BitThunder

A RELIABLE RTOS – GITHUB.COM/BITTHUNDER/

BitThunder

- Open-Source Project
- Somewhere between bare-metal and linux…
- Can use any tasking kernel, default FreeRTOS. Eventually BtKernel.
- Forces an abstract platform model…
  Encourages strong encapsulation, and robust interfaces.
  Almost impossible for developers to break.
- Large amount of code re-use… smaller applications, better drivers.
  E.g. Shared FIFO implementation across UART/CAN etc drivers.
- Opportunity to open-source common drivers... under a RT framework.

Why?

- Unified platform/architecture across embedded devices:
  - Embedded application processors (Cortex-A9). E.g. sub-processors.
  - Micro-Controller (Cortex-M0/M3… in theory also PIC/Atmel/Atmega)
    - Smallest BSP: 32kb ROM 8kb RAM. (Smaller possible… single function devices).
- Common Design Language / Shared developer knowledge.
  - More flexible development resource scheduling.
- Encourage developers to use encapsulation and abstract / SOLID design principles.
- Reliable and testable firmware. (Discuss embedded TDD later).
- Learn robust development processes around git and gitlab.

Development Timeline

- BitThunder development started 12th Dec 2012. (BlueT was binned).
- Version 0.5.0 as of April 2013.
- Version 0.6.0 is current development line… to be released May 2013.
- Version 1.0.0 Expected ~September 2013. (Proposal Oct 2012).
- Currently 51,000 lines of code:

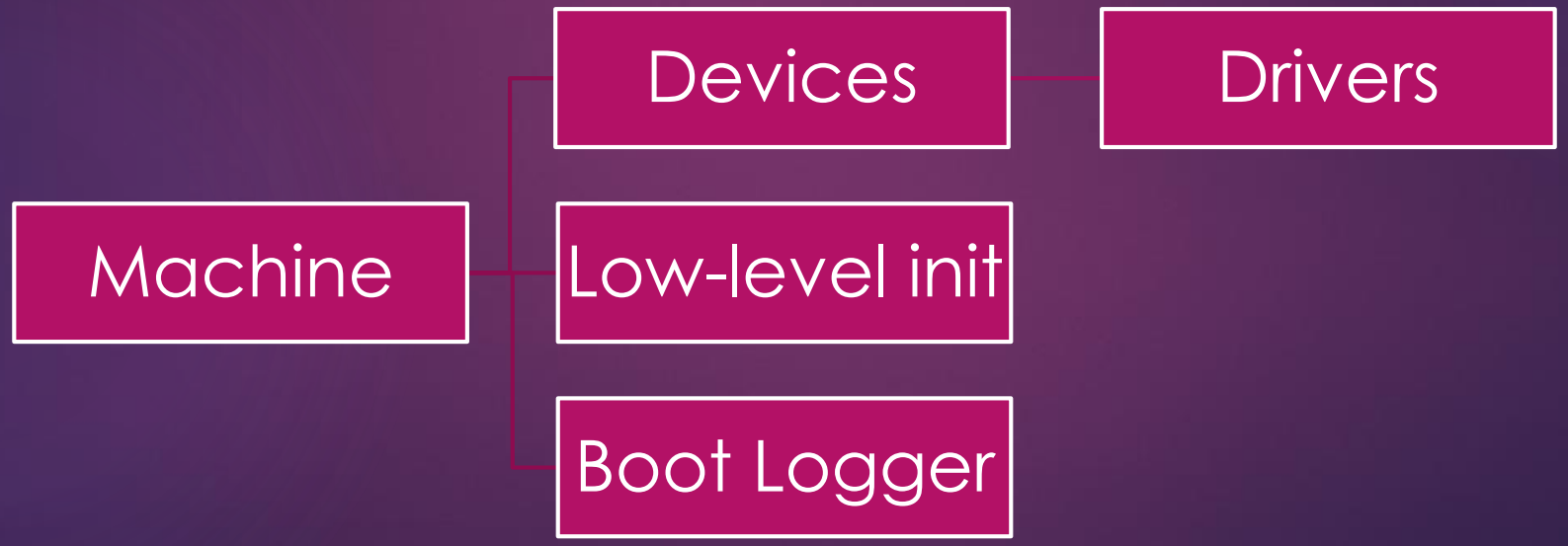
Subsystem	LOC
OS	17,292
LIB	534
Arch/MACH (HAL layer)	10,460
Drivers	1,160
BSP	9,474
FreeRTOS & kernel interface	12,992

