Vergleich der einfachen IO Funktionen von 8 Bit Atmel AVR und 32 Bit ST32Fxxx ARM Mikrocontroller

Juni 2013 A.Schnell

Für einen neuen Mikrocontroller muss als erstes die Entwicklungsumgebung auf dem jeweiligen Rechner installiert und zum Laufen gebracht werden. Hier sind bereits die ersten Hürden zu überwinden, wobei die IDEs für 8 Bit Mikrocontroller wie zB. die AVR Serie von Atmel erheblich leichter zu installieren sind als die entsprechenden Entwicklungsumgebungen für 32Bit ARM Prozessoren. Dies liegt zum großen Teil an den wesentlich umfangreicheren Bibliotheken, die für die ARM Prozessoren mit in die auch komplexere 32Bit IDE eingebunden werden müssen.
Das erste Programm ist dann üblicherweise eines, welches eine LED (mit einen strombegrenzenden Widerstand) an einem IO Port ansteuert und zB. durch einen Button an einen der IO Ports ein bzw. ausschaltet oder zum Blinken bringt.
Die Funktion der IO Ports ist also das Erste was untersucht werden muss.

Die Struktur der 8 Bit AVR Mikrocontroller IO Ports ist hier ziemlich einfach zu verstehen:
es gibt ein Output Register, ein Input Register und ein Datenrichtungs Register, wobei jedes Bit dieser Register mit den entsprechenden Bits des IO Ports korrespondiert.

Um eine LED an zB. PortA Bit0, also PA0 einzuschalten muss also zunächst Bit0 des entsprechenden Datenrichtungsregisters auf 1 gesetzt werden, damit ist der PA0 auf die Outputfunktion geschaltet und anschließend wird Bit0 im Outputregister auf 1 gesetzt. Damit liegt an PA0 3 oder 5V an, je nach Betriebsspannung des Mikrocontrollers und die LED leuchtet.

Falls zB. an PA1 ein Taster angeschlossen ist, der den Eingang wahlweise auf Massepotential oder 3 bzw. 5V legt, und dieser Wert vom Programm als 0 oder 1 erfasst werden soll, muss im entsprechenden Datenregisterbit eine 0 stehen, dies ist im Normalfall nach einem Reset sowieso gegeben. Das Programm liest dann einfach den aktuellen Portwert ein (alle 8 Bit) und maskiert das gewünschte Bit mit einer bitweisen AND Verknüpfung heraus.

Bei einem 32Bit ARM Prozessor wie der ST32F1xx Serie von ST sieht das ganze allerdings etwas komplexer aus. Ich will versuchen, am Beispiel des STM32F3 Discovery Boards von ST (sehr preiswert, ca. 16Euro) dies hier zu erläutern.

Da sich die interne Registerbelegung der IO Komponente bei den verschiedenen STM32Fxxx Prozessoren unterscheidet (im Gegensatz zu AVR Serie) betrachte ich zunächst die Situation beim STM32F1xx.
Zunächst stellt man fest, das die vielen Ports des Prozessors jeweils 16 Bit Ports sind, also jeweils 16 Leitungen zur Verfügung stellen. Es gibt wie beim AVR auch hier jeweils 2 Register (allerdings 16 Bit), die direkt mit den jeweiligen Portleitungen verbunden sind und als GPIO Input Data Register **GPIOx_IDR** und GPIO Output Data Register **GPIOx_ODR** bezeichnet werden.

Die Funktion der einzelnen Portbits wird allerdings nicht durch ein einzelnes Bit in einem Datenrichtunsregister definiert, es sind je Portbit insgesamt 4 Bit (STM32F1xx Serie), die die jeweilige Funktion festlegen. Damit wird schon deutlich, das es deutlich mehr Konfigurationsmöglichkeiten je Portbit gibt. Die 4 Konfigurationsbits der STM32F1xx Serie, je 2 Bit genannt **cnf** und **mode** sind in 2 weiteren 32Bit Konfigurationsregistern entsprechend vor der ersten Verwendung des Ports zu setzten. Diese beiden 32Bit Register sind **GPIOx_CRL** und **GPIOx_CRH**. Abb.1 zeigt, wie jedes der Portbits mit seinen 4 Konfigurationsbits in den

Konfigurationsregistern verknüpft ist.

