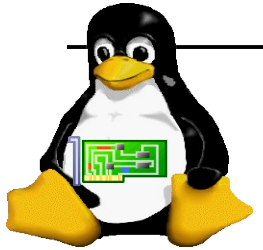
Optimize uClinux for ARM Cortex-M4

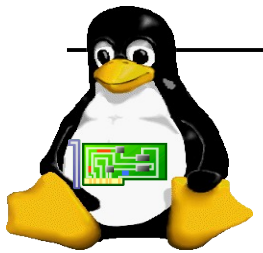
Jim Huang <jserv.tw@gmail.com>
Jeff Liaw <rampant1018@gmail.com>

Mar 23, 2015

Embedded Linux Conference

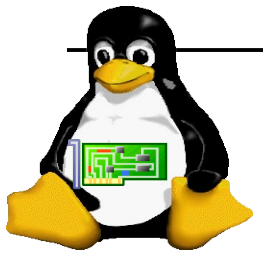


Rights to copy

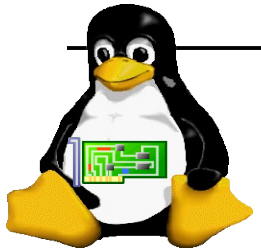


Agenda

- Use uClinux for Your Own Devices
- Evaluate uClinux
  - storage
  - performance
- Optimizations
- Pending Issues

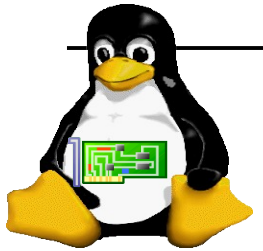


Use uClinux for Your Own Devices



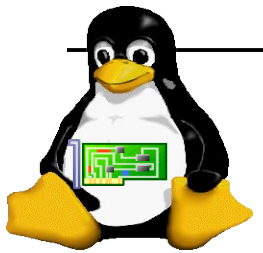
uClinux vs. Linux

- Generally Smaller
- No memory protection
  - Optional MPU
- No virtual memory (demand loading)
  - Can do XIP
- Source level compatibility
  - mmap(), fork, sbrk()
- Flat format instead of ELF



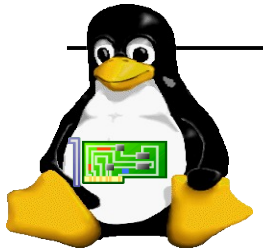
uClinux vs. Bare-Metal

- Linux device drivers
- Linux software stack
- No need for simulations
- Reuse/share code with full Linux
- Use existing technologies



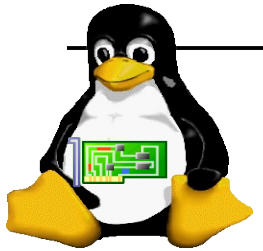
uClinux vs. RTOS

- Free software + open source technologies
- Well-developed applications
- Stable and portable
- (almost) POSIX compliant
- Device driver and software stack (net, usb, bluetooth)



uClinux Advantages

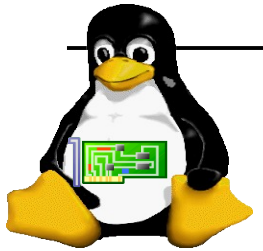
- Cheaper hardware prototyping
- Reusable with full Linux
- Cortex-M share Thumb2 instructions with Cortex-A
- File systems
- Networking



ARM Cortex-M

- built on the ARMv7-M architecture
- Cortex-M3/M4: 1.25 DMIPS/MHz with a 3-stage pipeline, multiple 32-bit busses, clock speeds up to 200 MHz
  - Cortex-M4 adds a range of saturating and SIMD instructions specifically optimized to handle DSP algorithms
- ideal target for uClinux
  - developed for ARM7
  - Faster & more efficient

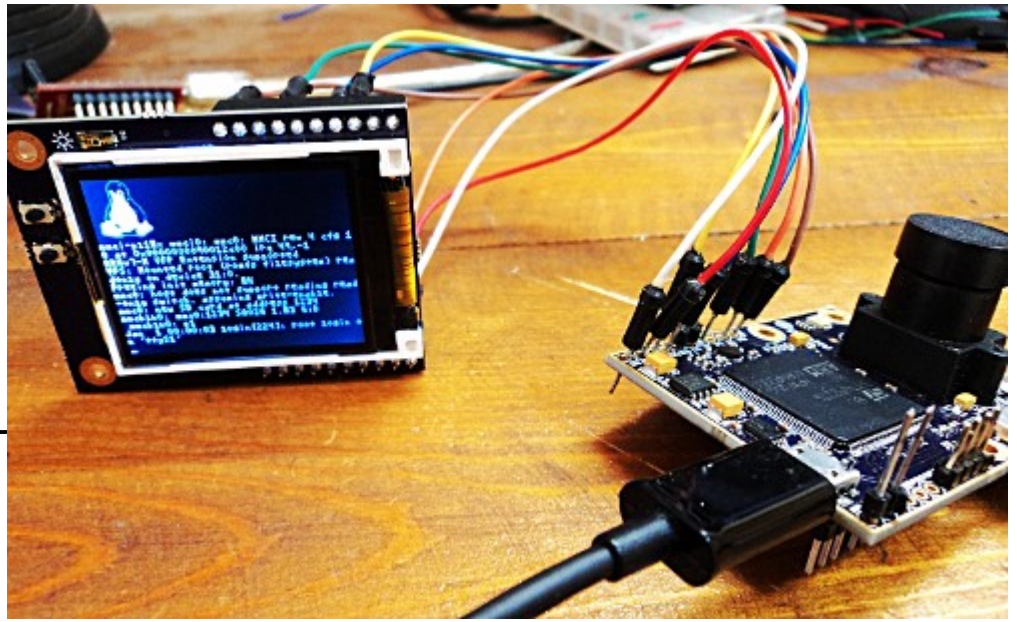
Features	ARM7TDMI-S	Cortex-M3
Architecture	ARMv4T (von Neumann)	ARMv7-M (Harvard)
ISA Support	Thumb / ARM	Thumb / Thumb-2
Pipeline	3-Stage	3-Stage + branch speculation
Interrupts	FIQ / IRQ	NMI + 1 to 240 Physical Interrupts
Interrupt Latency	24-42 Cycles	12 Cycles
Sleep Modes	None	Integrated
Memory Protection	None	8 region Memory Protection Unit
Dhrystone	0.95 DMIPS/MHz (ARM mode)	1.25 DMIPS/MHz
Power Consumption	0.28mW/MHz	0.19mW/MHz
Area	0.62mm2 (Core Only)	0.86mm2 (Core & Peripherals)*

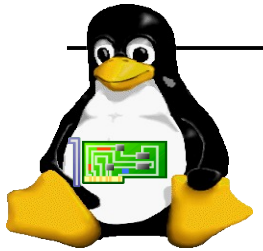


Use Case: OpenMV

low-cost, extensible, Python-powered machine vision modules, that aims at becoming the Arduino of machine vision

- Hardware: STM32F4xx at 168/180MHz
- kernel is configured to XIP
  - not relocated to SDRAM and runs directly from flash
- the 2MBs of flash hosts the u-boot bootloader, kernel image and the romfs
- takes less than 1 sec to boot

