Is Raspberry Pi Usable for Industrial and Robotic Applications?

Pavel Pisa
pisa@cmp.felk.cvut.cz

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

PiKRON s.r.o.
ppisa@pikron.com

2015-3-8
InstallFest 2015

Content of Presentation

Raspberry Pi Overview

- affordable/cheap single board computer developed for promotion of the teaching computer science
- the low cost demand lead to significant compromises
  - version 1 is based on Broadcom BCM2835 chip based on ARM1176JZF-S (ARMv6 architecture, insufficient for Debian armhf port which demands ARMv7-A and VFPv3-D16)
  - the cheap mobile applications SoC does not include ETHERNET controller which is added as USB converter
  - no memory technology devices (MTD) or integrated storage – uses standard or micro SD-card
  - CPU performance is not great and VideoCore IV GPU is proprietary/closed
  - extension connectors mindlessly (much better on B+ and version 2)
- the Debian compatibility solved by version 2 (BCM2836 quad-core ARM Cortex-A7). Performance enhanced.

Raspberry Pi Applications

- facts
  - intended for education
  - provides decent performance for video playback
  - is really cheap
- the last point is important
  - used for many hobby projects, strong community
  - used even in commercial solutions due to low cost even that it is not intended for such use

Alternatives

Many better alternatives exists for industrial applications.
There are listed few ones

- FreeScale i.MX53 – ARM Cortex A8, ETHERNET, CAN, USB, ...
- FreeScale i.MX6 – ARM Cortex A9
- TI AM335x Sitara ARM Cortex-A8 (Beagle Bone black) – adds quadrature encoder inputs, two real time coprocessor units (PRU)

Why We Are Involved in RPi World?

- Inquiry to PiKRON company to resolve SD-card failures in already running industrial application
  - RPi has been chosen (by price) as probe to collect data from workshops and send them into cloud infrastructure
  - the SD-card proves to be main point of failures and correction required personal assistance on distant plants
- RPi is hardware bought by many students and hobbyist
  - quite often more powerful solution is found for initial dream multimedia applications and boards are free for experiments
  - RPi can be bridge to broad world of electronic tinkering, ideas prototyping and can open eyes people that there is much more to play with than virtual worlds and clouds

Outline

Storage Technologies

- Rotating disks
- Flash based technologies
  - Small capacity NOR based devices – still used for BIOSes, used for firmware, usually only for boot in embedded systems, can be mapped to CPU memory address space (even serial SPIFI)
  - NAND based Flash – much larger capacity on same silicon, sequential access, much more susceptible to errors and wear, requires wear-leveling management and error correction control
    - can be achieved solely by software (slow)
    - by controller integrated into central SoC or controller
    - by controller integrated into device (SD-card, SSD)
  - SLC (Single-level cell) $\times$ MLC (Multi-level cell – cheaper)
  - Industrial SLC SD-card 4 GB $\geq$1000 kč, 16 GB $\geq$3500 kč, MLC Industrial 16 GB 900 kč
  - MLC standard/customer grade Micro SDHC 16 GB 200 kč, SDXC 64 GB 900 kč

Customer Grade SD-cards

- MLC (Multi-level cell) – two or three bits per cell
- minimum spare blocks to replaced broken ones
- single block writes limit can be lower than 1000
  - thanks to big capacity, wear-leveling and obsolesce of customer devices (cameras, phones, etc.) the device manufacturer counts with only few complete overwrites during device lifespan
  - but if the SD-card is used as a root filesystem then logs and other data are overwritten quite often – even that chunks are small an erase blocks are large (megabytes) and even clever wear-leveling cannot save device
- Manufacturers count with typical large capacity devices usage
  - FAT/exFAT filesystem, video, pictures or film stored and then read only a few times
  - time, temperature and reads contribute to cell (dis)charge, SD-cards could be designed for small (tens?) of reads before content rewrite

Outline

Possible Ways to Lower Write Count

- Use noatime, nodiratime or relatime mount options, select scheduled (ie. anticipatory), commit=300 for Ext4, /proc/sys/vm/dirty_bytes, /proc/sys/vm/dirty_background_byte, TRIM
- Select suitable filesystem
  - has been easy for small capacity, i.e. JFFS2, then attempts LogFS for MTD, UBI layer and UBIFS, but most of today media has internal controller – try Samsung F2FS, may it be that BtrFS COW can be better then Ext4 journal
- Use systemd "stateless" setup
- **Left distribution intact** and use **RAM/tmpfs overlay** setup and add partition for application persistent data

Root Overlay Options

- Use and modify initial RAM filesystem (initramfs) to setup overlay
  - Solution designed and maintained for diskless boot in laboratories for more than 10 years at Department of Control Engineering (persons involved Aleš Kapica, Pavel Píša, Michal Sojka, past Lukáš Moc)
  - The setup presented at InstallFest 2011 DiskLess Debian/GNU Linux at DCE FEL CVUT.cz
  - The more complete documentation by Aleš Kapica at `https://support.dce.felk.cvut.cz/mediawiki/index.php/Diskless`
- Modified/replaced system init command
  - I.e. append `init=/sbin/init-overlay` to the kernel command line and implement required logic in that script
  - This solution is more limited but much simpler for moderately experienced users

Raspberry Pi Boot Process

- VideoCore IV GPU loads the first stage bootloader from a SoC ROM
- GPU loads and executes `bootcode.bin` found on FAT32 or FAT16 partition on SD-card
- Third stage loader `start.elf` is load and executed by GPU and loads and parses `config.txt` file before the ARM core is initialized (see config.txt documentation at raspberrypi.org and eLinux.org for detailed description)
- This loader loads kernel specified by line of `config.txt` which requests `kernel.img` by defaults

  ```
  kernel=kernel.img
  ```

- The memory size and kernel command line found in the file cmdline.txt are placed to ATAGS parameters structure at memory address 0x100

Starting the Linux Kernel

- Linux kernel is finally started on ARM CPU with register r0 = 0, r1 specifying machine and r2 provides physical address of passed ATAGS structure or device tree block (dtb) in system RAM, more at kernel Documentation/arm/Booting
- The first SD-card partition has to be FAT and is unsuitable for GNU/Linux system root. The another partition or device is formated for suitable filesystem (ext4, btrfs, etc.) and used as root
- For init-overlay, check that root is initially mounted read only (`ro` kernel parameter) and append `init=/sbin/init-overlay` to `cmdline.txt`

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200
kgdboc=ttyAMA0,115200 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline
rootwait ro init=/sbin/init-overlay
```

Available Overlay Filesystems

- UnionFS (`http://unionfs.filesystems.org/`)
- Aufs3 by Junjiro R. Okajima (`http://aufs.sourceforge.net/`)
- OverlayFS, now Overlay by Miklos Szeredi (kernel Documentation/filesystems/overlayfs.txt)
  - included in mainline form 3.18.0 version
  - clean implementation but lacks some features
  - unusable with NFS even as lower/base filesystem for now

Script /sbin/init-overlay

- Finds available overlay support (aufs overlay overlayfs unionfs)
- Code tests /proc/filesystems and tries to load appropriate module by modprobe and insmod `/lib/modules/$(uname -r)/extra/`*x*`.ko` and `/lib/modules/$(uname -r)/kernel/fs/`*x*`/`*x*`.ko`
- `OVERLAY=`*x* can be specified on kernel command line for explicit selection
- The script requires mount-point `/overlay` on the root filesystem and executes

```
/bin/mount -n -t tmpfs none /overlay
/bin/mkdir -p /overlay/rwdata   #ovr_rwdata
/bin/mkdir -p /overlay/robase   #ovr_robase
/bin/mkdir -p /overlay/combined #ovr_combined
/bin/mount --bind / /overlay/robase
```

Setup Combined Filesystem

- Setup in memory temporary filesystem for changed data