Abb.1 IO Register STM32F1xx Mikrocontroller

Die beiden **mode** Bits legen fest, ob Input oder Output und welche maximale Frequenz in der Outputfunktion möglich ist. Die beiden **cnf** Bits legen fest, ob bei der Inputfunktion ein analoger Wert eingelesen werden soll oder ob mit oder ohne internem PullUp bzw. PullDown Widerstand gearbeitet werden soll. In der Outputfunktion wird festgelegt, ob der Ausgang ein Push/Pull, Open Drain Ausgang sein soll oder an einer anderen internen Komponente angeschlossen werden soll. Damit ist es allerdings noch nicht genug. Es gibt ein weiteres 32 Bit Register, das Bit Set und Reset Register **GPIOx_BSRR** mit dem einzelne Bits des Outputregisters auf 1 oder 0 gesetzt werden können, ohne das ein Maskierung mit logischen Bitfunktionen erforderlich ist. Mit den unteren 16 Bit wird ein gesetztes Bit im entsprechenden Bit im Outputregister auf 1 gesetzt, ein gesetztes Bit im oberen 16 Bit Teil des BSRR setzt ein entsprechendes Bit im ODR auf 0 zurück.
Es gibt weiterhin je Port noch ein Lock Register **GPIOx_LCKR**, mit dem die Configurationsbits gegen Veränderung geschützt werden können und ein Alternate Funktion Register **GPIOx_AFR**, mit dem die alternativen Funktionen des IO Ports festgelegt werden können. Diese beiden Register sind hier zunächst für die elementaren IO Funktionen nicht notwendig,

Beim STM32F3xx ist das ganze etwas anders organisiert. Die mode bzw. cnf Bits sind in eigenen 32Bit Registern aufgeteilt, für Input bzw. Output gibt es eigene Register, sodass insgesamt 7 Bit je Portleitung konfigurierbar sind:

Abb.2. IO Register STM32F3xx Mikrocontroller

So enthält das **GPIOx_MODER** für die einzelnen Portleitungen jeweils die 2 Bit, die zwischen der Input Funktion und 3 möglichen Outputfunktionen unterscheiden. Das **GPIOx_OTYPER** legt mit seinen unteren 16 Bit für eine auf Output geschaltete Portleitung fest, ob der Output auf Push/Pull (0) oder OpenDrain (1) Funktion geschaltet werden soll.
Ein weiteres 32Bit Register **GPIOx_OSPEEDR** definiert mit jeweils 2 Bit pro Portleitung die maximale Ausgabegeschwindigkeit: x0: 2MHz, 01:10MHz und 11:50MHz.
Falls die Portleitung auf Input geschaltet ist, legen je 2 Bit im 32Bit **GPIOx_PUPDR** fest, ob die jeweilige Portleitung mit oder ohne PullUp bzw. PullDown Funktion betrieben werden soll. 00: no PullUP/PullDown, 01 mit PulUP, 10 mit PullDown, 11:reserved.
Das Lock und 2 Alternate Funktion Register je Port gibt es beim STN32F3xx natürlich auch noch.

Aber selbst wenn man alle Register korrekt konfiguriert hat, wird man feststellen das am jeweiligen Port nichts passiert. Dies liegt daran, das nach einem Reset zunächst jeder Port absolut inaktiv ist (floating), und erst durch Einschalten eines Peripherietaktsignals aktiv wird. Alle IO Komponenten sind zunächst abgeschaltet (ohne Taktsignal) um Energie zu sparen, nur die Komponenten die benötigt werden müssen durch ein entsprechendes Taktsignal aktiviert werden. Hierfür muss ein passendes Bit in einem Reset und Clock Control Register gesetzt werden. Sinnvollerweise verwendet man hierfür eine Bibliotheksfunktion.

