Linaro connect
Bangkok 2016

# Internet of Tiny Linux (IoTL)

## status and progress

**Presented by**

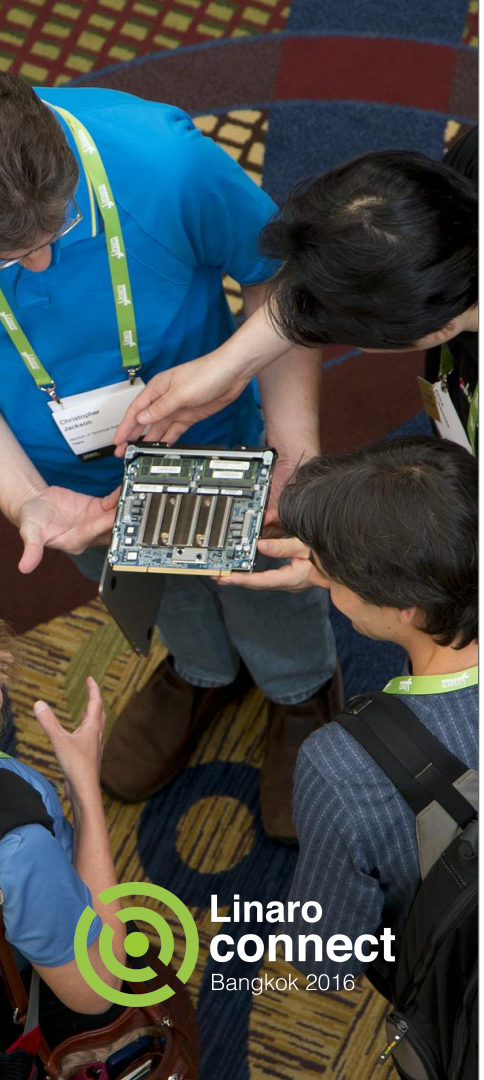
Nicolas Pitre

**Date**

BKK16-211 March 8, 2016

**Event**

Linaro Connect BKK16




# Internet of Tiny Linux (IoTL)

Discussion about various methods put forward to reduce the size of Linux kernel and user space binaries to make them suitable for small IoT applications.

# Very pervasive:

- Cable/ADSL Modems
- Smart Phones
- Smart Watches
- Internet Connected Refrigerators
- WI-FI-Enabled Washing Machines
- Smart TV Sets
- Wi-Fi enabled Light Bulbs
- Connected Cars
- Alarm Systems monitored via Internet
- etc.

# "Internet of Things" (IoT)

Problem: **Cost**

**Software Development is**

- **Hard**
- **Expensive**
- **Slow**

**Legacy Software Maintenance is**

- **Hard**
- **Expensive**
- **Uninteresting**

# "Internet of Things" (IoT)

## Another Problem: *Security*

- All solutions _will_ eventually be broken
  - Think NSA…
  - Then professional thieves...
  - Then script kiddies…
- Security Response is a _must_... even for gadgets!*

quote, Bruce Schneier:

https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html

“The Internet of Things Is Wildly Insecure-And Often Unpatchable”

# "Internet of Things" (IoT)

Solutions:

- Avoid custom base software
- Leverage the Open Source community
- Gather critical mass around common infrastructure
- Share the cost of non-differentiating development

# "Internet of Things" (IoT)

**Linux** is a logical choice

- Large community of developers
- Best looked-after network stack
- Extensive storage options
- Already widely used in embedded setups
- Etc.

# "Internet of Things" (IoT)

## The Linux kernel is a logical choice... BUT

- it is featureful -> Bloat
- its default tuning is for high-end systems
- the emphasis is on scaling up more than scaling down
- its flexible configuration system leads to
  - Kconfig hell
  - suboptimal build
- is the largest component in most Linux-based embedded systems

**Linux Kernel Size Reduction is part of the solution***

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

What can be done?

Existing resources:

- Linux Kernel Tinification Wiki https://tiny.wiki.kernel.org/
- Kernel Size Reduction Work - eLinux.org http://elinux.org/Kernel_Size_Reduction_Work

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Already Done (few examples):

- Compile out block device support.
- Compile out NTP support.
- Compile out support for capabilities (only root is granted permission).
- Compile out support for non-root users and groups.
- Compile out printk() and related strings.