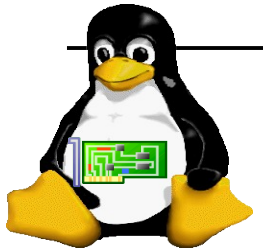




Use Case: OpenMV

Features

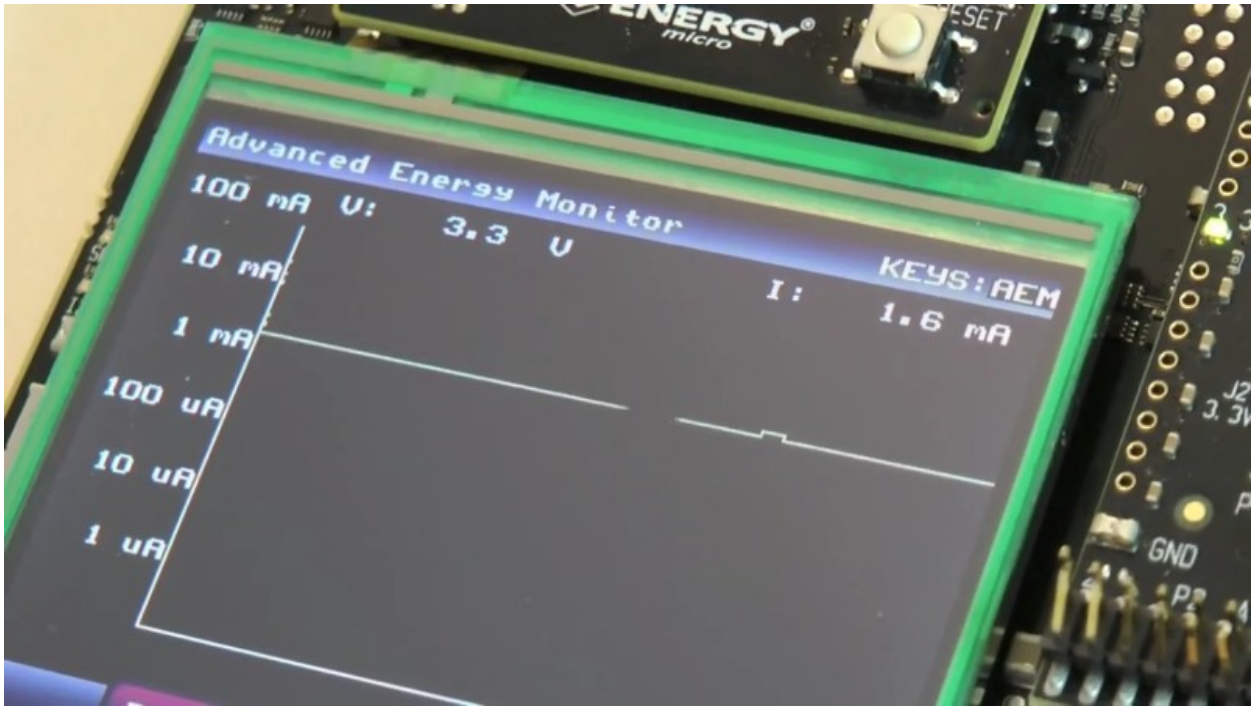
- Scriptable in Python3
- $15 BOM
- 2MP RGB/YUV/JPEG sensor (OV2640)
- Recording/Streaming MJPEG: to SD or via external WiFi shield.
- 16MB SDRAM: on-board enables uClinux to run
- Image processing:
  - object detection; template matching; face recognition
- Wireless expansion: CC3K module from TI

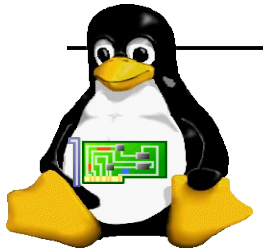


Use Case: EFM32 Giant Gecko

energy sensitive applications with high memory and connectivity requirements

- Cortex-M3: capable to run uClinux, up to 48 MHz
- Measured current when system idle: 1.6 mA



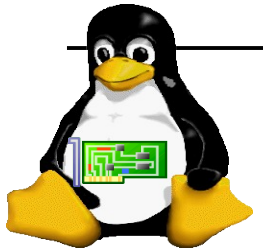


source: https://www.youtube.com/watch?v=3WS3pvsOmp4

Use Case: WiFi Sniffing Encryption

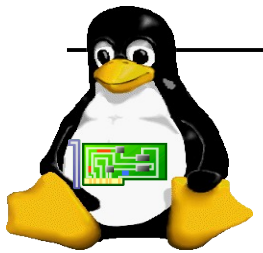
Features

- High security 64/128/256bit WEP Encryption, TKIP, WPA, 802.11
- cross-compile the Aircrck program to Thumb2 for Cortex-M3.
  - To use Aircrack to capture enough packets of a WEP wireless network.
  - After we have captured enough packets we will decrypt the password.



Source: http://wiesel.ece.utah.edu/redmine/projects/wesec/wiki

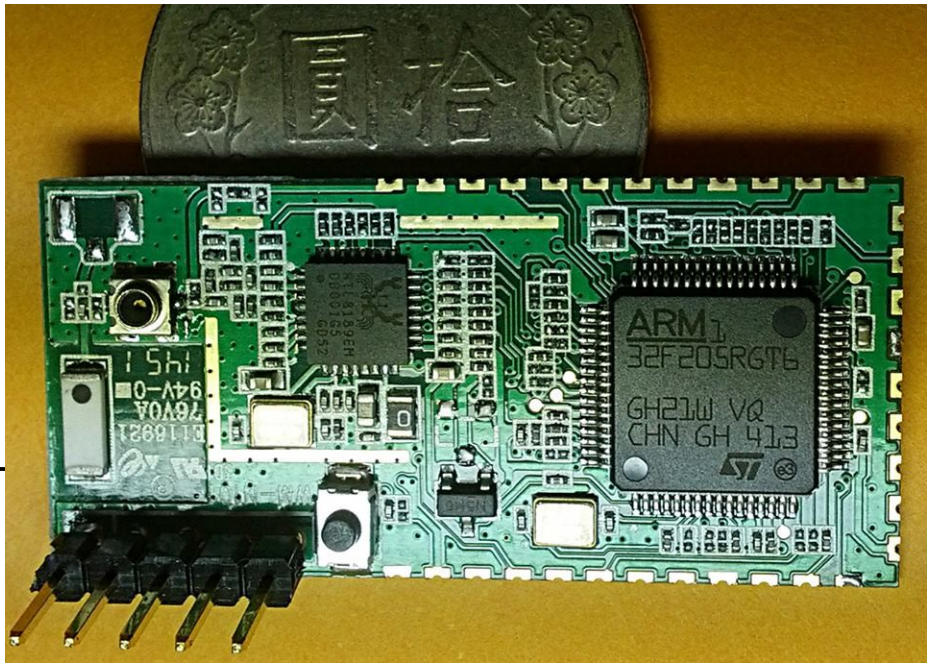
Evaluate uClinux

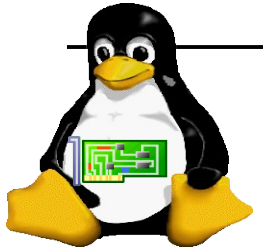


Considerations

- Storage requirements
  - vague & architecture-dependent
  - RAM/persistent, XIP, root file system
- Performance issues
  - Boot time, latency, overhead
  - Application optimizations

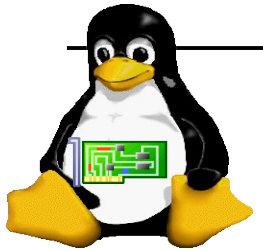
Reference design:
STM32F2 + Realtek 802.11b/g
small formfactor





