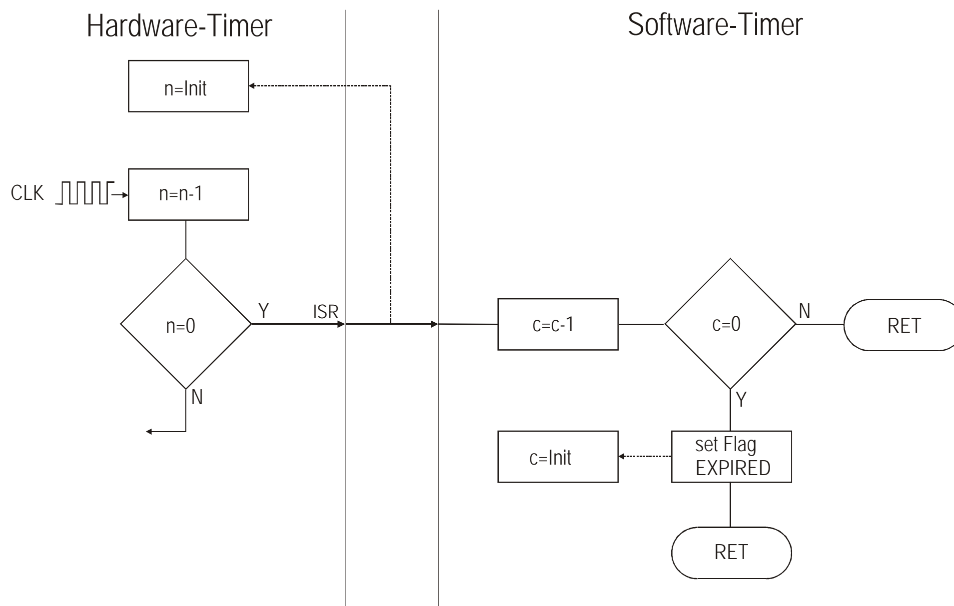
embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

1. Scope

Das Modul swTimer stellt Funktionen zur Verwendung von Software-Timern zur Verfügung. Software-Timer sind im Gegensatz zu Hardware-Timern unabhängig von der jeweiligen Hardware und können je nach Bedarf angefordert, verwendet und wieder freigegeben werden.

Bei der Implementierung von Software-Timern ist immer eine periodische Zeitreferenz notwendig. Diese wird üblicherweise durch einen Hardware-Timer realisiert, der fast immer auch Interrupt-fähig ist. Ein solcher Zähler verwendet ein 8 oder 16 Bit breites Register, das in einer vorgegebenen Taktrate (meistens Prozessortakt oder Vielfache/Teile davon) inkrementiert (+1) oder dekrementiert (-1) wird. Nachdem der Hardware-Timer mit einem Zählwert initialisiert und freigegeben ist, wird der Zählwert bei jedem Takt verändert. Erreicht der Zählerstand den Wert 0 (beim Inkrement Übergang von 0xff oder 0xffff nach 0) wird ein Interrupt ausgelöst. Dieser Interrupt löst den Aufruf einer Interrupt-Service-Routine (ISR) aus, in der der Timer Re-Initialisert werden muss (manchmal geht das auch automatisch - sogenannte LoopTimer) und der Vorgang kann sich zyklisch wiederholen.



(Bild HW-Timer mit ISR-Aufruf)

In der ISR, die durch den Hardware-Timer-Interrupt aufgerufen wird, erfolgt auch der Aufruf der Aktualisierungsroutine der Software-Timer, die damit im gleichen Zeitraster (=Auflösung) aktualisiert werden.

