Albrecht Weinert

Raspberry for distributed control

Abstract and Introduction

This technical report is about installing and using Raspberry Pis as little distributed servers especially for embedded process control applications, often in co-operation with other little machines in the same (W)LAN. That means: With the applications in question we always control external devices. And we do that almost always via Pi's I/O ports. Additionally we use MQTT and Modbus devices (screw terminals A&B in Fig. 1) as well and sometimes VoIP telephones (by Asterisk).

This report's predecessor "Raspberry for remote services" [33] was first published 2016 as a laboratory handbook for process control set ups with then Pi 2. Until July 2024, it was continuously used, updated and expanded based on experiences with Pi0, Pi3 and Pi4 (very seldom Pi1 or Pi2.) on a wide spectrum of real control applications, some of them in 24/7 use for years.

Some modules, libraries and tools we investigated and reported on originally turned out to be unfit to our real time requirements and we had stopped using. Hence, in October 2024, I decided to remove topics unused in the last 4..8 years, update and restrict to newer Pi OS and tools. The chapters were re-ordered under a new title as [31]. For omitted topics still see [33].

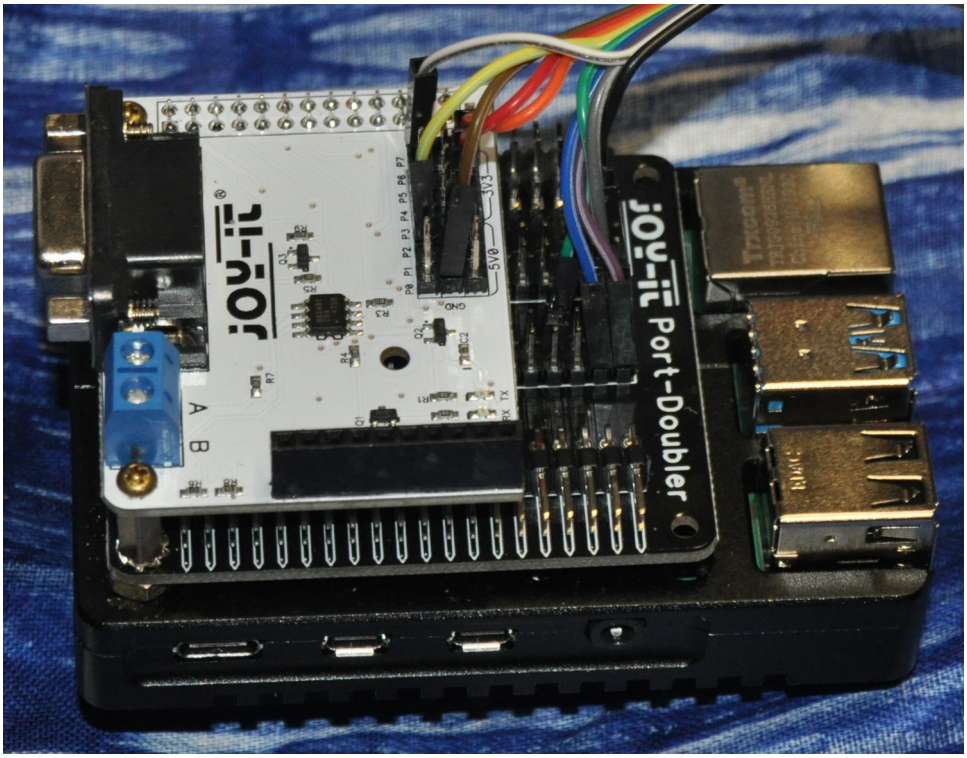


Fig. 1: "Raspi"4 with I/O

The operating systems will be a Raspberry Pi OS a Debian/Ubuntu derivate. Pi OS (formerly called Raspbian) per se is no real time OS. Nevertheless, obeying some caveats described here, the lite variant may well be used in applications where latencies of 200 µs are tolerable.

With careful C programming we can implement PLC like applications with 1 ms as fastest cycle based on a Pi OS lite. Under some conditions I/O programs in Java ([32]) are feasible.

Motivation and goals

One might use a Raspberry 4 or even 3, as a little low performance PC, nevertheless well outperforming last century's early PCs. And as PC substitute one would nowadays perhaps better use a Pi5. Such little workstations are not our topic here.

And to state it clearly from beginning: Until now we recommend *not to* use Pi5 for process IO or headless embedded control systems; see the Pi5 warnings on page 21.

Our Raspberries will be configured as a device for special purposes, often embedded and with some process IO. Distributed co-operating Pis will communicate over a private protected (W)LAN. Workstations, PCs etc. on that LAN will be used for human observation and control.

Monitor, keyboard and mouse will hardly ever be present on our process control Pis. Maintenance and installation jobs can be made via remote access (by ssh or even putty); observation and control can be implemented by web interface (Apache and CGI).

Following that route we will we will always get a headless controller that

- boots quickly and robustly at power on
- starts all its applications and/or services automatically
- runs 24/7 and has good real time performance
- fits, connects and re-connects robustly to its LAN and/or WLAN
- connects to process I/O via GPIO, SPI, RS232/445, private LAN, 1-wire and more
- provides administrative/service access remotely via ssl/ssh/putty and if applicable human process operation and observation via Web interfaces by a web server
- provides access to files via ftp (FileZilla, WinSCP)
- decent comfortable cross platform working environment using Eclipse and GCC tools.

Note: 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.

Part II is about process IO, communication and real time services. We look at hardware interfaces, GPIO handling, timing, latencies, threading and cyclic tasks.

In **Part III** we deal with additional hardware and some exemplary real use cases.

Using names

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

For References and Abbreviations please see the Appendix (page 73)., Refer to [29] for some Linux and server basics, abbreviations and else, also used here.

Prof. Dr.-Ing. Albrecht Weinert is founder and director of weinert-automation.

He developed safety critical and highly available automation systems with Siemens AG and was Professor of Computer Science at Bochum University of Applied Sciences.

a-weinert.de



Table of Content

A b s t r a c t and I n t r o d u c t i o n 1
Motivation and goals 1
On the content 2
..... 4
P A R T I The headless Pi 5
Preparing the SD-card 5
SD-card: save and restore 5
The size problem 6
First commissioning and enabling ssh (remote shell) 7
Enable WLAN – or disable it 9
Force IPv4 10
Getting a decent directory listing 10
Update the Pi OS 11
User interfaces 11
Apache 2.4 12
Installation 13
A cause of trouble 16
Installing PHP 8 18
Enabling file access on our workstation 19
SFTP – FileZilla 19
SFTP – WinSCP 19
Pi5 warning regarding pigpio and BCM2835 libraries 20
Cross-compile C for Raspberry from a powerful workstation 21
Eclipse 24
Command line – make project 24
Later – update, upgrade and beyond 26
Part I's results 27
P A R T I I I/O and process communication 28
The pigpio(d) library 28
PWM and RC servos on every GPIO 30
Resume for piGpioD and a look on other GPIO libraries 31
Other protocols (also) for process IO 32
Modbus 33
libmodbus 34
MQTT 37
MOSQUITTO 38
libmosquitto 39
Part II's results 40

P A R T III Process Periphery and control 41
Using the GPIOs 41
An application as service 50
rc.local 50
cron 50
Mimic a service – start stop restart enable disable 51
Making a library 52
Process IO hardware 52
LEDs and buttons – direct IO 53
Speakers and beepers 54
Relays 55
Power transistors 56
Hardware for serial communication – RS485 (232) 56
Serial impertinences 58
PoE – power over Ethernet 59
Communication 60
1-wire 60
Real time 61
Absolute timing 62
Latency and accuracy 62
Cycles and threads 63
Threading and synchronising 63
Co-operating applications 64
Shared memory 65
Semaphore sets 66
Watchdog 66
GCI programs 67
Data exchange with AJAX & JSON 69
Getting Web data – cURL 69
Asterisk 70
The result – and where we are 71
A p p e n d i x 73
Miscellaneous commands 73
Use nano without syntax highlighting 75
Abbreviations 79
References 81

P A R T I The headless Pi

The OS images mostly used are variants of Pi OS, successor of Raspbian.
For servers and process control with real time requirements we always use the “lite” variant and won’t have a GUI nor graphical tools. Standard Linuxes call that a “server distribution”.

Warning: GUIs with pointing devices (mouse, touch screen &c.) do definitely spoil Pi OS’s real time performance essential for our control applications. Hence, our Pis with “lite” OS come without monitor, keyboard and mouse. Computers without such devices are called “headless”.

Part I’s basic installation gives us a Pi with fixed IP address(es) we can handle, control and observe remotely. So we can put it to its (embedded) dwelling and connect to its (process) IO.

Preparing the SD-card

Raspberry's OS and the file system usually reside on a µSD card – without the “µ” for very old Pis. New Pi OSs can also be on a SSD-drive; this features we did not (yet) use on our embedded systems. On one Pi4 installation we added a 2T SSD with an USB-adapter as extra storage for a SMB service (as a tiny NAS).

On embedded systems use a SD-card of suitable size and speed – scrapping here for a long running server application with logs and critical timing is wrong. On the other hand too large cards can be annoying when handling back up images. 16 GB (SanDisk Ultra e.g.) is most often OK.
Note **): 64GB with 200/140MB/s would take about 14 minutes. Slower cards get in range of hours.

To prepare Raspberry's OS and the file system we use rpi-imager.exe (figure 2) to choose a suitable Pi OS version, make the basic settings and write that image to the µSD card. Formatting will be done in the process. Just put the SD-card in an up-to-date USB-adapter and start the imager.



Fig. 2: Pi Imager

In the first steps we chose the Raspberry type, the OS and the card adapter’s drive, in our example Pi3b, Pi OS lite 64bit (Trixie) and the µSD card adapter’s drive letter. Double check that drive to be absolute sure it is the SD card in question and you don’t need any data there.
Hint: If you have both 32 and 64bit Pi OS you need at least two different C cross tool chains.
See “building for different target OS” as well as all warnings on trixie and Pi 5, page 36 ff, etc..
In the next step edit the settings: name, WLAN, user:password, SSH etc..

Having all checked and the appropriate settings, start the formatting and imaging. It will take time. When ready, un-mount the device, remove the card from the PC and put it in the target Pi.

But before going on much further it is essential to be able to handle saving and restoring our Pi’s SD cards. There are pitfalls.

SD-card: save and restore

For the basic installation we prepared the [µ]SD card with `rpi-imager.exe`. We inserted the card into the Pi, logged in and did the basic settings. We may start installing programs and transferring data files. All work – until now or further changes and updates – resides in the tiny SD card. If it is damaged electrically or crucial parts are damaged by a bad program all that work can be lost.

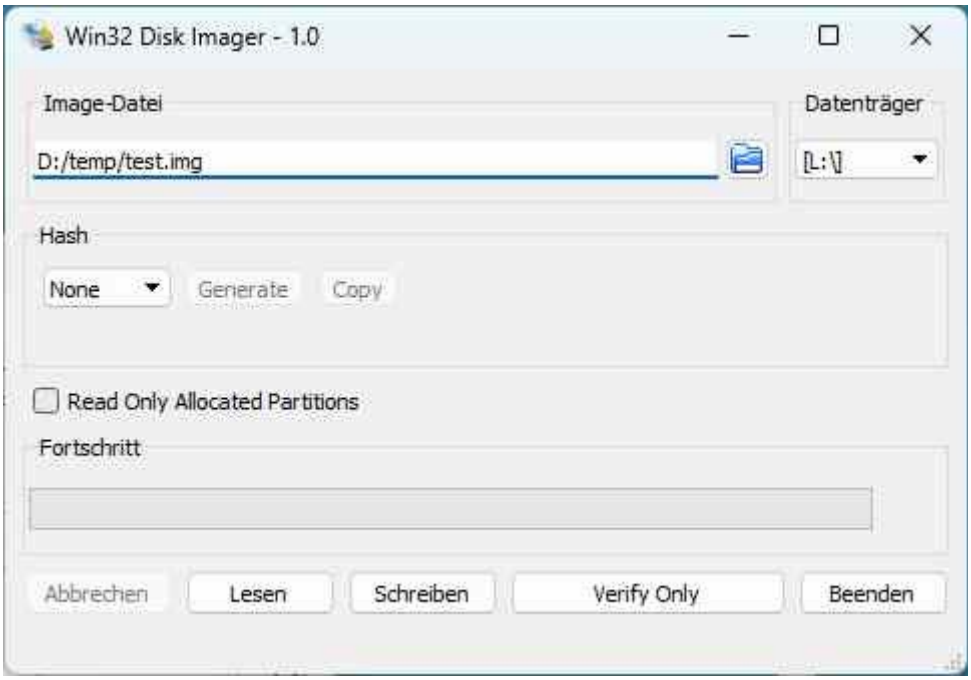


Fig. 3: Disk Imager

The only rescue can be saving total images in a (big) file on a PC or disk station and when needed from there on another SD card of equal size and speed. On a Windows PC we can do this interactively with the “Disc Imager” (figure 3): `C:\util\ImageWriter\Win32DiskImager.exe`

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

The same slot or an actual USB adapter as used with `c:\...\rpi-imager.exe` will do.

To save as .img file with Win32DiskImager chose a target file and click “Read” (German: “Lesen”), see figure 3. The target file will usually not exist, just chose an appropriate name.

The `Win32DiskImager.exe` works on Windows 11. But, alas, there are limitations:
- it must be started as admin! (One has to accept it.)

And occasionally – but then very annoying – we had the following effects
- it had to be shutdown before every next operation (even with the same card)
- sometimes the SD card must be present before the start
- typing or copying the file name, `D:\temp\test.img` in the example fig. 3, into the text field may fail. In that case click the blue folder icon, find the best fitting file in the opening explorer window and then edit the line “Image-Datei” (image file).

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

Warning: When hitting the **size problem** and ticking “write anyway” Win32DiskImager crashes leaving the target unusable. Even when reporting a normal end we had cards never booting in the Pi.

Caveats for Win32DiskImager:
- Be extremely careful with “Write”.
  Double check the drive letter. Win32DiskImager can kill your system (as admin).
- Be patient 64GByte µSD Read – Write – Verify may take 1h20min.

The size problem

Most probably, its cause is no two [µ]SD-card types having the same size.
- Do not go ahead when getting complaints on the operation planned.
  Click exit – especially when hitting the size problem, see Fig. 4.

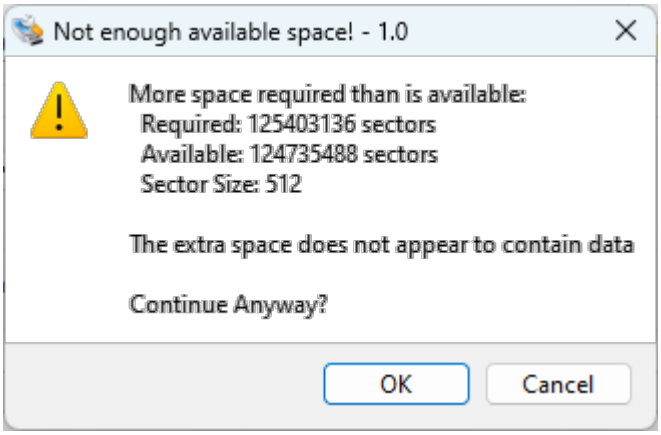


Fig. 4: The size problem

If you go ahead on a some percent size misfit the card will not work in the Pi even if “Verify” says OK. Do not believe in "The extra space does not appear to contain data" in Fig 4, even when here the 64GByte source µSD was 95% empty and the missing size is less than ½%.

For the size problem as such Win32DiskImager is not to be blamed. We used `dd` with the same cards on an Ubuntu laptop, it finished the read and write operations (taking ages) without crashing but complaining on the non fitting size afterwards [sic!]. The card didn’t work in the Pi either.
Here, Win32DiskImager would have saved you hours (when taking the warning seriously).

Writing an image file read from one SD card to another smaller one did always fail – at least we never experienced it otherwise.
A bigger size as destination will work. But you will not persevere with that approach in the long run.

Tip 1: If you want e.g. some µSD cards for saving **n** working states, and having **m** for experimental improvements and **r** for reserve, do have/buy **n+m+r** cards of same make and size. Requiring same speed was not necessary in out lot of SanDisk 32G, as all had 124735488 sectors (see Fig 4), We did a complete re-install as we could not get other 32G µSD cards with 125493136 sectors.
Tip 2: Choose a reasonable size but do not think too big (128G when 16G would be enough, e.g.). Consider also: Each SD card image file will be as large as the card with Win32DiskImager.

First commissioning and enabling ssh (remote shell)

These are first steps to bring in a new OS with a freshly prepared [µ]SD card:

- take the not powered target Raspberry and put the card in
- check your DNS server for your new Pi's IP addresses. In a distributed control system we strongly recommend fixed LAN and/or WLAN IP addresses. Tell your DNS server, router, “Fritzbox” … to do so:
- connect power (usually by USB 5V 2A..4A)
- fire up the remote console ssh.exe (not putty) command and connect to by

```
D:temp> ssh theOnlyUser@theNewPi
```

When `theNewPi` is not resolved by by your DNS (FritzBox) repair it or work around by using the IP. At first login from a PC ssh will complain on not knowing `theNewPi`.

```
The authenticity of host 'thenewpi (192.168.178.30)' can't be established.
ECDSA key fingerprint is SHA256:T5JGd/h3sPuseMuckelME.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Failed to add the host to known hosts list (/home/ego/.ssh/known_hosts).
```

If Windows complaints on a Linux type directory (`/home/user…`) for the list of known hosts you use a the wrong ssh.exe. The `where` command shows the candidates in PATH order.

```
D:\temp>where ssh
C:\util\msys32\usr\bin\ssh.exe
C:\Windows\System32\OpenSSH\ssh.exe
```

Here `msys32\usr\bin\ssh.exe` is the Linux type, wrong here but higher on the PATH as the fitting Windows one. Delete it or resort the PATH. Having reasons not to do either make in `c:\bat` (usually the highest in PATH) a file `c:\bat\ssh.bat` containing:

```
@C:\Windows\System32\OpenSSH\ssh.exe %*
```

Then this will run the fitting ssh.exe.

```
D:temp> ssh theOnlyUser@theNewPi
```

Note: If the machine or name was used before it might be necessary to delete its entry in `C:\Users\<name\.ssh>\known_hosts` and trust the machine on next log-in.

When all went well, you'll be prompted to log-in (with `user:password` you choose in the imager).

A note on password authentication

And this password authentication we do want! In our private networks this is no problem and some traditional tools and scripts rely on password authentication. But, alas, the newer the just installed Pi OS is the higher the chances are not to have password authentication:

Now you will see the name of a key pair and be asked for a password. In most cases it is not the one just set in Pi Imager – except by co-incidence. The key pair is one found on your PC and if you know it – i.e. the private key's password – you now may log in.

But in the light existing or coming scripts (often with another Pi user) sticking to key pair authentication often is no option. In our first experience with this well meant but dubious progress the key pair belonged to a GitHub account and I hadn't the slightest intention to use it for a Pi nor did I want the feature at all. The Raspberry Pi Imager should have an option to turn off this feature but v1.9.6 seems to have less options (?) than its predecessor.

To get rid of key pair authentication and back to password do edit the file `/etc/ssh/sshd_config` to have the following non-comment lines:

```
# This is the sshd server system-wide configuration file.
# See sshd_config(5) for more information.

# There are no such files at the moment. This is may be essential.
# Otherwise check all these files, too.
Include /etc/ssh/sshd_config.d/*.conf

# Authentication:
PermitRootLogin yes
PubkeyAuthentication no
PasswordAuthentication yes
KbdInteractiveAuthentication no
UsePAM yes
AcceptEnv LANG LC_* COLORTERM NO_COLOR
Subsystem       sftp    /usr/lib/openssh/sftp-server
```

If above changes in `/etc/ssh/sshd_config` did the job after several reboots or if using this log-in

```
D:\temp>ssh -o PreferredAuthentications=password
               -o PubkeyAuthentication=no   theOnlyUser@theNewPi
```

only once finally turned off the key pair log in, I don't know.
Hopefully future fresh installations will clear this and do not impose other unwanted features!

In case of success you have an empty working Pi with one user. *Congratulations.*
"Empty" in our case means just the programs and libraries belonging to the lite version chosen.

Final note: The problem won't occur if you choose "SSH", "password / Passwort" and NOT "key / Schlüssel" in Pi Imager v2.0.3 .

A Note on the nano editor

It may happen that nano's syntax highlighting will display some text items in almost the same colour and brightness as the background. As many text editor programmers the authors of nano libidinously include all parsers they can get hold of to add multi colour syntax highlighting at their gusto. Hint: When in trouble to read coloured text just start nano by

```
nano -Ynone myfile.txt
```

Have more than one user

Formerly Pi OSs came with a pre-defined user:password = "pi:raspberry". It is most strongly recommended not to use this combination known by all hackers. Cases were reported of well meaning software (updates) eliminating that user even when it had another password. If the murdered user was only one the machine is practically unusable. We did experience some losses of user pi. Fiddling with the µSD may get you a one time log in. If that does not work, one falls back to a saved image of the machine saved – or to a complete re-install. Due to those experiences we recommend additional users with sudo privileges to have still ssh and ftp:

```
sudo useradd -u 1001 -g 1000 -c "sweet's substitute" -m pius
sudo ls -la /home/pius
sudo passwd pius
sudo usermod -a -G sudo  pius
sudo usermod -a -G sweet pius
sudo usermod -a -G adm   pius
sudo usermod -a -G dialout pius
sudo usermod -a -G cdrom   pius
sudo usermod -a -G audio   pius
sudo usermod -a -G video   pius
sudo usermod -a -G plugdev pius
sudo usermod -a -G netdev  pius
sudo usermod -a -G games   pius
sudo usermod -a -G users   pius
sudo usermod -a -G input   pius
sudo usermod -a -G render  pius
sudo usermod -a -G netdev  pius
sudo usermod -a -G spi  pius
sudo usermod -a -G i2c  pius
sudo usermod -a -G gpio pius
id pius
id sweet
```

The last two commands allow an easy check of both users belonging to the same groups.

Note1: Traumatised by the dead of some user accounts "pi" we strongly recommend having two sudo users (sweet & pius not pi, e.g.). Feel free to add an extra standard user to play with.

Note2: If you have a second user and still have and need user pi we recommend renaming it as soon as possible or when an update disallowed logging in as pi.

Note3: Besides being better known to hackers than to new users, "raspberry" is and never was a good password. And avoid passwords (and perhaps names, too) typed on keys misplaced on US keyboards (like "y") nor special ones on German or French keyboards.

Enable WLAN – or disable it

The Pi3, Zero, 4 and 5 have WLAN on board, Pi1 has not. The build in WLAN's only disadvantage is the antenna being fixed to the board. Encasements with good protection, shielding and heatsinks may 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]. On the other hand, when having prepared headless WLAN access above in rpi-imager, no further action may be needed.