```
/bin/mount -n -t tmpfs none ${ovr_rwdata}
```

- Setup combined mount for Aufs

```
/bin/mount -n -t aufs \
  -o dirs=${ovr_rwdata}=rw:${ovr_robase}=ro aufs \
  ${ovr_combined}
```

- or for Overlay

```
mkdir -p ${ovr_rwdata}/data
mkdir -p ${ovr_rwdata}/work
/bin/mount -n -t overlay -o up-
perdir=${ovr_rwdata}/data,workdir=${ovr_rwdata}/work,\
lowerdir=${ovr_robase} overlay ${ovr_combined}
```

Move Moun-points and Pivot Root

- Move mount-points from /overlay tmpfs to {ovr_combined}/overlay to be accessible from running system

```
/bin/mkdir -p ${ovr_combined}/overlay/rwdata
/bin/mount -n --move ${ovr_rwdata} \
  ${ovr_combined}/overlay/rwdata
/bin/mkdir -p ${ovr_combined}/overlay/robase
/bin/mount -n --move ${ovr_robase} \
  ${ovr_combined}/overlay/robase
/bin/mkdir -p ${ovr_combined}/overlay/pivot
chmod 755 ${ovr_combined} # Disable generic rw access
```

- proceed by final root switch

```
cd ${ovr_combined}
/sbin/pivot_root . overlay/pivot
```

- and then start regular init

```
exec /usr/sbin/chroot . sbin/init
```

Utility /sbin/overlayctl

Simple utility to automate `overlay-init` setup and control

- status – show actual setting status

  ```
  overlay is active
  overlay enabled for next boot
  ```

- disable – disable overlay support for following boots
  creates `$overlay_base/overlay/disable` file in the base root filesystem, it remounts root temporarily read write
- enable – reenables overlay by removing disable file
- unlock – remounts base filesystem read-write
  this allows to do changes in base filesystem, it is even possible to proceed `chroot /overlay/robase` and run some distribution maintenance there (dangerous, does not work with overlay)
- lock – returns to read only base

Installation of init-overlay

The utility `overlayctl` provides commands for easy overlay setup on standard Raspberry Pi

- install – RPi specific setup - appends `init=/sbin/init-overlay` to the line in `/boot/cmdline.txt`
- uninstall – removes init specification from `/boot/cmdline.txt`

Overlay Remarks

- The init-overlay creates fastboot file to bypass Raspbian/Debian distribution root device and filesystem consistency check
- Check content of `/etc/fstab` `/boot` entry to specify `defaults,ro`. The default provided fstab mounts FAT `/boot` partition read-write (each reboot modifies FAT mount state which is hazard). Remount `/boot` read-write before changes or distribution updates

```
mount -o remount,rw /dev/mmcblk0p1 /boot
```

Overlay Conclusion

Successfully used by company which ordered support/solution from PiKRON on Raspberry Pi systems
Solution is even used for regular x86_64 Debian installs at other FEE department computer laboratories where only single NFS root is distributed to all stations and local changes are hold in local RAM/tmpfs.
The complete work is available on GitHub `https://github.com/ppisa/rpi-utils` .
The init-overlay directory layout mimic locations in the target system including basic documentation /usr/share/doc/init-overlay/init-overlay.txt. The core files are /sbin/init-overlay and /sbin/overlayctl .

Outline

Why U-Boot on Raspberry Pi

- Raspberry Pi provides its own configuration loader and parser but it is quite limited
- U-Boot enables to select at startup and script different boot scenarios
- Stephen Warren's version provides extlinux menu and script support and not only for local boot but even for PXE, DHCP and static IP cases
- Even kernel can be loaded from network which enables fast testing during development and no data are lost when NFS root is used as well when system crashes
- U-Boot provides full support for Flattened Device Tree passing to the kernel (the current official the first+second stage boot loader adds this support as well)

Building U-Boot for Raspberry Pi

- Download and build U-Boot

```
git clone git://github.com/swarren/u-boot.git
cd u-boot
git checkout -b rpi_dev origin/rpi_dev
make rpi_defconfig ARCH=arm CROSS_COMPILE=arm-
rpi-linux-gnueabihf-
make ARCH=arm CROSS_COMPILE=arm-rpi-linux-
gnueabihf-
```

- Copy `u-boot.bin` to the target system `/boot` directory (the first/FAT partition)
- Modify /boot/config.txt kernel line

```
kernel=u-boot.bin
```

U-Boot Basic Commands

- printenv - print actual environment variables
- savenev - set environment variables
- setenv - save environment to be kept over reboot
- load - load some file to memory location
- boot - restart boot process
- sysboot

```
sysboot ${devtype} ${devnum}:${bootpart}
  any ${scriptaddr}
  ${prefix}extlinux/extlinux.conf
```

- The full documentation at
  `http://www.denx.de/wiki/DULG/Manual`

U-Boot Prompt Example

```
load mmc 0:1 0x00200000 cmdline.txt
md.b 0x00200000 156
setenv bootargs "dwc_otg.lpm_enable=0
  console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
  root=/dev/mmcblk0p2 rootfstype=ext4
  elevator=deadline rootwait ro
  init=/sbin/init-overlay"
load mmc 0:1 0x00200000 vmlinuz-3.18.8-rt2+
bootz 0x00200000
```

Extlinux Setup

`Files in new directory /boot/extlinux:`

- bcm2708-rpi-b-plus.dtb – Device Tree
  can be obtained from kernel build
  arch/arm/boot/dts/bcm2708-rpi-b-plus.dtb
- bcm2835-rpi-b-plus.dtb
  copy of above because U-Boot uses this name and it name preferred by Linux mainline
- extlinux.conf – menu for kernel/boot selection
- vmlinuz-3.18.8-rt2+ – standard Linux zImage kernel format

The configuration is searched on mmc0, usb0, pxe, dhcp devices and network

U-Boot Extlinux Setup

/boot/extlinux/extlinux.conf :

