Albrecht Weinert

Raspberry for remote services

Abstract and Introduction

This technical report is about installing and using Raspberry Pi machines as little servers and for embedded process control applications.

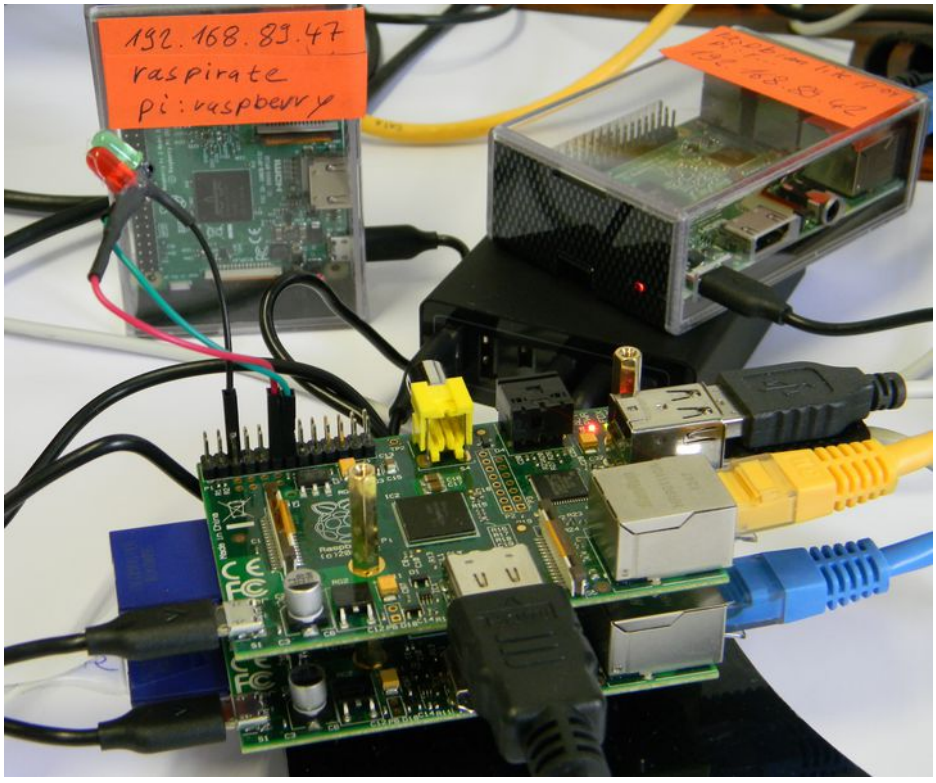

Fig. 1: The Raspi Zoo
Note: The Pi 1s in the foreground we only use for comparison. The same is to be said on the Raspirate distribution.

Target machines and operating systems

Target machines are the Raspberry Pi3 and quite seldom P1, see Figure 1.

The operating system will mostly be Raspbian Jessy lite. Raspbian is a Debian derivative as is Ubuntu. Raspian, of course, is no real time OS. Nevertheless, obeying some caveats the lite variant may well be used in applications where latencies up to 200µs are tolerable. Hence, we can have PLC like applications with 1ms as fastest cycle.

In one case we used piXtend board and stainless case to extend a Raspberry PI. For some 200€ piXtend tries to resemble the standards of industrial process control. It offers graphical Codesys configuration/programming (for 50€ + VAT extra).

Motivation and goals

One might use a Raspberry 3 as a little low performance PC, but well outperforming first PCs. Usually a Raspberry will be configured as a device for special purposes, often embedded. Then monitor, keyboard and mouse will hardly be present except for tough maintenance jobs and very first start. Only when part of the application, monitor etc. will be kept attached, of course.

Hence we will have a headless server or device that

- boots quickly and robustly at power on
- starts all its applications and/or services automatically,
- runs 24/7
- fits, connects and re-connects robustly in its LAN and/or WLAN
- connects to process I/O via GPIO, SPI, RS232/445, private LAN, CAN or other
- provides administrative access remotely via ssl/ssh/putty
- provides administrative access to files via ftp (FileZilla, WinSCP)
- thus offering a decent comfortable working environment,
  best on a Windows workstation, as to have an operational clipboard

Looking up those points we find us quite near the properties we must have in a “real” server, compare the Fujitsu Siemens RAID examples in [29] (notwithstanding the latter being a 1000 times heavier). Hence, some solutions and procedures may look astonishingly alike.

On the content

In **Part I** we describe the basic installation and commissioning with Raspberries.

Here we get the common knowledge and procedures and tools almost always needed when working with these (little) systems.

Part II deals with some use cases.

Main points are using (process) IO and implementing automatically starting, unattended running applications respectively services. We look at hardware interfacing, Web HMI, GPIO libraries, timing, latencies, threading and cyclic tasks, using little exemplary applications.

Using names

Names and addresses used here (and in [27..31]) are not fictitious. This helps bringing real and really working and proven examples of commands, files, outputs etc. – without errors introduced by obfuscating. Of course, you'll have to adapt IP addresses, names numbers etc. to your environment and needs, even when this will not always be explicitly mentioned.

For References and Abbreviations please see the Appendix (page 47)., Refer to [29] for some Linux and server basics, abbreviations and else, also used here.

Dr. Albrecht Weinert is computer science professor at
Bochum University of Applied Sciences or Hochschule Bochum.
He is founder and director of
MEVA-Lab – Laboratory for versatile distributed applications –
as well as of the service provider weinert - automation.
albrecht@a-weinert.de

Rev. 1.19c 08.12.2017

Table of Content

P A R T I

Basic installation

The OS images mostly used are

- NOOBS
- Raspbian both of them in “lite” and not lite variant.

The obvious characteristics of the “lite” variants is Raspian or another distribution having no GUI and bringing no graphical tools. Standard Linuxes call that a server distribution. With NOOBS the “lite” variant means bringing no bundle of distributions requiring internet access at the very first boot (to get a new system out “of the box”).

Raspbian

Raspbian is the most used Linux for Raspberry; it's based on Debian. The distributions are commonly downloaded as .zip. Unzipping it with command line tools, like jar, 7zip or in the explorer reveals one disk file (.img):

```
10.04.2017   1.297.862.656   2017-04-10-raspbian-jessie-lite.img
or  "heavy"  4.285.005.824   2017-04-10-raspbian-jessie.img
05.07.2017   1.725.629.563   2017-07-05-raspbian-jessie-lite.img
```

We started with April version and are at the newest / last July version now. With the still newer “Stretch” instead of “Jessie” Pi3's WLAN won't work.

The .img is a pattern for the sole content of a µSD (Pi 3) respectively SD-card (Pi1). The Raspberry will boot from that card and at the first boot make it its SSD.

That “burning” of the .img is done on a Linux/Ubuntu workstation by command line. Example:

```
lsblk
sudo umount /dev/mmcblk0p2
sudo umount /dev/mmcblk0p1
cd ~/Downloads/

sudo dd of=/dev/mmcblk0  if=2017-04-10-raspbian-jessie-lite.img
  bs=4M

sudo umount /dev/mmcblk0
```

Insert the [µ]SD; the the first command (lsblk) reveals the often dazzling names given to the SDcard and its partitions.. Be very sure to recognise the [µ]SD card!

With the next commands un-mount all partitions (p1 …) if any; do not un-mount the device. From where you unzipped the .img to use the dd command to burn the .img.

Using Windows tools (best Win32DiskImager), you would get a simple drive letter – but here the same warning applies.

Notes:

- dd and Diskimager (write) will destroy all former content.
  Be really sure to have the right device (respectively drive).
- Be patient. Burning may run some minutes (up to 2 per GB) and dd gives no feedback.
- If dd ends immediately, something went wrong (check the return code, if in doubt).

Un-mount the device, remove the [µ]SDcard, put it in the target Raspberry and proceed with the chapter “First commissioning” (page 6).

Note on (semi) automating the burning: On Windows' mostly pure graphical tools the .img burning may hardly be (batch) automated. And if the tools have a command line there's still the “drive letter surprise” effect. Here Linux/Ubuntu is hardly better as you have the funny device and partition name surprise.

NOOBS

Contrary to Raspbian being a concrete Linux distribution for Raspberry, NOOBS is a street hawker offering a bunch of distributions on first boot ready to be installed. In the lite version the offers look the same but hawker's tray is empty and what you order NOOBS will let the Raspberry fetch from internet.

Hence, when knowing the distro to be used, or doing multiple similar installations NOOBS (lite) hardly makes sense. instead of getting an .img of the distribution to be installed, hardly makes sense. With the outdated Raspberry Pi1, on the other hand, NOOBs came to rescue, when distros downloaded wouldn't install or start.

The “burning” of NOOBS to a card is different – it's no burning:

Unzipping the downloaded .zip file reveals a directory tree. This has to be copied into root of the [µ]SDcard. Hence, all can very well be done in a Windows workstation with the explorer or shell alone – except for formatting the [µ]SDcard in a Raspberry compatible way.

Her fore you may get and use the tool SDformatter with appropriate settings, see Figure 2.

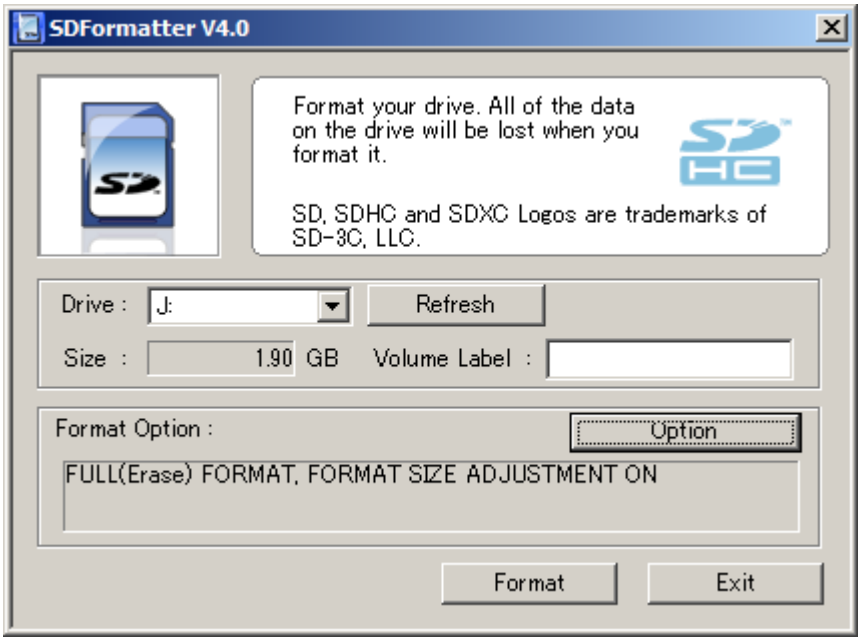

Fig. 2: SDFormatter.exe

After formatting the [µ]SDcard and copying the NOOBs tree to it, remove it and put it in the target Raspberry. When asked choose the distribution and proceed with “First commissioning”.

First commissioning

These are first steps to bring in a new OS with a freshly prepared [µ]SD card.
Here we almost always need local as well as [W]LAN access to the Raspberry. So

- take the not powered target Raspberry and put the card in
- get the machine to mouse, keyboard, LAN and monitor
- and at last re-connect power.

When all went well, you come to the point when prompted to log-in. The pre-defined user is “pi”, password is “raspberry”. As you almost always start with an US image and as there's no real installer – the keyboard layout and many other things are utterly wrong.

As a first consequence, the standard login will fail, while pi:rasberr**z** will succeed. If pi:rasberrz and pi:raspberry both fail, someone had the prudence to change the public first log-in of a public image.
Note: When making “public” name:password (for images) never use any keys that are misplaced on US keyboards nor special ones on German or French. And never rely on the existence of the numeric pad.

After taking the login hurdle search the – (minus) or use the numeric keyboard to enter

```
raspbi-config
```

To use this tool, you need space, cursor keys and tab. Here the keyboard bug should not be a problem. Otherwise you're lost. Now use raspi-config to

- set your locale
- the keyboard layout
- the time zone
- your country's WiFi settings
- and enable ssl (found in interface options)
- ~~enable~~ using all the rest of the µSD as disk (done automatically with NOOBS)
  *do **not** do this before reading on "size problem"* below
- change the (host) name (*this may not really work with raspi-config)*
- enable serial interface, but not as console *(if you wish to use it for devices)*

With working LAN or WLAN and if in a network with a DHCP server you should get the Raspberry's IP address(es) by: `ifconfig`

If the machine met the same DHCP server before the address(es) should stay the same.

Do not forget to enable ssl. It is the condition to get a remote shell via ssh or putty. Using putty on Windows instead of Ubuntu gives superb clipboard support that sooner or later will get essential for any remote work.

Setting hostname

When having more than one Raspberry in the [W]LAN you must give each one an unique name.

With Raspbian changing the host name must be done in three places:
by both editing /etc/hostname as well as /etc/hosts and by the hostname command:

```
sudo nano /etc/hostname
sudo hostname theNewName
sudo nano /etc/hosts       # correct / set 127.0.1.1 entry here
```

raspi-config may do the first step, only, leading to schizophrenic effects and error messages.
Note: With the last Jessie and with Stretch this problem is gone.

Enable WLAN

The Raspberry Pi3 has WLAN on board. Pi1 had none and fewer USB ports, one of which would go for a WLAN device. Pi3's build in WLAN's only disadvantage of the Pi3s is the antenna being fixed to the board. Encasements with good protection and shielding will render it unusable.

While LAN (with DHCP) is working out of the box, WLAN will require some settings. This is best done by command line, as well described in [53].

A most useful command – even when not wanting to use WLAN – is:

```
sudo iwlist wlan0 scan
```

This gives a list of all visible WLANs. In our case it didn't see the 5 GHz cells. This was expected as the Pi3 is said to support 2.4 GHz only, even if the Broadcom chip probably should do both. If not sure if the WLAN is wlan0 use ifconfig. From the iwlist scan output (excerpt below) we see one cell of our WLAN in acceptable quality (atMEVAnet here) we want to connect to.

```
Cell 03 - Address: 44:94:FC:88:B0:40
                   Channel:9
                   Frequency:2.452 GHz (Channel 9)
                   Quality=59/70  Signal level=-51 dBm
```

```
                    Encryption key:on
                    ESSID:"atMEVAnet"
                    Bit Rates:11 Mb/s; 12 Mb/s; 18 Mb/s
                    Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
                    Mode:Master
                    Extra:tsf=0000000000000000
                    Extra: Last beacon: 80ms ago
                    IE: IEEE 802.11i/WPA2 Version 1
                        Group Cipher : CCMP
                        Pairwise Ciphers (1) : CCMP
                        Authentication Suites (1) : PSK
```

To make the Raspberry connect to one WLAN we have to append the name and encrypted password in a given syntax to the configuration file /etc/wpa_supplicant/wpa_supplicant.conf. This appendix is almost generated by

```
wpa_passphrase atMEVAnet
```

“almost” here means one should obfuscate or delete the clear text password comment when appending the command output to /etc/wpa_supplicant/wpa_supplicant.conf.

The result will be something like:

```
country=DE
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
        ssid="atMEVAnet"
        priority=1
        #psk="Kr........wunkel"
        psk=e891c0f1aw3746098509ae92612cafe4d844b06cee0aab8a178488a88bb66712
}
```

So far this file is not device specific. It can be transferred to other Raspberries.
Note: This file, stupidly, requires tabs (for the indentation required).

It's also possible to add more than one network={} entry, giving all an additional priority attribute with different values. If the configuration change has no effect, force it by:

```
sudo wpa_cli reconfigure
```

Force IPv4

Even when all your LAN addresses are IPv4, you may notice wlan0 using Ipv6 only, even when the access points live in the same LAN and the DHCH server distributes IPv4 to all.
Note: In our case WLAN's the Pi's imaginary IPv6, of course, didn't work.