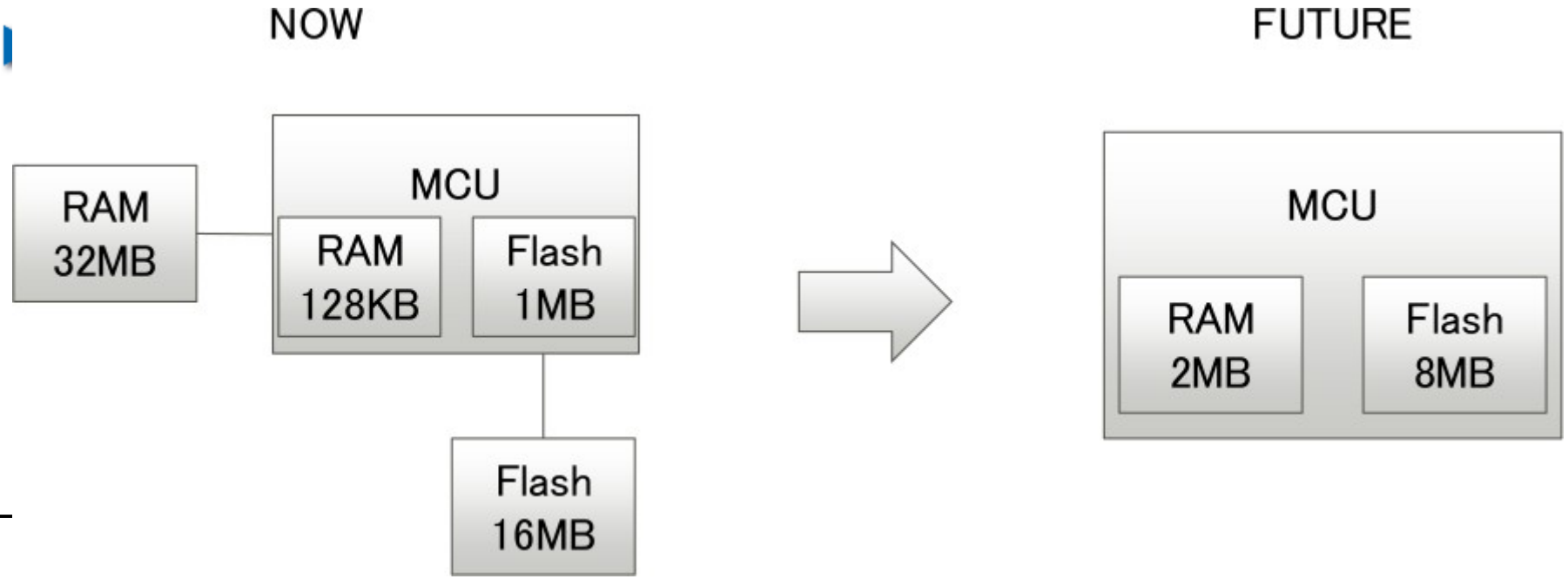
Storage Requirements

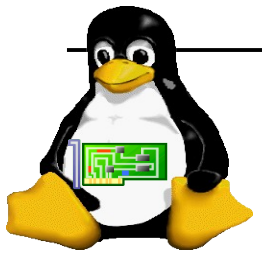
- storage estimations:
  - RAM: 8+ MB (+something for userspace)
  - Persistent: 2+ MB (+some for filesystem)
- Estimations for XIP
  - RAM: 1+ MB (+something for userspace)
  - Persistent: 4+ MB (+some for filesystem)
- XIP support for ARM



eXecute In Place

- .text segment can reside in flash memory and need not be copied to RAM at all
- Kernel XIP
  - General setup->Kernel Execute-In-Place from ROM (=y)

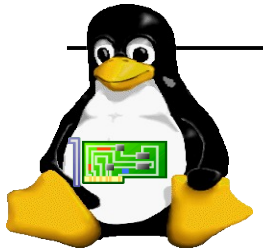




Source: Running uClinux on ARM Cortex-M3 Platform, Fujitsu Computer Technologies Limited

External SRAM

- STMicro STM32F4xx
  - up to 256 KB RAM
  - up to 2MB flash
- In addition, external SRAM is supported
  - STM32F429i Discovery is equipped with 8MB SDRAM chip
  - SDRAM memory chip is connected to SDRAM bank 2 of Flexible Memory Controller of STM32F429 MCU.
    - SDRAM memory can be used as frame buffer for big LCDs (up to 800x600).

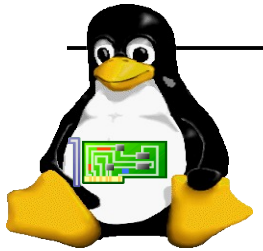


Preferable Design

- Hardware configurations (STM32F429i Discovery)
  - ARM Cortex-M4!
  - only SRAM, no DRAM!
  - only NOR flash (possibly uSD/eMMC)
- Kernel size: 700 KB XIP image

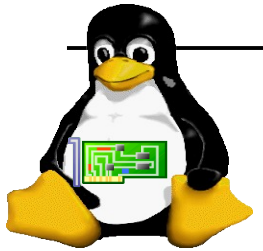
Source tree: https://github.com/uclinux-cortexm

```
size vmlinux
   text       data        bss        dec        hex   filename
 656818      66656      51856     775330     bd4a2   vmlinux
```



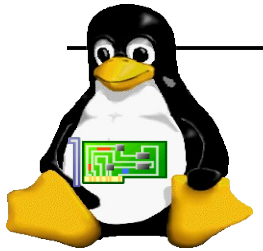
Linux Root Filesystem

- Root filesystem: XIP/Squashfs!
  - executables come uncompressed
- The whole size reaches 900 KB for internet-enabled applications
- Need extra 64 KB for JFFS2 filesystem

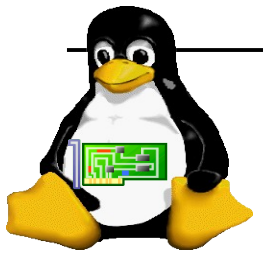


Performance Issues

- Always remember: CPU frequency is generally < 200 MHz
- Boot time
- Latency
  - Generally higher than RTOS
- Overhead
  - Generally higher than RTOS
- But... applications might benefit from the optimizations done by open source communities
  - We will discuss later