```
TIMEOUT 100
DEFAULT default
MENU TITLE Boot menu
LABEL default
    MENU LABEL Linux 3.18.8-rt2+ with Overlay
    LINUX vmlinuz-3.18.8-rt2+
    FDTDIR .
    APPEND ... console=ttyAMA0,115200
      smsc95xx.macaddr=${usbethaddr}
      root=/dev/mmcblk0p2
      rootfstype=ext4 elevator=deadline
      rootwait ro init=/sbin/init-overlay
```

Outline

RT-Preempt Patch

Realtime is not as fast as possible - realtime is as fast as specified – Doug Niehaus, Summer 2001

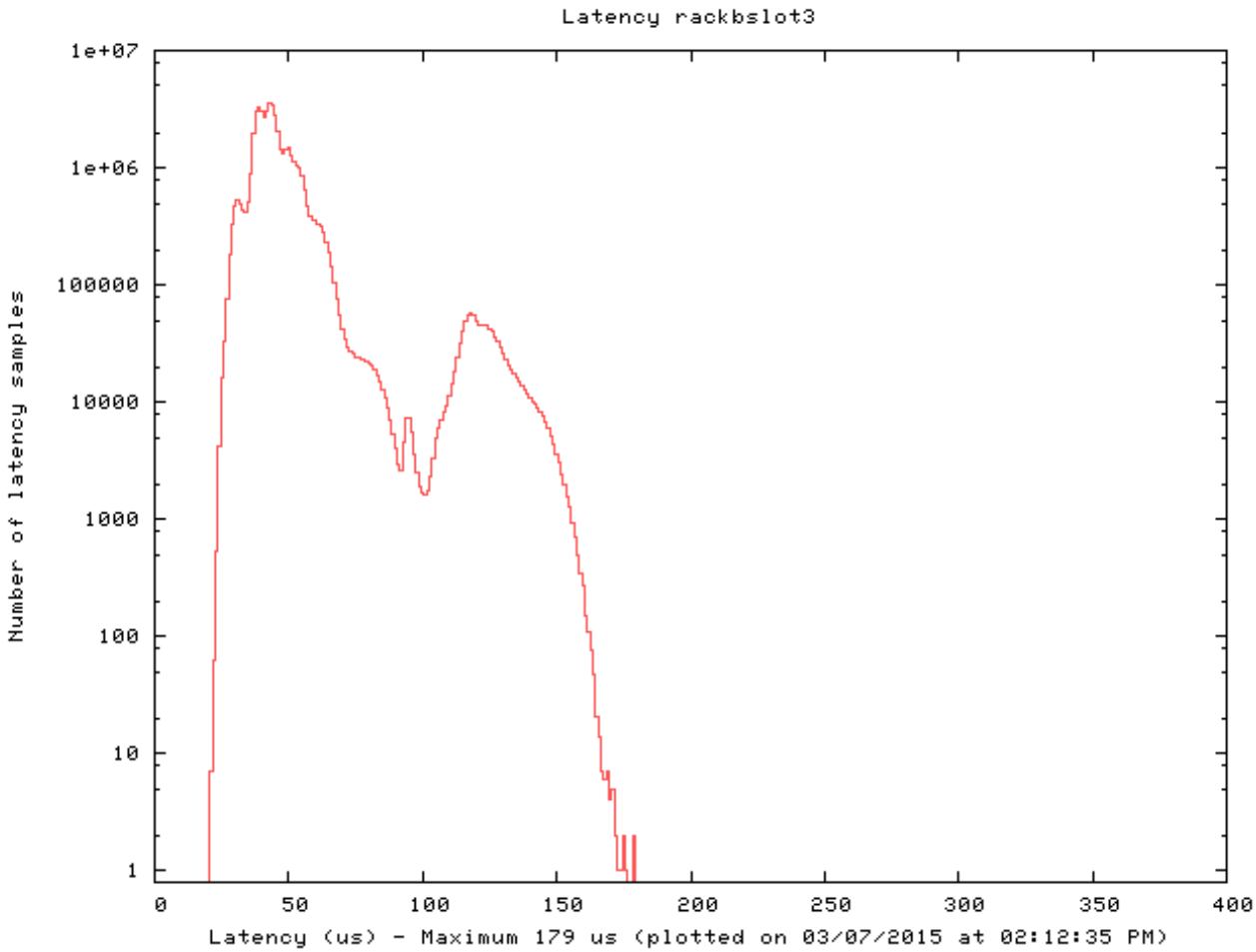
- More attempts to run RT task parallel to Linux base on same CPU (RT-Linux, RTAI) existed. But around 2001 and 2006 KURT/KUPS project tries to make whole kernel real-time. Work followed by Timesys, Thomas Gleixner, Ingo Molnar and OSADL.org now.
- The main idea behind changing Linux kernel to RTOS is to use already present support for multiple cores SMP and provide to system as many virtual CPUs as there are running threads/task.
- Realized by replacement of spin-lock synchronization by RT mutexes. redefinition of spin_lock/spin_unlock, spin_lock_irqsave/spin_unlock_irqrestore to use struct rt_mutex instead of atomic variables based lock

Actual RT State (for RPi)

- Open Source Automation Development Lab – long term testing and Quality Assurance Realtime Farm
- Latest available RT-Preempt for 3.18.7 kernel
  `https://www.kernel.org/pub/linux/kernel/projects/rt/`
- Maximal under about 100 µsec on powerful SMP x86 systems
- But what to expect at Raspberry Pi (BCM2708/BCM2835) Check at OSADL.org QA Farm Realtime rack-b-slot-3

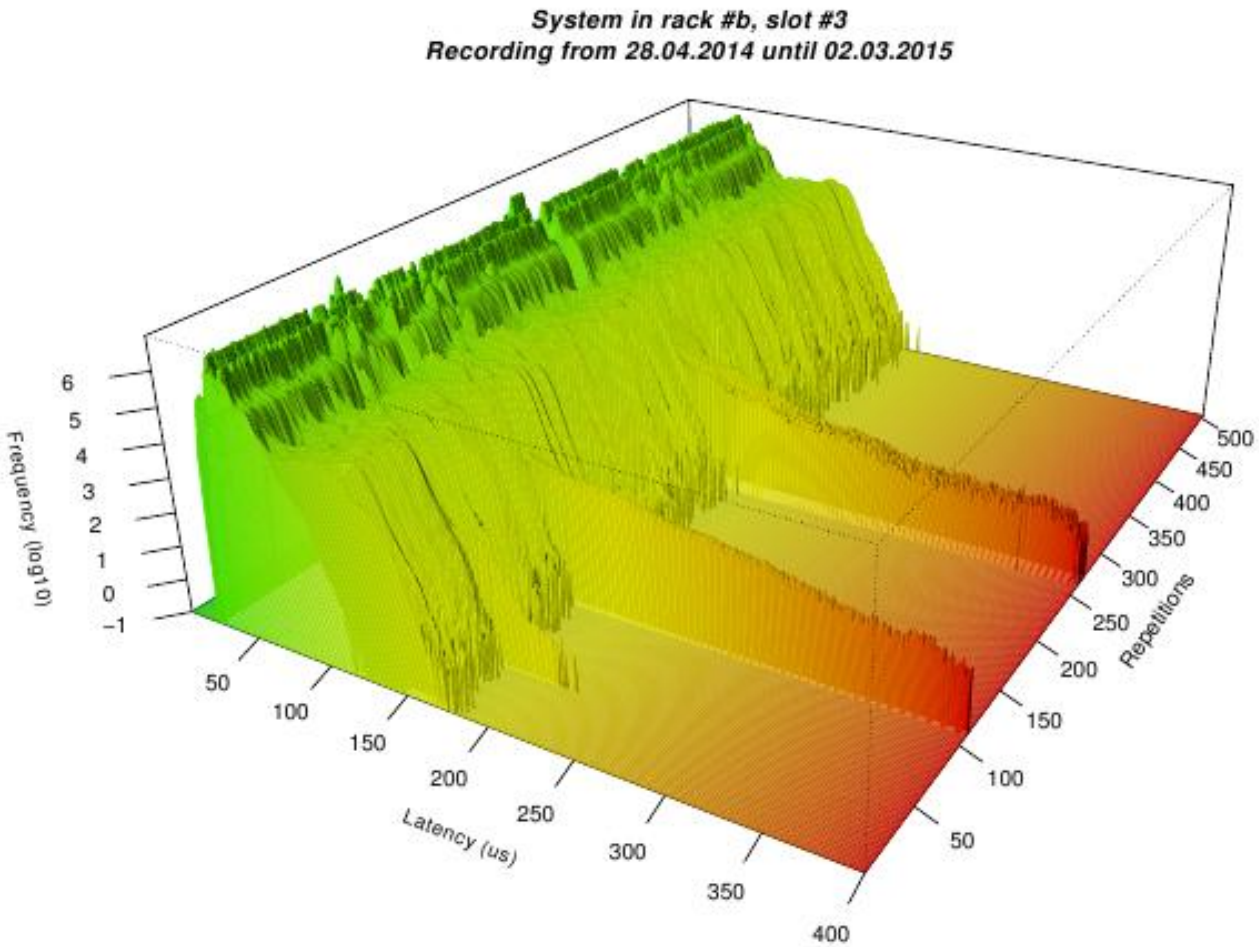
RPi 3.18.7-rt2 Latency Plot

OSADL.org – OSADL.org QA Farm Realtime – BCM2835 rack-b-slot-3
cyclictest -l50000000 -m -n -a0 -t1 -p99 -i400 -h400 -q



RPi Latency Long Term 3D

OSADL.org – OSADL.org QA Farm Realtime – BCM2835 rack-b-slot-3
Long term latency trancing for organization members available



Is It Enough?

- The standard control application is motor servo control
- Small DC and permanent magnet synchronous motors mechanical time constant is between 3 and 10 msec (electrical one can be under 1 msec, but mechanical is prevalent for control).
- System, when controlled for position, is not stable (pure integrator)⇒controller sampling period should be shorter than time constant to achieve proper control
- Maximal latencies under 200 µsec⇒RPi should be suitable for such control

RT Kernel Patches

Official RPi kernel `git://github.com/raspberrypi/linux.git` branch rpi-3.18.y

Patches applied

