Monitors & Boot-Loaders

Mark McDermott

Fall 2009

Agenda

- Monitors
- Boot-Loaders
- Booting Embedded Linux

Monitors

- A monitor provides, in addition to booting capabilities, a command-line interface that can be used for debugging, reading/writing memory, flash reprogramming, configuring, etc. The features of a monitor generally include:
  - Extensible built-in flash file system (TFS)
    - Support for JFFS2 and FAT
  - TFTP client/server for network file transfer
  - Xmodem for serial file transfer
  - On-board ASCII file creation (i.e. target resident file editor)
  - File-based scripts with conditional branching
  - ASCII-script-driven startup options
  - Command line history and editing
  - UDP and RS232 based command entry
  - Versatile configuration management using files
  - Symbols and shell variables
  - Symbolic display of variables, stack trace, runtime profiling and memory-based runtime trace
  - GDB server for application loads and post-mortem analysis.
  - Network host supporting ICMP and DHCP/BOOTP as a startup option
  - Large API to hook the application to facilities provided by monitor.

Micro-Monitor (µMON)

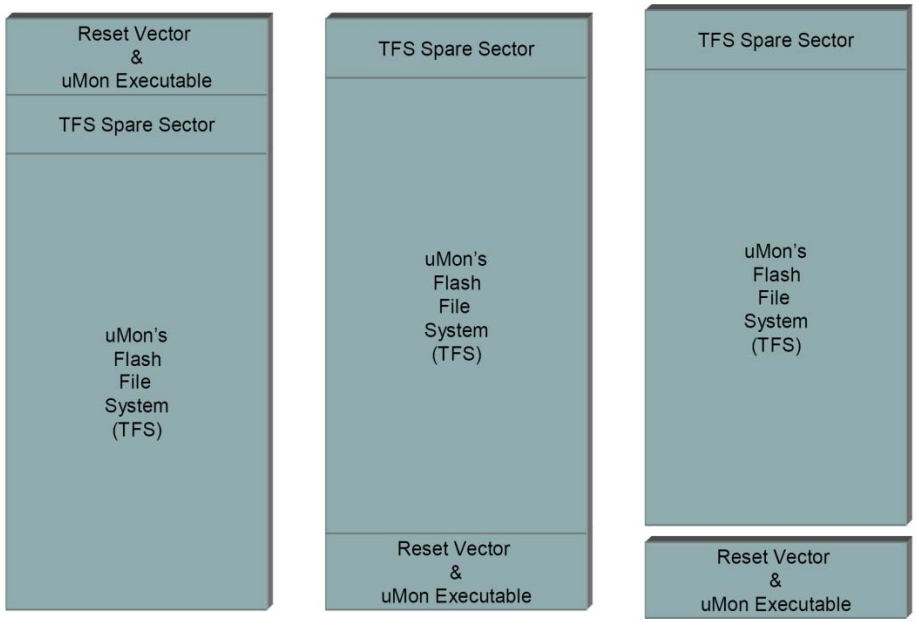
- Micro-Monitor is used as a startup environment for many different embedded system applications.
- There are three main types of applications:
  - Those that run single-threaded, without any need for an embedded operating system.
  - Those that run multi-threaded, using an RTOS that runs with a flat, basically unprotected memory space.
  - Those that run multi-processed, using an RTOS or Embedded Operating System (Linux) that provides memory protection between processes.

uMon and Embedded Linux

- **Embedded Linux presents two issues that need to be dealt with when using uMon as the boot loader.**
  - Embedded Linux runs with the MMU enabled in such a way that it is impossible to access the uMon API from user space.
  - TFS is not a native file system to Linux.
- **The solution depends on the problem; hence, there are several different configurations, each of which have their own applicability depending on the system requirements. These include:**
  - TFS is the only file systems
  - TFS for boot, Linux FS for runtime
  - TFS with Access to JFFS2 and/or FAT