Needed: more similar config options.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Work In Progress: A poor man's LTO[]

LTO is cool... BUT

Table 1: Full Kernel Build Timing

| Build Type | Wall Time |
|---|---|
| Standard Build | 4m53s |
| LTO Build | 10m23s |

Table 2: Kernel Rebuild Timing After a Single Change

| Build Type | Wall Time |
|---|---|
| Standard Build | 0m12s |
| LTO Build | 6m27s |

NOTE: Build Details: Linux v4.2 ARM multi_v7_defconfig gcc v5.1.0 Intel Core2 Q6600 CPU at 2.40GHz

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Alternative to LTO: `ld -gc-sections`

| Build Type | Size (bytes) | Reference % |
|---|---|---|
| allnoconfig | 860508 | 100% |
| allnoconfig + CONFIG_NO_SYSCALLS | 815804 | 94.8% |
| allnoconfig + CONFIG_NO_SYSCALLS + CONFIG_GC_SECTIONS | 555798 | 64.6% |
| allnoconfig + CONFIG_NO_SYSCALLS + CONFIG_LTO | 488264 | 56.7% |

The `-gc-sections` result is somewhat bigger but so much faster to build.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

The `ld -gc-sections` approach: more intrusive than meets the eye

```
--- a/arch/arm/include/asm/assembler.h
+++ b/arch/arm/include/asm/assembler.h
@@ -88,6 +88,17 @@
 #endif


 /*
+ * Special .pushsection wrapper with explicit dependency to prevent
+ * garbage collection of the specified section.  This is needed when no
+ * explicit symbol references are made to this section.
+ */
+       .macro          .pushlinkedsection name:vararg
+       .reloc          . - 1, R_ARM_NONE, 9909f
+       .pushsection \name
+9909:
+       .endm
+
+/*
```

Let's not forget changes to linker scripts, modpost, etc.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size (continued)

The `ld -gc-sections` approach: more intrusive than meets the eye

```
 * Enable and disable interrupts
 */
#if __LINUX_ARM_ARCH__ >= 6
@@ -239,7 +250,7 @@

#define USER(x...)                                      \
9999:   x;                                              \
-       .pushsection __ex_table,"a";                    \
+       .pushlinkedsection __ex_table,"a";              \
        .align  3;                                      \
        .long   9999b,9001f;                            \
        .popsection
```

Let's not forget changes to linker scripts, modpost, etc.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Submitted Upstream: Trim unused exported kernel symbols

- EXPORT_SYMBOL(foo_bar) forces foo_bar() into the kernel even if there is no users.
- Let's export only those symbols needed by the set of configured modules.
- Allows LTO and -gc-sections to get rid of related code.

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

List of symbols required by modules

```
----
$ nm linux/crypto/sha512_generic.ko
00000000 T cleanup_module
         U crypto_register_shashes
00000880 T crypto_sha512_finup
00000bc0 T crypto_sha512_update
         U crypto_unregister_shashes
00000000 T init_module
         U memcpy
000000e8 r __module_depends
00000000 t sha384_base_init
00000000 d sha512_algs
00000090 t sha512_base_init
00000d00 t sha512_final
...
```

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

List of symbols required by modules

linux/include/generated/autoksyms.h

```
----
/*
 * Automatically generated file; DO NOT EDIT.
 */

#define __KSYM_crypto_register_shashes 1
#define __KSYM_crypto_unregister_shashes 1
#define __KSYM_memcpy 1
----
```

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols == some preprocessor magic

Excerpt from mainline submission

```
--- a/include/linux/export.h
+++ b/include/linux/export.h
@@ -65,6 +65,24 @@
                __attribute__((section("___ksymtab" sec "+" #sym), unused))          \
                = { (unsigned long)&sym, __kstrtab_##sym }

+#ifdef CONFIG_TRIM_UNUSED_KSYMS
+
+#include <linux/kconfig.h>
+#include <generated/autoksyms.h>
+
+#define __EXPORT_SYMBOL(sym, sec)                                              \
+       __cond_export_sym(sym, sec, config_enabled(__KSYM_##sym))
+#define __cond_export_sym(sym, sec, conf)                                      \
+       ___cond_export_sym(sym, sec, conf)
```