To force Raspberry (Raspbian) to use respectively accept IPv4 after the next reboot you may add the line

```
net.ipv6.conf.all.disable_ipv6=1
```

to /etc/sysctl.conf.

Now it's a good time to implement the tip “Make consistent and comparable directory listings”; see Appendix page 47. And, if your workstation is ready, you may work on with the Pi and install the recommended pigpio library (page 29) and libmodbus (page 38).

Configuring putty

With ssl enabled, from a workstation or laptop in the same network connect to your Raspberry with ssh. Better use putty respectively putty.exe when expecting more than two minutes remote work. And making a good putty configuration is well worth the trouble.

Start with nice and readable colours, think about font, window title as well as on clipboard+mouse behaviour. When satisfied give the configuration a recognisable name, like e.g. rasp61. Best use the Raspberry's unique hostname. When needing multiple accesses (LAN and WL AN) you'll need extra names; as the configurations are connection specific. To re-use the configuration command

```
putty -load rasp61 -l pi
```

Change the “-l pi” (meaning log-in as user pi) when wanting another user account to log-in or omit the “-l name” option to interactively entering the user.

Best make an icon with the command if needing this concrete connection regularly.

After having spared no efforts in making a good configuration for this one Raspberry connection we want to reuse the settings, mutatis mutandis, for other machines. You may use the putty GUI by
a) load, b) modify c) save and/or open.

But there's an other way for making putty configurations that also crosses workstation borders easily. On Linux/Ubuntu those configurations are just text files of the configuration name in the (hidden) directory

```
~/.putty/sessions/
```

Copy the pattern (file) you like best to a new name and edit it according to the new purpose. Look for host names respectively IP addresses and window titles, lest being in for some confusion.

On Windows putty.exe holds configurations in the registry. That seems to make re-using configurations more complicated, but it does not.

The place to look for is

```
[HKEY_CURRENT_USER\Software\SimonTatham\PuTTY\Sessions\theSession]
```

Once you have a nice pattern you want to reuse do

```
regedit.exe
```

and navigate to

```
[HKEY_CURRENT_USER\Software\SimonTatham\PuTTY\Sessions\]
```

Export the configuration/session/key you like as text-file sessionPattern.reg to a directory, where you want to organise your putty configurations. Copy and name it accordingly open it in a pure text editor (best editpad.exe).

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\SimonTatham\PuTTY\Sessions\lite042]
"Present"=dword:00000001
"HostName"="192.168.89.42"
"LogFileName"="putty042.log"

::::::
```

Make the appropriate changes as described above. And do never forget to change the key name (between \ and] in the third line consistently. Then, save the changes and double click on your new .reg file. And, voilà: a new session/configuration.

As with the Linux files the .reg text files may be transferred to other machines.

Enabling file access

For administrative and programming work remote file access from full grown Linux/Ubuntu or Windows workstations is essential. Accessing headless / GUI-less systems that way, you'll get decent editors and (on Windows) an operational clipboard handling worth the name.

By default most Linux distributions for Raspberry including Raspbian will not have a ftp server. But almost all of them, will have SSH – you should have enabled. This offers remote file access by protocol sftp. SSHFS, FileZilla and WinSCP *) can handle that protocol as clients.
Hence, no ftp server installation on our Raspberries will be needed.
Note *: Windows' ftp.exe can't – but on Windows you've WinSCP.

Using SSHFS (not recommended)

On the Ubuntu laptop (e.g.) and on other Raspberries you may use the SSHFS client plus directories to mount your Pies' file systems on:

```
mkdir -p ~/Fhpis/61       # example empty directory as mount point
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install sshfs              # geting fuse libfuse ...
```

Thus having sshfs on your Ubuntu WS or other on another Pi, you may connect and mount by:

```
sshfs pi@192.168.89.61: ~/Fhpis/61    # accept host's key for ever
```

Now you can list (ls) the directory used for mount (~/Fhpis/61 in our example) or work on it in the explorer – whatever its name and feature set is on diverse graphical Linuxes.

Note 1: As expected one only sees the respective user's home directory tree (~) remotely in the mount – not the Pi's root and else.
Editing /etc/fuse.conf to allow the sshfs's “-o allow_root” option is of no avail as the root password is disabled in Raspbian.

Note 2: The sshfs client software seems not to made with focus on robustness. When loosing the connection (weak WLAN e.g.) open explorers and shells on the Ubuntu WS grind to halt.

Note 3: To make the behaviour criticised in note 2 worse, there is no proper way to disconnect or unmount a sshfs connection, neither by sshfs nor by fuse.
Not even “sudo umount ...” worked nor did any googled trick in our case.

Hence with sshfs your workstation keeps being tight to the Rasberries with and all risks until reboot. Well:

```
ps aux | grep sshfs
sudo kill -9  PIDshownByPs
```

killed all connections, but made ~/Fhpis/ – i.e. all mount point sub-directories – unusable until re-boot.

Due to these sshfs flaws, when feasible, use FileZilla for graphical access – on all platforms.

For command-line and automated batch processing use WinSCP ([55]), see the chapter “SFTP – WinSCP” (page 11). WinSCP is simply the best and no Linux equivalent is found so far.

SFTP – FileZilla

You really should have FileZilla on your Ubuntu and Windows workstations.
You may even have FileZilla on a graphical Raspbian, by:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install filezilla
filezilla &
```

Now set up FileZilla as client:

Open FileZilla, goto site manager, make a new site and name it accordingly, say “raspi61”.

In general setting do: host = 192.168.89.61; protocol: sfpt; login: normal, pi, raspberry.
In “advanced” tick “bypass proxy”, when your workstation and Pies share a local network.

That's it. Connect should work. For every other Pi just copy and modify IP and name. Enjoy

- disconnect and re-connect working perfectly.
- comfortable view/modify integration with text editor set (best editpad on Windows)
- see and explore all files from root (/) on, not just /home/pi/

The last point allows, e.g., to comfortably view /etc/apt/sources.list (with editpad on Windows) and copy its content comfortably by clipboard.

Modifications in the editor on the (automatic) local copy will be possible, but FilleZilla's mirroring the changes back will fail for files with root:root permissions only. But we always can see, work on, copy, use clipboard etc. on multiple windows in an comfortable environment.

One workaround for modifying system files is opening a putty connection in parallel with FilleZilla and transfer system files to be comfortably worked on to and (sudo) from a user working directory.

SFTP – WinSCP

WinSCP is considered as one of the best FTP client programmes – not for giving us the n + 21st graphical FTP surface but as powerful command line programme and for its automated batch processing capabilities.

To install it, download the .zip file of a portable build, unpack it and move the three files

```
19.04.2017  14:44           282.960   WinSCP.com
19.04.2017  14:44        18.905.808   WinSCP.exe
29.05.2017  18:40            14.497   WinSCP.ini
```

to a path directory (C:\util\ in our case).

```
WinSCP.com
```

will get put you to the command line client. Play with it starting with 'help', 'open', 'close' and 'exit'. Some find WinSCP a bit bitchy on first encounter. If so, don't give up – once you master it you won't miss it any more.

```
winscp.com /script=progTransWin
```

will transfer two programmes from the actual directory to a (Raspberry Pi) target machine by the WinSCP script progTransWin. Such script can be parametrised and in the end used in a generalised transfer recipe in an Eclipse C make project; see the downloadable examples given in Part II.

```
# transfer programmes rdGnBlinkBlink and rdGnSimpleBlink
# to the target machine
# Copyright 2017 Albrecht Weinert           a-weinert de
# $Revision: 2 $ ($Date: 2017-05-29 18:37:55 +0200 (Mo, 29 Mai 2017) $)
open sftp://pi:raspberry@192.168.89.67
cd bin
option batch continue
option confirm off
put rdGnBlinkBlink -preservetime -permissions=775
put rdGnSimpleBlink -preservetime -permissions=755
exit
```

Listing 1: WinSCP script “progTransWin”for batch transfer of two programmes (see Part II).

Note: Looking at the Part II example's enhanced script and the make file you'll notice again that we found no Linux/Ubuntu equivalent. No free FTP command line tool for Linux with WinSCP's flexibility, features and professional quality seems to exits.
We tend to blame our incapacity to search for such Linux programmes. On the other hand we even found expert comments saying WinSCP is keeping them with Windows.
What we got as nearest to WinSCP's elegance – and have often used for automated FTP transfers from Linux servers in the past – is LFTP. On the other hand recurring fingerprint/certificate refusals with sftp:// and parametrising with make variables may drive you nuts – all this a no topic with WinSCP.

Transferring to non-home

Consider the automated (i.e. scripted and parametrised) file transfers with WinSCP shown in Part II (in the demo project). Extending the automated application deployment to the target Raspberry a bit further leads to the deployment of (updated) libraries and hence to scripted writing to non home/ non user pi's directories like e.g. /usr/local/lib/ for libraries.

Playing with WinSCP as pi:raspberry we can see, list etc. those directories (or at least a part or them) but can't write. And we can't remote sudo in WinSCP.com.

To enable WinSCP remote transferring to non home/pi directories is to give root a password and enable root to ssh log-in. Then we can

```
winscp.com root:rudolpf@192.168.89.67
```

from command-line or within a script. To enable this do:

```
sudo passwd root
sudo nano /etc/ssh/sshd_config
```

At the entry PermitRootLogin exchange the setting without-password by yes.

Note: The usual setting “without-password” is misleading. It does not allow root log-in without password; it forbids log-in even if you have one.

Note 2: Giving root a password to log in has security implications. Consider which networks your Raspberry is in and which influence it has on others or process IO.

Note 3: Part II's demo project may be checked out (as guest:guest) by
svn checkout https://weinert-automation.de/svn/rasProject_01/
or just seen it in the browser.

Save and restore – and clone

For the basic installation we prepared the [µ]SD card with an installation image downloaded from a trusted source on an Ubuntu PC/laptop by the dd tool:

```
sudo dd of=/dev/mmcblk0 if=2017-04-10-raspbian-jessie-lite.img bs=4M
```

This copies a disc image (from the .img file) to the target card.

On a Windows PC/laptop we can do this interactive with the “Disc Imager” (figure 3):

```
C:\util\ImageWr1ter\Win32DiskImager.exe
```

Consider the facts:

- Those tools do also work the other way round.
- A Raspberry's [µ]SD card is its one and only SSD
  (in all standard “single drive” configurations).

Hence, both tools – dd and Win32DiskImager – seem good for complete save, restore or copy/port to other device in any installation or working state.

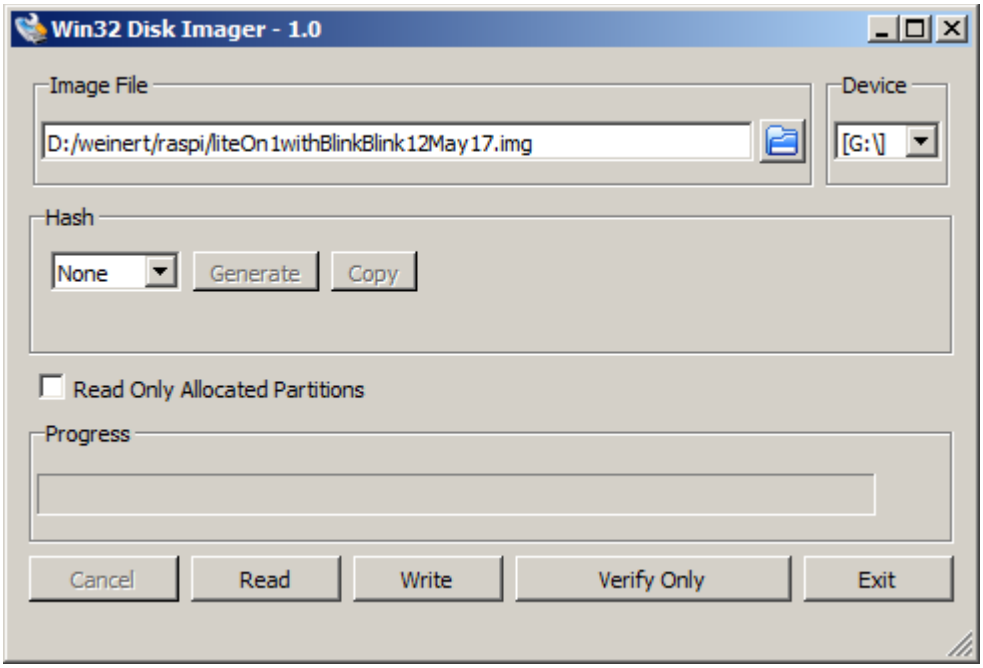

Fig. 3: Disk Imager

To save the actual state
- shutdown and un-power the Raspberry,
- remove the card and insert it to your laptop/PC

To save as .img file with Win32DiskImager chose a target file and click “Read”, see figure 3. This file may not exist, just chose an appropriate name.

To write the saved image to another SD card, one of the same size as the original should work. A bigger one is always usable, and has to be used if Win32DiskImager complains on size.

Warning: When hitting the size problem and ticking “write anyway” Win32DiskImager crashes. And that brings the drive used (G: in fig. 3) in an unusable and not removable state.

Two caveats for Win32DiskImager:
- Do not go ahead when getting complains on the operation planned. Tick exit.
- Be extremely careful with “Write”.
  Double check the drive letter. Win32DiskImager can kill your system.

For the size problem as such Win32DiskImager is not to be blamed. Using dd with the same devices on an Ubuntu laptop finishes the read and write operations (taking ages) without crashing but complaining on the non fitting size afterwards. The card then will start to boot in the Raspberry but will grind to a halt on root sector errors. Doubling the SD card size does the job – as with Windows. And reducing the -bs size is of no avail.

Note: On the Ubuntu laptop recognising a SD card – both directly or, cause of not fitting slot, via an USB adapter – seems sheer luck. The device may neither appear in lsblk nor fdisk. In the USB case the command lsusb sometimes may trigger the recognition.

The following commands should save the current state of a Raspberry in an .img file.
The 4G source SD card is put in an USB adapter as the modern laptop eats only µSD.
Here it may appear as sdb when having luck or when the lsusb triggering works:

```
sudo fdisk -l
lsusb
sudo fdisk -l
sudo dd of=~/FHpis/61/Noob4lite41b_may17.img if=/dev/sdb bs=4M
sudo dd if=~/FHpis/61/Noob4lite41b_may17.img of=/dev/mmcblk0 bs=4M
```

The write command to µSD (mmcblk0) comes to an end without complains with 8G cards but fails with the “fitting” 4G, in all cases when the Win32DiskImager would complain.

The size problem

The root cause of the “size problem” most probably is no two [µ]SD cards having the same size.

By “use all rest of card as drive” in raspi-config this deficiency is made virulent. But that is often done on first configuration. And when using NOOBS “use all card” will be done automatically. Hitting this problem is very annoying. It effectively forces double size cards for restore or copy of saved system states.

In most cases raspi-config's enlarge command should just be omitted. Without it we see about 3.7G used of a 4G card (which seems be well below the lowest size category of cards sold as 4G). Without enlarging, we never had problems to clone such card with the Win32DiskImager.

The too clever dd problem

When cloning a µSDcard with dd, the obvious way is via an .img file with two dd commands with interchanged 'if=' and 'of='; example:

```
sudo dd of=~/FHpis/61/Noob4lite41b_may17.img if=/dev/mmcblk0 bs=4M
# remove source card, insert destination card
sudo dd if=~/FHpis/61/Noob4lite41b_may17.img of=/dev/mmcblk0 bs=4M
```

Especially when

- using two cards of same size and type and
- getting no complains and seeing an .img file of fitting size produced … –

then one intuitively expects a one to one / bit by bit clone.