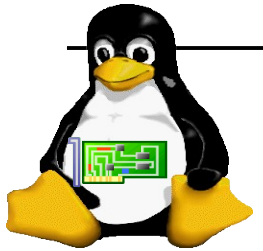


Optimizations: Interrupt Latency

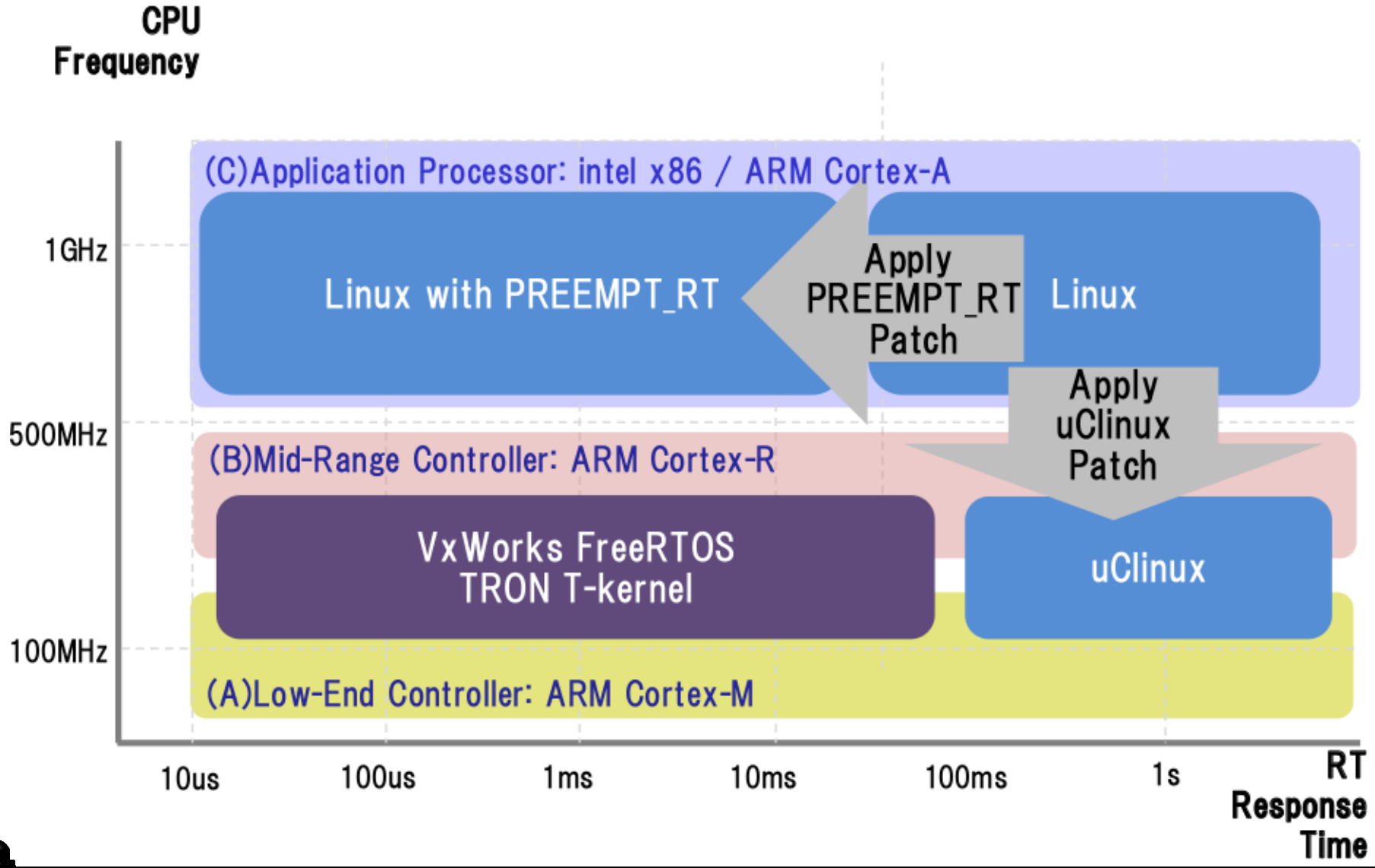


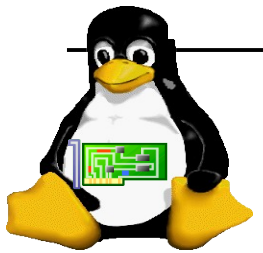
Realtime Capability of Linux

- Linux is originally for PCs. PCs have faster CPUs and larger memory and disks, compared to customized control systems
- PREEMPT_RT is a mechanism to respond faster to the external inputs
  - Making in-kernel locking-primitives (using spinlocks) preemptible by reimplementing with rtmutexes.
  - Implementing priority inheritance for in-kernel spinlocks and semaphores.
  - Converting interrupt handlers into preemptible threads



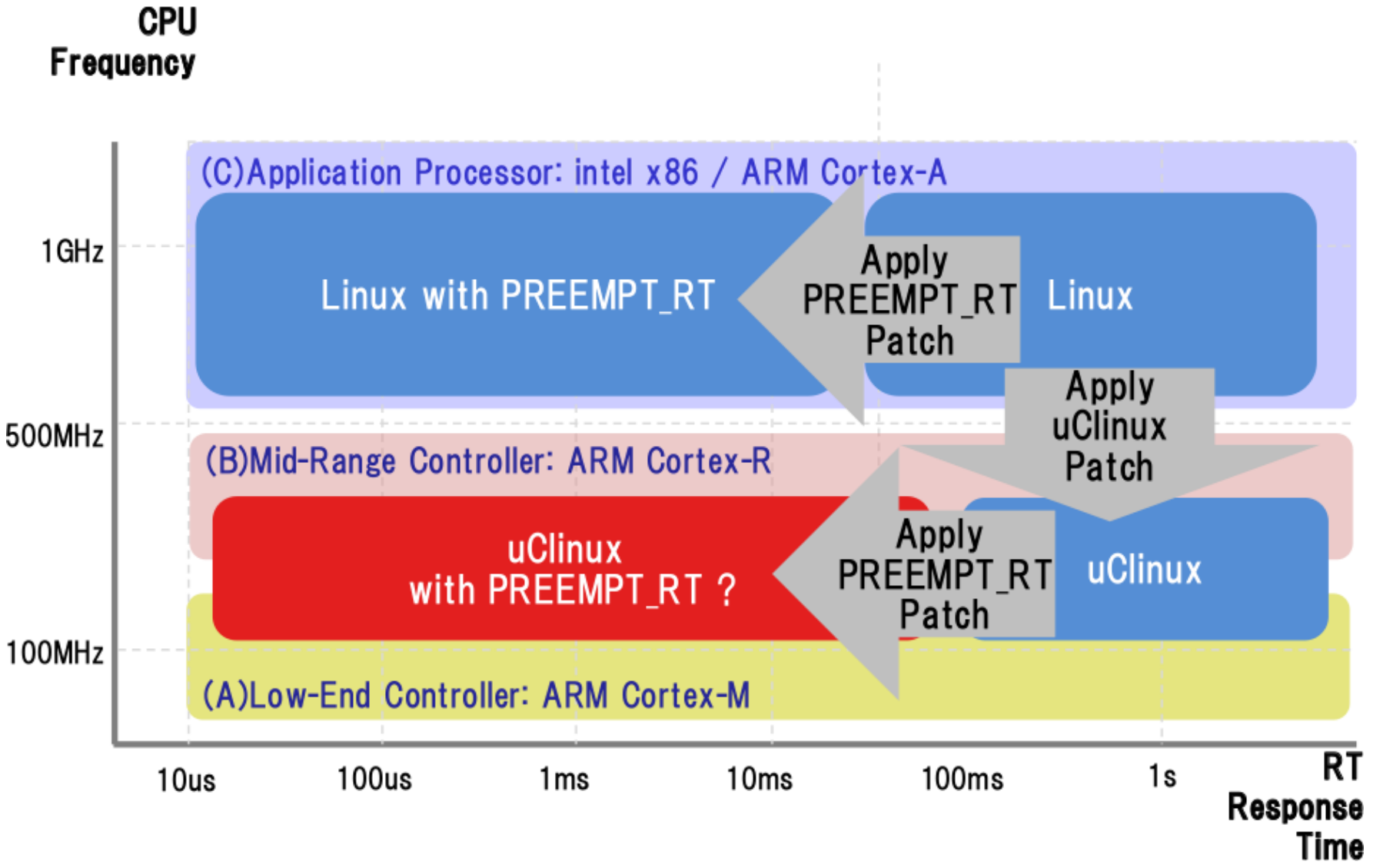
CPU Frequency vs. Response Time

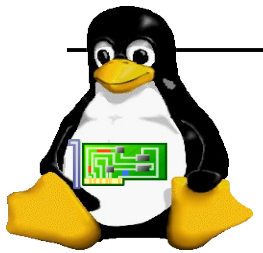




Source: Evaluation of uClinux and PREEMPT_RT for Machine Control System, Hitachi, Ltd

