ESP32forth and Arduino C++ - notes from beginners corner.

ESP32forth is written in Arduino C and some knowledge of C is very helpful. As amateur programmer, with basic knowledge of Forth only, I resolved to learn also basics of Arduino C to be able to better understand and use ESP32forth. I created some notes about my first examples to help others in the same situation.

I came to ESP32forth from FlashForth on Arduino UNO. This Forth is written in ATmega328 assembler and Forth itself. ATmega chip is not so complicated, with good documentation, for full usage of I/O pins, timers, interrupts, I2C aso. is possible to directly program registers and have chip under control from Forth. ESP32 chip is more complicated, 2 cores, firmware, WiFi, Bluetooth, A/D, D/A etc. ESP32forth uses ready made C libraries, it is possible to attach ready C code. And C code exists for practically everything. I suppose there is also possible to control chip GPIOs, timers, interrupts directly from Forth code, but life is short and ESP32 tech docs are huge. So I think there is space for C code for control chip specific features like interrupts, timers, WiFi etc. Also experts as Don Golding suggest this approach.
For learning I at first test some code in C and than the same I create in ESP32forth. For Arduino C code examples I use web *randomnerdtutorials.com* , part ESP32, where are nice explanations of C code and also schematics of used circuits.. Raw code of my programs is attached in separate files.
First example is interrupt routine, where you switch on/off LED with connected button with activated pullUP resistor. Button activates interrupt routine. Arduino C code:

```
// program for simple interrupt possibility test
// case sensitive code
#define buttonPin 33
#define LEDPin 32

void IRAM_ATTR menLED(){
    digitalWrite(LEDPin, !digitalRead(LEDPin));
    Serial.println("Here one interrupt!");
}

void setup(){
    pinMode(LEDPin, OUTPUT);
    pinMode(buttonPin, INPUT_PULLUP);
    attachInterrupt(buttonPin, menLED, FALLING);
    Serial.begin(115200);
    digitalWrite(LEDPin, HIGH);
}

void loop(){
  Serial.println("ESP32 goes");
  Serial.print("Input 32: ");
  Serial.println(digitalRead(LEDPin));
  delay(1000);

}
```