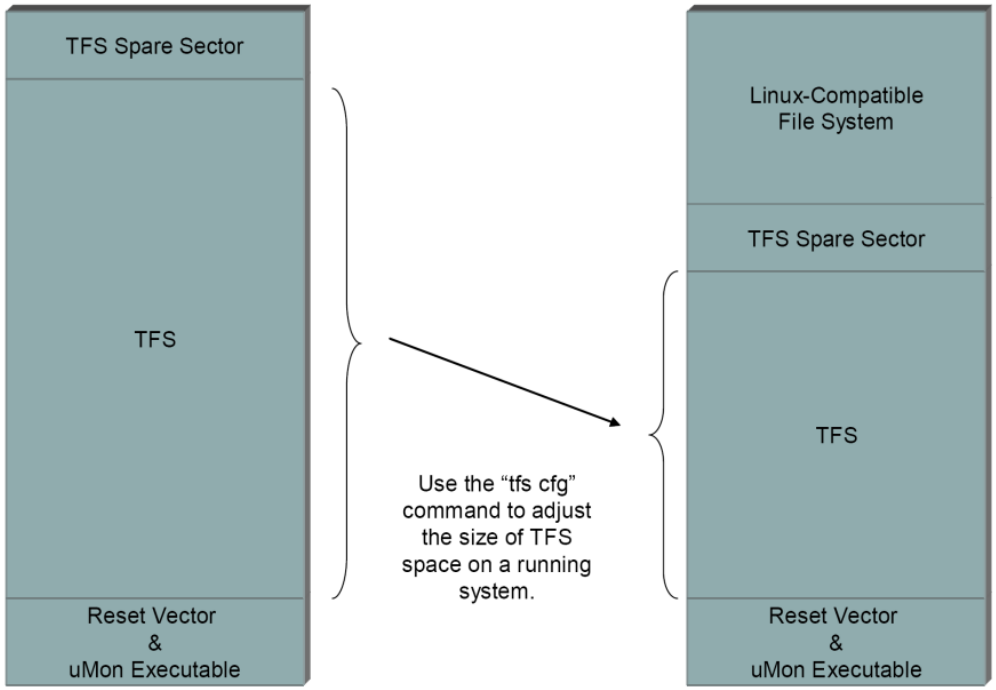
Basic uMon Memory Map: TFS

- A simple uMon-based embedded target has a flash map that is broken up into two main sections: the uMon executable and TFS.
- The relative position of the uMon executable and TFS is arbitrary. The executable may reside above TFS, below TFS or even on a separate device that is in non-contiguous memory space.



Memory Map: TFS for boot, Linux FS for runtime

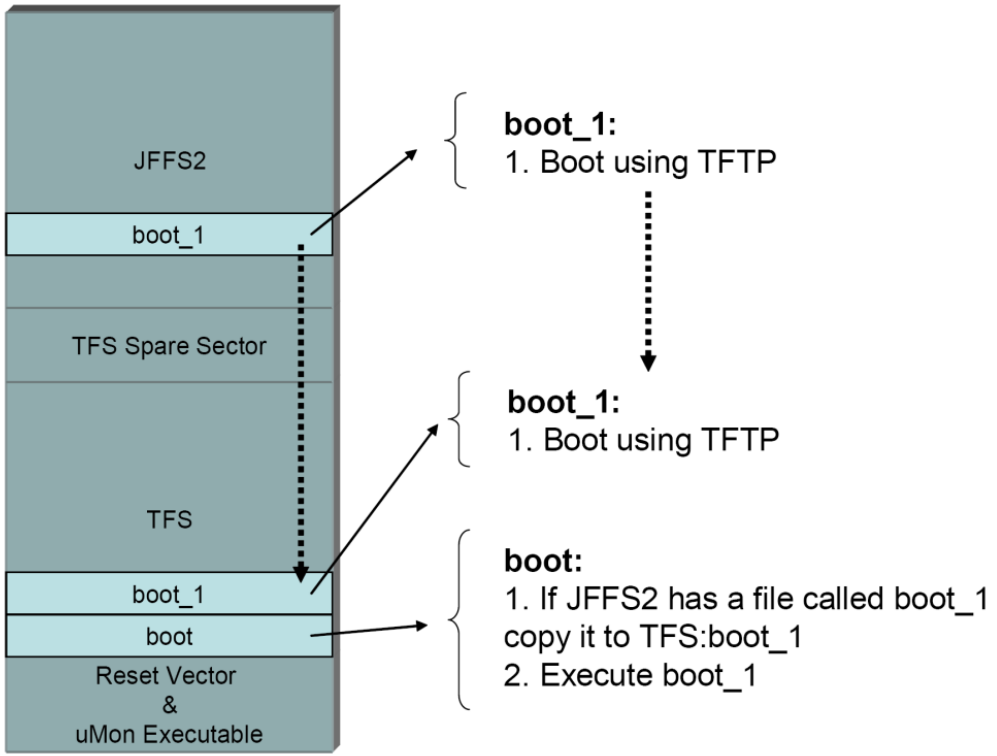
- Slightly more complicated flash footprint than basic TFS.
- Need a section of memory for the boot monitor executable and for TFS storage space
  - TFS storage space is reduced and some other Linux-compatible FFS is overlayed on a portion of the flash.



- All images are in a single flash device containing the uMon executable, TFS and a Linux-compatible FFS. The space allocated to TFS can be adjusted at runtime using the "tfs cfg" command in uMon.

