Multitasking mit dem MSP430

Jens Altenburg

Die klassische Main-Loop verknüpft mit einigen Interrupt-Service-Routinen dürfte wohl immer noch das am häufigsten verwendete Softwaredesign in Mikrocontrolleranwendungen sein. Multitasking Systeme, egal ob kooperativ oder preemptiv, sind für den "Einmann-Softwareentwickler" wohl eine Nummer zu groß, so die landläufig geäußerte Meinung. Kommerzielle Betriebssysteme, wie OSEK, QNX, etc. bleiben doch lieber hochkomplexen Anwendungen in Großprojekten vorbehalten.
Der besagte "Einmann"-Programmierjob spielt aber in vielen Ingenieurbüros oder kleineren Entwicklungsfirmen immer noch eine wesentliche Rolle. Im folgenden wird deshalb ein pfiffiges Multitasking-System für den MSP430 vorgestellt, welches überschaubar bleibt und nur geringe Ressourcen erfordert.

Noch ein Betriebssystem

Noch ein Betriebssystem, stöhnt der Experte. Multitasking-Systeme gibt es doch beinahe soviel wie Sand am Meer. Auch der Einwand, dass für viele Systeme Lizenzgebühren fällig werden, ist kein rechter Grund, eine neue Variante zu programmieren. Die Verwendung einer Reihe kommerzieller Systeme ist an Universitäten und Hochschulen bzw. für den privaten Gebrauch (wozu braucht man eigentlich ein RTOS privat?) kostenfrei. Und designt man Großserienprodukte, fallen Lizenzgebühren meist nicht ins Gewicht.
Bleiben die bereits erwähnten "Einmann"-Programmierer. Doch besonders hier sind die Animositäten auch nicht ohne. Immerhin wird der eingesetzte Controller oft bis zum äußersten ausgereizt, jeder minimale Softwareoverhead riskiert einen Bausteinwechsel und das kostet. Und außerdem, warum sollte ein Multitasking irgendetwas verbessern? Wohin das führt, sieht man ja beim PC. Das Win95 auf meinem Schreibcomputer booted in 86 Sekunden, das Win2000 auf dem Firmendesktop braucht glatte 35 Sekunden länger (und das bei einem mehr als 10fach schneller getakteten Prozessor).
Nun, diese Diskussion bringt nichts. Die wesentlichste Frage ist die nach der auf dem Controller laufenden Applikation. Die typische Standardanwendung der vergangenen Jahre fragt einige Tasten ab, schaltet ein paar Lampen und wenn’s hoch kommt, werden noch einige Kennlinien aus Tabellen abgelesen. Härteste Echtzeitforderungen stehen oft auf der Tagesordnung dieser Programme. Hier ein Multitasking-System 'draufzupfropfen ist wirklich nur hemmungslos verblendeten Software-Freaks anzuraten.
Interessant wird es jedoch genau dann, wenn umfangreiche Berechnungen mit verwickelten Algorithmen Kern der Applikation sind.

Komplexe mathematische Berechnungen sind durch eingebaute Hardware-Multiplizierer und Hochsprachen-Programmierung auch für kleinere Mikrocontroller kein Angst-Thema mehr (der MSP430 ist u.a. zum Teil auch deswegen mit einer leistungsfähigen CPU + Multiplizierer ausgestattet).
Die Berechnung der Laufzeit solcher Algorithmen ist jedoch im Vorfeld einer Entwicklung praktisch unmöglich. Es wird demnach enorm schwierig, diese Berechnungen in ein (möglicherweise nur schlecht modifizierbares) Laufzeitsystem zu implementieren. Die Gretchenfrage der Controllertechnik erhebt ihr häßliches Haupt; einfache Implementierung der Algorithmen und schlechtes Zeitverhalten oder umgekehrt. Der Worst-Case-Fall, kurz vor Produktauslieferung können selbst geringste Änderungen fatale Folgen haben. Stellt sich natürlich die Frage, womit ein Multitasking-System derlei Schrecken bannt?

Kooperatives vs.präemptivesMultitasking

Es gibt zwei unterschiedliche Ansätze für Multitaskingsysteme, kooperativ bzw. präemptiv. Der wesentlichste Unterschied zwischen beiden ist die Art der Zuteilung der CPU-Zeit, denn eines ist klar, auch ein noch so komplexer Rechner kann mit einer CPU nur ein Programm zu einem bestimmten Zeitpunkt ausführen. Dass der PC beim Artikelschreiben nebenbei eine CD abspielt oder Seti@Home ausknobelt, ist das Ergebnis, der mehr oder minder intelligenten Ressourcenverteilung des Betriebssystems.
Die CPU schaltet zwischen der Bearbeitung der verschiedenen Tasks um. Auf dem PC wäre das Schreibprogramm, der CD-Spieler sowie weitere Programme als jeweils ein Task anzusehen. Auf dem Mikrocontroller ist ein Task oft ein Programmodul. Ein derartiger Task wird initialisiert, kommt dann in eine Warteschleife (der Task ist *pending*), wird bearbeitet (*running*) und kann beendet werden (*terminate*).