In other cases or problems a most useful command is listing visible WLANs by:
`sudo iwlist wlan0 scan`

In doubt if the WLAN is wlan0 use `ifconfig`. In our case it didn't see 5 GHz cells. The Pi3 (not 3+) supports 2.4 GHz only, even if the Broadcom chip should do both. From the `iwlist` scan output we chose a WLAN in acceptable quality (Micha42Netz in the example here) to connect to.

```
Cell 01 - Address: 5C:49:79:6C:03:81
                   Channel:11
                   Frequency:2.462 GHz (Channel 11)
                   Quality=42/70  Signal level=-68 dBm
                   Encryption key:on
                   ESSID:"Micha42Netz"
                   Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; .....
                   Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
                   Mode:Master
                   IE: IEEE 802.11i/WPA2 Version 1
                       Group Cipher : CCMP
                       Pairwise Ciphers (1) : CCMP
                       Authentication Suites (1) : PSK
```

`sudo ifconfig wlan0 down/up` disables/enables the WLAN. To disable it permanently put `@reboot sleep 10; ifconfig wlan0 down` into the crontab file (`sudo crontab -e`).

Caveat: Loosing the WLAN as only connection (no cable, Pi0 e.g.)
is as catastrophic as loosing the last/only user.

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 DHCP server distributes IPv4 to all. Add the lines

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

to `/etc/sysctl.conf` by `sudo nano /etc/sysctl.conf`.

Getting a decent directory listing

The commands dir, ls and even variants ls -la etc. produce unsatisfactory output to put it mildly. It's just ugly and information hiding. Hence type
`nano ~/.bash_aliases` and add the following three lines:

```
alias dir='ls -lAh --time-style=long-iso'
alias diR='ls -lARh --time-style=long-iso'
alias sudo='sudo '
```

Those lines could have been put into `.bashrc` also, but `.bash_aliases` can be copied or transferred to other user accounts with less risk of spoiling other individual settings, by e.g.

```
sudo cp -p ~/.bash_aliases  /home/pius/.bash_aliases
```

Changes in `.bash_aliases` and `.bashrc` will have an effect at next login.

Update the Pi OS

Having the very basic configurations done you should have a first thorough update:

```
sudo apt update  # <-\
sudo apt upgrade  #    \_ loop until stable
sudo apt autoremove
sudo apt dist-upgrade
```

After commissioning and running a productive 24/7 server dist-upgrade should be handled with care. Instead of `apt …` you may get additional results by `apt-get…` .

When getting a warning like

```
Key is stored in legacy trusted.gpg keyring (/etc/apt/trusted.gpg),
see the DEPRECATION section in apt-key(8) for details.
```

Read the chapter “The keyring problem” in an older version of this paper / report. The problem seems gone since years.

User interfaces

We can do real time process control with precise 1ms cycle time with a Raspberry Pi 3 and Raspbians / Pi OSs from Jessie on – as long as we do **not** install an OS with graphical HMI.

In other words: Take the lite variants for real time control! Only!

As long as we use the little machine as headless server all is well – and installing graphics spoils all real time cyclic process control endeavour. Linuxes on other OSs with graphics do this by dozens of all time busy extra services all running around. Adding graphics to a non graphical OS like Unix/Linux means adding a lot of foreign objects in various variants.

The standard remote access to our headless little machine is ssh.
And yes: It is possible to write a console program (to be started by and usable via ssh) that waits on a signal, gets data via shared memory and displays them on the console. (An example is hometersConsol.c.) It is also possible to get input from such console program and put the values and commands via shared memory to the process control application.

This "remote HMI by ssh as shared memory coupled application" works perfectly reliable – but looks as stone age as the nano editor. It is quite natural to want this functionality in a web interface, and this at least *) in the same local (W)LAN where the headless process control machine server can be “ssh”-consoled.
Note *): One should specify “only” instead of “at least” for safety and security.
Graphics just by being installed impedes even modest real time applications.
The detrimental effect works” without anybody doing anything locally nor X-remote.

On the other hand an Apache 2.4 with static web pages, pages with JavaScript and CGI, directory listings serving files has no [sic!] measurable detrimental effect on said real time performance. Your web design may even play with boxes, tables, pictures etc.. And we had no problems with 5 active web clients at the same time in the LAN or even remote via VPN. But one should not exaggerate the number of active web clients.

At first sight this may seem astonishing. But this approach, used judiciously, only transfers small amounts of data – partly static and cacheable – and puts all rendering and graphical toils to the workstation, laptop, tablet, telephone or whatever web-client. And in case of VPN the extra communication and encryption work is done by the router (Fritz!Box e.g.) and the remote client.

Hence, we use Apache 2.4 – on our process control Raspberries – for remote HMI and ssh for administrative work.

ssh – our remote console

We do not use putty since year for esthetic reasons. ssh’s presentation handling of nano etc. usually outperforms putty. With newer Pi OS putty seems not to work. No loss!

On Windows you can connect the command shell (usually cmd.exe) with a Pi by

```
ssh user@thePi && REM parameter is user at machine (name or IP)
```

Ending the ssh session with logout, exit or reboot returns to the Windows shell.

A disadvantage is the shell instance (cmd.exe) being the terminal and hence out of use until the session ends. On the other hand it is possible to scroll back to the closed “ssh terminals”.

In seldom cases you might also be annoyed by the following ssh behaviour: *)
1.) At every start it complains on not to be able to create directory /home/<winGuy>/.ssh.
2.) At every session start it asks for the acceptance of the Pi’s fingerprint.
Note *): This was explained shorter on page 7.

It seems ssh (most stupidly) is trying to store the Pi’s fingerprint on the Pi under the account of the windows user, not existing there. It’s totally crazy. By the way providing the wanted directory with all rights to everyone on the Pi is no good idea – and it does not help.
If, like me, haunted by that look for ssh.exe or just for ssh:

```
D:\temp>where ssh
C:\util\msys32\usr\bin\ssh.exe
C:\Windows\System32\OpenSSH\ssh.exe
```

This shows us having two ssh.exe, a Windows one and a ported but badly adapted Linux tool – the latter being on a better place on the PATH. This is for good reasons: We want some Linux tools have preference over Windows ones with the same name. Remedies are:

a) call C:\Windows\System32\OpenSSH\ssh.exe [parameters] directly every time
b) delete C:\Windows\System32\OpenSSH\ssh.exe
c) put the line @`C:\Windows\System32\OpenSSH\ssh.exe %*`
in a new file C:\bat\ssh.bat

I recommend (and took) solution c):

```
D:\eclipCe18-12WS\rasProject_01>type C:\bat\ssh.bat
@C:\Windows\System32\OpenSSH\ssh.exe %*
```

Now we see:

```
D:\temp>where ssh
c:\bat\ssh.bat
C:\util\msys32\usr\bin\ssh.exe
C:\Windows\System32\OpenSSH\ssh.exe
```

Log in by

```
ssh user@myPi
```

and logged in as (whatever) user after entering the password use the opportunity to create a directory /home/user/bin if not yet there.

```
sweet@meterPi:~ $ mkdir bin
sweet@meterPi:~ $ sudo mkdir /home/pius/bin
```

After the next reboot **bin** will be on the path and a dwelling for the user’s scripts and programs.

Apache 2.4

As said on our Pis a web server’s main purpose is to provide a HMI for process control, delivering log and documentation files. All this is fixed content set/changed via FTP.
For the HMI communication and (dynamic) process control pages we use CGI, AJAX, JASON and JavaScript (see below), but not PHP except perhaps in one homeopathic dose.

And we are on a protected, non public LAN. Hence, we need no domains, virtual hosts nor https. We address those Pis by their name or fixed IP address, provided by the own DNS / DHCP. Hence we can (should) live with a very simple Apache setting with generous access for and to Apache.

Installation

Apache 2.4 is (apt-get) provided in a reasonable way since the Raspian Stretch distribution. Installing Apache starts by

```
sudo apt update
sudo apt upgrade
sudo apt install apache2 -y
```

In this state (on success) the web server delivers only the “Apache2 Debian Default Page” It’s thee configuration 000-default; see /etc/apache2/sites-available/. According to our Pi’s name we want a site “meterpi” (en lieu de “default”. Change to your naming accordingly. We prepare it by:

```
dir /etc/apache2/sites-available/
sudo mkdir /var/www/meterpi
sudo chgrp sweet /var/www/meterpi
sudo chown sweet:sweet /var/www/meterpi
dir /var/www/
sudo chgrp sweet  /etc/apache2/sites-available/
sudo chmod g+rwx  /etc/apache2/sites-available/
dir /etc/apache2/
```

We left Apache’s main configuration file untouched, except stripping most of the original comments to get an overview:

```
#    /etc/apache2/apache2.conf   01.12.2025   Albrecht Weinert
# changes: strip most comments   see them in the original file
# This is Apache's main configuration file on meterpi

DefaultRuntimeDir ${APACHE_RUN_DIR}
PidFile ${APACHE_PID_FILE}
Timeout 300
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 5
User  ${APACHE_RUN_USER}
Group ${APACHE_RUN_GROUP}
HostnameLookups Off
ErrorLog ${APACHE_LOG_DIR}/error.log
LogLevel warn

IncludeOptional mods-enabled/*.load
IncludeOptional mods-enabled/*.conf
Include ports.conf

<Directory />
  Options FollowSymLinks
  AllowOverride None
  Require all denied
</Directory>

# /usr/share was omitted on most installations; try comment out/delete
<Directory /usr/share>
  AllowOverride None
  Require all granted
</Directory>
```

```
<Directory /var/www >
  AllowOverride All
  Require all granted
  Options +Indexes +FollowSymLinks
</Directory>
AccessFileName .htaccess
<FilesMatch "^\.ht">
  Require all denied
</FilesMatch>
LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" vhost_combined
LogFormat "%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" combined
LogFormat "%h %l %u %t \"%r\" %>s %O" common
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{User-agent}i" agent
IncludeOptional conf-enabled/*.conf
IncludeOptional sites-enabled/*.conf
```

The reason for this readable compactness will get clear in "**A cause of trouble**" below.

By

`nano  /etc/apache2/sites-available/meterpi.conf`

we made a site configuration file according to our Pi's name:

```
# /etc/apache2/sites-available/meterpi.conf 01.12.2025 18:51
# configuration for http://meterpi
<VirtualHost *:80>
  ServerAdmin webmaster@localhost
  DocumentRoot /var/www/meterpi
  CustomLog ${APACHE_LOG_DIR}/access.log combined
  HeaderName /logs/logsHeader.html
  ReadmeName /logs/logsFooter.html
## The next is is one compact line !!!
  IndexOptions SuppressHTMLPreamble FancyIndexing FoldersFirst
                                                  IconsAreLinks HTMLtable
  IndexIgnore logsHeader.html logsFooter.html
## may be omitted if (exactly so) in /etc/apache2/apache2.conf
 <Directory /var/www >
  AllowOverride All
  Require all granted
  Options +Indexes +FollowSymLinks
 </Directory>
##                    .<-- ~/var/www
##                   /    ## link to load site via user space
## /var/www/meterpi/index.html
##                 /server
##                /icons
##               /include
##              /cgi              .--> /var/log/apache2/
##             /logs/apache2 ___/   ## link to show Apache's logs, also
##             \..<-- ~/logs ## link to access (all) logs locally, too
```

```
 <Directory "var/www/meterpi/cgi >
  Options ExecCGI
  SetHandler cgi-script
 </Directory>

 <Directory /var/www/meterpi/logs >
  AllowOverride All
  Require all granted
  Options +Indexes +FollowSymLinks
 </Directory>

 Alias "/icons/" "/var/www/meterpi/icons/"
</VirtualHost>
```

Enable your and Apache's [sic!] access to it's log files by

```
sudo chmod -R a+r /var/log/apache2
sudo chmod a+x /var/log/apache2
mkdir /var/www/meterpi/logs
ln -s /var/log/apache2/ /var/www/meterpi/logs/apache2
```

And enable the site access from user space by

```
mkdir ~/var
ln -s /var/www/meterpi/ ~/var/www
```

Optionally make a local ~/logs by

```
ln -s /var/www/meterpi/logs logs
```

If you like to enjoy the "Apache2 Debian Default Page" also under http://meterpi/server/ do

```
mkdir /var/www/meterpi/server
cp /var/www/html/index.html
  /var/www/meterpi/server/debianApache.html
```

And, finally, we switch the Apache configuration by

```
sudo a2dissite 000-default
sudo a2ensite meterpi
```

To make configuration change work enter

```
sudo systemctl reload apache2
```

When all went well http://meterpi/server/debianApache.html displays Apache's welcome page and http://demopi/logs/apache2/ shows Apache's log files.

From this base you can start to use the web server for your applications. Nevertheless a bit of styling with icons and .htaccess files may improve the impression.

Here two .htaccess examples (shortened):

```
#  /var/www/meterpi/server/.htaccess
#  $Revision: 237 $ $Date: 2024-11-20 18:11:02 +0100 (Mi, 20 Nov 2024) $

IndexIgnore logsHeader.html  logsFooter.html
IndexOptions +SuppressHTMLPreamble

AddDescription "minimal server info by PHP" hello.php
AddDescription "show most things PHP knows" info.php
```

and

```
#  /var/www/meterpi/logs/.htaccess
#  $Revision: 237 $ $Date: 2024-11-20 18:31:02 +0100 (Mi, 20 Nov 2024) $
IndexIgnore logsHeader.html  logsFooter.html
IndexOptions +SuppressHTMLPreamble
ReadmeName /logs/logsFooter.html
HeaderName /logs/logsHeader.html
AddDescription "Web server log files" apache2
AddDescription "page view log" apache2/access.log*      ## Note above
AddDescription "http server errors" apache2/error.log*  ## symlinks
AddDescription "Ereignisse und Werte" .txt ### &c. pp. **)

## ... and more
```

Note **): If you have our make apparatus you have all site (here meterpi) and common configuration and content files in your process control Ecplipse project and transfer these by:

`make PROGRAM=www TARGET=meterPi progapp`

Here www is a special pseudo program not do be compiled by gcc having .html, ..conf, .php, .jpg etc. pp. files in a tree ready to be transferred (as would be compiled binaries).

As we use the web server for process observation and control we like to see all relevant logs by http://meterpi/logs and in the directories shown there.

With the simple configuration indicated here we got everything working, look and feel like it did for about 6 years in a 24/7 process control job.

Alas, this a long story told from from its short (happy) end.

We had started as we did many times and with great success before: We just copied all our Apache data and configuration files from an Eclipse project handling common and target specific files using and automated building and transferring applications and web content to a divers flock of Raspberries – and expected all or at least almost all would to work, also with Apache.

Well – almost nothing did.

A cause of trouble

In the previous Installation (Raspbian Stretch/Buster running 24/7 for some years) we had ~`/var/www` as document root and all automated update activities are programmed to manipulate this document tree. (~/ was `/home/pi` and is `/home/sweet/` now.) The automated updates respectively first provisions worked at once but the web server did not.

Apache seemed not to read most of its configured directories and not to follow symbolic links. Most .conf files and all .htaccess had no effect as one ugly directory listing demonstrated.

Internet research brought no clear causes or remedies. Some recipes brought nasty side effects but not the healing. There were speculations on new options like **SymLinksIfOwnerMatch** which should be turned on instead of **FollowSymLinks**. Most tips didn't help. And nothing trustworthy on the causes of the new troubles was found. The only clear web search result was:

- There were many fellow sufferers.
- From Jessie + then Apache to Bookworm + current Apache
  Apache's behaviour seemed to have changed drastically.

Nightmarish hours of fruitless experiments and refined searches gave this impression:

- Apache does its start up work as root and gets its basic files as such. (Not new, never mind).
- Apache delivers web content and reads the relevant files as user www-data. (Not new, too)
- Apaches reading of these files is restricted to `/var/www` (+ subdirs).
  and to `/var/log` (+ subdirs)

- The second one only worked after putting user www-data additionally in the group `adm` by
  `sudo adduser www-data adm`
- The access to those (two) directory trees can be further restricted by file access rights and directives – but never ever expanded.

These restrictions and perhaps more are obviously hardcoded in Bookworm or Apache. And, alas, there is no option `+OneSiteClosedShopOnlyFewTrustedUsers` to bring back the old freedom. One rule is

- When the user www-data can't enter a directory or can't read a file forget all fiddling with directives, symbolic links etc. This file will never be part of your web site.

The accesses needed, wanted, planned shall – if in doubt – be tested as user www-data.

- If a test fails in spite of sufficient access rights for www-data or its groups, abandon all hope and change your file/directory structure.

For those tests you need no running Apache server, it might well be stopped.
But I had to do something absurd for those tests:

- log in as user www-data

I never ever thought of it – and others didn't either and it never was provided:
www-data has no (known) password and no shell. To mimic it by

```
sudo su www-data
```

is not sufficient. You might either give www-data a password plus a shell. This I did not fearing side effects. The other way to the shell needed are su options.

```
sudo su -l www-data -s /bin/bash
```

will do the trick. And to repeat the important rule

- If you as this user www-data can't read a file needed for your web site and if you can't change that quickly, easily and without unwanted side effects move that file elsewhere.

My new "world view" on Apache and said rules lead to a – quite simple in my humble opinion – structure and configuration files described above. This "world view", gained by painful experience, may partly be not correct or missing important aspects. Corrections are welcome.

> But, as a hypothesis, it did work very well – from the advent of Apache's changed behaviour until now.

Final remark: Why did we install Apache first?

The web server is our / **the HMI** on a **headless** Pi. You can read log files, documentation, user guides and more. With a little bit of PHP with strict restrictions (see next chapter), and / or JavaScript and CGI you can get and display non static information. So we implement process observation and control (German B&B: Bedienen und Beobachten).

As will be presented below and is backed up by extensive experiences a headless Pi can handle real time process control including 1 ms cycles. This is in no way impaired by an Apache delivering static pages even when flavoured with CGI.

And be quite sure: Displays, pointing devices, mouses, touch screens are the death of real time on a little Pi.

Hint: As process IO (also / mainly) with GPIO is our goal and we strongly recommend pigpiod it might now be the time to install it. If this would fail in your Pi + Pi OS combination you might be back at the very start: putting another Pi OS version on an µSDcard. Hence it might be wise to work through the chapter "The pigpio(d) library" on page 28, as well as testing the "Watchdog" (page 66) soon – or even now before installing PHP8.

Installing PHP 8

In our process control projects we did all server side dynamic content generation, including AJAX with GCI and used C for GCI programs. Doing so, one stays consistently in one language on cross compiling workstations with IDEs etc.

Hence we never (**!**) use PHP for its original purpose of generating complex pages from databases on embedded Pis.

But for completeness and for tiny task as displaying the Pi's name by the very same static html page used on several Pis (see below) we sometimes use half a line of PHP.

On trixie – no watchdog, hence not (**!**) recommended – do:

```
sudo apt install php8.4 libapache2-mod-php8.4
```

Up to bookworm (or if 8.4 is not working) try 8.2 or do:

```
sudo apt install php libapache2-mod-php php-cli php-common
     php-mysql php-gd php-curl php-xml php-mbstring  ## all in one line
```

Check the PHP installation by

```
php -v
```

```
PHP 8.2.29 (cli) (built: Jul  3 2025 16:16:05) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.2.29, Copyright (c) Zend Technologies
    with Zend OPcache v8.2.29, Copyright (c), by Zend Technologies
```

To get Apache acquainted with PHP we made (by nano) a file
/var/www/meterpi/server/hello.php :

```
<?php
 echo "The server <b>", gethostname(), "</b> is online at ";
 echo date("H:i:s"), " (UTC) on ", date("Y-m-d") ,".<br /><hr />";
 echo "<small>(A. Weinert, 26.10.2024)</small>";
?>
```

It will show the Pi's name and date and time (UTC not local) by http://meterpi/server/hello.php :

> The server **meterPi** is online at 16:38:13 (UTC) on 2025-12-03.
> ———————————————————————————
> (A. Weinert, 26.10.2024)

Hint: If the PHP source code is displayed, shift re-load the page.

Pi's name and date and time (UTC not local) are server site data here put on the html page by Apache via PHP at every (re-) load. Loading this page is as fast as a purely static one and hurting a real time process was never observed by us. But . . .
Now make a small add-on to `hello.php` as `/var/www/meterpi/server/info.php` :

```
<?php
 echo "The server <b>", gethostname(), "</b> is online at ";
 echo date("H:i:s"), " on ", date("Y-m-d") ,".<br /><hr /><br />";
 phpinfo();
 echo "You are on <b>", gethostname(), "</b>.";
 echo "       <small>(A. Weinert, 16.11.2024)</small>";
?>
```

Browsing to this file by http://meterpi/server/info.php provides a lot of information – more as one should show to everybody or as can be printed here. On older PHP and Apache we saw first loading times of 40s giving the impression of a dead web server. 8.4 was fast from start.

But anyway: Be very cautious with server side work on your tiny "Apache on real time Pi" site!

Enabling file access on our workstation

For administrative tasks and programming remote file access from workstations is essential. On headless / GUI-less systems, you'll get decent editors on your workstation for files on the Pis and (on Windows only) an operational clipboard handling worth the name.

Most Linux distributions for Raspberry, like Raspbian, will not have a FTP server. But almost all will have SSH. This offers remote file access by protocol sftp. SSHFS, FileZilla and WinSCP *) can handle that as clients. Hence, **no FTP server** installation on Pi Os will be needed.
Note *: Windows' ftp.exe can't – but on Windows you may have the excellent WinSCP.

SFTP – FileZilla

You really should have FileZilla on your Ubuntu and Windows workstations. (You may even have FileZilla on a graphical Raspbian.) On a Linux PCs – **n.b. not on your Pi** – install it by:

```
sudo apt-get install filezilla
```

On Windows download and install the newest Filezilla.

Now set up FileZilla as client: Open FileZilla, go to site manager, make a new site and name it accordingly, say “raspi61”.

In general setting do: host = 192.168.89.61; protocol: sftp login: normal, sweet, 0123.
In “advanced” tick “bypass proxy”, when your workstation and Pis share a local network.

That's it. Connect should work.

Tip: If getting “connection refused” and if all else – WinSCP, SFTP,.. – works with your Pi but not Fillezilla, try
server manager → … → Advanced → Server type = Unix instead of “autodetect (default)”.
This made Fillezilla show fingerprints and ask if OK instead just refuse the connection.

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/sweet/

The last point allows to comfortably view /etc/apt/sources.list (with Editpad on Windows) and copy its content without ado by clipboard.

Modifications in the editor on the (automatic) local copy will be possible, but in 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 a comfortable environment.