Der Hardware-Timer muss so eingestellt werden, das die minimal notwendige Auflösung der Software-Timer erreicht wird. Normalerweise beträgt die Auflösung 1ms oder 10ms. In besonderen Fällen können auch andere Auflösungen eingestellt werden – dieses hat aber zur Folge, dass den Initialisierungswerten der Software-Timer besondere Beachtung geschenkt werden muss. Benötigt man z.B. einen Timer mit einem Intervall von 60 Sekunden und verwendet man eine Auflösung von 4,5ms / Takt ist ein Startwert von 13333,3 für den Zähler notwendig. Hier muss man entweder eine Ungenauigkeit (Startwert 13333 statt 13333,3 => 0,0000225% Ungenauigkeit!) hinnehmen oder eventuell mit Fließkommazahlen bzw. anderen Tricks arbeiten.

Natürlich läßt sich der Ablauf des Hardware-Timers auch durch ständiges Abfragen (= Polling) erreichen – durch die ständigen Abfragen werden aber viele Taktzyklen verschwendet und somit leidet die Performance des Systems erheblich. Deshalb ist eine Interrupt-Abarbeitung immer vorzuziehen.

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

Bei der Realisierung eines Software-Timer-Moduls müssen verschiedene Dinge beachtet werden:

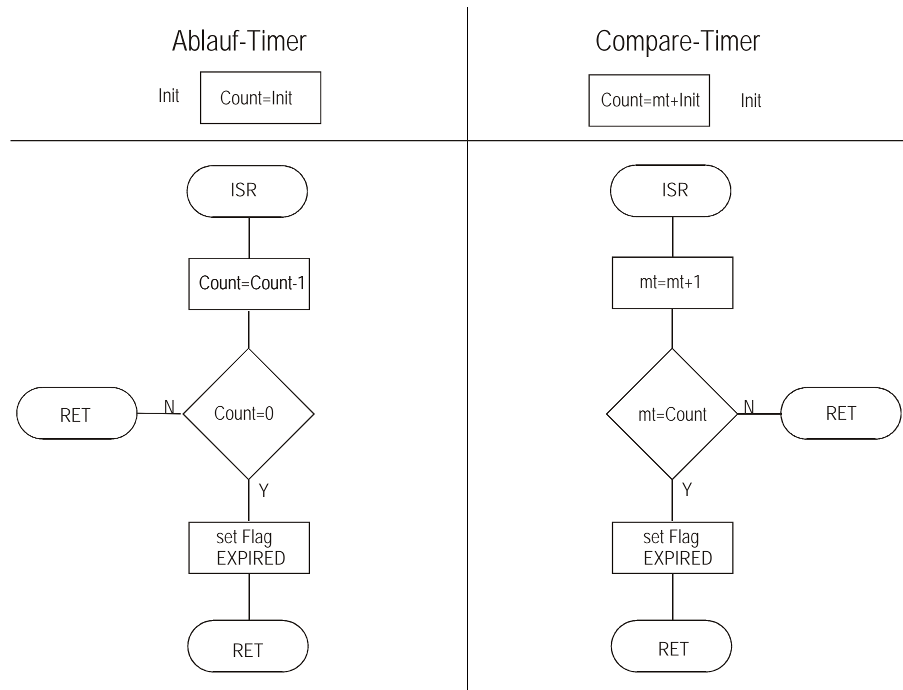
1) Die Auflösung der Software-Timer muss so gewählt werden, dass die minimale Zeitanforderung der Applikation bedient werden kann.
2) Die Routine zur Abarbeitung des Hardware-Timer-Interrupts darf nie länger dauern als der Abstand zwischen zwei Aktualisierungsaufrufen
3) Die Anzahl der benötigten Software-Timer sollt so kalkuliert werden, dass keine unnötige Zeitverschwendung innerhalb der Aktualisierung stattfindet

2. Konzept

Bei Software-Timern gibt es zwei unterschiedliche Konzepte:

1) Einfache Ablauf-Timer, die bei jeder Aktualisierung dekrementiert und beim Zählerstand 0 als Abgelaufen markiert werden

2) Compare-Timer, die beim Start mit dem momentanen MasterZählwert (mt) plus einer Ablaufzeit initialisiert und dann bei jeder Aktualisierung mit dem aktuellen Masterzählwert verglichen werden. Bei Übereinstimmung wird der Timer als Abgelaufen markiert.