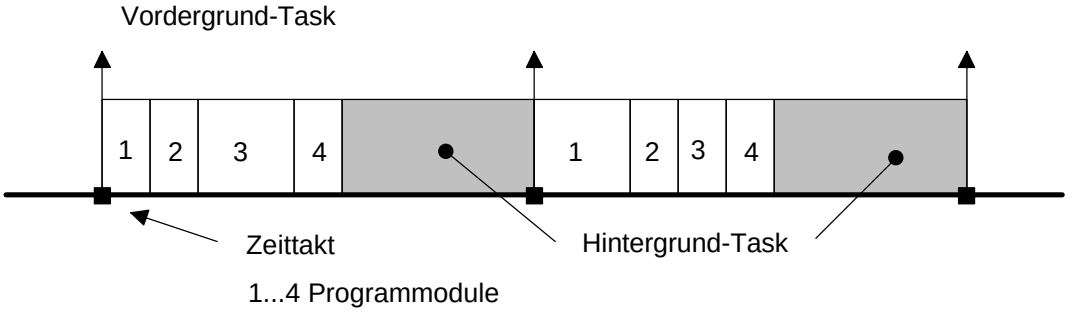

Bild 1 Zeitablauf der Tasks eines kooperativen Multitaskings. Die Einzeltasks (Task 1...4) werden von einem Zeittakt zyklisch gestartet. Jeder Task terminiert freiwillig.

Die Abbildung 1 verdeutlicht dieses Konzept. Eine Anzahl von, im Beispiel vier, verschiedenen Programmodulen wird von einer Main-Loop zyklisch aufgerufen. Der Zeittakt, z.B. 50 ms, wird von einem Timer abgeleitet und die Programmodule werden nacheinander abgearbeitet.

Jedes Programmodul besitzt eine eigene Modul-Loop. Diese kann nun z.B. in eine Funktionspointerliste, eventuell zusammen mit einer Variablen, die den Modulaufruf in Vielfachen des Grundtaktes enthält, zusammengestellt werden. Die Liste dieser Funktionsrufe repräsentiert damit das Softwaregrundsystem.

```
/* Implementierungsbeispiel   eines einfachen kooperativen   Multitaskingsystems   */
/* Funktionspointerliste    */
const  stCyclFunc  SYS_Active[]  = {  vTask1           /*   */
                                   , 19                /* Aufruf alle 19+1 * 50 ms   */
                                   ,   vTask2
                                   , 0                 /* alle 50 ms  */
                                   ,   vTask3
                                   , 0                 /* dito  */
                                   ,   vTask4
                                   , 500
                                     };
/* Main-Loop  */
while(1){
       if(SYS_stFlag.Timer1){                    /* Flag  wird durch Timer gesetzt   */
       SYS_stFlag.Timer1   = 0;
       for(i  = 0; i <  (sizeof(SYS_Active)   / sizeof(stCyclFunc));   i++){
              if(abActiveTick[i]   == 0){
                     abActiveTick[i]   = SYS_Active[i].bTimeStamp;
                     SYS_Active[i].fFunc();
                     }
              else  {
                     abActiveTick[i]--;          /* Timestamp  jedes  Tasks  */
                     }
       }
```

Weitere Module werden einfach in die Funktionspointerliste eingesetzt. Das Systemverhalten ist relativ gut vorhersehbar und die Softwarestruktur bleibt überschaubar.
Ein Pferdefuß dieses Vorgehens resultiert aus der festen Zeitstruktur. Es können nur so viele Tasks bearbeitet werden, wie in dem vorgegeben Zeittakt unterzubringen sind. Die einfache Variante, bei Zeitproblemen den Takt zu verlängern, ist gleichzeitig beinahe die schlechteste. Die Taktzeit bestimmt die Reaktionsgeschwindigkeit des Systems, wird sie zu groß, leidet die Performance. In Grenzen kann durch Verlagern von Systemfunktionen in Interrupts hier Abhilfe geschaffen werden.
Das andere Problem ist beinahe noch gravierender. Hauptmerkmal eines kooperativen Multitasking ist die freiwillige Freigabe der CPU-Zeit durch den aktiven Task. Für die praktische Implementierung bedeutet dies, dass rechenaufwendige Algorithmen in kurze Segmente zergliedert werden müssen. Eine solche Anforderung kann schnell zur Herausforderung für den Programmierer mutieren. Insbesondere die Abschätzung der Rechenzeit für jeden Abschnitt ist riskant. Was auf einem Prozessortyp schnell berechnet ist, kann auf einer anderen Plattform für Zeitprobleme sorgen. Der ganze mühsam portierbar geschriebene Code muss adaptiert werden.
Ohne konkrete Tests ist eine Rechenzeitabschätzung im Vorfeld einer Entwicklung auf dieser Basis eher Glückssache. Der konservativ denkende Ingenieur wird also den maximal leistungsfähigen Controller verlangen, währenddessen sich die Verkaufsabteilung wegen der Kosten die Haare rauft.
Beim präemptiven Multitasking braucht sich der Anwender um die CPU-Zeitverteilung keine Gedanken zu machen. Die Algorithmen werden nicht in Stücke zerteilt, jeder Task besteht aus

einer "Endlosschleife". Es spricht also vieles für ein präemptives Multitasking, nur die Einkaufsabteilung nicht, denn nun rauft die sich wegen der Kosten und Lizenzgebühren die Haare. Der (mit nur wenig Haaren versehene) Entwickler darf sich nun aussuchen, wessen Haare verstrubbelt besser aussehen.
Nun, man könnte es ja auch selber machen. Dieser vorsichtig formulierte Gedanke, führt vorschnell einem Verantwortlichen anvertraut und zur Realisierung angemahnt, mit Sicherheit zu schlaflosen Nächten, überzogenen Terminen und Gesundheitsschäden des Entwicklers (verstrubbeltes Resthaar), denn so ein Multitasking programmiert sich nicht so nebenbei, oder doch?