Features (for 1.0.0)

Feature	Complete
RTOS Scheduler	100%
Process and Thread management	90%
Memory management	100% Basic Heap.
Device Filesystem.	100% (static). 90% (Dynamic).
FS/Volume/Partition Manager	90%
GPIO/UART/Timer/I2C/CAN/SPI/DMA	75%
SoftIRQ / Tasklets	100%
Syslog	100% (Direct print) 0% FS.
Process Watchdog	50%
TCP/IP stack.	0%
IPC/RPC	0%
Powerful build system	100%
Powerful configuration (Kconfig)	100%

Driver Model

- Complete separation of Device and Driver.
- Single kernel for multiple architectural variants.
  - Requires multiple machine description, and a method for getting machine id.



Zynq Machine

```
static BT_u32 zynq_get_cpu_clock_frequency() {
    return BT_ZYNQ_GetCpuFrequency();
}

BT_MACHINE_START(ARM, ZYNQ, "Xilinx Embedded Zynq Platform")
    .pfnGetCpuClockFrequency    = zynq_get_cpu_clock_frequency,
    .pInterruptController       = &oZynq_intc_device,
    .pSystemTimer               = &oZynq_cpu_timer_device,

#ifdef BT_CONFIG_MACH_ZYNQ_BOOTLOG_UART_NULL
    .pBootLogger                = NULL,
#endif
#ifdef BT_CONFIG_MACH_ZYNQ_BOOTLOG_UART_0
    .pBootLogger                = &oZynq_uart0_device,
#endif
#ifdef BT_CONFIG_MACH_ZYNQ_BOOTLOG_UART_1
    .pBootLogger                = &oZynq_uart1_device,
#endif
BT_MACHINE_END
```

arch/arm/mach/zynq/zynq.c

- Single point for platform initialisation / description.
- Provides access to core resources to get the OS up and running.
- Most “INTEGRATED_DEVICES” (from linux… think platform devices) are described here.
- Many device descriptors, with associated fundamental resources.
- Some have static “INODES”… i.e. a device filesystem entry.

Device Descriptor

- Separation of concerns.
- Forces abstraction on driver developers.
- Super configurable.
- Inode “remaps” devices from a user-space perspective.

```
#ifdef BT_CONFIG_MACH_ZYNQ_UART_1
static const BT_RESOURCE oZynq_uart1_resources[] = {
    {
        .ulStart            = 0xE0001000,
        .ulEnd              = 0xE0001000 + BT_SIZE_4K - 1,
        .ulFlags            = BT_RESOURCE_MEM,
    },
    {
        .ulStart            = 82,
        .ulEnd              = 82,
        .ulFlags            = BT_RESOURCE_IRQ,
    },
};

static const BT_INTEGRATED_DEVICE oZynq_uart1_device = {
    .name                   = "zynq,uart",
    .ulTotalResources       = BT_ARRAY_SIZE(oZynq_uart1_resources),
    .pResources             = oZynq_uart1_resources,
};

BT_DEVFS_INODE_DEF oZynq_uart1_inode = {
    .szpName = BT_CONFIG_MACH_ZYNQ_UART_1_INODE_NAME,
    .pDevice = &oZynq_uart1_device,
};
#endif
```

arch/arm/mach/zynq/zynq.c

Driver Encapsulation – Step Through

- Drivers implement well-defined interfaces.
- Private data is encapsulated inside an “OPAQUE_HANDLE”.
- Type-safety is implicitly guaranteed. (Impossible for a driver to receive an invalid handle).
- User-space use the dedicated BitThunder API provided….
  - May be almost identical to the driver interface, e.g. CAN.
  - May be extremely different… e.g. SDIO etc etc.

```
BT_ERROR BT_CanSetBaudrate          (BT_HANDLE hCAN, BT_u32 ulBaudrate);
BT_ERROR BT_CanSetConfiguration     (BT_HANDLE hCAN, BT_CAN_CONFIG *pConfig);
BT_ERROR BT_CanGetConfiguration     (BT_HANDLE hCAN, BT_CAN_CONFIG *pConfig);
BT_ERROR BT_CanEnable               (BT_HANDLE hCAN);
BT_ERROR BT_CanDisable              (BT_HANDLE hCAN);
BT_ERROR BT_CanSendMessage          (BT_HANDLE hCAN, BT_CAN_MESSAGE *pCanMessage);
BT_ERROR BT_CanReadMessage          (BT_HANDLE hCAN, BT_CAN_MESSAGE *pCanMessage);
```