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

SFTP – WinSCP

WinSCP is one of the best FTP client programs – not for giving us the n + 21st graphical FTP but as a most powerful command line program 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   ; console, only, recommended
19.04.2017  14:44    18.905.808   WinSCP.exe.. ; console and GUI
29.05.2017  18:40        14.497   WinSCP.ini
```

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

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

```
# transfer two programs to the target machine
# Copyright 2017 Albrecht Weinert      weinert-automation.de
# $Revision: 2 $ ($Date: 2017-05-29 18:37:55 +0200 (Mo, 29 Mai 2017) $)
open sftp://sweet:0123@192.168.89.67
cd bin
option batch continue
option confirm off
put gnBlinkSimple -preservetime -permissions=775
put rdGnSimpleBlink -preservetime -permissions=755
exit
```

Listing 1: WinSCP script “progTransWin” for batch transfer of two programs.

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 programs. 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.

On Windows 10 / 11 a single file may be transferred by command line:

```
scp notes.txt sweet@pi61:/home/sweet/notes/
```

As there’s no option to give scp the password that command is hardly usable for batch processing. But therefore we are quite happy with WinSCP scritps.

Pi5 warning regarding pigpio and BCM2835 libraries

In all our Pi process control applications we used the piogpio(d) library ([61]..[63]) since many years on all Pi models 0 to 4. Our good reasons see at at the beginning of Part II “I/O, page 28.

But be warned: As of now (2025, [70]) there is no pigpio library for the Pi5, and probably won’t ever be.

The cause are significant incompatibilities mainly in the IO hardware and addressing between Pi5 and all others. pigpio for a Pi5 will never be compiled from just the same C source. It would be a separate branch and, worse, it might not be fully compatible. In the end all previous IO software is broken.
C cross build toolchains, working for all other Pis are broken for IO and also for just “Hello world”.

In a sense the current (?) lack of a Pi5 library won’t hurt: Using an expensive and power hungry Pi5 for (embedded) control applications is seldom a good choice. Usually, even the Pi4 is overkill here.

Nevertheless, the lack of a pigpio(d) library for Pi5 and its extensive incompatibility is a severe setback, to put it very mildly.

Rudimentary tests of binary process IO attached to Pi5 can be done by pinctrl (included in Pi OS).
For example

```
 pinctrl set 7,8,25 op && pinctrl set 7,8,25 dh
```

e.g., turns GPIOs 25, 8 and 7 (being the even pins 22, 24, 26) ON as output.

Be also warned, again: Refrain from trixie, if you intent to use the watchdog (page 66) for your main 24/7 control application.

Cross-compile C for Raspberry from a powerful workstation

In the very basic example (Listing 2) we use the piogpio(d) library ([61] ..[63]). The reasons for it and the installation are described at the beginning of Part II “I/O and process communication”, page 28, where it belongs. Install it now or skip the example until done.

```
/**   (Listing 2)         \file gnBlinkSimple.c       (2024-11-12)
 A very first program for Raspberry's GPIO pins
 Copyright (c) 2024  Albrecht Weinert
 weinert-automation.de      a-weinert.de
 This is a most simple GPIO pin output program using pigpio(d).
 Compile on Pi by:  g++ gnBlinkSimple.c -o gnBlinkSimple  -lpigpiod_if2
        and run by:  ./gnBlinkSimple
 Compile, build and transfer on Windows by:
 arm-linux-gnueabihf-gcc -I./include  -c -o gnBlinkSimple.o gnBlinkSimple.c
 arm-linux-gnueabihf-gcc -I. gnBlinkSimple.o --output gnBlinkSimple.elf -lpigpiod_if2 -lpthread
 cp gnBlinkSimple.elf gnBlinkSimple
 winscp.com /script=progTransWin /parameter sweet:0123  targetPi bin gnBlinkSimple
*/
#include <stdio.h>
#include <pigpio.h>
#include <pigpiod_if2.h> // set_pull_up_down set_mode gpio_write
#include <unistd.h>

// 40 pin con | GPIO number excerpt from include/arch/config_raspberry_03.h
#define PIN37   26

#define LEDgn PIN37   // LED gn 1 active is attached to Pin 37
#define ON  1  //  true On  An  marche go.  // exc. from include/baseTyCo.h
#define OFF 0  // false Off Aus arret  stop halt.
#define uMS 1000 // 1ms is 1000 us
int thePi; // the Pi handle for pigpoid

int main(int argc, char* argv[]){
  printf(
   "\n                Blink green LED    on Pin 37 \n"
     "gnBlinkSimple.c  R.01 12.12.2024    Albrecht Weinert \n\n");
  int blinkS = 3600; //   blink for 1h

  thePi = pigpio_start(NULL, NULL); // connect to pigpio daemon(local, 8888)
  if (thePi < 0) { // initialise pigpio  daemon failed
      printf(
    "\ncan't initialise IO handling  (piGpioD) \n"
      "gnBlinkSimple.c  Error / return code 99 \n\n");
      return 99;
  } // can't initialise piGpioD (the essential gpIO library)

  set_mode(thePi, LEDgn, PI_OUTPUT);
  while(--blinkS) {            // green
    gpio_write(thePi, LEDgn, ON);
    usleep(500 * uMS); //  500ms gn
    gpio_write(thePi, LEDgn, OFF);
    usleep(500 * uMS); //  500ms dark
```

```
  } // while endless                1 s loop
  printf(
   "\nstopped blinking the LED on Pin 37 \n"
    "gnBlinkSimple.c end (return code 0) \n\n");
  return 0;
} // main()
```

Listing 2: gnBlinkSimple.c

Also with a headless Pi, it is possible to transfer C-sources of GPIO programs (like the example gnBlinkSimple above, listing 2, page 21) to the Pi or even type/edit them there by putty/ssh & nano. Then one can translate and build them on the Pi with command (given also in the comment):

```
g++ gnBlinkSimple.c -o gnBlinkSimple  -lpigpiod_if2 -lpthread
```

Transfer the executable binary to a directory in the path (~/bin e.g.) and run it. This gives a first impression and shows a base for going 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 best of them).
- Or when adding the GUI, only few would like to handle IDEs, version control, documentation generators and all else on a Raspberry (besides from having killed even modest real time capabilities by adding GUI).
- 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. Therefore we download a “Prebuilt Windows Toolchain for Raspberry P” from http://gnutoolchains.com/raspberry/, e.g.:

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

Run this (or a newer version) to

- install at C:\util\WinRaspi
- 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
  It makes sense, even if no one likes extending the PATH by every install.

To test this best make an empty directory,

- have our .C source (gnBlinkSimple.c, listing 2) in a working directory and
- cd there.

Then use the Windows commands given in the listing. Hint: In such “one source file only” case without need for extra information compile and build may be shortened to:

```
arm-linux-gnueabihf-g++.exe  gnBlinkSimple.c  -o gnBlinkSimple
           -lpigpiod_if2 -lpthread
```

You'll get the compiled and linked runnable `gnBlinkSimple` in no time.
Transfer it to the Raspberry, cd there, make it executable if not so and start by

```
chmod 755  gnBlinkSimple
gnBlinkSimple &
```

It works! A program made on a Windows PC does GPIO on a Raspberry. Well, it just blinks one LED for one hour – but this is the start.

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

Note on the prefixes arm-linux-gnueabihf- **and** aarch64-linux-**:**

These confusing prefixes are used for cross tool-chains: They are unique prefixes for the target processor (architecture) and specific library sets. When setting up a new cross compile project in your GCC enabled Eclipse 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 32 bit Raspbians and `C:\util\winRaspi64\bin` and "`aarch64-linux-`", for 64 bit Pi OSs / Raspbians. With Eclipse and make you may handle both on your Windows workstation.

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 tool chain we should also have a make file understanding at least make clean and make all. See more in chapter – make project (p. 24).

A note on Windows:

Evidently, we prefer Windows on powerful development workstations – with Java, Eclipse, OpenOffice, GNU tools, SVN etc.. We have all liberty of open tools, while enjoying Windows' comfort and professionalism: domain and network file system integration, decent powerful explorer (not changing its name with every upgrade) with tool integration e.g. for SVN (tortoise), decent text editors (EditpadLite, notepad++), common clipboard support and so on. And (almost) all just works fine.

With Ubuntu, more than once, dragging files to shells suddenly stopped them working or changed its behaviour. Unclear, pure text or no clipboard support is a good recipe to drive Linux users mad. Experiencing regular total crashes or loosing tools on upgrades of Ubuntu and derivatives, one is, alas, constantly shooed back to Windows.

Well this happiness with Windows plus open tools did last up to Windows 7 professional respectively Windows Server 2008 R2 enterprise. Windows 10, at its beginning, brought first disappointments: It was slow (on hardware very happy with its predecessors) and at first unreliable.

The less controllable updates / upgrades did render installed tools inoperable. Nevertheless, even when finally forced to "upgrade" to W10 professional and other hardware, we stayed there. The good news is: All our tools (Eclipse, Gnu, SVN, Git, Java, doxygen, AVR, …) were kept working almost always by just moving all binaries to the new Windows installation*). And even the Explorer was still SVN aware **). In between, we accepted the impertinence of being forced to Windows 11 and partly new hardware and licences, again.

On the other hand, almost all what we describe for Windows can be (and was) done on a powerful Ubuntu workstation, too. And yes: Most Windows have their "drive nuts" potential, for example, the dangerous faking directory and file names, when accessing the file system graphically, like showing the fake name like "Programme" (German) for "Program Files", which won't work in tools and scripts – not to mention the stupidity of having blanks in system file names.

But the comfort, stability and usability balance is positive. Hence use Windows, Ubuntu or what ever you like as your cross-development platform.

Note *): Those cross toolchains portable as binaries to other Windows worked for Pi4 over the years as well as for all other Pis (0..3). The only exception is the Pi5 (see p. 21).

Note **): If the SVN icons in the explorer are gone type `regedit.exe` .
Goto `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers`
and kill all entries not needed – in my case all not beginning with Tortoise.

Background: On updates Windows has the habit of floating this node with icons for applications the user should install and surrender to – but stubbornly will not. Additionally there is a low limit of the number of entries the explorer will consider when building a file list.

Eclipse

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

Put your sources, like listing 2, page 21, for example in Eclipse's project folder. Best make the folder and the copies before making the new cross-C project. Set tool path and a tool prefix as said above 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 notoriously fail. Before stepping into (great!) configuration trouble to get Eclipse's taste of `make` to work, drop the generated make files and write an own old fashioned makefile which Eclipse would use with targets all and clean. Those targets you can use without Eclipse on the command line and scripts. And you can add more targets like "progapp" to organise the transfer of binaries and web content to the Pis.

Thereby you get

- \+ immense flexibility considering targets, devices and else which in the end will often be needed,
- \+ building and cleaning 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.

Command line – make project

We use one main `makefile` plus about 50 (make) include files describing the handling of the target machine `makeTarg_DNSname_settings.mk`. The target file names among other things the Pi type leading to including the fitting `make_raspberry_piType_settings.mk` file. Finally, the program to be build and uploaded is described in an include `makeProg_programName_settings.mk`.

```
08.01.2025  18:51        16.992 makefile                  ..... Listing 3
18.12.2024  12:27         1.450 makeTarg_meterPi_settings.mk
07.01.2025  19:09         1.537 makeTarg_pi4Ast_settings.mk
18.12.2024  12:27         1.481 makeTarg_pi4cam_settings.mk
18.12.2024  12:27         1.128 make_raspberry_00_settings.mk
18.12.2024  12:27           909 make_raspberry_01_settings.mk
18.12.2024  12:27           896 make_raspberry_02_settings.mk
18.12.2024  12:27           893 make_raspberry_03_settings.mk
18.12.2024  12:27         1.123 make_raspberry_04_settings.mk
18.12.2024  12:27         1.163 make_raspberry_05_settings.mk
18.12.2024  12:27         1.096 makeProg_gnBlinkSimple_settings.mk
18.12.2024  12:27         1.505 makeProg_growattReadSimple_settings.mk
18.12.2024  12:27         1.691 makeProg_growattRead_settings.mk
18.12.2024  12:27         1.089 makeProg_helloCrossWorld_settings.mk
18.12.2024  12:27         1.428 makeProg_homeDoorPhone_settings.mk
18.12.2024  12:27           992 makeProg_hometersConsol_settings.mk
18.12.2024  12:27         1.889 makeProg_hometersControl_settings.mk
18.12.2024  12:27         1.409 makeProg_www_settings.mk
18.12.2024  12:27           656 progTransWin              ..... and some more
```

Listing 3: Some make files (sources on demand)

By this mechanism you may compile and build a program by, e.g.

```
make PROGRAM=growattReadSimple TARGET=growPi  clean  all
```

and upload it to the target machine by, e.g.

```
make PROGRAM=growattReadSimple TARGET=growPi  progapp
```

The latter command would compile and build also if not yet done.

The non existent pseudo program www, respectively `makeProg_www_settings.mk`, would upload the web files for the Pi's Apache web server.

```
make PROGRAM=www TARGET=growPi  progapp
```

With all sources in a wide Eclipse screen and this make mechanism you get very fast turn around cycles when developing Pi programs.

Another example, including some of the help commands:

```
make help
make help_comm
make clean  all
make PROGRAM=gnBlinkSimple  clean  all
make PROGRAM=gnBlinkSimple TARGET=thePi  progapp
```

The last two commands generate our little Raspberry IO program gnBlinkSimple.
And by the command displayed by the last make command, we do the transfer.

```
winscp.com /script=progTransWin /parameter sweet:0123
  192.168.178.87 bin gnBlinkSimple
```

As already said: We stay with the "old" command line make and do not let Eclipse take over with Eclipse make projects.

Eclipse – troubles and hints

SVN client chaos

With development projects we still prefer Subversion (SVN) over Git in most cases. On commit you can get any plain text file (like a source) marked with the revision and date of this individual file's last changes. You won't get this feature with git push.

One trouble with Eclipse is putting in a decent SVN client. SubClipse made the least trouble. And there were problems of different client versions with in-compatibly different structures 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, see "our configuration". Fortunately, in the last years those compatibility troubles vanished. May be, the success of Git brought the yearly changes in SVN protocols and internal data structures to a welcome rest. Anyway, if you have a stable compatible state between all your SVN servers and clients – do not change a thing.

Our current configuration:

- eclipse.exe Eclipse IDE for C/C++ Developers Version: 2018-12 (4.10.0) [sic!]
- Subclipse 1.10.13 + Subversion Client Adapter 1.10.3 tigris.org
- svn.exe SVN 1.8.10-SlikSvn-1.8.10-X64 Aug 2014
- dto. Tortoise svn, version 1.14.3 (r1914484) Dec. 2023
- arm-linux-gnueabihf-gcc.exe (GCC) 4.9.2
- make.exe GNU Make 4.3 Built for i686-pc-msys (2020)
- doxygen.exe 1.9.5 (2f6875a5ca481a69a6f32650c77a667f87d25e88)
- pdflatex.exe pdfTeX 3.141592653-2.6-1.40.26 (TeX Live 2024)

Not young any more, but … "running" – and running well.

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 a usable result – Eclipse marks the sources with non-existent errors and warnings and loads of red and yellow marks …. This problem haunted us for a long while in the past but went away beginning with Eclipse CDT IDE Oxygen, 4.7.0, June 2017. If still affected read the subchapter corresponding to this one in [33].

The tip “do not let Eclipse make your make files” seems still valid. And additionally: Make your main “makefile” having valid defaults for all parameters. It must build the main or the actually worked on part of your project without extra parameters. And having that in your own hand getting multiple cross tool chains for different target Pis working is no vodoo.

But before getting too enthusiastic on Oxygen: For the AVR projects in the same workspace updating to Oxygen (with AVR plug-in) led to a catastrophic failure with > 1000 false errors.

Note 1: To emphasise the last point, projects used successfully for years got a useless IDE by just upgrading Eclipse. Eclipse versions good for one target may fail on others.

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 less than secondary.

Cross tooling summary

Now we can successfully

- cross-compile/cross-develop with
  - GNU-tools,
  - using Eclipse (Oxygen) with make, or
  - make (alone / on shell or automated batch)
- bringing all under version control, here SVN
- having all on Windows (or almost all on Ubuntu)

for our Raspberries and even

- upload the program just build from Windows to the Raspberry, almost *) automated by make.

So, we can go on to utilise Raspberry Pis in a professional development environment.

Note *) on almost:
At any point in past `make.exe` stopped being able to run `winscp.com` scripts. That is probably not `make`’s fault but a security “improvement” of `winscp.com` which slipped through our precaution “Never change a running programme”. The only remedy was: Let `make` print instead of execute the command and copy it to `cmd.exe` by hand.
Tips to get rid of this disability welcome.

Later – update, upgrade and beyond

As will be said many times below, also, before installing new software we do the usual update/upgrade exercise:

```
sudo apt-get update
sudo apt-get upgrade
```

Or by using the more modern apt (in normal use cases no difference) we do:

```
sudo apt update
sudo apt upgrade
```

The subtle differences between apt-get and apt can be found elsewhere [https://www.debian.org/doc/manuals/debian-faq/pkgtools.en.html]. In short: On the console use apt and be careful to change the choice in scripts. We’ve seen cases when an update upgrade loop with apt stopped changes apt-get did some (or the other way round). But that seemed past.

To update the version of the current Pi OS or Raspian distribution, Bullseye e.g., do:

`sudo apt full-upgrade` or `sudo apt-get dist-upgrade.`

Here some care is recommended, like saving a copy of the µSD-card before. Packages may be de-installed in the process and your application or the Pi's communication may not work afterwards.

The commands `full-upgrade` or `dist-upgrade` alone won't upgrade the underlying Debian version, from Buster to Bookworm e.g.. The canonical way to do this is using the pi-imager (V1.9.6) to install the new Raspbian from scratch. Of cause, you must install all your applications, settings users etc. afterwards.

If you won't or can't do this you may

A stick to your current Debian version or
B resort to one of the recipes found in the Internet to update the distribution on the Pi

If your (embedded) Pi does its work (24/7) to your complete satisfaction **A** would be the best. But after a while you won't get normal updates/upgrades any more as now with Jessie. Updating installed applications and installing new ones may then fail, also.

In that situation we tried the "recipes" **B**. All procedures start by changing some system files (to name the source of the new distribution). Then you have to follow a strict choreography of commands some of which ask cryptic questions. Some warn you not to leave the keyboard no matter how long it takes.

Most "recipes" **B** are time consuming and some fail totally. Finally, one seemed to work including the applications but some communication settings failed or kept changing in a malicious way rendering the process control application permanently unusable.

Hence, when having 24/7 embedded applications and **A** starts to fail and the well chosen recipe **B** fails on the first attempt **do a clean new installation**. And as the best 24/7 embedded real time control systems can totally fail by hardware damage be always prepared to build it new or have a spare (cold redundancy) ready.

We had to do it lastly for a Pi3b controller bringing it from 32 bit Jessie to 64 bit Trixie by a new installation. It was easier and faster than expected. Having the old installation's recipe / installation diary documented does help.

Part I's results

We can install a usable OS and tools on Raspberries. We provided all basic tools and set up (human) communication and file transfer. We can save and restore the current complete µSD card state of our "little servers". And we know how not to run into the dreadful "size problem".

The basic installation gave us a Pi with fixed IP address(es) we can handle, control and observe remotely. We may have put it to its destined location and/or connected it to its (process) IO.

Hence we could do the "program – install – test" cycles to get it to its "real" work.

Therefore we will establish the means for I/O and process communication in Part II. The most basic one will be the using of general purpose IO pins (GPIO) of our Raspberry. We may have prepared this already for sake of the very basic example gnBlinkSimple (listing 2, page 21).

Otherwise let's do it now and perhaps come back to gnBlinkSimple.c.
Note: This example is in Part I instead of II to demonstrate local and cross building.

Experience with our zoo of embedded Pis showed that having very similar and good

- accesses
- drivers & libraries
- tools / services
- look and feel

is a great help in the long run. Doing the basics right and re-usable pays.

P A R T I I I/O and process communication

The pigpio(d) library

As said above, pigpio(d) is our preferred approach to Raspberry IO and the only one we use for real control applications – except on a Pi5, see the warning on page 21.

Other IO libraries considered before do burden the production code with the trouble to
- initialise the GPIO (memory) usage,
- adapt to changing (virtual) addresses as well as
- the fighting with access rights

or in a certain sense worse
- giving users the rights for standard IO with all libraries and approaches.

No decent OS offering any protection will let user code do IO, as did early Raspbians by requiring sudo to access GPIO. In between the standard GPIO usages (read write) can be made accessible without sudo, but more settings or alternate functions may not.
Note: Standard IO without sudo works, when a user is in group gpio.

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

- socket as in basic example gnBlinkSimple, listing 2, page 21 and as in all our control applications or by
- pipe never used here.

Both approaches need no sudo. In the case of socket the control program and the GPIO pins may be on different Raspberries on the same network or one control program can use multiple Pi’s IO.

To tell it the other way round, the pigiod daemon or server can be used remotely or locally by other programs using the socket or pipe approach. And, not really surprising after all, pigpio forces its daemon's singleton property. No program or operator can start another instance on the same machine when the daemon (pigpiod) is running, already.

Even when a pigpio daemon (piGpioD) is forced to be a singleton, nevertheless, one should make program's usage of pigpiod singleton, too. And, when using only piGpioD, one should exclude all other approaches from sudo-less IO if not yet done so. This usually means removing standard users from the group gpio by

```
sudo deluser pius gpio
```

But now let's go on: 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 #  make the library    see note *)
```

Note *): If the installation fails on python check its version and install the fitting distutils. If, e.g., `python3 --version` shows 3.92, do `sudo apt install python3.9-distutils`
For Trixie (not recommended) above may not work nor enable the pigpiod installation. The try

```
cd ~/
sudo rm -r ~/pigpio-master
rm master.zip
sudo apt install -y python3-setuptools python3-full
sudo apt install python-setuptools python3-setuptools
```

and start the installations and tests above again with

```
wget https://github.com/joan2937/pigpio/archive/master.zip
```

After the installation do all the tests provided **) as recommended in [61] by

```
sudo ./x_pigpio # check C I/F  see note **)
sudo pigpiod    # start daemon with default sample rate 5 µs etc.
./x_pigpiod_if2 # check C      I/F to daemon
./x_pigpio.py   # check Python I/F to daemon
./x_pigs        # check pigs   I/F to daemon (1 fail, Cbs, wavef.)
./x_pipe        # check pipe   I/F to daemon
cd ~            # ready return to user's directory
```

Note **): Put a 1-active LED at pin 22 = GPIO25 to see some test steps working. And see also that the tests forget to set pin 22 inactive (by making it an input). The LED stays on. In one case we had a relay at pin 22. It was not happy. But the test was well visible and audible.

As said in [61]: If few tests fail and many more are passed; this is OK. We saw some irrelevant test fails on Pi3s with Jessy, Stretch but with later OS on several Pi types4 (almost) all tests pass.
N.b.: With 10 µs sample rate some test will will fail saying “half the expected number of yxz”.

And when using piGpioD on a Raspberry it's recommendable to start it at boot. Best use:
`sudo crontab -e # sudo here(!) but not at program to add; see note`
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 400Hz, port 8888, both interfaces (socket, pipe) enabled; -s 10 sets 10 µs sampling; -p might change port. You may test it by rebooting and repeating above tests without prepending `sudo pigpiod`.

Note: You must put sudo in front of 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 and sudo seem having own cron editor settings and files. Getting this “sudo business” wrong is the source of notorious “crontab setting not working” complaints.

Note also: cron tasks will start without having your environment and path settings. Hence, always 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).

piGpioD does solve the GPIO sudo hassle. Additionally, Joan N.N.'s pigpio library has a lot of rich and useful features, like PWM on every pin, a substitute for digital input interrupts and much more. By the way: Do not use interrupts!

