Linux Board Porting

how to add Linux support for a new board

Patrick Bellasi

Dipartimento di Elettronica ed Informazione
Politecnico di Milano
bellasi@elet.polimi.it

Agenda

- Introduction
- Hardware design
  - customizing a reference design
  - HW verification
- Board Support Package
  - board specific code
  - Boot-loader porting
  - booting Linux kernel
- User-space support
  - build your own distribution
- Build you own application

Introduction
Presentation objectives

- Show you how is *relatively* simple to develop you own Linux board
  - demo-boards are suitable for fast prototyping
  - real-applications require ad-hoc HW
    - *stability, both mechanical and functional*
    - *efficiency*
    - *Cost-reduction*
- Needs
  - reference design
  - a development methodology
    - *possibly using open-source tools*
  - mostly basic HW design skills

- Few simple steps
  1) customize an HW reference design to suite your needs
  2) build a cross-compilation toolchain
     using tools adopted on step 5
  3) setup boot code
     first (and Second) Level Bootloader
  4) add board support to Linux kernel
  5) build your own distribution

- We will consider as example the porting to a new board based on Atmel's AT91SAM9260 SoC

Agenda

- Introduction
- Hardware design
  - customizing a reference design
  - HW verification
- Board Support Package
  - board specific code
  - Boot-loader porting
  - booting Linux kernel
- User-space support
  - build your own distribution
- Build you own application

Hardware Design
Identify your needs

- Target system requirements
  - processing power
  - memories: type and amounts
  - devices
    - *communication channels (USART, USB, CAN, I2C, …)*
    - *digital I/O*
    - *ADC (channels, resolution, timings, …)*
    - *networking (ethernet, modem, zigbee, bluetooth, …)*
    - *sensors (GPS, Temp/Lux/Humidity...)*
  - power needs and budget
  - application fields constraints (usability, certifications, ...)
- Identify a SoC that match most of these requirements
  - consider your skills!
  - a reference design is usually provide by the producer

Hardware Design

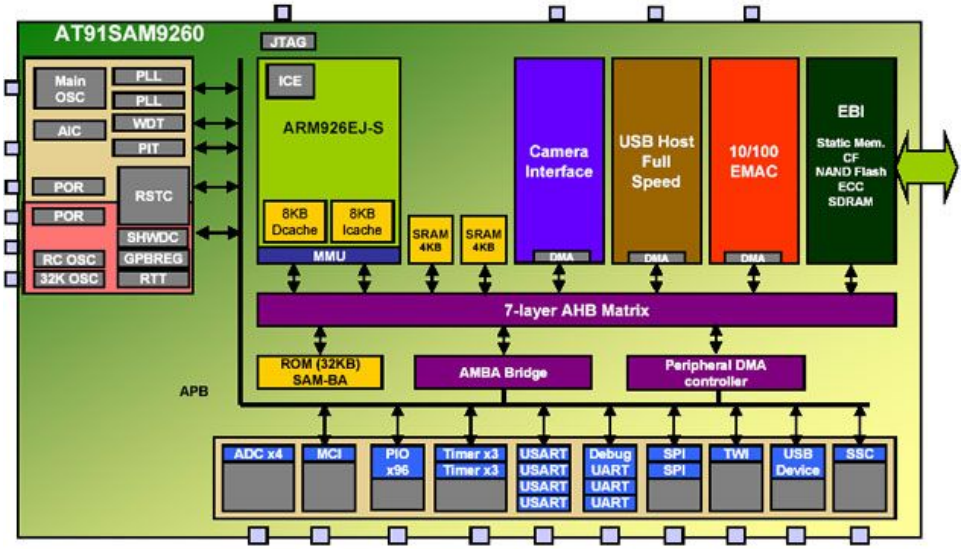
Example - AM9 Reference Platform: AT91SAM9260 by Atmel

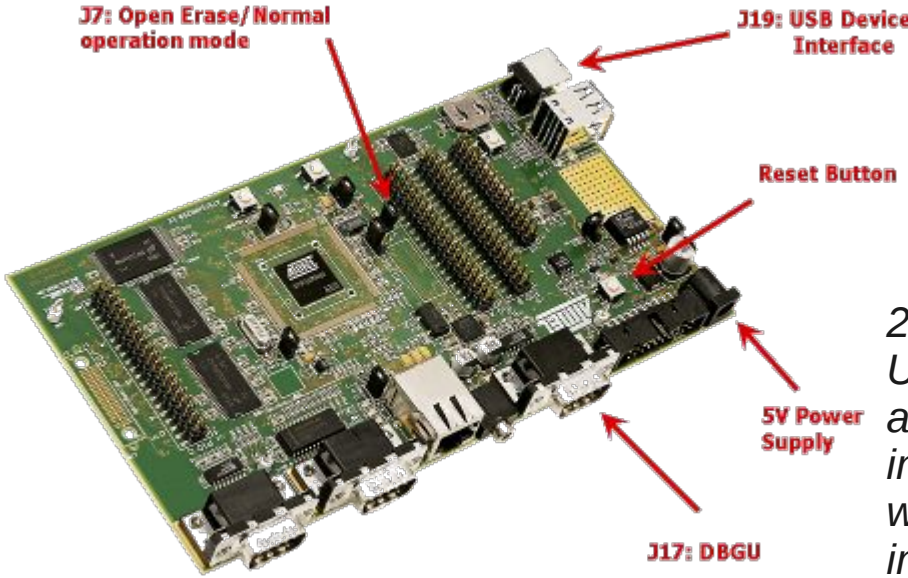
High performances 7-layer high-speed bus matrix

8 kByte instruction cache

Advanced system controller

Low power mode support

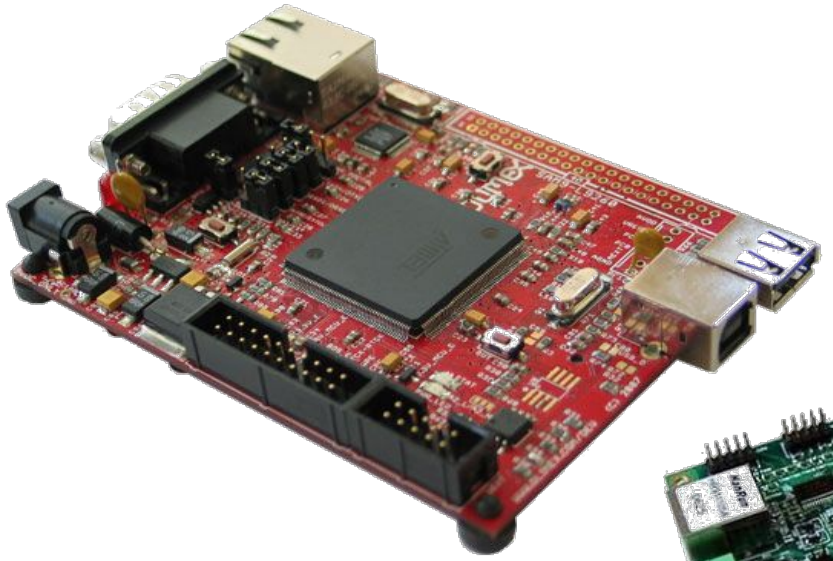




200 MIPS ARM926EJ-S® core, camera interface, seven USARTs, 10/100 Ethernet MAC, 12 Mbps USB device and host controller with on-chip transceivers, external bus interface (EBI) supporting SDRAM, Flash, NAND Flash with built-in ECC, SD, SDIO, and Multimedia Card interface (MMC), three synchronous serial controllers (SSC), two master/slave Serial Peripheral Interfaces (SPI),a three-channel 16-bit timer/counter, two-wire interface (I²C) and IEEE® 1149.1 JTAG Boundary Scan on all digital pins.