But. alas, dd more than often does not do this. The best case are missing partitions or files inhibiting the boot. This at least is discovered by a simple test that should be done anyway. Worse are the cases where the “clone” boots and seems OK and only closer look reveals one to five missing directories and files.

It seems dd just is not an imager: dd is trying too cleverly to consider partitions files and rights.

In all such bad cases where dd failed – sometimes in a quite perfidious way – and with the same cards and devices we never had problems with Win32DiskImager on Windows (7 professional).

Part I's results

We can install headless / GUI-less Raspberries.

By having provided all basic tools and communication we can administer Pies with some comfort remotely from both Ubuntu and Windows workstations and laptops. And we can save and restore the current state of our ”little servers”, sometimes a bit spoiled by the “size problem”.

Comparing Raspian lite=GUI-less with with the GUI variant we see with the latter

- ▼ 39 more services/processes running immediately after reboot
- ▼ 2..5 s delay when typing on putty after a short pause.
  On a GUI-less/lite we always saw an immediate reaction as we type.

And in Part II we see:

- ▼ The GUI variant showed just slightly worse latency results (sometimes max. latency above 200µs compared to always below 200µs on lite).

So far good news, but

- ▼ loads (ping etc.) having no measurable effect on lite significantly increase the latency on the GUI variant.
  And the latency literally “explodes” by just moving the mouse.

Hence we must state the GUI making the Raspberry/Raspbian unsuitable for embedded/realtime/ server work. This resembles our experiences in the large with “real” servers ([29]).

P A R T II

Using the GPIOs

Our first goal is to use Raspberry's GPIO pins in an own programme, preferably written in C. And in any case, we want them cross-compiled (-made, -build) from comfortable workstations, usually with Windows. For a minimal proof of concept we start with binary output using a small assembly with two LEDs, which can be driven by the Raspberry's µP directly. See figure 4. For more IO and to connect a logic analyser etc. we'll use a breakout board, instead; see figure 5 on page 30.

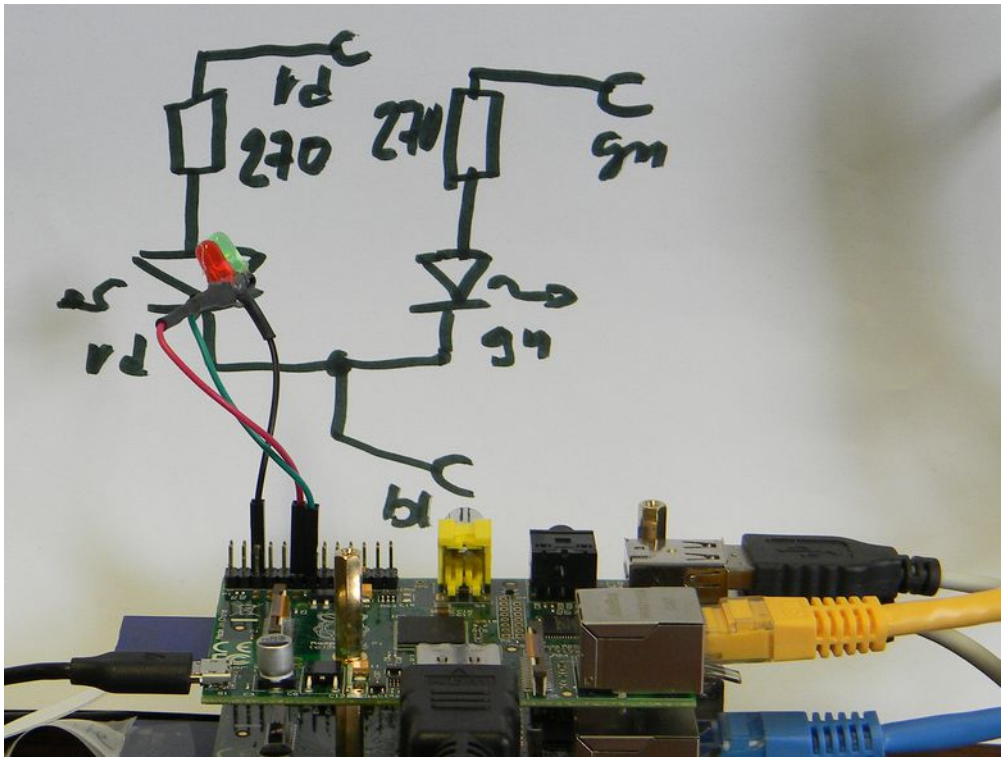

Fig. 4: Minimal IO

Anyway, first of all we would need a library to use Raspberry Pi1's and Pi3's IO pins. WiringPi is a well known one and used by piXtend (we bought for ~200€). So, let's' start with wiringPi.
In the end we will not favour wiringPi and switch to pigpoi[d] for all applications.
And, in the end, for real time process IO applications, we take Pi3's, contrary to figure 4.

The following recipe gets git and wiringPi, builds the latter and makes it known to the C compiler:

```
sudo apt-get update
sudo apt-get install git-core
git clone git://git.drogon.net/wiringPi
cd wiringPi         # you may keep this directory for reference
./build
```

Now we make a directory ~/bin which will be on our path ($PATH) after next re-boot, a work directory ~/progWork and an empty C source file.

```
mkdir ~/bin
mkdir ~/progWork
cd ~/progWork
touch rdGnBlinkBlink.c
```

To work on the C source rdGnBlinkBlink.c and other text files we best use FileZilla and editpad on our Windows PC. In the end we'll use Eclipse and cross-build by make on the PC.

Hint, warning: When having finished and stored the .c source in editpad, FileZilla will ask if it should propagate the changes. Do not forget to to make FileZilla do so.

```
// A first programme for Raspberry's GPIO pins
// Rev. 0.7 17.05.2017 Copyright (c) Albrecht Weinert
// weinert-automation.de                a-weinert.de
// It uses two pins as output assuming two LEDs connected to as H=on
// wiringPI[0] (GPIO17/17): red
// wiringPI[2] (GPIO21/27): green

// This program forces application singleton and may be used as service

// Compile on Pi by:  g++ rdGnBlinkBlink.c  -o rdGnBlinkBlink  -lwiringPi

#include <wiringPi.h> // pinMode, digitalWrite
#include <stdio.h>    // perror
#include <unistd.h>   // close
#include <fcntl.h>    // O_RDWR
#include <sys/file.h> // flock
#include <signal.h>   // signal SIGTERM
#include <stdlib.h>   // atexit

int lockFd;
// The following file must exist for this programme to start work
// Make the lock file by: touch /home/pi/bin/.lockRdGrBlinkBlink

char const  * const fileSpec = "/home/pi/bin/.lockRdGrBlinkBlink";

// So, deleting this file inhibits the start even by cron etc.

static void onSign(int s){
  if (s == SIGINT)  exit(0); // cntl C terminates normally
  exit(s);
} // onSign(int)

static void onExit(void){
  pinMode(0, INPUT); // release red LED pin
  pinMode(2, INPUT); // green LED pin
  flock(lockFd, LOCK_UN);
  close(lockFd);
} // onExit()

int main(){
  if ((lockFd = open(fileSpec, O_RDWR, 0666))  < 0) {
    perror("can't open lock file  (must exist)");
    return 97;
  } // can't open lock file (must exist)

  if (flock(lockFd, LOCK_EX | LOCK_NB) < 0) {
    perror("can't lock lock file (other instance running)");
    close (lockFd);
    return 98;
  } // can't lock lock file

  if (wiringPiSetup() == -1) { // initialise wiringPi (this is essentiel)
    perror("can't initialise IO handling (wiringPi)");
    flock(lockFd, LOCK_UN);
    close(lockFd);
```

```
    return 99;
  } // can't initialise wiringPi

  atexit(onExit); // register exit hook
  signal(SIGTERM, onSign); // signal hook
  signal(SIGABRT, onSign);
  signal(SIGINT,  onSign);
  signal(SIGQUIT, onSign);

  pinMode(0, OUTPUT); // red LED pin as output
  pinMode(2, OUTPUT); // green output

  while(1) {            // red  green
    digitalWrite(0, 1); // on
    delay(200);         //           200 ms red
    digitalWrite(2, 1); //      on
    delay(100);         //           100 ms both
    digitalWrite(0, 0); // off
    delay(100);         //           100 ms green
    digitalWrite(2, 0); //      off
    delay(200);         //           200 ms dark
  } // while  endless loop          600 ms sum
} // main()
```

Listing 2: The first process IO programme to start with.

After finishing the work on the .c source (Listing 2) translate it:

```
cd ~/progWork   # already there
g++ rdGnBlinkBlink.c -o rdGnBlinkBlink  -lwiringPi
ls -l
```

```
-rwxr-xr-x 1 pi pi 6816 Mai 13 12:37 rdGnBlinkBlink
-rw-r--r-- 1 pi pi  486 Mai 13 12:36 rdGnBlinkBlink.c
```

We see the source and the ready to run (executable) programme. Run by:

```
touch /home/pi/bin/.lockRdGrBlinkBlink   # make lock file
./rdGnBlinkBlink
```

The first line is necessary only once, as the programme won't start when its lock file doesn't exist.

The ./ is necessary as our working directory is not on the path for executables ($PATH). Copying it to a directory on the path will allow the start without extra ado from anywhere.

```
cp rdGnBlinkBlink ~/bin/    # do this as to have it on the PATH
```

Now it may be time to explain why our first output programme seems so long and may look complicated – it's neither. A minimal “hello output” with exactly the same 600 ms loop, two LEDs blinking would be 16 non comment lines easily excerpted from listing 2, see listing 3 (page 18).

But according to this paper's title our requirements are a bit higher.

As the germ cell of of a full grown 24/7 process control / IO application we at least want it to

- clean up and leave a specified, controlled output state when finishing or being killed and
- prevent more than one instance of such control programme running (singleton).

Comparing listing 3 and 2 shows the “process control” part implementing the external behaviour being exactly the same. The complication by those two – minimal by the way – requirements will

not grow substantially when getting to a full grown process control programme and will partly be put in extra sources and includes respectively own libraries.

But even for this small common part it is evident (and easily demonstrated with listing 3) that running two instances of the programme would spoil the timing behaviour on the process outputs. These multiple starts happen quite easily, often by implementing automatic starts (boot, cron, etc.) forgetting one already has one. Additionally well meaning users tend to start control applications without noticing their state.

Process control applications in almost all cases must not run in more than one instance and should enforce this by themselves. To be more precise: No more than one application at a time must access process related IO.

The Unix style solution in listing 2 is to use a fixed lock file that has to exist. Before entering any process control part including its initialisation, it is tried to lock that file. If it can't be locked or if it does not exist the programme terminates. As a welcome side effect we can delete the lock file to prevent all future starts – those by hand as well as the automatic ones.

Of course, the lock file must be unlocked when programme ends – no matter why or how the programme was terminated or killed. Hence, we best implement a clean up and put the unlock there.

This clean up, we need anyway – so it's no extra complication for the unlock. When controlling process outputs it is essential to bring them in a specified state when the programme ends, no matter how. In listing 2 this is done by catching the relevant signals (interrupts) as well as the the programme end and putting the clean-up in the registered hooks. In our case we release (and de-energise) our outputs, which most often is the adequate (default) procedure.

```
// A first simple programme for Raspberry's GPIO pins
// Rev. 0.0 17.05.2017 Copyright (c) Albrecht Weinert
// This is the simple (non process control) version of rdgnBlinkBlink
// see the comments there.
// compile by:  g++ rdGnSimpleBlink.c -o rdGnSimpleBlink  -lwiringPi

#include <wiringPi.h> // pinMode, digitalWrite

int main(){
  if (wiringPiSetup() == -1) return 99;
  pinMode(0, OUTPUT); // red LED pin as output
  pinMode(2, OUTPUT); // green output
  while(1) {            // red green
    digitalWrite(0, 1); // on
    delay(200);         //          200 ms red
    digitalWrite(2, 1); //      on
    delay(100);         //          100 ms both
    digitalWrite(0, 0); // off
    delay(100);         //          100 ms green
    digitalWrite(2, 0); //      off
    delay(200);         //          200 ms dark
  } // while endless               600 ms loop
} // main()
```

Listing 3: The simplified (non process IO) programme – not to use just to play with.

An application as service

When using the Raspberry as server – or device – we usually want one or more applications (process control, web server, helper etc.) start automatically when powering up without an user to have to login and give commands.

rc.local

One way is putting the starting command at the end of `/etc/rc.local`.

For our process IO example (listing 2) put

```
/home/pi/bin/rdGnBlinkBlink &
```

at the end (before the last line exit 0). Our small programme will be started at the end of the boot process. Test it by reboot. For long running tasks or, as in rdGnBlinkBlink's case, endless ones do not forget the & at the to put the programme in the background and go on with the script or shell. Otherwise the rc.local script would hang.

To stop a programme started this way as extra process a logged in user with administrative privileges must determine the process ID number and use it in a kill command:

```
ps aux | grep rdGnBlinkBlink
sudo kill TheProcessNumberProvided
```

The process ID number is the second word in each line given by ps aux, hence lookup by:

```
ps aux | grep rdGnBlinkBlink | awk '{print $2}'
```

If valiant enough replace the number lookup by direct kill using this:

```
ps aux | grep rdGnBlinkBlink | awk '{print $2}' | xargs kill
```

or (if available)

```
sudo killall rdGnBlinkBlink   # also works on most raspbians
```

cron

The cron service knows an event “@reboot”. Say

```
crontab -e      # Note: when adding a programme requiring sudo do:  sudo crontab -e
```

to (phoney) edit cron's time-table and add the line

```
@reboot /home/pi/bin/rdGnBlinkBlink
```

This works without & at the end. Remove the command from /etc/rc.local and reboot as test.

```
ps aux | grep rdGnBlinkBlink
```

reveals

```
pi      526           0.0  0.0   1912   388 ?        Ss    16:53   0:00
/bin/sh -c /home/pi/bin/rdGnBlinkBlink

pi      528           0.0  0.1   4008  1696 ?         S    16:53   0:00
/home/pi/bin/rdGnBlinkBlink

pi      818           0.0  0.1   4776  1916 pts/0     S+   17:16   0:00
grep --color=auto rdGnBlinkBlink
```

Besides grep itself we see two processes: a shell which cron used to start our programme (probably the cost of allowing no &) the and the programme itself. Our experiments showed killing the shell (526 in above listing) would leave the programme (528) running.

Hint: Above “mass murderer” command (ps aux | grep … kill) would kill both.

The cron service is probably running on every Linux by default. But cron logging may not always be enabled. If something is doubtful or wrong when using cron the first look would go to its log file, /var/log/cron.log by default. You might wish to enable cron logging by:

```
sudo nano /etc/rsyslog.conf
```

and put in or uncomment this line (found under # rules #):

```
cron.*                          /var/log/cron.log
```

In our above example cron would log e.g.

```
May 26 08:53:49 rasp67 CRON[499]: pi CMD (/home/pi/progWork/rdGnBlinkBlink)
```

It shows our programme's start time and command, but only the shell's pId.