This library's socket approach with all good points brings one disadvantage:
A (binary) single write to one pin takes 98 µs on slow Pis. Making more than 4 such IO calls in a 1 ms cycle is, hence, a no go. This sounds harder than it is in reality. There are several remedies:

a) A programming style not endlessly repeating the same IO operation, like turning on a relay already on, is almost always adequate.

b) Use pigpio's bulk / bank functions whenever feasible. With **one** call you can affect any set of I/O registers in **one** bank. And nota bene: **All** IO-pins are in **one** bank!

c) pigpiod has output functions which it can perform with 10µs resolution (in our setting above, 5µs is default) by itself as server/daemon. PWM with individual frequencies and ratios **on every pin** being the most modest example.

d) You may have programmable waveforms, samplings, UART outputs on every pin etc. pp.

You don't have to use all this, but bear it in mind, when hitting any limit.

The pigpio site (http://abyz.co.uk/rpi/pigpio/pdif2.html) offers no offline .pdf document. The very good on-line documentation allows being .pdf-printed (60 pages, links partly working).

PWM and RC servos on every GPIO

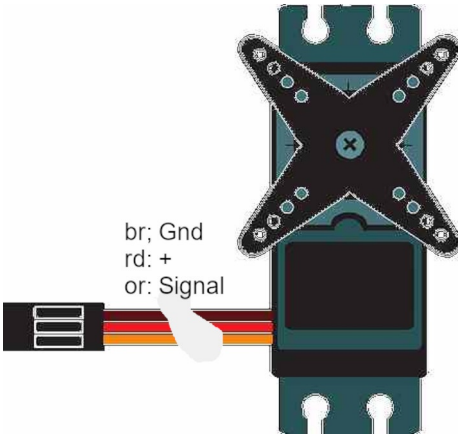


Fig. 5: Small servo for remotely/radio controlled (RC) models.

Also without extra interfaces, a Pi can control standard RC servos by connecting an output pin to the signal input, orange in Fig. 5. These servos are available in a large range of sizes and torques up to more than 8Nm. They come handy as actors in process control applications.

The supply voltage ranges from 4.8 to 8.4V; in Fig. 5: red= **+**; and br = **-** and signal ground. The control signal is a train of digital CMOS/TTL pulses. The position is determined by the pulse width:

Left: 1.9ms | Center: 1,5 ms | Right: 1.1ms positive pulse: __⎍__

The distance of the pulse is not relevant, but 20ms is usual and worked in all cases. Hence, a 50Hz PWM signal 24…14 (for range 256) would do the job. But the pigpio saves us all arithmetic trouble by giving us fine grained servo control by the function:

```
int gpioServo(unsigned user_gpio, unsigned pulsewidth)
```

The parameter pulsewidth is in the range 500…2500 (1500 = middle) or is 0 for "Off". Note: The off behaviour depends on the servo type. Most hold the last position, same go to a default position.

Of course, `gpioServo()` uses the PWM on every GPIO feature.

PWM on the Pi

Hardware PWM is available on GPIO12, 13, 18 and 19 i.e. pins 32, 33, 12 and 35. To generate a PWM-signal the following is done, roughly:

a) choose a frequency / hardware clock (then shared by multiple) GPIOs

b) set a counter fed by clock a) counting 0…pwmRange-1
Accordingly the frequency of the PWM signal will be clockFreqency/pwmRange.

c) For a comparator (connected to this counter) set a value
dutyCycle in the range 0…pwmRange

dutyCycle == 0 means always off; dutyCycle == pwmRange means always on; (highest priority) on counter overflow set output 1; on counter == dutyCycle set output off. Result: the PWM signal.

pigpioD can use this PWM hardware roughly described here on those GPIOs automatically, when appropriate.

But as it is a server/daemon running at it's sample rate – 1, 2, 4, 5, 8 or 10µs – with hardware clock accuracy it can implement said PWM clocks, counters and comparators partly or fully by its software (and as very well know by now) for **every** GPIO pin.
pigpioD's default sample rate is 5µs and we always used 10µs as sufficient in all our applications so far. The values of the 18 available PWM frequencies depend on the sample rate and the (real) range depends on the frequency (number); see table PWM.

frequNo	1µs	2µs	4µs	5µs	8µs	10µs	range
1	40000	20000	10000	8000	5000	4000	25
2	20000	10000	5000	4000	2500	2000	50
3	10000	5000	2500	2000	1250	1000	100
4	5000	4000	2000	1000	1000	800	125
5	4000	2500	1250	800	625	500	200
6	2500	2000	1000	500	500	400	250
7	2000	1250	625	400	313	250	400
8	1600	1000	500	250	250	200	500
9	1000	800	400	200	200	160	625
10	1600	625	313	160	156	125	800
11	800	500	250	125	125	100	1000
12	625	400	200	100	100	80	1250
13	500	250	125	80	63	50	2000
14	400	200	100	50	50	40	2500
14	250	125	63	40	31	25	4000
16	200	100	50	25	25	20	5000
17	100	50	25	20	13	10	10000
18	50	25	13	10	6	5	20000

Table PWM: Available PWM frequencies/Hz by frequency number (first column) and by sample rate (first line) as well as (real) PWM ranges (last column) by frequency number.

Well, pigpio makes every effort to hide these complications. You just order your signals or servo position via the pigpio C interface ([62], [62b]). But is is no fault to have a bit of background.

Resume for piGpioD and a look on other GPIO libraries

To quickly resume for piGpioD:

- It is our choice – except on a Pi5, see the warning on page 21 – and
- service proofed several in real time process IO applications running 24/7 for over 8 years
- worked on all Pi 0..4
- good base for pure Java process IO applications without any extra libs; see [34].

Nevertheless some years ago we installed and tried *wiringPi, bcm2835* and a derivative, see [33] if interested in spite of everything. Download and installation described there still seems to work.

wiringPi tries to cover the range of Raspberry Pi1, 2 and 3 with its diverse variants as well as some alternate Pi lookalikes. Additionally wiringPi knows a wide selection of extension boards or so called “shields” considered at that time popular. Well this "serve all" approach is doomed to fail under other than the assumed "what is all" conditions.

Probably as consequence of this “cover everything” approach wiringPi introduces indirection respectively abstraction layers away from the µP's (BCM2835, BCM2836, BCM2837) GPIO numbers or (virtualised) IO register addresses ([56]). Wisely, pigpiod does nothing of that sort. This “feature” (addressing Pin numbers instead of GPIO register numbers) is easily added by some simple Pi type specific macros, as we do by type respectively architecture specific macros like e.g. config_raspberry_04.h (plus 00 and 03, but not 05):

```
#define PIN37 26  //!< GPIO 26
#define PIN38 20  //!< GPIO 20
#define PIN40 21  //!< GPIO 21
```

And you can easily add a next abstraction/marco layer to name process signals and put all “wiring” definitions in one place:

```
#define LEDgn PIN37 //!< The green LED to blink
```

Implementing those abstraction layers by marcos does no harm at runtime nor on real time behaviour, as C handles all macros completely at compile time.

And as pigpiod transparently adds functions as PWM and UART input on every IO pin, you may think in pins from the hardware IO perspective.

Re-consider our example gnBlinkSimple, listing 2, page 21. With pigpio(d) you may dim the LED on pin 37 with PWM by just changing the line

```
gpio_write(thePi, LEDgn, ON);
```

to

```
set_PWM_dutycycle(thePi, LEDgn, 128); // 128 = half power, 50% PWM
```

Nota bene: `LEDgn → PIN37 → GPIO 26` is none of Pi’s four outputs with PWM !

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 bcm2835 documentation stops between 2012 and 2014 with the Pi1, leaving the rest to forums and speculation. Having switched totally to pigpiod (2017 - 2020), we haven't investigated other libraries further.

Other protocols (also) **for process IO**

As said: With pigpio(d) we have total control over binary input and output devices like switches, LEDs, optocouplers, power transistors, relays, RC servos and much else.
Other devices we can handle by special protocols (Modbus, MQTT, …) over serial connection (RS232, RS485, 1wire, …) including also USB and Ethernet.

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).
Note: For micro-controllers, like e.g. most Atmel AVR boards, this is not their “natural” way.

(W)LAN and TCP/IP communication is an integral part of Raspberries and most available OSs, like Raspbian respectively Pi OS. 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.

For process control communication (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. Most often it's wiser to use a standard protocol, like Modbus or MQTT.

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 RS232 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 mistakes known to computer scientists or good programmers already in 1979:

▼ Modbus has no layer concept and mixes physics, transport and application in an unfortunate way. The standard makes inadequate references to concrete devices and their addressing idiosyncrasies. 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 and what was meant by "only for a 984A/B/X machine"

▼ Modbus has no data type concept worth the name.
One type is "**coil**" [sic!] = copper wire for relays. This is a boolean forced to one bit. Modbus insists to transfer thousand of bits (sorry "coils") starting at arbitrary odd bit [sic!] addresses aligned to bytes – meaning RS232/485 bytes since 1979 and also TCP bytes a bit later. This shifting nonsense is a burden to implement and test as well as detrimental to runtime performance.
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 “register” approach brings the full computational load and risk of errors by endianness handling – without doing any good for applications' or devices' 32 or 64 bit data.

For the serial interfaces RS232/485 – still in wide use for small controllers – Modbus RTU won't use UART parity supplemented by a simple checksum. The standard requires a complex CRC telegram checksum. This overkill is, again, overcharging some controllers even with a clever double look-up algorithm, requiring 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 and requiring extra programming. 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 intentionally reject to communicate with such non-conformer – and, alas, quite rightly so.

Modbus, by itself, has no security measures. This can and must be handled by using protected networks. Serial RS232/485 lines should always fall in that category (by physics). For Modbus TCP/IP use protected private LANs (or tunnels). On the other hand:
If devices have TCP do use other communication protocols if you can influence it.

Resume: Modbus is a widespread industry standard and we should be able to handle Modbus RTU (and TCP) on our Raspberry controllers.

Serial (RTU) Modbus mostly via RS232/485 has to be used when devices of high quality lack other communication means. The most prominent example are good (industry quality) one and three phase power meters, like Eastron SDM630 to name just one. Besides an unsigned “impulses per kWh” output they communicate by RS485 Modbus (as slave) only. Some allow up to 38,4 Kbaud but we saw seldom stable communication above 9,6 KBaud (the default) on a RS485 bus.

As some solar inverters want to have such smart house meter in some modes, many inverters do have RS485 or RS232 Modbus. – sometimes as their only communication interface. Some allow other modes than being the sole master over the houses smart meter (Growatt nm00-S e.g.) and are thus possible clients for our Pis.

Raspberries need extra hardware (like MAX485/232 modules, cf. Figure 10, page 58) to have RS485 or 232. On the other hand there are many process IO devices around with good value for money featuring a Modbus interface, like said smart meters and inverters. Linking those to a Raspberry opens a rich field of applications with professional process IO. So let's have serial “Modbus RTU” in our portfolio. For the serial hardware and the handling of Pi’s serial interface (Pins 8 & 10) see the chapter “serial communication hardware” (and software hints, page 56).

libmodbus

Implementing a minimal Modbus subset (class 0, TCP IP e.g.) is relatively easy. Going to higher classes or other interfaces will get hard work. But even with small subsets it is re-inventing the wheel, considering age and wide use of the protocol. But looking for a reliable and conforming Modbus library for Raspbian (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 (RTU),
+ using an (▼ inconsistent) abstraction layer concept for the interfaces. So the application software would be less affected should one ever switch from RTU to TCP or back.

▼ The library is a “heavy malloc user”, which should be avoided having real time and process control. (In fact, we never observed timing faults caused by the library.) A remedy would be extracting a simple RTU only and a TCP only subset from the library. But that’s harder than expected and without real troubles not worth the while.

▼ 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().

▼ The linked multi-stage (C) pointing leads to horrible readability (impeding the an easy making of subsets).

▼ 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, ..... const char *  template, ...)
```

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

On the other hand, this little list sounds much worse than it is meant. Monsieur Raimbault's library just works – and quite well so. To resume: With libmodbus we do have the complete sources of an excellent library to work with – and (not so easyly) to learn from.

Installation

Do the usual update/upgrade exercise. The installation needs git (-core); if not installed yet do:

```
sudo apt install git  ;: before it was git-core
```

Then do:

```
sudo apt install -y autoconf libtool
git clone https://github.com/stephane/libmodbus/
cd libmodbus/
dir
```

```
./autogen.sh                        ## *)
./configure -prefix=/usr/local      ## or as shown just .\configure
```

```
libmodbus 3.1.11
:::::::::                                *)
```

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

cd ~/libmodbus/tests/
./unit-test-server &          ## best note the process number **)
./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 unit-test-server      ## or try: kill process number noted
```

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 our biggest mental hurdle at first encounter a lot of years ago.

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, many pages of scripts! – do with my system?
Gives you some 7 files needed.
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.
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:
When wanting libmodbus, "Yes, do it". Be courageous or make a backup before.

In the end you need the following files – and transferring those to another Raspberry with same PiOS to another one does the installation job:

- the sources of the library 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 (all 6) 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
```

***) Version .10 has another set of include files in /usr/local/include/modbus/
config.h may be in ~/libmodbus/ instead of ~/libmodbus/src/ .

```
-rw-r--r-- 1 root root 12960 2024-10-19 13:19 modbus.h
-rw-r--r-- 1 root root  1195 2024-10-19 13:19 modbus-rtu.h
-rw-r--r-- 1 root root  1360 2024-10-19 13:19 modbus-tcp.h
-rw-r--r-- 1 root root  2205 2024-10-19 13:19 modbus-version.h
```

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 program on another Raspberry an unlinked libmodbus.so ftp-transferred there should suffice after copying 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 programs required `libmodbus.so.5` even when made with -lmodbus.

Cross compile and build – 32 bit

After a successful installation or transfer one can locally compile:

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

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`

```
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 “private” .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`

Cross compile and build for different target OS

When cross-building on a PC – almost always Windows in our case – for different target machines with 32 and 64 bit Pi OS respectively Raspbian we have to have separate tool chains with different compilers, linkers as well library (.so) and most often include (.h) files. The two toolchains we have on Windows at

`C:\util\WinRaspi\arm-linux-gnueabihf\` for 32Bit Pi OS/Raspbian or at
`C:\util\winRaspi64\aarch64-linux-gnu\` for 32Bit Pi OS/Raspbian.

The tools are in the respective subdirectory bin\ and are additionally distinguished by a prefix, e.g.
`arm-linux-gnueabihf-gcc.exe` and `aarch64-linux-gnu-gcc.exe`

for the C compiler. Hence we can put both subdirectories bin/ on the Windows PATH. Our makefile has a variable whit it prepends to the tools needed, excerpt:

```
TOOLPREFIX ?=arm-linux-gnueabihf-
CC = $(TOOLPREFIX)gcc
```

Obviously, TOOLPREFIX is set to 32 bit by default, but can set otherwise in a target specific (make) include file, as the operating system is a property of the target machine.

Cross compile and build – 64 bit

Having (and choosing) the tools is perfectly solved by PATH and prefix.

Having the include (.h) and library (.so) files a the right places is perfectly solved for pigpiod. The Windows cross toolchains have those files for pigpio (and wiringpi) at the right places. And the installation of pigpiod does the same on the target Raspberry Pi.
Alas, libmodbus has and does neither.

You have to copy all necessary .h files, .so files and .so links to the right places. Do not let the copy command make file copies from .so links! “Right places” in our case were:

```
C:\util\winRaspi64\aarch64-linux-gnu\sysroot\usr\include
C:\util\winRaspi64\aarch64-linux-gnu\sysroot\usr\lib
C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\include  ## 32 bit
C:\util\winRaspi\arm-linux-gnueabihf\sysroot\usr\lib      ## 32 bit
/usr/local/lib/                                          ## Linux PC / Pi
```

Hint: Preserve the links on the Pi when copying by

```
sudo   cp -P /usr/local/lib/libmod*   /usr/lib/
```

64bit-Warning: These little in-compatibilties may only be the tip of the iceberg. When a modbus application fails, finding the hardware and/or software causes is quite hard. In one case it might (also) be a problem with 64 bit cross toolchain “aarch64-linux-gnu”. When resolved, it will be documented in a future edition.
Since about two years those of us, using Pis for automation and control, suffered some in-combatibilities by hardware and Linux developers: Pi5 killing pigpiod process IP, Trixie stealing tPi’s hardware Watchdog (page 66), ...

MQTT

MQTT ([67]) is a TCP based lightweight M2M messaging protocol claiming to be simple, low network bandwidth and small code footprint. Notwithstanding those claims, at least the "simplicity" might be slightly doubted.

MQTT is quite successful and widely used in the world of IoT and home automation. Most sources and the protocol are open. The protocol specification is managed by OASIS. To use it, one has to have at least one broker service running on a machine reachable by all clients via TCP/IP. This machine may very well be a Raspberry Pi doing other process control work.

A client program producing events might publish those to the broker. The event message must be assigned to a "topic". Topics are to be organised hierarchically in a tree, e.g.

```
labExp/sweetHome/meters/alarms
labExp/sweetHome/hmi/actuators
```

The publisher might chose several service quality levels from "once or lost", "at least once or duplicated" to "just once" when delivering a message to the broker.

The broker would run by default with no security or encryption. It might be configured with an own user/password base for access to ports and topics, for requiring client certificates and for using TLS. Before jumping to those features in a closed, secured (W)LAN, one should consider these

requirements hitting all clients at the site. Think twice before burdening our small Raspberry and ESPxy devices with complicated protocol extensions. But beware: This "have no fear" only holds until we open our private process control LAN to someone else.

A client program wanting to react respectively listen to events has to subscribe to one or more topics. Besides specific topics, like the two examples above, single level (+) and multi-level (#, at the end, only) wildcards can be used:

```
labExp/sweetHome/+/alarms
labExp/sweetHome/hmi/#
```

In a certain sense, we have the Listener respectively Observer pattern implemented by the MQTT broker to which clients (programs) can publish or subscribe. A client program may very well have both the publisher and the subscriber role.

A client may publish one special "last will" or "testament" (LWT) message. This message will be stored by the broker and pushed to subscribers, only, after this client failed or died. Criteria for death are network and MQTT protocol errors and time outs.

MQTT products

There are a lot of MQTT enabled products with good price performance ratio on the market:

- ESP8266 WiFi modules with MQTT
- single to quadruple power (relay) switches, optionally
  - 230V AC or 24V
  - DIN rail mounted or wall plug-in
- touch wall switches
- lamps / bulbs and a lot more.

As did Modbus with smart meters, pumps etc., MQTT opens another range of products as sensors and actuators as well as “headless” three phase meters for our process control Raspberries.

MOSQUITTO

A Raspberry Pi 3 doing a 24/7 process control task in say 20 ms cycles (or even 1ms) may very well execute some extra services, like an Apache 2.4 server for HMI (see page 13) and a MQTT broker, when used judiciously and almost only related to the process control task.

Mosquitto is an open source implementation of a MQTT broker etc. by Eclipse. It seems to be the most used implementation for small and embedded systems, notwithstanding the lack of findable respectively usable documentation and manuals.

To install Mosquitto do:

```
sudo apt update && sudo apt upgrade
sudo apt install mosquitto
sudo service --status-all | grep mosq
```

The installation command brings the broker and starts it as service. The broker service can be switched off and on by:

```
sudo service mosquitto stop
sudo service mosquitto start
```

If we want the broker alone (on an extra Raspberry, e.g.) we would stop here. For having the client (test) program, the library and the include file we do:

```
sudo apt install mosquitto-clients
sudo apt install libmosquitto-dev
sudo ldconfig
```

To test our new broker we utilise the command-line clients, just installed:

```
mosquitto_sub -d -t labExp/sweetHome/# &
```

`-t` precedes the topic(s) we subscribe to, and `-d` brings some extra (debug) messages, helping to get acquainted with the protocol. To see the broker and our (one) running subscriber working we publish a fitting message by command-line, best in another putty console:

```
mosquitto_pub -d -t labExp/sweetHome/alarm -m "bathtub overflow 3"
```

If all went well, our subscriber sees the event, immediately. When sending an off topic message

```
mosquitto_pub -d -t hello/world  -m "Hello 11"
```

nothing should happen.
Note: Besides mosquitto_pub and mosquitto_sub, there's a third command-line tool mosquitto_passwd.

libmosquitto

By some searching and “greping” (due to lack of documentation) one can see, we have just one library and one include file in /usr/lib/ and /usr/include/ (32bit):

```
lrwxrwxrwx  root root     17 2017-05-29 07:21 libmosquitto.so
                     -> libmosquitto.so.1
-rw-r--r--  root root  46960 2017-05-30 00:32 libmosquitto.so.1
-rw-r--r--  root root  54155 2017-05-29 07:21 /usr/include/mosquitto.h
```

respectively in /usr/lib/aarch64-linux-gnu/ (64bit):

```
lrwxrwxrwx 1 root root   17 2025-03-21 19:15 libmosquitto.so
                     -> libmosquitto.so.1
lrwxrwxrwx 1 root root   22 2025-03-21 19:15 libmosquitto.so.1
                     -> libmosquitto.so.2.0.21
-rw-r--r-- 1 root root 130K 2025-03-21 19:15 libmosquitto.so.2.0.21
-rw-r--r-- 1 root root 124K 2025-03-06 17:23 /usr/include/mosquitto.h
```

Compile and cross-build

To cross-compile and -build on the workstation we put the include file to

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

(32 bit) respectively (64 bit)

```
C:\util\WinRaspi64\aarch64-linux-gnu\sysroot\usr\include
```

and the library both as libmosquitto.so.1 as well as as "un-linked" (copied) libmosquitto.so to

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

That's it. As first test we make a simple publisher example (with accompanying make include) in Eclipse and having the make tool build it and ftp it to the target Raspberry, as usual. With this proof of feasibility – and when having understood the API ([69]) – we can add MQTT functions (public and/or subscribe) to our C process control software. And we can make own small (ESP8266 based, e.g.) MQTT devices.

Similar cross tooling problems occur when libraries are in folder where the cross compiler linker wouldn’t look. For example files libpthread.* where in

```
C:\util\winRaspi64\aarch64-linux-gnu\sysroot\usr\lib\aarch64-linux-gnu\
```

instead of `C:\util\winRaspi64\aarch64-linux-gnu\sysroot\usr\lib\ .`

```
cd C:\util\winRaspi64\aarch64-linux-gnu\sysroot\usr\lib\
copy .\aarch64-linux-gnu\libpthrea* .\
```

solved the problem.

Part II's results

Since Part I we can install a usable OS and tools on Raspberries. Having provided all basic tools and having set up communication we can administer Pis with some comfort remotely from both Ubuntu and Windows. And we can save and restore the current state of our ”little servers”, sometimes a bit spoiled by the “size problem”.

Now we just added libraries to control process I/0 of many sorts in real time.

Experiences and measurements show: Comparing Pi OS lite = GUI-less with with the heavy GUI variant we see at non lite:

- ▼ 39+ more services/processes running immediately after reboot
- ▼ 2..5 s delay when typing on ssh (or putty) after a short pause.

On a GUI-less/lite we always saw an immediate reaction as we type.

And when it comes to even modest real time requirements with PLC like cyclic 1 ms tasks (threads) we experience no problems with Pi3 and OS lite, while the GUI variant reproducibly always and totally fails in latencies and many other aspects:

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

This seems not so bad for the non lite graphical OS. But:

- ▲ Loads (ping etc.) have no measurable effect on the lite Pi OS variants.
- ▼ Such loads, on the other hand, significantly increase he latency on the GUI variant. And the latency literally “explodes” by just moving the mouse.