Hardware Design
Example – Custom boards based on same reference desing

- Atmel provides reference board's schematics
  - use to build you own board
    - *e.g. add new external devices (GSM, Memories, etc.)*



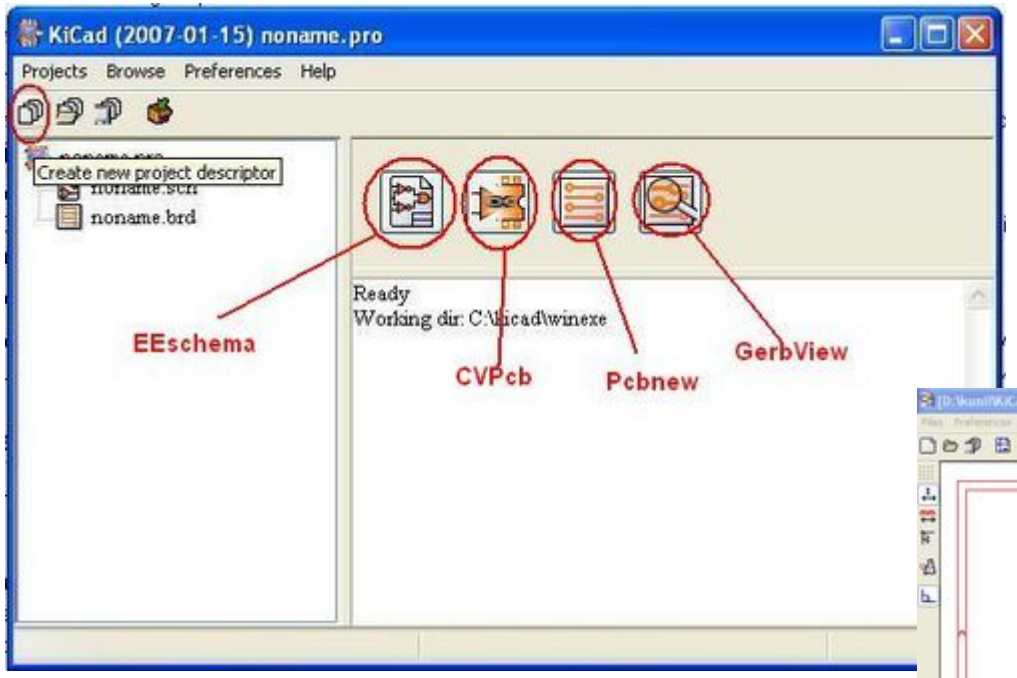
Omelix's SAM9-L9260

Daricom's C3-Ax03

Hardware Design
Design your own board

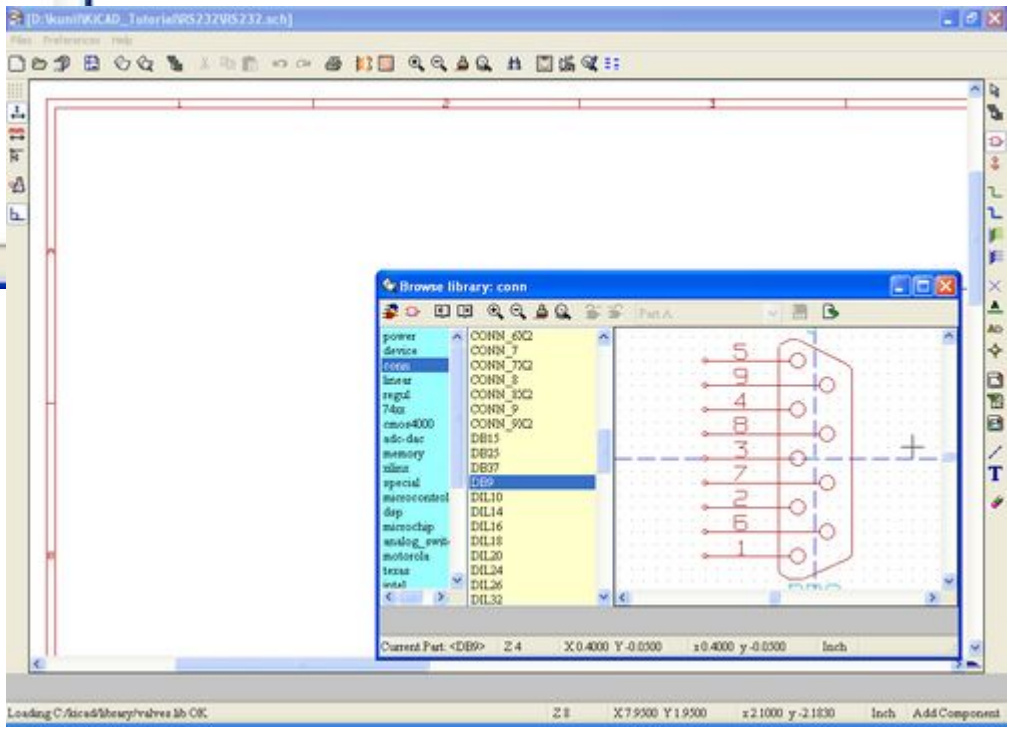
- Keep as much as possible from the reference design
  - substitute wrong components
  - add missing components
- Use an IDE to support both circuit design and verification
  - opensource solutions: KiCAD, Eagle
    - *export standard garber files*
  - use them correctly
    - *define pin properties*
    - *exploit circuit verification tools*
- Carefully review power lines design
  - you may need an help from an electronics engineer
  - if possible rely on power companion IC
- Add test-points to help HW debugging

