OS-Funktionalität ohne OS für das IoT

Multicore-Anwendungen und komplexe Betriebssysteme sind heute das Maß aller Dinge. Selbst Mobiltelefone können mit leistungsstarken Octa-Core-Prozessoren und Taktraten mit mehr als 1,5 GHz punkten. Ist das wirklich das untere Ende der eingebetteten Systeme? Sicherlich nicht – 8-bit-Prozessoren oder kleine »32-biter« mit weniger als 100 MHz Taktrate haben immer noch eine Existenzberechtigung. Low-Power ist eben nur mit geringen Ressourcen zu erreichen.

PROF. DR. JÖRG WOLLERT

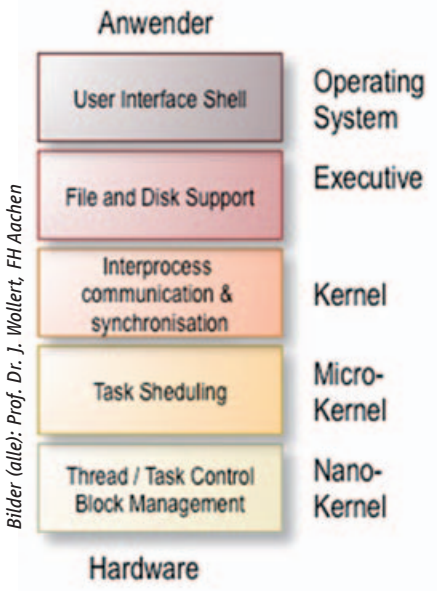
Rund um Cyber-Physical-Systems und Industrie 4.0 spielen Eingebettete Systeme eine immer größere Rolle. Es sind nicht normale Computer und Laptops, die als intelligente Geräte allgegenwärtig sind: Viel wichtiger werden kleine, intelligente Systeme, die mit der Umwelt Daten austauschen können. Diese sogenannten IoT-Devices (IoT = Internet of Things, Internet der Dinge) sind in der Regel minimale Mikrocontroller, die – energieautark oder batteriegespeist – unauffällig vor sich hin werkeln und einen spezifischen Nutzen haben. Es wird schnell offensichtlich, wo die Problematik steckt. Die Leistungsaufnahme eines Prozessors hängt quadratisch von der Spannung und linear vom Takt ab – ein klares Plädoyer für niedrige Core-Spannungen und niedrige Frequenzen: Embedded-Lösungen mit wenigen 10 kHz Taktfrequenz und etwa 1,8 V Core-Spannung, wie bei einem Cortex-M-Prozessor.

Eine weitere spannende Herausforderung ist die Software. Durch die Betriebssysteme der heute üblichen Computer von Windows über Linux bis Mac OSX und bei den Mobiles von Android oder iOS ist man heute sehr verwöhnt. Als Betriebssystem wird landläufig eine Ansammlung von Software verstanden, die für alle Bereiche des Lebens nützt. Heute sind dafür Gigabyte an Speicher notwendig, um die Softwaresammlungen zum Laufen zu bekommen.

Doch braucht man das alles wirklich, um einfache IoT-Anwendungen zu realisieren? Sicherlich nicht. Embedded-Systeme brauchen keine großen Betriebssysteme, sie brauchen nicht viel Prozessorleistung und Speicher und vor allem kein kompliziertes Ressourcen fressendes Betriebssystem.

Grundlagen Betriebssysteme

Das Betriebssystem gibt es ohnehin nicht. Was wir heute alles an Anwendungsprogrammen und grafischen Benutzerschnittstellen bekommen, ist mehr, als man erwarten kann. Das ist auch ein Grund, warum Windows und Co. ständig damit rechnen müssen, dass die eine oder andere Funktion abgeschaltet werden muss. Was immer auch die Praxis sagt, in der Betriebssystemtheorie unterscheidet man zwischen unterschiedlichen Größen und Komplexitätsstufen von Betriebssystemen.



Bilder (alle): Prof. Dr. J. Wollert, FH Aachen

Bild 1: Ebenen von Betriebssystemen.