Multitasking "light"

Für die Industrie werden eine Reihe leistungsfähiger Produkte angeboten. Systeme wie QNX™ (*www.qnx.com*) oder CMX-Tiny+™ (*www.cmx.com*) sind dazu einige Beispiele. Im Automotive-Bereich sind Systeme auf der Basis von OSEK stark verbreitet.
Allen diesen Produkten wird eine entsprechende Leistungsfähigkeit bescheinigt. Features wie z.B. *control tasks*, *control events*, *true preemption*, *cooperative scheduling allowed*, *fast context switch times* oder *low interrupt latency* sind Bestandteil des Funktionsumfangs. Hinzu kommen noch Grafikbibliotheken oder TCP/IP-Protokoll Stacks, die ebenfalls angeboten werden.
Diese Eigenschaften haben dann natürlich auch ihren Preis. Je nach System werden einmalige Kosten oder Lizenzgebüren (*royalties*) auf jedes verkaufte Gerät erhoben. Für den Einsatz in der Forschung bzw. Ausbildung gibt es teilweise Sonderpreise, gelegentlich ist die Verwendung hier kostenlos, z.B. µCOSII.
Neben den Kosten ist zum Teil aber auch der Funktionsumfang der Systeme so groß, dass entweder erhebliche Ressourcen (RAM bzw. ROM) belegt werden oder der Einstieg in die komplexen Systeme so kompliziert ist, dass der eine oder andere damit schlichtweg überfordert wird.
Ein einfaches System für den MSP430 benötigt längst nicht alle der genannten Funktionalitäten. Folgende Minimalforderungen scheinen unverzichtbar:

- starten, verwalten und abbrechen von Tasks
- zeitgenauer Aufruf von Funktionen (Echtzeittasks)
- Möglichkeit zur Priorisierung der Rechenleistung
- Debug-Unterstützung
- minimaler Verbrauch von Ressourcen, RAM, ROM, CPU-Zeit

Auf den Gebrauch von *Messages*, priorisiert präemptiver Taskverwaltung und Interrupttasks kann verzichtet werden. Die Verwendung externer Biliotheken (Grafik, TCP/IP,...) wird zum gegenwärtigen Stand ebenfalls nicht als notwendig erachtet.
In [1] wird ein Konzept zu einem minimalistischen Multitasking-Kernel beschrieben, welches viele der geforderten Eigenschaften aufweist. Das dort beschriebene Multitasking-System ist für ein einfaches präemptives Multitasking ausgelegt. Jedem Task wird eine spezifische Laufzeit zugemessen, nach dem er unterbrochen und der nächste Task aus der Taskliste gestartet wird. Alle Tasks besitzen die gleiche Priorität, d.h. kein Task kann einen anderen vor Ablauf dessen Zeitscheibe unterbrechen. Das System stellt folgende Funktionalität bereit:

- *UEXC_CreateTask()* - Task starten
- *UEXC_KillTask()* - Task abbrechen
- *UEXC_HogProcessor()* - Rechenzeit zurTasklaufzeitanfordern

Diese drei Funktionen scheinen auf den ersten Blick keine besondere Leistungsfähigkeit zu versprechen. Ein einfaches System ist zwar akzeptabel, aus 'einfach' darf allerdings nicht 'primitiv' werden, sonst überwiegen die Nachteile eines solchen Systems gegenüber der bekannten Main-Loop.
In einem präemptiven Multitasking-System gibt der gerade aktive Task die Kontrolle über die CPU nicht "freiwillig" ab. Vom Prinzip her ist dem Einzeltask das Schicksal seiner Konkurrenten gleich. Für die Umschaltung zwischen den Tasks ist ein spezieller Mechanismus erforderlich. Dieser Vorgang wird als Kontext-Switch bezeichnet. Bekanntermaßen liegen die lokalen Variablen einer Funktion (der Task ist eine Funktion) im Stackbereich. Theoretisch könnten diese Werte natürlich in einen reservierten Speicher umkopiert werden. Praktisch lässt sich die Anzahl dieser Variablen zum Umschaltzeitpunkt schwer ermitteln.
Wesentlich eleganter ist es, den Stackpointer zu "verbiegen". Jeder Task benötigt einen für ihn reservierten Speicherbereich. Die Größe des Speichers richtet sich nach der Zahl der lokalen Variablen, der Verschachtelungstiefe weiterer Funktionsaufrufe und eines minimal notwendigen Bereiches für Interrupts, die den Task unterbrechen dürfen.
Der Stackpointer zeigt zur Laufzeit des Tasks in diesen Speicherbereich. Aufgabe des Kontext-Switch ist die Neupositionierung des Stackpointers in den Bereich des neu aktivierten Tasks bei gleichzeitiger Sicherung der Variableninhalte des "alten" Tasks.
Das Verständnis dieser Umschaltung setzt eine minimale Kenntnis des inneren Aufbaues des MSP430 voraus. Die CPU besitzt 16 Register. Über deren Zuweisung für lokale Variable entscheidet der verwendete Compiler. Dazu später.
Eine Programmunterbrechung des laufenden Tasks erfolgt durch einen Interrupt, vornehm als Scheduler bezeichnet. Beim Interrupt werden automatisch der Program Counter (PC) und das Status Register (SR) auf den Stack gelegt. Jetzt entscheidet der Scheduler, ob ein Kontext-Switch stattfindet. Ist dem so, wird der Stackpointer umkopiert und ein so genannter