Da sich die Konfigurationsbits in den verschiedenen Serien des STM32F in unterschiedlichen Registern befinden, ist das direkte Setzen der Konfigurationsbits bei der Initialisierung mühsam, Fehleranfällig und insbesondere nicht zwischen den verschiedenen Serien portabel.
Glücklicherweise gibt es aber portable Bibliotheksfunktionen, die dies ermöglichen.
Anbei nun ein kleines Programm, übersetzt mit dem gcc der IAR IDE. Diese Programm lässt zwei der LEDs auf dem Discovery F3 Board blinken, beim Betätigen des User Buttons blinken alle 8

LEDs.

Listing 1

Zunächst die obligatorischen include Dateien, für Standard Datentypen, die gpio defines und Strukturen und die Takt Funktionen. Diese müssen natürlich korrekt für die jeweilige STM32F ausgewählt werden.

Im Gegensatz zu den Fuse Bits der AVR Controller, welche die Taktquellen festlegen ist bei den ARM Prozessoren nach einem Reset immer zunächst ein interner Oszillator aktiv, der dann nach Start des Programms auf die gewünschte interne oder externe Taktquelle umgeschaltet werden kann. Dies passiert in der Funktion **InitClock()**, in der auf den externen High Speed Clock (8Mhz) des Discovery Boards umgeschaltet wird, intern ist der ARM dann mit 50 Mhz getaktet.

Als nächstes müssen die beiden Ports initialisiert werden. Das manuelle Setzen der pro verwendeten Portleitung 4 Bits (STM32F1xx) bzw. 7 Bits (STM32F3xx) in den diversen Konfigurationsregistern ist zwar möglich, aber einfacher ist dies durch die entsprechenden Bibliotheksfunktionen zu erledigen. Dies passiert in den Funktionen **InitGPIOE()** und **InitGPIOA().**

In diesen Funktionen wird zunächst das jeweilige Porttaktsignal eingeschaltet und anschließend ein struct **GPIO_InitStructure** mit den gewünschten Werten gefüllt, dessen Adresse dann anschließend der Bibliotheksfunktion **GPIO_Init()** als Parameter übergeben wird. Diese Funktion setzt dann die jeweiligen Bits in den verschiedenen Konfigurationsregistern der STM32F Serien für den jeweiligen Port.

Es folgt noch eine einfache Delayfunktion.

Im eigentlichen Hauptprogramm wird nach den Initialisierungen dann gezeigt, wie die Portbits beeinflusst werden können. Entweder durch Verwendung der Bibliotheksfunktion **GPIO_SetBits()** bzw. **GPIO_ResetBits()**. Man kann aber auch um mehr Transparenz und mehr Geschwindigkeit zu bekommen direkt das BSRR Register verwenden. Im C Program werden die IO Register indirekt adressiert zB. als **GPIOE->BSRR**. Wir erinnern uns das die unteren 16 Bit zum Setzen verwendet werden, also mit **GPIOE->BSRR = LED8|LED10;** werden beide LEDs eingeschaltet, mit **GPIOE->BSSR = (LED8|LED10)<<16;** werden die oberen 16 Bit angesprochen und entsprechend die LEDs ausgeschaltet. Da die direkt mit dem Port verbunden IO Register identisch sind bei den verschiedenen Serien, können diese Register direkt angesprochen werden.

Der anschließende **if(GPIOA->IDR&BUTTON)** Block zeigt, wie ein Portbit direkt abgefragt werden kann durch Lesen des Input Daten Registers IDR und Maskierung des entsprechenden Bits. Das Output Register ODR kann natürlich auch insgesamt gesetzt werden, allerdings werden dann alle 16 Portleitungen gleichzeitig beeinflusst. Besser ist hier sicherlich die Verwendung des BSRR Registers.

Das Beispiel zeigt, das zwar die Initialisierung der IO Ports deutlich aufwendiger ist im Vergleich zum AVR, allerdings das eigentliche Beschreiben und Lesen der IO Ports genauso übersichtlich programmiert werden kann. Die Möglichkeiten der Ports bei den STM32Fxxx Prozessoren sind aber deutlich vielfältiger.