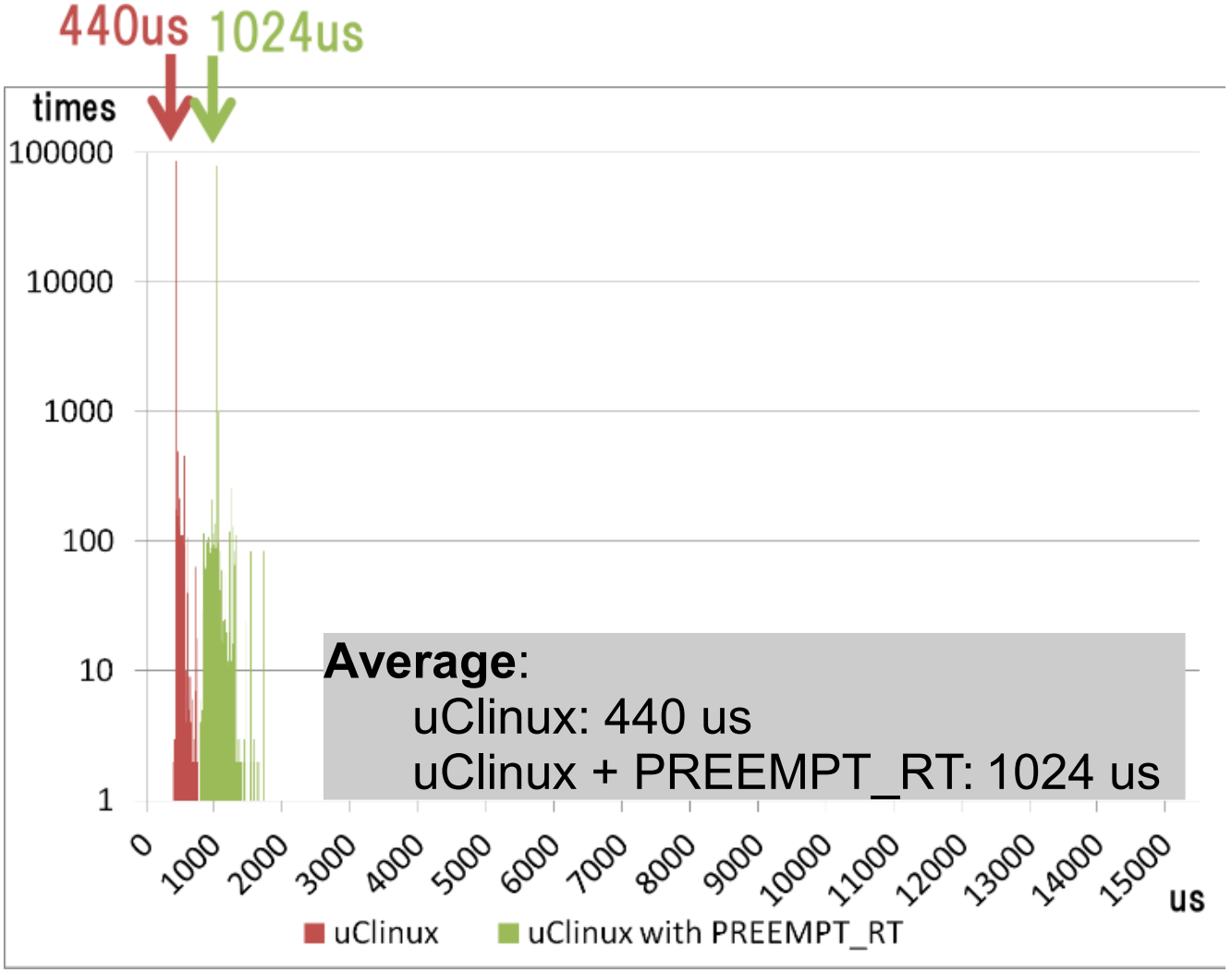
CPU Frequency vs. Response Time

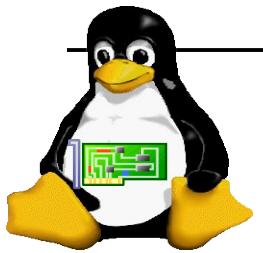




Source: Evaluation of uClinux and PREEMPT_RT for Machine Control System, Hitachi, Ltd

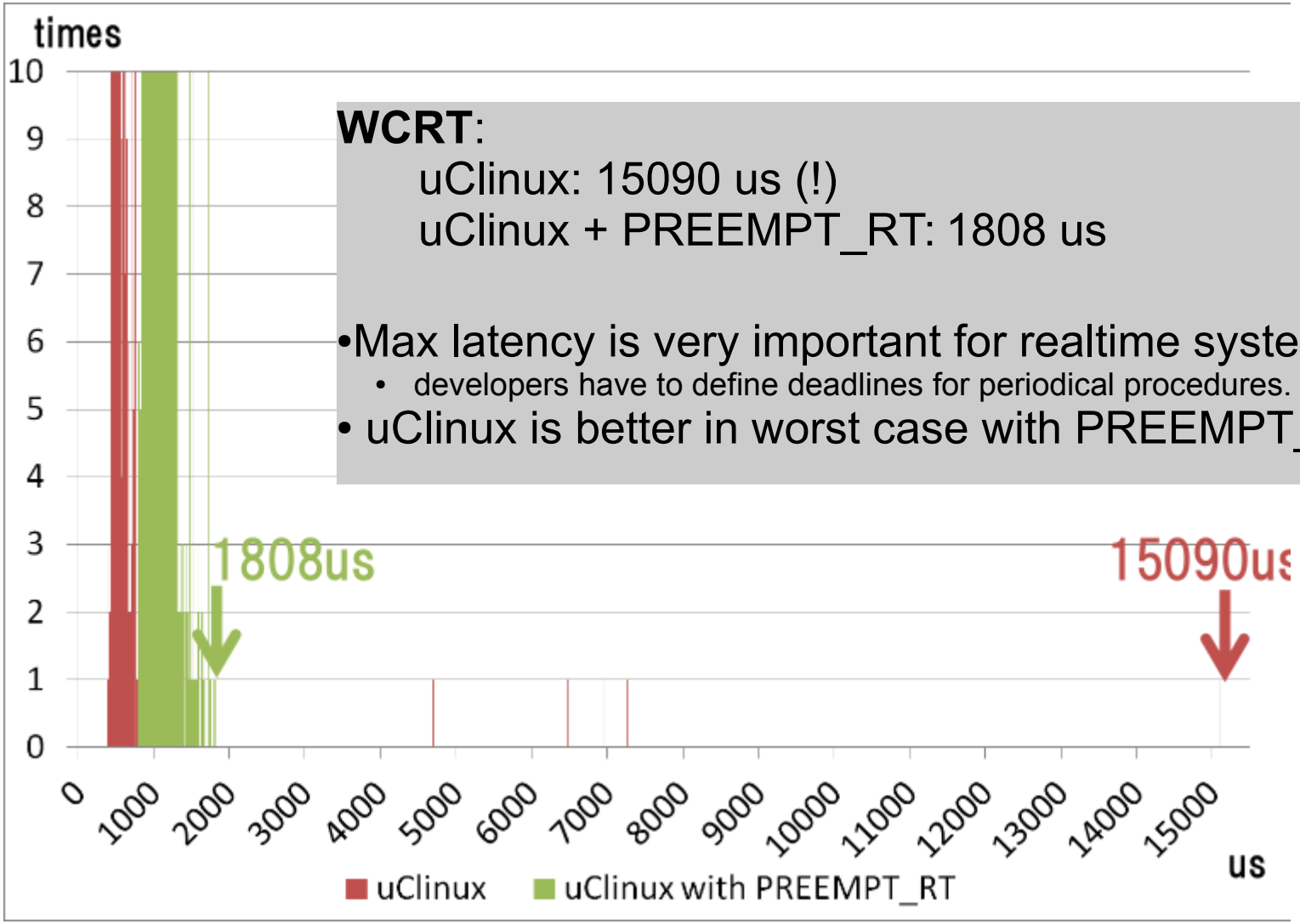
Cyclictest on STM32F407 (168MHz)





Source: Evaluation of uClinux and PREEMPT_RT for Machine Control System, Hitachi, Ltd

Cyclictest on STM32F407 (168MHz)

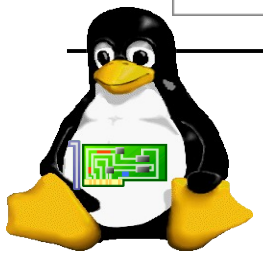


WCRT:

uClinux: 15090 us (!)