Mimic a service – start stop restart enable disable

With a bash script (listing 4) rdGnBlinkCntl we control our exemplary “process control” programme (listing 2, ending page 17). Make the script by

```
cd ~/bin
touch rdGnBlinkCntl
chmod 755 rdGnBlinkCntl
```

and work on it via FileZilla and editpad.exe. Do not forget to set the Unix (LF only) option.

```
#!/bin/bash
showRdGnBlinkCntlVers () { echo "
#  /usr/local/bin/rdGnBlinkCntl ~resp. ~/bin/rdGnBlinkCntl
#  control the rdGnBlinkBlink service      V.01 22.05.2017
#  (c) 2017  Albrecht Weinert                 a-weinert.de
"; }

rdGnBlinkCntlHelp () { echo "
#  call by: rdGnBlinkCntl command
#  commands are:
#    start | stop: start or stop rdGnBlinkBlink
#    restart:      stop and then start
#    disable:      inhibit the (next) start
#    enable:       allow the service to be started
#    version:      show version info
"; }

if [ "X--help" == "X$1" -o "X" == "X$1" ]; then
  showRdGnBlinkCntlVers
  rdGnBlinkCntlHelp
  exit 0
fi

if [ "--version" == "$1" -o "version" == "$1" ]; then
  showRdGnBlinkCntlVers
  exit 0
fi

progPath=/home/pi/bin/rdGnBlinkBlink
lockPath=/home/pi/bin/.lockRdGrBlinkBlink
searchPt=rdGnBlinkBlink
```

```
stop () {
  ps aux | grep ${searchPt} | awk '{print $2}' | xargs kill
}

start() {
  ${progPath} &
  pid=$!
  sleep .3
  if ps -p $pid > /dev/null; then
    echo "rdGnBlinkBlink started with PId ${pid}"
    exit 0
  fi
  exit 99
}

if [ "start" == "$1" ]; then start; fi
if [ "stop" == "$1" ]; then
 stop
 exit 0
fi

if [ "restart" == "$1" ]; then
 stop
 start
fi

if [ "enable" == "$1" ]; then
  if [ -f $lockPath ]; then
    echo "rdGnBlinkBlink was enabled, already."
    exit 0
  fi
  touch $lockPath
  exit $?
fi

if [ "disable" == "$1" ]; then
  if [ -f $lockPath ]; then
    rm $lockPath
    exit $?
  fi
  echo "rdGnBlinkBlink was disabled, already."
  exit 0
fi

rdGnBlinkCntlHelp
```

Listing 4: Script rdGnBlinkCntl to control rdGnBlinkBlink (listing 2) as service.

For testing you may use:

```
gpio readall
rdGnBlinkCntl start
gpio readall
rdGnBlinkCntl stop
gpio readall
```

```
 | BCM | wPi |  Name   | ModeStart | Val | ModeStop | V |  Physical | ..
 |  17 |  0  |  wPi 0  |    OUT    | rd  |     IN   | 0 |  11 || 12 | ..
 |  27 |  2  |  wPi 2  |    OUT    | gn  |     IN   | 0 |  13 || 14 | ..
```

When rdGnBlinkBlink is running you should see GPIO 17&27 as OUT and when stopped as IN.

Cross-compile C for Raspberry from a powerful workstation

Now we have our first GPIO C example (rdGnBlinkBlink, listing 2, page 17), also implemented as service (listing 4) not counting the play variant (rdGnSimpleBlink, listing 3).
This is a base to go further to useful and greater projects.

But now its also time to pause and re-think tooling and deployment:

- Very few of us like to develop software on a GUI-less Raspberry with local tools and stone age editors (nano being the least evil of them).
- Or when adding the GUI, only few would like to handle IDEs, version control, documentation generators and all else on a Raspberry.
- At least this will be true for those of us using all this and more in all comfort and speed on their powerful Windows (or Ubuntu) workstations.

The pre-condition to develop C software for Raspberries on Windows is being able to cross-compile and cross-link; all else will in the end build upon this. To get this going we download

```
23.05.2017  17:06       773.928.611      raspberry-gcc4.9.2-r4.exe
```

from http://gnutoolchains.com/raspberry/. Run it to

- install at C:\util\WinRaspi ....● usable for all
- make no links for duplicate files
  (we're on Windows, links exist and do work there, but no one will expect seeing one)
- add `C:\util\WinRaspi\bin\` to the path or let installer do it
  (In this case it makes sense, even if no one likes extending the PATH by every install.)

To test this best make an empty directory,

- get our .c source (rdGnBlinkBlink.c, listing 2) there via FilleZilla and
- cd there.

Then command

```
arm-linux-gnueabihf-g++.exe  rdGnBlinkBlink.c  -o rdGnBlinkBlink
  -lwiringPi
```

You'll get the compiled and linked runnable rdGnBlinkBlink in no time.
Transfer it to the Raspberry, best to another directory, cd there and

```
chmod 755 rdGnBlinkBlink
rdGnBlinkBlik &
```

It works! A programme made on a Windows PC does GPIO on a Raspberry.

`arm-linux-gnueabihf-g++` is the equivalent to plain g++ we used on the Raspberries.

Note on the prefix arm-linux-gnueabihf-:
Please do not get upset on the long name or its reasonableness – “Glühabhilfe?”. These prefixes are used for cross tool-chains: They are unique prefixes for the target hardware and target specific library sets. When setting up a new cross compile project in your GCC enabled Eclipse (so far perhaps used for AVR project as in our case) you will be asked for a tool path and a tool prefix; Here it would be: `C:\util\WinRaspi\bin` and “`arm-linux-gnueabihf-`”.
For the AVR- amily it was just short and intuitively “`avr-`” (and `C:\util\WinAVR\bin`). On the other hand, some special compiler and linker options where required to fit the target MCU, board type, frequency etc.

Note on the g++ or gcc choice:

Both gcc and g++ are GNU compilers respectively tool-chain drivers, doing almost the same. g++ treats .c files as C++ source while gcc expects and handles plain C. In the case of our IO example, listing 2, both work and both produce an executable of almost the same length and content.

To round up our (Windows) cross-compile tools chain we should also have a make file understanding at least make clean and make all. We can do it now or postpone it to after having Eclipse working for Raspberry GCC projects, see chapter Eclipse – make project.

A note on Windows:

Evidently, we prefere Windows on powerful development workstations – with Java, Eclipse, OpenOffice, GNU tools, SVN etc.. We have all liberty of those open XY tools, while enjoying Windows' comfort and professionalism: domain and network file system integration (without samba fumbling), decent powerful explorer with tool integration e.g. for SVN (tortoise), decent text editors (editpad), common clipboard support and so on. And (almost) all just works fine.

With Ubuntu more than once dragging files to shells suddenly stopped working or changed its behaviour. (Stupid user must learn to enter long path names.) Unclear, pure text or no clipboard support is a good recipe to drive Linux users mad. We experienced regular total crashes on upgrades (more with Mint than with pure Ubuntu). This and more constantly chases one back to Windows.

Well this happiness with Windows plus open tools hold up to Windows 7 respectively Windows Server 2008 R2 enterprise. Windows 10 changed this: Not regarding questions of taste and usability or HMI continuity, its unreliable and the less controllable updates / upgrades do render installed tools inoperable at “Mint rate”.

On the other hand, almost all what we describe for Windows can be (and was here) done on a powerful Ubuntu workstation, too.
And yes: Most Windows (even <10) have their "drive nuts" potential, for example faking directory and file names, when accessing the file system graphically.

Eclipse – step zero

Now (cross-) developing on a powerful workstation, we want, of course, the comfort of a powerful IDE. Our choice is Eclipse, used since years for Java projects, Web, AVR C and much more.

Put your sources, listing 2 & 3 in our examples, in an Eclipse's project folder. Best make the folder and the copies before making the new cross-C project.
As said above, set tool path and a tool prefix to:

C:\util\WinRaspi\bin and arm-linux-gnueabihf- and

… enjoy Eclipse's support and comfort.

Well, a little work on make files and project setting will be unavoidable.

A reliable source of trouble are Eclipse's automatically generated make files, which on first set-up notoriously fail. Before stepping into (great!) configuration trouble to get this automated make working, better drop the automatically generated make files and make an own makefile which Eclipse would use with targets all and clean.

Thereby you get

- **+** immense flexibility considering targets, devices and else which in the end will often be needed,
- **+** making automatically by scripts or by just running make by command line without starting or even needing the IDE
- ▼ involved with the ill syntax and semantic of the make tool.

So we make the appropriate changes in project → settings C/C++ → build and add a makefile to the project. And while we're at it we put the exemplary project in a SVN repository.

Eclipse – make project

See at the browser or get (by `svn checkout https://...`) all makefiles, sources etc. at

https://weinert-automation.de/svn/rasProject_01/

You'll need to login (guest:guest). In this paper ([31]) there will be no listing for

```
26.05.2017  15:27         9.521   makefile
26.05.2017  15:27         1.163   make_raspberry_01_settings.mk
26.05.2017  15:27         1.165   make_raspberry_02_settings.mk
26.05.2017  15:27         1.116   make_raspberry_03_settings.mk
29.05.2017  18:37           770   progTransWin   ..... and some more
```

The makefile and its includes do work in our Eclipse C make project via project build and clean.

As postulated above you can use the makefile with more functionality and in automated batch files directly (without Eclipse). Be in the project/source directory, say D:\eclipCeWsOx\rasProject_01:

```
make help
make help_comm
make clean  all
make PROGRAM=rdGnSimpleBlink  clean  all
```

The last two commands generate our two Raspberry IO programmes rdGnBlinkBlink (PROGRAM default setting) and rdGnSimpleBlink (listings 2 and 3). And by

```
winscp.com /script=progTransWin /parameter pi:raspberry
  192.168.89.67 bin rdGnBlinkBlink

winscp.com /script=progTransWin /parameter pi:raspberry
  192.168.89.67 bin rdGnSimpleBlink
```

we transfer them to /home/pi/bin of our Raspberry 192.168.89.67 with the parametrised WinSCP script. This is integrated in the Makefile and can (on Windows, only) also be done by:

```
make PROGRAM=rdGnBlinkBlink  clean  progapp
make PROGRAM=rdGnSimpleBlink clean  progapp
```

Eclipse – troubles and hints

SVN client chaos Where there is light, there is shadow

One trouble with Eclipse, is putting in a decent SVN client. SubClipse made the least trouble so we stayed with it. Remains the problem of different client versions having different structures / versions of local working copies. You may well have two or more SVN clients on your workstation: command line, TortoiseSVN (explorer plug-in), SubClipse (eclipse plug in) etc. If the SVN client's working copy versions differ, your in trouble as virtually no client offers backward compatibility. Tips:

- Try to keep the number low (<=3). At least the three clients named share settings.
- Do not try to stay with old working copy versions too long. Have a strategy when and how to update all your clients and to upgrade all local working copies (with batch files).

While the SVN client chaos is an inconvenience, as not seeing the versions in Eclipse or not having compare with revision, the next point renders an IDE useless.

Eclipse marking non-existent errors

This problem hits mainly, but not only, cross-compile, cross-build projects:
While “make all” directly on the shell or indirectly in Eclipse by “build project” goes with zero errors and warnings – and yields an usable result – Eclipse marks the sources with non-existent errors and warnings. This gives lots of red and yellow where none should be; and “open declaration”, language searches and else don't give correct or all results. Hence, a useless IDE.

With CDT (C/C++ plug in) Eclipse uses own partly buggy compilers and indexers for judging sources. The “real” make uses the ”real” target compilers etc., those one has anyway to live with. Googling reveals a lot of in-official remedies concerning configuration and partly source code, some of them useless or harmful and some also sheer voodoo.

One tip is to check Eclipse's the include path configuration, even if it was correct yesterday. It has to “include” all local includes (those of -I./include e.g.) and all implicit includes the, target tool chain has. If those are absent errors are explicable. Even after repairing omissions errors of this category (uint8_t not defined) may remain. Here the root cause is CDT/indexer being unable to handle indirect and conditional includes correctly. This bug is less or more present on all Eclipse versions. This is hardly believable. It seems Eclipse CDT tries to do something locally peepholing with help of indexers, which will work well with Java but can't work on languages with source text includes and complicated macro expansions.

This include handling problem may (hopefully) be worked around by mucking up (one may also say spoiling) the sources a bit, as already seen in this old AVR (not raspi) example:

```
#include "arch/config.h"  // for sake of Eclipse (4.2.x)
#include "we-aut_sys/ll_system.h"
#include "pt/pt.h" // Eclipse 4.2.x; can't handle nested includes
#include <avr/io.h> // for sake of Eclipse (4.7; can't even less includes)
#include <stdint.h> // for sake of Eclipse (4.7; can't even less includes)
#include "we-aut_sys/network.h" // for sake of Eclipse (4.7)
#include "we-aut_sys/smc2fs.h" // for sake of Eclipse (4.7 can't either)
#include "uip/uip.h" // for sake of Eclipse (4.7 can't either)
#include "uip/uip_arp.h" // for sake of Eclipse (4.7 can't either)
#include "we-aut_sys/syst_threads.h" // for sake of Eclipse (4.7)
```

Problem remains that those extra unconditional includes may spoil the real build (correct before) under certain conditions, besides the aesthetic damage.

Sometimes unexplainable errors remain or can't be voodooed away without collateral damage. This was the case in our Raspberry C projects when growing beyond our introductory examples. Here updating to the newest Eclipse CDT IDE – Oxygen, 4.7.0, June 2017 – was the rescue. Adding SubClipse 1.10.13 and Web-tools (for some .xml, .css for doxygen e.g.) was no problem and brought the projects back to no errors/warnings/red/yellow.

But before getting too enthusiastic: We also have AVR-projects (see example above) in the workspace co-existing under Mars. Updating to Oxygen (adding the AVR plug-in) made things worse. We suddenly had hundreds of errors (quasi all red). Seven of the nine extra includes in the AVR sources shown above came with Oxygen and put the errors down from hundreds to dozens in one AVR project. But beyond that and on other projects all tricks were of no avail. Repairing this wrong errors – this false positives – is stabbing in the dark.
We recommend not to use Oxygen for AVR projects especially not to update the AVR project IDE from a working Eclipse (Mars e.g.) version.

This "stabbing in dark" or "voodooing" seems symptomatic for this whole “false positive” business. CDT's internal compiler/indexer configuration seems a mess of automatic heuristics and unclear – often inhibited or vanished – configuration, complicated by this obstructive “active” / “debug” settings. When fully hit by false positives, there's usually no understood cause and no reliable remedy – but a lot of always recurring bugs to work on for the CDT developers.

Note 1: To emphasize the last point, projects used successfully for years got a useless IDE by justupgrading Eclipse.

Note 2: An IDE for C or any other language marking false errors and warning is worse than none. When marking errors or warnings correctness and consistency with the target tool chain has the absolute top priority. Speed – wrong answers fast – is secondary.

At the moment: Use Oxygen for Raspberry and stay with proven older Eclipse for AVR.

Starting with GPIO – a look at wiringPi, bcm2835 and a derivative

Now we can

- cross-compile/cross-develop with
- GNU-tools,
- Eclipse,
- make and
- SVN
- on Windows

for our Raspberries and even

- upload the programme just build

from Windows to the Raspberry automated by make.

Hence, now, as we have all comfort, it's time to have a closer look at the wiringPi – the GPIO library used first – and at libbcm2835 used and demonstrated by the programme rdGnBcm2845Blink also in https://weinert-automation.de/svn/rasProject_01/
Please look there (login guest:guest).

Install the library on the Raspberry by

```
wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.50.tar.gz
tar xvfz bcm2835-1.50.tar.gz
cd bcm2835-1.50/
dir
./configure
make
sudo make install
find / -name "libbcm2835.a"
```

To test the just installed library locally on the Raspberry translate and run the lib's examples/blink

```
cd ~/progWork
cp /home/pi/bcm2835-1.50/examples/blink/blink.c  ./
g++ blink.c -o blink  -lbcm2835

./blink
^C
```

This little programme blink In will flash the red LED on our test harness (figure 4, page 15). If this works so far, you got an alternative GPIO library in addition to wiringPi.