Hardware Design
KiCAD – Main interface and Component Library



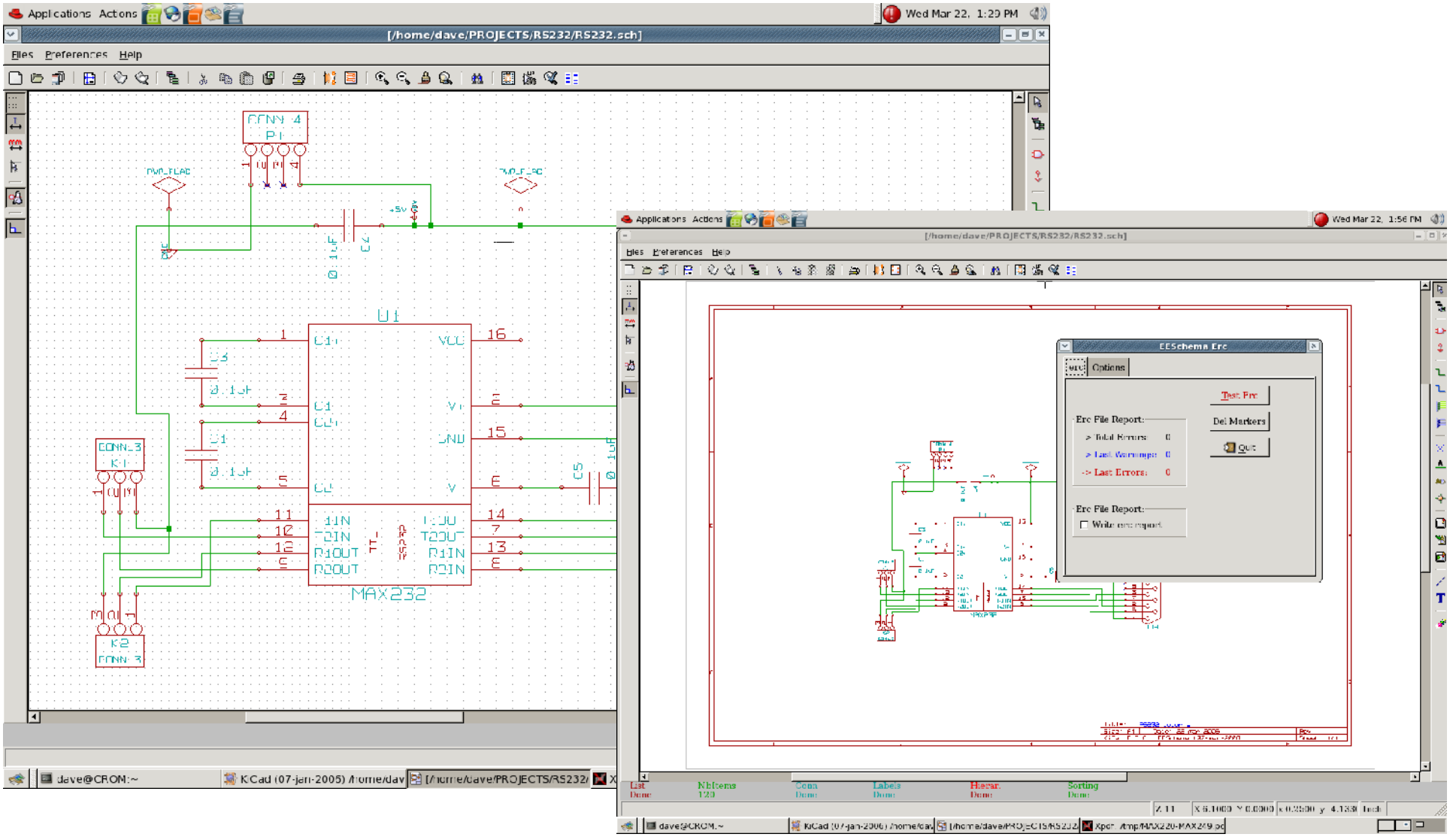
Main interface

Component Library



Hardware Design

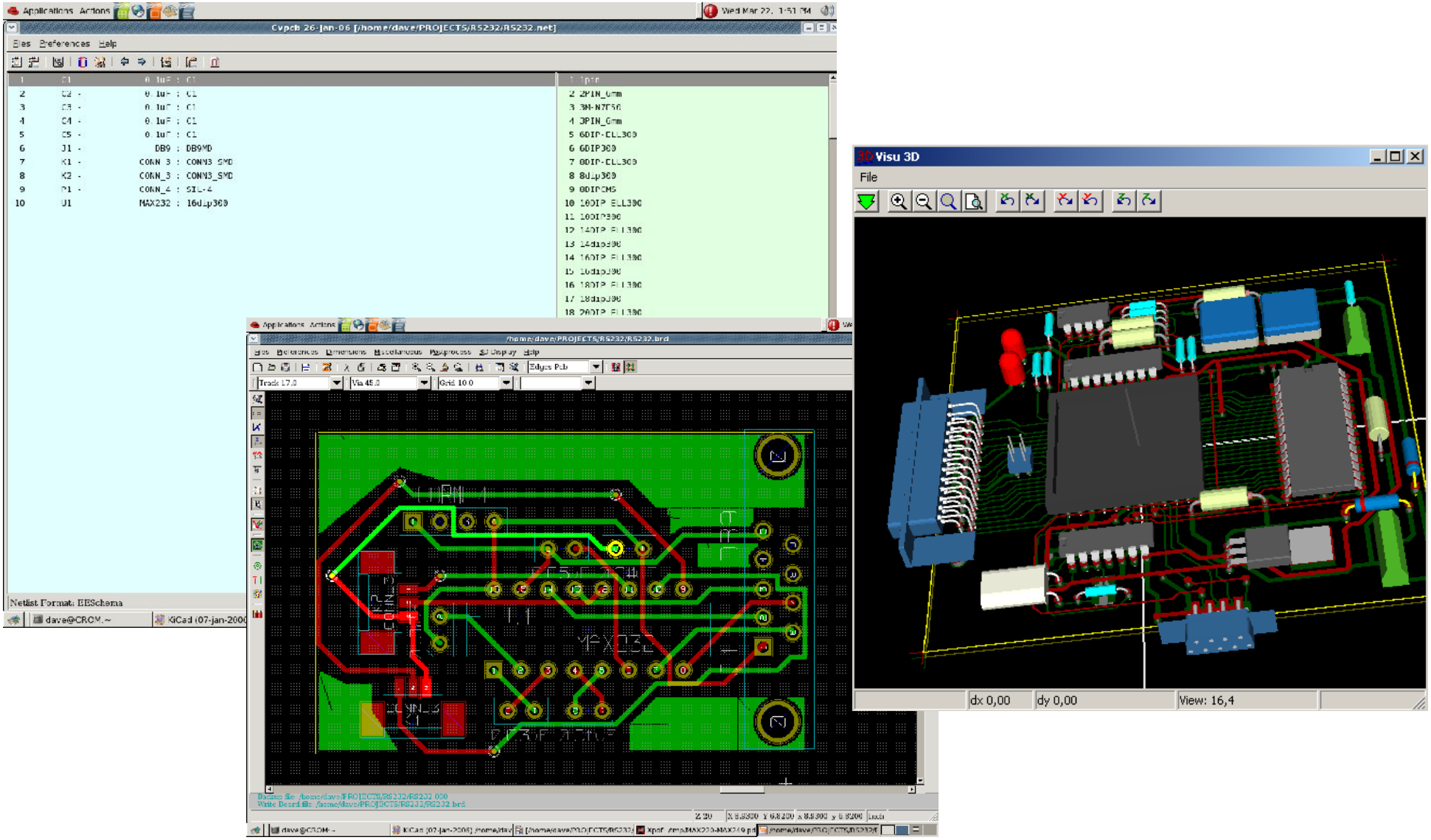
KiCAD – Schema Definition and Verification



