Zephyr® Project
Developer Summit

LLEXT: Extending Zephyr at Runtime

Tom Burdick, Intel Corporation

#EmbeddedOSSummit @tomb

What is LLEXT?

Linkable Loadable Extensions

Extensions are runtime changeable behavior

Set of tooling to build, manage, and link extensions

An ELF loader like...

- Linux Kernel Modules
- Linux Userspace Programs

Richard Masoner - Lloyd the Llama CC BY-SA 2.0

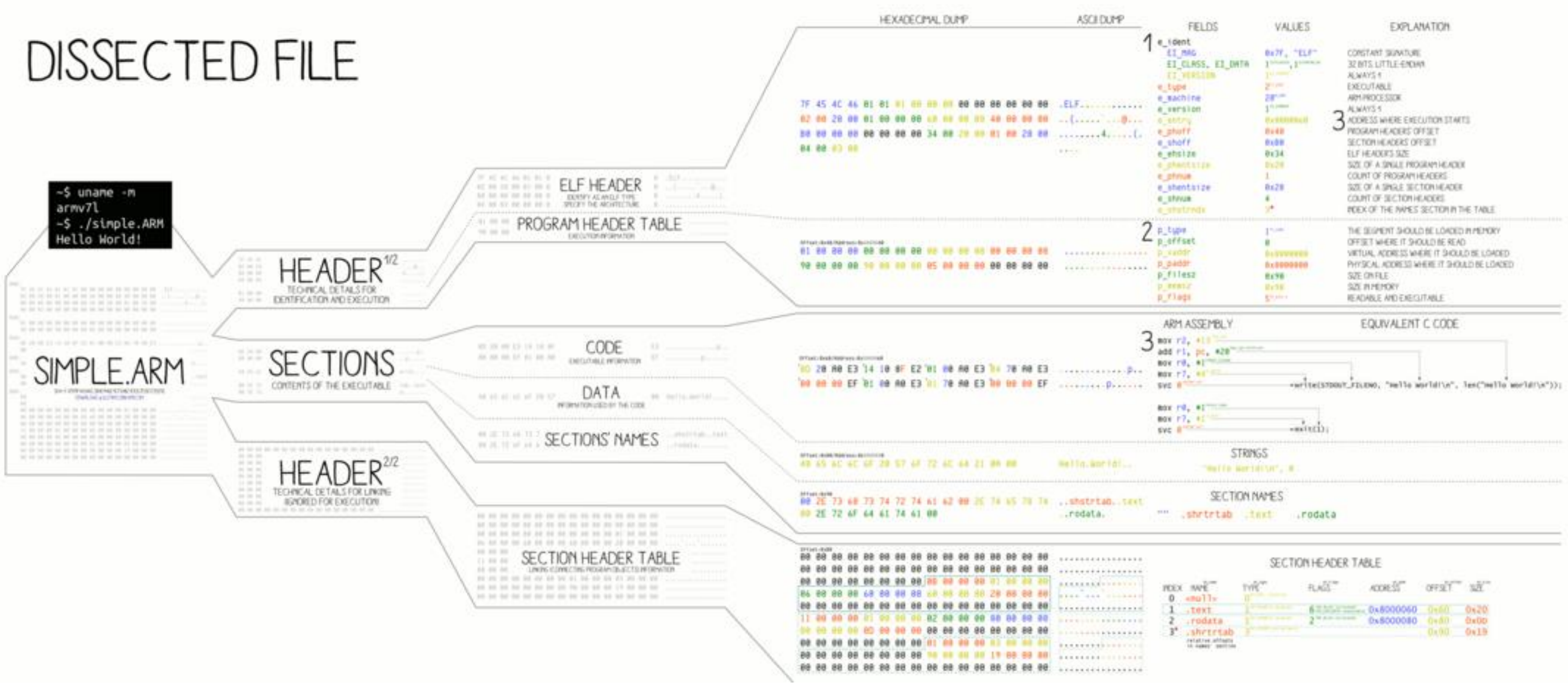
Agenda

- Extensions and Memory Management
- Relocations
- Building Extensions
- Use cases
- Q/A