Hence we must state:
A standard PC like GUI is making the Raspberry unsuitable for any embedded / real time/ server work. This resembles our experiences in the large with “real” servers ([29]).

P A R T III Process Periphery and control

Pin	GPIO	funct.			funct.	GPIO	Pin
01		3,3V	o	o	+5V		02
03	02	sda	o	o	+5V		04
05	03	clk.1	o	o	Gnd		06
07	04	pud	o	o	Tx	14	08
09		Gnd	o	o	Rx	15	10
11	17		o	o	pwm	18	12
13	27		o	o	Gnd		14
15	22		o	o		23	16
17		3,3V	o	o		24	18
19	10	MOSI	o	o	Gnd		20
21	09	MISO	o	o		25	22
23	11	CLK	o	o		08	24
25		Gnd	o	o		07	26
27		ID_SD	o	o	ID_CC		28
29	05		o	o	Gnd		30
31	06		o	o	pwm	12	32
33	13	pwm	o	o	Gnd		34
35	19	pwm	o	o		16	36
37	26		o	o		20	38
39		Gnd	o	o		21	40

Fig. 6: Raspberry Pi's 40 pin two-row IO connector

Using the GPIOs

Our first goal is to use Raspberry's GPIO pins in an own program, preferably written in C. In any case, we want our programs cross-compiled (cross-made, cross-build) from comfortable work-stations. For a minimal proof of concept we had the example gnBlinkSimple in Part I, see listing 2, page 21. If skipped, do it now!

A Pi 1 Model A, see figure 7, would also be sufficient for those experiments, but it has no Pin 37. The equivalent of figure 6 would end at Pin26.

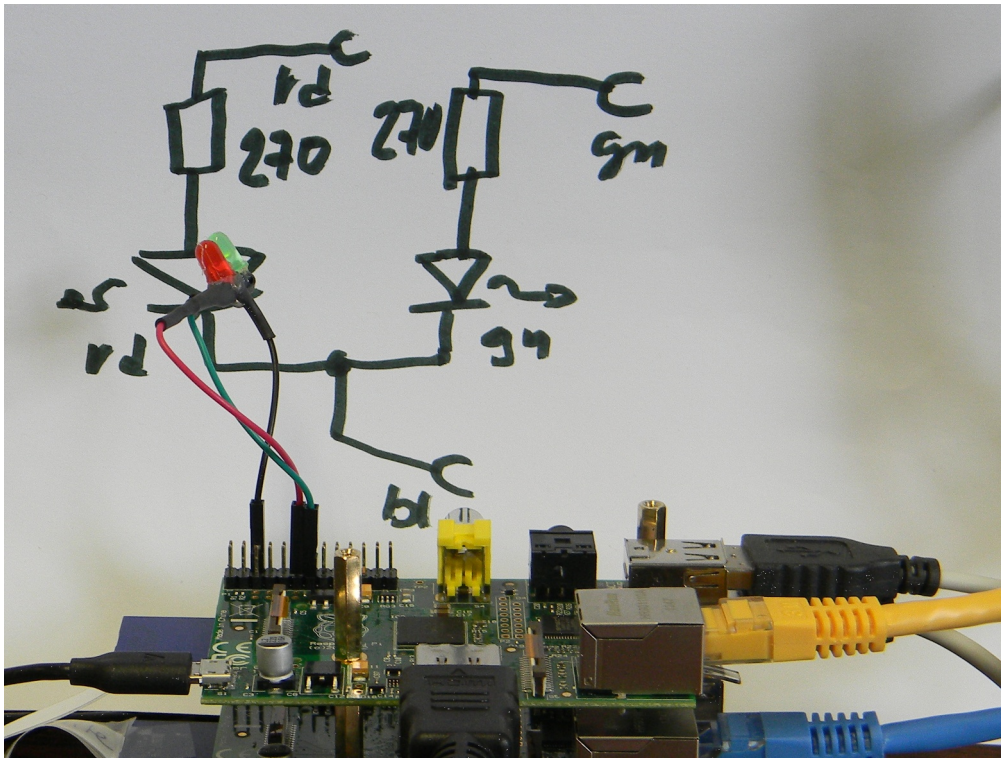


Fig. 7 :Minimal I/O

As hinted above, we handle the IO differences between the Pi types in include files. For a Pi1 (fig. 7) it would be include/arch/config_raspberry_01.h (Listing 4, page 42), and for Pi 0, 3 and 4 include/arch/config_raspberry_00.h, .._03.h etc.; (shortened listing 5, page 43.

** @file config_raspberry_01.h (Listing 4)

```
 *  Configuration settings for Raspberry Pi1 (Models A)

 *  This file contains some platform (type) specific definitions. These
 *  settings influence the compilation and build process. Most of those
 *  settings can not be changed later at runtime.
 *  Since end 2017 we mostly use Pi3 ++; see config_raspberry_03.h.
\code
   Copyright  (c)  2019   Albrecht Weinert
   weinert-automation.de      a-weinert.de

 *     /         /      /\
 *    /         /___    /  \      |
 *    \        /____\ /____\ |  _|__
 *     \  /\  / \    /       \|   |
 *      \/  \/   \__/         \__/|_                                  \endcode

   Revision history \code
   Rev. $Revision: 271 $ $Date: 2024-11-15 19:32:54 +0100 (So, 29 Okt 2023) $
   Rev. 209 10.07.2019 : stdUARTpath
   Rev. 231 13.08.2020 : two digit PIN0x  and  GPIO2pin added
\endcode   */

#ifndef CONFIG_PLATFORM_H
# define CONFIG_PLATFORM_H raspberry_01
#else
# error "more than one platform specific config_platform.h file"
#endif

// by using Joan NN's pigpio(d) we don't need those address values
//#define ARM_PERI_BASE 0x20000000
//#define ARM_GPIO_BASE 0x20000000
//#define ARM_PERI_SIZE 0x01000000

// 26 pin con |GPIO number
#define PIN03   0 //!<  SDA0  (GPIO 02 on other Pis)
#define PIN05   1 //!<  SCL0  (GPIO 03 on other Pis)
#define PIN07   4
#define PIN08  14   //TXDI
#define PIN10  15  //RXDI
#define PIN11  17
#define PIN12  18
#define PIN13  21 //!< GPIO 21  (GPIO 27 on other Pis)
#define PIN15  22
#define PIN16  23
#define PIN18  24
#define PIN19  10  // MOSI | S
#define PIN21   9  // MISO | P
#define PIN22  25
#define PIN23  11  // SCLK | I
#define PIN24   8  // CE0  /
#define PIN26   7  // CE1 /
```

```
// 6 9 14 20 25 : gnd
// 1 17 : 3.3V
// 2 4  : 5V

/** GPIO [0..39] to pin number (1..40) lookup list.
 *
 *  0 means has no pin on the 40 pin connector; see also ::gpio2pin
 */
//                0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
#define GPIO2pin   3, 5, 0, 0, 7, 0, 0, 0,24,21,19,23, 0, 0, 8,10, 0,11,12, 0,\
                   0,13,15,16,18,22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

/** Pin number [0..26] to GPIO number lookup list.
 *
 *  0..56: GPIO number; 95: 5V pin; 93: 3.3V; 90: Ground 0V; <br />
 *  99 means not existent. N.b.: [1..26] are: valid pin numbers, only.
 *  @see ::pin2gpio
 */
//                0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
#define PIN2gpio  99,93,95, 0,95, 1,90, 4,14,90,15,17,18,21,90,22,23,93,24,10,\
                  90, 9,25,11, 8,90, 7,99,99,99,99,99,99,99,99,99,99,99,99,\
                  99,99,99,99
//                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
// 0..56(?) GPIO; 95: 5V; 93: 3.3V; 90: Ground 0V;
// 99: not existent; [1..40]: valid pin number/index; [1..26]: existing
pins

/** /def stdUARTpath
 *  Pi's standard UART.
 *  It is the one on the Pins 8 (GPIO14) for Tx and 10 (GPIO15) for Rx.
 */
#define stdUARTpath "/dev/ttyS0"
```

Listing 4: include file config_raspberry_01.h

```
** @files config_raspberry_03.h  config_raspberry_04.h          (Listing 5)
*  Differences to config_raspberry_01.h (Listing 4 page 42)
  ::::::::::::::::::::

// 40 pin con|GPIO number
#define PIN03  2  //!< GPIO 02  SDA1  (GPIO 0 on Pi1)
#define PIN05  3  //!< GPIO 03  SCL   (GPIO 1 on Pi1)
#define PIN07  4  //!< GPIO 04  GPCLK0
#define PIN08 14  //!< GPIO 14  TXDI
#define PIN10 15  //!< GPIO 15  RXDI
#define PIN11 17  //!< GPIO 17
#define PIN12 18  //!< GPIO 18
#define PIN13 27  //!< GPIO 27       (GPIO 21 on Pi1)
#define PIN15 22  //!< GPIO 22
#define PIN16 23  //!< GPIO 23
#define PIN18 24  //!< GPIO 14
```

```
#define PIN19 10  //!< GPIO 10  SPI.MOSI  | S
#define PIN21  9  //!< GPIO 09  MSPI.ISO  | P
#define PIN22 25  //!< GPIO 25
#define PIN23 11  //!< GPIO 11  SPI.SCLK  | I
#define PIN24  8  //!< GPIO 08  SPI.CE0   / .
#define PIN26  7  //!< GPIO 07  SPI.CE1  / .
#define PIN27  0  //!<  SDA0
#define PIN28  1  //!<  SCL0
#define PIN29  5  //!< GPIO 05
#define PIN31  6  //!< GPIO 06
#define PIN32 12  //!< GPIO 12
#define PIN33 13  //!< GPIO 13
#define PIN35 19  //!< GPIO 19
#define PIN36 16  //!< GPIO 16
#define PIN37 26  //!< GPIO 26
#define PIN38 20  //!< GPIO 20
#define PIN40 21  //!< GPIO 21

// 6 9 14 20 25 30 34 39 : gnd
// 1 17 : 3.3V
// 2 4  : 5V
// 27 28: ID_SD/SC (HAT board & EEPROM interface)  or GPIO 0 and 1

/** GPIO [0..39] to pin number (1..40) lookup list.

 *  0 means has no pin on the 40 pin connector; see also ::gpio2pin
 */
//               0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
#define GPIO2pin  27,28, 3, 5, 7,29,31,26,24,21,19,23,32,33, 8,10,36,11,12,35,\
                  38,40,15,16,18,22,37,13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

/** Pin number [0..43] to GPIO number lookup list.

 *  0..56: GPIO number; 95: 5V pin; 93: 3.3V; 90: Ground 0V; <br />
 *  99 means not existent. N.b.: [1..40] are: valid pin numbers, only.
 *  @see ::pin2gpio
 */
//               0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
#define PIN2gpio  99,93,95, 2,95, 3,90, 4,14,90,15,17,18,27,90,22,23,93,24,10,\
                  90, 9,25,11, 8,90, 7, 0, 1, 5,90, 6,12,13,90,19,16,26,20,90,\
                  21,99,99,99
//               0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,
// 0..56(?) GPIO; 95: 5V; 93: 3.3V; 90: Ground 0V;
// 99: not existent; [1..40]: valid pin number/index
```

Listing 5: include files config_raspberry_03.h, 04h; Differences to config_raspberry_01.h, only

Hint: With some #define magic (in `arch/arch/config.h`) those include files can be selected by a variable PLATFORM in the make process. To get that useful “gimmick” and more, check out the project. Without those you’ll have to control the make process by hand and make the sources less common by hard coding some definitions.

A look at rdGnBlink.c (listing 6, page 45) will make these point clear.

```
/** @file rdGnBlink.c                                                (Listing 6)

A second program for Raspberry's GPIO pins
 Copyright  (c)  2024   Albrecht Weinert
 weinert-automation.de      a-weinert.de

 This is a simple GPIO pin output program using pigpio(d), still.But it has
 features qualifying it as a process control service. It locks gpio, it can
 run endlessly and it cleans up when ended. Hence it may be the germ cell
 of (simple) real time control applications.

 Compile on Pi by:  gcc rdGnBlink.c -o rdGnBlink  -lpigpiod_if2

 Compile on Windows by:\code
 arm-linux-gnueabihf-gcc -DF_CPU=1200000000 -DPLATFORM=raspberry_03 -DMCU=BCM2837
    -DTARGET=Pi4all  -I./include    -c -o rdGnBlink.o rdGnBlink.c
 arm-linux-gnueabihf-gcc -DF_CPU=1200000000 -DPLATFORM=raspberry_03 -DMCU=BCM2837
    -DTARGET=Pi4all  -I./include    -c -o weRasp/weLockWatch.o weRasp/weLockWatch.c
 arm-linux-gnueabihf-gcc -DF_CPU=1200000000 -DPLATFORM=raspberry_03 -DMCU=BCM2837
    -DTARGET=Pi4all  -I./include    -c -o weRasp/sysBasic.o weRasp/sysBasic.c
 cp  rdGnBlink.elf rdGnBlink
 winscp.com /script=progTransWin /parameter sweet:0123  targetPi bin  gnBlinkSimple
 \endcode
 or on Windows simply by\code
 make PROGRAM=rdGnBlink TARGET=pi4play clean all
 make PROGRAM=rdGnBlink TARGET=pi4play FTPuser=sweet:0123 progapp  \endcode
*/
#include <stdlib.h>
#include <stdio.h>
#include <pigpio.h>
#include <pigpiod_if2.h> // set_pull_up_down set_mode gpio_write
#include <unistd.h>
#include <signal.h>   // signal SIGTERM SIGINT
#include "weLockWatch.h"
#include "arch/config.h"  // evaluates PLATFORM

#define LEDgn PIN13   // green LED 1 active is attached to Pin 13
#define LEDrd PIN11   // red LED 1-active is Pin 13
#define ON  1  //  true On  An  marche go.
#define OFF 0  // false Off Aus arret  stop halt.
#define uMS 1000 // 1ms is 1000 us

int thePi; // the Pi handle for pigpiod

//---------  basic configuration and names  -----------------------------
char const prgNamPure[] = "redGnBlink";
//                     ....0123456789x1234567
char const prgSVNrev[] = "$Revision: 271 $   ";
//........      ..........0123456789x123456789v123456789t123456789q
char const prgSVNdat[] = "$Date: 2024-11-15 10:52:04 +0100 (So, 29 Okt 2023) $";
//                         0123456789x123456789v12345

static void onSign(int s){
   if (s == SIGINT) { printf(
       "\n   rdGnBlink (%s) is shutting down normally \n", lckPiGpioPth);
```

```
      exit(0); // cntl C terminates normally
    } else {  printf(
       "\n   rdGnBlink (%s) was terminated, code : %d \n", lckPiGpioPth, s);
      exit(s);
    }
} // onSign(int)

static void onExit(){
   set_mode(thePi, LEDgn, PI_INPUT);  // release red LED pin
   set_mode(thePi, LEDrd, PI_INPUT);  // green LED pin
   closeLock();
} // onExit(int, void*)

/* Relative delays for the specified number of milliseconds */
void delay(unsigned int millis){
   usleep(500 * uMS);
} // delay(unsigned int)

int main(int argc, char* argv[]){
  printf(
   "\n             Blink green and red LEDs on Pins 13 and 11 \n"
     "rdGnBlink.c  R.01 15.11.2024          Albrecht Weinert \n\n");
  if (openLock(lckPiGpioPth, ON)) return retCode; // no lock/singleton
  thePi = pigpio_start(NULL, NULL); // connect pigpio daemon(local, 8888)
  if (thePi < 0) { // initialise pigpio  daemon failed
      printf("\ncan't initialise IO handling  (piGpioD) \n"
               "rdGnBlink.c      Error / return code 99 \n\n");
      return 99;
  } // can't initialise piGpioD (the essential gpIO library)

  atexit(onExit); // register exit hook
  signal(SIGTERM, onSign); // signal hook
  signal(SIGABRT, onSign);
  signal(SIGINT,  onSign);
  signal(SIGQUIT, onSign);
  set_mode(thePi, LEDgn, PI_OUTPUT);
  set_mode(thePi, LEDrd, PI_OUTPUT);

  while(1) {             //                    red green
    gpio_write(thePi, LEDrd,ON);
    delay(200);          //          200 ms    red
    gpio_write(thePi, LEDgn, ON);
    delay(100);          //          100 ms      both
    gpio_write(thePi, LEDrd, OFF);
    delay(100);          //          100 ms        green
    gpio_write(thePi, LEDgn, OFF);
    delay(200);          //          200 ms      dark
  } // while endless     //    loop  600 ms
} // main()
```

Listing 6: rdGnBlink.c, the second more professional gpio program

Listing 6 is still is a simple GPIO output program using pigpio(d). Compared to he example gnBlinkSimple, see listing 2, page 21, the visible difference is its

a) blinking with two LEDs instead of one.

gnBlinkSimple was self-contained, while rdGnBlink (listing 6, page 45)

b) uses extra sources, includes and make files.

This is purely technical and makes the sources flexible and re-usable.

Additionally rdGnBlink has

c) features qualifying it as a process control service.
It locks gpio, it can run endlessly and it cleans up when ended.

The real decisive difference is c). Now it may be time to explain why rdGnBlink.c (listing 6) 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 6.

But aiming at real time, real processes and distributed control 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 well behaving *) control program running.
- A watchdog is easily added (and should be, see page 66 below).

Note *): “Well behaving” means: All programs using GPIO (i.e. pigdiod daemon) shall get a certain lock or die making a daemon user be a “singleton”.

Note also: Holding lock by itself requires a watchdog.

Comparing listing 2 and 6 shows “process control” part implementing the external behaviour being essentially the same. The complication by those two/three – minimal by the way – requirements won’t grow substantially when getting to a full grown process control program.
And we put the required code in extra sources and include files, see Listing 7 below.

Using the simple version (listing 2, page 21), it will easily get evident or demonstrated that running two instances of the program gnBlinkSimple 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 did so. Additionally well meaning users tend to start control applications without noticing or checking their state.

The Unix style solution in listing 6 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 program 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.

```
/** Excerpt from files weRasp/sysBasic.c,                    (Listing 7)
                       weRasp/weLockWatch.c etc.  (.h-files omitted here)

/*  Basic start-up function failure. */
int retCode;

/* Common path to a lock file for GpIO use */
char const  * const lckPiGpioPth = "/home/sweet/bin/.lockPiGpio";

/*  Open and lock the lock file.
 *  This is the basic implementation of ::openLock. Applications not
 *  wanting its optional logging or doing their own should use this
 * function directly.
 *  @param lckPiGpioFil   lock file path name
 *  @return 0: OK, locked; 97: lckPiGpioFil does not exist;
```

```
 *         98: can't be locked
 */

int justLock(char const * lckPiGpioFil){
   if (! useIOlock) return 0;
   char const * lckPiGpio = lckPiGpioFil != NULL ?
                                lckPiGpioFil : lckPiGpioPth;
   if ((lockFd = open(lckPiGpio, O_RDWR, 0666))  < 0) {
      return retCode = 97;
   } // can't open lock file (must exist)
   if (flock(lockFd, LOCK_EX | LOCK_NB) < 0) {
      close (lockFd);
      return retCode = 98;
   } // can't lock lock file
   return retCode = 0;
} // justLock(char const *)
```

Listing 7: The code for central locking of IO

The gpio lock file is best put in the bin directory of the main process control resp. IO user. In our example listing 7 it is `/home/sweet/bin/.lockPiGpio`. Logged in as that user create it as file of length 0 just once by

```
touch ~/bin/.lockPiGpio
```

If this user shall be the only one to such correctly locking process IO programs that was it. If you want to run alternative programs or test versions as other users in the same group (would be sweet in our example) do also

```
chmod g+w  ~/bin/.lockPiGpio
```

In Linux the right to lock a file is the right to write cause of the just three bits for assigning rights. If other users shall be able to lock that file do

```
chmod a+w  ~/bin/.lockPiGpio
```

Of course, the lock file must be unlocked when program ends – no matter why or how the program 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 program ends, no matter how and why. In listing 6 (find signal() and atExit()) this is done by catching the relevant signals (interrupts) as well as the program 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 procedure.

If this is not adequate the design of the process IO attached is wrong. If the outputs ar de-energised i.e. high impedance state by (our making them inputs) or by turning the (µ) controller of no actor, relays etc. should come in an active (moving, energised, …) state.

Turning off power or making outputs high impedance must lead to a “safe state” as golden rule.

Having just a “singleton” GPIO user would not mean that another application may not observe the GPIO states (if well written and behaving).

A small program `gpioList.c` gives us snapshot of gpio 0..31 for an exemplary controller running 24/7 on a Pi3 by just typing

```
sweet@meterPi:~ $ gpioList
```

The program `gpioList` won’t change any state of the GPIO registers 0 to 31 as H(igh, 1) or L(low, 0). It tells the Pin number, if applicable, according to the Pi type. The register contents for alternative functions, like PWM, are only relevant when such function is actve.

```
 Pin  | GPIO | Mode | freq | duty | range
------|------|------|------|------|------
  27  | 00 H |  in  |  400 |  -92 |  255
  28  | 01 H |  in  |  400 |  -92 |  255
  03  | 02 H | out  |  400 |  -92 |  255
  05  | 03 H |  in  |  400 |  -92 |  255
  07  | 04 H |  in  |  400 |  -92 |  255
  29  | 05 L | out  |  400 |  -92 |  255
  31  | 06 H | out  |  400 |  -92 |  255
  26  | 07 H | out  |  400 |  -92 |  255
  24  | 08 H |  in  |  400 |  -92 |  255
  21  | 09 L |  in  |  400 |  -92 |  255
  19  | 10 L |  in  |  400 |  -92 |  255
  23  | 11 L |  in  |  400 |  -92 |  255
  32  | 12 L | out  |  400 |  -92 |  255
  33  | 13 L | out  | 1000 |  118 | 1000
  08  | 14 H | alt2 |  400 |  -92 |  255
  10  | 15 H | alt2 |  400 |  -92 |  255
  36  | 16 L | out  |  400 |  -92 |  255
  11  | 17 H | out  |  400 |  -92 |  255
  12  | 18 H | out  |  400 |  -92 |  255
  35  | 19 H |  in  |  400 |  -92 |  255
  38  | 20 L | out  |  400 |  -92 |  255
  40  | 21 L | out  |  400 |   48 |  255
  15  | 22 H | out  |  400 |  -92 |  255
  16  | 23 H | out  |  400 |  -92 |  255
  18  | 24 H | out  |  400 |  -92 |  255
  22  | 25 H | out  |  400 |  -92 |  255
  37  | 26 L | out  |  400 |  -92 |  255
  13  | 27 H | out  |  400 |  -92 |  255
      | 28 H | alt0 |  400 |  -92 |  255
      | 29 L | alt0 |  400 |  -92 |  255
      | 30 L | alt5 |  400 |  -92 |  255
      | 31 L | alt5 |  400 |  -92 |  255
------|------|------|------|------|------
 Pin  | GPIO | Mode | freq | duty | range
  Low/High^           no PWM=-92
```

gpio 4 (pin 7) is used for 1wire which can't be seen here; we just see input.

gpio 13 (pin 33) generates the 1kHz (1000 Hz) 50% duty cycle. At this Pi it is a CP (control pilot) signal waiting for an electrical car's request for AC one or three phase loading *).

gpio 14 and 15 (pins 8, 10) are a UART, "alt2", controlling two smart meters via RS485 Modbus.

gpio 28 ..31 do not end in IO pins. On a compute module they are usable.