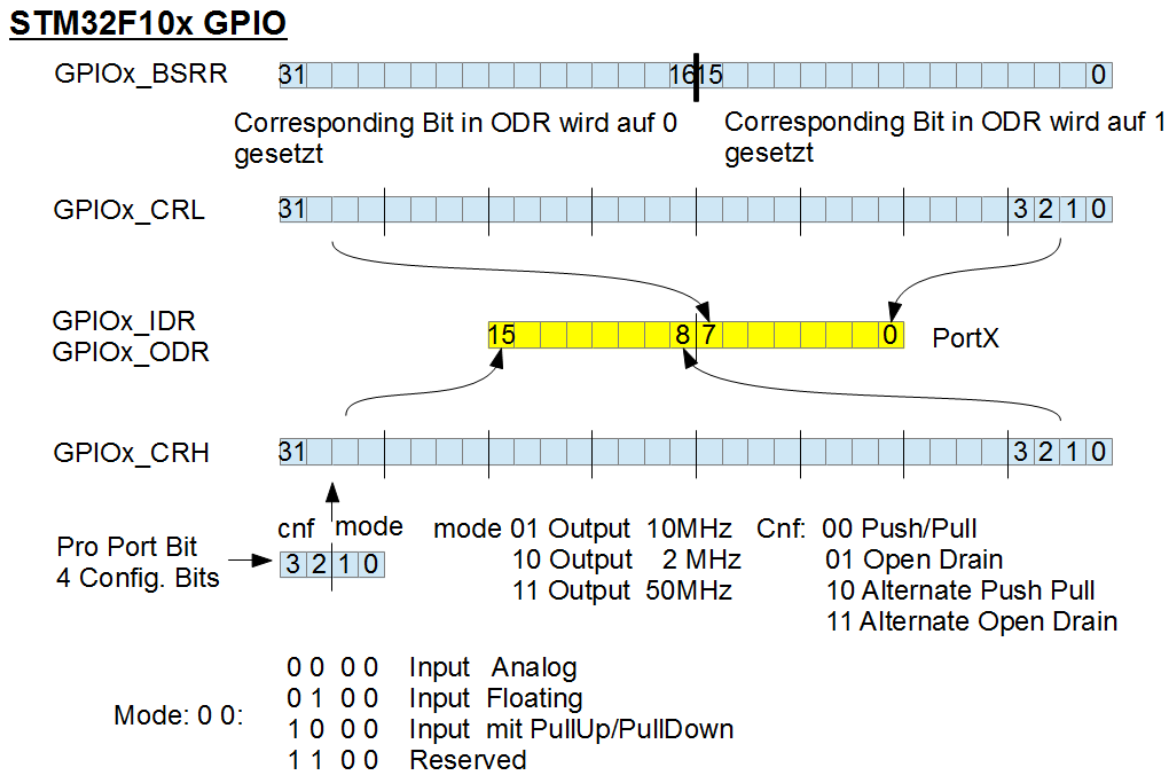


Abb.1 STM32F10x GPIO Register Struktur