(Bild zu 1 und 2)

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

Beide Konzepte haben Vor- und Nachteile. Diese beziehen sich im Wesentlichen auf die resultierende Anzahl von benötigten Speicherplätzen (=Größe der notwendigen Bytes pro Software-Timer) und auf die notwendigen Funktionalitäten der Applikation.

Bei den Ablauftimern kann man die Anzahl der notwendigen Speicherstellen reduzieren und damit auch ggf. den Zugriff auf die einzelnen Variablen im RAM beschleunigen.

Werden Ablauf-Timer mit bis zu 60 Sekunden pro Ablauf eingesetzt, so sind bei einer Auflösung von 1 ms jeweils 2 Bytes (= Integer) für Zähler und Nachladewert ausreichend. Verringert man die Auflösung auf 10ms so können Zeiten bis zu 600 Sekunden gezählt werden. Werden sowohl hohe Auflösung als auch lange Zählzeiten gewünscht , so kommt man um 4 Bytes (=LongInt) je Zähler und Nachladewert nicht herum. Dafür kann man aber selbst bei einer Auflösung von 1ms Zähler bis zu 1000 Stunden realisieren!

Bei den Software-Timern, die nach dem Prinzip des Vergleiches mit einem Masterzähler arbeiten, muss der Zählendwert beim Start aus dem aktuellen Stand des MasterZählers plus einem Zähloffset berechnet werden. Der MasterZähler wird bei jeder Aktualisierung der Software-Timer inkrementiert. Anschließend werden alle aktiven Software-Timer auf ihren Zählerendstand hin geprüft. Ist eine Übereinstimmung vorhanden wird der jeweilige Timer als abgelaufen markiert.

3. Implementierung

Basis der Software-Timer ist eine Datenstruktur, die für jeden Timer die Zählwerte und eine Steuervariable enthält. In der Header-Datei werden jeweils Datentypen für die einzelnen Variablen definiert und können je nach Bedarf angepasst werden.

```
(aus swTimer.h)

typedef unsigned int tht;     // TimerHandleType
typedef unsigned long tvt;    // TimerValueType


(aus swTimer.c)

typedef struct _tswt
{
  tht Ctrl;                   // Status / Control
  tvt Count;                  // Zaehler
  tvt Reload;                 // Reload-Zaehler
} tswt;

#define cswt_MAX           32            // Anzahl der maximalen SW-Timer
#define cswt_STOP          ((tht)0x00)
#define cswt_RUN           ((tht)0x01)
#define cswt_EXPIRED       ((tht)0x02)
#define cswt_RELOAD        ((tht)0x04)
#define cswt_FREE          ((tht)0x08)

#define cswt_NOTIMERFREE   ((tht)  -1)   // Error-Return value

tswt aTimer[cswt_MAX];        // Timer-Array, type tswt, size = cswtMAX
long MasterTimer=0;           // MasterCounter, Init = 0
```

Über die Steuervariable *Ctrl* werden die einzelnen Zustände (Stop, Run, Expired usw.) verwaltet. Der Zähler *Count* enthält den aktuellen Zählerstand. Der Zähler *Reload* wird nur für Timer verwendet, die nach dem Ablauf wieder initialisiert und neu gestartet werden. Die einzelnen Konstanten *cswt_???* werden innerhalb der Steuerung verwendet.

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

Das Software-Timer-Modul definiert die einzelnen Timer-Variablen über das Array *aTimer[]* mit der Größe *cswt_MAX*. Innerhalb der Update-Routine sowie in einzelnen Applikationsroutinen wird auf dieses Array über ein Zeiger vom Type *tswt* zugegriffen. Der Zugriff über Zeiger hat Geschwindigkeitsvorteile gegenüber den (besser lesbaren) Zugriffen über Indizes (z.B. *aTimer[Handle].Ctrl*).