```
aufs3.18.1+ kbuild patch
aufs3.18.1+ base patch
aufs3.18.1+ mmap patch
aufs3.18.1+ standalone patch
Apply Aufs 3.18-20150305 sources by J. R. Okajima
aufs3.18.1+ module build enabled in RPi default config.
Allow ARMv6/ARM1176 to be selected for ARM Versatile PB.
Apply patch-3.18.7-rt2.patch fully preemptive ker-
nel patch by Sebastian Andrzej Siewior
Provide individual CPU usage measure-
ment based on idle time by Carsten Emde (OSADL)
Save the current patchset in the kernel
Reading /proc/slabinfo may cause large latencies
Add trace latency histogram to monitor context switch time
Workaround to access sysfs cpufreq variables
ARM bcmrpi: add bcmrpi_rt_defconfig .
ovl: do not reject NFS when used as lowerdir (change is inse-
cure, could lead to crash when NFS content is changed).
```

RT Kernel Build

- Patched kernel available in repository `https://github.com/ppisa/linux-rpi` , branch rpi-3.18.y-aufs-rt-ppisa
- The ARMv6 toolchain is required to build kernel. Unfortunately, arm-linux-gnueabihf- official Debian cross compiler targets ARMv7 VFPv3-D16. It can be used to build kernel (controlled by options) but user applications are miss-compiled and link against bad GLIBC
- Custom arm-rpi-linux-gnueabihf- toolchain used for crosscompile
- Kernel build