ELF 101 a ~~Linux executable~~ walkthrough

ANGE ALBERTINI
CORKAMI.COM

Zephyr Extension

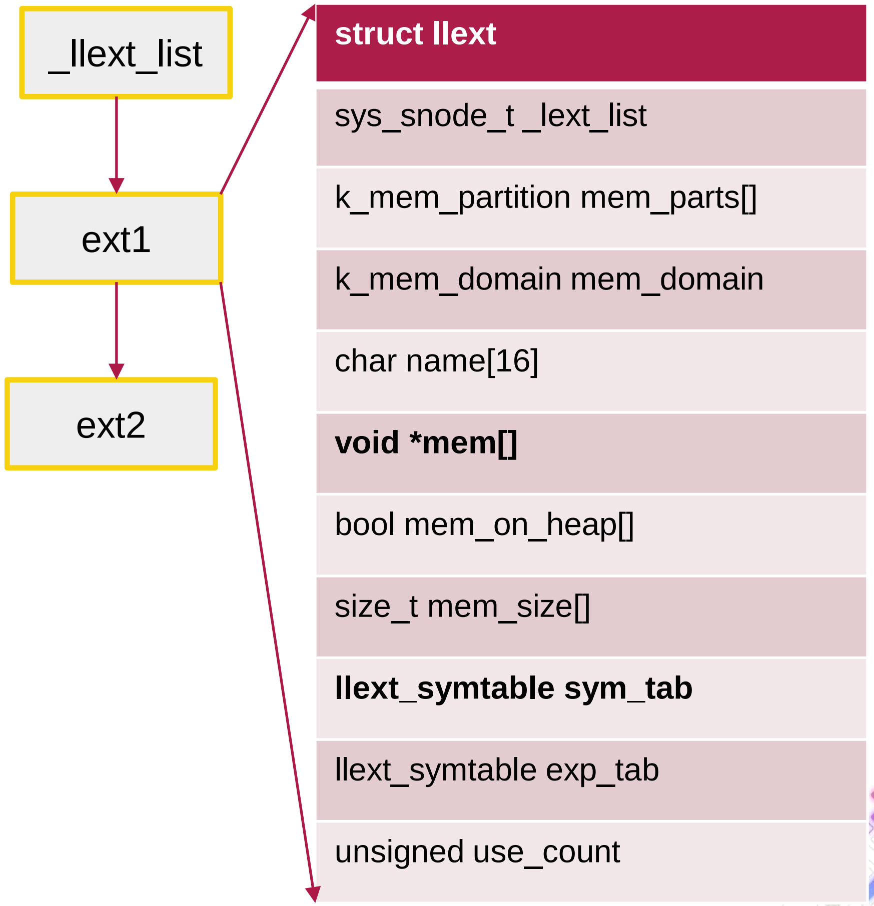


```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

EMBEDDED OPEN SOURCE SUMMIT

LLEXT: Extension manager

- llext_load(), llext_unload(), llext_find_sym()
- Maintains a list of loaded extensions
- Maintains a reference count for each extension
- Manages memory for extensions
- Manages symbol tables for the base firmware and each extension



LLEXT: Memory Manager

- Extensions need memory
- Can be directly referenced in some cases or allocated on a LLEXT dedicated heap
- Memory permissions need to be set with MPU/MMU in use
- Metadata for the extension
- .text needs read+execute
- .data, .bss needs read+write
- .rodata needs read

Metadata – read + write + supervisor
.text – read + execute + user + supervisor