os/include/interfaces/bt_dev_if_can.h

Driver Encapsulation – Interface

- Interfaces easily extended.
- Easy to maintain many different drivers.
- Extend as required.

```
typedef struct {
    BT_ERROR (*pfnSetBaudrate)   (BT_HANDLE hCAN, BT_u32 ulBaudrate);
    BT_ERROR (*pfnSetConfig)     (BT_HANDLE hCAN, BT_CAN_CONFIG *pConfig);
    BT_ERROR (*pfnGetConfig)     (BT_HANDLE hCAN, BT_CAN_CONFIG *pConfig);
    BT_ERROR (*pfnEnable)        (BT_HANDLE hCAN);
    BT_ERROR (*pfnDisable)       (BT_HANDLE hCAN);
    BT_ERROR (*pfnSendMessage)   (BT_HANDLE hCAN, BT_CAN_MESSAGE *pCanMessage);
    BT_ERROR (*pfnReadMessage)   (BT_HANDLE hCAN, BT_CAN_MESSAGE *pCanMessage);
} BT_DEV_IF_CAN;
```

os/include/interfaces/bt_dev_if_can.h

Driver Encapsulation – Data Hiding

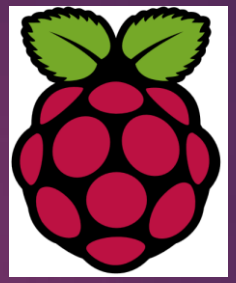
```
struct _BT_OPAQUE_HANDLE {
    BT_HANDLE_HEADER        h;          ///<
    LPC11xx_UART_REGS       *pRegs;
    const BT_INTEGRATED_DEVICE   *pDevice;
    BT_UART_OPERATING_MODE  eMode;      ///<
    BT_UART_BUFFER          oRxBuf;     ///<
    BT_UART_BUFFER          oTxBuf;     ///<
};
```

- All handles have a header… contains pointer to correct interface implementation.
- All other data is private.
- NO GLOBALS.

```
static BT_ERROR uartWrite(BT_HANDLE hUart, BT_u32 ulFlags, BT_u32 ulSize, const BT_u8 *pucSource) {
    volatile LPC11xx_UART_REGS *pRegs = hUart->pRegs;
    switch(hUart->eMode) {
    case BT_UART_MODE_POLLED:
    {
        while(ulSize) {
            while(!(pRegs->LSR & LPC11xx_UART_LSR_THRE)) {
                //BT_ThreadYield();
            }
            pRegs->FIFO = *pucSource++;
            ulSize--;
        }
        break;
    }
```

arch/arm/mach/lpc11xx/uart.c

Currently Supported Platforms



CPU Core	Platforms
ARM11	BCM2835 (Raspberry Pi)
Cortex-M3	STM32 / Stellaris / NXP (LPC17xx)
Cortex-M0(+)	NXP (LPC11xx) Infineon
Cortex-M4	Infineon
Cortex-A9	Xilinx Zynq

BT & Bootloader Demo

- Configuration System
- Multiple BSP Build
- Out of tree build
- BT booting Linux on Zynq from SD-Card.
- Linux runs on second core, and BT continues to run.

- uS Accurate syslog entries showing load time of each kernel module.
- Linux loaded and booting within ~240ms.
- Lower 16MB (or multiples thereof) reserved for BT.

Shutterboard – RIDINTPRT (MS)

- Flexible configuration and size optimisation of RIDINTPRT – MS.
- Utilising MACH layer from St. (UART etc).
- Running on BitThunder 0.6.0.
- Makes use of tiny libc functions adapted from linux kernel.