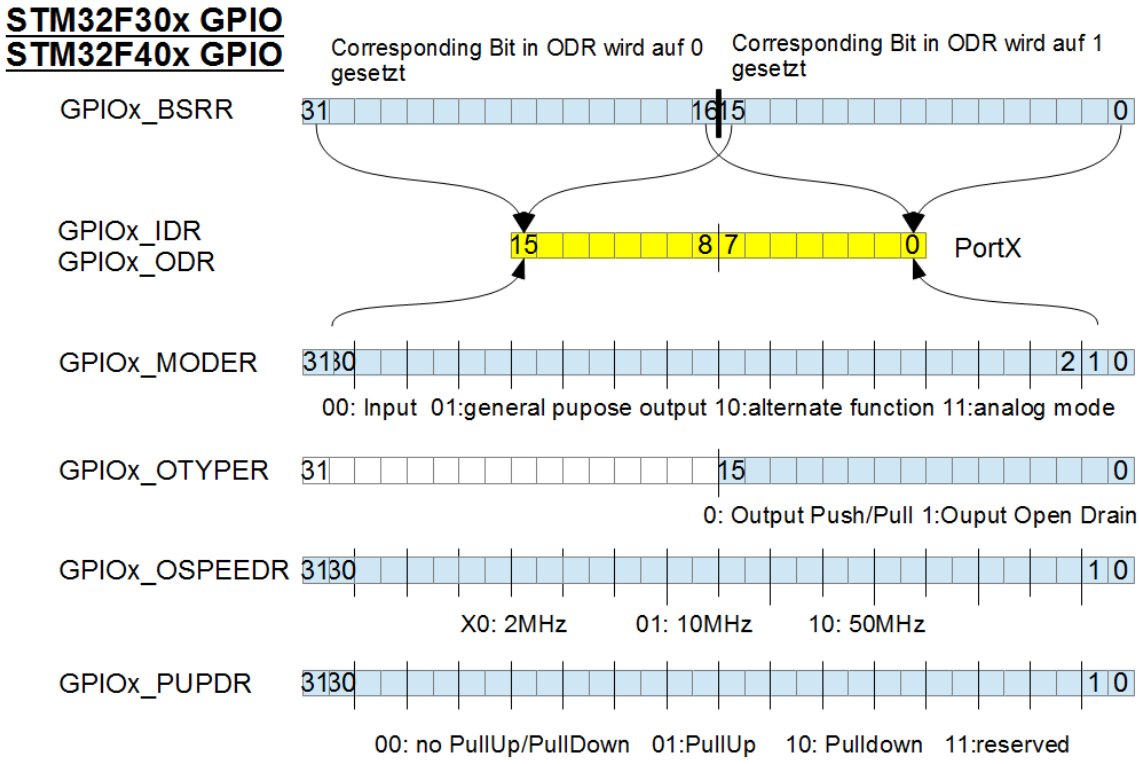
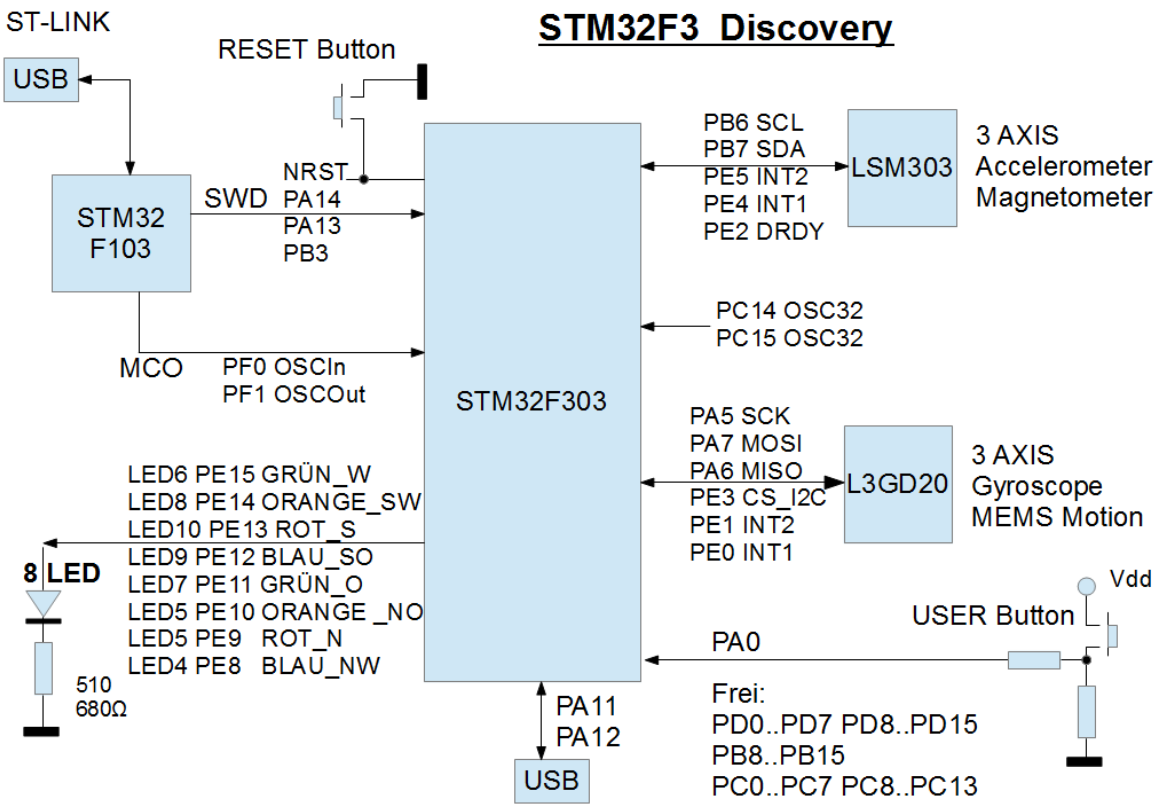