Außerdem gibt es einen MasterTimer, der nach einem Reset mit 0 initialisiert und bei jedem Aktualisierungsaufruf um 1 inkrementiert wird.

Bevor die Timer verwendet werden können (und vor(!) dem ersten Aufruf von *swtUpdate()*) müssen alle Timer initialisiert werden. Das geschieht durch den Aufruf von *ResetAllTimer(0)*.

```
/******************************************************************************
name            : ResetAllTimer
description     : reset all timer (=free & stop)
*******************************************************************************
input parameter: StartHandle = number of 1st timer to clear
return value    : NO_ERROR
******************************************************************************/

tht swtResetAllTimer(tht start_thdl)
{
 tht i;
 for(i=start_thdl; i<cswt_MAX; i++) swtFree(i);
 return 0;
}
```

Danach kann der Hardware-Timer initialisiert und freigegeben werden.

Als Aktualisierungsrountine, die aus der HW-Timer-ISR aufgefruefen werden muss, dient *void swtUpdate(void)*:

```
void swtUpdate(void)            // call from HW-Timer-ISR
{
 tht i;
 tht Ctrl;
 tvt Count;
 tswt *pTimer;

 MasterTimer++;                 // aktualisize MasterTimer
 pTimer=aTimer;                 // reference to Timerarray
 for (i=0;i<cswt_MAX;i++)
 {
  Ctrl = pTimer->Ctrl;          // get control-info
  if (Ctrl & cswt_RUN)          // if timer is running...
 {
   Count = pTimer->Count;
   Count--;
   if (Count <= 0)              // count <= 0
   {
    Ctrl |= cswt_EXPIRED;       // yes, set expired-flag
    if (Ctrl & cswt_RELOAD)     // reload-timer-type?
     Count += pTimer->Reload;   // yes, reload counter
    else
     Ctrl &= ~cswt_RUN;         // no, stop timer
    pTimer->Ctrl = Ctrl;        // save control
   }
   pTimer->Count = Count;       // save count-value
  }
  pTimer++;                     // next timer
 }
}
```

Bevor ein Timer in der Applikation verwendet werden kann muss ein Handle erzeugt werden. Der Aufruf *timerhandle = swtRequest()* erzeugt ein solches Handle. Hat das handle den Wert –1, steht kein Timer zur Verfügung.

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

```
/*****************************************************************************
name           : TimerRequest
description    : Fordert einen Timer an
*****************************************************************************
input parameter: none
return value   : >=0                = Handle des Timers
                 cswt_NOTIMERFREE = kein Timer mehr frei !
****************************************************************************/
tht swtRequest(void)
{
  tht i;
  tswt *pTimer;
  pTimer = aTimer;

  for(i=0; i<cswt_MAX; i++)             // find first timer not in use
  {
    if (pTimer->Ctrl == cswt_FREE)      // if free timer found ...
    {
      pTimer->Ctrl = cswt_STOP;         // default = stop
      pTimer->Count = 0;                // count = 0
      pTimer->Reload = 0;               // reload = 0
      return i;                         // return handle
    }
    pTimer++;
  }
  return (cswt_NOTIMERFREE);            // Att. NO TIMER FREE !!!!
}
```

Wird ein Timer nicht mehr benötigt kann dieser wieder freigegeben werden. Mit *swtFree(timerhandle)* wird der Timer gestoppt und wieder freigegeben:

```
/*****************************************************************************
name           : TimerFree
description    : Gibt den Timer thdl (=Timernr) wieder frei
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value   : immer cswt_NOTIMERFREE
****************************************************************************/

tht swtFree(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Ctrl = cswt_FREE;
  aTimer[thdl].Count = 0;
  aTimer[thdl].Reload = 0;
 }
 return (cswt_NOTIMERFREE);
}
```

