HAM Radio Controller / Raspberry PI

prepared functions and drivers

User Application

create two functions:

void app_setup() ... do all initial configuration and return

void app_loop() ... is called in an endless loop, do all working jobs here

To stop the application set variable „keeprunning“ to 0.

GPS

initialize the GPS interface

gps_open(char *idserial, char *idVendor, int speed)

idserial and idVendor: IDs of an USB/serial adapter.
these ID can be read in a terminal with the commands (set USB0 to the needed device number):

```
udevadm info -a -p  $(udevadm info -q path -n /dev/ttyUSB0) | grep
'{serial}' | cut -d \" -f2 | head -n 1
udevadm info -a -p  $(udevadm info -q path -n /dev/ttyUSB0) | grep
'{idVendor}' | cut -d \" -f2 | head -n 1
```

to use the Raspberry onboard serial interface enter NULL for idserial and idVendor

read GPS information:

char *getTime(); returns the time in GPS format HHMMSS
char *getDate(); returns the date in GPS format
float getLatitude(); latitute in degrees
float getLongitude(); longitude in degrees
char *getQTHloc(); ham radio QTH locator

int azimuth, elevation;
void getSunPos(&azimuth, &elevation); returns the actual sun position with a resolution of about 1 degree

if no GPS is available this function uses date/time from the local clock which is synced to NTP.
The default latitude/longitude can be defined as
#define DEFAULT_LATITUDE 48.123

#define DEFAULT_LONGITUDE 12.123

Stepper Motor Controller

up to 5 stepper motors are supported (can be increased in STEPPERNUM)

create a handler for a stepper motor:
create_stepper(int sp, int dp, int ep, int pol, int spd, int refp, int refdir)
sp ... step port (see gpio.h for valid values)
dp ... direction port (see gpio.h for valid values)
en ... enable port (-1 if not used) (see gpio.h for valid values)
pol .. polarity (sink=1 or source=0)
spd ... speed (pulse frequency in 100us steps. This is limited by I2C speed and I2C load)
refp .. input port for a reference switch (see gpio.h for valid values) or -1 if not used
refdir ... polarity of signal on reference switch input port or -1 if not used

returns: -1=error, other values are the reference ID of the created stepper driver

move stepper by „steps“ into direction „dir“
void move_stepper(int ID, int steps, int dir, int wait)
ID ... stepper id returned by create_stepper
steps ... number of steps to move
dir ... direction 0 or 1
wait ... 0=return immediately 1=wait until position is reached

move stepper into direction „dir“ until condition comes TRUE
void move_cond_stepper(int ID, int steps, int dir , int wait, int (*stepcond)(int step, int dir))
ID ... stepper id returned by create_stepper
steps ... number of steps to move (or -1 to move infinite)
dir ... direction 0 or 1
wait ... 0=return immediately 1=wait until position is reached
stepcond ... user supplied callback function which must return 1 to stop movement (must be thread safe!)

step to specified position, makes only sense when a reference switch is used
void gopos_stepper(int ID, int pos, int wait)
ID ... stepper id returned by create_stepper
pos ... step-position to go to
wait ... 0=return immediately 1=wait until position is reached

move to reference switch position
void ref_stepper(int ID)
ID ... stepper id returned by create_stepper
waits until refpos is reached

read current step position **int getpos_stepper(int ID)**

disable all steppers **void disable_stepper()**

Linear Motor Controller

create a new process handling a linear motor
int create_motor(int en, int lft, int rgt, int spd, int maxtm, int (*stopcond)())

en ... enable port (see GPIO.h for available port names)
lft ... draw left port (see GPIO.h for available port names)
rgt ... draw right port (see GPIO.h for available port names)
spd ... Speed in % (1..99)
maxtm ... shut off motor after maxtm seconds
stopcond ... user supplied stop funktion, must return 1 to stop movement

returns: -1=error, other values are the reference ID of the created motor driver

turnleft_motor(int ID, int wait=1, int spd=0) ... enable left turning
turnright_motor(int ID, int wait=1, int spd=0) .. enable right turning
stop_motor(int ID) ..... stop immediately

ID ... motor ID (as returned by cretae_motor)
wait ... 1=wait for motor stop, 0=return immediately
spd ... re-set the motor speed in % (1..99)

Rotary Encoder

initialize and assign ports to the encoders (max 3)

void init_rotencoder(int portA1, int portB1, int portA2, int portB2, int portA3, int portB3)

see gpio.h for available port names. Unused ports: set to -1

read encoder steps since last read:
int getEncSteps(int idx)
idx is the encoder number 0,1 or 2

Serial Interfaces

serial port located on the Raspberry PI board

void open_serial(int speed)

the speed value MUST be a linux-speed value, so do not enter 9600, instead enter B9600 !

int write_serial(int data) send one byte
int write_serial_free() check if the TX buffer is free
int read_serial() read one byte

serial ICOM CIV port which is a USB/serial adapter mounted on this controller board

void open_civ(int speed)

the speed value MUST be a linux-speed value, so do not enter 4800, instead enter B4800 !

