I/O Ports

Gebruiken vanuit C/C++

- volatile byte* variabele.
- volatile struct ports* variabele.
- volatile byte array.
- volatile struct ports* variabele met union en bitfields.
- volatile union ports array met bitfields.
- volatile byte& variabele (C++).

Gebruik verbergen

- functies (inline C++).
- macro's
- class

THRijswijk

I/O Ports

volatile byte* variabele

```
int main() {
   typedef unsigned char byte;
   volatile byte* portc=(byte*)0x1003;
// ldy  #1003
   volatile byte* portb=(byte*)0x1004;
// ldx  #1004
   byte in, out, l, r;
   while (1) {
      in=*portc;
// ldab 0,y
// stab ...
      out=0;
      for (l=0x01, r=0x80; l!=0; l<<=1, r>>=1)
         if (in&r)
            out|=l;
      *portb=out;
// ldab ...
// stab 0,x
   }
   return 0;
}
```

THRijswijk

I/O Ports

const

```
int main() {
   typedef unsigned char byte;
   volatile const byte* const portc=(byte*)0x1003;
   volatile byte* const portb=(byte*)0x1004;
   byte in, out, l, r;
   while (1) {
      in=*portc;
// ldx  #1003
// ldab 0,x
// stab ...
      out=0;
      for (l=0x01, r=0x80; l!=0; l<<=1, r>>=1)
         if (in&r)
            out|=l;
      *portb=out;
// ldx  #1004
// ldab ...
// stab 0,x
   }
   return 0;
}
```

Het gebruik van **const** levert minder efficiënte code op. Hier is wel een (omslachtige) oplossing voor, zie http://bd.thrijswijk.nl/micb2/ports.htm

THRijswijk

I/O Ports

Wat willen we bereiken?

We willen dat 1 index register gebruikt wordt om de poorten te benaderen. De verschillende registers kunnen dan met verschillende offsets geadresseerd worden.

Op de een of andere manier moeten we de compiler "vertellen" dat portc en portb via opeenvolgende adressen te bereiken zijn.

THRijswijk

I/O Ports

volatile struct ports* variabele

```
int main() {
   typedef unsigned char byte;
   struct ports {
      byte c;
      byte b;
   } __attribute__((packed));
   volatile struct ports* port=(struct
                                ports*)0x1003;
// ldx  #1003
   byte in, out, l, r;
   while (1) {
      in=port->c;
// ldab 0,x
// stab ...
      out=0;
      for (l=0x01, r=0x80; l!=0; l<<=1, r>>=1) {
         if (in&r)
            out|=l;
      }
      port->b=out;
// ldab ...
// stab 1,x
```

THRijswijk

I/O Ports

volatile byte array

```
#define C 0
#define B 1

int main() {
   typedef unsigned char byte;
   volatile byte* port=(byte*)0x1003;
// ldx  #1003
   byte in, out, l, r;
   while (1) {
      in=port[C];
// ldab 0,x
// stab ...
      out=0;
      for (l=0x01, r=0x80; l!=0; l<<=1, r>>=1)
         if (in&r)
            out|=l;
      port[B]=out;
// ldab ...
// stab 1,x
   }
   return 0;
}
```

In plaats van #define kan ook een **enum** gebruikt worden:

```
enum {C, B};
```

THRijswijk

Vergelijk

struct en array

- Machine code is exact gelijk!
- Voordelen struct:
  - Kan ook word (16 bits) registers bevatten.
  - Zichtbaar in HLL variabelen window van THRSim11.
- Nadelen struct:
  - Niet standaard C/C++ __attribute__((packed)) nodig

```
-port @X              $1003
 -*port @$1003
  c @$1003            $03
  b @$1004            $c0
```

THRijswijk

I/O Ports

volatile struct ports* variabele met **union**s en **bitfield**s om afzonderlijke bits te benaderen

```
typedef unsigned char byte;
struct ports {
   union {
      byte c;
      struct {
         byte c7:1,c6:1,c5:1,c4:1,c3:1,c2:1,c1:1,c0:1;
      } __attribute__ ((packed));
   } __attribute__ ((packed));
   union {
      byte b;
      struct {
         byte b7:1,b6:1,b5:1,b4:1,b3:1,b2:1,b1:1,b0:1;
      } __attribute__ ((packed));
   } __attribute__ ((packed));
} __attribute__ ((packed));

int main() {
   volatile struct ports* port=(struct ports*)0x1003;
   port->b=0;
   while (1) {
      port->b7=port->c0;
```

THRijswijk

I/O Ports

volatile **union** ports array met **bitfield**s om afzonderlijke bits te benaderen

```
#define C 0
#define B 1

typedef unsigned char byte;

union ports {
   byte b;
   struct {
      byte b7:1,b6:1,b5:1,t4:1 b3:1,b2:1,b1:1,b0:1;
   };
};

int main() {
   volatile union ports* port=(union ports*)0x1003;
   port[B].b=0;
   while (1) {
      port[B].b7=port[C].b0;
```