Memory Map: TFS with Access to JFFS2 and/or FAT

- Flash footprint is considerably more complex.
  - Improved versatility
- JFFS2 and FAT file formats are supported
- Allows for flexible and scriptable boot strategy
  - uMon can query the non-TFS FS for a specified file
  - uMon can copy that file from the non-TFS FS into TFS or memory.



uMon Command Summary

- arp: Address resolution protocol
- call: Call embedded function
- cm: Copy Memory
- dhcp: Issue a DHCP discover
- dis: Disassemble memory
- dm: Display Memory
- echo: Print string to local terminal
- edit: Edit file or buffer
- ether: Ethernet interface
- exit: Exit a script
- flash: Flash memory
- gosub: Call a subroutine
- goto: Branch to file tag
- heap: Display heap statistics.
- help: Display command set
- history: Display command history
- icmp: ICMP interface
- if : Conditional branching
- item: Extract an item from a list
- mt: Memory test
- mtrace: Configure/Dump memory trace.
- pm: Put to Memory
- read: Interactive shellvar entr
- reg: Display/modify content of monitor's register cache
- reset: Reset monitor firmware
- return: Return from subroutine
- set: Shell variable operations
- sleep: Second or msec delay (not precise)
- sm: Search Memory
- strace: Stack trace
- ulvl: Display or modify current user level.
- tftp: Trivial file transfer protocol
- tfs: Tiny File System Interface
- unzip: Decompress block of memory.
- xmodem: Xmodem file transfer y

uMon Startup Environment

- During startup the uMon firmware does basic initialization of the target system hardware. This includes:
  - configuring flash and RAM accesses
  - configuring a serial and/or Ethernet port
- The configuration process creates information that is useful to the user, and some of it requires information from the user to properly complete:
  - Board MAC Address
  - Board IP Address
  - Netmask
  - Gateway IP Address
  - Console Baud Rate
- The ‘monrc’ file is used for this purpose.
  - Type ‘tfs cat monrc’ display contents of file

File Transfer Capability

- **xmodem**
  - RS232 based file transfer protocol
    - Very slow but useful if ENET port is not working
  - Supports the ability to download a binary image and automatically invoke the flash command sequence needed to reprogram the boot flash
    - Dangerous but quite useful ☺

- **tftp: trivial file transfer protocol**
  - Requires ENET port to be working
  - Connectivity must be established with target machine
    - Not easy with Windows based machines
  - uMon supports both client and server modes of operation for TFTP. By default, a TFTP server is always running.
  - The “tftp” command on the target is primarily used for the mode where the target is the client;

uMon Application Startup

- uMon system startup can range from a single auto-bootable executable file installed in TFS, to a series of scripts that require an understanding of TFS, the scripting capabilities, shell variables and DHCP/BOOTP.
- Examples of how uMon can be configured to startup a system:
  - A configuration that simply has an autobootable application
  - A configuration using a script to start up an application
  - A configuration that uses the network to retrieve its IP, then boots an application
  - A configuration that uses DHCP/BOOTP/TFTP to do a complete network-based bootup
  - A configuration that boots off the network, but is also prepared to run standalone in the event that the network server is down.

Application Programming Interface

- uMon provides an extensive API that can be used to connect the application software to the services provided by the uMon kernel. API functions include:
  - monConnect: Connect the application to the monitor.
  - mon_addcommand: Add an application-specific command list to the monitor.
  - mon_appexit: Allows the application to return control to the monitor.
  - mon_crc32: CRC32 calculation.
  - mon_decompress: Decompress a block of data that has been previously compressed.
  - mon_delay: A millisecond-resolution delay loop
  - mon_docommand: A mechanism by which the application can invoke a command that is part of the monitor.
  - mon_flasherase: Erase a sector of flash using uMon's flash operations
  - mon_flashinfo: Retrieve information (base address and size) of a specified flash sector.

Application Programming Interface (cont)

- **Additional API functions include:**
  - **mon_free: Similar to standard free()**
  - **mon_getargv: Retrieve an argument list**
  - **mon_getchar: Provide similar functionality as standard getchar().**
  - **mon_getline: Retrieve characters from console**
  - **mon_intsoff: Allows the application to turn off interrupts**
  - **mon_intsrestore: Allows the application to restore interrupt state**
  - **mon_malloc: Similar to standard malloc()**
  - **mon_portcmd: An API call that is 100% port-specific, defined by the port originator/maintainer**
  - **mon_printf: Similar to printf() but limited in the formatting capability**
  - **mon_profile: Call the monitor's profiling facility**
  - **mon_putchar: Provide similar functionality as standard putchar().**