POLITECNICO DI MILANO

Hardware Design

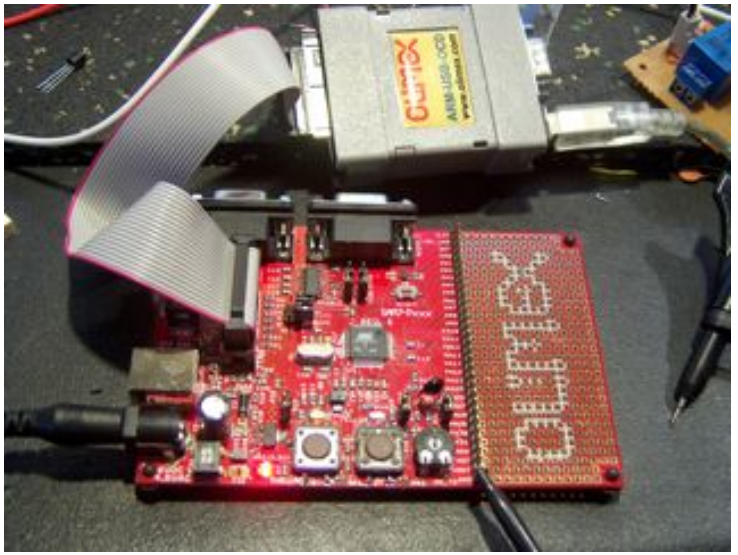
KiCAD – Part List and PCB Generation



POLITECNICO DI MILANO

Hardware Design
Produce and test the board

- Verify the production quality
  - power lines voltage levels
  - connections among components
  - test circuit functionalities
    - *you can test well behavior of SoC's external components*
      - *e.g. USART working, control GPIOs, …*
- JTAG (IEEE 1149.1) boundary-scan
  - device pin signals can be observed in real-time
    - *without interfering with the normal device operation*
  - used to control SoC's pins



Hands On

KiCAD Project and JTAG Boundary Scan

Agenda

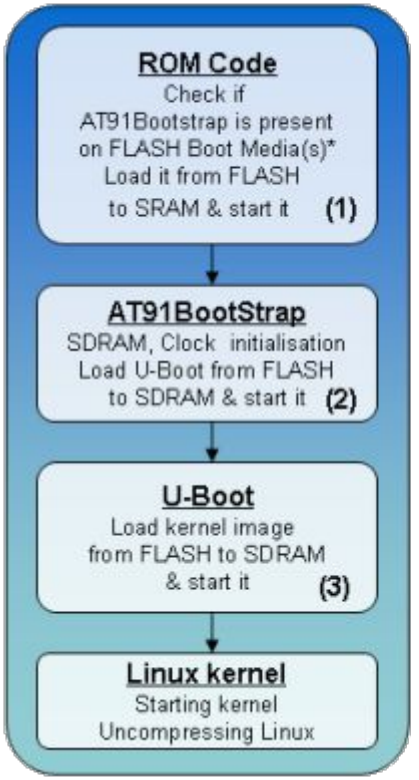
- Introduction
- Hardware design
  - customizing a reference design
  - HW verification
- Board Support Package
  - board specific code
  - Boot-loader porting
  - booting Linux kernel
- User-space support
  - build your own distribution
- Build you own application

Board Support Package
Introduction

- Contains everything needed to run an operating system on a given hardware platform
  - Microprocessor support
  - Board specific resources
    - *Bootloader*
    - *Board initialization code*
    - *Device drivers: buses and peripherals*
  - User-space distribution
- Integration into OpenSource standard tools
  - preferred delivery method (i.e. public available)
    - *u-boot and Linux board support, OpenEmbedded metadata*
- SoC's producer provide BSP for reference design
  - we can update the BSP to include support for our own board

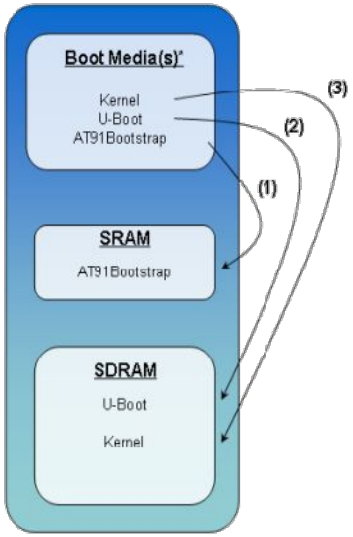
Booting Linux
SAM9: Boot Sequence

- Understand the boot procedure
  - several pieces of software are involved
    - *ROM code check if a valid application is present on supported media*
      - *FLASH, DATAFLASH, NANDFLASH, SDCARD*
    - *download a valid application into internal SRAM*
    - *run the First Level Bootloader (AT91Bootstrap)*
      - *in charge of HW configuration, download U-Boot binary from FLASH to SDRAM, start the Second Level Bootloader*
    - *run the Second Level Bootloader (U-Boot)*
      - *in charge of download kernel binaries from FLASH, network, USB key, etc.*
    - *start the kernel*
- Each architecture has a specific sequence
  - for the first two steps only



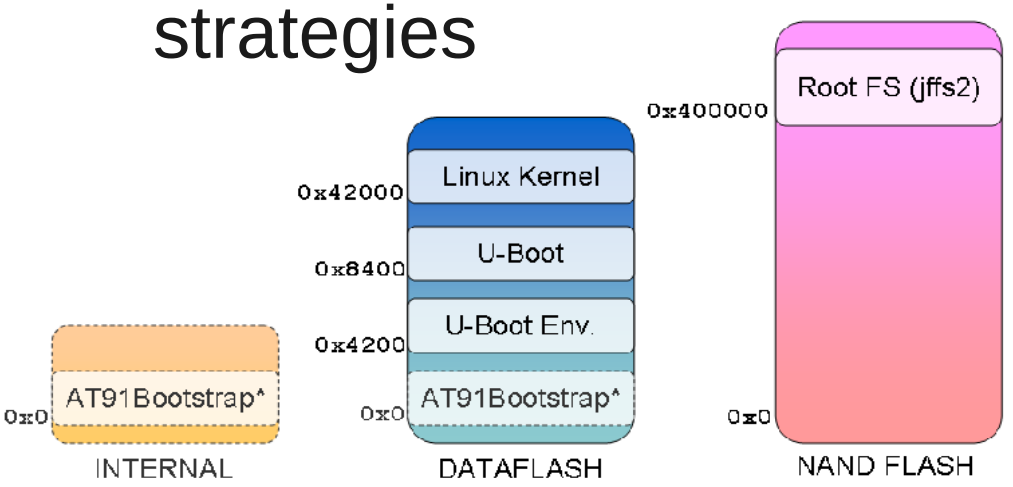
Booting Linux
SAM9: Understanding Boot Possibilities