But we want to use it on our (Windows) workstation and projects, too. Therefore we copy two of the library's files

```
02.06.2017  11:47            80.726    bcm2835.h
02.06.2017  13:03            63.756    libbcm2835.a
```

to our workstation. The .h file goes to the tool chain's include directory

```
C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\include\bcm2835.h
```

The '.a' file – the compiled library – we will add to our Raspberry cross tool-chain, as

```
C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib\libbcm2835.a
```

At this stage you should be able to translate a little example by command line:

```
arm-linux-gnueabihf-g++.exe blink7.c -o blink7  -lbcm2835
del blink7 ; this won't run on Windows
```

The projects example rdGnBlinkBlink (listing 2, based on wiringPi) has been ported as rdGnBcm2835Blink to libbcm2835 and is fully supported by the makefile – including the programme transfer to the Raspberry. See the SVN project for details.

The bcm2835 library is a bit heavyweight compared to its features and suffers from its initialisation ritual. We tried to improve, and reduce this library to a minimum. While the improvement was promising, we could not reduce respectively get rid of the initialisation part. The GPIO usage initialisation is a bit tricky and described more by tradition as by specification. In the end we gave up this approach as well as using this library as is.

So far we considered the Raspberry IO libraries

- wiringPi
- bcm2835
- pigpio (see next chapter).

And we used and tried them in small process IO applications – forcing singleton, being usable as service etc. – like seen and discussed with listing 2 and 4.

Get all those examples from the SVN repository mentioned above on page 24.

wiringPi

wiringPi is well known and widely used.

It tries to cover the range of Raspberry Pi1, 2 and 3 with its diverse variants as well as some alternate Pi lookalikes. Additionally wiringPi covers a wide selection of extension boards or so called shields considered popular.

Probably as consequence of this “cover all including shields” approach wiringPi introduces indirection respectively abstraction layers away from from the µP's (BCM2835, BCM2836, BCM2837) GPIO numbers or (virtualised) IO register addresses ([56]). With wiringPi's indirection one can directly refer to the 28 respectively 40 pin header numbering or an own special wiringPi numbering scheme. One example:

Pin header 28 / 40 Pi1 / most else	µP BCM2835 (Pi1)	µP BCM2837 / µP BCM2836 Pi3 / most else	wiringPi's own number
HW pin 13	GPIO 21	GPIO 27	2

This scheme seems unique; all other libraries utilised use the GPIO numbers that refer to the µP's “truth”, the pin's properties and available functionality etc. For a shield or extension occupying all 40/28 header pins in a fixed layout the HW pin number describes the extension's interface. wiringPi is eager to support that directly; all others may do the translation directly or better by some macros to get all flexibility for the targets to cover.

All this makes wiringPi quite big. On the other hand programmes for wiringPi are the smallest ones. They rely on finding their library ready (as .so which is a Linux kind of .dll):

```
lrwxrwxrwx 1 root staff      libwiringPiDev.so  ->  libwiringPiDev.so.2.44
-rwxr-xr-x 1 root staff   28420   2017-05-17 10:51  libwiringPiDev.so.2.44
lrwxrwxrwx 1 root staff         libwiringPi.so  ->  libwiringPi.so.2.44
-rwxr-xr-x 1 root staff   70288    2017-05-17 10:51  libwiringPi.so.2.44
```

Using wiringPi on a Raspberry requires it having been installed there. On the Windows workstation for cross-compiling you'll find those files with the tool-chain in e.g.:
`c:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib`

For further insight we put the sources or wiringPi and other IO libraries in an extra Eclipse (GCC make) project, not made public. The resume for libwiringPi:

- quite large 100 .c files and 52 .h (with examples)
- compiles (by Eclipse / make) only after some 20 corrections
  about 6 more corrections brought us to “no warnings”, too

- seems to adapt automatically to all types of Raspberries respectively BCM µPs
- relatively small executables (< 9kB for rdGnBlinkBlink, listing 2, page 17)
- needs library installation on every target Raspberry

bcm2835

The bcm2835 library, as the name suggests, just handles and refers to the µP's GPIO. As the name does not suggest, it is also usable for BCM2837 and the Pi3 (and others).
Note: Regrettably a lot of documentation seems to stop between 2012 and 2014 with the Pi1, leaving the rest to forums and speculation. As of this writing (July 2017) when buying Raspberries you would order Pi3s, wouldn't you.

Referring to just GPIO in the end plus the most important alternate functions makes bcm2835 comparably simple and small. The library is linked to the executable making those quite large compared to a wiringPi variant of equal functionality. On the other hand, an executable cross-made on Windows for Pi3 e.g. can be FTP-transferred to all Pi3s and run without having to install anything. To resume libbcm2835:

- relatively small 1 .c + 1 h. (w/o examples)
- compiles as downloaded with just two style warnings and no errors on Eclipse / workstation
- the library is written for Raspberry Pi1 / BCM2835, only
- this can be repaired (so far for Rapberry Pi3) by not using
  the lib's enums or marcos for GPIO numbers – but own macros or direct literals
- large executables compared to wiringPi (~57kB for rdGnBcm2835Blink)
- needs no library installation on a target Raspberry when transferring cross-builds

bcm2835 – improvements due (?)

So bcm2835 seems outdated seems to need

- stylistic improvements l
  - like typos comments but first of all
  - making indices out of all relative addresses
    to get rid of “y[x/4] respectively y + x/4 in production code
- could be made smaller by
  - separating special and alternate functionality as well as
  - by getting rid of wrapped system functions
    well defined by Linux (man page e.g.) and better used directly

As said, we went this way with a reduced and improved library and test examples , fulfilling all said requirements:

- better adapted to Raspberry Pi3, too,
- making the production code clearer and easier to read by
- being less error prone in usage
- reduces the runnable size of the little demo (by 42kB)

One may very well ask if this is worth the trouble – especially
- when concentrating on pigpio in the future (strongly recommended) and
- having to admit that the (naive) aspiration to get rid of the complex
  initialisation code to get the GPIO's ever changing virtual addresses failed.

And the answer is, respectively our answer was "No".

Hence, still utilising the lessons learned, we deleted this sub-project.
In the end we consider bcm2835 (and our offspring deleted) dead end street and do not recommended to use it.

Let's see – and use – pigpio[d] instead.

The pigpio library

All IO libraries considered so far are linked to / called by the production code, giving it the trouble to initialise the GPIO (memory) usage, adapt to changing (virtual) addresses as well as fighting with access rights. No decent OS offering any protection will user code letting do IO. Early approaches to do GPIO with Raspberry OSs required sudo. In between the standard GPIO usages (read write) can be made accessible without sudo, but more settings or alternate functions may not.

The pigpio library uses a different approach. It defines a server or daemon which does all initialisations and has control over all GPIO functions. It has to be started with sudo and may run forever in background. Programmes doing (process) IO do just communicate with the daemon by

- socket (example rdGnPiGpioDBlink) or
- pipe.

Both approaches need no sudo. In the case of socket the control programme and the GPIO pins may be on different Raspberries on the same network or one control programme can use multiple Raspberries' GPIO. A third way is

- linking the pigpio library to the programme (rdGnPiGpioBlink).

This is a bit contradictory to pigpio's philosophy, and this programme has to be started with sudo always. But the programme made and run so is also the said daemon/server and can be used remotely or locally by other programmes using above socket or pipe approach. And, not really surprising after all, pigpio forces its daemon's singleton property. A programme using the link approach won't start when the daemon (pigpiod) is running.

To get and install pigpio (see also [61]) do:

```
wget https://github.com/joan2937/pigpio/archive/master.zip
unzip master.zip
cd pigpio-master
make
sudo make install
```

Then do all the tests provided as recommended in [61] (better outcome with latest Jessie):

```
sudo ./x_pigpio # check C I/F
sudo pigpiod    # start daemon
./x_pigpiod_if2 # check C      I/F to daemon (all passed)
./x_pigpio.py   # check Python I/F to daemon (one or four fails)
./x_pigs        # check pigs   I/F to daemon (no or one fail)
./x_pipe        # check pipe   I/F to daemon (same result)
```

As said in [61] a few fail (ours documented in the #comment) with many more passed are OK.

Please find and see the working examples rdGnPiGpioBlink and rdGnPiGpioDBlink in the SVN repository mentioned above on page 24.

And when using pigpiod on a Raspberry it's recommendable to start it with boot. Best use:

```
sudo crontab -e # sudo here(!) when programme to add needs sudo
```

and add one line at the end:

```
@reboot  /usr/local/bin/pigpiod  -s 10
```

Starting pigpiod without parameters uses default settings: 5 µs sample rate, PCM clock, port 8888, both interfaces (socket, pipe) enabled; -s 10 would set 10 µs.; -p might change port.

Note again: You must put sudo before crontab -e when adding or editing a task requiring sudo. Do not prepend the task command with sudo. The crontab command suggests your editing a single system configuration file directly. That's an illusion. Every user + sudo seems to have own cron editor settings and files. Getting this (sudo) business wrong is the source of most “My crontab (@boot) setting isn't ..” complaints.

Note also: cron tasks might be started without having your environment and path settings. Hence, use the full path to the “real” executable (like /usr/local/bin/pigpiod e.g.).

A last note: When cross-compiling / cross-building it may happen (after first time using a feature) linker ending with not being able to satisfy externals. In this case the workstation tool chain may have other / outdated libraries.

```
23.06.2017  15:15           256.624   libpigpio.so
23.06.2017  15:15            65.128   libpigpiod_if.so
23.06.2017  15:15            75.624   libpigpiod_if2.so
```

Copy those files from the Raspberry's /usr/local/lib to your workstation's C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib (by ftp).

Besides solving the GPIO sudo hassle when using the daemon variant, Joan N.N.'s pigpio library has a lot of other rich and useful features, like PWM on every pin.

This library's socket approach bringing all good points is responsible for the hardest disadvantage: speed. A single (binary) write to a pin takes 98 µs. So making more than 5 IO calls in an 1 ms cycle becomes a no go. (Have a look at bulk / bank functions!)

The site offers no offline (.pdf) documentation but has very good on-line documentation (http://abyz.co.uk/rpi/pigpio/pdif2.html), the “much in one page approach” of which allows being .pdf-printed. This gives 60 pages, links partly working. (Or download all .html.)

Process IO hardware

In our introductory examples (rdGnBlinkBlink, listing 2, page 17) the process IO is directly connected to the the IO pins of Raspberry's µP. This might be feasible to a certain extend when all sensors and actuators are nearby in a closed encasement. Figure 5 shows a professional key matrix, LEDs and a piezo beeper as suitable example for directly attached process IO. The break-out board is just a test harness and helps to connect a logic analyser (black box on left).

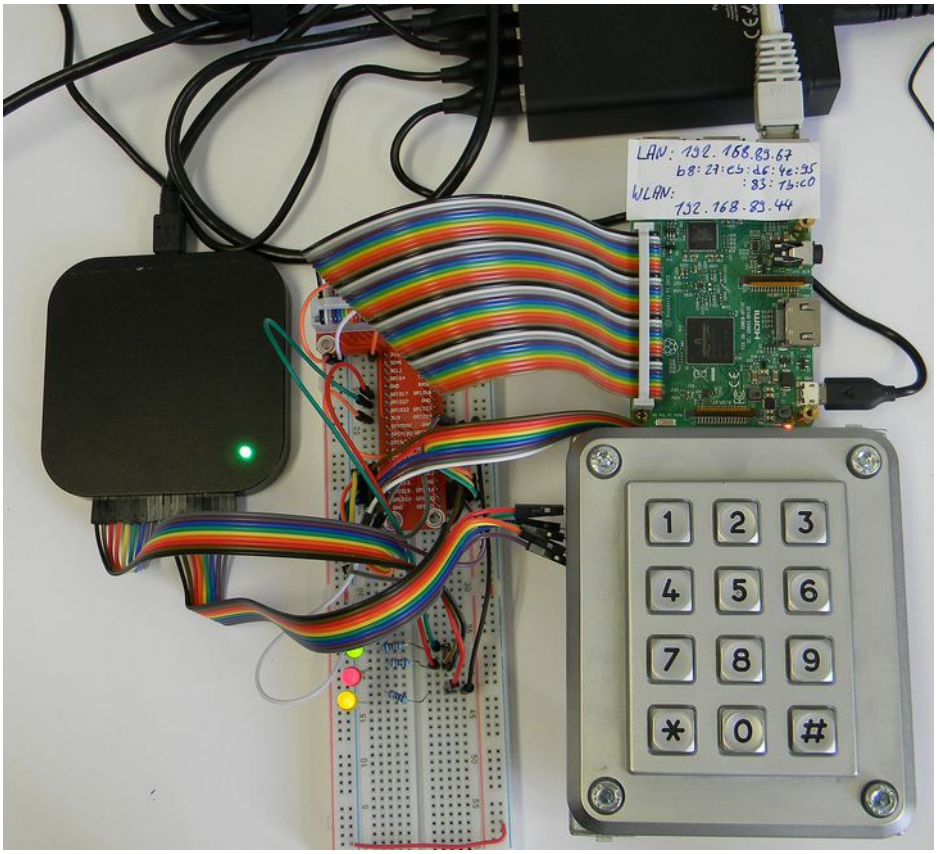

Fig. 5: Direct attached IO

In the production version the Pi is fixed to the back of the key matrix.

In all other cases and when controlling power beyond 48 mW, interface and protecting circuitry is mandatory. A lot of offers are found, often under the genus “shield”. Beware of useless or outright dangerous offers.

LEDs and buttons – direct IO

As soon as process IO signals go beyond the borders of a Raspberry, protective circuitry is mandatory. Low power LEDs, piezo speakers and buttons in the same well crafted protective encasement as the Raspberry are an acceptable exception, as may be the Raspberry camera.

A real use case had three LEDs and a 12 keys matrix EOZ Clavier S.series, 12 touches. Imagine the hardware setup as figure 5 without the breadboard and logic analyser in a metal box.

The key matrix was directly attached to 7 GPIO, see listing 6 for the concrete setup. Single or multiple key presses were determined by a common algorithm. Its seven steps according to the seven scan lines defined in the keyMatrix structure were put in the 1ms cycle.

```
#define ROW123  PIN37
#define ROW456  PIN35
#define ROW789  PIN31
#define ROWa0h  PIN33

#define COL147a PIN40
#define COL2580 PIN38
#define COL369h PIN36

#define NoCols  3 // number of columns
#define NoRows  4 // number of rows in key matrix

static keyMatrix thePad = {
    .noCols = NoCols,
    .noRows = NoRows,
    .colRow = {COL147a, COL2580, COL369h, // noCols + noRows,
                                ROW123, ROW456, ROW789, ROWa0h},
    .keyVal= { // [ rowInd * noCols + colInd ]
        '1', '2', '3',
        '4', '5', '6',
        '7', '8', '9',
        '*', '0', '#' },
}; //  thePad
```

Listing 6: Defines and structure for exemplary12 keys matrix EOZ Clavier S.series

The device's IO was controlled by a next tier system via Modbus (see below). This general approach of using Raspberries as TCP/IP attached remote subsystems calls for a PoE solution.

LEDs directly driven by GPIO pins, of course, need a resistor of about 270 Ω and the pad drive strenght can be reduced down to 8 mA.

Speakers and beepers

Piezo speakers should get a series resistor of e.g. 47 Ω, too. The drive strength may be set to 2 mA. Without resistor, one may observe spikes when applying voltage to the speaker, generated by its mechanical (resonance) vibrations. Magnetic speakers must be 200 Ω, be it alone or with series resistors, when directly attached to one or two (push-pull) output pins.