```
make ARCH=arm CROSS_COMPILE=arm-rpi-linux-gnueabihf- \
     bcmrpi_rt_defconfig
make ARCH=arm CROSS_COMPILE=arm-rpi-linux-gnueabihf-
make ARCH=arm CROSS_COMPILE=arm-rpi-linux-gnueabihf- \
     INSTALL_MOD_PATH=$(pwd)/_modules modules_install
```

Toolchain RPi ARMv6

- Can be downloaded from DCE RTime server. There is described GCC 4.9.x sources configuration as well.
  `https://rtime.felk.cvut.cz/hw/index.php/Raspberry_Pi`
- Critical configure options to achieve compatibility with RPi ARMv6

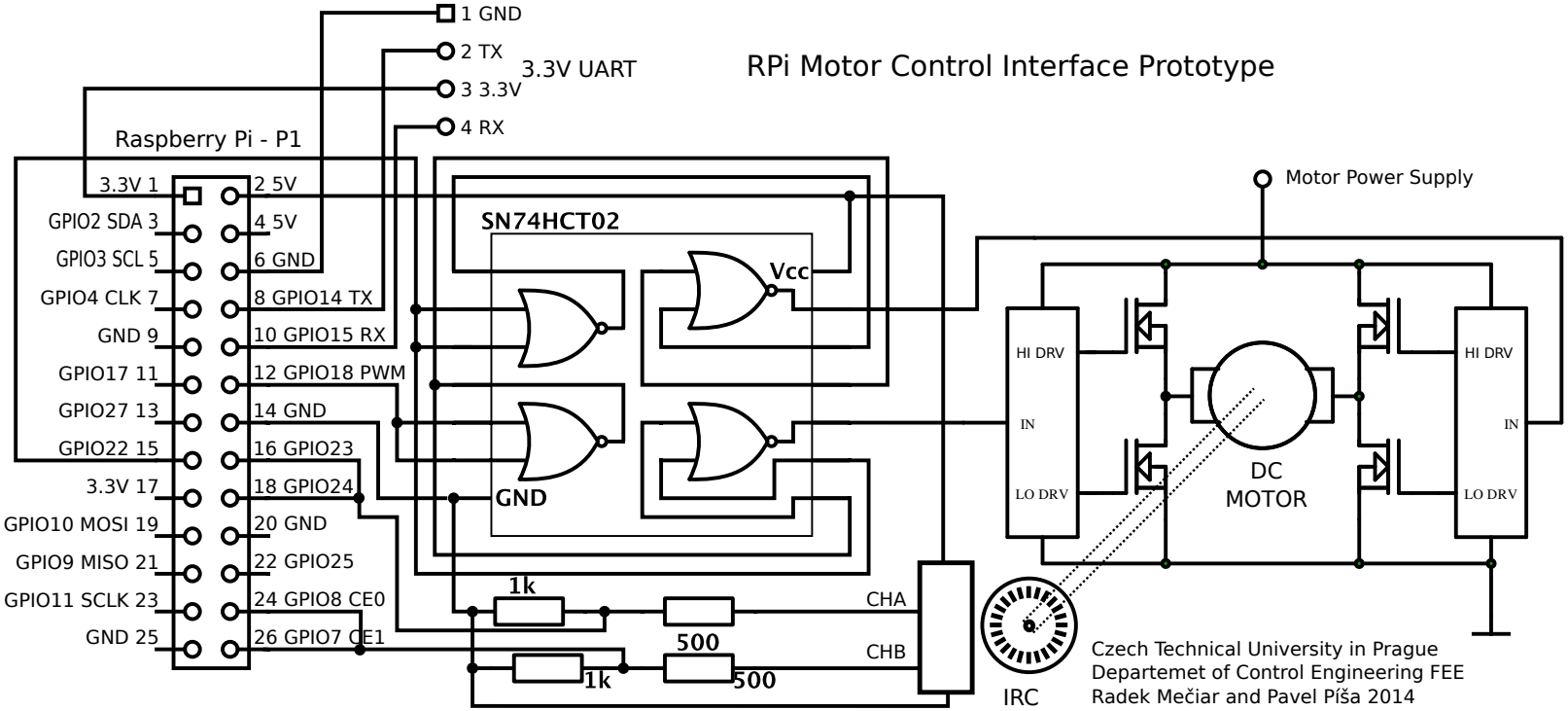
```
--with-arch=armv6 \
--with-fpu=vfp \
--with-float=hard \
--enable-multiarch \
--disable-sjlj-exceptions
```

Outline

System Events Rates

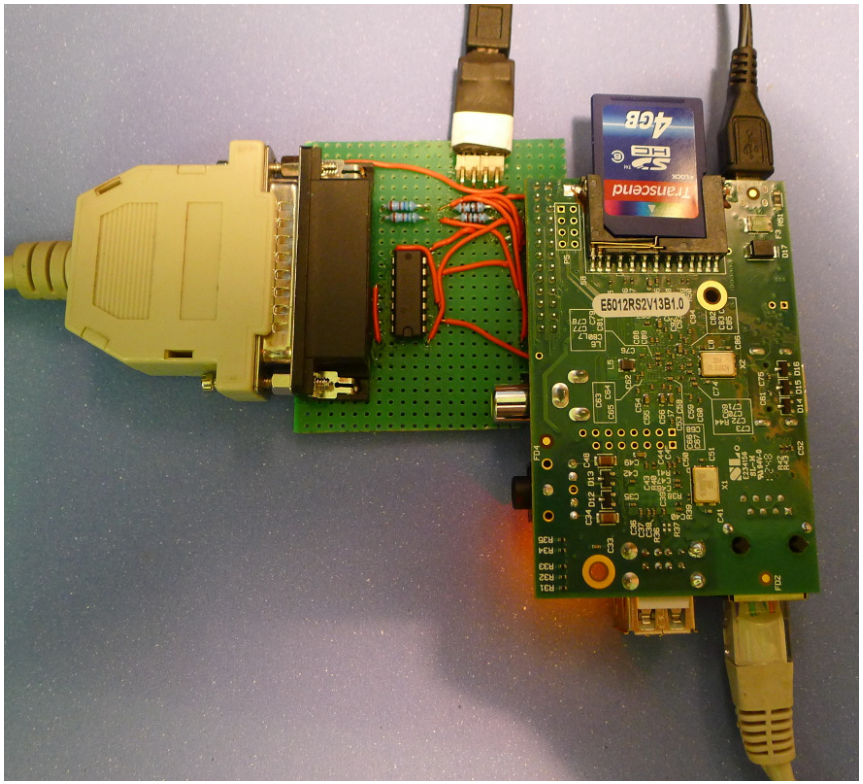
- The 1 kHz control loop sampling frequency can be achieved by RPi
- But RPi has no hardware for position (usually quadrature incremental) sensor interfacing
- Incremental encoder output changes frequency can reach MHz units
- For 500 slots wheel and 4000 RPM the required frequency is about 150 kHz
- This is too much for user-space and even proper kernel space processing on RPi but it is nice experiment for system stability under load testing
- Serious solution requires to extend RPi by incremental encoder interface implemented in hardware (FPGA)

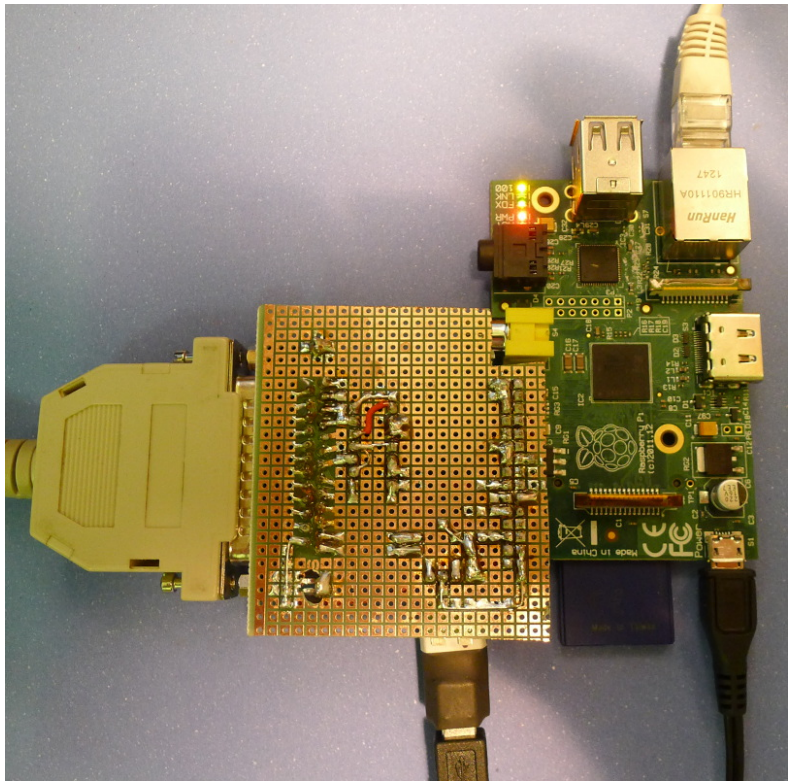
GPIO Only Based DC Motor Interfacing



- As simple as possible
- Four NOR gates (SN74HCT02)
- H-bridge (L6203)

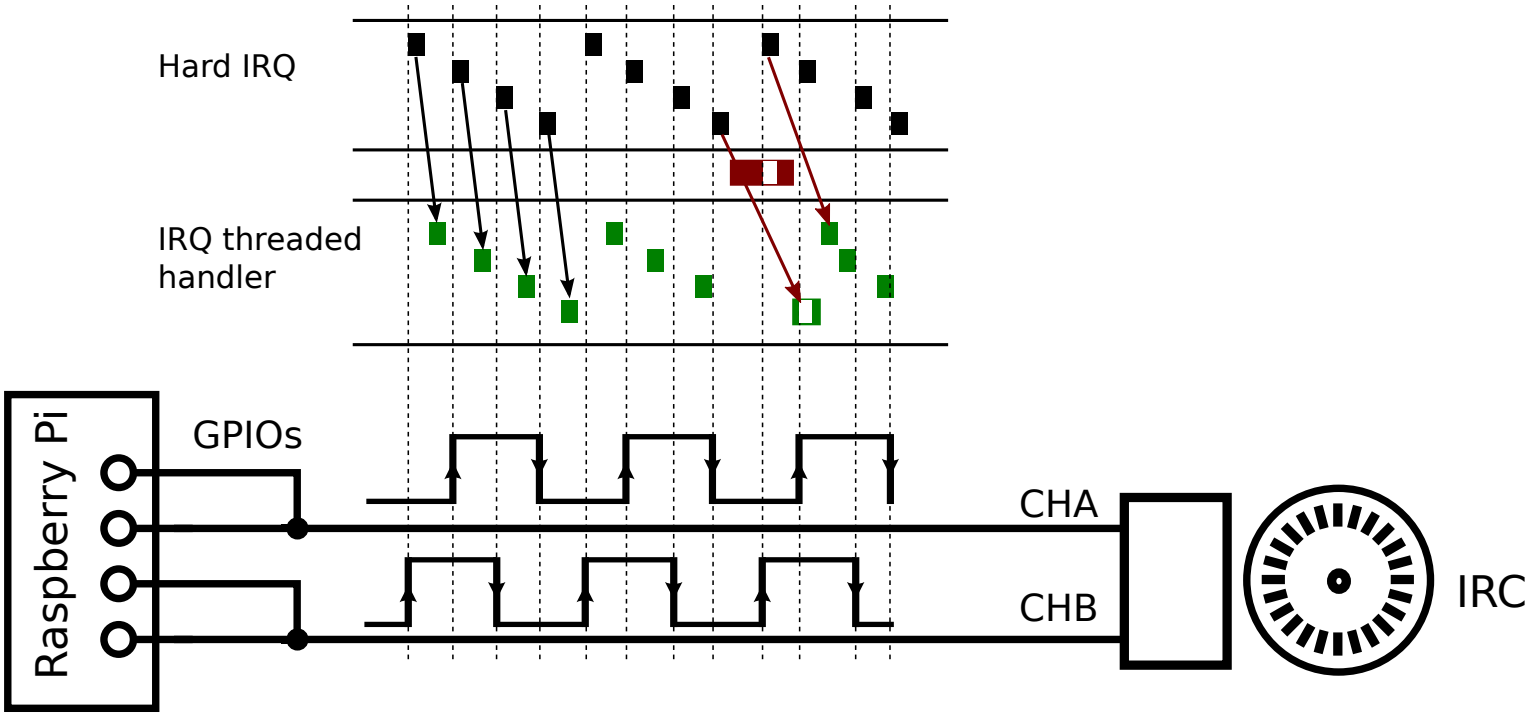
Wire-wrapped Prototype Design





- H-bridge (L6203) present on PSR course DC motor kit used

Software IRC Signals Processing



- Position calculation works better if derived from the order of IRQs than from the signal values read in the handler.
- FIFO run queue preserve order! (Observation from Mečiar Radek Motor control with Raspberry Pi board and Linux 2014 bachelor thesis)

IRC Kernel Driver

- Simple character device `/dev/irc0` which counts motor position
- Order of interrupts is converted to increment and accumulated in the kernel variable
- uint32_t (4-bytes) actual value is read from the device
- driver source available at GitHub repository
  `https://github.com/ppisa/rpi-rt-control`
  The core is a file kernel/modules/rpi_gpio_irc_module.c
- Test from user-space