Linaro connect Bangkok 2016

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols == some preprocessor magic (Continued)

Excerpt from mainline submission

```
+#define ___cond_export_sym(sym, sec, enabled)                                  \
+                __cond_export_sym_##enabled(sym, sec)
+#define __cond_export_sym_1(sym, sec) ___EXPORT_SYMBOL(sym, sec)
+#define __cond_export_sym_0(sym, sec) /* nothing */
+
+#else
+#define __EXPORT_SYMBOL ___EXPORT_SYMBOL
+#endif
+
 #define EXPORT_SYMBOL(sym)                                                    \
                 __EXPORT_SYMBOL(sym, "")


---
```

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols == more preprocessor magic

Excerpt from <linux/kconfig.h> for the config_enabled() definition

```
----
/*
 * Getting something that works in C and CPP for an arg that may or may
 * not be defined is tricky.  Here, if we have "#define CONFIG_BOOGER 1"
 * we match on the placeholder define, insert the "0," for arg1 and generate
 * the triplet (0, 1, 0).  Then the last step cherry picks the 2nd arg (a one).
 * When CONFIG_BOOGER is not defined, we generate a (... 1, 0) pair, and when
 * the last step cherry picks the 2nd arg, we get a zero.
 */
#define __ARG_PLACEHOLDER_1 0,
#define config_enabled(cfg) _config_enabled(cfg)
#define _config_enabled(value) __config_enabled(__ARG_PLACEHOLDER_##value)
#define __config_enabled(arg1_or_junk) ___config_enabled(arg1_or_junk 1, 0)
#define ___config_enabled(__ignored, val, ...) val
----
```

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Trim unused exported kernel symbols: build refresh dependencies

Avoid rebuilding the whole kernel when content of <linux/autoksyms.h> changes.

First attempt: augment the fixdep parser

- EXPORT_SYMBOL()
- EXPORT_SYMBOL_GPL()
- EXPORT_PER_CPU_SYMBOL()
- EXPORT_EARLY_SYMBOL()
- ACPI_EXPORT_SYMBOL()
- ACPI_EXPORT_SYMBOL_INIT()

And combinations such as EXPORT_PER_CPU_SYMBOL_GPL(), etc.

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols: build refresh dependencies

Now this:

```
#define PCI_USER_READ_CONFIG(size, type)                \
int pci_user_read_config_##size                         \
        (struct pci_dev *dev, int pos, type *val)       \
{                                                       \
[...]                                                   \
}                                                       \
EXPORT_SYMBOL_GPL(pci_user_read_config_##size);
```

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols: build refresh dependencies

Extract symbol name warnings:

```
# Filter out exported kernel symbol names advertised as warning pragmas
# by the preprocessor and write them to $(1). We must consider continuation
# lines as well: they start with a blank, or the preceeding line ends with
# a ':'. Anything else is passed through as is.
# See also __KSYM_DEP() in include/linux/export.h.
ksym_dep_filter = sed -n \
	-e '1 {x; $$!d;}' \
	-e '/^ / {H; $$!d;}' \
	-e 'x; /:$$/ {x; H; $$!d; s/^\ /; x;}' \
	-e ':filter; /^.*KBUILD_AUTOKSYM_DEP: /! {p; b next;}' \
	-e 's//KSYM_/; s/\n.*//; w $(1)' \
	-e ':next; $$!d' \
	-e '1 q; s/^\ /; x; /^ /! b filter'
```

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols: build refresh dependencies

Dedicated preprocessor pass:

```
--- a/include/linux/export.h
+++ b/include/linux/export.h
@@ -65,7 +65,18 @@ extern struct module __this_module;
        __attribute__((section("___ksymtab" sec "+" #sym), unused))     \
        = { (unsigned long)&sym, __kstrtab_##sym }

-#ifdef CONFIG_TRIM_UNUSED_KSYMS
+#if defined(__KSYM_DEPS__)
+
+/*
+ * For fine grained build dependencies, we want to tell the build system
+ * about each possible exported symbol even if they're not actually exported.
+ * We use a string pattern that is unlikely to be valid code that the build
+ * system filters out from the preprocessor output (see ksym_dep_filter
+ * in scripts/Kbuild.include).
+ */
```

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols: build refresh dependencies (Continued)

Dedicated preprocessor pass:

```
+#define __EXPORT_SYMBOL(sym, sec)      === __KSYM_##sym ===
+
+#elif defined(CONFIG_TRIM_UNUSED_KSYMS)


 #include <linux/kconfig.h>
 #include <generated/autoksyms.h>
```

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Trim unused exported kernel symbols: some numbers.