Agenda

- Monitors
- **Boot Loaders**
- Real Time Operating Systems

Boot Terminology

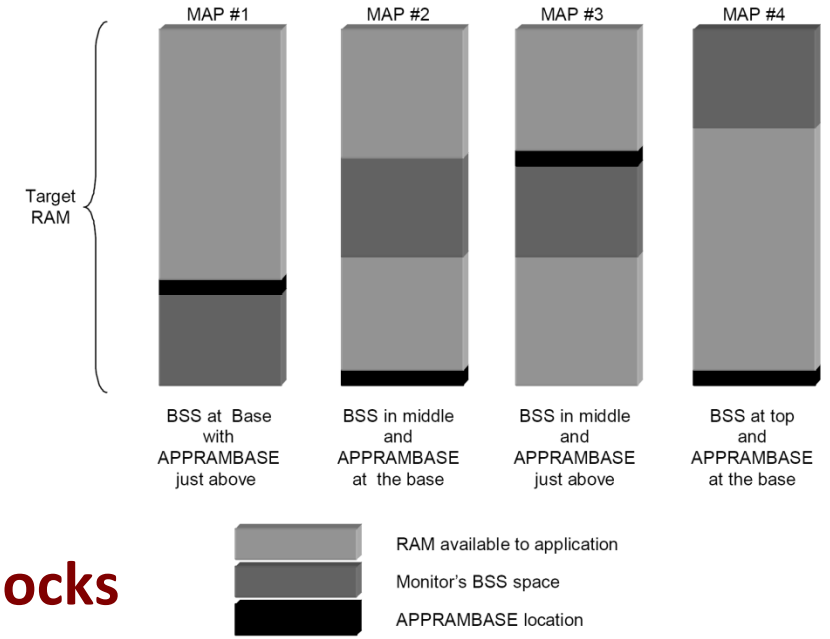
- **Loader:**
  - Program that moves bits from disk (usually) to memory and then transfers CPU control to the newly "loaded" bits (executable).
- **Boot-loader / Bootstrap:**
  - Program that loads the "first program" (the kernel).
- **Boot PROM / PROM Monitor / BIOS:**
  - Persistent code that is "already loaded" on power-up.
- **Boot Manager:**
  - Program that lets you choose the "first program" to load.

Linux Open Source Boot-Loaders

Bootloader	Monitor	Description	X86	ARM	MIPS	Black Fin	PPC
LILO	No	The main disk bootloader for Linux	X				
GRUB	No	GNU's successor to LILO	X				
ROLO	No	Loads Linux from ROM without a BIOS	X				
EtherBOOT	No	ROMable loader for booting systems through Ethernet cards	X				
LinuxBIOS	No	Linux-based BIOSreplacement	X				
YABOOT YAMON	Yes	Loader for MIPS processors			X		
U-Boot	Yes	Universal loader based on PPCBoot and ARMBoot		X			X
RedBOOT	Yes	eCos-based loader	X	X	X	X	X
uMON	Yes	Universal loader and monitor		X	X	X	X