Interrupt-Stack-Frame gebastelt. Dann erfolgt der RETI-Befehl. Das Ganze sieht sehr einfach aus, ist dennoch eine kitzelige Angelegenheit. Das Verlassen des Schedulers gaukelt der CPU die Rückkehr aus einer ganz anderen Funktion, nämlich den nun aktiven Tasks vor. An diesem geht der Task-Switch völlig unbemerkt vorüber. Das ist vergleichbar mit einem gewöhnlichen Interrupt. In diesem Falle "merkt" die Main-Loop auch nur dann etwas von einem Interrupt, wenn gezielt über globale Variablen kommuniziert wird.
Die Informationen, die die Task-Umschaltung benötigt, werden im Task-Control-Block zusammengefasst. Die meisten Elemente dieser Struktur sind einsichtig. Die drei Zeiger beinhalten die Werte des jeweiligen Task-Stacks sowie die Grenzen des Stackbereichs. Klar ist auch der Funktionspointer, er enthält den Verweis auf den Task (Funktion). Über die *char* Variablen wird der zeitliche Aufruf bzw. die Zustandssteuerung gehandelt. Mit dem *struct *next* Zeiger wird der nächste zu aktivierende Task indiziert.

```
typedef  struct  TaskControlBlock
    {
    struct   TaskControlBlock   *next;
    unsigned  char  tid;                  /* Task ID */
    unsigned  char  state;                /* Taskzustand  */
    unsigned  char  ticks;                /* Anzahl der  Zeitscheiben  */
    unsigned  char  current_ticks;        /* verbleibende Zeitscheiben   */
    void     (*func)(void);               /* Zeiger auf  Task */
    unsigned  char    *stack_start;       /* Stack-Beginn  */
    unsigned  char    *stack_end;         /* Stack-Ende  */
    unsigned  char    *sp;                /* Stackpointer  */
    }  TaskControlBlock;
```

Die Hauptfunktion des UEXEC-Multitasking-Systems ist die Speicherverwaltung und Zeitscheibenverteilung der CPU-Zeit an die jeweiligen Tasks. Da jeder Task unabhängig von allen anderen Tasks arbeitet, hat er seinen eigenen Stackbereich, der beim Taskwechsel umgeschaltet wird (*sp,...). Die Zeitscheibenverwaltung erfolgt über Timer-Ticks. Der Scheduler wird zyklisch aufgerufen und entscheidet anhand der verstrichenen *ticks*, ob eine Taskumschaltung erfolgt.
Mit Hilfe der variablen Zeitscheibenzuweisung ist eine Priorisierung der Tasks untereinander möglich, derjenige mit der höchsten Priorität bekommt anteilig die meiste Rechenzeit.
Diese Art der Zeitzuweisung ist jedoch nicht einer wirklichen Priorisierung gleichzusetzten. UEXC unterscheidet nicht zwischen "normalen" und "Interrupt-Task". Wird in der Interruptservice-Routine anstelle der direkten Bearbeitung ein Task gestartet, so reiht sich dieser in die Liste der auszuführenden Tasks ein. Wann dieser Task zur Ausführung kommt, ist unbestimmt. Beträgt der Timer-Tick z.B. eine Millisekunde und es wären zwei Tasks mit je 5 ms Laufzeit vor dem neuen Task in der Liste, wäre eine Verzögerung von 10 ms die Folge. Da aber ein Task zur Laufzeit neue Zeit anfordern kann, z.B. in Abhängigkeit eines Sensorsignales, besitzen diese 10 ms allenfalls statistischen Charakter. Dieser Zeit-Jitter in der Bearbeitung zeitkritischer Programmteile wäre das k.o.-Kriterium, das den Einsatz von UEXC verbieten würde; er muss also umgangen werden.
Es ist zuerst zu untersuchen, welche Programmodule harten Echtzeitanforderungen genügen müssen und welche nicht. Zu den Kandidaten mit harten Echtzeitanforderungen zählen z.B. Kommunikations-Tasks oder Schrittmotorsteuerung´, die im Beispiel starre Zeitregimes

erfordern. Hinzu kommt der periodische Aufruf des Schedulers für die Taskumschaltung. Die Abbildung 2 verdeutlicht die Zusammenhänge. Der Scheduler wird zyklisch aufgerufen und schaltet die Tasks um. Programmteile, die harten Echtzeitanforderungen genügen müssen, werden als Interrupt-Funktion angelegt. Deren Priorität ist höher als der Scheduler-Interrupt. Ein Interrupt kann nur durch einen höher priorisierten Interrupt und nur wenn das Interrupt-Enable-Flag gesetzt ist, erneut unterbrochen werden. Bei diesem mit *nested interrupt* bezeichneten Verfahren muss auf eine entsprechende Stackgröße geachtet werden, zuviele verschachtelte Interrupts sollten vermieden werde.