- MS – A bad point on BT, a good point…
- ST – A bad point on BT, a good point…

Development Processes

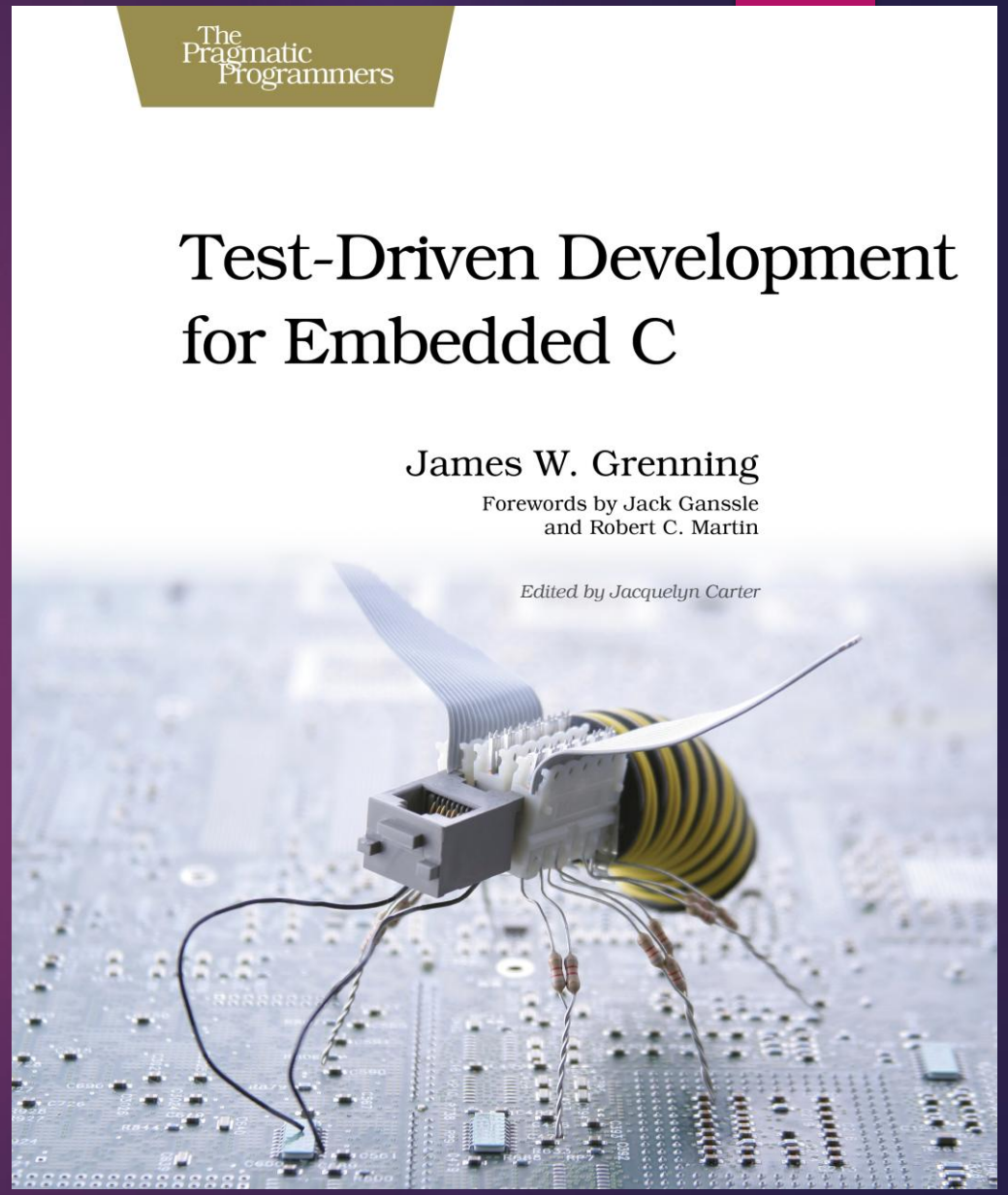
- Git managed workflow, requiring code-review on gitlab.
- Developers work on feature branches.
  "A branch for every feature!" ~ Scott Chacon (founder github.com).
- Master branch is locked to developers.
- Integrator integrates, and merges feature branches after a lightweight review process.

- Some CAN design collaboration/review.
- Could be more…

TDD – A Next Step?

- Can this work for embedded projects?
- Can it help us develop faster?
- Can we write reliable software?
- Can we prevent regressions?
- Can we prove the defined behaviour of a complex system?
- Will it be boring?

- Obvious answer… not possible.
- Real answer… read the book!



BTW – 3 copies, I want to keep 1.


maintainable
private
kernel
abstract
fundamental
abstraction
reliable
workflow
interface
implementation
guaranteed
FreeRTOS
framework
developers
TDD
configurable
resources
code
interfaces
strong
development
encapsulation
opaque
unified
testable
collaboration
typesafe
robust
open
platform
extensible
source
processes
drivers
baremetal
opensource
separation of concerns
linux
reuse
embedded
handle
architecture
model
devices