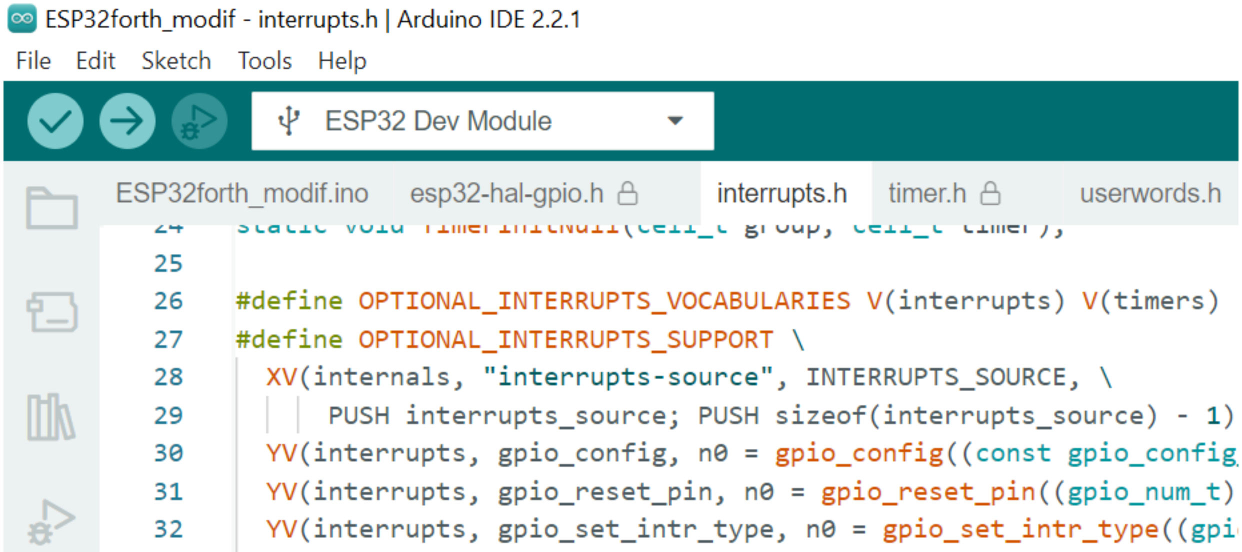
Interrupt routine menLED has type IRAM_ATTR which means store code in RAM for quicker respond. Normaly Arduino code is in Flash opposed to ESP32forth user words (program) stored in RAM.
Next is my code in ESP32forth doing the same:

```
\ program for simple interrupt possibility test
\ case unsensitive code
\ from esp32-hal-gpio.h :
\ GPIO FUNCTIONS
\ #define INPUT             0x01
\ Changed OUTPUT from 0x02 to behave the same as Arduino pinMode(pin,OUTPUT)
\ where you can read the state of pin even when it is set as OUTPUT
\ #define OUTPUT            0x03
\ #define PULLUP            0x04
\ #define INPUT_PULLUP      0x05
\ #define PULLDOWN          0x08
\ #define INPUT_PULLDOWN    0x09

\ Interrupt Modes
\ #define DISABLED  0x00
\ #define RISING    0x01
\ #define FALLING   0x02
\ #define CHANGE    0x03
\ #define ONLOW     0x04
\ #define ONHIGH    0x05
\ #define ONLOW_WE  0x0C
\ #define ONHIGH_WE 0x0D

defined? MARKER 0<> [if] forget MARKER [then]
create MARKER

only Forth definitions also interrupts
\ attach words in interrupts vocabulary

decimal
33 constant ButtonPin
32 constant LEDPin
3 constant OUT_IN
5 constant INPUT_PULLUP

: pinIntHandle ( xt pin intMode -- ) \ attach xt isr word + mode to pin
    over >r gpio_set_intr_type throw    \ pin type--0/err
    r> swap 0 gpio_isr_handler_add throw    \ pin xt 0--0/err
    ;

```

```
41 : manLED    ( -- ) \ word for interrupt action reversing LED
42     LEDpin digitalRead  0= \ reverse LED light
43     if  1 else 0 then       \ possible read and write from GPIO32 LEDpin
44         LEDpin swap digitalWrite
45     ;
46
47 : setup ( -- ) \
48     LEDPin OUT_IN pinMode   \ with out_in is possible to write and read
49     ButtonPin INPUT_PULLUP  pinMode  \ input with pull_up resistor
50     LEDPin HIGH digitalWrite
51     ;
52 setup   \ make setup
53 ' manLED ButtonPin #GPIO_INTR_NEGEDGE pinIntHandle \ attach interrupt on hi->lo
54             \ with action word manLED
55
56 : mainLoop ( --) \ main program loop for test only
57     begin
58     ." ESP32forth goes" cr
59     ." Input 32: " LEDPin digitalRead . cr
60     1000 ms
61     key? until
62     ;
```

Program in C code gave me interesting new info for Forth code. For word *pinMode* there are more possibilities than only INPUT and OUTPUT as is explained in start of forth code after GPIO FUNCTIONS.
So I use definitions *3 constant OUT_IN* and *5 constant INPUT_PULLUP.* OUT_IN adjusts GPIO to be used as output and input in the same time. INPUT_PULLUP adjusts GPIO as input with pullup resistor in one command.
Word *pinIntHandle* does the same as *attachInterrupt* in C code, it is attaching interrupt routine and interrupt mode to GPIO pin. In word manLED is used new possibilty to read and also write from the same pin. With ESP32forth simple OUTPUT it is not possible. Line 53 starts interrupt sensitivity and *mainLoop* is there only to do something between interrupt actions in background. Start *mainLoop* manualy.

For study of ESP32forth is perfect to use Arduino IDE version 2.xxx with good editor with help possibilities.