All others are are used as inputs or outputs or they are not used here (and listed as "in).

Note *): The loading circuit amplifies this to a ±12V rectangular signal with 1 kOhm impedance. The car would load the positive half of the ±12V with a resistor to show its presence, . . . and so on in the "analog protocol". The duty cycle 118 / 255 is not 50%. The -10 here compensates the asymmetries introduced here by optocouplers and the analogue circuit.

An application as service

When using the Raspberry as server – or device – we usually want applications (process control, web server etc.) start automatically when powering up without a user having to login.

rc.local

One way is putting the starting command at the end of `/etc/rc.local`.
For `rdGnBlink.c`, listing 6, page 45, put there:

```
/home/sweet/bin/rdGnBlink &
```

Our small program will be started at the end of the boot process. Do not forget the & at the end. It puts the thing in the background goes on with the script.

To stop a program started this way, best determine the process ID number and kill it:

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

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

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

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

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

or (if available) a "mass murderer" command

```
sudo killall rdGnBlink   # also works on most Pi OSs
```

The last one -- `killall` -- the most simple one and is preferable if used by hand. For a well behaving process control program forcing itself being the singleton handler of IO there is no danger of mass murdering. The other complicated are used / found in listing 8, page 51.

cron

The cron service knows an event “@reboot”. Add an @reboot line by

```
crontab -e                          ## note no sudo here for a user process
@reboot /home/sweet/bin/rdGnBlink   ## add a delay if needing services
```

This works without & at the end at the cost of an extra shell process.

```
ps aux | grep rdGnBlink
```

reveals the extra (avoidable) shell process process, the can be killed. Putting an & at the end avoids the extra /bin/sh. On the other hand this seems to multiply the boot time.

```
pi 526   1912   388 ?  16:53  0:00  /bin/sh -c /home/sweet/bin/rdGnBlink
pi 528   4008  1696 ?  16:53  0:00  /home/sweet/bin/rdGnBlink
pi 818   4776  1916 S+ 17:16  0:00  grep --color=auto rdGnBlink
```

As more complicated programs require other services (gpiod e.g.), DHCP settled etc. one would like to start them a certain time (35 s in example) after reboot:

```
@reboot sleep 35 && /home/sweet/bin/hometersControl
```

The cron service is probably running by default, but 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/sweet/bin/rdGnBlink)
```

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

Mimic a service – start stop restart enable disable

With a bash script (listing 51 below) rdGnBlinkCntl we control our exemplary “process control” program (listing 6: rdGnBlink.c, page 45). Make the script by

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

and work on it via FileZilla and editpad.exe to have this:

```
##!/bin/bash                                                          listing 8
showRdGnBlinkCntlVers () { echo "
#  /usr/local/bin/rdGnBlinkCntl  resp. ~/bin/rdGnBlinkCntl
#  control the rdGnBlink service          V.59  30.11.2024
#  (c) 2017  Albrecht Weinert        weinert-automation.de
"; }
rdGnBlinkCntlHelp () { echo "
#  call by: rdGnBlinkCntl command
#  commands are:
#    start | stop: start or stop rdGnBlink
#    restart:      stop and then start
#    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/sweet/bin/rdGnBlink
searchPt=rdGnBlink
stop() {
  ps aux | grep -m 1 ${searchPt} | awk '{print $2}' | xargs -t kill
}
start() {
  ${progPath} &
  pid=$!
  sleep  .3
  if ps -p $pid > /dev/null; then
    echo "rdGnBlink 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; sleep 2;
 start
fi
rdGnBlinkCntlHelp
```

Listing 8: Script rdGnBlinkCntl to control rdGnBlink (listing6) as service.

Making a library

Having common utility functions, variables and values in extra .c and .h files, let's see how to make a library from them, when not wanting to link them to every executable in question.

In a key matrix example ([33]) once we had three sources

- keysPiGpioTest.c main program
- 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

The usual (and no bad) way is to translate all three .c files to .o files best organised in the makefile and link all three .o files to the executable keysPiGpioTest.

With this simple procedure (just translate all) you may

- change every source before cross-build. The 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

without having to transfer your private library

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

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

Then you translate one source less and link the extra library with the -lsysUtil 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
sudo ldconfig
```

This topic is put here just for completeness. As long as the cross make / cross build chain works we see no real advantage in building libraries and to have keeping track of their versions.
Well, if you want give a customer functionality without the sources this is one way.

Process IO hardware

In our introductory examples (rdGnBlink, listing6, page 45) the process IO is directly connected to 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 8 shows a professional key matrix, LEDs and a piezo beeper as one suitable example for directly attached process IO. The break-out board is just a test harness and helps to connect a logic analyser – it is the black box with the green LED on the left of Figure 8.

In the production version the Pi is fixed to the back of the key matrix. When WLAN is sufficient, we now use a Pi Zero in that place.

In all other cases and when controlling power beyond 48 mW, or distant actuators and sensors, interface and protecting circuitry is mandatory. Many offers are found, often under the genus “shield”. But beware of useless or outright dangerous devices.

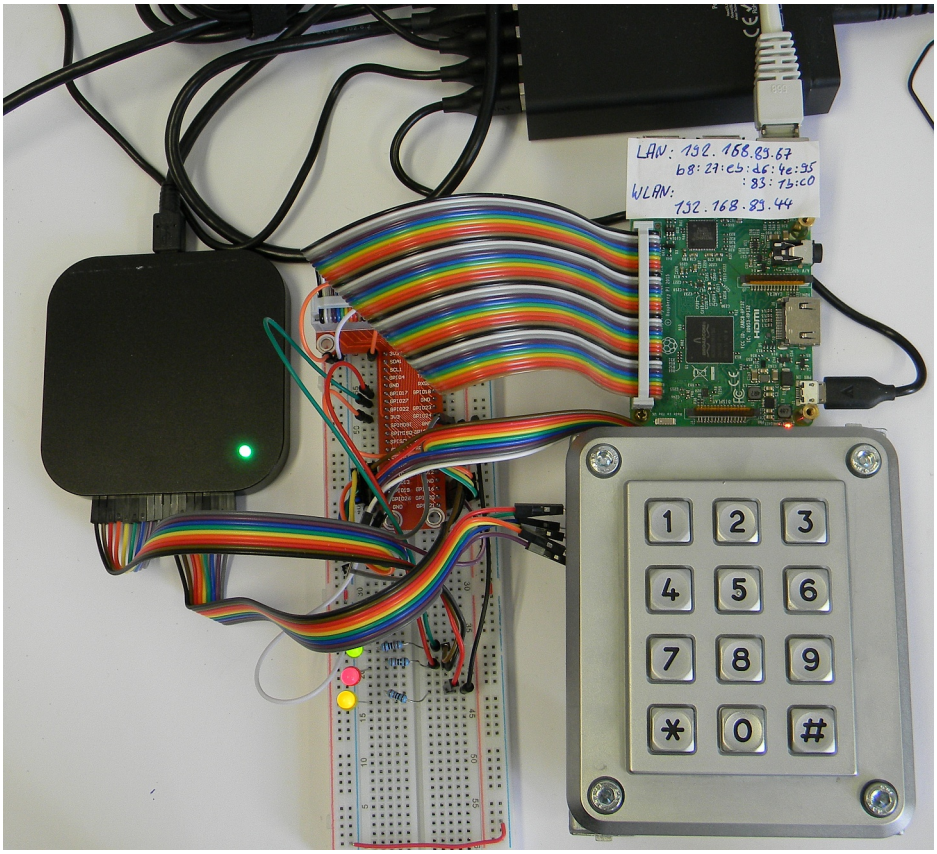


Fig. 8: Direct attached IO

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 within the same protective encasement are an acceptable exception.

A real use case had three LEDs and a 12 keys matrix EOZ Clavier S.series, 12 touches. Imagine the hardware set-up as figure 8 without the breadboard and logic analyser in in a flush-mounted wall box.

The key matrix was directly attached to 7 GPIO, see listing for the concrete set-up. 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 9: Defines and structure for exemplary12 keys matrix EOZ Clavier S.series

The device's IO was controlled by a next tier system via Modbus. 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 strength can be reduced down to 8 mA.

For all the Pi GPIO pins it is essential to keep the voltage (Upin) in the range:

$$0V <= Upin <= Ub <= 3.3V$$

where Ub is the processors actual supply voltage (Pins 1 & 17). For binary inputs best use contacts (as in fig. 8, page 53) or open collector/drain transistors respectively optocouplers to ground (pins6, 9 &c.). By enabling the input pin's pull up resistor you need no extra components.

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, 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 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 (or output 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

Relays are one way to handle remote actors, high power and to provide insulation. But, 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 with optocoupler isolated control input or
- use transistor circuitry to switch relay coils,
- ideally isolated by optocoupler input.

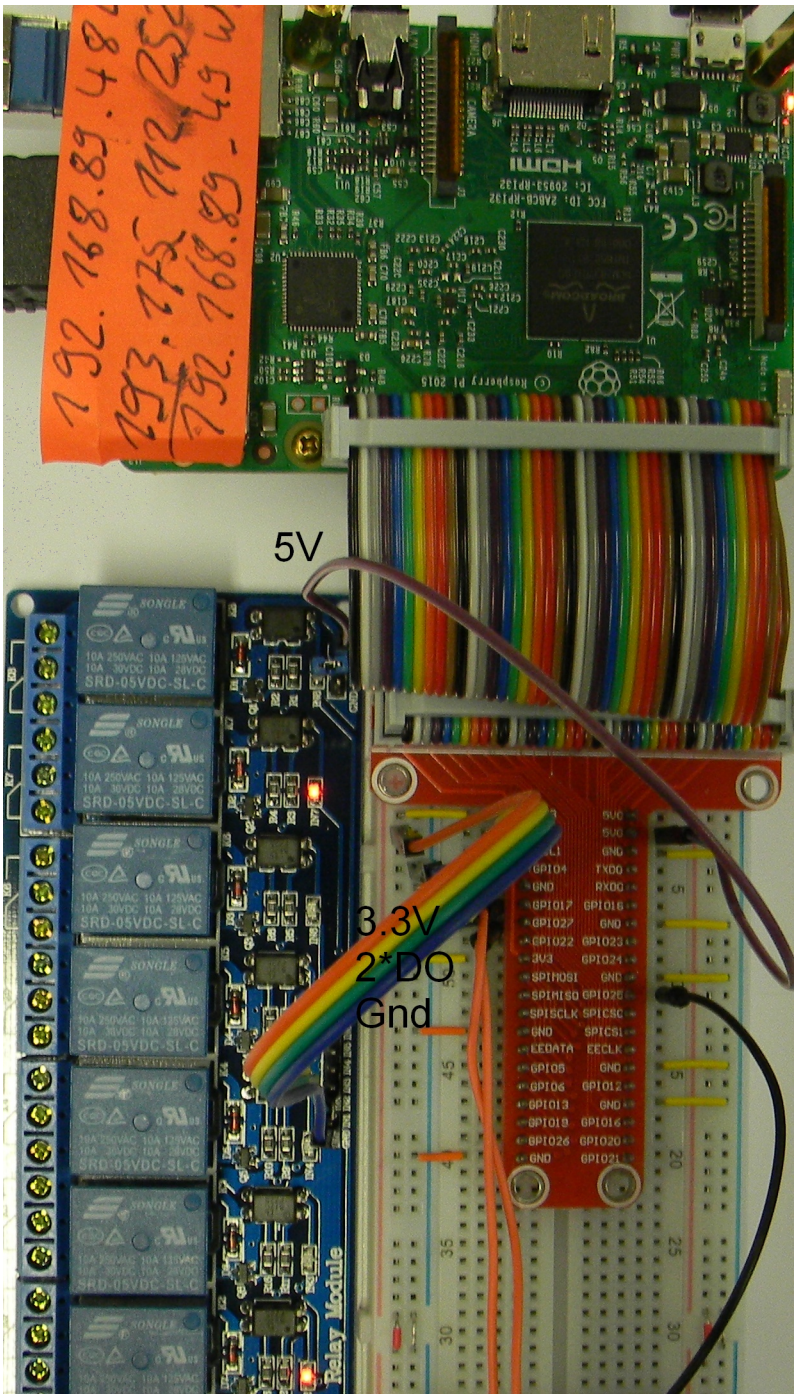


Fig. 9: Eight relays module 10A

The break out board is used for attaching test equipment (logic analyser). It is omitted in a production system, of course.

The latter solution comes quite handy in (semi-) professional modules, like in figure 8, page 53. Figure 9 shows a module with eight 5 V relays, each well controllable by Raspberry's GPIO. The separate 5 V supply for the relays may come from a separate source, or 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 9).

Figure 9 shows an eight relays module with 10 A changeover contact 250 V~ or 30 V=.*) Obviously it has one status LED per relay powered by the 3.3 V side (OK). These LEDs are red, which violates a rule: In all professional process control standards red means error/fault.

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

Similar higher power relay modules, best with 12 V coil supply, are also available and have been used in production systems.
Hint *): We recommend NOT to use a relay like the ones in figure 9 promising 250V * 10A~ load for a 230V~ load of some 100W or more. In that cases we use a DIN rail contactor with an interference suppressor (RC-unit) on its coil.

Power transistors

N-channel power MOS-FETs or npn Darlington transistors or Darlington arrays may be used as (open drain, open collector) N-switches. The transistors would have to be placed near the Raspberry and can be connected directly (gate, array input) or via a fitting resistor (base) with a GPIO pin. (Load M and Raspberry's Gnd have to be the same than.)

Depending on the transistor / array type and a suitable supply and grounding layout, the loads may be in the range of 400mA ..10A and up to 60V. The loads and or their supplies can be placed quite separated from the Raspberry.

Note: There are (probably Chinese, totally undocumented) single power MOS-FET modules offering a red LED to signal on (nice, except the red) but featuring no current or temperature protection at all. Under certain circumstances, this might be acceptable.

Nevertheless, most power MOS-FETs, as the IRF520, need >= 4 V gate voltage to switch fully on. That a 3.3 V powered µC cannot deliver!

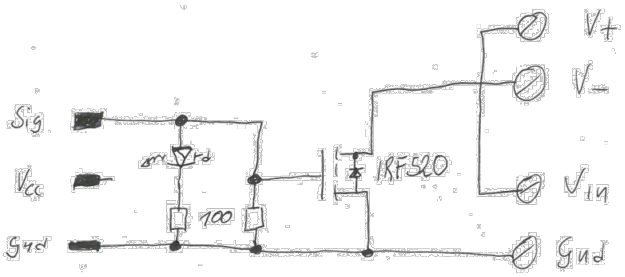


Fig. secret: Chinese module's circuit diagram

Replacing the power MOS-FET by a small lo power “signal” one (if sufficient) heals the switching problem. And the 100 Ohm gate to source resistor and the LED can / should be omitted or made larger if gate on voltage is critical.

Hardware for serial communication – RS485 (232)

As the Raspberry Pis 3 and 4 have 4 USB, Ethernet, WLAN and with some extra software/configuration even Bluetooth, there's seldom need for extra communication modules. One exception may be Modbus (page 33) over RS485/323 often found in quite interesting equipment, like heat pumps, gas ovens and "smart" power meters, to name just a few as well as EPROM programmers.

Many DIN rail-mounted one or three phase energy meters come with a so called S0 bus. That's just an optocoupler or switch output giving 300..1000 (as configured) 100 ms closures per kWh. These may 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 century old 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. Do NOT use S0 if the meter in question offers Modbus.

Better electronic meters offer a real bus connection, usually at a slightly higher price. And, usually in this and some other fields, that will be Modbus over RS485or seldom RS232. 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 much more.

In our context, a process control set-up with those devices would be a RS485 bus with the Raspberry as Modbus master / client and one or many of those devices as Modbus slave / server.

RS485 is a serial differential 1..5V two wire serial link, half-duplex link allowing one master and multiple slaves. RS232 is a full duplex +-12V two wire (send and receive) serial link allowing just two devices. Both use 8 bit UART bytes + start and stop bits. To get the RS485 or the RS323 hardware signals with their respective signal levels and meaning from a UART's TTL one can

- attach a TTL to RS485 / RS323 converter to Raspberries standard serial link:
  UART: TxD = GPIO14 = pin8; RxD = GPIO15 = pin10 like that shown in figure 10, or
- omit the UART and use an USB to RS485 / RS323 stick (not considered here *)).

Note *): The USB stick solution on a Pi is commonly reported to bring driver problems and, if finally brought to work, causing resource conflicts and system crashes. The reason is Raspberry's USB link already being overused / misused for other build-in USB to XYZ converters.

Hence, one would stick with Raspberry's standard UART (pins 8 and 10). Besides having to deal with the application/Modbus related tasks, we have to face two problems:

- The sheer simple usage of this standard UART is more difficult than it should really be; see "**Serial impertinences**" below.
- With the half duplex, multiple slaves RS485 only: The TTL to/from differential signal converter (chip, MAX485, e.g.) needs a transmit enable (TE) signal.
  Note: On the full duplex RS232 send and receive are separate signals/wires.

When having the UART working, 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 of that simple kind are offered for less than one Euro. Do not confuse those with the much better module shown in figure 10.

Problem with the minimalistic aproach "extra GPIO output for TE" (not figure 10) is: One is left alone with task to generate/program the "transmit enable" signal (= "DE" on MAX485).

As we have logically a one wire half-duplex bus, a sender must

- apply "transmit enable" before the first start bit of its (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 – usually better – always on.

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

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

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

This "forwarding of covered GPIO" should be taken for granted but is not. There are "shields" with the 40 pin connector using supply and two IO – and burying all else.
Figure 10, page 58, shows this "GPIO forwarding" for a relays module as example.

Remark: In all our Modbus applications we used the module shown in figure 10. A version with two sets of "A, B, Gnd" screw clamps instead of the never used sub-D would have been ideal. The RS323 version of the module very much look alike figure 10 and the sub-D connector is quite common here; we use this variant, too.

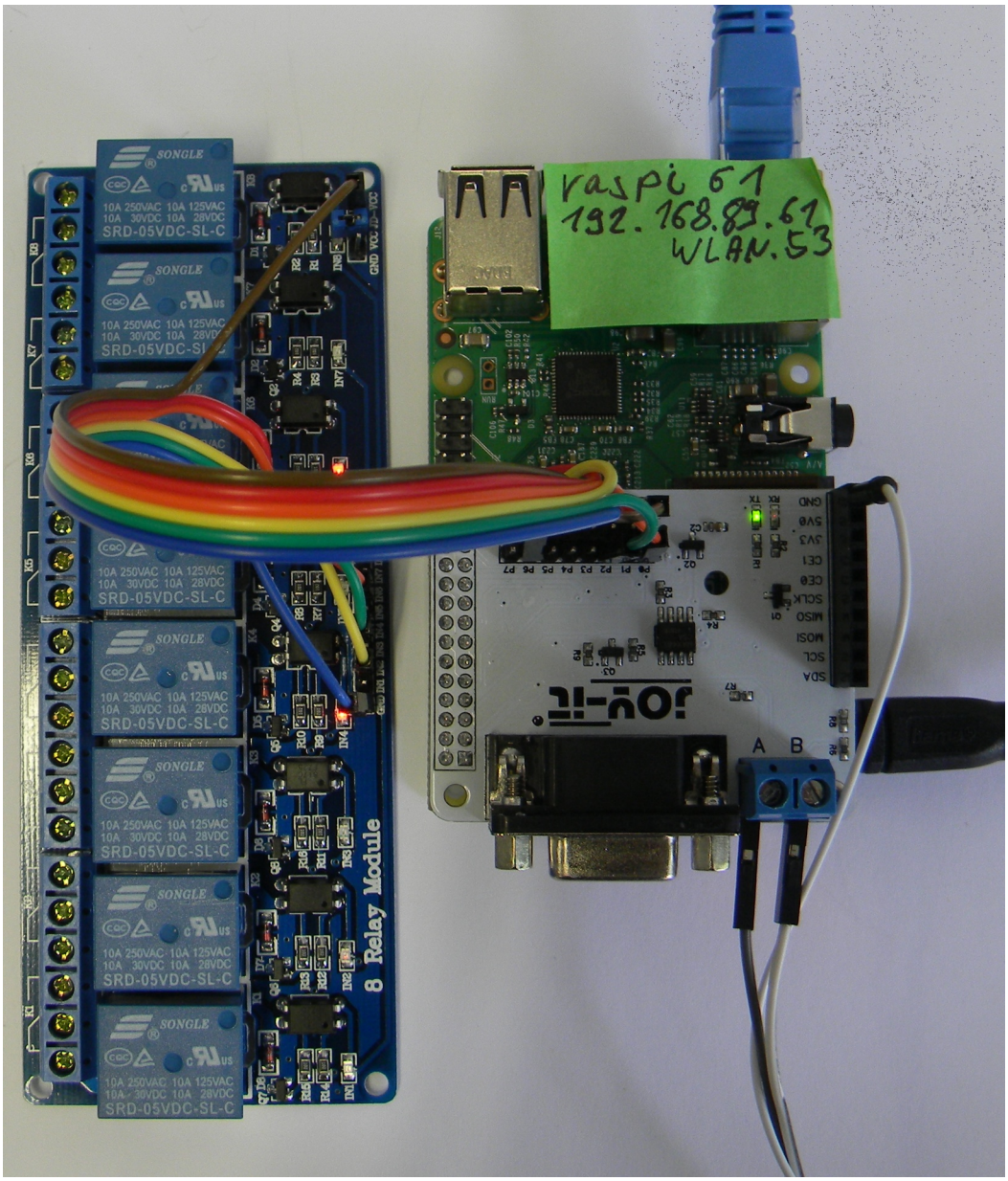


Fig. 10: RS485 module with automatic DE (for MAX485), i.e. transmit enable as well as "forwarding" the GPIO pins covered by the module, used in this example for relays

Serial impertinences

Using Pi's standard UART (GPIO pins 8 & 10) brings a lot of pitfalls and crashes (making it a bit hard not to assume evil intention). Using this UART has two pre-conditions:

A) a1) Enable the serial interface and a2) disable serial console by raspi-config and a3) add `enable_uart=1` in `/boot/firmware/config.txt` formerly `/boot/config.txt` .

B) In your software use the right device name. This, in Linux, is the name of a pseudo file in directory `/dev/` .

Disabling the serial console (a2) seems to require disabling the serial interface (not a1), too, and re-enabling it (a1) afterwards. Don't forget / check a3). Every (sub-) step seems to require reboot. A) or a part of it was seen to be killed, probably by OS updates or installations.

Choosing the right device name is, of course, essential. One would expect a UART present at the same pins on all Raspberry Pi types and available on all Raspbians and predecessors should get the same name in all history. But over the time we had to use /dev/ttyAMA0, /dev/ttyS0, /dev/tty, dev/serial0, /dev/serial1 and more.