Nachdem ein Timer-Handle mit *timerhandle=swtRequest()* erzeugt wurde, kann dieser verwendet werden. Die Funktionen *swtStart()*, *swtReStart* und *swtStop()* sind hierfür vorgesehen:

```
/*****************************************************************************
name           : TimerStart
description    : Startet den Timer thdl (=Timernr) mit dem Wert time
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
                 time         : Zeit bis zum Ablauf des Timers (in ms)
return value   : NO_ERROR = kein Fehler festgestellt
****************************************************************************/

tht swtStart(tht thdl, tvt time)
{
```

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

```
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Count=time;
  aTimer[thdl].Reload = 0;
  if (!time) aTimer[thdl].Ctrl = cswt_EXPIRED;
  else       aTimer[thdl].Ctrl = cswt_RUN;
  return 0;
 }
 else
  return (cswt_NOTIMERFREE);
}
```

Die Funktion *swtReStart* aktiviert den Timer mit Wiederholfunktion. D.h. ist der Timer abgelaufen, wird das Expiredflag gesetzt und anschließend wird der Timer erneut gestartet:

```
/*******************************************************************************
name           : TimerReStart
description    : Startet den Timer thdl (=Timernr) mit dem Wert time
                 und setzt das Reload-Register
********************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                               [ =Rückgabewert der Funktion TimerRequest() ]
                 time         : Zeit bis zum Ablauf des Timers (in ms)
return value   : NO_ERROR = kein Fehler festgestellt
*******************************************************************************/

tht swtReStart(tht thdl, tvt time)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Count=time;
  aTimer[thdl].Reload = time;
  if (!time) aTimer[thdl].Ctrl = cswt_EXPIRED | cswt_RELOAD; // new 07/17/01
  else       aTimer[thdl].Ctrl = cswt_RELOAD | cswt_RUN;
  return 0;
 }
 else
  return (cswt_NOTIMERFREE);
}
```

Gestoppt wird ein Timer mit *swtStop()*. Die Funktion hält den Timer an und setzt die Variablen für *Count* und *Reload* zurück. Das Handle für den Timer wird nicht freigegeben!

```
/*******************************************************************************
name           : TimerStop
description    : Versetzt den Timer thdl (=Timernr) in den Stop-Zustand
********************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                               [ =Rückgabewert der Funktion TimerRequest() ]
return value   : NO_ERROR = kein Fehler festgestellt
*******************************************************************************/

tht swtStop(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Ctrl = cswt_STOP;
  aTimer[thdl].Count=0;
  return 0;
 }
 else
  return (cswt_NOTIMERFREE);
}
```

Soll der Timer nur vorübergehend angehalten werden verwendet man die Funktion *swtPause()*:

```
/*******************************************************************************
name           : TimerPause
description    : Versetzt den Timer thdl (=Timernr) in den Pause-Zustand
```

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

```
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value   : NO_ERROR = kein Fehler festgestellt
****************************************************************************/

tht swtPause(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Ctrl &= ~cswt_RUN;
  return 0;
 }
 else
  return (cswt_NOTIMERFREE);
}
```

Soll ein Timer, der sich im Pause-Zustand befindet, wieder weiter laufen, steht die Funktion *swtContinue() zur Verfügung*:

```
/****************************************************************************
name           : TimerContinue
description    : Versetzt den Timer thdl (=Timernr) wieder in den Run-Zustand
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value   : NO_ERROR = kein Fehler festgestellt
****************************************************************************/

tht swtContinue(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  aTimer[thdl].Ctrl |= cswt_RUN;
  return 0;
 }
 else
  return (cswt_NOTIMERFREE);
}
```

