Linux Loadable Kernel Modules (LKM)

- A way dynamically ADD code to the Linux kernel
- LKM is usually used for dynamically add
  - device drivers
  - filesystem drivers
  - system calls
  - network drivers
  - executable interpreters

Why use LKMs

- Need not to rebuild kernel
- Diagnosing system problems
  - Easier to locate in which part of the kernel problems occur
- Modules are faster to maintain and debug
- LKMs are not slower than base kernel parts
- However, if the system startup is dependent on a module, it has to be included in the base kernel
  - E.g. File system driver

Configuring the kernel

- Before building the kernel, it has to be configured:
  - `make config/menuconfig/xconfig`
  - Select drivers into base kernel / as loadable module / skip
- Kernel is builded with
  - make zImage
- Modules are builded with
  - make modules

Kernel menuconfig

```
Linux Kernel v2.4.21 Configuration
 qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
  lqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq File systems qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
  x  Arrow keys navigate the menu.  <Enter> selects submenus --->.          x
  x  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,  x
  x  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.      x
  x  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable      x
  x lqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk x
  x x   [ ] Quota support                                                  x x
  x x   < > Kernel automounter support                                     x x
  x x   <*> Kernel automounter version 4 support (also supports v3)        x x
  x x   <M> Reiserfs support                                               x x
  x x   [ ]   Enable reiserfs debug mode                                   x x
  x x   [*]   Stats in /proc/fs/reiserfs                                   x x
  x x   <M> Ext3 journalling file system support                           x x
  x x   [ ]   JBD (ext3) debugging support                                 x x
  x x   < > DOS FAT fs support                                             x x
  x x   < > Compressed ROM file system support                             x x
  x x   [*] Virtual memory file system support (former shm fs)             x x
  x x   <*> ISO 9660 CDROM file system support                             x x
  x x   [ ]   Microsoft Joliet CDROM extensions                            x x
  x mqqqv(+)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj x
  tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
  x                    <Select>    < Exit >    < Help >                       x
  mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

Placement of standard modules

- Standard modules (distributed with the kernel) are located in
  /lib/modules/<kernel-version>
  - different subdirs depending on contents of the modules
    - kernel/arch, kernel/drivers, kernel/fs, kernel/net
- Own modules
  - can in principle be anywhere

Own loadable modules

- Modules not part of Linux (not distributed with the Linux kernel)
- Modules are always ELF-object files (.o)
  - (In Linux 2.6 extension: .ko)

The “HelloWorld” module

```
// Hello.c
// test kernel module

#include <linux/module.h> //Needed by all modules
#include <linux/kernel.h> //Needed for KERN_ALERT
#include <linux/init.h>   //Needed for macros

MODULE_AUTHOR("Jerker Bjorkqvist");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A minimal Linux Kernel module");

static int hello_init(void) {
  printk(KERN_ALERT "Hello, world\n");
  return 0; // =Success
}

static int hello_exit(void) {
  printk(KERN_ALERT "Goodbye, world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Compiling the module

- Linux 2.4.x
  - `gcc -c -O2 -W -Wall -isystem /lib/modules/`uname -r`/build/include Hello.c`
- Linux 2.6.x
  - New module build system
    - must use makefiles

```
// Makefile
obj-m: Hello.o

$make -C /path/to/source SUBDIRS=$PWD modules
```

Inserting the module

- 2.4.x: `$ insmod Hello.o`
- 2.6.x: `$ insmod Hello.ko`
- In general: `$ modprobe Hello`
- Checking the module insertion (any string may be written...):

```
$ dmesg | tail
EXT3 FS on hda3, internal journal
EXT3-fs: mounted filesystem with ordered data mode.
Hello, world
```

Command line arguments to module

- Not the normal argc/argv-way
- Macro MODULE_PARM()