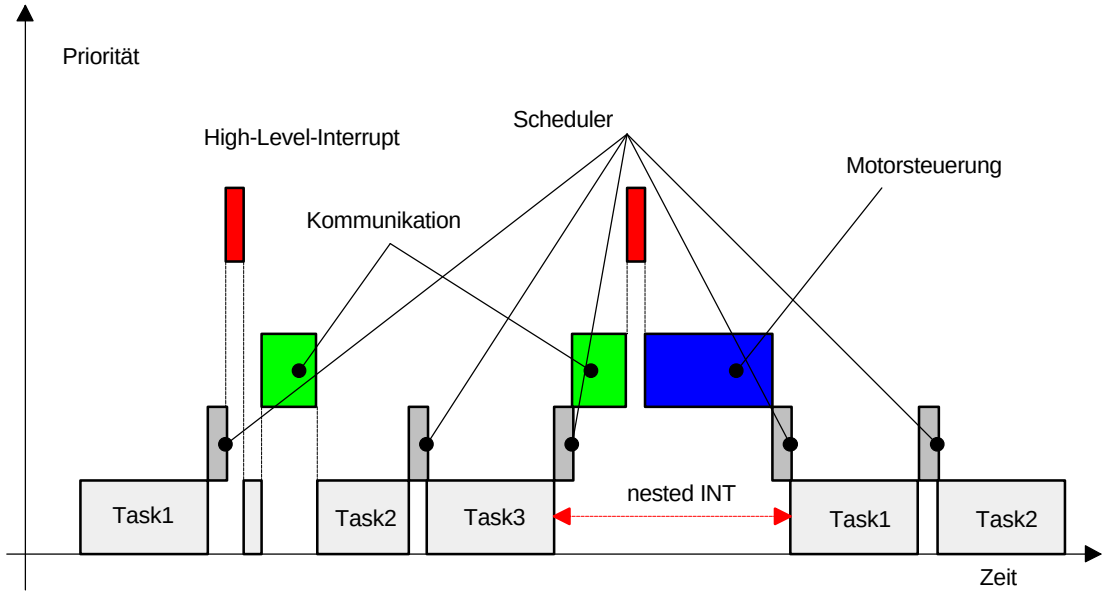

Bild 2 Grafische Darstellung der zeitlichen Zurodnung der CPU-Zeit zu Tasks, Interrupts und Scheduler. Besonders wichtig ist die exakte Priorisierung von Tasks, Scheduler und Interrupts.

Es ist streng darauf zu achten, dass der Scheduler keinesfalls einen laufenden Interrupt unterbrechen darf. In einem solchen Fall würde ein Kontextswitch stattfinden, der anschließende RETI aber nicht zum neuen Task sondern zum unterbrochenen Interrupt zurückkehren. Ein Systemchrash ist die zwangsläufige Folge.

Portierung auf den MSP430

Bis jetzt war das alles reine Theorie. Bei der Umsetzung geht es ans Eingemachte. Das eigentliche Multitasking liegt als C-Quellcode vor (*www.imagecraft.com*). Zwei Beispielimplementationen für den HC11 bzw. HC12 liefern Ansatzpunkte zur Portierung auf den MSP430. Mit geschätzten zwei Dutzend Programmzeilen Code scheint dies eine Fingerübung zum Auffrischen der angestaubten Assembler-Kenntnisse zu werden. Wieso nur die Scheu der Programmiergurus vorm Multitasking?
Vom Prinzip her ist die saloppe Aussage nicht einmal falsch. Wenn alles funktioniert, sieht die Lösung tatsächlich simpel aus. Im übrigen sind beinahe alle guten Lösungen simpel.

Doch wenn es nicht auf Anhieb gelingt, entwickelt sich die Fehlersuche zum Horrortrip. Schon die Initialisierung des Systems mit seinen verketteten Zeigern und TCB-Listen treibt Schweißperlen auf die Stirn. Wohl dem, der die Professional-Version des Compilers besitzt und sich die Inhalte der zirkular verlinkten Structs im Debugfenster direkt anzeigen kann. Die anderen dürfen mittels Memory-Dump und dem Abzählen der Offsets zur Startadresse eines TCBs den Inhalt desselben erraten.
Doch halt, so schlimm ist es auch wieder nicht. Auf jeden Fall sollte man den Mechanismus des Kontext-Switch genau durchdenken.

