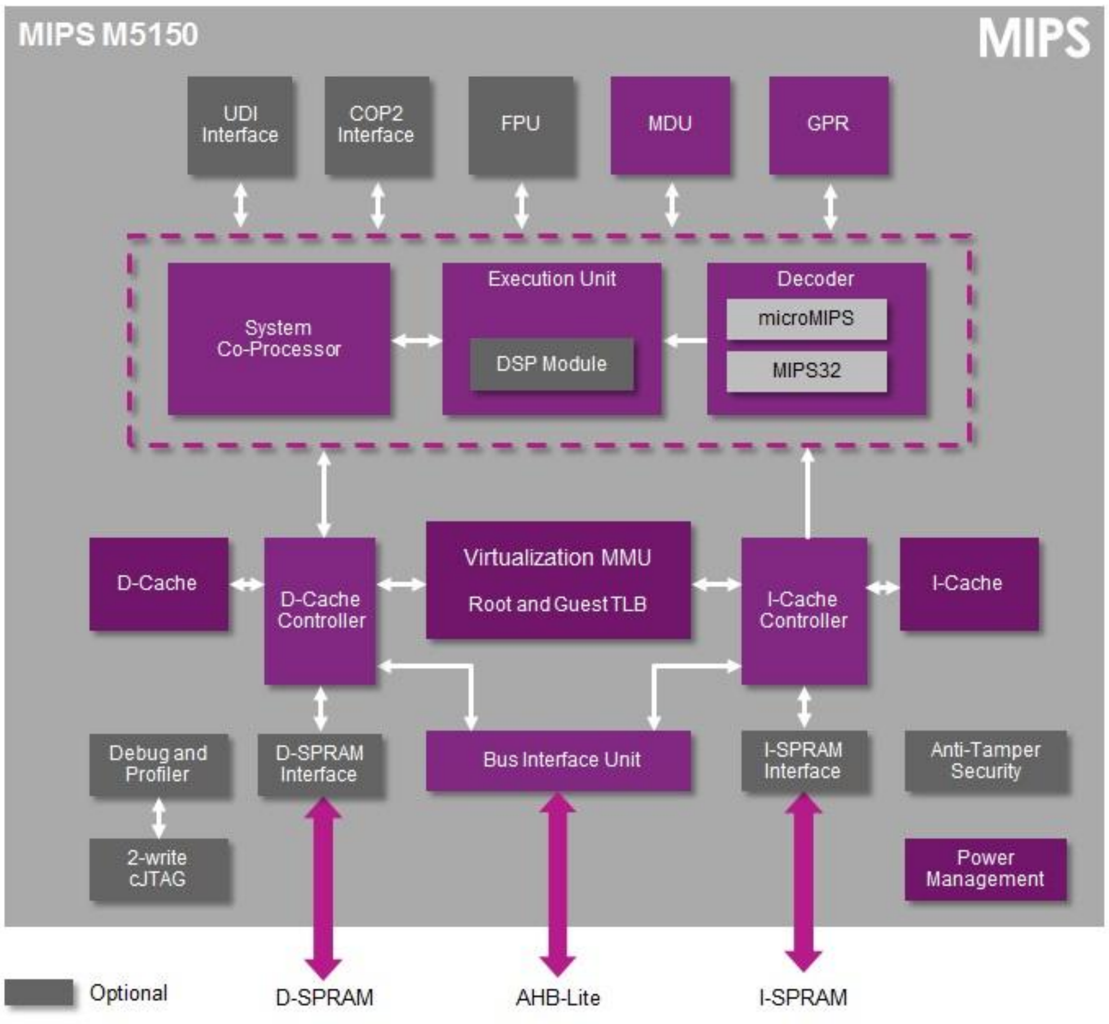
- IID_PRPL_GetSoftwareVersion()
- IID_PRPL_Init()
- IID_PRPL_Enroll()
- IID_PRPL_Start()
- IID_PRPL_Stop()
- IID_PRPL_WrapKey()
- IID_PRPL_UnwrapKey()
- IID_PRPL_GetAuthenticationResponse()

See prpl-hypervisor/bare-metal-apps/include/iidprplpuf.h header file for more information about this API.

Appendix – MIPS M5150 Technology Overview

The MIPS M-class M5150 CPUs IP core with integrated DSP and FPU engines deliver leading performance efficiency and a unique feature set for MCU and embedded markets including M2M, IoT and embedded control. The M5150 supports Linux class operating systems in addition to maintaining the real-time/low latency characteristics needed to run an RTOS on the same system. MIPS M-class CPUs deliver 2x higher DSP performance and higher overall performance/MHz versus the competition – with a free, industry-standard GCC compiler.