uClinux + PREEMPT_RT: 1808 us

- Max latency is very important for realtime systems
  - developers have to define deadlines for periodical procedures.
- uClinux is better in worst case with PREEMPT_RT

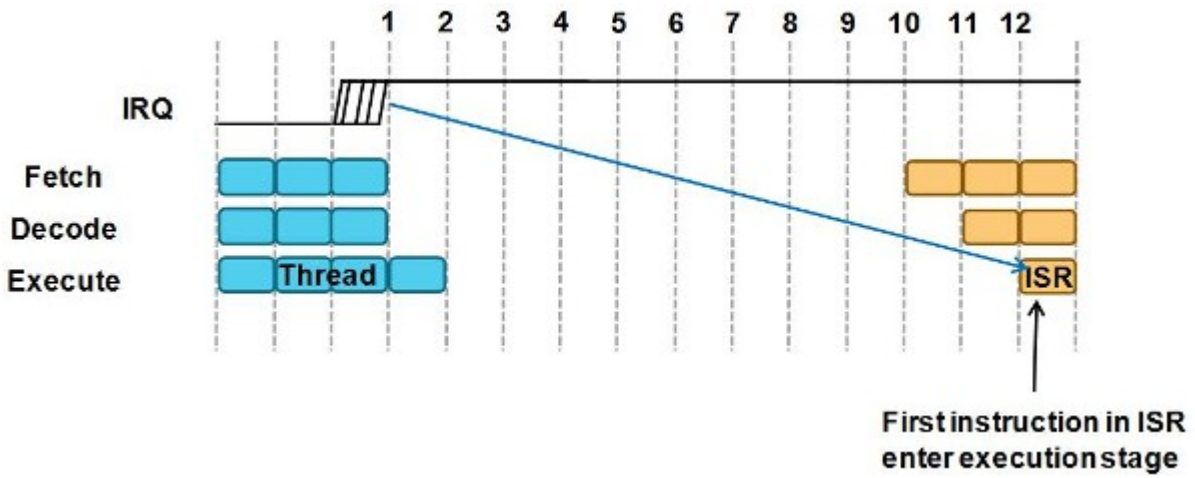


Source: Evaluation of uClinux and PREEMPT_RT for Machine Control System, Hitachi, Ltd

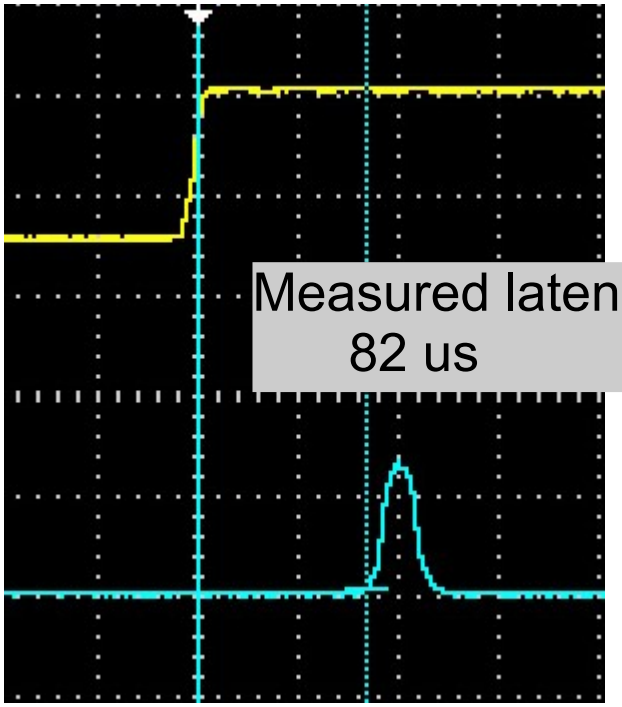
Optimize IRQ latency

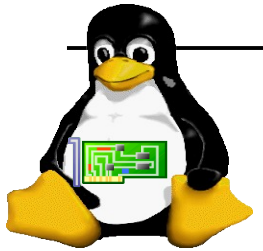
uClinux's IRQ latency is shorter with PREEMPT_RT

- Max latency could be 80us if code + data of IRQ context are placed in SRAM rather than reported max latency 1808 us
- The overhead is still quite higher!

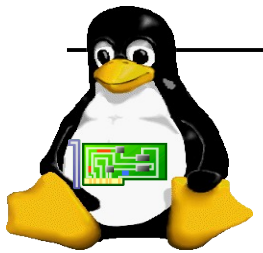


Source: A Beginner's Guide on Interrupt Latency - and Interrupt Latency of the ARM Cortex-M processors





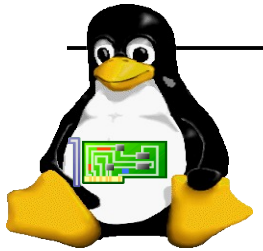
Optimizations: Boot Time



Enable XIP

Experiments on STM32F42x

- Start-up time
  - u-boot is 41% faster
  - Kernel is 18% slower
  - u-boot + kernel combination is almost the same
- It is worthy for kernel because
  - lower RAM reserved by kernel
  - more free RAM for applications

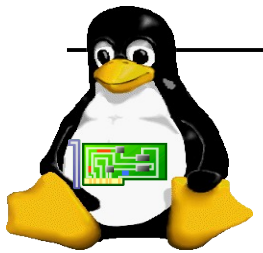


No u-boot

Directly load Linux kernel by really small program

- Fast!

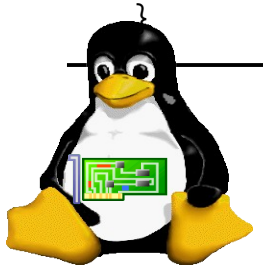
```
__attribute__((section(".vector_table")))
void (*vector_table[16 + 91])(void) = { void (*))&_stack_top, reset, …};
void reset(void) {
        unsigned int *src, *dst;
        asm volatile ("cpsid i"); src = &_end_text; dst = &_start_data;
        while (dst < &_end_data) *dst++ = *src++;
        dst = &_start_bss;
        while (dst < &_end_bss) *dst++ = 0;
        main();
}
```



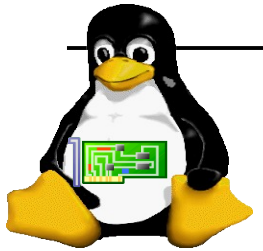
Source: https://github.com/afaerber/afboot-stm32

Tiny boot loader

```
int main(void)
{
    volatile uint32_t *FLASH_KEYR = (void *)(FLASH_BASE + 0x04);
   …
    clock_setup();
   …
    ptr = (void *)0xD0000000UL;
    i = 0x00800000UL / sizeof(*ptr); while (i-- > 0) *ptr++ = 0;
    ….
    void (*kernel)(uint32_t reserved, uint32_t mach, uint32_t dt) =
            (void (*)(uint32_t, uint32_t, uint32_t))(0x08008000 | 1);
    kernel(0, ~0UL, 0x08004000);
}
```

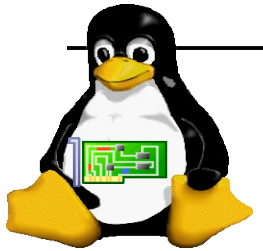


Optimizations: Toolchain



Build Your Own uClinux Toolchain

- The official site uclinux.org maintains source distribution along with specific GNU Toolchain, but it was too old for our experiments
- OSELAS.Toolchain() project provides a complete build system for recent GNU toolchains.
  - uses PTXdist build system
  - http://www.pengutronix.de/oselas/toolchain/download/
- TARGET “arm-cortexm3-uclinuxeabi” in Release 2014.12
    - arm-cortexm3-uclinuxeabi
    - gcc-4.9.2, uclibc-0.9.33.2, binutils-2.24, kernel-3.16