.data – read + write + user + supervisor
.bss – read + write + user + supervisor
.rodata – read only + user + supervisor

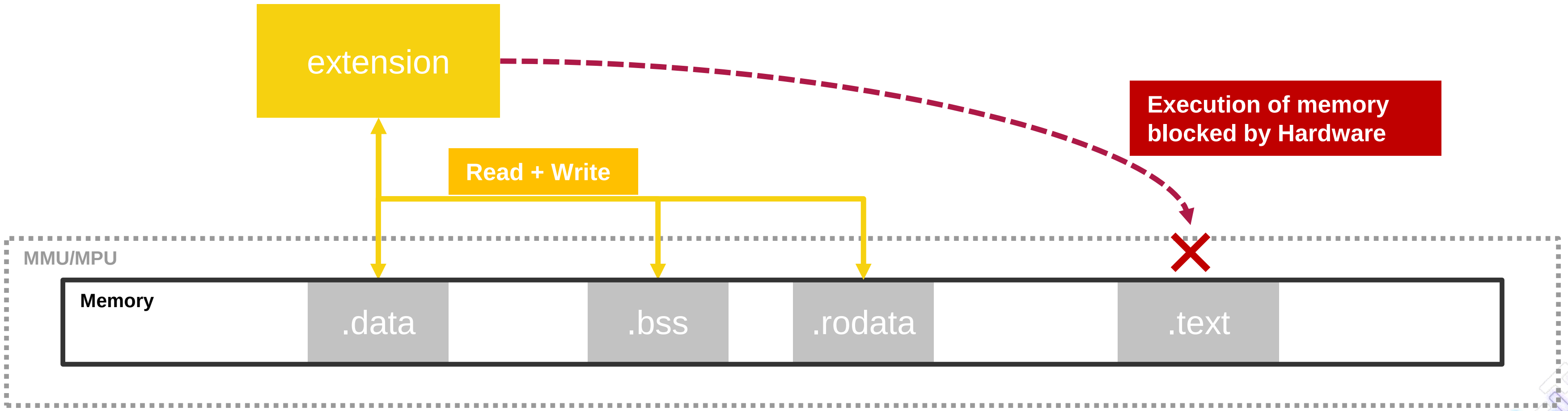
LLEXT: Memory Protection

CONFIG_MPU/CONFIG_MMU **AND**
CONFIG_USERSPACE

OR

NOT CONFIG_MPU/CONFIG_MMU

Memory, with MPU/MMU, is not executable outside of original .text in base firmware!



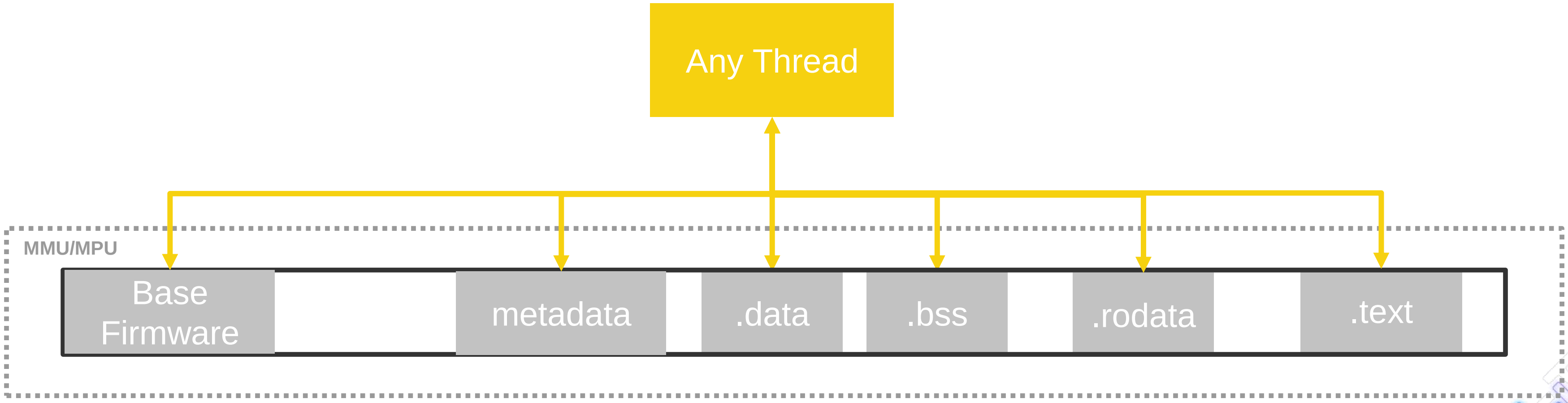
LLEXT: Kernel space execution

NOT CONFIG_MPU/CONFIG_MMU

OR

Kernel mode thread calls into extension code

All memory is accessible



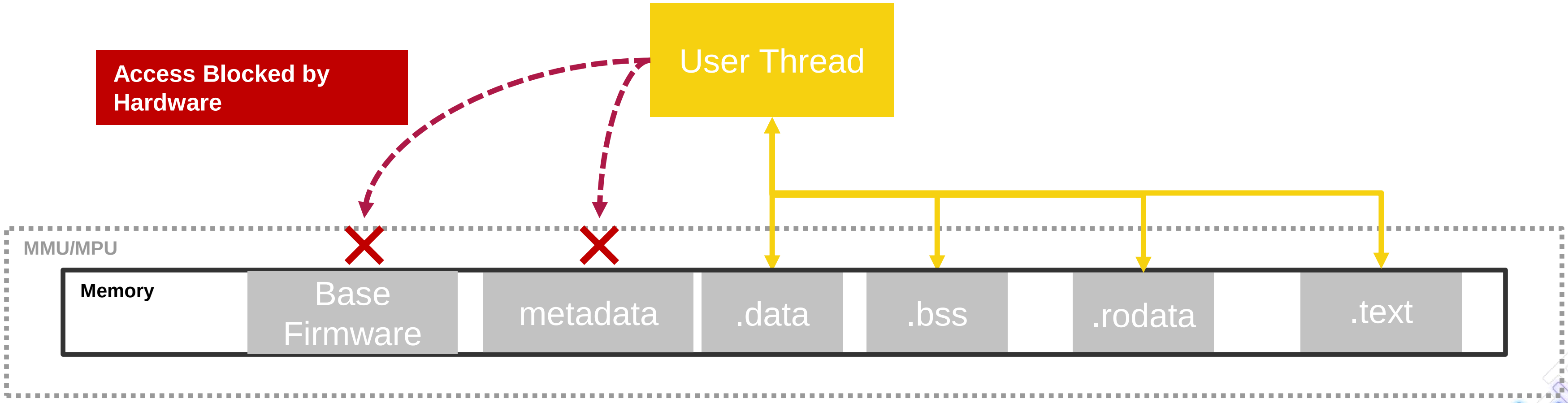
LLEXT: User space execution

CONFIG_MPU/CONFIG_MMU **AND** CONFIG_USERSPACE

User thread assigned extension domain

A Process is Born!

Only extension code, and data is accessible! Syscalls may be used.



Relocations: Linking and Placement

arm-zephyr-eabi-objdump -r -d -x hello_world.elf