OK, that is first example.
For next we need to use possibility to add code from C Arduino libraries as new words into ESP32forth. I use possibility to add code into special file with name *userwords.h* compiled together with *ESP32forth .ino* file. This is explained on Brad Nelsons page *esp32forth.appspot.com/ESP32forth.html* at the bottom. This is for me complicated by use of X-macros, maybe others can give some deep explanation for beginners as me. But ok, I did it.
So in next two examples I focused to watchdog timer function and analog read function. I use task watchdog timer, which could have usage in constructions running round o clock without users control, where automatic reset can restart and rerun construction after some not expected conditions. Here is my very simple *userwords.h* file:

```

#include <esp_task_wdt.h>
#define USER_WORDS \
    Y(analogReadMilliVolts, n0 = analogReadMilliVolts(n0)) \
    Y(analogReadResolution, analogReadResolution(n0); DROP) \
    Y(analogSetAttenuation, analogSetAttenuation((adc_attenuation_t) n0); DROP) \
    X("WDinit", watchdoginit, n0=esp_task_wdt_init(n1, b0); NIP ) \
    X("WDdeinit", watchdogdeinit, PUSH esp_task_wdt_deinit()) \
    X("WDtaskAdd", wdtaskadd, PUSH esp_task_wdt_add(NULL)) \
    X("WDreset", watchdogreset, PUSH esp_task_wdt_reset()) \
    X("WDtaskRemove", wdtaskremove, PUSH esp_task_wdt_delete(NULL))

```

Code for watchdog function is located in *esp_task_wdt.h* . I have added 5 new words starting with WD…. Here is explanation of usage:
/* *Task Watchdog Timer (TWDT)*
* *WDinit (timeout panic-- err) timeout in seconds, panic 0 is log output,*
* *-1 is firing reset, err 0 is OK, other err codes according esp_err.h, this*
* *initializes TWDT to run*
* *WDdeinit (--err) this ends TWDT, can be used after tasks unsubscribed*
* *WDtaskAdd (--err) subscribes current running task to TWDT*
* *WDreset (--err) resets TWDT to not fire*

** WDtaskRemove (--err) unsubscribe current task from TWDT*
**/*

Basic word is *WDinit*, which adjusts watch dog timeout and action. For testing is good log only output, for real application reset would be the right one. Correct sequence of actions for watchdog usage is WDinit – WDtaskAdd – periodicaly WDreset to restart timer before firing panic – WDtaskRemove . For testing I started with this C code from web *iotassistant.io/esp32/enable-hardware-watchdog-timer-esp32-arduino-ide :*

```
#include <esp_task_wdt.h>

//3 seconds WDT
#define WDT_TIMEOUT 3

void setup() {
  Serial.begin(115200);
  Serial.println("Configuring WDT...");
  esp_task_wdt_init(WDT_TIMEOUT, true); //enable panic so ESP32 restarts
  esp_task_wdt_add(NULL); //add current thread to WDT watch

}

int i = 0;
int last = millis();

void loop() {
  // resetting WDT every 2s, 5 times only
  if (millis() - last >= 2000 && i < 5) {
      Serial.println("Resetting WDT...");
      esp_task_wdt_reset();
      last = millis();
      i++;
      if (i == 5) {
        Serial.println("Stopping WDT reset. CPU should reboot in 3s");
      }
  }
}
```

Code 5 times resets watchdog timer after 2 seconds and next it leaves it to fire watchdog timer after adjusted 3 seconds.
The same situation in ESP32forth has 2 variants of code. First is code in word *WDlogTest* . There is adjusted 10 seconds watchdog time-out and than in lines 27 to 36 there is 6 times possible to restart WD timer with keyboard button in 10 seconds limit or generate log panic output. After 6 times there is deactivated watchdog with *WDtaskRemove* word.
Second example is word *WDresetTest* . There is the same time-out 10 seconds. For reseting WD timer there are on lines 54, 55 2 forth tasks reseting WD timer in 2 seconds and 3 seconds periods. Action words for tasks are defered, so it is possible to install different action words during task run. For this there is NOOP word doing nothing except mainly not reseting WD timer. This is created on lines 43-