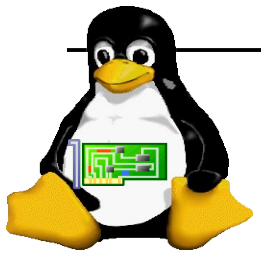
GCC & Thumb2

- Kernel is 29% smaller in Thumb-2 compared to ARM
  - Reported by “Experiment with Linux and ARM Thumb-2 ISA”, Philippe Robin
- However, we might encounter compiler optimization issues

Source: drivers/char/random.c

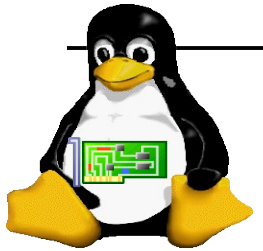
```
static ssize_t extract_entropy(struct entropy_store *r, void *buf,
                               size_t nbytes, int min, int reserved)
{
        __u8 tmp[EXTRACT_SIZE] __attribute__((aligned(32)));
        ...
        while (nbytes) {
                extract_buf(r, tmp);
                if (r->last_data) { ...
                        memcpy(r->last_data, tmp, EXTRACT_SIZE);
```

In extract_buf(), hash will be copied to out. Compiler optimizes memcpy() aggresively and does not test the unaligned destination address. Therefore, we have to specify aligned attributes to tmp[]



minimize the literal load

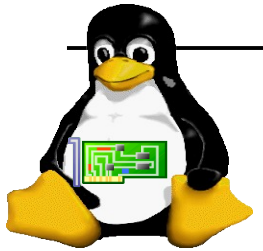
- gcc-4.9 supports a new option, `-mslow-flash-data`, which assumes that loading data from flash is slower than fetching instruction.
- Therefore literal load is minimized for better performance.



Memory operations

Generally, 2x speed up

- ARM contributes a customized memcpy routine optimisez for Cortex-M3/M4 cores with/without unaligned access. (newlib)
  - Step 1: Align src/dest pointers, copy mis-aligned if fail to align both
  - Step 2: Repeatedly copy big block size of __OPT_BIG_BLOCK_SIZE
  - Step 3: Repeatedly copy big block size of __OPT_MID_BLOCK_SIZE
  - Step 4: Copy word by word
  - Step 5: Copy byte-to-byte
- Tunable options:
  - __OPT_BIG_BLOCK_SIZE: Size of big block in words. Default to 64.
  - __OPT_MID_BLOCK_SIZE: Size of big block in words. Default to 16.

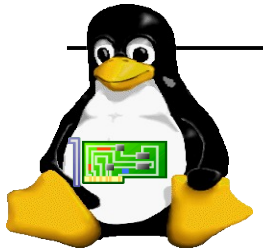


Utilize Hardware FPU

Algorithms' turnaround time and CPU load measurements

Hardware: Cortex-M3 NXP LPC1759 operating at its maximum core speed of 120MHz.

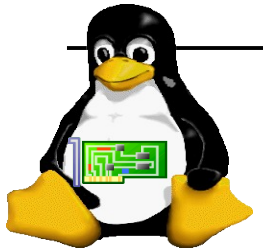
	CM4 with Hardware FP	CM4 with Software FP	CM3 with Software FP
RLSDF	5.86µs	43.20µs	2
Control Task CPU load	4.2%	35.3%	27
Total CPU load	13.1%	63.2%	13



Source: “A Self Tuning Regulator based on the ARM Cortex-M4”, R´omulo Ant˜ao, Alexandre Mota, Rui Escadas Martins

Auto-reduce Project

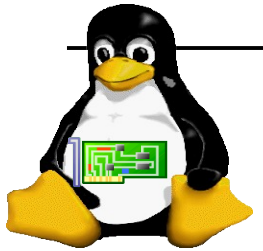
- Find automated ways to reduce the kernel
  - Link-time optimization
  - System call elimination
  - Kernel command-line argument elimination
  - Kernel constraint system
- Additional research
  - Link-time re-writing
  - Cold-code compression



Source: Advanced Size Optimization of the Linux Kernel, Tim Bird

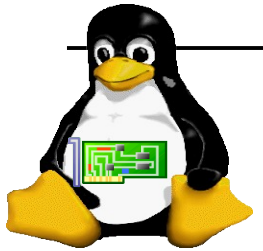
Link Time Optimization

- LTO is a new GNU toolchain feature (gcc 4.7+)
  - Save extra meta-data (GIMPLE) at compile-time
  - Use meta-data at link time to do whole-program optimization
- Option: `-flto`
- Has slow link step, but better code optimization



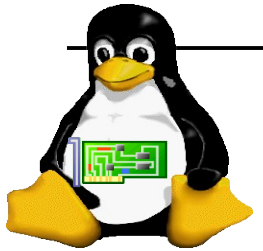
LTO Benefits

- Opens up a whole new class of optimizations
- Performance improvements:
  - No visible results for ARM Cortex-M4
  - Only microbenchmark results available
- Size improvement:
  - ~4% kernel size reduction for STM32F429



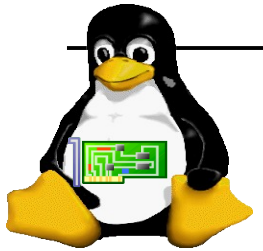
Potential LTO Benefits

- Can automatically drop unused code and data
  - Partial inlining – e.g. only inline some code, like tests at beginning of functions
- Optimize arguments to global functions
  - Drop unnecessary args, optimize inputs/outputs, etc.
- Detect read-only variables and optimize
- Performan constant propagation, and function call specialization based on that
  - If a function is called commonly with a constant, make a special version of the function optimized for that
  - e.g. kmalloc_GFP_KERNEL()



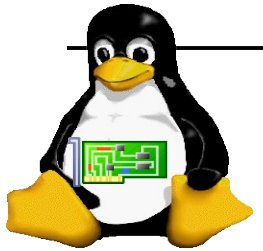
LTO Benefits

- Opens up a whole new class of optimizations
- Performance improvements:
  - No visible results for ARM Cortex-M4
  - Only microbenchmark results available
- Size improvement:
  - ~4% kernel size reduction for STM32F429