```
Disassembly of section .text:

    00000000 <hello_world>:
      0:   b580           push    {r7, lr}
      2:   af00           add     r7, sp, #0
      4:   4b08           ldr     r3, [pc, #32]   ; (28 <hello_world+0x28>)

     28:   00000004       .word   0x00000004
                      28: R_ARM_ABS32 .rodata
     2c:   00000000       .word   0x00000000
                      2c: R_ARM_ABS32 printk
     30:   00000014       .word   0x00000014
                      30: R_ARM_ABS32 .rodata
     34:   4718           bx      r3
     36:   46c0           nop                     ; (mov r8, r8)
```

Rewrite with address of .rodata + 0x00000004

Rewrite with address of printk function

EMBEDDED OPEN SOURCE SUMMIT

Relocations: Some key Points

- Kinds of Relocations that show up depend on
  - Architecture: x86, armv7m, xtensa lx7, etc
  - ELF linkage: shared, static, relocatable
  - Compiler flags: e.g. -mlong-calls
- May require opcode decoding, updating, and reencoding
- May require generation of a jump table
  - Opcodes are sometimes location dependent
  - Limits range of address accessibility for that opcode
  - E.g. PC relative call instructions encoded as 16bit instruction opcode

Relocations: Finding symbolic locations

- Sections may be interdependent (.text needs data from .data/.bss/.rodata)
- Symbolic linking requires a name and address pair to be found
- Relocation instructions are in ancillary ELF sections
  - .rel.text
  - .rel.bss
  - etc

Loading an ELF, High Level

1. ELF header and sections are read
2. Memory is Allocated or Referenced
3. Relocations are applied
4. Extension added to global list

Building an Extension

Manually, great for exploring and tinkering

```
arm-zephyr-eabi-gcc -mlong-calls -mthumb -c -o hello_world.elf tests/subsys/llext/hello_world/hello_world.c
```

Using Zephyr provided CMake Functions

```
add_llext_target(${ext_name}_ext
     OUTPUT  ${ext_bin}
     SOURCES ${ext_src}
     )
```

Building an Extension: EDK

Extension Developer Kit – PR 69831 – Base Application Source Not Needed

1. Application developer creates an application, with its own API in addition to that of Zephyr;
2. Application developer builds the EDK via west build -t llext-edk;
3. Application developer somehow makes the EDK available to extension developer;
4. Extension developer extracts EDK and includes it in its build system - the EDK provides some files to aid getting CFLAGS for cmake or make;
5. Extension developer can build the extension.

Use Cases – Today and Tomorrow

- Plugins; E.g. audio/video/sensing processing modules/plugins
- Isolated and Updatable Processes; E.g. multi-tenant firmware with base image
- Faster application build and load; Majority of application Logic in an Extension

Caveats Today, but maybe not Tomorrow

- Extension Signing is Needed; Untrusted ELF could do damage
- Debugging at an assembly level is tenable but slow
- Kernel objects need special care
- Dictionary logging; Missing some puzzle pieces
- Execute in Place (XIP) would require some additional work

Credits

LLEXT was created by numerous people, and built on Zephyr infrastructure

- Chen Peng, Initial ELF Loader (Former Intel)
- Lucas Burelli, Armv7m and CMake tooling (Arduino)
- Guennadi Lyakh, Xtensa Support (Intel – Sound Open Firmware)
- Ederson de Souza, EDK (Intel – Zephyr Team)

- Myself, Gluing it together (Intel – Zephyr Team)
- A growing List of others!

Zephyr® Project

Developer Summit