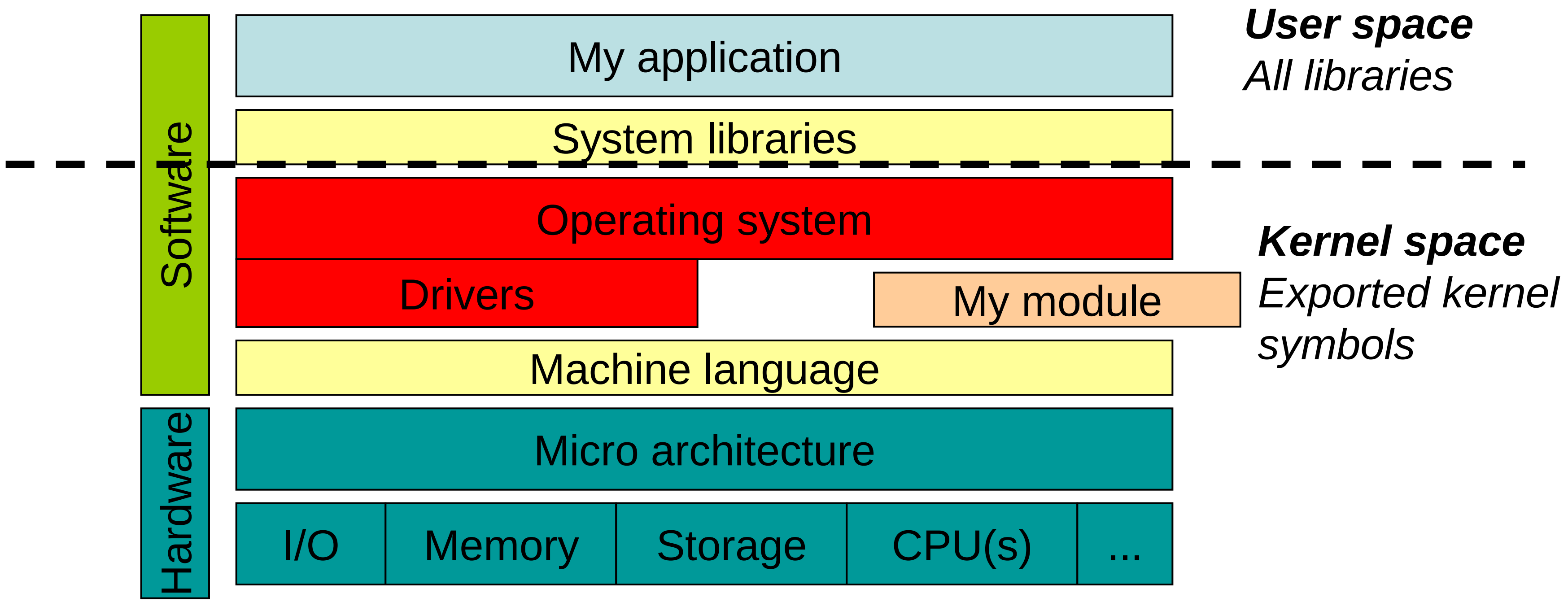
```
int myint = 3;
char *mystr;
MODULE_PARM(myint, "i");
MODULE_PARM(mystr, "s");
```

- Usage: Example IO-port settings for module

Modules vs. Programs

	C-Program	**Module**
Program start	main()	init_module module_init() MACRO
Program end	exit() return from main()	cleanup_module() module_exit() MACRO
Libraries	Standard libraries (libc)	No libraries, only functions exported by kernel
Environment	User space, safe environment	Kernel space
Memory	Process virtual memory space	Kernel´s code / dataspace

Modules vs. programs



Name space and kernel code

- Variable names should be meaningful!
- However, if using global variables, variable names can clash (namespace pollution)
- Kernel code (e.g. module): code will be linked against complete kernel
  - Static variables
  - Well-defined prefix for your symbols
  - If symbol needed for rest of world
    - EXPORT_SYMTAB/EXPORT_SYMBOL() macro

Memory space

- Kernel has separate memory space from user process
- Special macros to access user space data from kernel side
  - get/put_user(x, ptr)
  - copy_to/from_user(to, from, size)
- Allocating memory
  - kmalloc() / kfree() – kernel memory
  - vmalloc() / vfree() – virtual memory in kernel space

Module programming

- A fault in kernel code is fatal to the current process and sometimes to the whole system
- Modules must support concurrency (calls by different processes). Distinct data for different processes
- Driver code must be reentrant: local (stack allocated) variables / dynamic mem allocation
- The code might be interrupted
- sleep_on(wait_queue) to yield processor
- /proc/ioport lists current ports. /proc/iomem memory

Module programming

- No floating point, no MMX
  - The FPU context is not saved
- Stack limit
  - Kernel stack about 6K in 2.2
    - No recursion!!!
- Portable code:
  - Minimize CPU specific
  - Minimize architecture dependent

Device driver

- A driver is
  - A set of routines that implements the device-specific aspects of generic I/O operations
- The operation system handles the device independent I/O aspects
  - A transparent API for accessing devices
  - If a device is replaced, the application software does not need to be altered
- Driver in kernel / application?
  - Word perfect: Printer device drivers in application
  - Windows ->: Printer device drivers in OS

C or C++ for driver development?