For generating a tone one may use the gpio library's (see chapter The pigpio library, page 29) ability of “PWM at every pin, by setting the desired frequency and turning the tone on by PW=50 % and off by 0 %:

```
#define PIEPS   PIN12

   set_PWM_frequency(thePi, PIEPS, 400);
   set_PWM_dutycycle(thePi, PIEPS, 128); // PW 50% → tone
   :::::
   set_PWM_dutycycle(thePi, PIEPS, 0); // PW 0% → silent
```

But, at least in one case, we did observe disturbed input on another pin as soon as using this library's set_PWM_... functions. Assuming no library bug the cause for these (serious) disturbances, impeding the programmes proper functionality, is unclear yet.

On the other hand, when not wanting to play music, but only giving little short signal beeps, one may produce good 500, 400 or (see example) 200 Hz beeps as little extra task in an 1 ms cycle thread, organising the matrix scan over a 10 ms period in 7 steps as main task:

```
   while(commonRun) {
      waitKey1ms(3);  // 3ms end scan
      gpio_write(thePi, PIEPS, beep);
      crScanStep(&thePad); // 0
      waitKey1ms(1);   // 1ms scan
      crScanStep(&thePad); // 1
      waitKey1ms(1);   // 1ms scan
      gpio_write(thePi, PIEPS, 0);
      crScanStep(&thePad); // 2
      waitKey1ms(1);   // 1ms scan
      crScanStep(&thePad); // 3
      waitKey1ms(1);   // 1ms scan
      crScanStep(&thePad); // 4
      waitKey1ms(1);   // 1ms scan
      gpio_write(thePi, PIEPS, beep);
     :::::
```

As you assumed, the (uint8_t) variable beep is set to 1 when a tone is wanted and to 0 for off.

Using speakers directly controlled by GPIO brings two problems

- generating the tone frequency by software either directly or by hardware or library PWM may eat resources or may bring problems with seldom used and hardly tested library functions,
- depending on the surrounding and the speaker the tones may be just or hardly audible.

The last point may be healed by amplification instead of using a GPIO as power source – in simple cases one n-channel MosFET alone may do the job.

Piezo buzzers, on the other hand, contain a fixed frequency generator circuitry and a piezo speaker of fitting resonance. With low electrical power – as deliverable from a GPIO – they can be quite loud. This solves both problems. On the other hand one has the fixed frequency, usually in the 2..3 kHz range. Nevertheless most type can very well by "on-off-modulated" up to about 600 Hz, allowing some distinguishable sound effects.

Relays

You hardly find 3V relays with >= 200 Ω coil resistance. And even if so

- the contact load of those very low power relays to control a three phase motor switch or three pole contactor and
- you need protective (diode) circuitry for Raspberry GPIO forbidding simple direct attachment in the end.

For switching some "real" power by GPIO one has to use either

- solid state relays or
- use transistor circuitry to switch relay coils.

The latter solution come quite handy in professional modules, like in figure 6. It shows a module with 8 5 V relays, each well controllable by Raspberry's GPIO. The separate 5 V supply for the relays may come from a separate source but quite handy also from Raspberry's 5 V power supply including PoE. So in the end one has 11 short female / female pin to pin connections: Gnd, 3.3 V, 5 V and up to eight GPIOs between the Raspberry's 40 pin connector and the relay module (omitting, of course, the experimental break out board used in figure 6).

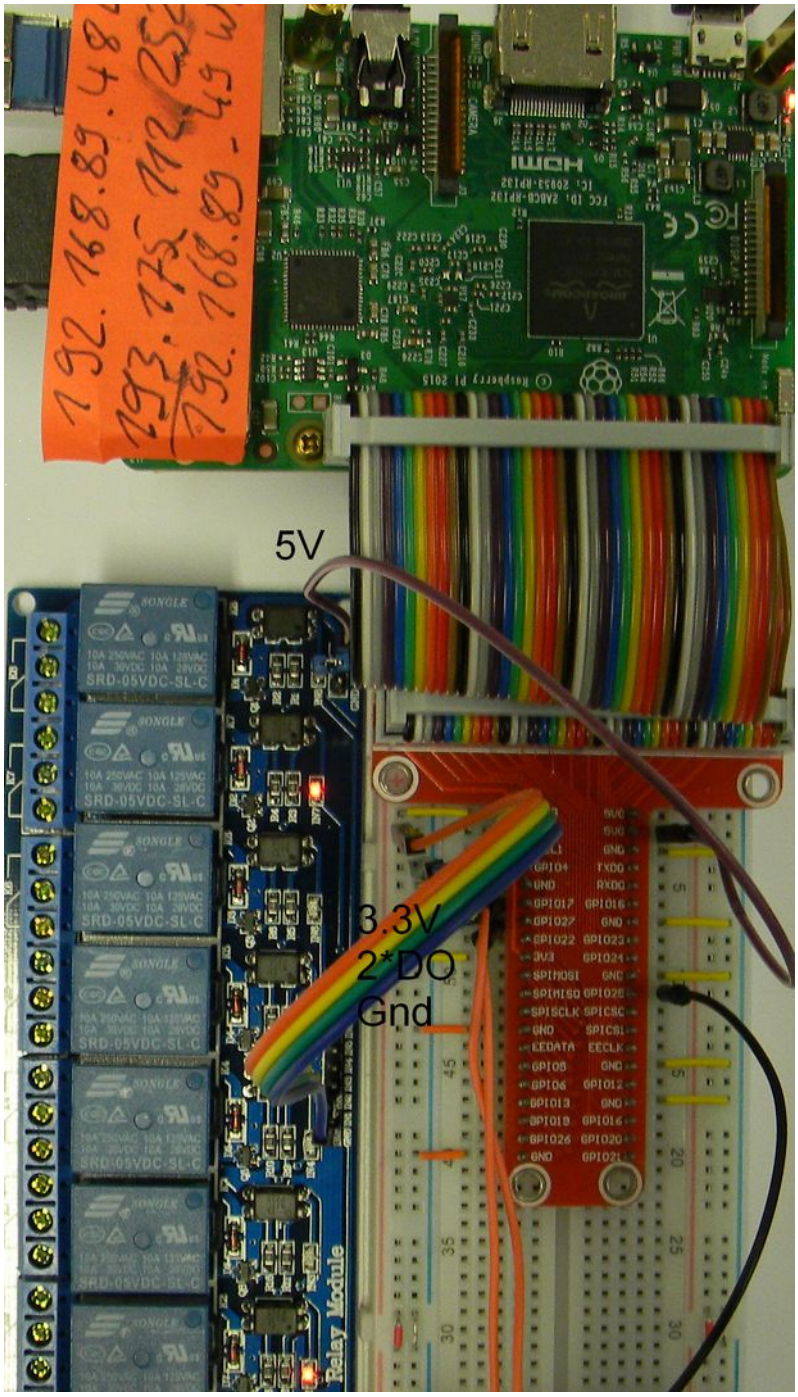

Fig. 6: Eight relays module 10A

Figure 6 shows eight relays module with 10 A changeover contact 250 V or 30 V=. Obviously it has one status LED per relays (which is quite nice) powered by the 3.3 V side (which is OK). These

LEDs are red, which is the same ~~sh...~~ mistake as with Raspberry's power LED: in all process control standards red means error/fault.

Besides this “red light sin” and besides being "no name and no documentation" it is more than OK for less then 10 €. As the controller side and the 5 V relay supply have common ground (in most exemplars), the opto-couplers seem nice overkill (no circuit diagram available). And as a surprise the 8 control inputs are low-active.

piXtend + codesys

As mentioned, piXtend + codesys is working. Details, application note and licence is found by http://www.pixtend.de/pixtend/downloads/downloads-v1-3-english/.

To sum up, considering the advertising and the complete price including case, DIN rail bottom and Codesys license, the properties as industrial PLC, the mechanical quality, the IDE etc. won't fill the customer with enthusiasm.

Additionally the piXtend (image) distribution ran into the libc6 dependency disaster at updates inhibiting all further update/upgrade at the Raspberry affected.

For a deep-rooted Codesys fan it might be attractive.

piXtend pure

The piXtend extension for Raspberries is also usable without Codesys by an extended wiringPi library. You need to install wiringPi, as done above. Then you might want to install the piXtend tools and libraries (cf. [54]):

```
cd ~
git clone git://git.code.sf.net/p/pixtend/pxdev pxdev
cd pxdev/
chmod +x build
./build
```

To install git see page 15.

In our case ./build reported one fatal error (on missing menu.h), but at least the command line tool pixtendtool was usable. It might be necessary to enable SPI by raspi-config → interfaces., before. Contrary to its title [54] tells nothing on the library and its use. So one is left high and dry when writing C programmes for piXtend.

Resume: The non-Codesys software accompanying piXtend is not usable for own control programs. On the other hand, besides flaws, the hardware has some nice features, like Can bus, RS485 and 12/24V IO and supply. The hardware interface to the Raspberry is essentially one SPI shared in a complicated way, additionally obfuscated by using an AVR controller as SPI slave for multiple IOs. This single SPI seems a severe bottleneck and one reason for the poor timing performance with Codesys: here 100ms seems the fastest guaranteed cycle, possible.

Nevertheless, having unused piXtend boards and encasements, it might be worthwhile to make a suitable driver software on base of pigpio[d].

Communication hardware – RS485

As the Raspberry Pi3 has 4 USB, Ethernet, WLAN and with some extra software/configuration even Bluetooth, there's seldom need for extra communication modules. Well, one exception may be Modbus (see below) over RS485 often to be found in quite interesting equipment, like heat pumps, gas ovens, "smart" one or three phase power meters and counters, to name just a few.

Most DIN rail-mounted energy meters come with a so called S0 bus. That's just an opto-coupler or switch output giving 300..1000 (as configured) 100 ms closures per kWh. These can be observed by Raspberry's GPIO input with pull-up and counted allowing seldom energy consumption updates and seldom and coarse non-equidistant average power samples.

Note: The so-called S0-bus is a good example of transferring older technology's solutions without any cogitation to a new one: It's just counting and stopwatching the Ferraris wheel – but without being able to see the energy flow direction / sign of power.

The better, often only slightly more expensive, electronic meters offer a real bus connection. And, usually in this and some other fields, that will be Modbus over RS485. In the exemplary case of power meters this gives precise and actual measurements of voltage, current, frequency, active and reactive power, overall consumption and often more.

In our context, a configuration with those devices would be a RS485 bus with the Raspberry as Modbus master or client and one or many of those devices as Modbus slave respectively server.

RS485 is a serial two wire (0..5 V, 3V sufficient push pull) serial link, half-duplex, standard UART bytes + start and stop bits. To get the two wire one can

- use an USB to RS485 stick or
- attach a TTL to RS485 converter to Raspberries standard serial link:
  UART: TxD = GPIO14 = pin8; RxD = GPIO15 = pin10

The USB stick solution is commonly reported to bring driver problems and, if gotten to work, causing resource conflicts and system crashes. The reason may be Raspberries USB link already overused / misused for other build-in USB to XYZ converters.

Hence, one would stick with Raspberry's standard UART (pins 8 and 10). Besides extra RS485 (or even Modbus) related tasks, using just this UART is more difficult than it should really be. This is partly due to no documentation at all or worse misleading documents and examples as not for which Pi variant they are applicable.

In short for the Pi3:
Enable serial and disable serial console by raspi-config or do / and check

```
enable_uart=1
```

in /boot/config.txt. And reboot.

Do use /dev/ttyS0 – or the link /dev/serial0 – e.g. like:

```
int serHnd = serial_open(thePi, "/dev/ttyS0", ...
```

Using other devices (like the notorious /dev/ttyAMA0) with Pi3 will not work and, sometimes worse, will make the application or at least the thread grind to halt.

When having the UART going, the TTL to RS485 solution as such is quite simple. In a minimal form a MAX485 IC and three resistors would do it. Little modules much like this are less than one Euro (never to be confused with figure 6's module).

Problem with this minimalistic solution is being left alone with the "transmit enable" signal = "DE" on MAX885. As we have logically a one wire half-duplex bus, a sender must

- apply "transmit enable" before the first start bit of its (e.g. Modbus) telegram and
- remove "transmit enable" very shortly after the telegram's last stop bit.

Note: “Receive enable” might be the inverse of "transmit enable" or always on.

The suggesting idea of using an extra GPIO pin to DE is naive:

- we would have to modify the drivers respectively Modbus library at every point, where sending a telegram would start.
- And while getting the start (even if error prone and killed by library updates) would be feasible, but hitting the telegram's end usually evolves to a not solvable problem.

In the end, pure hardware solutions generating the the DE signal from TxD are the most simple and (well made) totally reliable. There are modules of that kind, just translating RxD/TxD to/from RS485-A&B; see figure 6. Besides doing the TTL to RS845 transmission line (A&B) translation by a MAX485 IC (with auto DE) for about 10 € it brings all covered GPIO and supply pins to extra connectors. Hence other (process) IO can still easily be attached.

Figure 6 shows this for a relays module as example.

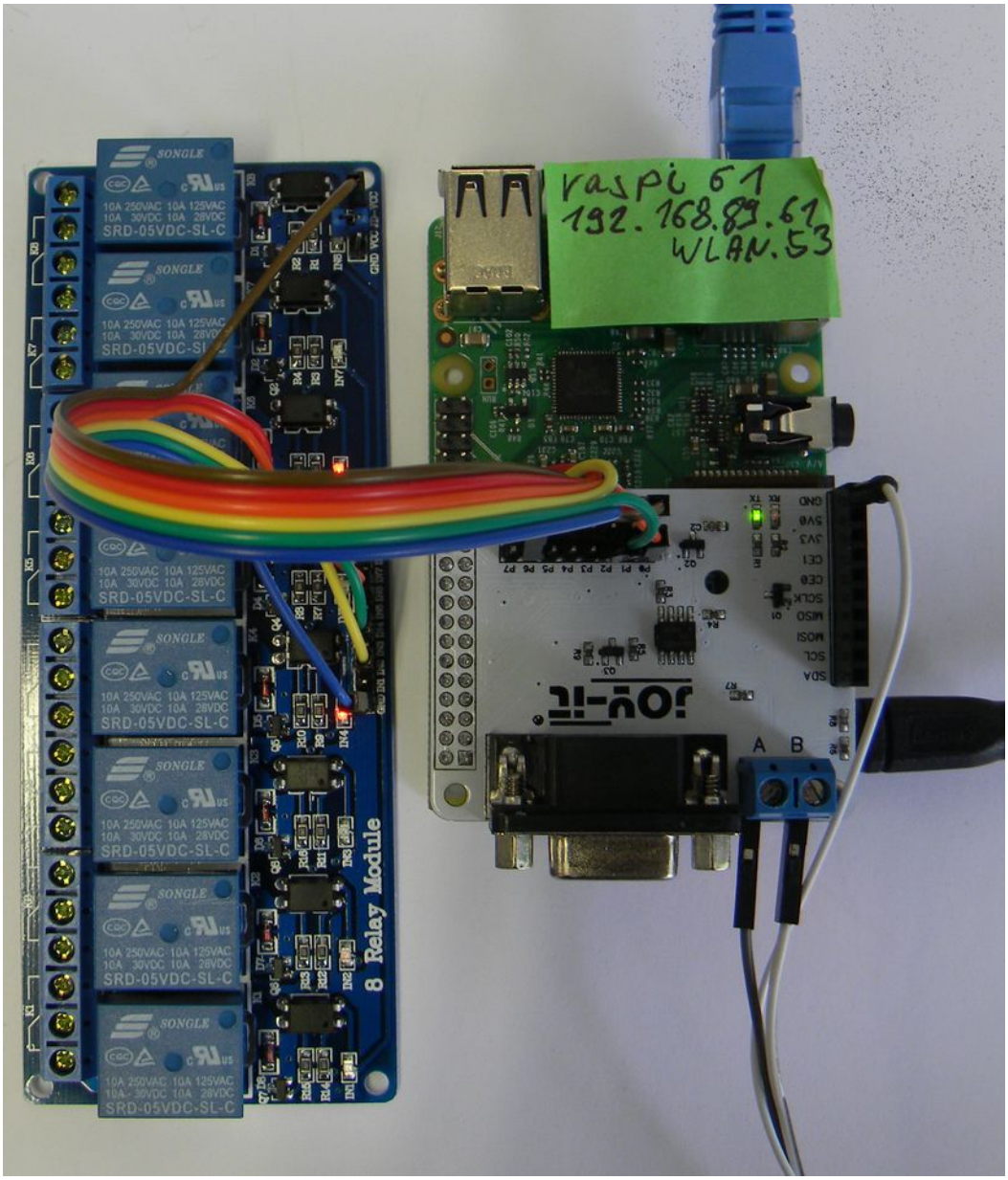

Fig. 6: RS485 module with automatic DE

The Plus side of this module ("joy-it") is

- the reliable DE automatism and
- forwarding the Raspberry pins unused for RS485 to an own pin header supplying an eight relay module and serving eight GPIO (outputs) to it

The Minus side is

- having only two screw clamps "A" and "B". One always needs "Gnd", too!