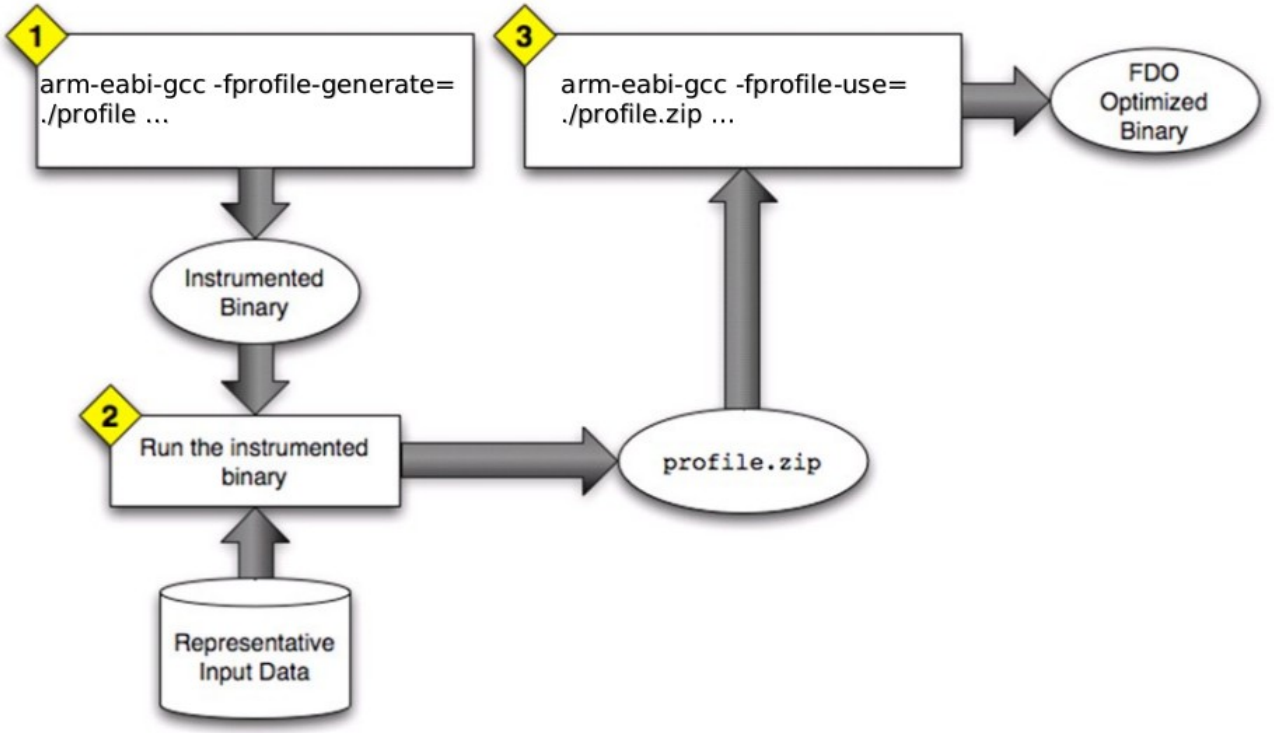
FDO in gcc-4.9

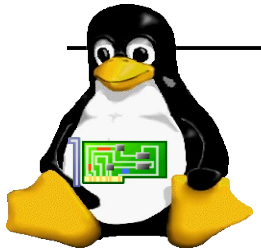
- Improved FDO (Feedback directed optimization)
  - New time profiling determines typical order in which functions are executed.
  - GCC uses heuristics to guess branch probabilities if they are not provided by profiling feedback (-fprofile-arcs).
- new function reordering pass (controlled by -freorder-functions) reduces startup time.
  - effective only with link-time optimization.
  - Reorder functions in the object file in order to improve code Locality.
    - using special subsections ".text.hot" for most frequently executed functions and ".text.unlikely" for unlikely executed functions.
    - profile feedback must be available to make this option effective.



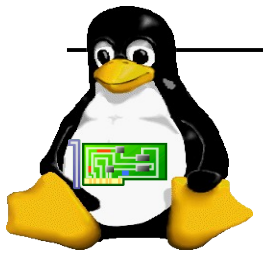
FDO for device

- Stable and fast storage is required: SD/MMC
- Significant changes against the existing build procedure to satisfy the “build-run-build” model
- Complex C++ programs like Qt can benefit from the combination of `-freorder-functions` and `-flto`





Optimizations: Kernel Parameters

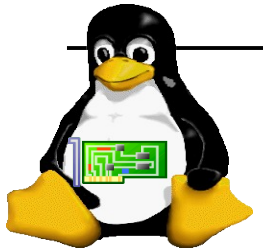


Kernel Parameters

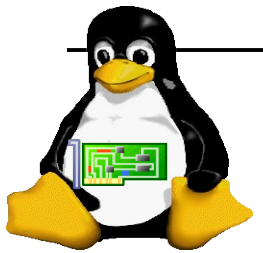
- Module parameters for loadable modules are specified only as the parameter name with optional '=' and value as appropriate:

  ```
  modprobe usbcore blinkenlights=1
  ```

- Parameters denoted with BOOT are actually interpreted by the boot loader, and have no meaning to the kernel directly
- For deeply embedded devices, we don't really need kernel command line since we might have no way to configure
- Defined with _setup() and early_param() macros from include/linux/init.h
- 12 KB reduction if we eliminate the handling



Pending Issues



Problems

- Rubustness of Thumb2 toolchain optimizations
- In-kernel debugging tools are too heavy and impractical for Cortex-M.

```
#error "CONFIG_TRACE_IRQFLAGS not supported on the
current ARMv7M implementation"
```

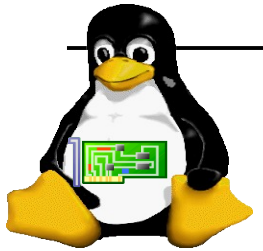
(source: arch/arm/kernel/entry-v7m.S)

- OLD codebase without collaboration
  - Recently, upstreaming is on-going...

From Maxime Coquelin

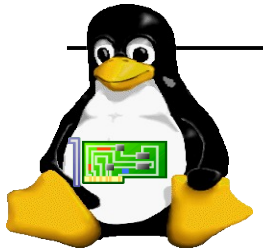
Subject [PATCH v3 00/15] Add support to STMicroelectronics STM32 family

Date Thu, 12 Mar 2015 22:55:46 +0100



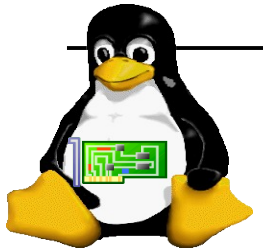
TODO: Tickless kernel

- kernel maintains a "kernel ticker" timer that triggers an interrupt at a fixed frequency
  - allowing the Linux scheduler to resume those processes for which timer-related events may have occurred → a software timeout may have expired for a process
- Default kernel ticker rate is 100 Hz
  - even when the system can be fully idle, the kernel would still wake up and switch back to dynamic power consumption levels 100 times per second → unnecessary increasing the overall power consumption.
- Tickless: kernel does not wake up at a 100Hz rate using the normal kernel ticker but instead explicitly keeps track of all timer-related events and calculates when exactly the system needs to wake-up next



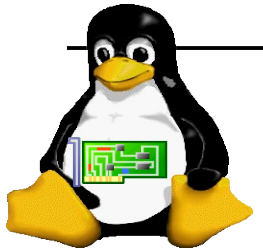
TODO: Tickless kernel

- hrtimer is critical for tickless but not available on evaluation environment
- STM32 System Timer is used as clockevent.
- Migrate to 3.10+ for full tickless operation (CONFIG_NO_HZ_FULL)
  - disabling the tick for non-idle processors
  - Fair and comprehensive measurements required



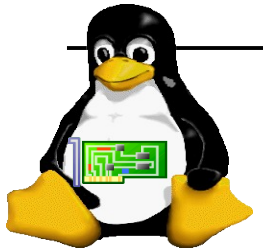
TODO: True Preemptive Kernel

- Initially, kernel threads runs in Handler mode with SVCall priority.
  - Disadvantage: kernel preemption is not possible since the Handler mode cannot be preempted by the Thread mode.
- Exception handling code can be improved
  - only runs for a short time and switches to the privileged Thread mode for executing the rest of the kernel code
  - acts as the tiny dispatcher responsible for switching between kernel and user applications and passing arguments
  - Conceptually compatible with the expectation of PREEMPT_RT



Conclusion

- BYOD with uClinux is quite reasonable for targets of rich features
- The techniques to improve uClinux are basically the same as ARM-Linux, but we need more aggressive methods to tweak certain areas such as toolchain optimization, boot loader, tickless kernel, and PREEMPT_RT.
- Few ARMv7-M boards (only efm32) are supported by mainline kernel. Need more efforts for upstreaming.
- There is still room for satisfying the constraints of deeply embedded devices especially when in-kernel debugger/tracker are enabled.



Reference

- Spreading the disease: Linux on microcontrollers, Vitaly Wool
- OpenMV: https://hackaday.io/project/1313-openmv
- Advanced Size Optimization of the Linux Kernel, Tim Bird
- Evaluation of uClinux and PREEMPT_RT for Machine Control System, Hitachi, Ltd
- Running uClinux on ARM Cortex-M3 Platform, Fujitsu Computer Technologies Limited