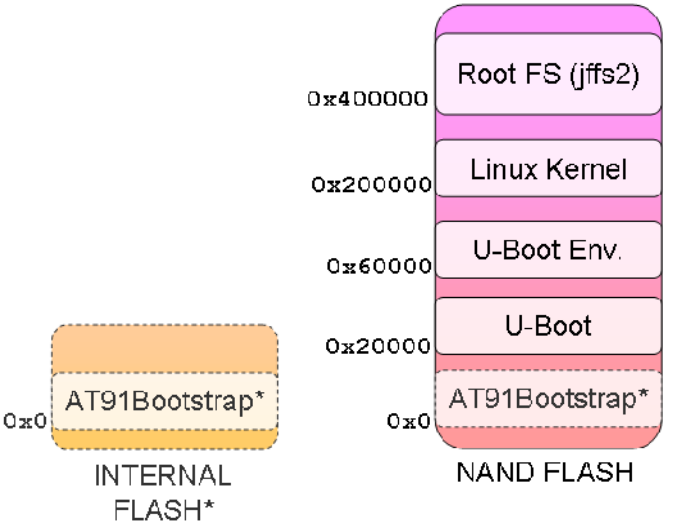
- Memory considerations
  - *4KB of internal SRAM*



- Memory mapping
  - give fixed addresses
    - *available memories*
    - *device configuration registers*

- Different booting strategies





Booting Linux
SAM9: ROM Code

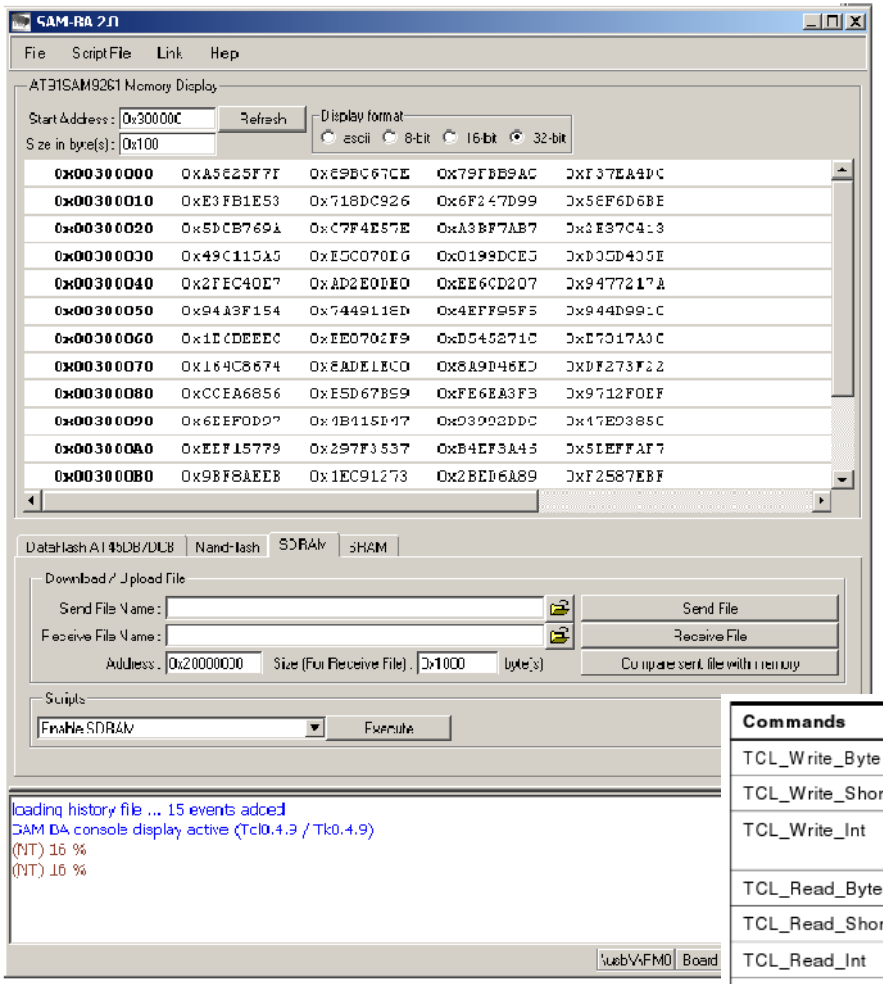
- Integrates different programs to download and/or upload into the different memories of the product
  - initializes the Debug Unit serial port (DBGU) and the USB device port
  - look for valid code in DataFlash CS0
    - *sequence of eight valid ARM exception vectors*
    - *all these vectors must be B-branch or LDR load register instructions*
  - if a valid sequence is found, code is downloaded into the internal SRAM
    - *remap and a jump to the first address of the SRAM*
    - *otherwise check DataFlash CS1 and NAND Flash*
  - If no valid ARM vector sequence is found
    - *SAM-BA® Boot is then executed*
    - *waits for transactions either on the USB device, or on the DBGU serial port*

Booting Linux

SAM9: SAM-BA® Boot Code

- Execute if a valid image to load is not found on flash
- Provide a simple command set
  - read/write internal SRAM memory
  - copy files to/from internal SRAM memory
  - jump execution to a specified address in SRAM
- Can be used with a graphical tool (SAM-BA GUI)
  - run on host, board attached via RS-232 or USB device
  - to upload code into SRAM and execute setup routines
    - *initialize external memories: SDRAM, Serial DataFlash and NAND Flash*
    - *available for Linux too*
  - can require some porting
    - *e.g. if you changed external memory configuration wrt reference design*
    - *C code*
      - *cross-compile using an arm-elf cross-compiler (e.g. wingcc)*

Booting Linux
SAM9: SAM-BA® Boot Code



SAM Boot Assistant

TCL API Commands

Commands	Argument(s)	Example
TCL_Write_Byte	Handle Value Address err_code	**TCL_Write_Byte** $target(handle) 0xCA 0x200001 *err_code*
TCL_Write_Short	Handle Value Address err_code	**TCL_Write_Short** $target(handle) 0xCAFE 0x200002 *err_code*
TCL_Write_Int	Handle Value Address err_code	**TCL_Write_Int** $target(handle) 0xCAFEDECA 0x200000 *err_code*
TCL_Read_Byte	Handle Address err_code	**TCL_Read_Byte** $target(handle) 0x200003 *err_code*
TCL_Read_Short	Handle Address err_code	**TCL_Read_Short** $target(handle) 0x200002 *err_code*
TCL_Read_Int	Handle Address err_code	**TCL_Read_Int** $target(handle) 0x200000 *err_code*
TCL_Compare	"fileName1" "fileName2"	**TCL_Compare** "C:/temp/file1.bin" "C:/temp/file2.bin"
TCL_Go	Handle Address err_code	**TCL_Go** $target(handle) 0x20008000 *err_code*
send_file	Memory Name Address	**send_file** {SDRAM} "C:/temp/file1.bin" 0x20000000
receive_file	Memory Name Address Size	**receive_file** {SDRAM} "C:/temp/file1.bin" 0x10000
compare_file	Memory Name Address	**compare_file** {SDRAM} "C:/temp/file1.bin" 0x20000000

Hands On

Using SAM-BA GUI

Booting Linux
SAM9: AT91Bootstrap

- The first level boot loader
  - small footprint (<4KB)
  - stored into dataflash or nand using SAM-BA utility
- Needed to
  - initialize external memory making them usable
    - *load “some code” into external memory*
    - *jump execution to some predefined address*
  - or reset boot sector to restore access to SAM-BA
- Can start either Linux or a second level bootloader
- May require few porting
    - *e.g. if you changed external memory configuration wrt reference design*
  - C code
    - *quite similar to SAM-BA code structure*