- In general C is a better choice
  - Advanced OOP features can cause code bloat
  - C++ compilers can generate many routines for a single function
  - Virtual methods and polymorphism slow program launch times significantly
- Size really *does* matter

Hello World char device

```
static struct file_operations fops= {
  .read = hello_read,
  .write = hello_write,
  .open = hello_open,
  .release = hello_release
};

static int hello_open(struct inode *inode, struct file *fp) {
  // Create a message for the opener
  sprintf(msg, "Hello PID %i, Greetings from device %i", current->pid, Major\
);
  return 0;
}
static int hello_release(struct inode *inode, struct file *fp) {
  return 0;
}

static ssize_t hello_read(struct file *fp, char *buf, size_t l, loff_t *off)\
 {
  size_t count=0;
  for (; msg[*off+count] != 0 && count<l && *off+count < MESSAGE_LENGTH; cou\
nt++) {
    put_user(msg[*off+count], &buf[count]);
  }
  *off += count;
  return count;
}

static ssize_t hello_write(struct file *fp, const char *buf, size_t l, loff_\
t *off) {
  return 0;
}
```

IOCTL

- Given a serial line interface, reading / writing corresponds to reading / sending bits on the line
  - How to send control to the actual serial line interface (setting baud-rates, stop-bits etc) ??
- Devices files have a special function ioctl() to control the device
  - *ioctl(int fd, int ioctl_nr, long par)*

Drivers and interrupts

- To request an interrupt
  - request_irq(int irq, void (*handler), long flags, char *devname, void *devid)
    - handler(int irq, void *devid, struct pt_regs *regs)
  - To types of interrupts
    - fast (flags = SA_INTERRUPT)
    - slow
  - Interrupts can be shared (flags = SA_SHIRQ)