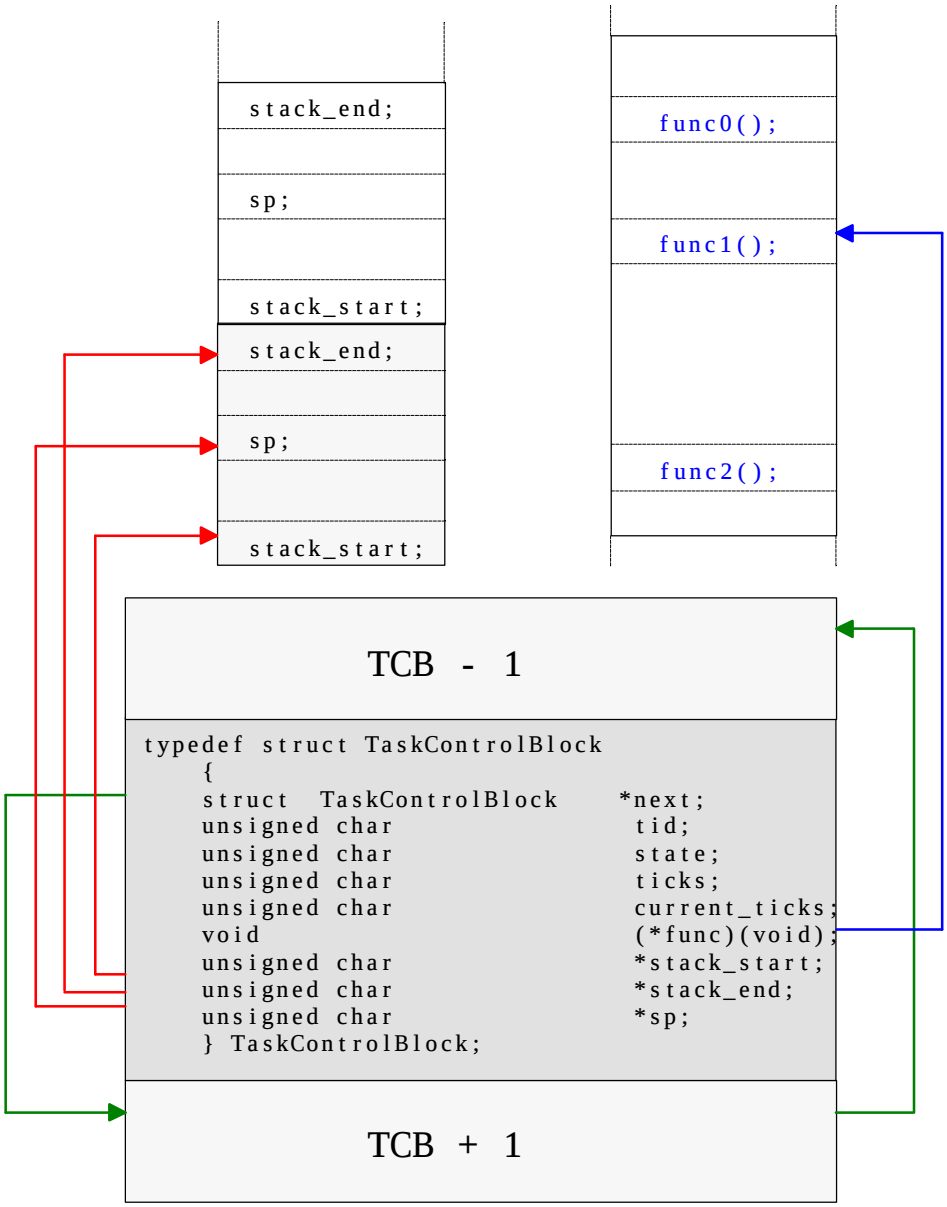

Bild 3 Zusammenhang zwischen Task-Control-Block (TCB), Speicherbereich des aktiven Tasks (RAM) sowie Verweis auf Programmcode der Task-Funktion (ROM). Die einzelnen TCB sind über eine Liste untereinander verbunden. Über einen Zeiger erfolgt der Verweis auf den jeweils nächsten Block (grüne Pfeile). Weitere Zeiger dienen zur Zuweisung von Task-Funktionen (blaue Pfeile) bzw. steuern den Zugriff auf den Speicher (rot).

Anhand des Beispielprojektes (ct04.prj vom Heise-Server) werden die Fußangeln der Portierung verdeutlicht. Nach der Initialisierung des MSP430 werden zwei Beispiel-Tasks kreiert, vTask0 und vTask1. Deren Aufgabe ist das Toggeln von Port P2.0 bzw. P2.1 mit unterschiedlicher Frequenz.
Mit *UEXC_CreateTask()* werden die Speicherbereiche (*abTask0[], abTask1[]*) zugewiesen und der zugehörige TCB initialisiert. Vom Nutzer unbemerkt, erfolgt auch das Aufsetzen eines NullTasks, welcher nicht beendet werden kann und das System am laufen hält, auch wenn kein User-Task aktiv ist.
Mit den ersten Aufruf des Schedulers (*UEXC_StartScheduler()*) startet die Programmabarbeitung. Der erste in der Linkliste stehende Task, d.h. der letzte kreierte Task, wird aufgerufen. Diesen Vorgang sollte man durchaus einmal mit dem Debugger schrittweise verfolgen.

```
/*************************************************************************
* Kurzbeschreibung     : Orginal UEXC Funktion
* Autor                : Richard F. Man
*************************************************************************/
void UEXC_Schedule(void){
    current_task = current_task->next;   /* Zeiger auf nächsten Task */
    current_task->current_ticks = current_task->ticks; /* Task-Zeitscheiben */
    uexc_current_sp = current_task->sp;         /* Zeiger auf neuen Stackpointer */
    if (current_task->state == T_READY)
        UEXC_Resume();                          /* Task existiert bereits */
    else {                                      /* task->state == T_CREATED */
        current_task->state = T_READY;
        uexc_current_func = current_task->func; /* Funktionspointer laden */
        UEXC_StartNewTask();                    /* MSP430 Implementierung */
        }
    }

/*************************************************************************
* Kurzbeschreibung     : Startet den neuen Task
* Autor                : Jens Altenburg
*************************************************************************/
void UEXC_StartNewTask(void){
asm(
     " mov     %uexc_current_sp, SP\n"   /* moves content of uexc_cu.. to SP */
     " push.w  #%UEXC_KillSelf\n"        /* Killself() onto stack */
     " push.w  %uexc_current_func\n"     /* new function to stack */
     " mov     SR,%wRegister\n"          /* Statusregister bearbeiten */
     " bis     #8,%wRegister\n"          /* GIE (global interrupt enable) */
     " push.w  %wRegister\n"             /* statusregister to stack */
     " reti\n"                           /* go */
     );
    }
```

Gelangt der Scheduler zur Ansicht, dass ein neuer Task gestartet wird, erfolgt dies über *UEXC_StartNewTask()*. Diese Funktion ist prozessorspezifisch und deswegen implementationsabhängig. Die Kommunikation erfolgt über die globale Variable *uexc_current_sp*
In der Assemblerfunktion wird zuerst der Stackpointer des MSP430 auf den neuen Speicherbereich gesetzt, Danach wird die Adresse der Funktion *UEXC_KillSelf()* auf den Stack gelegt. Diese Funktion entfernt den gerade initialsierten Task aus der Taskliste wenn dieser beendet wird. Als nächstes landet die Startadresse des Tasks auf dem Stack. Die folgenden Befehle basteln einen so genannten Interrupt-Stack-Frame, d.h. den gleichen Stackzustand, den der Prozessor bei einem normalen Interrupt automatisch generiert. Der Assemblerbefehl *reti* startet den neuen Task.