Hands On

Adding Board Support to AT91Bootstrap

Booting Linux

Das U-Boot – The Universal Bootloader

- Is a monitor program
  - command line interface with basic command set
    - *information, memory, flash memory, execution control, network, environment variables, filesystem support, special and miscellaneous commands*
- It is relatively easy to port U-boot to a new board
    - *this explains its success*
  - many supported architectures
    - *PPC, ARM, MIPS, x86, m68k, NIOS, Microblaze*
    - *thousand boards supported by public source tree*
- It is responsible of configuring main interfaces and launching a Linux system
  - it is possible to directly boot Linux from AT91Bootstrap
  - in a production phase for instance

Booting Linux
Das U-Boot – CLI and Environment Variables

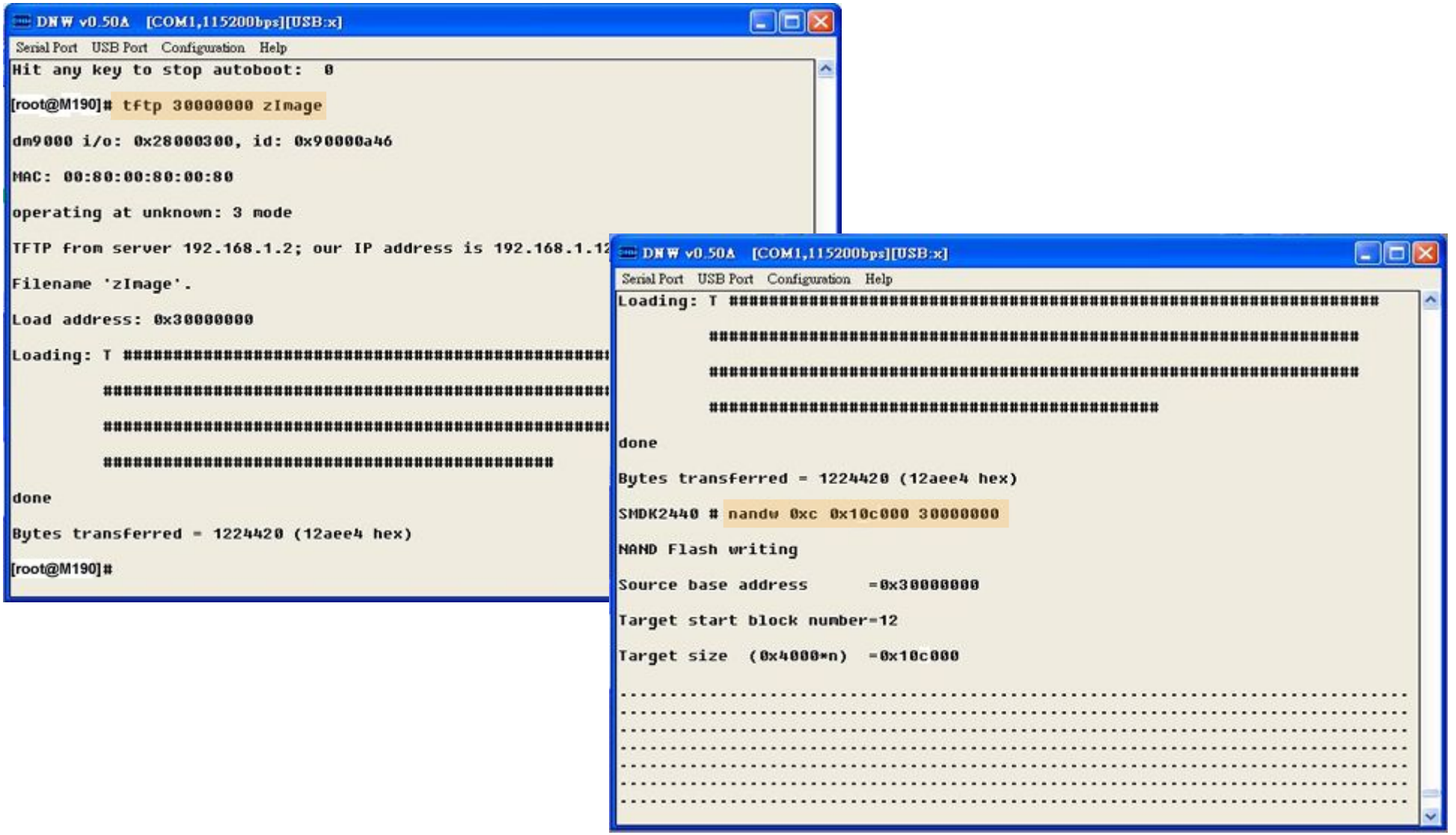
```
Welcome to minicom 2.1

OPTIONS: History Buffer, F-key Macros, Search History Buffer, I18n
Compiled on Sep  9 2005, 16:35:35.

Press CTRL-A Z for help on special keys
```

← Hit return once to get command prompt, U-Boot is running in RAM

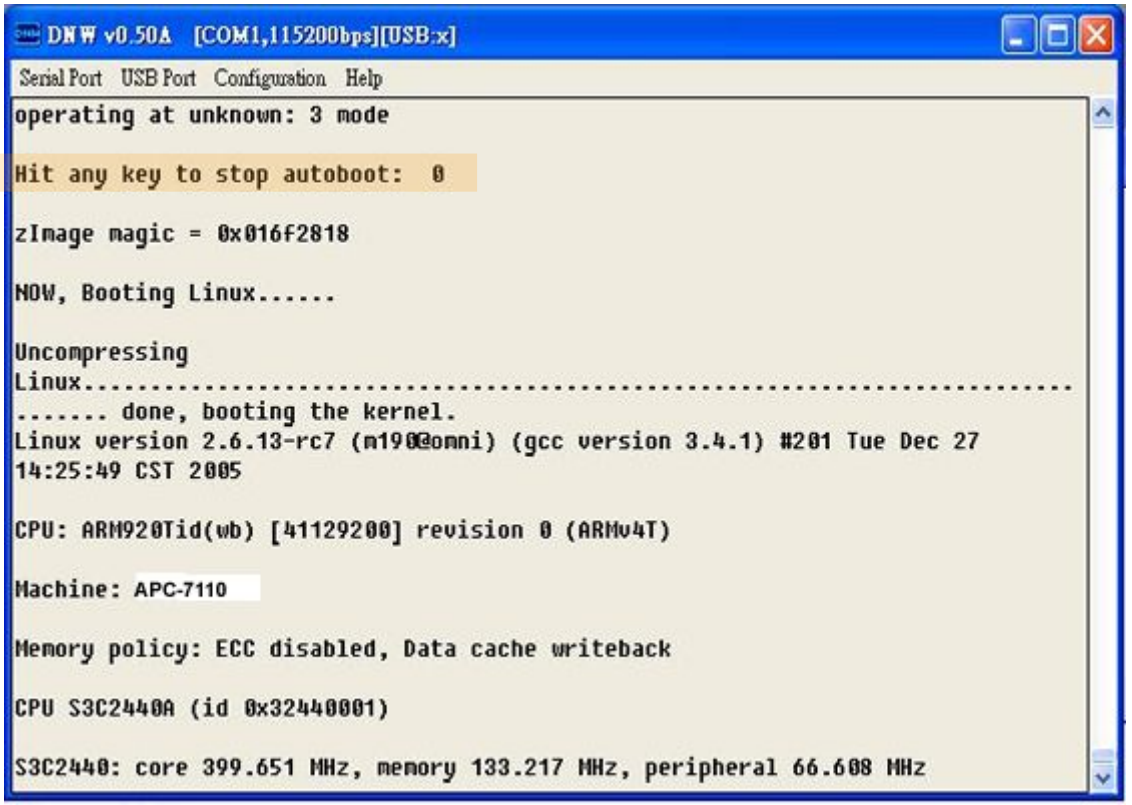
```
U-Boot > printenv
bootargs=mem=120M console=ttyS0,115200n8 root=/dev/hda1 rw noinitrd ip=dhcp
bootcmd=setenv setboot setenv bootargs \$(bootargs) video=dm64xxfb:output=\$(videostd);
baudrate=115200
bootfile="uImage"
stdin=serial
stdout=serial
stderr=serial
videostd=ntsc
ver=U-Boot 1.3.0 (Dec  1 2007 - 08:57:52)

Environment size: 323/65532 bytes
U-Boot >
```

```
CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.1     | VT102 |      Offline
```