int write_civ(int data) send one byte
int write_civ_free() check if the TX buffer is free
int read_civ() read one byte

other USB/serial adapters connected to the RPI's USB ports

int open_serialUSB(int speed, char *idserial, char *idVendor)

for a description of the values idserial and idVendor, see the documentation of the GPS functions

the speed value MUST be a linux-speed value, so do not enter 4800, instead enter B4800 !

int write_serialUSB(int data) send one byte
int write_serialUSB_free() check if the TX buffer is free
int read_serialUSB() read one byte

---

functions to send and receive UDP messages

start UDP reception
void UdpRxInit(int *sock, int port, void (*rxfunc)(uint8_t *, int, struct sockaddr_in*), int *keeprunning)

- sock ... pointer to a socket (just a pointer to an int)
- port ... own port, messages only to this port are received
- rxfunc ... pointer to a callback function, will be called for received data
- keeprunning ... pointer to an int. If it is set to 0, the function exits

A callback function must be supplied:
void rxfunc(uint8_t *data, int len, struct sockaddr_in* sockaddr)

This function is called when an UDP message was received. sockaddr can be used to get information about the sender. This function MUST be thread safe !

send an UDP message:
void sendUDP(char *destIP, int destPort, uint8_t *pdata, int len)

RPI 3,5" TFT Touch Display (Waveshare)

there are a couple of settings to do for display activation. Please refer to the explanation in display.cpp.

If a display is connected it will be initialized and cleared at program start automatically. You just need

to call the following drawing functions:

void display_clear() ... clear the display, black.

void TFT_Hor_Line(int xs, int xe, int ypos, COLOR rgb, COLOR brgb, int linewidth, uint32_t pattern)
Draw a horizontal line.

xs,xe ... start end endpoint
ypos ... vertical position
rgb ... line color
brgb ... background color, used for i.e. dotted line to fill the gaps
linewidth ... width of the line
pattern ... 32 bit value which is used to draw a line pattern. I.e.: 0xf0f0f0f0 will draw a long dotted line

void TFT_Vert_Line(int xpos, int ys, int ye, COLOR rgb, COLOR brgb, int linewidth, uint32_t pattern)

same as above, but vertically

void TFT_DrawLine(int x0,int y0,int x1,int y1, COLOR rgb, int linewidth)

Draw any straight line
x0, y0 ... start of the line

void TFT_Rectangle(int xs, int ys, int xe, int ye, COLOR rgb, int linewidth)

draw a rectangle
xs,ys ... first edge
xe,ye ... second edge
x1,y1 ... end of the line
rgb ... line color
linewidth ... width of the line

void TFT_Rectangle_filled(int xs, int ys, int xe, int ye, COLOR rgb)

as above, but fill the rectangle with rgb

void TFT_drawcircle(int x0, int y0, int radius,COLOR rgb, int linewidth)

draw a circle
x0,y0 ... center
radius ... radius of the circle
rgb ... line color
linewidth ... width of the line

void TFT_drawcircle_filled(int x0, int y0, int radius,COLOR rgb)

as above, but filled with rgb

void TFT_drawJPGfile(char *filename, int x, int y)

draw a jpg file
filename ... name of the jpg file (ohne jpg files, NON progressive !)

x,y ... top left edge of the graphics

void draw_font(char *word, int init_x, int init_y, int fontidx, COLOR rgb, int font_size)

draw text

word ... text to be printed
init_x,init_y ... position
fontidx ... one of the available fonts, see below
rgb ... line color
font_size ... size of the text (20 is a good starting value)

available fonts:

0 ... Verdana
1 ... Verdana_Bold
2 ... Verdana_Italic
3 ... Verdana_Bold_Italic
4 ... LiberationMono-Regular
5 ... LiberationMono-Bold
6 ... LiberationMono-Italic
7 ... LiberationMono-BoldItalic
8 ... Courier_New
9 ... Courier_New_Bold
10 .. Courier_New_Italic
11 .. Courier_New_Bold_Italic

Bar Graphs on 3,5" TFT Display

mainly used to show measurement values

create all bargraphs only once at program start with this command:

int create_bargraph(t_bargraph *settings)

for each bargraph fill a t_bargraph structure and the call this function. This will create and initialize the bargraph but will not print anything on the screen.
the structure t_barlabel (the labels below the x-axis) may also be filled. For a description of the values see bargraph.h

void putvalue_bargraph(int id, double value, int redraw);

draw the bargraph and/or show values
id ... ID of the bargraph returned by create_bargraph
value ... the value to be shown
redraw ... 0=draw everything, 1=draw value only
after showing a new screen call with redraw=0, this will draw the complete bargraph. To show values use redraw=1, this will only refresh the bar and value which avoids flickering (almost).

Touch Buttons on 3,5" TFT Display

All touch buttons are created only once after program start, by filling a structure TOUCHBUTTON and then calling:

int create_touchbutton(TOUCHBUTTON *ptb)

for a detailed description of the values in this structure see touchbutton.h

void draw_button(int id)

draw a button previously created by create_touchbutton.
id ... ID of the button returned by create_touchbutton

When drawing a new screen call:

void deactivate_allbuttons()

this will deactivate all buttons. Then draw only the buttons needed in this screen.