Bei dem eben beschriebenen Mechanismus muss man ein wenig um die Ecke denken. Nach dem *reti* holt der MSP430 das Statusregister vom Stack. In diesem wurde zuvor das GIE-Bit gesetzt, sonst wären alle weiteren Interrupts gesperrt. Anschließend lädt der Prozessor den Programmcounter mit dem folgendem Wert vom Stack. Wie durch Zauberhand ist der Prozessor nun im neuen Task. Jetzt werden alle lokalen Variablen, Unterprogrammaufrufe, etc. wieder über den Stack abgewickelt.
Ist der Task beendet, räumt er seinen Stackbereich auf und die Funktion kehrt regulär mit *ret* zurück. Auf dem Stack liegt hier die Adresse der Funktion *UEXC_KillSelf()*. Selbige ordnet nun die Liste der TCB neu.
Damit ist klar, wie ein Task gestartet wird. Doch wie erfolgt der Wechsel zwischen den Tasks?
Über einen Interrupt wird die Funktion *UEXC_CheckTask()* gestartet. Eine Sicherheitsabfrage prüft, ob der Stackpointer im erlaubten Bereich ist, dann wird getestet, ob die Anzahl der dem Task zugemessenen Zeitscheiben (*time slices*) abgelaufen ist. Wenn ja, kommt der Scheduler zum Zuge. Da alle Tasks nun vom Typ *T_READY* sind, wird nur der Stackpointer umkopiert und mit *UEXC_Resume()* der Kontext-Switch veranlasst. Dazu muss eine eigene Assemblerroutine geschrieben werden.

```
/*****************************************************************************
* Kurzbeschreibung     : Resume a previously stopped task
* Autor                : Jens Altenburg
*****************************************************************************/
void UEXC_Resume(void){
asm(
     " mov   %uexc_current_sp, SP\n"     /* Stackpointer setzen */
     " pop   r15\n"               /* die vorher geretten CPU-Register */
     " pop   r14\n"               /* vom Stack holen */
     " pop   r13\n"
     " pop   r12\n"
     " pop   r11\n"
     " pop   r10\n"
     " pop   r9\n"
     " pop   r8\n"
     " pop   r7\n"
     " pop   r6\n"
     " pop   r5\n"
     " pop   r4\n"
     " reti\n"                        /* go ... */
     );
    }
```

Der Stackpointer wird verschoben, die im Interrupt geretteten CPU-Register werden zurückgeholt und mit *reti* geht es zurück zum "alten" Task. Man beachte, dass die zurückgeholten Regisiterinhalte nicht mit denen gleichzusetzen sind, die beim Start der Interruptroutine gesichert wurden! Um keine Unklarheiten aufkommen zu lassen, die gesicherten Register gehören zum unterbrochenen Task, die geretten Register zum nun aktiven Task.
Der Nebel lichtet sich. Multitasking ist offenbar doch kein Hexenwerk. Stackpointer verbiegen, Stack "faken" und in den Interrupts lustig den Stackinhalt verwürfeln, was wüscht sich das Programmierherz noch mehr?
Wären da nicht die Kleingeister und Erbsenzähler, die es immer ganz genau wissen wollen. Wozu ist der Nulltask den eigentlich nütze? Na, der läuft, wenn sonst nichts mehr anderes

aktiv ist. Richtig, das ist aber nur die halbe Wahrheit. Der Nulltask soll nicht mehr als notwendig kostbare Rechenzeit verbrauchen. Deshalb bekommt er keine vollständige Zeitscheibe. Sobald er gestartet wurde, führt er ein Rescheduling durch. Und das so schnell wie möglich und gleich in Assembler. Der Mechanismus des normalen Scheduling, Zeittick zurückzählen, am TCB manipulieren, etc. entfällt.

```
/*****************************************************************************
* Kurzbeschreibung     : save state and then reschedule
*                        this called by Defer to give up control
* Author               : Jens Altenburg
*****************************************************************************/
void UEXC_SavregsAndResched(void){
asm(
     " mov     SR,%wRegister\n"          /* interrupt stack frame */
     " bis     #8,%wRegister\n"             /* enable GIE */
     " push.w  %wRegister\n"                /* "SR" to stack */
     " push    r4\n"
     " push    r5\n"
     " push    r6\n"
     " push    r7\n"
     " push    r8\n"
     " push    r9\n"
     " push    r10\n"
     " push    r11\n"
     " push    r12\n"
     " push    r13\n"
     " push    r14\n"
     " push    r15\n"
     " mov.w   r15,%wRegister\n"            /* r15 for indexed adressing */
     " mov.w   %uexc_current_sp,r15\n"
     " mov.w   SP,(r15)\n"                  /* save stackpointer */
     " mov.w   %wRegister,r15\n"
     " br      #%UEXC_Schedule\n"           /* Schedule */
     );
    }
```

Wie bereits bekannt, wird der Interrupt-Stack-Frame zusammengebastelt, die CPU Register werden auf dem Stack gesichert und dann schlägt der Assembler-Profi endlich zu. Über einen registerindizierten Speicherzugriff wird der aktuelle Stackpointer gesichert und in den normalen Scheduler verzweigt.

Auf diese Weise kommt der Nulltask genauso zyklisch an die Reihe wie alle User-Tasks, benötigt aber nur geringe CPU-Zeit. So einfach geben sich aber die Zweifler nicht zufrieden. Da der Nulltask eigentlich nichts tut, brauchte man ihn ja auch gar nicht erst zu starten, argumentieren sie.

Doch der Nulltask hat eine Aufgabe. Er ist da. Das ist alles. Dies mag im ersten Augenblick verwundern. Bedeutsam wird es dann, wenn man sich überlegt, was passiert, wenn alle User-Tasks terminieren würden. Es würde nichts mehr laufen, auch kein Multitasking.

Das ist aber nicht gewünscht. Denkbar wären zum Beispiel Interrupt initierte Anwender-Tasks. Eine Tastenbetätigung löst eine Berechnung aus, zeigt das Ergebnis auf einem Display und verschwindet wieder. Ein äußerst reales Szenario.

Mit normalen Mittel programmiert, müsste ein Zustandsautomat immer aktiv bleiben, z.B. alle 100 ms laufen. Wenn eine Tastenbetätigung erkannt wird, schaltet er dann weiter, berechnet, zeigt an und geht in seinen Grundzustand zurück.