```
modprobe rpi_gpio_irc_module
hexdump -e '"%d" ' /dev/irc0
```

IRC Access from C Code

```
int irc_dev_fd;

int irc_dev_init(void)
{
    irc_dev_fd = open(irc_dev_name, O_RDONLY);
    if (irc_dev_fd == -1) {
        return -1;
    }
    return 0;
}
int irc_dev_read(uint32_t *irc_val)
{
    if (read(irc_dev_fd, irc_val, sizeof(uint32_t))
            != sizeof(uint32_t)) {
        return -1;
    }
    return 0;
}
```

Boost IRC Kernel Threads Priority

- The fully preemptive kernel runs even IRQ processing in thread context, all interrupts use priority 50 by default

```
modprobe rpi_gpio_irc_module
IRC_PIDS=$(ps Hxa -o command,pid | \
  sed -n -e 's/^\[irq\/[0-9]*-irc[0-9]_ir\][ \t]*\([0-
9]*\)$/\1/p')

for P in $IRC_PIDS ; do
  schedtool -F -p 95 $P
done
```

- List threads by real-time priority

```
ps Hxa --sort rtprio -
o pid,policy,rtprio,state,tname,time,command
```

Power Output – PWM

- Only single PWM signal generator easily available on Raspberry Pi
- Direction has to be controlled by GPIO pin
- GPIO access possible through `/sys` interface

```
echo 22 >/sys/class/gpio/export
echo out >/sys/class/gpio/gpio22/direction
cat /sys/class/gpio/gpio22/value
echo 1 >/sys/class/gpio/gpio22/value
```

- But access over /sys represent significant overhead
- Direct access from users-pace to GPIO and PWM peripheral registers is used instead

Direct GPIO and PWM Registers Access

- Implemented in `int rpi_peripheral_registers_map(void)` function
- The physical address range can be accessed from user-space by `mmap()` syscall

```
mem_fd = open("/dev/mem", O_RDWR|O_SYNC)
gpio_map = mmap(NULL, BLOCK_SIZE, PROT_READ|PROT_WRITE,
                MAP_SHARED, mem_fd, GPIO_BASE);
```

- Detailed description at `http://elinux.org/RPi_Low-level_peripherals`
- Fast GPIO and PWM access functions for controller application can be found in files rpi_gpio.c and rpi_bidirpwm.c found in appl/rpi_simple_dc_servo example of `https://github.com/ppisa/rpi-rt-control` repository

Simple PID Based Speed Controller

- Implemented in file rpi_simple_dc_servo.c
- Next commands are available setpwm, readirc and runspeed.
- The real time task cannot be swapped or code paged in on demand

```
mlockall(MCL_FUTURE | MCL_CURRENT)
```

- real time priority has to be used for control task

```
pthread_attr_t attr;
struct sched_param schparam;
pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SC
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
schparam.sched_priority = sched_get_priority_min(SCHED_
pthread_attr_setschedparam(&attr, &schparam);
pthread_create(thread, &attr, start_routine, arg);
pthread_attr_destroy(&attr);
```

The Controller Timing

- The use CLOCK_MONOTONIC for all control related timing is critical, CLOCK_REATIME can skip forward and backward due to user or NTP adjustment

```
sample_period_nsec = 20*1000*1000;
clock_gettime(CLOCK_MONOTONIC, &sample_period_time);
do {
  sample_period_time.tv_nsec += sample_period_nsec;
  if (sample_period_time.tv_nsec > 1000*1000*1000) {
    sample_period_time.tv_nsec -= 1000*1000*1000;
    sample_period_time.tv_sec += 1;
  }
  clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                  &sample_period_time, NULL);
  /* Run timed actions there */
  ...
} while(1);
```

Ensure Proper Stop When Interrupted

```
void stop_motor(void)
{
  rpi_bidirpwm_set(0);
}

void sig_handler(int sig)
{
  stop_motor();
  exit(1);
}

...
  struct sigaction sigact;
  memset(&sigact, 0, sizeof(sigact));
  sigact.sa_handler = sig_handler;
  sigaction(SIGINT, &sigact, NULL);
  sigaction(SIGTERM, &sigact, NULL);
```

The Controller Sample Time Step

Please consult application code for complete solution with overflow and anti-windup protection