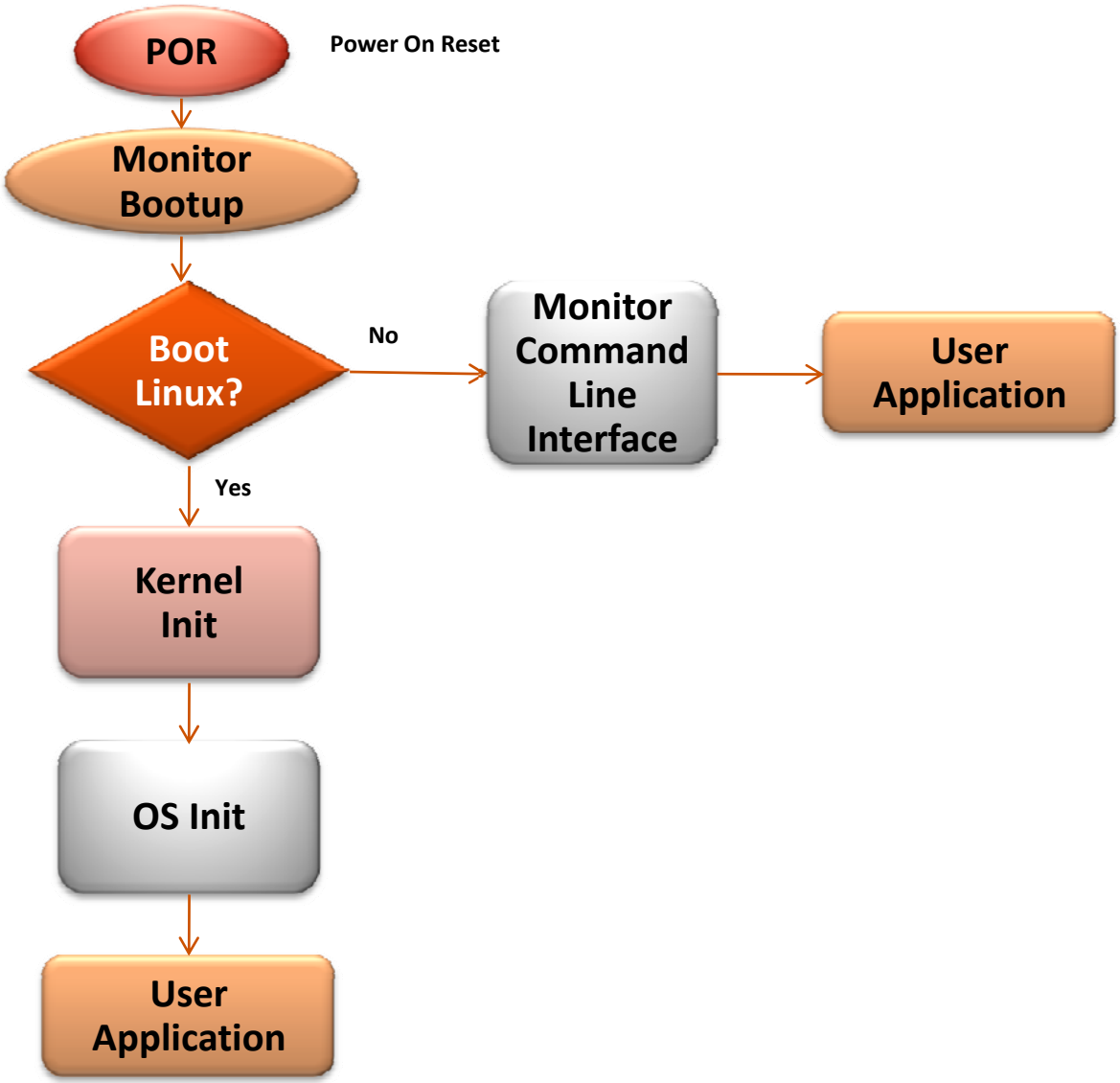
Boot-loader Operation

- Stop all interrupts
- Stop watchdog timer
- Initiate memory interface
- Copy DATA & BSS section
- Allocate system stacks
- Initiate other items, such as Clocks & GPIO
- Detect user input and decide what to do
  - Run diagnostic program
  - Update kernel / data
  - Jump to kernel (normal boot-up)



BSS (Block Started by Symbol) is the name of the data segment containing static variables that are filled solely with zero-valued data initially (ie, when execution begins). It is often referred to as the "bss section" or "bss segment". The program loader initializes the memory allocated for the bss section when it loads the program.

TLL6219 System bootup flow



uMon Boot-loader

- uMon performs the following functions when booting Linux
  - System reset and basic CPU initialization (initialize cache & interrupts, etc…)
  - Kernel and/or rootfs are copied/decompressed to DRAM.
  - Initialization of site dependent data (start & size of memory, console baud rate, etc…) passed from boot-loader to Linux kernel.
    - Via the LDATAGS mechanism
  - Transfer of control from boot-loader to the kernel through the kernel's entry point.

ARM LDATAGS Mechanism

- LDATAGS are a tagged structure that is read by the kernel.
- The structure parameters are passed to the kernel's entry point as a pointer to this structure.
- The LDATAGS command has a hardcoded tag structure that can be modified by the command line.
  - If a new member of the tags structure is needed, then the LDATAGS command code must be modified and the monitor must be updated.

```
struct binfo {
long        memstart;
long        memsize;
long        flashstart;
long        flashsize;
long        flashoffset;
long        sramstart;
long        sramsize;
long        bootflags;
long        ip_addr;
char        enetaddr[6];
short       ethspeed;
long        intfreq;
long        busfreq;
long        baudrate;
char        s_version[4];
char        r_version[32];
long        procfreq;
long        pci_busfreq;
char        pci_enetaddr[6];
char        cmdline[256];
}
```

uMon Generalized Linux Boot Script

- A high level view of the script to the right shows four main sections:
  - The structure declaration,
  - The structure initialization
  - The file preparation
  - the transfer of control from uMon to kernel.
- The following script define the board-information structure used by the kernel at startup.