Abb.2 STM32F30x und STM32F40x GPIO Register Struktur



STM32F3 Discovery Board Blockschaltbild

```
// Simple GPIO Example for STM32F10x  und STM32F30x Discovery Modul  Juni 2013
//                                                                  A.Schnell

#include "stm32f30x.h"
#include "stm32f30x_gpio.h"
#include "stm32f30x_rcc.h"

#define BUTTON  GPIO_Pin_0              // User Button
#define LED6    GPIO_Pin_15
#define LED8    GPIO_Pin_14
#define LED10   GPIO_Pin_13
#define LED9    GPIO_Pin_12
#define LED7    GPIO_Pin_11
#define LED5    GPIO_Pin_10
#define LED3    GPIO_Pin_9
#define LED4    GPIO_Pin_8


void InitClock(void)
{  RCC_HSEConfig(RCC_HSE_ON);          // enables HighSpeedExternal HSE oscillator
    while(!RCC_WaitForHSEStartUp());   // wait till clock stabilizes
}

void InitGPIOE(void)                   // PE8 .. PE15 Output for LEDs
{                                      // ResetClockControl RCC
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOE,ENABLE); // enables clock to portE

    GPIO_InitTypeDef GPIO_InitStructure;
                              GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13|
GPIO_Pin_14|GPIO_Pin_15;
                              GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;          // _IN   _AF:Alternate  _AN:Analog
                              GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;      // _2MHz   _10MHz
                              GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;         // _PP:Push/Pull _OD:OpenDrain
                              GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;       // _UP   _DOWN
    GPIO_Init(GPIOE,&GPIO_InitStructure);
}

void InitGPIOA(void)                    // PA0  Input for User Button
{                                         // ResetClockControl RCC
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA,ENABLE); // enables clock to portA

    GPIO_InitTypeDef GPIO_InitStructure;
                              GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
                              GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;      // _OUT  _AF:Alternate  _AN:Analog
                              GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; // _2MHz  _10MHz
                              GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;    // _PP:Push/Pull _OD:OpenDrain
                              GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;    // _NOPULL  _UP   _DOWN
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

void Delay(uint32_t count)           // simple delay function
{  while(count>0) count--;  }

void main(void)
{
    InitClock();  // High Speed External Clock
    InitGPIOA();  // discovery modul PA0  User Button Input
    InitGPIOE();  // discovery modul PE8..PE15 LED Output

    for(;;)
    {
      GPIO_SetBits(GPIOE,LED8|LED10);   // Std Library
      GPIOE->BSRR = 0x00006000;         // direct write BitSetResetRegister BSRR
      GPIOE->BSRR = LED8|LED10;         // direct write with defines

      Delay(0x0000FFFF);

      GPIO_ResetBits(GPIOE,LED8|LED10);
      GPIOE->BSRR = 0x60000000;
      GPIOE->BSRR = (LED8|LED10)<<16;

      Delay(0x0000FFFF);

      if(GPIOA->IDR&BUTTON)    // direct read of IDR with mask
      {
        GPIOE->ODR = 0xff00;   // direct write to ODR, alle LEDs an  PE15..PE8 but PE7..PE0 to 0!
        Delay(0x0000FFFF);
        GPIOE->ODR = 0x0000;   // direct write to ODR, all 16 Portbits 0
      }
    }
}
```

Listing