Device drivers in NT

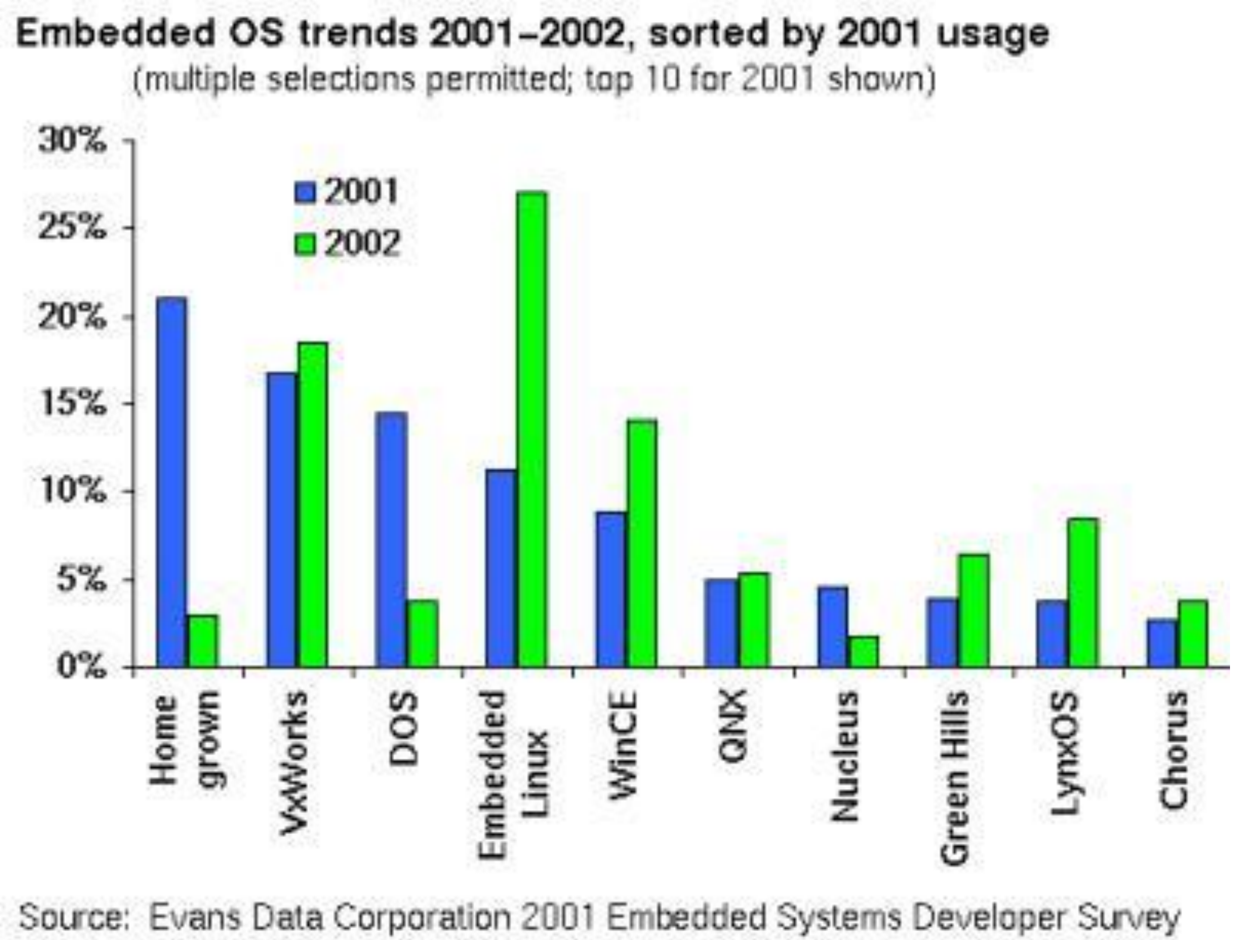
- Virtual Device Drivers (VDD)
  - Win32 DLL with specific entry point and installation requirements
  - Alloc 16 bit applications to ”access” certain I/O addresses
- Win32 Graphics Drivers (GDI)