THRijswijk

I/O Ports

volatile byte& variabele C++

```
int main() {
   typedef unsigned char byte;
   volatile byte& portc=
               *reinterpret_cast<byte*>(0x1003);
//  ldy#1003
   volatile byte& portb=
               *reinterpret_cast<byte*>(0x1004);
//  ldx#1004
   byte in, out, l, r;
   while (1) {
      in=portc;
//  ldab0,y
//  stab...
      out=0;
      for (l=0x01, r=0x80; l!=0; l<<=1, r>>=1)
         if (in&r)
            out|=l;
      portb=out;
//  ldab...
//  stab0,x
   }
   return 0;
}
```

THRijswijk

I/O Ports

Verbergen met class C++

! In C++ kun je zelf een class maken waarin alle details verborgen zijn:

```
class IOBox {
public:
   IOBox();
   byte getSwitch0() const;
// ...
   byte getSwitch7() const;
   byte getAllSwitches() const;
   void setLed0(byte b);
// ...
   void setLed7(byte b);
   void setAllLeds(byte b);
private:
   struct ports {
      byte c;
      byte b;
   };
   volatile ports* port;
};
```

THRijswijk

I/O Ports

class IOBox

```
#include "iobox.h"

int main() {
   IOBox iobox;
   iobox.setAllLeds(0);
   while (1) {
      iobox.setLed7(iobox.getSwitch0());
      iobox.setLed6(iobox.getSwitch1());
      iobox.setLed5(iobox.getSwitch2());
      iobox.setLed4(iobox.getSwitch3());
      iobox.setLed3(iobox.getSwitch4());
      iobox.setLed2(iobox.getSwitch5());
      iobox.setLed1(iobox.getSwitch6());
      iobox.setLed0(iobox.getSwitch7());
   }
   return 0;
}
```

THRijswijk

I/O Ports

class IOBox implementatie

! Door alle memberfuncties **inline** te definiëren (in iobox.h) levert deze versie ook een snel (en kort) programma op!

```
inline IOBox::IOBox(): port(
      reinterpret_cast<volatile ports*>(0x1003)
   ) {
}
inline byte IOBox::getSwitch0() const {
      return port->c&0x01;
}
// ...
inline byte IOBox::getAllSwitches() const {
      return port->c;
}
inline void IOBox::setLed0(byte b) {
      if (b) port->b|=0x01;
      else port->b&=~0x01;
}
// ...
inline void IOBox::setAllLeds(byte b) {
      port->b=b;
}
```

THRijswijk

I/O Ports

class IOBox implementatie

! Probleem bij gebruik van gcc: De machine code van de **inline** gedefinieerde functies (subroutines) komt ook in de .o file en in de .out file voor!

! De functieaanroepen worden wel keurig **inline** vervangen door de juiste machinecode.

! De in de .out file gedefinieerde functies worden dus niet aangeroepen maar nemen wel **242** bytes programmageheugen in beslag!

! Oplossing (na lang zoeken):

```
#pragma interface
class IOBox {
// ...
```

THRijswijk

Vergelijking

```
while (1) {
// ...
// PB5 = PC2
// ...
}
```

```
1 if (*portc&0x04) *portb|=0x20; else *portb&=~0x20;
2 if (port->c&0x04) port->b|=0x20; else port->b&=~0x20;
3 if (port[C]&0x04) port[B]|=0x20; else port[B]&=~0x20;
4 port->b5=port->c2;
5 port[B].b5=port[C].b2;
6 if (portc&0x04) portb|=0x20; else portb&=~0x20;
7 iobox.setLed5(iobox.getSwitch2());
```

Methode	optie -Os		optie -O3	
	size	speed	size	speed
1/6 pointer/ref	24	20	24	20
1 met const	25	25	24	20
2 struct	19	19	19	19
3 array	19	19	19	19
4 struct + union	27	42	27	42
5 array + union	27	42	27	42
7 class	24	23	24	19

size = bytes in main
speed = clockcycles nodig om while lus 1x uit te voeren

THRijswijk

I/O Ports

Waarom is machine code bij gebruik C++ class langer en trager?

Methode 2 struct

```
      ldx   #1003
.L1:  ldab  0,x
      andb  #4
      beq   .L2
      bset  1,x #$20
      bra   .L1
.L2:  bclr  1,x #$20
      bra   .L1
```

Methode 7 class

```
      ldx   #1003
.L1:  ldy   #1003
      ldab  0,x
      andb  #4
      beq   .L2
      bset  1,x #$20
      bra   .L1
.L2:  bclr  1,x #$20
      bra   .L1
```

! Verschillen:
" L1 staat niet op de optimale plaats
! oplossing: compileren met -O3 i.p.v. -Os
" ldy #$1003 is zinloos
! oplossing: ???

THRijswijk