Als Task angelegt, könnte dieser im Tastaturinterrupt gestartet werden und wenn er fertig ist verschwindet er wieder. Es ist durchaus erlaubt, dass sich mehrer Tasks einen gemeinsamen Stackbereich teilen können. Sie dürfen nur nicht gleichzeitig darauf zugreifen.
Die Durchsicht der eingangs aufgestellten Wunschliste an das Multitasking-System hat nur noch einen offenen Punkt, zeitkritische Funktionen. Hier passt UEXC. Das Zeitverhalten ist nicht präzise genug vorhersehbar. Insbesondere wenn die Zahl der aktiven Tasks schwanken kann.
Ein Ausweg ist die Nutzung eines hochprioren Timer-Interrupts in Verbindung mit der bekannten Funktionspointer-Liste. Gelingt es, schnelle übersichtliche Funktionen zu schaffen, können diese nach der Art eines kooperativen Multitasking zeitgenau ausgeführt werden.

```
/* Liste mit  Echtzeitfunktionen  */

const  CallBackFunc  TimerA0[]  = {
                          vFunc 1             /* Applikation 1  */
                       ,   n10msTick          /* Zeittakt   */
                       , vFunc2               /* Applikation 2  */
                       , 0x04                 /* ...  */
                       ,   vMainTimer         /* Timereinheit * /
                       , 99                   /* Aufruf pro Sekunde   */
                       };
/* End of List  */
```

Die Anwendung ist denkbar einfach, die Applikationsfunktion wird in die Liste eingetragen und es wird angegeben, zu welchen Zeitpunkten (Timer-Ticks) der Aufruf erfolgen soll. Die Timer-Ticks werden zurückgezählt und bei Nulldurchlauf die zugehörige Funktion gestartet. Durch dieses Vorgehen ist jetzt auch der Zeit-Jitter genau definierbar. Da alle n Ticks die Funktion gestartet wird, müssen die Laufzeiten der zum gleichen Zeitpunkt aktivierten Funktionen Berücksichtigung finden. Kritisch wird es an solchen Stellen, wo mehrere (oder alle) Funktionen aus der Liste "dran" sind. Das lässt sich minimieren, wenn es möglich ist, die einzelnen Zeitspannen so zu wählen, dass sie einen möglichst großen gemeinsamen Nenner bilden.

Beispielapplikation

In der Datei ct04.zip steht eine Beispielapplikation für das beschriebene Multitasking-System zum Download bereit. Wer die Testschaltung für die "Minilichtorgel" aus dem ersten Teile der Artikelserie "Elektronische Zwerge" [2] aufbaute, kann mit dieser Hardware gleich testen.
Das Beispielprogramm besteht aus drei Tasks, *vTask0(), vTask1(), vTask2()* . Jeder Task kontrolliert eine Leuchtdiode, diese sollen mit unterschiedlicher Geschwindigkeit blinken. Nach der Initialisierung sind zunächst nur *vTask0()* und *vTask1()* aktiv. Nach Ablauf der ersten Wartezeit in *vTask1(),* startet dieser einen neuen Task, *vTask2()*. Diese Funktion ist im Gegensatz zu den beiden anderen nicht als "endless loop" angelegt. Damit terminiert *vTask2()* nach Absolvieren seiner Aufgabe selbstständig. Sobald das Programm läuft, blinken die Leds an den Ports 2.0 und 2.1 regelmäßig. In festen Abständen wird dazu ein weiteres Blinksignal an Port 2.2 aktiviert.

Dieses einfache System kann nun nach eigenen Vorstellungen erweitert werden. Die beiden wichtigsten zu beachtenden Parameter stehen im Header "uexc.h". Das ist zum ersten die Anzahl der maximal möglichen Tasks (#define NUM_TASKS 4). Der andere Parameter definiert die minimale erforderliche Speichergröße für einen Task (#define UEXC_MIN_STACK_SIZE (100)). Besonders diesem Parameter ist einige Aufmerksamkeit zu schenken. Wählt man ihn zu groß, wird Speicher belegt der womöglich woanders gebraucht wird. Ist der Wert zu klein arbeitet das System instabil. Sehr ärgerlich ist die Tatsache, dass ein zu kleiner Wert nicht unbedingt einen sofortigen Absturz des Mikrocontroller zur Folge hat, sondern sich u.U. in schwer faßbaren Effekten manifestiert. Im Beispiel hat die Verminderung der Speichergröße auf 59 Byte die seltsame Folge, die Zeitberechnung von *vTask1()* über den Haufen zu werfen. Wie kommt das?
Die Speicherbereiche der einzelnen Tasks im Beispiel folgen aufeinander. Während des Task-Switchs wird fleißig am Stack manipuliert. Zu diesem Zeitpunkt prüft niemand ob der zugewiesene Speicherbereich verlassen wird. Läuft deswegen der Stackpointer in den Speicher eines anderen Tasks hinein, werden dort Werte überschrieben.
Ein exakte Berechnung der richtigen Speichergröße für den einzelnen Task ist kompliziert. Der Speicher muß so groß gewählt werden, dass die lokalen Variablen des Tasks Platz finden, eventuell mögliche Interrupts brauchen auch Speicher. Und nicht zuletzt werden beim Task-Switch auch die Prozessorregister R4...R15 in diesem Bereich gesichert. Hier sollte man also eher etwas großzügier dimensionieren.

Literaturverzeichnis

[1] Man, Richard; Willrich, Christina: "A Minimalist Multitasking Executive" Circuit Cellar Issue 101, 12/1998

[2] Altenburg, Jens; "Elektronische Zwerge - Einführung in die Mikrocontroller-Praxis" c't Magazin für Computertechnik, 24/2003