```
err =  (pos_req - actual_pos);
ctrl_i_sum += err * ctrl_i;
action = ctrl_p * err + ctrl_i_sum + ctrl_d * (err -
ctrl_err_last); ctrl_err_last = err;
rpi_bidirpwm_set(action >> 8);
```

More information can at pages of DCE's Real-Time systems Programming course `http://support.dce.felk.cvut.cz/psr/` and subject's Semestral Work – Motor Control pages. Other valuable information on former subject page `https://support.dce.felk.cvut.cz/pos/hlavni_uloha/`

Test Results

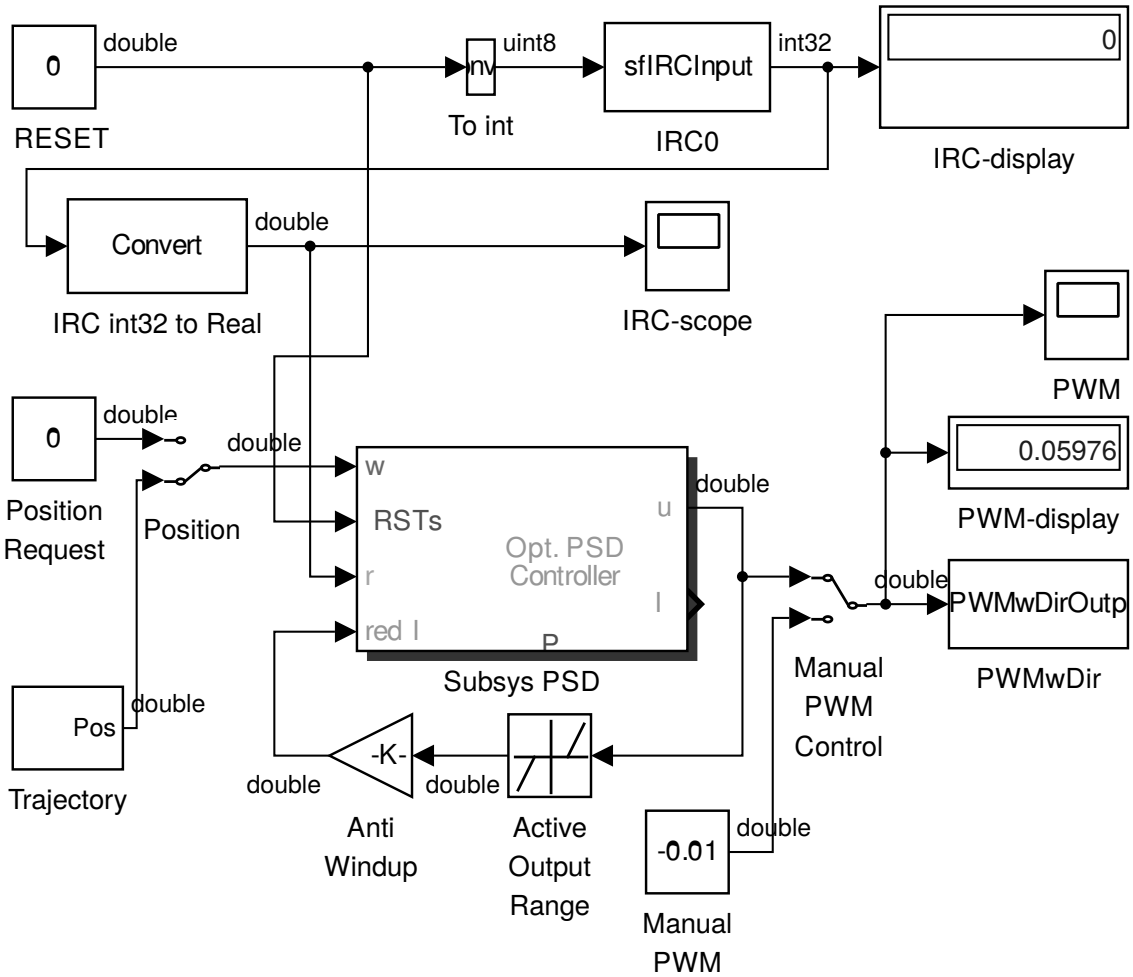
- Reliable speed control achieved for smaller speeds
- RPi capable to cope with almost full speed of PSR course DC motor with low resolution IRC sensor
- The speed limit is much lower for industry grade motor used in real PiKRON's applications
- The RPi is capable to process about 28 000 IRQ events per second but when more arrives the system and controller is overloaded/blocked and motor runs out of control
- But even in overload case system is stable when motor is externally braked/hold/slow down controller receives CPU time again and system recovers from overload
- Solution is nice for education but not safe for industrial use
- The test with FPGA based IRQ processing in in preparation but even in this case RPi is not considered by us as platform reasonable for industrial motion control applications

Outline

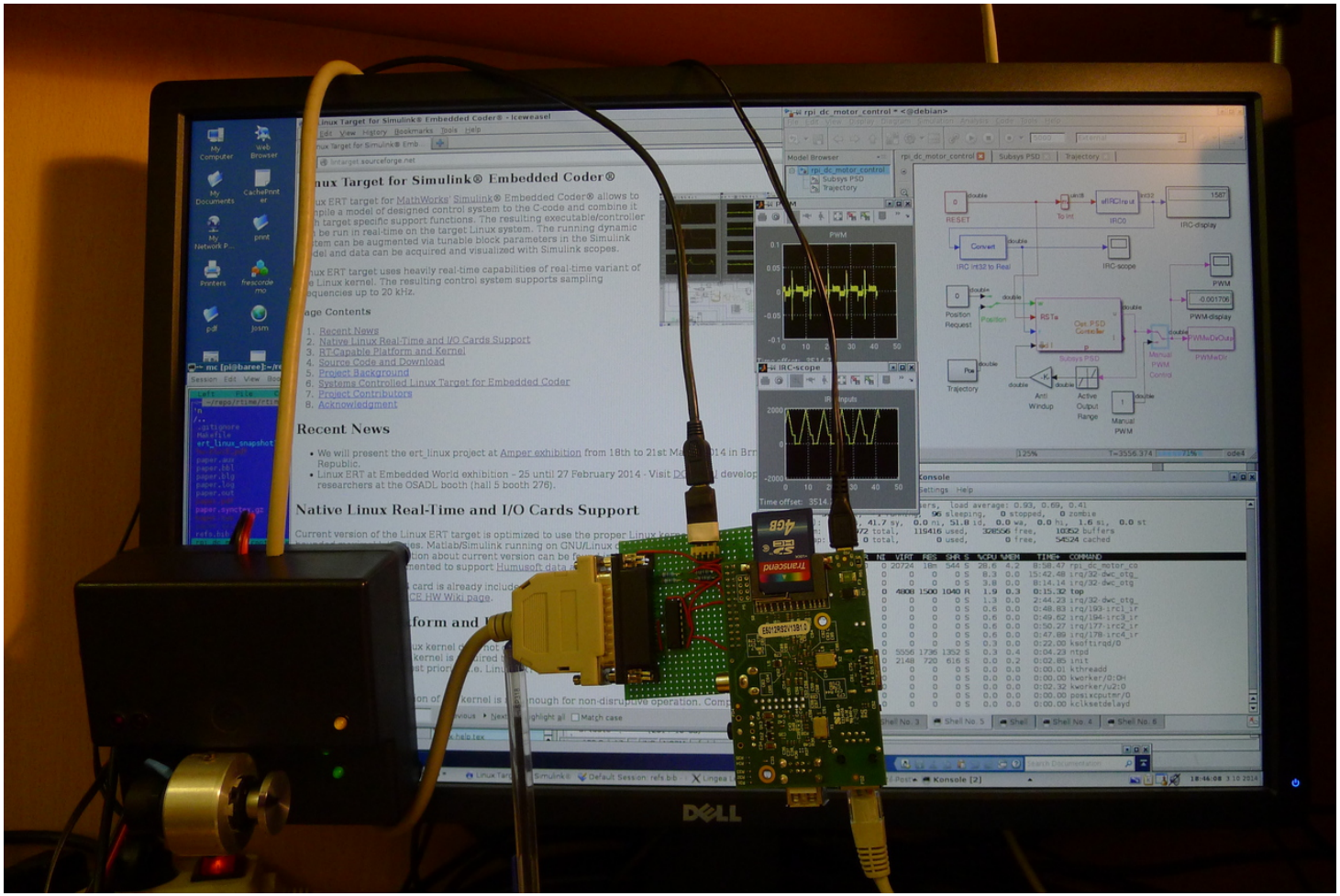
Embedded Real Time and the DCE Department

- CTU FEE Department of Control Engineering has been and is involved in Matlab/Simulink real-time support from its beginning (origin of real-time toolbox can be trace to our department)
- We have long term experience with fully preemptive kernel and hardware interfacing
- Embedded Real-time Target has been adapted/partially rewritten by Michal Sojka to be usable for real applications (MathWork included embedded solutions are often Windows only and use POSIX timers and signals which have uncontrolled latencies during delivery)
- The blocks for SocketCAN, Humusoft data acquisition PCI cards and minimal set of RPi periferals has been implemented
- COMEDI blockset has been updated and tested with our Linux ERT version as well

RPi DC Motor Control Simulink Diagram



RPi DC Motor Control Simulink Prototype



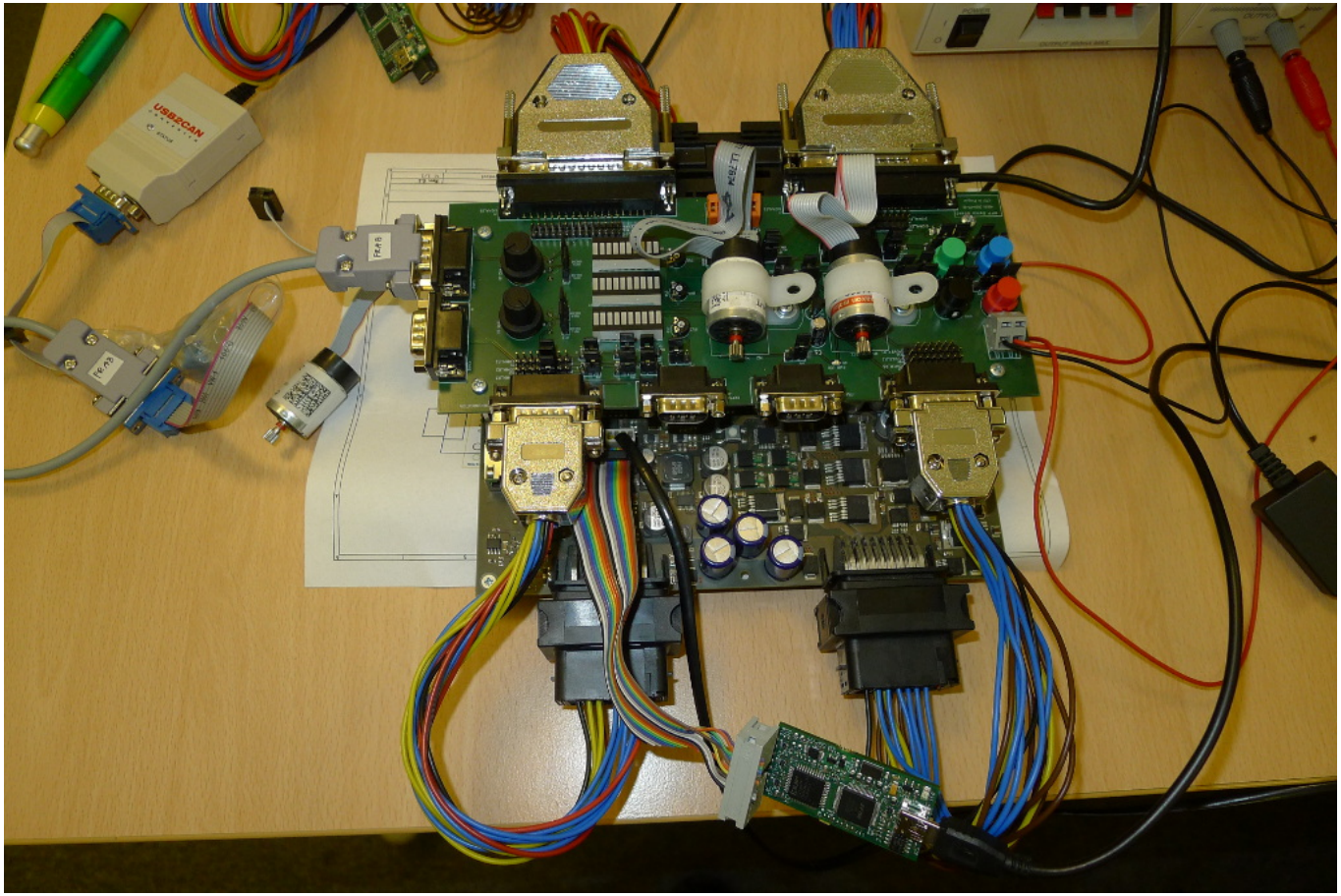
RPi DC Motor Control Simulink Remarks and Pointers