You may get stupidly long list of candidates by

```
dir /dev/ | grep tty
```

and the check probable ones by

```
 stty < /dev/ttyS0
speed 9600 baud; line = 0;
```

The example output is a hit, seeing “permission denied” is a miss.
We had a functioning (hard-coded) `/dev/serial0` being killed (by updates?) and to be replaced by `/dev/ttyS0` *) or other such changes in a lot cases to be detected just by log files reporting communication / Modbus errors.
Remark*): Really `tty` = teletype? Which century are we in?

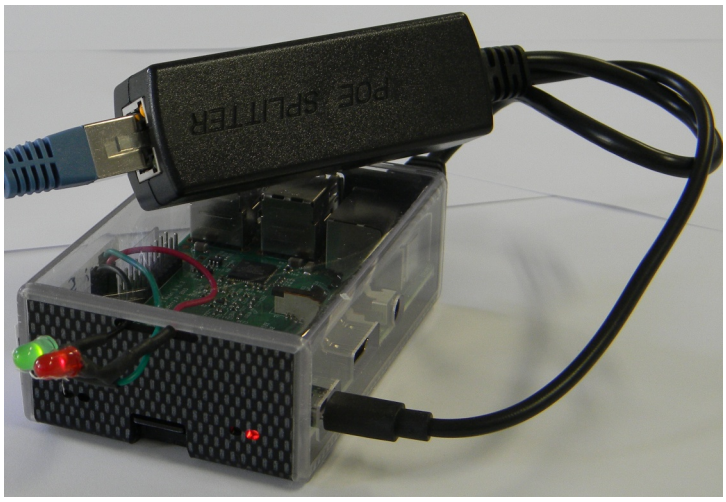


Fig. 11: PoE splitter 5V, 2..4A

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 often controlled via LAN. Then getting the 5 V, 1.3 A is an extra complication, often underestimated. IEEE 802.3af Ethernet switches can provide power over Ethernet (36..57 V, up to 25 W). An on site step down converter then provides the 5V.

IEEE 802af splitters to "LAN without power" and "5V on microUSB", see figure 11, are on purchase for about 13 €. We have some of those in 24/7 use without any problems. And do not use non IEEE “solutions” without converter to feed the 5 V from the other end of the LAN cable.

Warning on Raspberry’s own PoE:
Raspberry Pi3+ (the “+” is new), the Pi4 and the Pi5 promise PoE, but they have not. It requires a special extra module, as expensive as a Pi 3, and an extra patch cable from the bulky module to the Pi. Hence, the P4 PoE solution is in no way better and more expensive than the figure 11 approach.
Note: Modules of the kind show in figure 11 are also available with a USB-C power supply connector for the Pi 4 and 5 as well as 12V output for other purposes.

Warning: For a Pi5 the standard 5V, 2.4A is not sufficient. Use a 4A type.

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).
Note: For micro-controllers, like e.g. most Atmel AVR 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 and the like. 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.

For process control communication (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, like Modbus or MQTT.

1-wire

1-wire is a bus and protocol specification allowing several IO devices attached to one IO pin. For one 1-wire bus on a Raspberry the default is GPIO04 (Pin7). Another pin may be configured as well for multiple 1-wire buses. But it is good practice to stick to the default, when feasible.



Fig. 12: Robust 1-wire temperature sensor -40...25°C , stainless steel wires black: ground; red: +3.3V optional supply; yellow: signal

1-wire devices do have a Vcc pin for 3..5V; 5V is allowed even when the bus is limited to 3.3V. By providing a 4K7 pull up resistor to 3.3V one may even omit the Vcc wire to the devices; they can live from the "One" bus wire alone. This feature is called "parasite power" and done by feeding an internal Vcc buffer capacitor by both the external Vcc pin and the I/O bus pin via decoupling diodes. Every 1-wire device gets a unique factory 64-Bit registration number/address starting with a family/type code (0x28 for our thermometers). If a service person exchanges a defect 1-wire thermometer you have to change an address in your code.

Available 1-wire ICs respectively devices are

- DS1822 thermometer -40°C..+125°C; 1mK resolution, accuracy 2K; 0x28
- DS18B20 thermometer -40°C..+125°C; 1mK resolution, accuracy 0.5K, see fig.12 0x28
- DS2430 small EEPROM 0x14
- DS2408 8 very small power I/O pins; input or open drain 0x29
- DS2413 2 very small power I/O pins; input or open drain 0x3A

For process I/O with Raspberries the last two, "digital IO via 1-wire", are not attractive. Better use GPIO directly or have a ESP8266 module do it via MQTT. And here would be no use case for an extra EEPROM as we can hibernate data on the SD-card in the OS's file system.

With Raspberry's lack of analogue input, the 1-wire temperature sensors in all their variants may come very useful. One can have the cheap naked IC as well as a device with 1..5 m cable sealed in a stainless steel tube fitting in the standard thermometer inlets of industrial tanks. When heating water with surplus solar power, e.g., one should read the temperature for safety and comfort limits.

1-wire devices are accessed by Linux's device as file approach, the behaviour being implemented in kernel modules (.so files). From the application's point of view this approach is orthogonal to the "use I/O pins directly or driver software approach". To make it happen, i.e. have the files and the kernel modules for 1-wire + thermometer available, we have to (sudo) modify two configuration files. In /boot/config.txt (old Raspbian) or in /boot/firmware/config.txt add (after ‘[all]’:

```
dtoverlay=w1-gpio  #  default GPio4-pin7  otherwise:dtoverlay=w1-gpio,gpiopin=x
```

Adding anything in /etc/modules is obsolete now. Alternatively you may change the interfaces by:

```
/sudo raspi-config
```

After re-boot one 1-wire is ready on the default pin *1). When a sensor is connected find it by

```
dir /sys/bus/w1/devices
```

```
lrwxrwxrwx root root 0 2025-12-21 17:11 w1_bus_master1 -> ../../../devices/w1_bus_master1
lrwxrwxrwx 2025-12-21 17:11 28-0206917706a2 -> ../../../devices/w1_bus_master1/28-0206917706a2
lrwxrwxrwx 2025-12-21 17:11 28-0113176afdb6 -> ../../../devices/w1_bus_master1/28-0113176afdb6
lrwxrwxrwx 2025-12-21 17:11 28-01131746e6fc -> ../../../devices/w1_bus_master1/28-01131746e6fc
```

Here 28-0113173ecfcc points to a thermometer's directory; name is then unique serial number. Optionally, resolve the symbolic links *2) down to the thermometer's file w1_slave to save time.

A read of this "file" by code or command

```
cat /sys/devices/w1_bus_master1/28-0206917706a2/w1_slave
```

reveals an actual measurement as a two line text:

```
1d 01 55 05 7f 7e 81 66 78 : crc=78 YES
1d 01 55 05 7f 7e 81 66 78 t=17812
```

If the first line ends with YES the last number in the second line is the temperature in m°C, that is 17.8°C in the example. If the file isn't readable or if YES is not on its position consider it an error.

Note *1 : If a GPIO is thus assigned to a kernel modul, never touch, configure, read nor write it directly. The pigpio(d) library / daemon promises not to do so when not explicitly requested by user code.

Note *2 : Do not let time critical code follow symlinks. It's quite a task and probably not optimised away.

A read of this "file" by code or command

```
cat /sys/devices/w1_bus_master1/28-0206917706a2/temperature
```

reveals an actual measurement (17.8°C in between) quite simple as pure number on a one line text:

```
17875
```

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 with consistent numbers and units. 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 application's 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 total failure.
It may be astonishing how “hard” that can be in some environments.

Wanting PLC / cycles and with a Raspberry Pi3 / Raspbian lite doing just process IO and related communication we can well implement 1ms cycles. Going faster might work but would be daring.

Note *) on common process IO standards: The same is true for hardware, electrical signals, maximum ratings of IO, EMC etc. Most sold IO extensions, shields etc. are totally inadequate in this respect.

Absolute timing

The approach in our very first LED blink examples (listing 2, page 21, e.g.)

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

never gives an exact timing. 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 6'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 6: 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 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 again will give comparable results on different run-times and loads. Experimental outcomes are:

- A well configured (specialised, single purpose, single use) Pi OS lite on a Pi3 is suitable for hard real time process control and 1 ms cycles.
- A graphical (non lite) OS in all our use cases never was and never will be.
- Using interrupts may also overthrow predictable / acceptable latencies and kill inter thread co-operation. Refrain from using them.

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 144 ms too long (~3.5 s/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).
Note: In the (latest) Jessie distribution `CLOCK_REALTIME` and `ABS_MONOTIME`'s clock will be NTP tuned. Own inventions as with earlier Jessie’s (or libs) aren't required any longer.

Cycles and threads

For PLC like cyclic execution we offer an 1 ms, 10ms, 20 ms, 100 ms and 1 s cycle by library 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.
Note: In a minimal runtime for AVR µControllers we based a similar solution
(proven 24/7 since over 9 years) on Adam Dunkel's protothreads.
As protothreads are well suited for GCC, they could have been used too. But with full grown Linux runtime on a multi-core processor it's strongly recommended to use the runtime's own threading (pthreads) and signalling system instead. See sysUtil.c and sysUtil.h.

Threading and synchronising

As most process control must handle multiple asynchronous tasks we utilise Linux threads for

- several cyclic tasks
- a central task for organising all timing, i.e. organising
  - 1ms and 100ms cycles and multiples of it as well as
  - time and date
- all else tasks the application in question requires.

The main() method will have to parse arguments, provide all necessary resources and make and start all all those threads. It may serve as one of the threads – or just sit suspended waiting on the other threads end, as show in this excerpt:

```
int main(int argc, char * * argv){
  ::::::
  for (;;) {
    ret = getopt_long (argc, argv, "h?v", longOptions, &optIndex);
    :::::
  } // for over options
  ::::
  if (theCyclistStart(290)) { ::::: } // start the central timing thread
  on_exit(onExit, NULL);   // register exit hook
```

```
  signal(SIGTERM, onSignalStop); // register signal hook
  :::
  signal(SIGQUIT, onSignalStop);
 ::::
   ret = pthread_create(&threadRS485Mod, NULL, rs485ModThread, (void*) NULL);
  ::::
  ret = pthread_create(&threadRelays, NULL, relaysThread, (void*) NULL);
  :::
   if (theCyclistWaitEnd()) { ::::::: } // wait for the other threads to end
  :::::
} // .main(int, char * *)
```

For full details see the project files sysUtil.c, sysUtil.h &c..

Preventing two or more threads to enter critical code – critical mostly by using common variables / structures or other resources – is done by a common mutex guarding the critical code:

```
   int ret = pthread_mutex_lock(&comCycMutex);                  // under lock
::::::   // the critical code
   int ret2 =  pthread_mutex_unlock(&comCycMutex);            // release lock
```

Under such mutex lock a thread may wait (optionally with timeout) for a signal from another thread. The wait call implies an unlock and re-lock for the (necessarily) used mutex:

```
  int ret = pthread_mutex_lock(&comCycMutex);                  // under lock
:::::: // there may be optinal critical code before and after the wait call
    ret = pthread_cond_wait(&cycTask->cond, &comCycMutex); // unlock / lock
  int ret2 =  pthread_mutex_unlock(&comCycMutex);            // release lock
```

Under the same mutex lock another thread may send the signal to one or all waiting threads:

```
  int ret = pthread_mutex_lock(&comCycMutex);                  // under lock
    ret r= pthread_cond_broadcast(&cykTask->cond); // wake threads; unlock / lock
  int ret2 =  pthread_mutex_unlock(&comCycMutex);            // release lock
```

Using this schema detailed in the example, threading and the necessary locking, synchronising and signalling is almost as easy as with Java. Limitations with Raspian are no pthread_yield() and no working thread priorities. The first, if needed, may be substituted by sched_yield() and the priorities by fine granular organising of the process control tasks.

Co-operating applications

Besides the cyclic and organisational tasks (threads) a process control application may have to handle HMI output, output to files (log, error, CSV etc.) and much more. Except for very small and simple applications it seems logical to move some (or all) tasks not belonging to core (real time) process control and process IO to other programs on the same computer.

On the minus or costs side are

- inter process communication

and implicitly or explicitly

- inter process synchronisation.

On the plus side we see

- the core process control / IO program getting smaller and
- kept as simple as possible,
- having less or ideally no start parameters (start options)
- being quite suitable to be started as background service (on boot)

and hence as depending on less resources (files, HMI and else)

- reliably running 24/7.

Shared memory

To get to this separation of tasks in several co-operating programs we need a be-directional communication from/to the process control program and all its helper programs. Here we could use pipes, sockets or queues, e.g. But here the programs tend to be tightly coupled as consumer and producer and one's failure may severely impede the other. For processes on the same machine shared memory is the easiest way to avoid such hard dependencies and, as well, allow flexible one-to-some relations.

A good approach for our (process control) use case is to organise all

- process input,
- process output,
- command input and
- status output

in one well-formed structure (of structures) called valFilVal_t in examples below. The structure must be clearly defined (and hence centrally modifiable) in one .h-file common to the application – i.e. all its processes respectively programs.

This common structure is held and actualised in shared memory. Each co-operating process may work on the shared memory or on a local copy. The latter approach – doing the work on a local copy and copying the parts needed / modified from / to the shared memory – is better in most cases, as almost each access to shared memory will have to be done under mutual exclusion by semaphore (not to be confused with above mutex) lock.

As the memory is shared between the cooperating processes they need a common handle or name, which in Raspian is an arbitrary int (32bit). The same is true for the semaphore set. This should be put in a .h-file common to the co-operating processes:

```
#define  ANZ_SEMAS 3                    //!< Semaphore set of three (3..10)
#define  SEMAPHORE_KEY      0xcaef1924 //!< Semaphore unique key "Käfig24"
#define  SHARED_MEMORY_SIZE 512        //!< Shared memory size 512 bytes
#define  SHARED_MEMORY_KEY  0xbaffe324 //!< Shared memory key "Buffer24"
```

Now a program can make or get the shared memory by:

```
int shMemId = -1; // shared memory ID (initialised none)

static valsSharMem_t * valsSharMem = (void *)-1; //!< pointer to shared memory
  //  ^ common data structure;   ^ initialised as error/none

   :::::
  shMemId = shmget((key_t)SHARED_MEMORY_KEY, SHARED_MEMORY_SIZE,
                                                  0666 | IPC_CREAT);
  if (shMemId == -1){ /* error handling */ }
  shMemPt = shmat(shMemId, (void *)0, 0);
  if (shMemPt == (void *)-1) { /* error handling */ }
```

This is best seen in more detail in the files weShareMem.h and weShareMem.c showing the detaching and destroying of shared memory also.

Using the shared memory attached in a program is best done in one place by e.g.:

```
  memcpy(valsSharMem, &valFilVal,
         sizeof(valFilVal_t)); //  put process values for other programs
  valsSharMem->copInCnt = copInCnt;  //  get commands from other program
```

Remark: An exception from guarding accesses by semaphores etc. to shared memory may be a single 64bit access. 64 bit accesses may/should be atomic on a Pi at least with a 64 bit OS. But beware: It has to be a single read or write (no modify) and a variable in question must nor cross memory cell borders (as my happen in multi-type data structures by compiler design). Double check everything if temped!

Semaphore sets

Of course, this putting values to and getting values from shared memory has to be guarded by a semaphore of a common (shared) set. Additionally, having put values for other programs can be signalled to consuming programs by another semaphore of the set.

Mutexes are private to one program and act as binary locks between this program's threads. Between threads they guarantee exclusive access to variables and other resources. And, under such lock, signals between threads by (two) special functions to be used under lock only.

Contrary to mutexes semaphores

- come in a named shareable set to
- work between programs (processes) and
- are not binary but have a value range of 0..max. (max is probably 96766 with Raspbian).

A basic usage of a semaphore is a binary lock when

- decrementing to 0 is the lock operation and
- incrementing (under lock!) to 1 is the unlock operation.

This works because the decrement blocks as long as the value is 0. The blocking hence waits until another process unlocks. As this can be quite long or worst case for ever the lock operation can and should be guarded by a timeout by using semtimedop() instead of semop(). See the handling in weShareMem.h. and .weShareMem.c..

Such binary lock is used to guard the access to all co-operating programs shared memory using one (number 0) semaphore of the common set.

Our second usage is signalling from program to a limited maximum number (say 9) of other programs by using other semaphores (1...) of the common set. The other programs wait for the signal just as above by lock best with a generous timeout.

The signalling program (here under semaphore number 0 lock) uses semctl() (not semop()!) to set the signal semaphore(s) to the maximum number (say 9) of other waiting programs. Then their lock will immediately succeed. After a time agreed upon (say T_{sig} = shortest signal period / 6) the signalling program uses semctl() to set the signal semaphores (1..) to 0. After a successful signal lock the other programs will wait > T_{sig} before waiting on the next signalling. (Again see the examples named above.)

Watchdog

A vital 24/7 control application will be started at boot to resume its tasks after power outages or at switching on the facility. Such (well written) useful, cyclic etc. process control application should run 24/7 – and normally will.
But from time to time it might might violate its strict timing (cyclic) requirements or even stop a vital thread or completely. If this happen a restart of this application directly or indirectly by reboot is appropriate.

As it is good practice in process control, this rescue reset / reboot should be done by an OS independent HW watchdog when the process control program's cyclic operation ceases. In our set-up we'd trigger such watchdog in every other run of the 1-s-cycle.

Luckily, Pi's BCM processor (BCM2835) has such watchdog *), which is exposed as a file.
By dir /dev/watchdog we see:

```
crw------- 1 root root 10, 130 2018-08-16 17:17 /dev/watchdog
```

Writing any character except 'V' to /dev/watchdog will trigger it, causing a reset after 16s, when not re-triggering it within this interval. Hence, triggering every 2 s, as said above, will fit. Writing a 'V' will disarm the watchdog. The process control program would do this at normal shutdown and after (!) bringing all process output in idle (off) state.

As seen from the dir above, only root can write the watchdog device. This “crw-------” will be restored at boot. As we won't use sudo to run the process control application we'll have to

`sudo chmod a+=rw /dev/watchdog`
before starting process control. Hence, we sudo crontab -e the following:

```
@reboot  sleep 5 && chmod a+=rw /dev/watchdog
```

to have “crw-rw-rw-” after reboot.
Finally place (distribute) the following lines of code at the appropriate places

```
static int watchdog;
   watchdog = open("/dev/watchdog", O_RDWR); // in initProcessIO()
   write(watchdog, "V", 1); // disarm wachtdog
   close(watchdog); // and close  at releaseProcessIO()

   write(watchdog, "X", 1);  // in 1s cycle (every second time)
```

By the following commands you may **test the watchdog:**

```
echo X > /dev/watchdog # turns on / triggers
echo V > /dev/watchdog # disarms / turns off
```

If you wait long enough after a trigger before the next trigger or a disarm your Pi will shutdown and reboot. *)

Notes *): **No watchdog with trixie**

This all worked very well with all 32 and 64 bit Raspbians and Pi OS up to bookworm. Beginning with trixie the operating system claims the BCM2835 watchdog for own purposes – probably for monitoring an OS thread. This feature seems badly documented as are all found internet recipes to recapture our watchdog – none of which, of course, worked, reliably. Our only solution: stay withbookworm.

GCI programs

As we did provide the basic CGI configuration for Apache 2.4, already, we just test it by a GCI program written C (greet.c) to deliver a simple complete html page:

```
#include <stdio.h>
int main(int argc, char * argv[]){
  printf("Content-type: text/html\n\n"); // Every CGI delivery starts with
  printf("Hello, World.");              // content-type followed by content
}
```

On the workstation cross-compile the program by
`arm-linux-gnueabihf-gcc -o greet greet.c`

and ftp the binary greet it to /var/www/myPi/cgi/ . Test is by browsing to http://myPi/cgi/greet.

`sudo a2enmod cgi ## if not yet enabled`

A more complete version of the greet.c program (listing 7) is to show the line of actions (and the ill URL coding) in more detail.

```
/**
 * \file greet.c
   $Revision: 76 $  ($Date: 2017-12-01 19:43:45 +0100 (Fr, 01. Dez 2017) $)
   Copyright  (c)  2017   Albrecht Weinert
   weinert-automation.de      a-weinert.de

   cross-compile  by:
     arm-linux-gnueabihf-gcc -o greet greet.c
   local compile  by:
     gcc -o greet greet.c
 */
```

```
#include <stdio.h>  // printf()
#include <stdlib.h> // getenv()
#include <string.h> // strstr()
#include <ctype.h>  // isdigit()
#include <stdint.h> // uint32_t

int urlDecode(char * str){
   char *ptr = str;
   char c = * str;
   char c2 = *++str;
   for (; c; c = c2, c2 = *++str) {
      if (c != '%') { *ptr++ = c;  continue; }
      char c3 = *(str + 1);
      if (!isxdigit(c2) || !isdigit(c3)) { *ptr++ = c;  continue; }
      uint8_t v1 = c2 <= '9' ? c2 - '0' :
            (c2 <= 'F' ?  c2 - 'A' + 10 : c2 - 'a' + 10);
      uint8_t v2 = c3 <= '9' ? c3 - '0' :
            (c3 <= 'F' ?  c3 - 'A' + 10 : c3 - 'a' + 10);
      *ptr++ = (v1 << 4) | v2;
      str += 2;
      c2 = *str;
   }
   *ptr = '\0';
   return 0;
} // urlDecode(char *)

int main(int argc, char * argv[]){
   printf("Content-type: text/html\n\n");
   printf("Hello World! <br />\n");
   char * browser = getenv("HTTP_USER_AGENT");
   if(browser != NULL) {
      if (strstr(browser, "Chrome") != NULL) {
         printf("You're using Chrome like most of use.<br />\n");
      } else {
         printf("Your browser: %s  <br />\n", browser);
      }
   } // got browser info

   char * query = getenv("QUERY_STRING");

   if(query != NULL) { // got query string
      printf("Your query: %s <br />\n", query);
      urlDecode(query);
      printf("Decoded : %s <br />\n", query);

   }  // got query string
} // main(int, char * [])
```

Listing 7: Extended greet.c GCI example.

Apache's mechanics behind CGI are seen best by listing 7:
When a requested URL points into a directory configured with "ExecCGI" and the filename points to an executable, Apache will

- provide request and browser information in an environment
- run the executable with that environment
  (Note the ill coding of the query string at blanks or non-US-ASCII characters!)

and

- deliver the redirected standard output to the requesting Web-client.

From this course of action it will be evident that the CGI program in question will be run once per request. Hence within itself it can't hold (session) state.

The little and the extended (listing 7) example delivers a (simple) complete web page. This allows to dynamically generate pages.

More interesting is to have a static html page's JavaScript send commands (in the query string) and get information in form agreed upon (plain text, XML, JSON or what ever) and use this data to dynamically modify the page displayed. This approach is called AJAX, and every modern browser on any platform can handle it. In our context its most important advantage is

- to reduce the communication load and
- put the load of filtering, rendering and displaying information away from the little (Pi) machine to the client.

If this dynamically getting data happens often or regularly it's evident, too, compiled C as CGI executable being better (for poor little server) than interpreted languages or worse scripts.