  - implements video controller-specific or printer-specific aspects of GDI function
- Kernel Mode Drivers (KMD)
  - Asynchronous drivers
  - Use hardware

Embedded Operating Systems

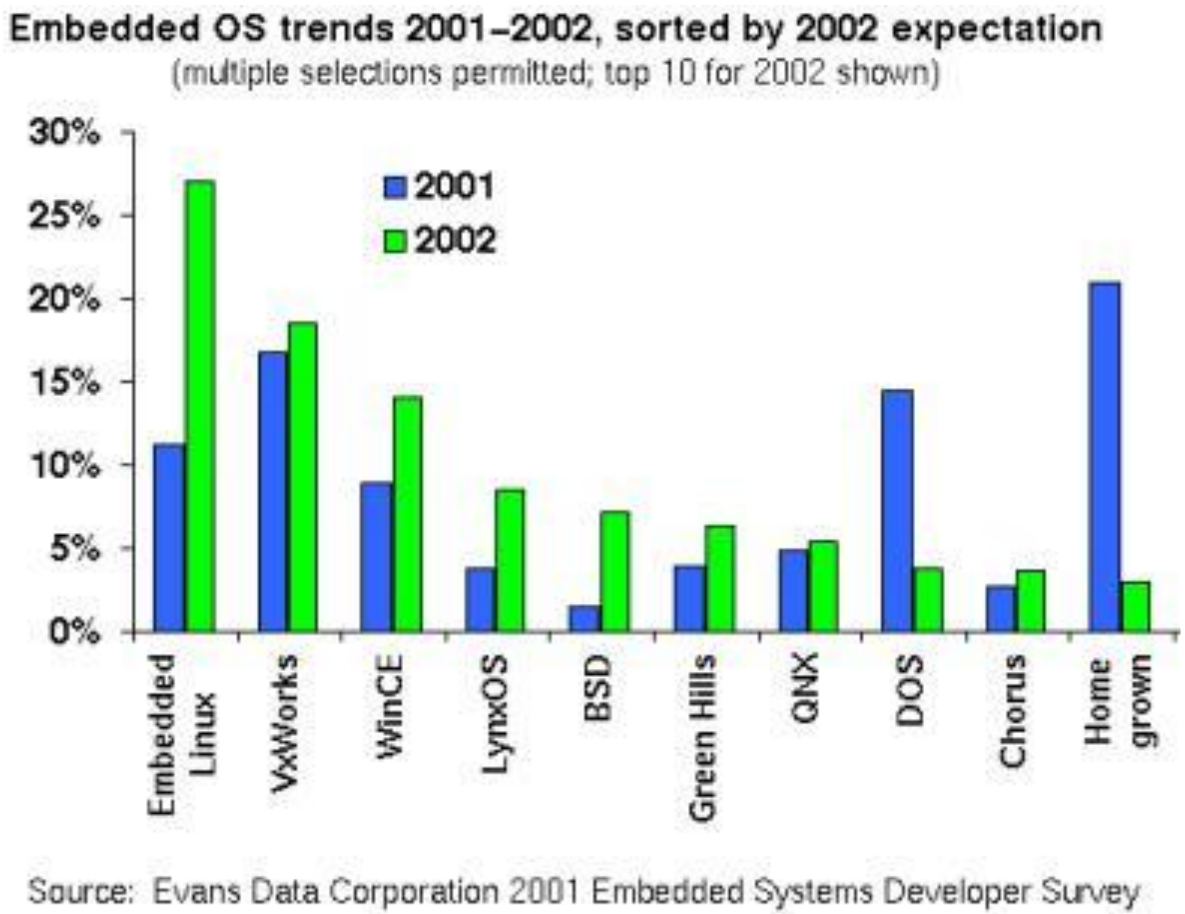
Embedded operating systems – why?