PoE – power over Ethernet

Switches, a key matrix in our example, low power LEDs and speakers / beepers and the Pi camera are peripherals one may use without extras hardware interfaces, often called shields. A Raspberry with such direct attached IO is put at its place of action and controlled via LAN. Then getting the 5 V, 1.3 A is an extra complication, often underestimated. IEEE 802.3af switches can provide power over Ethernet (36..57 V, up to 25 W). On site a step down converter has to provide the 5V.

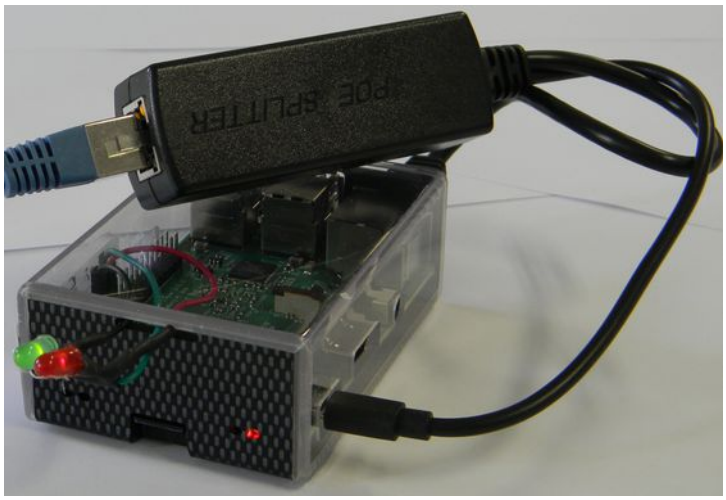

Fig. 8: Power over Ethernet splitter 5V, 2A

IEEE 802af splitters to LAN without power and 5V on microUSB, see figure 8, are on purchase for about 10€. Do not use non IEEE “solutions” without converter to feed the 5 V from the other end of the LAN cable.

Communication

For Raspberries, the natural way to communicate with companions on the same process control tier, with servers on higher tiers or with HMI systems is IP via LAN or WLAN (best private). By the way, for micro-controllers, like e.g. most Atmel AVRs boards, this is not their “natural” way.

Protocols

(W)LAN and TCP/IP communication is an integral part of Raspberries and most available OSs, including Raspbian lite. A whole bunch of protocols, applications and libraries is available, from start or after a bit of apt-getting. This includes ftp, http, SCP, SSH, Telnet, rlogin and more – some of them used here.

For process control communication (of mostly of IO values and usually time critical) one might be tempted to write own binary protocols. On base of the GCC socket library (sys/socket.h etc., [63]), this can be done. Often it's wiser to use a standard protocol.

Modbus

Modbus [65], [66] is an industry standard protocol to communicate process IO data. It is in wide spread since its origins in 1979 by the PLC manufacturer Modicon (now Schneider). Until today it is supported by most small automation systems or PLCs. It started as a P2P RS323 link and was later extended to multi-slave/server by RS485 and also by TCP/IP. Like many beloved programming languages and protocols of such age, Modbus has a bundle of architectural flaws the worst of which were a fault already in 1979.

Modbus has no layer concept and mixes physics, transport and application in an unfortunate way. Additionally the standard makes inadequate references to concrete devices and their addressing idiosyncrasies – as indexing from 1 or putting registers to local address 4000 (hex?). As none of those has any effect to the protocol and its telegrams these references to “4xxx registers” and the

like are a rich source of confusion. A lot of secondary literature just dwells on what belongs to the standard, how to interpret it and what was meant only for a “984A/B/X” machine.

Modbus has no data type concept worth the name.
One type is "coil" [sic!]. That would be copper wire for relays, but is, of course, just a boolean forced to one bit in the end. Modbus insists to transfer thousand of bits (sorry "coils") starting at arbitrary odd bit addresses aligned to bytes – meaning RS323 bytes since 1979 and also TCP bytes a bit later. Imagine an AVR Atmega controller with just single bit shift machine instructions at one or both ends. The load of shift instructions may be larger then the whole automation task.

The only other Modbus data type is called "register" which is just a16 bit something. (Above transport Modbus knows no "byte"). Even when the application "thinks" in bytes, this approach brings the full computational load and risk of errors by endianess handling – without doing any good for the nowadays application's 32 or 64 bit data.

For the serial interfaces RS232/485 – still in wide use for small controllers – Modbus won't use UART parity (available in 1979) supplemented by a simple (XOR or ADD) checksum. Instead the standard requires a complex CRC telegram checksum. This overkill is, again, overcharging poor small controllers (even when using a clever double look-up algorithm) and forces low baud-rates just to reduce the telegram and CRC load. Serial Modbus has no control flow concept, but a set of pause and time-out requirements forcing modern buffered UARTs to low gear with the cost of extra CPU instructions. Some newer Modbus implementations ignore these requirements by implementing "full speed" – the better ones at least documenting or advertising it. But beware: Other stations might just fail or reject to communicate with a non-conformer. This can't be complained on, as those timing requirements are the base for telegram synchronisation.

Lets not dwell longer than necessary on list of principal Modbus flaws. And let's neither bash nor reject using Modbus just therefore – similar could be said of other geriatrics like C, C++, FTP ...

One last point: Modbus by itself has no security measures. This can and must be handled by using protected networks, only. Serial RS232/485 lines should always fall in that category. For the preferred TCP/IP use protected private LANs (or tunnels).

Resume: As it is a widespread industry standard we should use Modbus on TCP on our Raspberry servers. With the GCC libraries and the Ethernet port we have all infrastructure at hand.

Serial (RTU) Modbus mostly via RS485 should be avoided as long as all partner are linked via Ethernet. Additionally Raspberries would need extra hardware (like MAX485 modules) to have RS485. On the other hand (and as said above) there are many small process IO devices with good value for money and RS485 Modbus interface around, like smart meters.

Linking those to a Raspberry opens a rich field of applications with professional process IO.

So let'sd do Modbus – preferably Modbus TCP – but lets be open for RS485 if an interesting application calls for it.

libmodbus

Based on socket libraries ([63]) implementing a minimal subset of of the Modbus protocol (class 0 on TCP IP e.g.) is relatively easy. Going to higher classes or other interfaces (and testing it all) will get hard work. But even with small subsets its re-inventing the wheel, considering age and wide use of the protocol. But looking for a reliable and conforming Modbus library for Raspberry (Linux) was harder than expected.

In the end we used Stéphane Raimbault's libmodbus. It is

- \+ function code complete
- \+ including even exotic function codes 17/11, which is
  - ▼ implemented in a useless way reporting fixed constants and always “PLC run”
- \+ available and tested on many platforms
- \+ in wide spread use. It has
- \+ implemented all interfaces, TCP/IP and serial,

- \+ using an (inconsistent) abstraction layer concept for the interfaces. So the application software would be less affected should one switch from TCP to RS485.
- ▼ The library is a “heavy malloc user”, which gives bad feelings in connection with real time and process control.
- ▼ The well and comfortable working modbus_receive() modbus_reply() server function pair offers no (call back) interface to handle output just put and input just requested. Time permitting you'd have to bring all possible input before and get all possible output after receive().
- ▼ There is no real documentation of the library's functions, the data structures, nor on semantics and architectural ideas.
- ▼ Besides some typos there are a few bugs inhibiting cross-compilation and Eclipse (F3) look-up.

One Eclipse / compiler troubling example: In

```
static int response_exception(modbus_t *ctx, sft_t *sft,
                              int exception_code, uint8_t *rsp,
                              unsigned int to_flush,
                              const char *  template, ...)
```

the name “template” is considered as keyword misuse by some tools – rename it “tempel”, e.g.

This little list may sound worse than it is meant.

The “heavy malloc” gives raise to fundamental objections and the linked multi-stage pointing leads to horrible readability. On the other hand, it works – and too naïve replacing malloced structure with fixed static ones may lead to funny conflicts with Raspberry's MMU (ARM, Linux activated).
To resume: With libmodbus we do have the complete sources of an excellent library to work with – and to learn from.

Installation

The installation need git-core; to install git see page 15.
On your Raspberry's shell or putty just do: (see note below)

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install -y autoconf libtool
git clone https://github.com/stephane/libmodbus/
cd libmodbus/
dir
./autogen.sh                                    ## *)
./configure --prefix=/usr/local
```

```
libmodbus 3.1.4
::::::::                                          *)
```

```
sudo make install
dir /usr/local/lib
cd src/
dir
dir .libs/  ## here are the .so files
dir .deps/  ## what the hell ?

cd ~/libmodbus/tests/
./unit-test-server &          ## **)
./unit-test-client
```

Note *): All installation scripts take considerable time and produce lots of output, be patient.

Note **): Normally a test-server started in background would end, when the test-client disconnects. If this fails do

```
killall lt-unit-test-server   ## Yes, the process' name differs
```

Due to a bug in the test-servers they work only with clients on the same machine; change the source accordingly and re-build.

A note on "just do this!" to install libmodbus

Well, due to lacking any respectable background documentation on libmodbus this "just do!" was the biggest hurdle.

To begin with, why should I use libmodbus at all?
(Because it's good and, when not fitting, adaptable.)
What does the above installation – many pages of scripts! – do with my system?
(Gives you some 7 files, needed: the sources and one .so.)
What is the – never before used – libtool and why would I need it?
(To understand get [64] and read it. You would not need it, and it does not hurt.
It's just a help for the project owner to serve many targets.)

So the short answer to "Should I do this complicated installation?" is: Yes, do it. Be courageous or make a backup before.

In the end you need the following files – and transferring those from your installation Raspberry to another one does the installation job:

- the sources of the library and and the tests if you like
- the include files. Put the includes to /usr/local/include/

```
-rw-r--r-- 1 root staff  11155 2017-08-04 14:34 modbus.h
-rw-r--r-- 1 root staff   2124 2017-08-04 14:34 modbus-version.h
-rw-r--r-- 1 root staff   1199 2017-08-04 14:34 modbus-rtu.h
-rw-r--r-- 1 root staff   1373 2017-08-04 14:34 modbus-tcp.h
-rw-r--r-- 1 root staff   3430 2017-08-04 14:34 modbus-private.h
-rw-r--r-- 1 root staff   7690 2017-08-08 09:20 config.h
-rw-r--r-- 1 root staff   1627 2017-08-04 14:34 modbus-rtu-private.h
-rw-r--r-- 1 root staff   1247 2017-08-04 14:34 modbus-tcp-private.h
```

- the library files. Put them to `/usr/local/lib/`

```
lrwxrwxrwx  root staff  2017-07-21 libmodbus.so   -> libmodbus.so.5.1.0
lrwxrwxrwx  root staff  2017-07-21 libmodbus.so.5 -> libmodbus.so.5.1.0
-rwxr-xr-x  root staff  123408    2017-07-21 14:32   libmodbus.so.5.1.0
```

Do not forget to say `sudo ldconfig` afterwards. Otherwise you might get incomprehensible errors when running (cross-build) Modbus applications on that machine or when linking there.

Just to run a cross-build Modbus programme on another Raspberry an unlinked libmodbus.so ftp-transferred there should suffice. after cpoying it to `/usr/local/lib/`:

```
sudo cp libmodbus.so /usr/local/lib/       ## to where it belongs
sudo ln -s  /usr/local/lib/libmodbus.so /usr/local/lib/libmodbus.so.5
sudo ldconfig                   ## what ever it does, never forget
```

But it turned out that for what ever reasons programmes required `libmodbus.so.5` even when made with -lmodbus.

Compile and cross-build

After a successful installation or transfer one can locally compile:

```
cd ~/ibmodbus/src/tests          ## or where ever the .c source is
gcc version.c -o version -lmodbus
./version
```

To be able to cross-compile, cross-make and cross-build from our (Windows) workstation – and to use Eclipse there – we have to get the include files to
`C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\include`

```
Verzeichnis von C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\include

22.07.2017  11:22             1.199 modbus-rtu.h
22.07.2017  11:22             1.373 modbus-tcp.h
22.07.2017  11:22             2.114 modbus-version.h
22.07.2017  11:22            10.912 modbus.h
```

One may also take

```
22.07.2017  21:37             3.405 modbus-private.h
22.07.2017  22:04             5.829 config.h
22.07.2017  11:22             1.627 modbus-rtu-private.h
22.07.2017  11:22             1.247 modbus-tcp-private.h
```

Those more “secret” .h files would be needed to (re-) build the library itself and when digging a bit deeper like building and using their data structures. It is no fault to take them from start.

And we need the get the ("un-linked") file libmodbus.so (~120kB)
and put it to `C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib`

In a (test) project we let see Eclipse the library sources primarily to have "Open declaration / F3". And we let them make by a suitable make file. While make (from command line or by Eclipse's build project) is no problem for all library sources Eclipse's CDT complains:

```
Warning "Code Analysis Problem"
No break at the end of case    modbus.c       /rasProject_02/modbus          line 439
```

The well commented case fall through is OK. To get also Eclipse to 0 warnings one has to add a "//no break" comment.