A point against PHP, and indirectly for C, is PHP's handling of Linux shared memory and semaphore sets being just buggy, while a C program wouldn't have the slightest problems in this respect. One may view a C CGI as just another one in a band of co-operation control processes co-ordinating themselves by semaphores and shared memory.

Data exchange with AJAX & JSON

The C CGI program forwards the commands received and get its data from the process control program as described above via shared memory with guards and signals by the common semaphore set. For the data forwarded to the requesting web page plain text and JSON has been used.

Getting Web data – cURL

The brethren of providing web content by C is getting it within a C program. One way is to install and use the cURL library on the Raspberry (Raspbian) and on the (Windows, Eclipse) development workstation. To install the library on the Raspberry do

```
sudo apt-get install libcurl4-openssl-dev
```

This brings the application curl, a wget equivalent, too. To locally compile and build a prepared application do:

```
g++ getLocalWeaterData.c -o getLocalWeaterData  -lcurlbd
```

To cross-work on the (Windows, Eclipse) development workstation ftp libcurl.so.4.3.0 from /usr/lib/arm-linux-gnueabihf to C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib\. And in C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\lib\ copy libcurl.so.4.3.0 to libcurl.so.

Also ftp all the complete directory curl from /usr/include/ to C:\util\WinRaspi\arm-linux-gnueabihf\sysroot\usr\include\. Now you can compile and build on the development workstation

```
arm-linux-gnueabihf-gcc  getLocalWeaterData.c   -o
  getLocalWeaterData  -lcurl
```

The executable made can be transferred to and run on a Raspberry with the cURL library installed.

Asterisk

Asterisk is a very powerful telephone system software. Using VoIP it needs no extra connection hardware besides (W)LAN – and it can be installed and run on a Pi. It can also act as one or more (software implemented) IP phones in the site's or home's telephone system (by Fritz!Box e.g.).

Such software phone may call one or more “real” phones on certain process events or alarms. A common use case is bringing door bells to the garden by portable phones. A software call is defined by a file, like e.g. /home/sweet/asteriskFiles/52call_tuer.txt:

```
Channel: SIP/629/**52
WaitTime: 19
Application: Playback
Data: /var/lib/asterisk/sounds/tuerklingel
```

This example calls number **52 for max. 19s and plays an audio “door bell” to the answering phone. On Asterisk that call is initialised by copying the respective file in the “outgoing” directory by

```
cp /home/sweet/asteriskFiles/625call_tuer.txt

/var/spool/asterisk/outgoing/
```

or in C by:

```
system("cp /home/sweet/asteriskFiles/52call_tuer.txt /var/spool/asterisk/outgoing/");
```

Note: Do not use home made copying loops to put a file to asterisk/outgoing/ .

Why not a “call VoIP” application

Just to call a phone once or twice a day, having the mighty Asterisk service running 24/7 on poor little Pi seem sheer overkill and a small “just call VoIP once” command line application the better choice. But in the end it was hard find usable ones and get them running. And (if working at all) they were dead slow. Up to 8 s delay to call are not acceptable.

Although a sleeping Asterisk seems a much heavier burden than a sleeping Apache, repentantly, we returned to Asterisk. On the other hand Asterisk never impeded the 1 ms process control cycle. Listing 8 shows an exemplary Asterisk configuration file for this minimal use case.

```
[general]
allowguest=no
port = 5060
bindaddr = 0.0.0.0
qualify = no
disable = all
allow = alaw
allow = ulaw
videosupport = no
dtmfmode = rfc2833
srvlookup = yes
localnet=192.168.178.0/255.255.255.0
directmedia = no
nat = no

register=klingByPi:secret#@192.168.178.1/629
;Die Caller-ID wird bei Anrufen auf kompatiblen Telefonen angezeigt.
callerid=klingelPi <629>

[629]
```

```
type = friend
insecure = invite,port
nat = no
canreinvite = no
; FritzOS / Fritzbox phone **629 user, name, password and the Fritzbox IP:
authuser = klingByPi
username = klingByPi
fromuser = klingByPi
fromdomain = fritz.box
secret = secretPW
host = 192.168.178.1

dtmfmode = rfc2833
context = incoming
disallow = all
callerid= Klingel
allow = ulaw
allow = alaw
transport = udp
```

Listing 8: Asterisk's sip.conf.

So the good side of Asterisk is: This works reliably.

The bad side is: This trifle took hours of trial and error and search in “documentation” and forums.

The result – and where we are

We can handle Raspberry's GPIO pins in C programs.

We can handle a bundle of IO libraries pigpio being the most promising one. And pigpio(d) is the only one we used for real process control applications in the end (and all others, including toys).

And we know how to cross-compile and cross-build from Windows (or Ubuntu) using make, Eclipse and SVN there. With the make tool and some include file wizardry it is possible to integrate the WinSCP.com file (Windows only) transfer to a target Raspberry in the automated make processing.

That means with one make call we can compile, build and document a program and transfer it to the target Pi.

C is still the lingua franca in embedded. On the other hand – under the same hard requirements and restrictions – alternative languages should be tried and considered. The first choice would be Java which has been experimentally used with some success; see [32].

Some actuators and sensors may directly connect to Raspberry's GPOI but often extra interfacing IO hardware will be needed. Note: Years ago, we used and tried piXtend which might be disappointing in the light of approach (putting all IO in an AVR slowly coupled to the Pi), price and promises. Nevertheless it opens the world of Codesys and Web interfaces to slow process control applications.

Low price semi-professional relay and other modules enable the Raspberry to control actuators with some power, including 230/400V equipment.

Modbus (via TCP IP and RS485) and MQTT enable more remote sensing and actuating. As MQTT attached periphery we can use bought and/or self-programmed ESP 8266 (or ESP 32) devices.

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/slaves as well as clients/masters. With Modbus via RS485 we attach one to four smart meters to our Raspberry and by RS232 a PV inverter.

With shared memory and semaphore sets we can have multiple (C) programs share data and status with or send commands to a process control program running 24/7 as service.

Such co-operating (C) program can as well be a CGI program run on request by an Apache 2.4 web-server. By this we can put any graphical user interface imaginable by HTML (5) and CSS on a user's browser in the process controlling Raspberry's LAN. Considering graphics on a Raspberry detrimental to even modest real time requirements, process control HMI is to be done by a Web interface via AJAX / GCI – and can very well be done so with acceptable response times.
or on another machine.

We can use barcode and QR code scanners attached via USB

We can use an Asterisk telephone server on the same Pi to signal process events to arbitrary phones directly or indirectly (Fritz!box) attached to the LAN.

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

The approaches described here allow real time applications like this in one Raspberry Pi3 or 4:

A home automation application featuring
- HMI via web interface (Apache 2.4),
- 2 smart three phase meters (RS485),
- five remote measuring and power switching devices (MQTT),
- three temperature sensors (1-wire) and
- a lot of locally attached binary actuators and sensors, i.e.
  relays, LEDs, buttons,
- fine grained heater power control by phase packet switching
  (using a solid state relay 100A/400V in the 20 ms cycle)

is running 24/7 for more than two years (after having an equivalent laboratory set-up for considerably longer).

A simple door bells and openers monitoring and logging application using
- Asterisk on the same Pi to forward door bell rings to telephones and
- an Apache 2.4 on the same Pi to show the log files.).

To sample and filter the four optocoupler isolated AC signals we take 4 * 5 equidistant samples in the 1ms cycle.

As said the C library used offers 1ms, 10ms 20ms 100ms and 1s cycles in a PLC like manner. It is recommended not to enable and use more than three of them. Slower cycles can easily implemented by sub-dividing a faster one.
And as also said: Do not use a non-lite Raspbian or Pi OS for real time process control in our sense. A graphical OS with graphic and pointing devices attached is the dead of all timing. And a graphical OS with graphic devices neither attached or used *) still kills the timing qualities the fast cycles and timestamps.

Note *): Why the hell should one install a graphic OS when never having screens or mice?
Isn't it just stupid? ²)

Note ²): Well, yes, but not always. Non lite Pi OSs offer some plug and play features. Putting a SSD or USB drive to a Raspbian lite is unpleasant.

Appendix

Miscellaneous commands

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

List all visible WLANs

```
sudo iwlist wlan0 scan
sudo iwlist wlan0 scan | grep "Cell\|Frequency"
```

```
          Cell 01 - Address: 2C:91:AB:40:EA:14
                    Frequency:5.26 GHz (Channel 52)
          Cell 02 - Address: 60:B5:8D:47:7D:55
                    Frequency:5.18 GHz (Channel 36)
          Cell 03 - Address: 2C:91:AB:40:EA:13
                    Frequency:2.437 GHz (Channel 6)
          Cell 04 - Address: 60:B5:8D:47:7D:54
                    Frequency:2.462 GHz (Channel 11)
```

Make consistent and comparable directory listings

Make a good file listing command command by:

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

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

```
alias dir='ls -lAh --time-style=long-iso'
alias diR='ls -lARh --time-style=long-iso'
alias sudo='sudo '
```

To have it for all users you may put it in a file /etc/profile.d/bash_aliases.sh instead.

Determine Pi's hostname

```
hostname
```

```
cat /etc/hostname
```

```
pi4play
```

or for a bit more information

```
hostnamectl
```

```
 Static hostname: demoPi
       Icon name: computer
      Machine ID: 35c36d21152447f985f117b22147d58e
         Boot ID: afee02c1b2b340b4a28af0aa8a3093e0
Operating System: Debian GNU/Linux 13 (trixie)
          Kernel: Linux 6.12.47+rpt-rpi-v8
    Architecture: arm64
```

Change Pi's hostname

```
sudo nano /etc/hostname
```

and reboot. At next login accept the new fingerprint. The change of the name per se is easy. Getting along with the consequences may get the hard part: DNS server / router, filezilla settings, scripted deployment (with make), … Any undetected use of the old name can cause trouble.

Determine OS and version

```
lsb_release -a
cat /etc/os-release
```

```
PRETTY_NAME="Debian GNU/Linux 13 (trixie)"
NAME="Debian GNU/Linux"
VERSION_ID="13"
VERSION="13 (trixie)"
VERSION_CODENAME=trixie
DEBIAN_VERSION_FULL=13.2
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

Determine Pi OS

```
hostnamectl
```

```
Static hostname: demopi
       Icon name: computer
      Machine ID: 9cf9feed34374cde9df58f311f35c148
         Boot ID: 0a7d7f20b37c4a6a91e2199399a4b2f4
Operating System: Debian GNU/Linux 12 (bookworm)
          Kernel: Linux 6.12.47+rpt-rpi-v8
    Architecture: arm64
```

Determine Pi OS 64 bit vs 32

```
getconf LONG_BIT
```

```
  64
```

This command shows the number of bits in a long integer type which is either 32 or 64.

Determine Pi RAM size

```
free -ght
```

```
               total        used        free      shared  buff/cache   available
Mem:           7.6Gi       189Mi       7.4Gi       3.5Mi       133Mi       7.4Gi
Swap:          199Mi          0B       199Mi
Total:         7.8Gi       189Mi       7.6Gi
```

Total/total 7.8GB in the example means a Pi with 8GByte RAM.

Determine Pi type

```
cat /proc/device-tree/model
```

```
Raspberry Pi 4 Model B Rev 1.4
Raspberry Pi 3 Model B Rev 1.2s
```

List installed libraries (best with | grep ….)

```
apt list --installed | grep libpig
```

```
WARNING (only with | grep): apt does not have a stable CLI interface.
                            Use with caution in scripts.
libpigpio-dev/stable,now 1.79-1+rpt1 armhf [installed,automatic]
libpigpio1/stable,now 1.79-1+rpt1 armhf [installed,automatic]
libpigpiod-if-dev/stable,now 1.79-1+rpt1 armhf [installed,automatic]
libpigpiod-if1/stable,now 1.79-1+rpt1 armhf [installed,automatic]
libpigpiod-if2-1/stable,now 1.79-1+rpt1 armhf [installed,automatic]
```

See all groups a user (pius e.g.) **is in**

```
id pius
```

```
uid=1001(pius) gid=1000(sweet) groups=1000(sweet),4(adm),20(dialout),
24(cdrom),27(sudo),29(audio),44(video),46(plugdev),60(games),100(users),
102(input),105(render),106(netdev),995(spi),994(i2c)
```

Add user to group

```
sudo adduser user group
```

Remove user from group

```
sudo deluser user group
```

Use nano without syntax highlighting

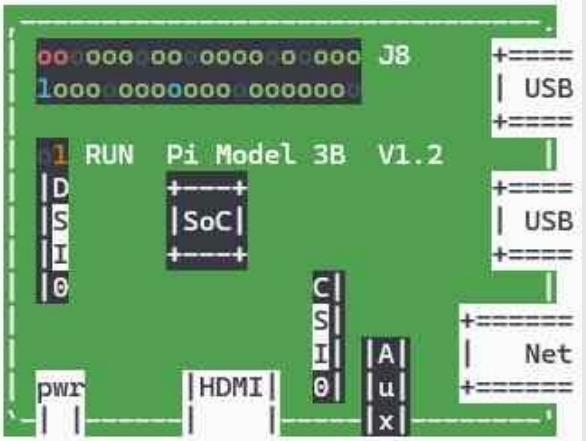
```
nano -Ynone myfile.txt
```

Get Pi's hardware interfaces and pinout

```
pinout
```

```
Description         : Raspberry Pi 4B rev 1.4
Revision            : d03114
SoC                 : BCM2711
RAM                 : 8GB
Storage             : MicroSD
USB ports           : 4 (of which 2 USB3)
Ethernet ports      : 1 (1000Mbps max. speed)
Wi-fi               : True
Bluetooth           : True
Camera ports (CSI)  : 1
```

```
Display ports (DSI): 1
```




```
J8:
   3V3  (1) (2)  5V
 GPIO2  (3) (4)  5V
 GPIO3  (5) (6)  GND
 GPIO4  (7) (8)  GPIO14
   GND  (9) (10) GPIO15
GPIO17 (11) (12) GPIO18
GPIO27 (13) (14) GND
GPIO22 (15) (16) GPIO23
   3V3 (17) (18) GPIO24
GPIO10 (19) (20) GND
 GPIO9 (21) (22) GPIO25
GPIO11 (23) (24) GPIO8
   GND (25) (26) GPIO7
 GPIO0 (27) (28) GPIO1
 GPIO5 (29) (30) GND
 GPIO6 (31) (32) GPIO12
GPIO13 (33) (34) GND
GPIO19 (35) (36) GPIO16
GPIO26 (37) (38) GPIO20
   GND (39) (40) GPIO21

RUN:
RUN (1)
GND (2)
```

← left figure is pure ASCII art

```
For further information, please refer to https://pinout.xyz/
```

Show all threads of a program

```
ps aux | grep rdGnBlink
```

```
sweet     1386  0.0  0.0   3440  1280 ?        Sl   Nov24   0:15 rdGnBlink
```

and with the pid (1386 in example) do

```
ps -T -p 1386
```

```
  PID  SPID TTY          TIME CMD
 1386  1386 ?        00:00:15 rdGnBlink
 1386  1389 ?        00:00:00 rdGnBlink
```

Show queues (none in example) **semaphores and sheared memory**

```
ipcs
```

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x25baffe6 0          www-data   666        1024       1
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x25caef19 2          sweet      666        3
```

Stop / start / restart web server

```
sudo service apache2 stop
sudo service apache2 start
sudo service apache2 restart
```

Act as Apaches user www-data

```
sudo su -l www-data -s /bin/bash
```

Copy a file preserving its date

On Windows: Use `robocopy` instead of `copy` or `xcopy`
On Linux: cp -p sourceFile copiedFileWithTheSameDate

Compare two files

On Windows: `fc file1 file2`
On Linux: `diff file1 file2`
On Linux: with visible report if equal: `diff -s rdGnBlink rdGnBlink01`

```
Files rdGnBlink and rdGnBlink01 are identical
```

Determine recursively all files containing “text”

```
grep -r "modbus_connect" D:\eclipCe18-12WS\stephane\libmodbus\
```

```
Binary file D:\eclipCe18-12WS\stephane\libmodbus\/.git/index matches :::::
D:\eclipCe18-12WS\stephane\libmodbus\/docs/modbus_connect.md:int
     modbus_connect(modbus_t *ctx);
::::
D:\eclipCe18-12WS\stephane\libmodbus\/src/modbus.c:    modbus_connect(ctx);
::::
D:\eclipCe18-12WS\stephane\libmodbus\/tests/unit-test-server.c:
     rc = modbus_connect(ctx);
```

Limit search to specific files

```
grep -r  --include=*.h "modbus_connect" D:\eclipCe18-12WS\stephane\libmodbus\
```

```
D:\eclipCe18-12WS\stephane\libmodbus\/src/modbus.h:
     MODBUS_API int modbus_connect(modbus_t *ctx);
```

```
grep -r  --include=*.c "modbus_connect" D:\eclipCe18-12WS\stephane\libmodbus\
```

```
D:\eclipCe18-12WS\stephane\libmodbus\/src/modbus.c:
          modbus_connect(ctx);
  ::::::  11 others :::::::
```

```
D:\eclipCe18-12WS\stephane\libmodbus\/tests/unit-test-server.c:
          rc = modbus_connect(ctx);
```

Add line numbers

```
grep -rn  --include=*.c "modbus_connect" D:\eclipCe18-12WS\stephane\libmodbus\
```

```
D:\eclipCe18-12WS\stephane\libmodbus\/src/modbus.c:193:
          modbus_connect(ctx);
  :::::: 11 others :::::::
D:\eclipCe18-12WS\stephane\libmodbus\/tests/unit-test-server.c:132:
          rc = modbus_connect(ctx);
```

Show all threads of a program

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

Check the availability of a (serial) device

```
stty < /dev/serial0
```

```
-bash: /dev/serial0: No such file or directory      Output if device does not exist

speed 9600 baud; line = 0;                          Displays some infos if the device exists
-brkint -imaxbel
```

List all configuration variables

```
getconfig -a
```

```
LINK_MAX                        65000
PATH                            /bin:/usr/bin
CS_PATH                         /bin:/usr/bin
POSIX_ALLOC_SIZE_MIN            4096
PAGESIZE                        4096
PAGE_SIZE                       4096            :::::: and more :::::::
```

grep on Windows

Above examples on listing files containing obviously came from a Windows machine. It has grep only because of ported Linux packages installed there.

```
where grep
```

```
C:\util\msys32\usr\bin\grep.exe
```

As `grep` on Windows is a ported Linux tool, all `grep` examples work on Linux, too.

Abbreviations

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

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

A (as unit) Ampere

ABI Application binary interface; here the C-library interface to OS services

ACL Access control list

AJAX Asynchronous JavaScript and XML (or JSON en lieu de XML)

AVR microcip AVR (ex Atmel ARV) 8 bit microcontroller architecture

CGI Common Gateway Interface

CLI Command line interpreter/interface

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

CRC Cyclic redundancy check, a polynomial division

CSS Cascading Style Sheets

CSV Coma separated value; a pseudo table format recognised by Excel, Calc and consorts

DHCP Dynamic Host Configuration Protocol

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

EIA Electronic Industries Alliance

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)

HTML Hypertext Markup Language

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

IDE Integrated development environment (like Eclipse)

imho in my humble opinion

IO Input and Output

IoT Internet of Things; devices with sensors and actuators connected via networks

IP Internet protocol

JSON JavaScript Object Notation

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

M2M Machine to machine

MMU Memory management unit

MQTT Message Queuing Telemetry Transport

MS Microsoft

Nb. Nota bene (Latin, Italian phrase): note well, important

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

OASIS Organization for the Advancement of Structured Information Standards

OS Operating System; run time

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

pigpio or pigpioD Pi GPIO daemon, see [61..63]

Pi OS Raspberry Pi's standard operating systems, formerly known as Raspbian – Debian variant

PoE Power over Ethernet

PV Photovoltaic, solar power

PWM Pulse width modulation (of a rectangular signal of foxed frequency)

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

RS232 or TIA-232, EIA-232 serial communication standard (1962 RS = recommended standard)
RS485 or TIA-485, EIA-485 serial communication standard (1983) usually half duplex, multipoint

sic! so, exactly so said/cited (even if unbelievable)

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

TIA Telecommunications Industry Association

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

V (as unit) Volt

VoIP Voice over IP, protocol for telephones / telephone system on (W)LAN

W (as unit) Watt

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

W10 MS Windows 10

W11 MS Windows 11

XML Extensible Markup Language

µP Micro-processor

References

We use identical reference numbers in [29 .. 33], hence some gaps.

[29] Albrecht Weinert, Ubuntu for remote services, Report, November 2016 – February 2019, weinert-automation.de/pub/ubuntu4remoteServices.pdf

[31] Albrecht Weinert, Raspberry for distributed control, Report, October 2026 — this paper weinert-automation.de/pub/raspberry4distributedControl.pdf → a-weinert.de/pub/

[32] Albrecht Weinert, Raspberry Pi GPIO mit Java — Java statt C, Java Magazin 1|2020, pp.78-84, GIT .pdf jaxenter.de/java/java-programmiersprache-c-raspberry-pi-90176

[33] Albrecht Weinert, Raspberry for remote services, Report, May 2017 .. Oct. 2024, This paper [31]'s predecessor 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] Kontron, Raspberry Pi and PiXtend kontron-electronics.com/products/raspberry-pi

[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. R0p4 Technical Reference Manual DDI0500D_cortex_a53_r0p2_trm.pdf (BCM2837 is Quad-core 64-bit ARM cortex A53 CPU) community.nxp.com/...//DDI0500J_cortex_a53_trm-1.pdf

[59] Broadcom, ARM® Cortex®-A Series, Version 1.0, Programmer's Guide for ARMv8-A cs140e.sergio.bz/docs/ARMv8-A-Programmer-Guide.pdf

[60] Shore, Chris, ARM, Porting to 64-bit ARM, white paper July 2014, https://community.arm.com/cfs-file/ ... /Porting-to-ARM-64_2D00_bit.pdf

[61] N.N., Joan, pigpio library – Download & Install, abyz.me.uk/rpi/pigpio/download.html

[62] N.N., Joan, pigpio library – pigpio C interface, abyz.me.uk/rpi/pigpio/cif.html

[62b] N.N., Joan, pigpio library – pigpio**d** C interface, abyz.me.uk/rpi/pigpio/pdif2.html

[63] Hall, Brian "Beej Jorgensen", Beej's Guide to Network Programming – Using Internet Sockets Version 3.2.11, 2025 beej.us/guide/bgnet/pdf/bgnet_a4_c_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.pdf

[66] Modbus Org, MODBUS Application Protocol Specification V1.1b3, 2012 modbus.org/file/secure/modbusprotocolspecification.pdf

[67] OASIS, MQTT V3.1.1 Protocol Specification, 2014 docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[68] Rahmann, Leila F., Tutorial on Mosquitto and Paho, Eindhoven 2017, trevargee.hopto.org/dump1090/...Odds/MQTT_2016.pdf

[69] Mosquitto, The API documentation https://mosquitto.org/api/files/mosquitto-h.html

[70] Fairhead, Harry, Raspberry Pi IoT in C, Third Editiom, I/O Press 2024