Importantly, these CPUs are the only embedded class CPUs with hardware virtualization technology. With hardware virtualization, you can build a controller that can run multiple, unmodified, operating systems and applications independently and securely at the same time on a single, trusted platform. You can use this feature to develop systems that provide a secure path to deliver updates/downloads, and benefit from enhanced IP protection.



M5150 Key Features

- A powerful DSP engine provides high performance, single cycle throughput DSP and SIMD capabilities to address the requirements of such applications as industrial/ machine control, voice processing and more.
- These IP cores deliver 2x higher DSP performance and higher overall performance/MHz versus the competition using a free, industry-standard GCC compiler – achieving leading 3.56 CoreMark/MHz and 1.7 DMIPS/MHz performance.
- Anti-tamper features provide an additional layer of security against potential external attacks including a secure debug feature that prevents external debug probes from accessing the core internals so an application's code stays safe and secure.
- An optional IEEE 754 floating point unit (FPU) provides high-performance support of both single and double precision instructions for accelerated real-time control in industrial, automotive and digital consumer applications. Double precision support not only leads to greater accuracy, but also provides an advantage in running a wide range of today's software algorithms.
- The microMIPS instruction set architecture provides up to 30% code compression for applications where memory size is critical. This ISA improves code density while maintaining a performance equivalent to MIPS32 mode.
- You can run Linux class operating systems on the M5150, which has required features such as a Translation Lookaside Buffer (TLB) MMU that are unique among other MCU-class CPUs. While supporting Linux, other unique features enable the M5150 to maintain the real-time/low latency characteristics needed to run an RTOS on the same system.
- The M5150 CPUs is the only embedded class CPUs with hardware virtualization technology. This means you can build a design with multiple secure domains where each application or OS can run independently and reliably in its own separate, trusted environment. You can use this feature to develop systems that provide a secure path to deliver updates/downloads, and benefit from enhanced IP protection.

Appendix – picoTCP Technology Overview

picoTCP is the reference TCP/IP Stack for The Internet of Things:

Feature complete, easy to extend, easy to use

picoTCP is feature complete with all major connectivity protocols supported. Yet its modular architecture enables rapid development of new protocols if needed. Its POSIX API integration allows easy integration with any real time embedded Operating system.

Production Grade Quality

Our QA team is checking the RFC- compliance of the stack 24/7, using their automated RFC-compliance test framework. Next to this, our engineers monitor their code quality using industry grade static analysis and dynamic analysis tools. Read more on our quality approach here: www.picotcp.com/picotcp-testing

Secure

A fully functional integration with wolfSSL provides Transport Layer Security (TLS) and cryptographic protocols adequate for industry-grade secure communication.

Small Footprint

Offering the possibility to easily select only the features needed for your application, picoTCP enables you to keep the footprint of your connectivity stack to a bare minimum.

Active and Industry supported community

The picoTCP community is industry-backed by Intelligent Systems / Altran, a worldwide leader in embedded software development. This ensures professional support and continuity in development.

Open source

We are strong believers in open source software and we support the integration with other open source communities. Furthermore, it allows everybody, including our customers, to have full insight into the code. That is why picoTCP is also available with an open source license.

Industry Recognition

picoTCP is recognized as an industrial reference in connectivity stacks. Amongst others, the adoption of the stack by Honeywell Customized Comfort Products is a clear proof of this. More info here

More info at http://www.picotcp.com

Appendix – Intrinsic-ID Technology Overview

Intrinsic-ID is a company that provides products and solutions for embedded security.

The products are based on the unique and patented technology called SRAM Physical Unclonable Function or SRAM PUF. The technology uses standard SRAM memory, which is available on any microcontroller or processor. A unique device fingerprint is extracted by reading the uninitialized startup data from a small region of SRAM. This fingerprint stems from deep submicron process variations at manufacturing that cannot be controlled or copied and hence is unclonable.

The fingerprint is used to reconstruct a reliable and device-unique cryptographic root key, which is invisible to attackers. It is only present when the device is switched on. The root key can be used for further cryptographic services such as key wrapping, authentication and encryption.

The PUF-based secure key management technology can be licensed from Intrinsic-ID and is delivered in hardware IP or firmware as the Quiddikey product. Quiddikey is available in different versions and optionally includes the most widely used cryptographic algorithms. Also available is a true random number generator based on the SRAM PUF.

Besides PUF-based key management solutions, Intrinsic-ID offers solutions to protect the electronics supply chain. These range from solutions for tracking and monitoring of chips for anti-counterfeiting purposes, to flexible key provisioning and authentication solutions. These products and solutions can be applied to all modern chips, microcontrollers and CPUs without making a change to the hardware. Currently this technology is being used by customers in the field to protect the most sensitive payment, content, sensor and government data and systems.