Syften:

- Exemplifiera undantagshantering.

Software TRAP:

Övningar med grafikdisplayen omfattar färdiga rutiner för funktioner:

```
void graphic_initalize(void)
void graphic_clearScreen(void)
void pixelset( int x, int y)
void pixelclear( int x, int y)
```

Funktionerna är implementerade i såväl laborationssystem som simulator.

Man kan skapa inträden till sådana inbyggda funktioner med hjälp av en så kallad "TRAP" instruktion. ARM kallar detta "*Software call – SVC*".

Software Call- undantag genereras av motsvarande assemblerinstruktion:

```
__asm volatile (" .HWORD 0xDFxx\n");
```

där `xx` är en användardefinierad parameter till undantagsfunktionen.
Här använder vi parametrarna 0xF0, 0xF1, 0xF2 och 0xF3 för våra färdiga rutiner.

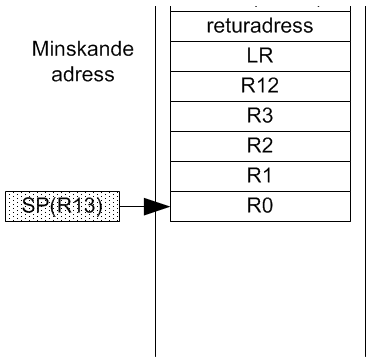
Låt oss se hur vi skapar kod för korrekt hantering av detta undantag.
Vi strävar efter att koda så lite som möjligt i assemblerspråk, men en liten del är oundvikligt.
Hanteringrutinen för undantaget utformar vi enligt:

```
__attribute__((naked))
void  SVC_Handler( void )   /* VARARGS ... */
{  /* Entry with exception stack */
   __asm volatile(" MOV  R0,SP\n");
   __asm volatile(" B    svc_call\n");
}
```

Adressen till denna funktion måste ha placerats på rätt adress i vektortabellen

Med kommentaren `VARARGS` menar vi helt enkelt att olika TRAP-anrop kan ha olika antal parametrar, etc. Minns att stacken vid denna tidpunkt ser ut som som figuren till höger.

Genom att vi i vår hanteringsrutin överför SP till R0 skapar vi en *godtycklig* aktiveringspost som kan avkodas i funktionen `svc_call`.



Kopplingen till de inbyggda funktionerna görs nu med en enkel "switch" sats där vi först extraherar "byte"-parametern från SVC-instruktionen till lokala variabeln oscall, därefter anropas den aktuella funktionen.

```
void  svc_call( unsigned int * call_args)
{
   /* Dispatcher */
   unsigned int oscall = ((char *) call_args[6]) [-2];
   switch( oscall )
   {
   case  0xF0: builtin_graphic_initalize(); break;
   case  0xF1: builtin_graphic_clearScreen();   break;
   case  0xF2: builtin_pixelset(call_args[0], call_args[1]); break;
   case  0xF3: builtin_pixelclear(call_args[0], call_args[1]); break;
   }
}
```

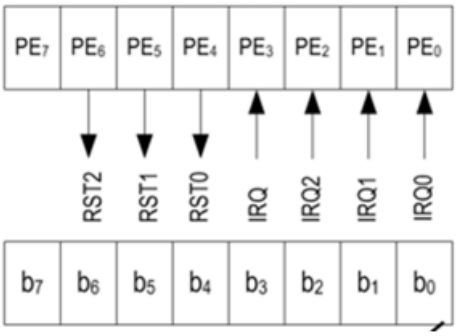
Anm: Innehållet på "returadress" är en 16-bitars instruktion. call_args[6] utgör hela instruktionen medan "-2" ger den lägre byten efter typkonvertering till char.

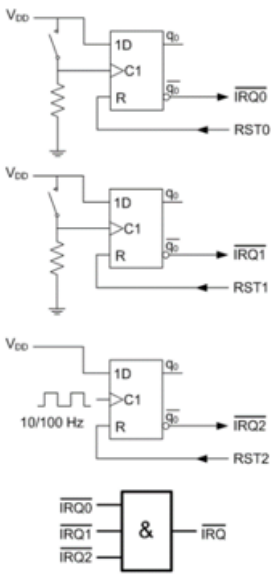
För de funktioner som har parametrar måste vi se till att dessa placeras korrekt i processorns register av kompilatorn. Men det blir enkelt, registerinnehållen finns ju som direkt offset till parametern call_args, så det gäller bara att på nytt pacera dessa i parameterregistren före anropet.

Externa avbrott – konfigurering och hantering

Vi bortser från applikationen och behandlar enbart den kod som krävs för att aktivera avbrottsmekanismerna.
Utgångspunkten är ett anslutet laborationskort "Flip-Flop".

Som vi kan se i figuren till höger kommer avbrottsignalen IRQ, vi kopplat till EXTI3 att aktiveras då någon av avbrottskällorna aktiveras. Vi kan, genom att läsa av statussignalerna identifiera vilken (IRQ0, IRQ1 eller IRQ2) och därefter kvittera det aktuella avbrottet genom att "toggla", dvs. först ettställa, därefter nollställa, motsvarande RST-signal.





Laborationskortet är anslutet till port E0..E7.

Förtydliga hur dessa makron bestäms:

```
#define EXTI3_IRQVEC          0x2001C064
#define NVIC_EXTI3_IRQ_BPOS   (1<<9)
#define EXTI3_IRQ_BPOS        (1<<3)
```

övriga bör vara förståeliga sedan tidigare...

```
void init_app ( void )
{
```

Konfigurera porten bit 0-3 och 7 som ingångar, bit 4-6 som utgångar.
Utgångar ska vara push/pull.
Ingångar PULL-UP, ty IRQ är aktiv låg.

```
   *GPIO_E_MODER = 0x00002500;
   *GPIO_E_PUPDR = 0x00000000;
   *GPIO_E_OTYPER= 0x00000000;
```

Toggla nu RST-bitarna (1-0 på RESET-ingången) på vipporna så att dessa återställs

```
   *GPIO_E_ODRLOW = 0x70;
   *GPIO_E_ODRLOW = ~0x70;
```

Koppla PE3 till EXTI3

```
   *SYSCFG_EXTICR1 |= 0x4000;
```

Möjliggör avbrott via EXTI3

```
    *EXTI_IMR |= EXTI3_IRQ_BPOS;
```

Avbrott genereras på negativ flank

```
   *EXTI_FTSR |= EXTI3_IRQ_BPOS;
   *EXTI_RTSR &= ~EXTI3_IRQ_BPOS;
```

Slutligen måste även NVIC möjliggöra detta avbrott

```
    *((unsigned int *) NVIC_ISER0) |= NVIC_EXTI3_IRQ_BPOS;
```

Och så avbrottsvektorn förstås....

```
    *((void (**)(void) ) EXTI3_IRQVEC ) = irq_handler;
}

void irq_handler ( void )
{
   unsigned char c = *GPIO_E_IDRLOW & 0xF;

   if(   *EXTI_PR & EXTI3_IRQ_BPOS )
   {
```

Återställ vippan som genererade avbrottet...

```
      if( c & 4 )
         *GPIO_E_ODRLOW = 0x40;
      else if ( c & 2 )
         *GPIO_E_ODRLOW = 0x20;
      else if ( c & 1 )
         *GPIO_E_ODRLOW = 0x10;
      *GPIO_E_ODRLOW = ~0x70;
```

Återställ slutligen avbrottet i EXTI-modulen

```
   *EXTI_PR |= EXTI3_IRQ_BPOS;
   }
}
```