53. At first tasks run action words *action1*, *action2* doing timer reset in time. Next is possible from console exchange action words and see reset of ESP32 chip.
Small note – there is difference between task used in TWDT, which is ESP32 firmware task and ESP32forth tasks. Maybe experienced users can explain it in more detail.

```
defined? MARKER 0<> [if] forget MARKER [then]
create MARKER

only Forth
decimal

10 value WDT_TIMEOUT \ 10 seconds time-out
0 constant logTWDT   \ watchdog activates log or hard reset
-1 constant resetTWDT

\ 1st test with user manualy reseting TWDT by pressing key
: WDlogTest ( -- ) \ test watchdog generating log only
    WDT_timeout logTWDT WDinit drop \ adjusts TWDT time and action
    WDtaskAdd drop  \ loose watchdog from chain
    5 for           \ loop giving 6 times chance to press key
        begin
        ." Watchdog running, press button to feed it!" cr
        ms-ticks . ."  ms" cr
        500 ms
        key? until key drop
    WDreset drop \ food for wd activated by key press
    cr ." great, food accepted by WD " cr cr
    ." loop no: " i . cr
    next
    WDtaskRemove drop \ unsubscribe
    \ WDdeinit drop   \ ends TWDT, but it generates error 0x103 ??
    ." TWDT  deactivated by unsubscribing"  cr
    ;

\ 2nd test with 2 tasks reseting periodicaly timer
tasks
defer hi1 \ defered action word for task
defer hi2
: noop ; \ does nothing, prepared to "deactivate" tasks
\ next 2 tasks with WDT reseting command
: action1  ." Time 1sttask is: " ms-ticks . cr 2000 ms
    WDreset drop  ;    \ food for wd - reset counter
: action2  ." Time 2ndtask is: " ms-ticks . cr 3000 ms
    WDreset drop  ;
' action1 is hi1        \ fill defered action words
' action2 is hi2
' hi1 100 100 task 1sttask \ create 2 tasks
' hi2 100 100 task 2ndtask
```

```
: WDresetTest ( -- ) \ test watchdog generating ESP32 reset
    WDT_timeout resetTWDT WDinit drop \ adjusts TWDT time and action
    WDtaskAdd drop  \ loose watchdog from chain
    1sttask start-task 2ndtask start-task \ and activate tasks
    ;
\ ' noop is hi1 \ this command removes reseting in task1
\ ' noop is hi2 \ this command removes reseting in task2
\ with resulting reseting ESP32 after 10 sec timeout
```

Third example is about A/D voltage measurement. There is potentiometer on 3.3 V generating voltage 0-3.2V on GPIO32 pin. In this example I add more words for A/D conversion to ESP32forth. This are:

```
/*
* analogReadMilliVolts ( pin--mV)
* analogReadResolution ( n-- ) adjust A/D convertor resolution 9-12bits.
*   Default is 12.
* analogSetAttenuation ( n-- ) adjust attenuation for all A/D measurements.
*   Range 0-3. Default 11dB.
* attenuation is defined:
* typedef enum {
*    ADC_0db,
*    ADC_2_5db,
*    ADC_6db,
*    ADC_11db,
*    ADC_ATTENDB_MAX
* } adc_attenuation_t;
*/
```

These new functions are part of standard Arduino C functions, but not included into *ESP32forth.ino* file. Now are added with *userwords.h* file .
Next is C code little shortened (full code attached) with A/D measurement. Again C explanation and schematics on *randomnerdtutorials.com*.

```
int analogValue;
int analogVolts;

void setup() {
  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);

  //set the resolution to 12 bits (0-4095)
  analogReadResolution(12);
  adc_attenuation_t attenuation = ADC_11db;
    Serial.println("The attenuation value is ADC_11db - default value ");
    analogSetAttenuation( attenuation );
    ReadAnalog();
    Serial.printf("ADC analog value = %d\n",analogValue);
    Serial.printf("ADC millivolts value = %d\n",analogVolts);
}
```

```
 void ReadAnalog() {  // read values from GPIO32
    analogValue = analogRead(32);
    analogVolts = analogReadMilliVolts(32);
    delay(100);
  }

void loop() {
  // read the analog / millivolts value for pin 32:
  ReadAnalog();
  // print out the values you read:
  Serial.printf("ADC analog value = %d\n",analogValue);
  Serial.printf("ADC millivolts value = %d\n",analogVolts);
  Serial.println();
  delay(1500);  // delay in between reads for clear read from serial
}
```