- Incremental encoder input implemented as S-function sfIRCInput.c which opens `/dev/irc0` and reads actual position from kernel driver
- Bidirectional PWM output is implemented in S-function sfPWMwDirOutput.c and uses same registers direct access approach as described in the previous section
- The whole setup is documented on respective Lintarget/Linux ERT project page `http://lintarget.sourceforge.net/rpi-motor-control/index.html`
- used ert_linux target and CC=arm-rpi-linux-gnueabihf-gcc set for make_rtw. scp and ssh are used to copy and run binary on target. Simulink external mode (parameters on-line tune and signals scope windows) is available.
- The generated code performance is the same as for hand written case – limitation is IRC events processing in the kernel

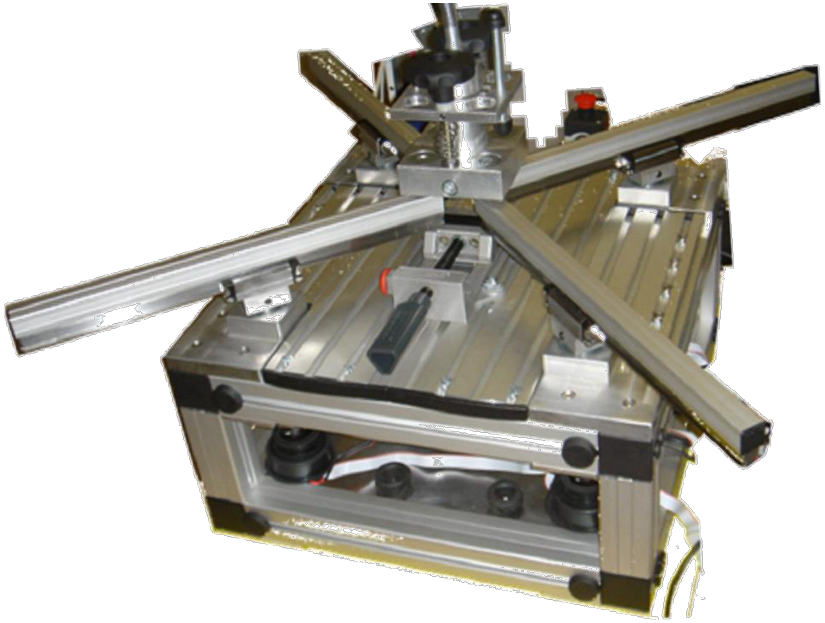
SocketCAN Simulink Blockset

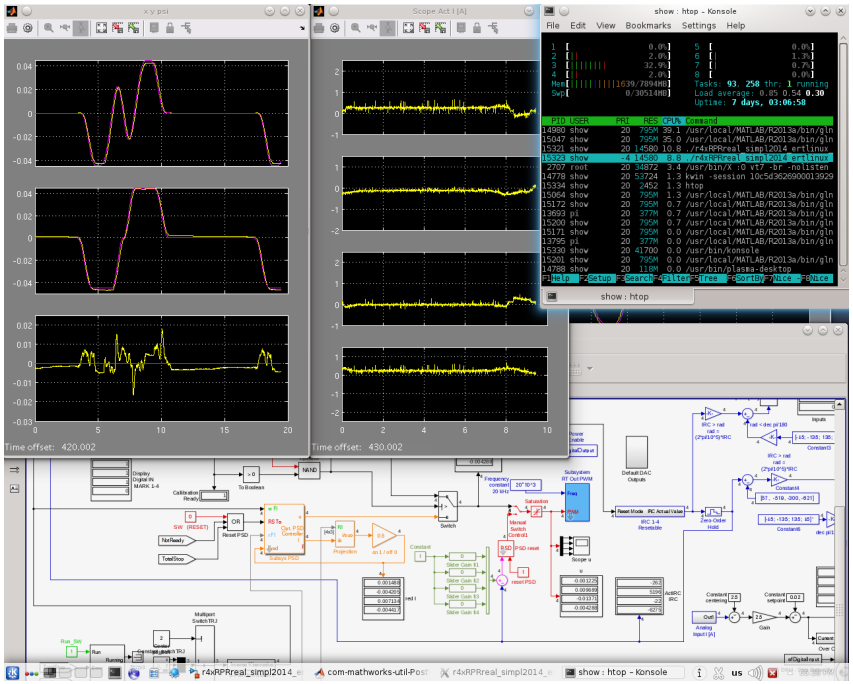
- The blockset is quick proof port of the CAN Autosar API based blocks developed at DCE initially for own automotive grade ARM Cortex-R4 based embedded platform
- The code is generated under designed control of TLC (Target Language Compiler) blocks description which allows to optimize blocks code for used data-types and interconnection
- For more information about embedded systems rapid prototyping support developed in our group look at `http://rtime.felk.cvut.cz/rpp-tms570/`
- Notices about more Linux and embedded hardware used, tested and even some designed look at `https://rtime.felk.cvut.cz/hw/`

Cortex-R4 Automotive Platform and Test Board



x86 Linux ERT and Parallel Kinematic Robot Control





- 4 DC motors, 4 incremental encoders, other I/Os
- Presented at Embedded world 2014
- Sampling period 1 ms but complex computations
- More reliable that previously used Windows target

Conclusion

- More ready to be uses open-source building blocks for control applications have been presented and are available online
- We are looking for students who has interest in real-time, operating systems and control/embedded hardware
- We cooperate with more industrial partners on many projects and students can gain experience and valuable knowledge during their work on the project in frame of thesis
- We offer control related courses Real-Time systems programming and participate on generic computer architectures courses at CTU FEE

Thanks for attention and questions