```
                    msg_type);
                if (length_to_read != 0) {
                    step = _STEP_META;
                    break;
                } /* else switches straight to the next step */
                //no break
            case _STEP_META:
                length_to_read = compute_data_length_after_meta(
```

Cross-compiling example:

```
arm-linux-gnueabihf-gcc -c -I. -DPLATFORM=raspberry_03  -g -Wa,-adhlns=bau/tests/version.lst
    tests/version.c -o bau/tests/version.o
```

Cross-build example:

```
arm-linux-gnueabihf-gcc -I. -DPLATFORM=raspberry_03  -g -Wa,-adhlns=bau/tests/version.lst
    tests/version.c -lmodbus -o au/tests/version
```

libmodbus application and experience

Refer to the SVN project by

`svn checkout` https://weinert-automation.de/svn/rasProject_01/

or have a look at them in your browser (guest:guest authentication allows browsing, checkout and update). And see the examples

- keysModTCP.c a Modbus server doing real prpcess IO, a key matrix (12 keys), 3 LEDs and one beeper (in an 1ms cycle) for
- keysModClient.c a Modbus client using the input and controlling the output (in an 100 ms cycle).

Real time

Every server doing work for others and every system doing process IO does so under timing requirements. These requirements should be well defined, best clearly documented and with consistent numbers and units.

But in some fields of application and communities much of it may be considered as implicitly clear, be it by the industry's standards or by properties of the usual standard products seen as not disputable *). If someone with PLC background says “someThingX is done in the 1ms cycle” following is implied:

- there is a trigger event by the underlying runtime every 1ms “exactly” … well ±
  - **±** accuracy (timing oscillator / source, short time)
  - **±** jitter (variability of delays/latency)
- there are 86.400.000 such trigger events per day “exactly” … well ±
  - **±** accuracy (long time; can sometimes be made better than short time value)
  - **+** exactly 0 or exactly 1000 events per leap second (Google, Oracle & others: 0)
- “someThingX” will run at every such trigger event “exactly” … well ±
  - **+** 0..max delay/latency (meaning max. .. later but never ever too early)

We have “real time” when all those times and intervals are specified to the applications requirements and “hard real time” when one occurred violation is to be considered as application failure. While the absolute numbers are important the “hard” property has per se nothing to do with speed. You may replace 1ms by 1s in our mental example and relax other requirements but consider not having (86.400 + number of leap seconds) runs per day as a failure . (It may be astonishing how “hard” that can be in some environments).

Keeping in PLC / cycles and with a Raspberry Pi3 / Raspbian lite doing just process IO and related communication we can offer respectively implement 1ms cycles. Going faster would be daring.
Note *): The same is true for hardware, electrical signals, maximum ratings of IO, EMC etc. Most IO extensions, shields and so on are totally inadequate in this context.

Absolute timing

The approach in our very first LED blink examples (listing 2, ending page 17, e.g.)

```
do something
delay (relative from now)
do something
delay (relative from now)
… and so on
```

never gives an exact timing, how ever exact the delay function used may be. Believe it or use a good logic analyser to verify. The basic recipe to start the “do something” at exact intervals relative to one start point in time is to use absolute time steps as supported by Listing 7's functions.

```
/* Absolute timer delay for the specified number of µs
 *
 * @return sleep's return value if of interest (0: uninterrupted)
 */
int timeStep(timTv * timer, unsigned int micros){
   timeAddNs(timer, (long)(micros) * 1000); // timer += micros
   return clock_nanosleep(CLOCK_ABS, TIMER_ABSTIME, timer, NULL);
} // delay(unsigned int)

/* Absolute timer initialisation
 *
 * This function initialises the time structure provided.
 *
 * @param timer the time structure to be used (never NULL!)
 */
void timeInit(timTv * timer){
   clock_gettime(CLOCK_ABS, timer);
} // timeInit(timTv *)
```

Listing 7: Absolute time steps; excerpt from weRasp/sysUtil.c

This is also the base idea to get PLC like cyclic tasks. Here the execution time of “do something” including all jitter and latencies and considering it's CPU/core usage must be shorter than the (next) timing step.

Latency and accuracy

Of course, it is of no avail to offer 1ms cycles for process control, when the runtime's latency (in the sense of delays on signals) is in the same order of magnitude or worse. One common command line tool to check latency is cyclictest. Get it by:

```
sudo apt-get install rt-tests
```

and run it by e.g.

```
sudo cyclictest -l100000 -m -n -a0 -t1 -p99 -i400 -h400 -q
```

After some patience on a Raspberry Pi3 with process load (keysPiGpioTest, pigpio daemon) and some communication load we get a typical result like

```
# Total: 0001000000
# Min Latencies: 00009 # observed  4...12µs
# Avg Latencies: 00012 # observed 11...13µs
# Max Latencies: 00062 # observed 70..118µs
# Histogram Overflows: 00000
```

The tool and its options as well as the interpretation of the results are non-trivial. Anyway, running the same test will give comparable results on different run-times and loads. Experimental outcomes are:

- A well configured (specialised, single purpose, single use) Raspian lite on a Pi3 is suitable for hard real time process control and 1ms cycles.
- A graphical (non lite) OS never is.

The accuracy of the 1ms cycle may very well measured by precise logic analyser observing outputs by the 1ms cycle and derived longer ones (n * 100ms) over a long period. In one case we observed an hour being 144ms too long (~3.5s/d). By (excerpt)

```
absNanos1ms = 1000000 + vcoCorrNs;

while(commonRun) { // timing loop in main thread
   timeAddNs(&cyc1msEnd, absNanos1ms); // 1 ms time step
   clock_nanosleep(CLOCK_ABS, TIMER_ABSTIME, &cyc1msEnd, NULL);
   if (++msTo100Cnt >= 100) {
```

with the signed byte vcoCorrNs set to -40 we improved the accuracy by 2 orders of magnitude. This correction by a calculated fixed value worked very well over many days of uninterrupted use. So we can conclude this Pi3's quartz stability being excellent while the accuracy could be better.

Of course, this “hand-made” correction value is no practical solution for wide use. vcoCorrNs should be automatically determined by a phase locked loop (PLL) “voltage controlled oscillator” (VCO) algorithm against a precise time source, like NTP (available) or DCF77 (extra hardware).

Cycles and threads

For PLC like cyclic execution we offer an 1ms cycle and a 100 ms cycle by library (sysUtil) and run time support organising the manager (supplied) as one thread. The cyclic tasks as well as other event triggered ones will have be supplied as user threads. In a minimal runtime for AVR µControllers we based a similar solution (proven 24/7 since over 6 years) on the lightweight protohreads.

As protothreads are well suited for GCC they could have been used too. But having here a full grown Linux runtime on a multi-core processor it is more appropriate to use the runtime's own threading (pthreads) and signalling system instead. See sysUtil.c and sysUtil.h after getting or updating the project.

Making a library

In the example of the cycle (1ms and 100ms) based, multithreaded programme for key matrix and LED handling we have three sources

- keysPiGpioTest.c — main programme organizing two cyclic tasks with three threads
- weRasp/sysUtil.c, include/sysUtil.h — utilities and cyclic task execution support
- weRasp/weGPIOd.c. include/weGPIOd.h — IO support for using the gpio library (daemon, socket), matrix scan support

The usual way is to translate all three .c to .o by

```
arm-linux-gnueabihf-gcc -DMCU=BCM2837  -I./include  -c -o
  weRasp/sysUtil.o  weRasp/sysUtil.c
```

best organised in the makefile and link all three .o files to the executable keysPiGpioTest. With this procedere you may

- change every source before cross-buildThe good point is
- get one monolithic (mid-sized) executable you may transfer to and
- run on every Pi where a pigpiod is installed and the daemon is is running.

On the other hand you may be tempted to make one (or more) of the utilities a library – in our exemplary case sysUtil.c. With this procedere you may

- separate stable utility and runtime code from the more volatile application sources
- keep the source code away from the application programmer.

Then you translate one source less and link the extra library with the -sysUtil option. The library libsysUtil has to be present on the cross-build workstation as well as on every Pi where an application linked against it must run.

To make the library on the workstation do:

```
arm-linux-gnueabihf-gcc -Wall -DMCU=BCM2837  -I./include  -shared
  -o weRasp/libSysUtil.so -fPIC weRasp/sysUtil.c

copy weRasp\libSysUtil.so
  C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib\
```

Transfer the library libSysUtil.so to the Raspberry to a directory remotely (ftp) accessible. There do:

```
sudo cp libsysUtil.so /usr/local/lib/
sudo chmod +x  /usr/local/lib/libSysUtil.so
ldconfig
```

The last command is necessary on some distributions as it builds the fast library cache used to avoid following the notorious link chains (libX in path directory is libXv10 ín directory Y is libXv10.31alfa ín directory Z ….). Think of it (and on the x bits, of course) when the application won't run complaining “missing library”.

The result – and where we are

We can handle the GPIO pins of both Raspberry Pi1 and Pi3 in C programmes. We can handle a bundle of IO libraries pigpio being the most promising one. And we know how to cross-compile and cross-build from Windows using make, Eclipse and SVN there. With make and some include file wizardry it is possible to integrate the WinSCP.com file transfer to a target Raspberry in the automated make processing.

C is still the lingua franca in embedded. On the other hand alternatives should be tried and considered – first of all Java.

Some actuators and sensors may directly connect to Raspberry's GPOI but often extra interfacing IO hardware will be needed. We used and tried piXtend which might be disappointing in the light of price and promises. Nevertheless it opens the world of Codesys and Web interfaces to process control applications.

Both topics, process hardware and web access will be worked on further.

And staying with C and the pigpio library we digged a little deeper on real-time, PLC like cyclic execution and put such extra features in utility libraries.

For communication process IO and related data to/from small systems the quite old Modbus protocol is still in wide use. On base of the libmodbus library we made our Raspberries Modbus servers.

A p p e n d i x

Miscellaneous commands

This is more or less an anthology of useful and proven tips.

Please find most of it the appendices of [29] (Linux server) and [30] (Docker).

Find all Raspberries in the private net (192.168.89.*)

```
sudo apt install nmap
sudo nmap  -sP  192.168.89.0/24 | awk '/^Nmap/{ip=$NF}/B8:27:EB/
  {print ip}'
```

This will find Raspberries connected via WLAN, too. Another tip was

```
sudo nmap -sP 192.168.89.0/24 | awk '/^Nmap/ { printf $5" " }
  /MAC/ { print }' - | grep Raspberry
```

which gives a nice listing assigning MAC to IP addresses.

List all visible WLANs (Pi3)

```
sudo iwlist wlan0 scan
```

Make consistent and comparable directory listings

Make a good file listing command command by:

```
alias dir='ls -lA --time-style=long-iso'
```

To make it permanent and and have some comfort and the usability of sudo with it, best add the following in ~/.bash_aliases (for one user):

```
alias dir='ls -lA --time-style=long-iso'
alias diR='ls -lAR --time-style=long-iso'
alias sudo='sudo '
```

To have it for all users put it in a file /etc/profile.d/bash_aliases.sh instead; make it if not there.

Generate encoded password for WLAN (ssid atMEVAnet e.g.)

```
wpa_passphrase atMEVAnet
```

The five liner output may be appended to /etc/wpa_supplicant/wpa_supplicant.conf after putting the encoded password in quotes and hiding the clear text password comment.

Get the host-key

```
ssh-keyscan -t rsa 192.168.89.42
```

and accept it

```
ssh-keyscan -t rsa 192.168.89.42 >> ~/.ssh/known_hosts
```

Show all threads of a programme

```
ps aux | grep programme ## get the pid 28344, e.g.
ps -T -p 28344 ## use right pid here; see main + the extra threads
```

Abbreviations

More abbreviation can be found in [29]'s Appendix.

24/7 24 hours on 7 days a week; uninterrupted service (by hardware or software)

ACL Access control list

CLI Command line interpreter/interface

CoDeSys Controller development system no-free IEC 61131-3 IDE for Windows

CRC Cyclic redundancy check, a polynomial division

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System; also used in the sense of domain name server.

FTP File Transfer Protocol; RFC1579

GPIO General purpose IO. The µP's IO pins that have no purpose for the Raspberry nor for its OS, but kindly having been made available for other purposes on pin grids.

GUI Graphical user interface / graphical HMI

HMI Human Machine Interface (without political correctness formerly MMI)

ID Identity; in the sense of a domain's ID management

IDE Integrated development environment (like Eclipse)

IO Input and Output

IP Internet protocol

LAN Local Area network; here in the sense of just Ethernet

MMU Memory management unit

MS Microsoft

NOOBS New out of the box software

NTFS NT File System; full featured file system with fine grained access rights, links and all else used on all Windows NT inheritors

OS Operating System; run time

P2P point to point, exclusive direct link between two communication partners

PoE Power over Ethernet

RAM Random access memory, also implies writeable

RDP Remote Desktop Protocol; from Microsoft

RFC Request for comment; internet standard

ROM Read only memory; storage for fixed values

sic! so, exactly so (even if unbelievable) or wrong in the (cited) source already

SD Secure digital memory card; interfaces and protocols by SD Assoc.

SPI Serial Peripheral Interface (shift register attachment protocol)

SSD Solid state disc, a disc drive made of non-volatile RAM

SSH Secure socket shell

SSHFS (remote) File system over SSH

SSL Secure Sockets Layer; former name of TSL

U[S]ART Universal [serial] asynchronous receiver and transmitter

WS workstation, often in the sense of PCs and laptops of all sizes

W10 MS Windows 10

xRDP Linux' RDP (particulate) adaptation on the (X) server side

µP Micro-processor

References

[29 ..31] use identical reference numbers, hence the gaps.

[29] Albrecht Weinert, Ubuntu for remote services, Report, November 2016, a-weinert.de/pub/ubuntu4remoteServices.pdf

[30] Albrecht Weinert, Ubuntu for docker, Report, April 2017, a-weinert.de/pub/ubuntu4docker.pdf

[31] Albrecht Weinert, Raspberry for remote services, Report, May 2017, This paper (the last actual version): a-weinert.de/pub/raspberry4remoteServices.pdf

[51] Raspberry Org, FTP, Tips on using SSHFS https://www.raspberrypi.org/documentation/remote-access/ssh/sshfs.md

[52] Raspberry Org, SFTP, Tips on using SFTP https://www.raspberrypi.org/documentation/remote-access/ssh/sftp.md

[53] Raspberry Org, SFTP, Tips on enabling WLAN by command line https://www.raspberrypi.org/documentation/configuration/wireless/wireless-cli.md

[54] Qube solutions, Linux Tools & Library for PiXtend – Manual for installation and use of the pxdev-Package with Raspberry Pi and PiXtend (says nothing on library use) V1.05, April 2017 http://www.pixtend.de/files/manuals/AppNote_pxdev_DE.pdf

[55] WinSCP, FTP [command line] client, documentation https://winscp.net/eng/docs/start

[56] Broadcom, BCM2835 ARM Peripherals, data sheet 2012 https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[57] Gert van Loo, QA7ARM Quad A7 core, Technical report on BCM2836, Rev3.4 2014 https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf

[58] Broadcom, ARM® Cortex®-A53 MPCore Processor, Rev. r0p2Technical Reference Manual DDI0500D_cortex_a53_r0p2_trm.pdf (BCM2837 is Quad-core 64-bit ARM cortex A53 CPU) http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf

[59] Broadcom, ARM® Cortex®-A Series, Version 1.0, Programmer's Guide for ARMv8-A http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf

[60] Shore, Chris, ARM, Porting to 64-bit ARM, white paper July 2014, http://malideveloper.arm.com/downloads/Porting%20to%20ARM%2064-bit.pdf

[61] N.N., Joan, pigpio library -- Download & Install, http://abyz.co.uk/rpi/pigpio/download.html

[62] N.N., Joan, pigpio library -- pigpio C interface, http://abyz.co.uk/rpi/pigpio/cif.html

[62] N.N., Joan, pigpio library -- pigpiod C interface, http://abyz.co.uk/rpi/pigpio/pdif2.html

[63] Hall, Brian "Beej Jorgensen", Beej's Guide to Network Programming – Using Internet Sockets Version 3.0.21, 2016 http://beej.us/guide/bgnet/output/print/bgnet_A4_2.pdf

[64] Gordon Matzigkeit, Alexandre Oliva, Thomas Tanner, Gary V. Vaughan, GNU Libtool Vers. 2.4.6, 2015 https://www.gnu.org/software/libtool/manual/libtool.pdf

[65] Modicon, Modbus Protocol, Reference Guide PI–MBUS–300 Rev. J 1996, http://modbus.org/docs/PI_MBUS_300.pd

[66] Modbus Org, MODBUS Application Protocol Specification V1.1b3, 2012 http://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf