1 iopin-library

De demo gebruikt pinnamen om SW1 uit te lezen en LED0 en LED1 aan te sturen.

Demo06 gebruikt C, demo07 gebruikt C++-wrappers voor de iopin-library.

De 3 voornaamste bestanden zijn:

1. `main.c(pp)`: bevat definities en initialisatie, plus de interruptroutine die het eigenlijke werk doet.
2. `iopin.c`: C-routines voor 'fast-io', vrij directe aansturing van de chip-hardware.
3. `iopin.h`: header met hardware-registerdefinities en IoPin-klasse voor C++.

1.1 main

Klokinstellingen van de microcontroller zijn als in voorgaande demo's, hier verder niet terzake doend omdat de functionaliteit vanaf 240 kHz uit demo00 tot en met 120MHz uit demo03, waarschijnlijk ook met 144 MHz uit de variant van demo03_144 werkt.

Verschillen tussen C- en C++-versie beginnen met de pindefinitie:

demo06 / main.c:

```
    //TB 5.5 User LED, p.12
#define LED0 PD6
#define LED1 PD7

    //TB 5.11 User Switch, p.15
#define SW1 PB1
```

demo07 / main.cpp:

```
//TB 5.5 User LED, p.12
IoPin led0(PD6);
IoPin led1(PD7);

    //TB 5.11 User Switch, p.15
IoPin sw1(PB1);
```

De C-versie gebruikt constanten, gedefinieerd middels de preprocessor-instructie #define en gebruik van hoofdletters. Snelle schatting levert 44 voor de waarde PB1, omdat het de 44^e IO-pin na PJ3 is.

De C++-versie gebruikt dezelfde constanten om een object-variabele aan te maken van de klasse IoPin, die de waarde van de constante opslaat. De variabelen zijn met kleine letters aangeduid.

Bij de C-versie is na deze regels nog geen code uitgevoerd, maar bij de C++-versie, zonder optimalisatie, zal code doorlopen zijn om deze, statische, objecten aan te maken en RAM-geheugen gebruikt zijn voor de opslag.

Een volgend verschil ontstaat bij void hardware_setup (void), dat in demo06 een C-routine en in demo07 een C++-routine is. In deze routine wordt voor het eerst de iopin-bibliotheek gebruikt:

demo06 / main.c:

```
    iopin_init(LED0,IO_LO);
    iopin_init(LED1,IO_HI);
    iopin_init(SW1, IO_IN);
    iopin_mpcfunc(SW1,1<<6);      //(IRQ4-DS)
```

demo07 / main.cpp:

```
    led0.init(IO_LO);
    led1.init(IO_HI);
    sw1.init(IO_IN);
    sw1.mpcfunc(1<<6);   //(IRQ4-DS)
```

In de C-versie worden functie-aanroepen gedaan, waarbij de eerste parameter de betreffende iopin aanduidt. Ter voorkoming van dubbele functienamen zijn deze voorafgegaan door 'iopin_', de module waarin de functies werkzaam zijn.

In de C++-versie zijn de aanroepen 'method calls', de functionaliteit zit in de IoPin-klasse en qua naamruimte daardoor afgeschermd. In C++ is de waarde van de iopin verstopt middels de verborgen eerste parameter die het objectadres meegeeft. De objectvariabele wordt gebruikt om van daaruit de functionaliteit te bereiken, theoretisch zelfs via een functietabel, en de aangeroepen functie benadert via het object-adres de gewenste waarde van het pinnummer.

In de praktijk zal bij slechts geringe optimalisatie bij deze kleine progrmma's de compiler rechtstreeks de betreffende waarden invullen, waardoor de 'overhead' beperkt blijft.

Het laatste verschil ontstaat bij de interrupt-routines, die in beide gevallen C-routines zijn:

demo06 / main.c:

```
#include "interrupt_handlers.h"

void INT_Excep_ICU_IRQ4(void)
{   if(!iopin_read(SW1) )    //LED0 (press)
        iopin_write(LED0,!iopin_read(LED0) );
    else iopin_toggle(LED1);//LED1 (release)
}
```

demo07 / main.cpp:

```
extern "C" {
    #include "interrupt_handlers.h"
}

void INT_Excep_ICU_IRQ4(void)
{   if(!sw1.read() )     //LED0 (press)
        led0.write(!led0.read() );
    else led1.toggle();//LED1 (release)
}
```

In de C-versie worden wederom functies aangeroepen via module_functie(pin,...), met alles expliciet.

De C++-versie moet hier de interrupt-routine benaderbaar maken vanuit C en gebruikt daarvoor een header-omhulling met `extern "C" {...}`.
Wederom geldt dat het pinnummer verborgen blijft in de C++-aanroepen, omdat het via het eerder beschreven mechanisme van 'method call' gedaan wordt.

Samenvattend levert het gebruik van C++ in main iets minder typewerk op, maar wat meer code-overhead en processorbelasting tijdens de uitvoering van het programma.

1.2 iopin.c

Aangezien iopin.c puur C is, zijn hier geen verschillen te melden tussen demo06 en demo07. Wel zijn er enkele interessante concepten in gebruikt, voor een gedeelte ook als object-geörienteerd te beschouwen.

```
    /* ------------------------------------------------------------
    private
    ------------------------------------------------------------ */
#define PROTECT if(_isiopin(iopin)) ; else return

#define portbit_(iopin)     (!iopin ? 3 : IOPORT[iopin]& 7)
#define portnr_(iopin)      (!iopin ? 18: IOPORT[iopin]>>4)

#define IOPIN2(a,b,...)         0x ## b,

    /* ------------------------------------------------------------
    protected
    ------------------------------------------------------------ */
uint8_t const IOPORT[eIOPINS] = {0, CHIP(IOPIN2) };         //PJ3 special
```

Onder het kopje 'private' staan zaken die alleen binnen het bronbestand bekend mogen zijn: `#define` wordt vaak in headers gebruikt, maar hier specifiek voor zaken die niet met een header mee naar buiten gebracht dienen te worden.

Het valt verder op dat portbit- en portnr-macro's een waarde opzoeken in IOPORT[ ] en deze waarde bewerken. De IOPIN2-macro is voor een onbekend aantal parameters aangemaakt, maar neemt alleen de tweede en maakt met de tekst een hexadecimaal getal voor opname in een groter geheel, omdat er een komma achter de waarde komt.

Onder het kopje 'protected' komt de aap uit de mouw, want daar wordt het array IOPORT[ ] aangemaakt met de macro CHIP(IOPIN2), ofwel een macro die een macro meekrijgt als parameter.
De betreffende constructie wordt een X-macro genoemd, van extensie, omdat de functionaliteit van de CHIP()-macro wordt uitgebreid door een andere macro mee te geven.

De initialisatie van IOPORT[ ] is met een aantal hexadecimale getallen, gescheiden door komma's, middels de IOPIN2()-macro in de CHIP()-macro.
In het commentaar staat PJ3 special, deze zal op de waarde 0 van toepassing zijn aangezien die niet met de X-macro CHIP(IOPIN2) meekomt, maar expliciet in de initialisatierij gezet is.

De macro PROTECT komt aan de orde in de C-code, ter illustratie in twee kleine routines:

```
int iopin_read (eIOPIN iopin)
{   PROTECT -1;
    return IO__PIDR[portnr_(iopin)] >> portbit_(iopin) & 1;
}
void iopin_toggle (eIOPIN iopin)
{   PROTECT;
    if(NMIPIN!=iopin)
        IO__PODR[portnr_(iopin)]^= (1<<portbit_(iopin) );
}
...
#undef PROTECT
```

PROTECT zorgt ervoor dat een functie niets doet als de parameter iopin geen IO-pin is op een speciale manier:

- met if(_isiopin (iopin)) ; wordt indien de parameter iopin een IO-pin is niets gedaan, de puntkomma rondt immers de if-opdracht af,
- anders wordt met else return een opdracht begonnen, maar nog niet afgerond door de macro.

Middels bovenstaande speciale constructie is de macro te gebruiken zowel in routines die geen resultaat teruggeven, door met een puntkomma de return-opdracht af te sluiten, als in routines die wel een resultaat geven, door de resultaatwaarde met opvolgende puntkomma toe te voegen.

Aan het bestandseinde worden de 'private' #define's ongedaan gemaakt.