- 98 % of CPU:s sold in 2001 where used in embedded systems
- Companies are shifting away from home-grown operating systems

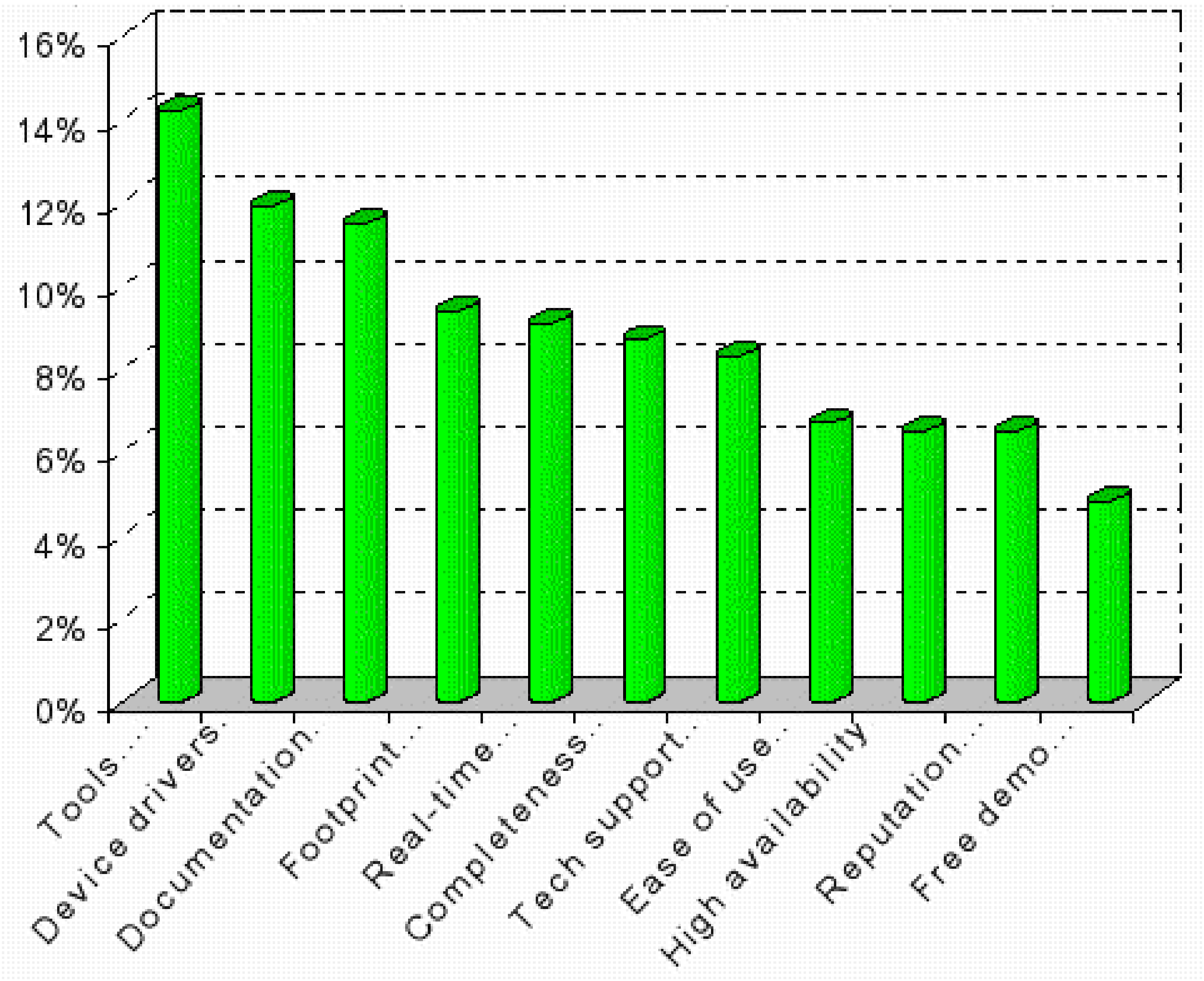
The Embedded OS Market 2001



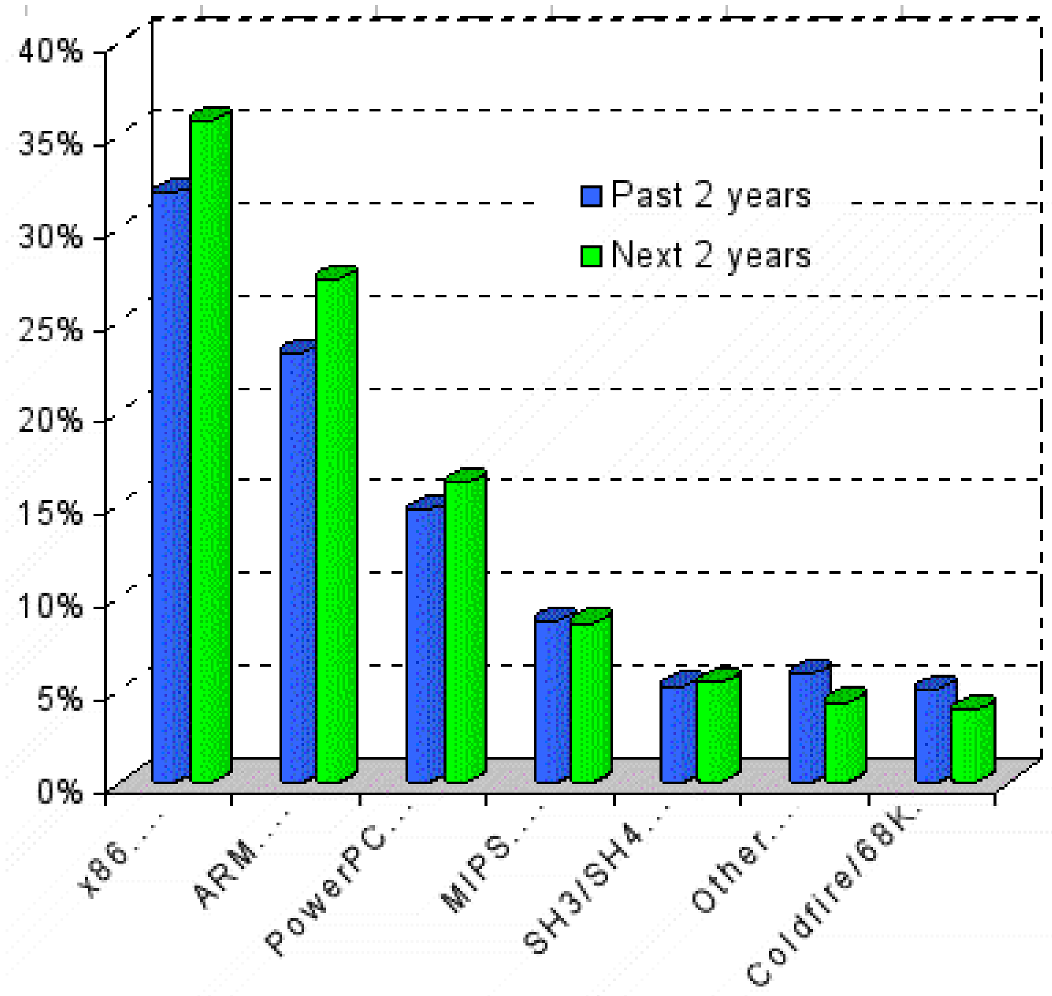
The Embedded OS Market 2002



Key factors for selection



Processors



40%
35%
30%
25%
20%
15%
10%
5%
0%
Past 2 years
Next 2 years
x86...
ARM...
PowerPC...
MIPS...
SH3/SH4...
Other...
Coldfire/68K...

Embedded operating systems

- Lite PC
  - Set-Top boxes, kiosks, thin clients
  - Windows NT/XP embedded, Linux
  - Similar to desktop OS
- Small devices
  - Cell phones, PDA:s, Broadband routers
  - PocketPC, PalmOS, Symbian, DOS, Linux
  - Small footprint, some real-time capabilities, no hard drive
- Hardend real-time
  - Missilies, satellites, Vehicles, Robots, Industrial Machinery
  - VxWorks, QNX, Windows CE, Integrity, Parh Lap, Linux
  - Tiny footprint, critical reliability, fully preemptive

What makes a good Embedded OS?

- Modular
- Scalable
- Configurable
- Small footprint
- CPU support
- Device drivers
- etc, etc, etc...

What makes a good RTOS?

- Multi-threaded and pre-emptible
- Thread priority has to exist because no deadline driven OS exists
- Must support predictable thread synchronization mechanisms
- A system of priority inheritance must exist

Embedded operating system

- Task management
  - Create, delete, suspend, resume
- Time management
  - System clock, delay
- InterTask communcation and synchromization
  - Multitasking
    - No-OS: Disable / Enable interrupts
    - OS: enter/exit critical section
  - Wait for event
  - Exchange data, queues, shared memory
- Memory management
  - Temporary buffers
  - Allocate, free (critical in ES)

Choosing an Embedded OS

- Memory requirements
  - Hard drives are rare
  - Usually some kind of flash