Die Frage, ob ein Timer bereits abgelaufen ist (= expired) kann durch die Funktion *swtExpired()* ermittelt werden.
Achtung: Die Funktion löscht den Zustand Expired nachdem die Funktion verlassen wird. Den Zustand Expired kann man also nur einmalig abfragen!

```
/****************************************************************************
name           : TimerExpired
description    : Es wird getestet, ob der Timer thdl (=Timernr)
                 abgelaufen ist. Wenn ja, erfolgt ein Rueckgabewert <>0
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value   : 0   = Timer läuft noch
                 <>0 = Timer abgelaufen
****************************************************************************/

tht swtExpired(tht thdl)
{
 tht ret = 0;
 if (thdl != cswt_NOTIMERFREE)
 {
  if (aTimer[thdl].Ctrl & cswt_EXPIRED)
  {
   aTimer[thdl].Ctrl &= ~cswt_EXPIRED;
   ret = 1;
  }
 }
 return ret;
}
```

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

Wer den aktuellen Zählerstand eines Timers ermitteln will bedient sich der Funktion *swtRead()*:

```
/*****************************************************************************
name            : TimerRead
description     : Liefert die Restlaufzeit von Timer thdl (=Timernr)
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value    : Restlaufzeit des Timers
*****************************************************************************/

tvt swtRead(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  return aTimer[thdl].Count;
 }
 else
  return (0);
}
```

Und wenn der Status eines Timers in der Applikation ausgewertet werden soll kann man diesen mit der Funktion *swtStatus()* ermitteln:

```
/*****************************************************************************
name            : TimerStatus
description     : Liefert den Status von Timer thdl (=Timernr)
*****************************************************************************
input parameter: thdl : Handle des verwendeten Timers
                                [ =Rückgabewert der Funktion TimerRequest() ]
return value    : Status des Timers
*****************************************************************************/

tht swtStatus(tht thdl)
{
 if (thdl != cswt_NOTIMERFREE)
 {
  return aTimer[thdl].Ctrl;
 }
 else
  return (cswt_FREE);
}
```

4. Verwendung in der Applikation

Die Implementierung und Verwendung von swTimer besteht aus folgenden Schritten:

1) include von swTimer.h
2) ResetAllTimer(0) aufrufen
3) HardwareTimer initialisieren und starten
....
4) Timer anfordern: *TimerHandle = swtRequest()*
5) Timer starten: *swtStart(TimerHandle, 1000);*
6) Ablauf abwarten: *while (!swtExpirde(TimerHandle)) { ... }*
7) Timer freigeben: *swtFree(TimerHandle);*
....

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

In dem Beispielprogramm kommen zwei Timer zur Anwendung. Zunächst wartet das Programm auf einen Tastendruck. Anschließen wird für 60 Sekunden eine LED im Rhythmus von 500ms zum Blinken gebracht. Als Hardware-Timer wird der Watchdog-Timer eine Hitachi H8S2612 verwendet.

```
(File: swt-Example.c)

#include „swTimer.h“

#include "ioh82612.h"

/*******************************************************************************
name            : WDTimerInit
description     : Initialisierung des WD-Timers
*******************************************************************************
input parameter: none
return value    : none
*******************************************************************************/

void WDTimerInit (void)
{
 swtResetAllTimer(0);            // alle SW-Timer init
 WDT_W_TCNT = 0x5A06;            // Interval Timer = 256-16000/64
 WDT_W_TCSR = 0xA539;            // Interval Timer Funktion, TimerEnable, clk/64/249 = 1kHz
}

//--------------------------------------------------------------------------------------//
// 1000 Hz / 1ms Interrupt as Interval Timer
//--------------------------------------------------------------------------------------//
interrupt [WDT_WOVIO] void tick_timer(void)
{
   char i;
   i = WDT_R_TCSR;               // dummy read
   WDT_W_TCNT = 0x5A06;          // Interval Timer = 256-16000/64
   WDT_W_TCSR = 0xA539;          // Interval Timer Funktion, TimerEnable, clk/64/250 = 1kHz
   swtUpdate();                  // call swtupdate
}


void main(void)
{
  int th1,th2;

  WDTimerInit();                                    // init WatchDog-/ Intervall-Timer
  RS232Init();                                      // init Debug interface
  set_interrupt_mask(0);                            // enable ints

  V24DebugOut("=\n\rTimer-Example\n\r");            // display start msg.

   th1 = swtRequest();                               // get timer1
   th2 = swtRequest();                               // get timer2

  while (1==1)                                       // loop
  {
      if (kbhit())                                  // wait for keypress
      {
         swtStart(th1,60000);                       // start timer1 for 60s
         swtReStart(th2,500);                       // start timer2 for 500ms
         LED = 1;                                   // LED = ON
         while (!swtExpired(th1))                   // while NOT timer1 expired
         {
            if (swtExpired(th2)                     // timer2 expired ?
            {
               if (LED == 1) LED = 0;               // yes, toggle LED
               else LED = 1;
            }
         }
      }
      LED = 0;                                      // LED = OFF
   }
```