Booting Linux
Das U-Boot – Storing Kernel Image into NAND



Booting Linux
Das U-Boot – Booting Linux



- Configuration depends on the combination of board and CPU type
  - such information is kept in a configuration file
    - *include/configs/<board_name>.h*
  - look for an already supported board which is as close as possible to yours
    - *e.g. have a look at the at91sam9260ek.h*
- Create support code
  - in board/atmel/<your_board>
  - have a look at similar atmel'2 boards:
    - *at91sam9260ek.c config.mk led.c Makefile nand.c partition.c*

- Adjust building scripts to add your board
  - MAKEALL:
    - *LIST_at91="at91cap9adk at91rm9200dk.... <your_board>"*
  - Makefile:
    - *c3ax03_config : unconfig*
    - *$(MKCONFIG) -a c3ax03 arm arm926ejs c3ax03 daricom at91sam9*
  - include/asm-arm/mach-types.h:
    - *#define MACH_TYPE_<YOUR_BOARD> <YOUR_MACH_TYPE>*
    - *(originating from Linux headers, should already be present)*
- Cross-compile and load the binary image into Flash
  - using SAM-BA GUI Application
  - save at address 0x8400
    - *defined into AT91Bootstrap code*

Booting Linux

Das U-Boot – Other Implementation Details

- CPU support implemented in the cpu/ directory

  74xx_7xx arm920t arm946es blackfin leon2 mcf5227x mcf532x microblaze mpc5xx mpc824x mpc85xx nios pxa sh3 arm1136 arm925t arm_intcm i386 leon3 mcf523x mcf5445x mips mpc5xxx mpc8260 mpc86xx nios2 s3c44b0 sh4 arm720t arm926ejs at32ap ixp lh7a40x mcf52x2 mcf547x_8x mpc512x mpc8220 mpc83xx mpc8xx ppc4xx sa1100

- Device drivers implemented in drivers/

  a lot of reuse from Linux drivers

- Headers implemented in include/ and in include/asm-<arch>/

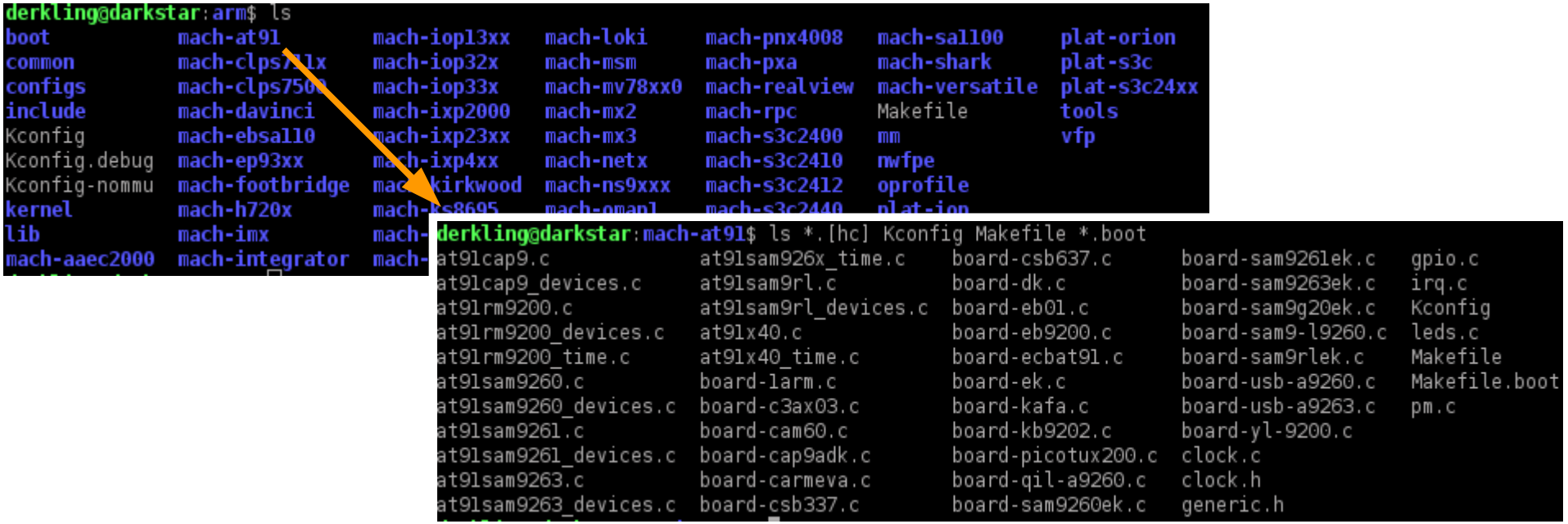
  mostly Linux headers

Hands On

Adding Board Support to U-Boot

Booting Linux
Linux Kernel – Architecture Specific Code

- Processor specific code is under arch/<arch> and include/asm-<arch>
  - e.g. arch/arm
- Machine specific code is under arch/<arch>/<subdir>
  - e.g. arch/arm/mach-at91



- On ARM done with the MACHINE_START macro

```
MACHINE_START(C3AX03, "Daricom C3-AX03")
        /* Maintainer: Patrick Bellasi <derkling@gmail.com> */
        .phys_io        = AT91_BASE_SYS,
        .io_pg_offst    = ((0xd8000000) >> 18) & 0xfffc,
        .boot_params    = AT91_SDRAM_BASE + 0x100,
        .map_io         = c3ax03_map_io,
        .init_irq       = c3ax03_init_irq,
        .init_machine   = c3ax03_board_init,
        .timer          = &at91sam926x_timer,
MACHINE_END
```

  declared in the board definition file

- Initialization routines
  - some are board specific
  - mostly composed of structures definitions and registration