Web Socket

a web socket is used by a browser to establish a tcp connection between the browser and the websocket-server. This library contains a websocket server, The idea is the make a nice GUI in HTML/Javascript and the push values from this applications into the web browser, or the receive use commands from the browser.

This websocket-server can handle 20 browsers simultaneously. Every one will see the same picture and data.

Starting the web socket server:

void ws_init(int *keepr, void(*onmessage_func_handler)(unsigned char *msg), int port)

keepr ... use the existing global variable „keeprunning“ as parameter onmessage_func_handler ... callback for text messages from browser port ... 0=default(40129), or port number

void ws_send(unsigned char *pdata, int len)

send data to a browser
pdata ... pointer to the data bytes
len ... number of data bytes to be sent

Receiving data sent by the browser:

the callback funktion
void onmessage(int fd, unsigned char *msg)
will be called by the web server. See the comments in this function (ws_callbacks.cpp) for details

information.

Example Java script to establish a connection to this websocket-server see the sceleton:

websocket_sample.html

use this html file as the basis for your own implementation

Thread Safe FIFO

one of the most important modules. This FIFO enables safe communication between parallel running processes.

int create_fifo(int maxelem_num, int maxelem_len)

create a new fifo which can hold *maxelem_num* elements and
each element may be maximum *maxelem_len* bytes long.

void write_fifo(int id, uint8_t *pdata, int len)

write into the fifo
ignore data if the fifo is full

id ... ID of the fifo returend by create_fifo
pdata ... data to be stored in the fifo
len ... length of pdata in bytes

int read_fifo(int id, uint8_t* pdata, int maxlen)

read data from the fifo

id ... ID of the fifo returned by create_fifo
pdata ... array where the retrieved data are stored
maxlen ... maximum length of pdata, which should be at least *maxelem_len* or data may be lost
returns: length of received data

void fifo_clear(int id)

removes all elements from a fifo
id ... ID of the fifo returned by create_fifo

int fifo_freespace(int id)

left free space for elements
id ... ID of the fifo returned by create_fifo

int fifo_dataavail(int id)

number of elements currently in the fifo
id ... ID of the fifo returned by create_fifo
returns: 0=no data in the fifo, 1=there are some data in the fifo

int fifo_usedspace(int id)

returns the number of element currently in the fifo
id ... ID of the fifo returned by create_fifo

Timer and Time functions

setup a timer which calls a callback function every x milliseconds

int start_timer(int mSec, void(*timer_func_handler)(void))

creates and starts a new timer
mSec ... timeout in milliseconds
func_handler ... this function is called every mSec milliseconds. This function must be supplied by the user. It MUST be thread safe. You can use a fifo for save data transfer if required. For simple setting variables consider to use a critical section to protect variables used by multiple threads.

Examples how to use critical sections can be found in kmfifo.cpp, see LOCK and UNLOCK

void sleep_ms(int ms)

waits for ms milliseconds

char *get_utctime()

return the UTC time hhmmss

char *get_utcdate()

returns the UTC date: ddmmyy

void measure_samplerate(int id, int samp, int prt)

Call from within a loop the measure the rate per second.

id ... 0..9, this function can be used for 10 different mesaurements
samp ... number of samples
prt ... wait for prt measurments before printing the result

System Functions

functions usually used in most programs.

int isRunning(char *prgname)

checks if a program is already running, can be used to check if a program is started twice.

void install_signal_handler(void (*signalfunction)())

installs a handled to capture Ctrl-C keypress. If Ctrl-C is pressed the function „signalfunction“ will be called which can be used to free/cleanup and close a program

void showbitstring(char* title, uint8_t* data, int totallen, int anz)
void showbytestring(char *title, uint8_t *data, int totallen, int anz)
void showbytestring16(char *title, uint16_t *data, int anz)
void showbytestring32(char* title, uint32_t* data, int anz)
void showbytestringf(char* title, float* data, int totallen, int anz)

set of functions to print a series of numbers in the terminal. Very helpful for debugging purposes.

title ... some title do print before the numbers
data ... data array conaining the numbers
totallen ... total length of the array (just informative)
anz ... this number of values will be printed

char* ownIP()

gets the local IP address

char *getConfigElement_string(char *elemname)
void getConfigElement_double(char *elemname, double *pv, double multiplier)
void getConfigElement_longlong(char *elemname, long long *pv, double multiplier)
void getConfigElement_int(char *elemname, int *pv, double multiplier)

set of functions to read values from a configuration file.
The name of the config file must previously be written into variable „configfile“ (which is char[512])

Config File Format:
... comment
ElementName-space-ElementValue
the returned value is a static string and must be copied somewhere else before this function can be called again

Example config file:

This is an example
IPADDRESS 192.168.1.2

usage:
strcpy(IPaddress,getConfigElement_string(„IPADDRESS“));

char *runProgram(char *parameter, int maxlen)

start a system program and return it's stdout text.

parameter ... full command line text
maxlen ... length of parameter string

the returned text is written to „parameter“, so it must be long enough to hold the returned string.