Basic reading is done in function ReadAnalog reading value 0-4095 and the same in milliVolts. This loops in main loop with 1.5 second delay.
Next forth code, also shortened:

```
only Forth decimal
32 constant ADpin
12 constant ADresolution
0 constant ADC_0db
1 constant ADC_2_5db
2 constant ADC_6db
3 constant ADC_11db

: ReadAnalog ( pin-- value mVvalue ) \ read analog values
    dup  analogRead swap analogReadMilliVolts
    ;
: .AnalogValues ( pin-- ) \ print AD readings from pin
    ReadAnalog
    ." ADC millivolt value = " . cr
    ." ADC analog value = " . cr
    ;
: ADtest    ( -- )  \ show values for different attenuations
    ADresolution analogReadResolution \ adjust 12 bit AD conversion
        ADC_11db analogSetAttenuation        \ adjust AD attenuation to default 11db
    ." The attenuation value is ADC_11db " cr
    ADpin .Analogvalues                   \ print results for adjusted attenuation
    ;
: mainLoop ( --) \ main program loop for test only
    ADtest
    cr ." Continous reading every 1.5 sec " cr
    begin
    ADpin .AnalogValues cr
    1500 ms
    key? until ;
```

Again basic reading is done with word *ReadAnalog* leaving 2 values on stack. *MainLoop* loops and measures A/D values from pin 32 each 1.5 second.

Some final notes:
For mentioned code I used ESP32forth 7.0.7.15, compiled with interrupts.h and userwords.h. Arduino IDE 2.2.1 with board ESP32 Dev Module.

Reset of ESP32 with TWDT generates this text:

```
-->
 ok
--> E (362770) task_wdt: Task watchdog got triggered. The
following tasks did not reset the watchdog in time:
E (362770) task_wdt:  - loopTask (CPU 1)
E (362770) task_wdt: Tasks currently running:
E (362770) task_wdt: CPU 0: IDLE
E (362770) task_wdt: CPU 1: loopTask
E (362770) task_wdt: Aborting.
```

For learning of usage X-macros for userwords.h I created simple testing userwords.h:

int soucet(int a, int b) {
int vysledek=a+b;
return vysledek;
}
*#define USER_WORDS *
*X("soucet1", summa1, n0 = soucet(n1, n0)) *
*X("soucet2", summa2, n0= soucet(n1, n0); DROP) *
*X("soucet3", summa3, n0= soucet(n1, n0); NIP) *
*X("soucet4", summa4, n0= soucet(n1, n0); NIPn(2)) *
X("soucet5", summa5, SET soucet(n1, n0))

and testing from REPL shows:

```
--> 2 3 soucet1        \ X("soucet1", summa1, n0 = soucet(n1, n0)) \
2 5 --> 2drop
 ok
--> 2 3 soucet2        \ X("soucet2", summa2, n0= soucet(n1, n0); DROP) \
2 --> drop
 ok
--> 2 3 soucet3        \X("soucet3", summa3, n0= soucet(n1, n0); NIP) \
5 --> drop
 ok
--> 2 3 soucet4        \X("soucet4", summa4, n0= soucet(n1, n0); NIPn(2)) \

--> 2 3 soucet5        \X("soucet5", summa5, SET soucet(n1, n0))
2 5 -->
```

This first steps with Arduino C and ESP32forth showed me plainly real advantage of Forth in testing/debuging parts of program using peripherals as GPIO, timers, interrupts etc. On other side amount of ready C code and support for Arduino C programming is irreplaceable. Best use both.