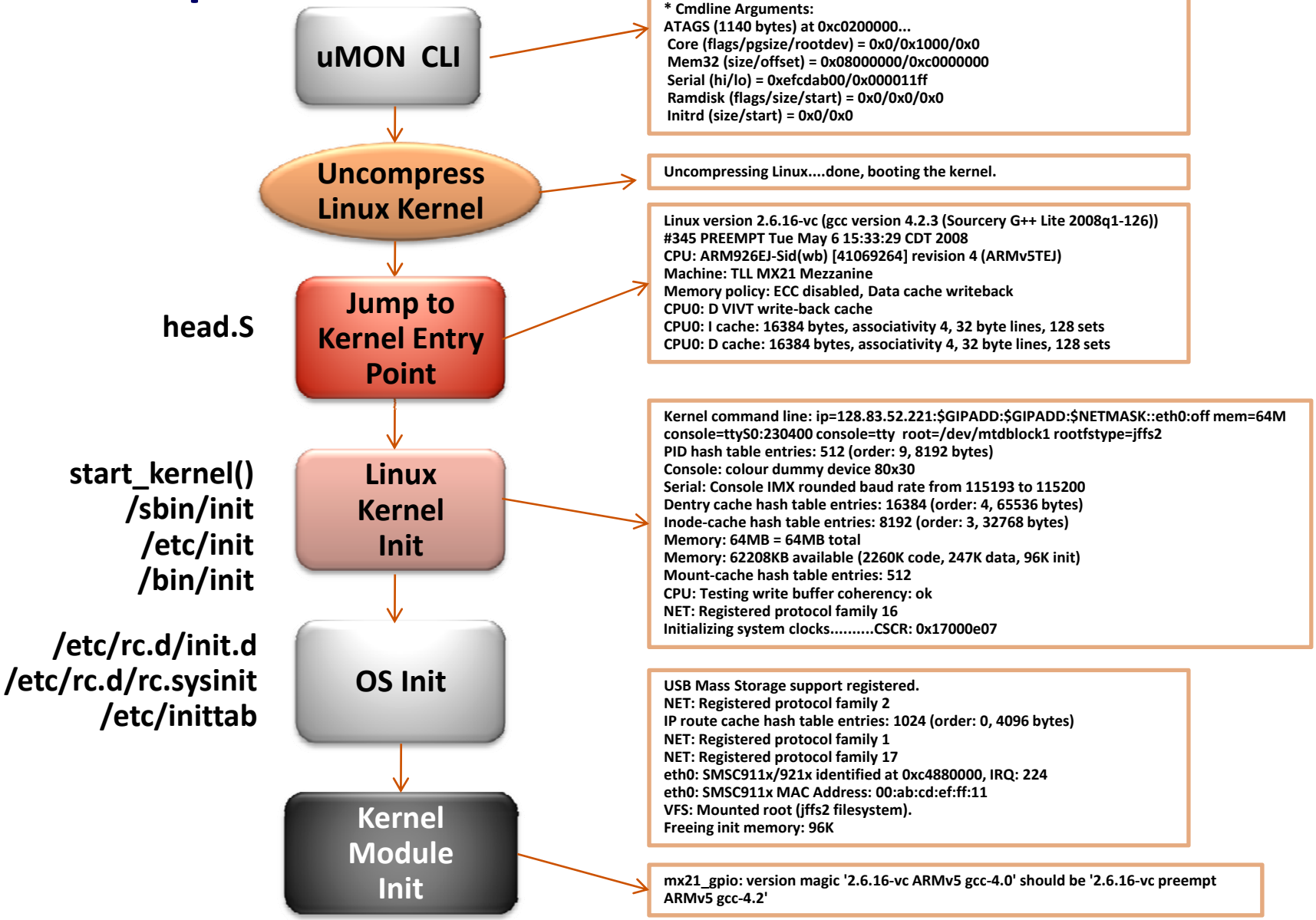
```
tfs size initrd.img.gz RAMDISK_SIZE
tfs base initrd.img.gz RAMDISK_BASE
unzip vmlinux.bin.gz 0

set STRUCTBASE 0x03000000
 set STRUCTFILE $ARG0
struct binfo=0
struct binfo.memsize=0x08000000
struct binfo.intfreq=300000000
struct binfo.busfreq=100000000
struct binfo.enetaddr[6]=e2b(${ETHERADD})
struct binfo.baudrate=${CONSOLEBAUD}
struct binfo.cmdline[256]=strcpy("console=tty1")
struct binfo.cmdline[256]=strcat(" console=ttyS0,${CONSOLEBAUD}“)
struct binfo.cmdline[256]=strcat(" ip=on“)
struct binfo.cmdline[256]=strcat(" root=/dev/ram rw")
struct binfo.cmdline[256]=strcat(" ramdisk_size=${RAMDISK_SIZE}")
struct binfo.cmdline[256]
set CMDLINE=hex(${STRUCTBASE}+${STRUCTOFFSET}) pm -S
    $CMDLINE ""
call 0 $STRUCTBASE $RAMDISK_START $RAMDISK_END
    $CMDLINE $STRLEN
exit
```

TLL6219 Linux Boot Script

```
#### TFS Files
set KERNEL_IMG zImage

# Memory Config
set MEMSIZE 0x8000000
set MEMBASE $APPRAMBASE
set -a MEMBASE 0xf0000000

# Get Kernel image
tfs size $KERNEL_IMG KERNEL_IMG_LEN
tfs base $KERNEL_IMG KERNEL_IMG_ADDR

echo * Loading Kernel: $KERNEL_IMG
echo *     Src: $KERNEL_IMG_ADDR Dst:
    $APPRAMBASE
echo *     Len: $KERNEL_IMG_LEN
echo *

#Prepare Kernel argument list
set ATAGS_ADDR $APPRAMBASE
set -i ATAGS_ADDR 0x0
ldatags -c -a $ATAGS_ADDR
ldatags -m -i -a $ATAGS_ADDR core_pgsize=4096
    core_flags=0 core_rootdev=0 mem32_size=$MEMSIZE
    mem32_start=$MEMBASE cmdline=""
ldatags -a $ATAGS_ADDR cmdline_append="ip=dhcp "
ldatags -a $ATAGS_ADDR cmdline_append="mem=64M
    console=ttyS0:230400 console=tty0  "
ldatags -a $ATAGS_ADDR
    cmdline_append="root=/dev/mtdblock1 rootfstype=jffs2 "

echo * Cmdline Arguments:
ldatags -v -a $ATAGS_ADDR cmdline_append=""

# Call the Kernel entry
set KERNEL_RAM_ADDR $ATAGS_ADDR
set -x -i KERNEL_RAM_ADDR 4096
tfs cp $KERNEL_IMG $KERNEL_RAM_ADDR

echo * Calling $KERNEL_RAM_ADDR with args
    $ATAGS_ADDR
echo ****************************************
echo
echo

call $KERNEL_RAM_ADDR 0 0 $ATAGS_ADDR

exit
```