1.3 iopin.h

Gezamenlijk deel, zowel voor C als voor C++:

```
#define RX65x100(X)\
/*  X( 4 ,J3,IO_LO  ,-          ,0,-                    )*/\
    X(11 ,37,IO_LO  ,-          ,0,XTAL                 )\
...
    X(100,05,IO_LO  ,-          ,0,-                    )\
    //ic  io init   assign       e func-alt
#define CHIP( X )   RX65x100( X )//RX65x64( X )

#define ENUM2(a,b,...)      P ## b,
typedef enum en_iopin       {PJ3, CHIP(ENUM2) eIOPINS   } eIOPIN;
#define _isiopin(iopin)     (0<=iopin&&eIOPINS>iopin)   //_isiopin(PJ3) => 1
```

In bovenstaand deel van het header-bestand wordt een X-macro RX65x100(X) gedefinieerd, bestaande uit diverse regels en kolommen. Het aansluitende commentaar geeft een idee over de functie van de kolommen.

We zien aan de kolom ic dat het hier om de 100-pinversie van de chip gaat, in het commentaar is ook Rx65x64(X) voor de 64-pinversie te vinden. Overigens heb ik niet de moeite genomen om de overige chips te op te nemen, op het target board zit altijd de 100-pin chip.

De ENUM2()-macro lijkt op de IOPIN2()-macro uit iopin.c, aangezien deze ook uit een onbekend aantal elementen/kolommen de tweede neemt, en deze met een komma erachter als lijstelement teruggeeft, hier met een P ervoor geplakt.

We zien dat de eerste regel, met J3, wordt overgeslagen, en vervolgens in het gebruik wordt "handmatig"PJ3 ingevoegd als eerste in de enum-lijst, afgesloten met eIOPINS.

Wetende dat een enum-lijst standaard begint met 0, en dan oploopt, komt het doel naar voren: de namen PJ3, P37, enzovoorts zijn volgnummers voor de IO-pinnen van de 100-pinversie van de chip, en eIOPINS is het aantal IO-pinnen, gebruikt om een parameter iopin te testen op geldigheid.

Het commentaar verduidelijkt dat 0, de waarde van PJ3 in de macro _isiopin(0) de waarde 1 levert ter bevestiging dat het een IO-pin is.

De oplettende lezer die goed onderlegd is in C/C++ zal uit het bovenstaande op kunnen maken dat _isiopin(iopin) in een functie als iopin_toggle() niet de meest efficiente bescherming oplevert qua processor-instructies, maar hier is de implementatie gekozen om deze zo eenvoudig mogelijk ook in C++ te kunnen verwerken.

Nu komt in demo07 nog een stukje voor gebruik met C++:

```
#ifdef __cplusplus
#   define CCALL    extern "C"  //C-call conventions
#else
#   define CCALL
#endif

...

CCALL void iopin_toggle(eIOPIN iopin);
CCALL void iopin_write(eIOPIN iopin, uint8_t ashigh);

#ifdef __cplusplus
class IoPin
{

...
    void toggle ()                  { iopin_toggle(this->nr);        }
    void write (uint8_t ashigh)     { iopin_write(this->nr,ashigh); }
protected:
private:
    eIOPIN nr;
};

#endif
```

Ik heb in het bovenstaande bewust gekozen voor het niet verstoppen van informatie, door voor elke C-functiedefinitie expliciet CCALL te zetten. Bij het zoeken van een functie valt het dan meteen op dat deze in C geschreven is, zonder de accolades van `extern "C" {...}` te moeten zoeken. Deze methode zorgt voor een sneller zicht op de aard van de functie.

De klassedefinitie is in de header gedaan, zonder .cpp-bestand, waardoor alle aanroepen 'inline' zijn. Deze keuze is ook hier gemaakt om expliciet zichtbaar te hebben wat er gebeurt en zodat meteen duidelijk is dat C++ hier exact dezelfde fucntionaliteit levert als C, uitgezonderd de variabelenopslag van het IO-pinnr.

Met bovenstaande 1-op-1 relatie tussen C en C++-code is het voor de compiler ook eenvoudig om de C++-code zodanig te optimaliseren dat deze nagenoeg even efficient is als C.

Bij flinke optimalisatie zou de compiler het object kunnen "virtualiseren" en de variabelenopslag achterweg laten om bij aanroepen de constante pinwaarde in te vullen in de C-routine en die rechtstreeks te benaderen.