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

```
   swtFree(th1);                                     // Timer1-Free
   swtFree(th2);                                     // Timer1-Free
}
```

5. Systembelastung durch Software-Timer

Die Implementierung von Software-Timern führt zu einer zusätzlichen Belastung des Systems. Die ständigen aber regelmäßigen Aufrufe der ISR-Routine sowie die Aktualisierung der Software-Timer-Zähler benötigen Taktzyklen, die für die Applikation nicht mehr zur Verfügung stehen. Die untenstehende Tabelle zeigt die Belastung einer Applikation durch die Verwendung von einer unterschiedlichen Anzahl von Timern.

Die Untersuchung wurde auf einem Hitachi-Controller (H8S2612) mit einer Taktfrequenz von 16 MHz durchgeführt. Als Hardware-Timer dient der Watchdog-Intervalltimer. Zur Ermittlung der Systembelastung wurde in die Hardware-ISR (Auflösung 1ms) ein Zähler eingebaut, der in jedem Interrupt um eins inkrementiert wird. Außerdem gibt es einen Hauptschleifenzähler, der solange inkrementiert wird, bis der ISR-Zähler den Stand 1000 (=1000ms = 1s) erreicht hat. Der resultierende Zählwert dient als Referenz für die jeweilige Software-Timer-Systembelastung.

```
(File: swt-Belastung.c)

#include "ioh82612.h"

#include „swTimer.h“
long TimerTickCount;
long LoopCount;

/*****************************************************************************
name           : WDTimerInit
description    : Initialisierung des WD-Timers
*****************************************************************************
input parameter: none
return value   : none
*****************************************************************************/

void WDTimerInit (void)
{
 swtResetAllTimer(0);            // alle SW-Timer init
 WDT_W_TCNT = 0x5A06;            // Interval Timer = 256-16000/64
 WDT_W_TCSR = 0xA539;            // Interval Timer Funktion, TimerEnable, clk/64/249 = 1kHz
}

//-------------------------------------------------------------------------------------------//
// 1000 Hz / 1ms Interrupt as Interval Timer
//-------------------------------------------------------------------------------------------//
interrupt [WDT_WOVIO] void tick_timer(void)
{
   char i;
   i = WDT_R_TCSR;               // dummy read
   WDT_W_TCNT = 0x5A06;          // Interval Timer = 256-16000/64
   WDT_W_TCSR = 0xA539;          // Interval Timer Funktion, TimerEnable, clk/64/250 = 1kHz
   TimerTickCount++;             // inkrement TimerTickCount
   swtUpdate();                  // call swtupdate
}

void main(void)
{
  int count,hndl;

  WDTimerInit();                                        // init WatchDog-/ Intervall-Timer
  RS232Init();                                          // init Debug interface
  set_interrupt_mask(0);                                // enable ints
```