Booting Linux

Linux Kernel – Board Definition Code

- Board definitions is done on a single file

  arch/arm/mach-at91/board-<your_board>.c

- Data structure definitions

  e.g. MACB Ethernet device

```
static struct at91_eth_data __initdata c3ax03_macb_data = {
        .phy_irq_pin    = AT91_PIN_PA7,
        .is_rmii        = 1,
};
```

- Data structure registrations

  e.g. MACB Ethernet device

```
at91_add_device_eth(&c3ax03_macb_data);
```

- Update configuration and compilation files

  enable building of your board code

  provide a default kernel configuration

  e.g. arch/arm/configs/c3ax03_defconfig

- Configure using default configuration

  cp arch/arm/configs/c3ax03_defconfig .config

  make ARCH=arm oldconfig

- Customize the kernel for your needs

  make ARCH=arm menuconfig

- Build the kernel image

  make ARCH=arm CROSS_COMPILE=<path_to_arm-gcc>

- Build the U-Boot kernel Image

  mkimage -A arm -O linux -C none -T kernel -a 20008000 -e 20008000 -n linux-2.6 -d arch/arm/boot/zImage uImage

Hands On

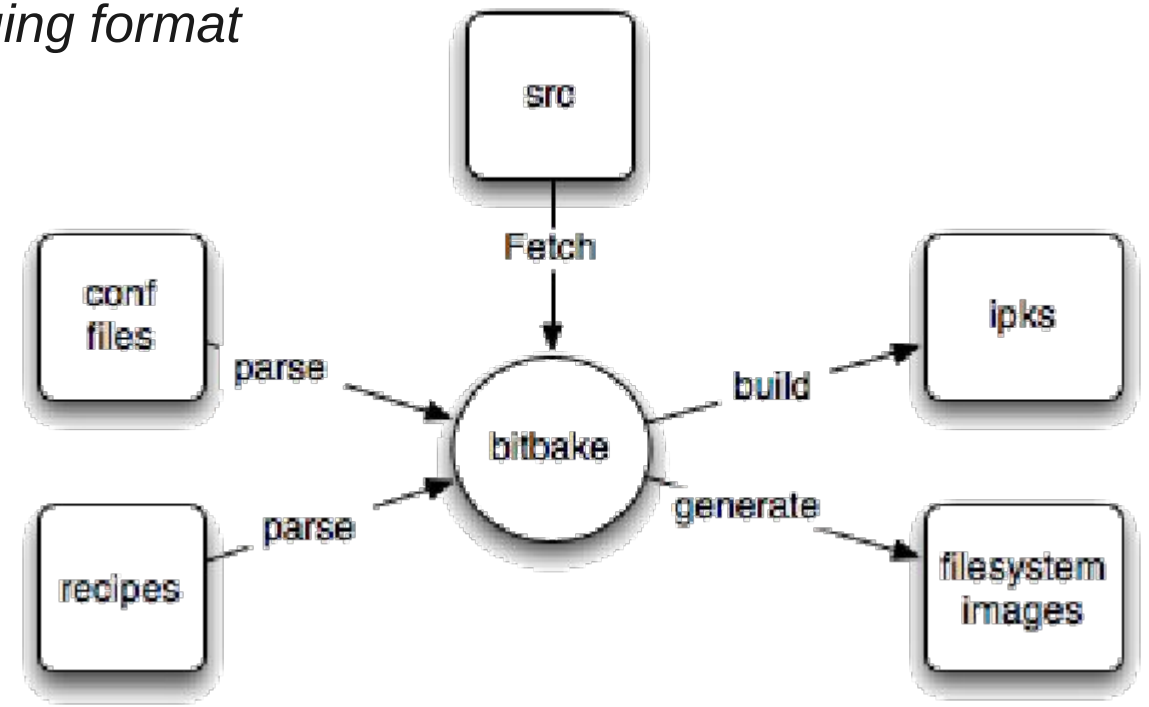
Adding Board Support to Linux

Agenda

- Introduction
- Hardware design
  - customizing a reference design
  - HW verification
- Board Support Package
  - board specific code
  - Boot-loader porting
  - booting Linux kernel
- User-space support
  - build your own distribution
- Build you own application

- Self contained cross build system
- Collection of metadata (recipes) that describe how to build a package
  - thousands of packages including bootloaders, libraries, and applications
  - for ~60 target machines including the N770 and x86
  - over 40 package/machine configurations (distributions)
- Does not include source code
  - fetches source using instructions in metadata
  - sopport for sources on tarball, SVN, GIT, ...
- Output is individual packages and filesystem images
  - jffs2, ext2, etc

- OE is powered by BitBake
  - parses the OE metadata to build the system
    - *parses recipes/confs*
    - *creates a database of how to fetch, configure, build, install and stage each package*
    - *determines package dependencies and builds in correct order*
    - *uses the IPK packaging format*



User Space Support
Openembedded Setup

- Decide on OE metadata version (snapshot or latest)
  - install bitbake
  - setup a pristine OE directory
  - keep changes in an overlay
    - *specific configuration files*
    - *internal package metadata*
    - *overloads on pristine metadata for classes, bb files, configuration*
  - download directory

```
derkling@darkstar:openembedded$ ls
bitbake  conf          files       packages     site
build    contrib       init.conf   README       sources
classes  COPYING.MIT   MAINTAINERS  removal.txt
```

User Space Support
Openembedded File Layout

- Six directories, three of them hold BitBake metadata
  - conf
    - *holding bitbake.conf, machine and distribution configuration*
    - *bitbake.conf is read when BitBake is started*
      - *will link a local.conf, the machine and distribution configuration files*
      - *all these files will be searched in the BBPATH environment variable*
  - classes
    - *holding BitBake bbclass that can be inherited by receipts BitBake*
  - packages
    - *store the BitBake files*
    - *a directory for each task or application*
      - *directories store the real BitBake files, ending with .bb*
      - *for each application and version we have one .bb file*
- Bitbake parses all configuration and recipes files found in the BBPATH environment variable
  - metadata files end with .conf, .inc, .bb and .bbclass

User Space Support

Openembedded Main Configuration Files

- local.conf
  - highest level configuration
    - *BBPATH variable*
    - *compilation options*
    - *images to produce*
    - *machine and distro*
- conf/machine/<machine>.conf
  - system architecture
  - kernel and its boot parameters
- conf/distro/<distro>.conf
  - version of each software packages

User Space Support
Using Openembedded

- Source the configuration file init.conf
- Build single packages

  bitbake busybox

  e.g. build the kernel

  bitbake linux
- Build meta packages

  e.g. build a standalone toolchain

  bitbake meta-toolchain
- Build a rootdisk image

  e.g. build a mininal system

  bitbake minimal-image

User Space Support
Openembedded Output

- Build output directories

cache	
conf	build specific configuration files
deploy	image and packages
staging	intermediate install for libraries and headers
work	build directory
cross	host tools for target
rootfs	expanded root filesystem
stamps	

Hands On

Configure and Use OpenEmbedded