Booting Embedded Linux

Linux Bootup flow



Kernel Entry Point: head.S

- Performs the following functions:
  - Checks for valid processor and architecture
  - Creates the initial page table entries
  - Enables the processors memory management unit
  - Establishes limited error detection and reporting
  - Jumps to the start of kernel proper, main.c

```
Linux version 2.6.16-vc (gcc version 4.2.3 (Sourcery G++ Lite 2008q1-
126)) #345 PREEMPT Tue May 6 15:33:29 CDT 2008
CPU: ARM926EJ-Sid(wb) [41069264] revision 4 (ARMv5TEJ)
Machine: TLL MX21 Mezzanine
Memory policy: ECC disabled, Data cache writeback
CPU0: D VIVT write-back cache
CPU0: I cache: 16384 bytes, associativity 4, 32 byte lines, 128 sets
CPU0: D cache: 16384 bytes, associativity 4, 32 byte lines, 128 sets
```

```
/*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr.
 *
 * This code is mostly position independent, so if you link the kernel at
 * 0xc0008000, you call this at __pa(0xc0008000).
 *
 * See linux/arch/arm/tools/mach-types for the complete list of machine
 * numbers for r1.
 *
 * We're trying to keep crap to a minimum; DO NOT add any machine specific
 * crap here - that's what the boot loader (or in extreme, well justified
 * circumstances, zImage) is for.
 */
    __INIT
    .type   stext, %function
ENTRY(stext)
    msr     cpsr_c, #PSR_F_BIT | PSR_I_BIT | MODE_SVC       @ ensure svc mode
                                                            @ and irqs disabled
    bl      __lookup_processor_type                         @ r5=procinfo r9=cpuid
    movs    r10, r5                                         @ invalid processor (r5=0)?
    beq     __error_p                                       @ yes, error 'p'
    bl      __lookup_machine_type                           @ r5=machinfo
    movs    r8, r5                                          @ invalid machine (r5=0)?
    beq     __error_a                                       @ yes, error 'a'
    bl      __create_page_tables

    /*
     * The following calls CPU specific code in a position independent
     * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
     * xxx_proc_info structure selected by __lookup_machine_type
     * above.  On return, the CPU will be ready for the MMU to be
     * turned on, and r0 will hold the CPU control register value.
     */
    ldr     r13, __switch_data                              @ address to jump to after
                                                            @ mmu has been enabled
    adr     lr, __enable_mmu                                @ return (PIC) address
    add     pc, r10, #PROCINFO_INITFUNC
```

Kernel Entry Point: …/init/main.c

- **Performs the following functions:**
  - This code starts up the kernel.
  - The "command line" from the boot loader is parsed and executed.
  - The cache is configured at this time
  - The scheduler initialized while preemption is turned off.
  - The various timers and interrupts are then initialized.

```
Kernel command line:
ip=128.83.52.221:$GIPADD:$GIPADD:$NETMASK::eth0:off mem=64M
console=ttyS0:230400 console=tty  root=/dev/mtdblock1 rootfstype=jffs2
PID hash table entries: 512 (order: 9, 8192 bytes)
Console: colour dummy device 80x30
Serial: Console IMX rounded baud rate from 115193 to 115200
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 64MB = 64MB total
Memory: 62208KB available (2260K code, 247K data, 96K init)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
NET: Registered protocol family 16
Initializing system clocks.....CSCR: 0x17000e07
```

```
asmlinkage void __init start_kernel(void)
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    page_address_init();
    printk(KERN_NOTICE);
    printk(linux_banner);
    setup_arch(&command_line);
    setup_per_cpu_areas();

    /*
     * Mark the boot cpu "online" so that it can call console drivers in printk() and can access its
      per-cpu storage.
     */
    smp_prepare_boot_cpu();

    /*
     * Set up the scheduler prior starting any interrupts (such as the timer interrupt). Full
      * topology setup happens at smp_init() time - but meanwhile we still have a
      * functioning scheduler.
     */
    sched_init();
    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */
    preempt_disable();
    build_all_zonelists();
    page_alloc_init();
    printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
    parse_early_param();
    parse_args("Booting kernel", command_line, __start___param,
            __stop___param - __start___param,
            &unknown_bootoption);
    sort_main_extable();
    trap_init();
    rcu_init();
    init_IRQ();
    pidhash_init();
    init_timers();
    hrtimers_init();
    softirq_init();
    time_init();
```