Kernel v4.5-rc2 X86 defconfig

```
$ size vmlinux
   text    data     bss     dec     hex filename
12362563 1856456 1101824 15320843 e9c70b vmlinux
$ wc -l Module.symvers
8806 Module.symvers
```

Kernel v4.5-rc2 X86 defconfig + CONFIG_TRIM_UNUSED_KSYMS

```
$ size vmlinux
   text    data     bss     dec     hex filename
12059848 1856456 1101824 15018128 e52890 vmlinux
$ wc -l Module.symvers
225 Module.symvers
```

Because the x86 defconfig only contains 18 modules, the number of needed exported symbols is only 225 out of a possible 8806 for this configuration. The kernel text size shrank by about 2.4%.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Trim unused exported kernel symbols: more numbers, on ARM this time.

Kernel v4.5-rc2 ARM defconfig

```
$ size vmlinux
   text    data     bss     dec     hex filename
13664222 1554068  351368 15569658  ed92fa vmlinux
$ wc -l Module.symvers
10044 Module.symvers
```

.Kernel v4.5-rc2 ARM defconfig + CONFIG_TRIM_UNUSED_KSYMS

```
$ size vmlinux
   text    data     bss     dec     hex filename
13255051 1554132  351240 15160423  e75467 vmlinux
$ wc -l Module.symvers
2703 Module.symvers
```

This time many more modules (279 of them) are part of the build configuration. Still, only 2703 out of 10044 exported symbols are required. And despite a smaller number of omitted exports, the kernel shrank by 3%.

NOTE: defconfig is equivalent to multi_v7_defconfig

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size

Trim unused exported kernel symbols: more numbers using LTO build.

Kernel v4.5-rc2 ARM defconfig + LTO

```
$ size vmlinux
   text    data     bss     dec     hex filename
12813766  1538324  344356 14696446  e03ffe vmlinux
$ wc -l Module.symvers
8415 Module.symvers
```

Kernel v4.5-rc2 ARM defconfig + LTO + CONFIG_TRIM_UNUSED_KSYMS

```
$ size vmlinux
   text    data     bss     dec     hex filename
12197437  1536052  338660 14072149  d6b955 vmlinux
$ wc -l Module.symvers
1742 Module.symvers
```

This time the kernel shrank by 5%.

# "Internet of Things" (IoT)

Reducing the Linux Kernel Size

Trim unused exported kernel symbols: more numbers on a small build. Let's have a look at a configuration that is potentially more representative of an embedded target.

Kernel v4.5-rc2 ARM realview_defconfig + LTO

```
$ size vmlinux
   text    data     bss     dec     hex filename
4422942  209640  126880 4759462  489fa6 vmlinux
$ wc -l Module.symvers
5597 Module.symvers
```

Kernel v4.5-rc2 ARM realview_defconfig + LTO + CONFIG_TRIM_UNUSED_KSYMS

```
$ size vmlinux
   text    data     bss     dec     hex filename
3823485  205416  125800 4154701  3f654d vmlinux
$ wc -l Module.symvers
52 Module.symvers
```

Here we reduced the kernel text by about 13.6%. Disabling module support altogether does reduce it by 13.9% i.e. only 0.3% difference.

This means the overhead of using modules on embedded targets is greatly reduced.

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size *AND* User Space Size

What next?

- Apply same trick to system calls
- Tool to determine list of system calls used by user space

To be effective, this implies:

- Statically linked applications
- Single executable file (busybox style)

For more flexibility:

- Shared library with minimal support (discard unused objects)
- Fine grained object dependencies (more .o files with less code for each)

# "Internet of Things" (IoT)

## Reducing the Linux Kernel Size *AND* User Space Size

Miscellaneous:

- Code utilization profiling
- Binary size regression tests
- NoMMU Linux on Cortex-A processors
- Mainline support for FDPIC executable format
- Revive XIP support for the kernel and user space
- More kernel feature modularization

# "Internet of Things" (IoT)

Questions ?