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

```
  V24DebugOut("=\n\rTimer-Testprogram\n\r");          // display start msg.

  for (count=0; count<cswtMAX; count++)               // Init-Timers (here 32)
  {
    hndl = swtRequest();                              // get timer
    swtReStart(hndl,count*3+5);                       // set time for restart timer
  }

  while (1==1)                                        // loop
  {
    LoopCount=0;                                      // reset loop counter
    TimerTickCount=0;                                 // reset TimerTickCounter
    while (TimerTickCount<1000) LoopCount++;          // wait until TimerTickCounter==1000
    V24DebugOut("\n\rLoopCount : %ld",LoopCount);     // display loop counter value
  }
}
```

Funktion	Counts	Belastung in %
ohne swtUpdate()	692725	
Timeranzahl = 0	668900	3,5%
Timeranzahl = 5	666200	3,8%
Timeranzahl = 10	665400	4,0%

Tabelle 1. Berechnung der Systembelastung durch Software-Timer (maximal 10 Software-Timer, cswt_MAX = 10)

Funktion	Counts	Belastung in %
ohne swtUpdate()	692725	
Timeranzahl = 0	668800	3,5%
Timeranzahl = 5	665100	4,0%
Timeranzahl = 10	661600	4,5%
Timeranzahl = 20	654970	5,5%
Timeranzahl = 32	649230	6,3%

Tabelle 2. Berechnung der Systembelastung durch Software-Timer (32 mögliche Software-Timer, cswt_MAX= 32)

Aus der Tabelle wird ersichtlich, dass selbst bei der aktiven Verwendung von 32 Software-Timern nur eine Systembelastung von 6,3% entsteht. Ergo stehen noch 93,7% für die Applikation zur Verfügung. Allerdings muss man beachten, dass die 6,3% jeweils en Block durch die ISR-Routine benötigt werden und damit die Applikation zyklisch für 6,3% der Zeit vollständig unterbrochen wird!

Die Anzahl der notwendigen Software-Timer sollte deshalb sorgfältig geprüft werden!

embesso®
embedded software solutions
Hinter der Bahn 1a • D 31162 Bad Salzdetfurth
Tel.:+49 5064 9504 -30 • Fax:+49 5064 9504 -59
http://www.embesso.com

6. Zeitmessung

Das Modul swTimer bietet auch Funktionen zur allgemeinen Zeitmessung zwischen Ereignissen. Diese sind insbesondere bei der Implementierung neuer Funktionen oder bei der Kommunikation mit externen Partnern hilfreich.

Über die Funktion *long GetMasterTimer()* läßt sich der Masterzähler auslesen und verwenden. Im folgenden Beispiel soll dies gezeigt werden:

```
(File: swtMeasurementDemo.c)

long MeasStartVal, MeasEndVal, MeasDiffTime;

void main(void)
{
   WDTimerInit();                                     // init WatchDog-/ Intervall-Timer
   RS232Init();                                       // init Debug interface
   set_interrupt_mask(0);                             // enable ints

   V24DebugOut("=\n\rTimer-Measurement\n\r");         // display start msg.

   while (1==1)
   {
      MeasStartVal = GetMasterTimer();                   // get StartTime
      while (...) { ... }                                // wait event
      MeasEndVal = GetMasterTimer();                     // get EndTime
      MeasDiffTime = MeasEndVal - MeasStartVal;          // calculate difference
      V24DebugOut("\n\rWait-Time: %ld ms",MeasDiffTime);   // display loop counter value
   }
  }
}
```

7. Files

Das swTimer-Tool besteht aus den Files:

swTimer.c // C-File mit alle swTimer-Funktionen
swTimer.h // Header-File,

swtExample.c // Beispielfile für die Verwendung von swTimer

swTimer.pdf // Dieses Dokument

8. Copyright / Haftung

Das Modul **swTimer** ist Freeware. Wird es verwendet ist der Copyright-Vermerk von **embesso**® im Quellcode anzugeben. Eine Weitergabe der Quellcodes gegen Entgelt ist unzulässig! Eine Verwendung unterliegt dem vollständigem Haftungsausschluss durch den Autor!