  - May not be flatly addressable
  - 512 KB-32 MB typical
  - Limited lifespan on write access
  - RAM is precious
  - Execute in Place (XIP)

Choosing an Embedded OS

- Real-time requirements
  - Interrupt latency
  - Interrupts from hardware or software
  - Consistency
  - Worst-case response
  - Driver layers reduce performance
  - DOS is the fastest...

Choosing an Embedded OS

- Fault tolerance
  - Memory protection
  - Avoid dynamic allocation
  - Avoid pointers
  - Watchdog timers
  - Microkernel

Embedded system development

- Very many ES programmers have degrees in some other field
- Not until recently ES software has become so large that more than one developer is required
- Traditionally, ES programming is in a software engineering view behind

Axis 2120 Network Camera

- uCLinux
- Built-in Ethernet port
- 100 MHz ETRAX CPU
- 16 MB RAM

Humanoid Robots

- HOAP

- Fujitsu
- RT-Linux
- Height: 48 cm
- Weight: 6 kg
- 100 units/yr

Real time and Linux

- Linux is not a hard Real-Time operating system
  - Hardware interrupts:
    - Worst case latencies cannot be given
  - Timers
    - Timer jitter too high: > 15 msec
- Soft Real-Time capabilities improved in Linux 2.6, however, same problems still remain

RT Linux performance

- Interrupt latency
  - Worst case 15 microseconds
- Period task
  - Jitter maximum 35 microseconds

Realtime requirements recap

- Text editor (no realtime requirements)
  - If it takes half a second to update display now and then, a few users will notice
    - Fast and responsive
- Video display (soft realtime)
  - Should almost always keep up with frame rate, half a second freeze is unpleasant
    - Must usually meet timing deadlines
- Airbag system (hard realtime)
  - Any random latencies in the system is totally unacceptable
    - Must guarantee response times