/sbin/init

- Ancestor of all processes (except idle/swapper process).
- Init parses the /etc/inittab file to determine the specifics of what programs to run and at what level. NOTE Busybox does not support Run Levels.
  - 0: used to halt the system. The system performs an init 0 command and the system is halted.
  - 1: Puts the system into single-user mode.
  - 2: Puts the system into a multiuser mode but does not support networking.
  - 3: Puts the system into the standard full multiuser mode but does not automatically start X.
  - 4: User defined.
  - 5: Puts the system into standard multiuser mode with a graphical (X-based) login.
  - 6: Shutdown DO NOT use inittab to set this
- Starts the /etc/rc.d/rc script with the appropriate run level.
  - The rc script executes all of the scripts pointed to by the symblic links contained in the directory for that run level.
  - For example, if the run level is 3, the scripts pointed to by the links in /etc/rc.d/rc3.d are run.
- On the TLL6219 “busybox” handles the init function.

TLL6219 /etc/inittab

```
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
#
# Note: BusyBox init doesn't support runlevels.
# The runlevels field is completely ignored by BusyBox init.
# If you want runlevels, use sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id       == tty to run on, or empty for /dev/console
# runlevels == ignored
# action    == one of sysinit, respawn, askfirst, wait, and once
# process   == program to run
```

```
# Startup the system
#null::sysinit:/bin/mount -o remount,rw /
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname
null::sysinit:/sbin/ifconfig lo 127.0.0.1 up
null::sysinit:/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 lo
# now run any rc scripts
::sysinit:/etc/init.d/rcS
```

```
# Set up a couple of getty's
#tty1::respawn:/sbin/getty 38400 tty1
#tty2::respawn:/sbin/getty 38400 tty2

# Put a getty on the serial port
#ttyAM0::respawn:/sbin/getty -L ttyAM0 38400 vt100

# Put a getty on the serial port
#ttyAM0::respawn:/sbin/getty -L ttyAM0 38400 vt100

# Put a shell on the default console
ttyS0::respawn:/bin/sh

# Logging junk
#null::sysinit:/bin/touch /var/log/messages
#null::respawn:/sbin/syslogd -n -m 0
#null::respawn:/sbin/klogd -n
#tty3::respawn:/usr/bin/tail -f /var/log/messages

# Stuff to do for the 3-finger salute
::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
null::shutdown:/usr/bin/killall klogd
null::shutdown:/usr/bin/killall syslogd
null::shutdown:/bin/umount -a -r
null::shutdown:/sbin/swapoff -a
```

Normally run from the rc.sysinit script

/etc/rc.d/rc3.d

- All the files here are only symbolic links to the actual scripts that exist in /etc/rc.d/init.d.
- The system first runs the scripts whose names start with K to kill the associated processes → /etc/rc.d/init.d/<command> stop
- The system runs the scripts whose names start with S to start the processes → /etc/rc.d/init.d/<command> start
- Changing a K name to start with S (e.g., K20nfs → S20nfs) makes Linux start the process rather than kill it.

/etc/rc.sysinit

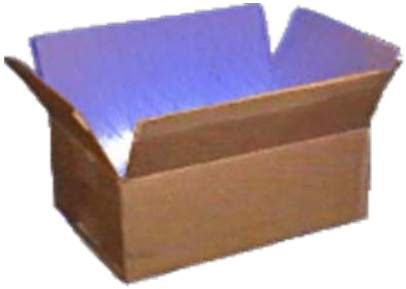
- Run once at boot time
- Sets the path and the hostname, and checking whether networking is activated.
- Mounts the /proc filesystem
- Sets the kernel parameters
- Sets the system clock
- Loading keymaps and fonts
- Start swapping
- Initializing the USB controller along with the attached devices.
- Checks the root filesystem for consistency
  - /sbin/fsck
- Remounts the root filesystem as read-write.
- Loads kernel modules as appropriate.
  - On the TLL6219 the following command is executed to install the GPIO command function:
  - insmod /etc/modules/mx21_gpio.ko

Shutdown

- Use /bin/shutdown to avoid data loss and filesystem corruption.
- Shutdown inhibits login, asks init to send SIGTERM to all processes, then SIGKILL
  - Clears out all buffered writes using sync command. Not guaranteed to work due to the non-deterministic nature of disk sub-systems.
- Low-level commands: halt, reboot, poweroff.
  - Use -h, -r or -p options to shutdown instead.
- Ctrl-Alt-Delete “Vulcan neck pinch”:
  - defined by a line in /etc/inittab
  - ca::ctrlaltdel:/sbin/shutdown -t3 -r now.

Embedded Linux Shell (aka CLI)

- Once the init process is completed the "shell" process is started.
- On most embedded Linux systems a general purpose toolbox is used: BusyBox
  - Most Unix command line utilities within a single executable! It even includes a web server!
  - Sizes less than < 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc).
  - Easy to configure which features to include.
  - The best choice for
    - Initramfs / initrd with complex scripts
    - Small and medium size embedded systems



http://www.busybox.net/

BusyBox will be covered in more detail in the Middle-Ware lecture