Realtime example

- A board sampling analog lines
  - 8 bit sample every 100 microseconds (=10 kHz)
  - Most boards nowadays have hardware buffers, e.g. for 512 samples
  - -> Must be read every 50 msec
    - Any response time over 50 msec will loose data, standard Linux WILL NOT guarantee this
- The problem with general OS:s
  - What you win in average performance, you loose in worst case performance
    - Good example: Paging system

Solution of small real time systems

- Often endless loops of simple tasks
  - longest time before a task will run is the sum of execution time of the tasks in the loop

```
counter=500
while (1) {
      if (data_on_sensor()) {
         read_sensor();
         compute_output();
         counter--;
      }
      if (!counter) {
            output();
            counter=500;
      }
}
```

- Problem: does not scale
  - Monitoring hundreds of sensors, displaying grahphical results...

Adding realtime support to non-realtime OS?

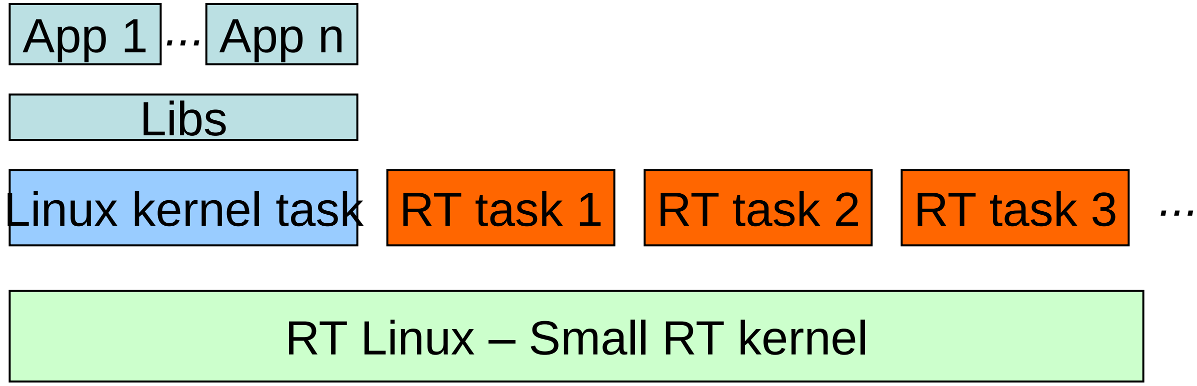
- Realtime support into kernel, locked memory pages (cannot be swapped out)
  - *mlock(), sched_setsched()*
- Still , worst time jitter is several milliseconds (18 in milliseconds in one report)
  - Compared to RTLinux, 25 microseconds – almost 1000 times better

Problems with Linux

- ”Coarse-grained” synchronization
- Scheduling: fairness gives even most unimportant nicest task CPU-time
- Reordering of tasks (e.g. disk requests)
- ”Batch” operations, e.g. freeing pages when swapping
- Missing preemption in system calls
- High priority tasks waits for low priority tasks to free resources

RT Linux solution

- The computer runs a hard real-time OS, Linux runs as a low priority task
- Standard UNIX programming environment available to realtime problems



RT Linux technique

- Software emulation of interrupt controller hardware
  - Linux cannot disable interrupts
  - Linux can never add latency to the realtime system interrupt response time
- RT kernel
  - never request memory
  - never waits for resources
  - no synchronization, spin-locks etc...

RTLinux

- Real time tasks are written as normal modules
  - Linux can handle device initialization, module loading, unloading etc.
- The RT task only handles the raw, time critical, interface to hardware, anything else is handled by the operating system

RTLinux example

```
/* Module to toggle output on the parallel port */
RT_TASK my_task;
#define STACK_SIZE 3000
void code_for_rtl_task(unsigned int pin) {
    static unsigned char bits = 0;
    while (1) {
        if (bits) bits = 0; else bits = (1<<pin);
        /* Write on the parallel port */
        outb(bits, LPT_PORT);
        rt_task_wait();
    }
}
int init_module(void) {
   RTIME now = rt_get_time();
   /* Init task with code, pin 3, STACK and priority 1 */
   rtl_task_init(&my_task, code_for_rtl_task, 3, STACK_SIZE, 1);
   rtl_task_make_periodic(&mytask, now, 450);
   return 0;
}
```