In *Bild 1* sind die unterschiedlichen Ebenen von Betriebssystemen skizziert. Auf der untersten Entwicklungsstufe sind sogenannte Nanokernel. Sie unterstützen ein einfaches Thread- bzw. Task-Control-Management. Hier sind keine komplexen und aufwendigen Scheduling-Algorithmen implementiert. Alleine die Bereitstellung essenzieller Verwaltungsfunktionen für die Thread-Bearbeitung zählt. Die in der nachfolgenden Ausführung dargestellten Protothreads können als Nanokernel interpretiert werden. Wird ein Scheduling-Verfahren für das Aufrufen der Tasks bereitgestellt, spricht man von einem »Microkernel«. Unterschiedliche Algorithmen sind denkbar, um eine priorisierte Task-Reihenfolge festzulegen. Die Art der Verfahren entscheidet über die Reaktionszeit der Anwendung. Werden darüber hinaus Mechanismen für die Interprozess-Kommunikation und die Synchronisation von Tasks festgelegt, hat man einen klassischen Kernel. Durch die Implementierung einer standardisierter I/O-Steuerung und der Bereitstellung eines Dateisystems wird ein Kernel zu einem »Executive«. In diesem Sinn ist z. B. der Linux-Kernel ein Executive und kein Kernel. Erst mit einer standardisierten Benutzerschnittstelle, wie einem Kommandointerpreter in einer Shell, wird der Executive zum Betriebssystem.

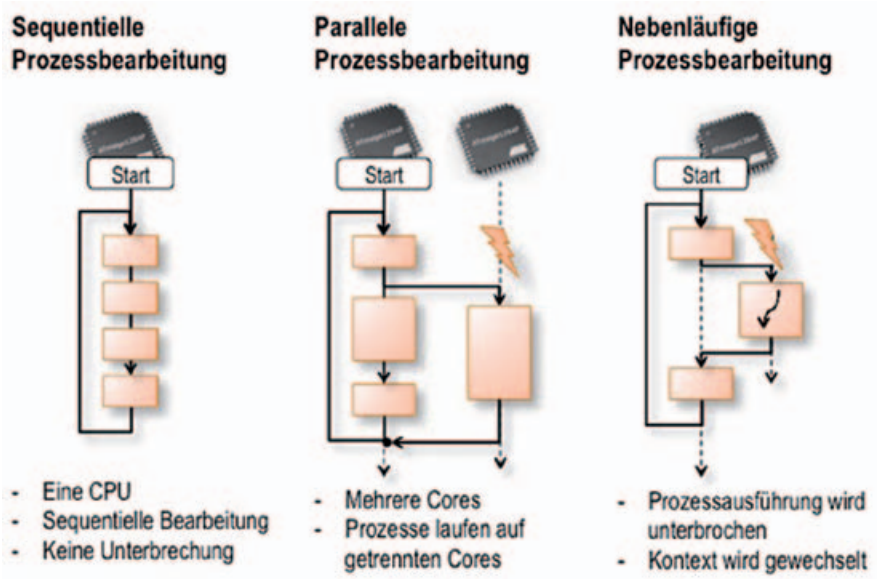


Bild 2: Prozessbearbeitung kann in unterschiedlichen Varianten ablaufen.

Heute hat sich der Standard POSIX 1003 durchgesetzt. In den USA ist beispielsweise eine POSIX-kompatible Shell Voraussetzung für die Zulassung zur behördlichen Nutzung. Ein Vorteil für alle, die nicht nur die Maus schubsen, sondern auf der Kommandozeile arbeiten. Spätestens wenn man bei POSIX-kompatiblen Betriebssystemen angekommen, ist wird es komplex. Windows, Mac OSX und Linux sind hier die typischen Vertreter, wobei sich Linux im Sinn von Embedded Systemen klein skalieren lässt. Wobei das »relativ klein« ist – im Vergleich zu Windows und Mac OSX – mit immerhin noch kaum unter 1 MB an Code. Soll es wirklich klein werden, dann ist das eher ein Fall für einen Nanokernel.

Kleiner Quellcode

Ein Meister kleiner Quellcodes ist Adam Dunkels. Während seiner Zeit am SICS, dem Swedish Institue of Computer Science in Stockholm, hat er sich maßgeblich mit speicherbegrenzten Eingebetteten Systemen beschäftigt. In seiner Dissertation »Programming Memory-Constrained Networked Embedded Systems« [1] publiziert er als kumulative Dissertation einige wesentliche vorhergehende Arbeiten, die auch heute noch eine große Bedeutung haben. Drei Themengebiete sind hierbei im Besonderen hervorzuheben:

- Protothreads sind eine minimale Realisierung eines Nanokernels, der kleinste 8-bit-Prozessoren multithreadding-fähig macht [2].
- lwIP (lightweight IP) ist eine minimale Implementierung des TCP/IP-Stacks mit allen relevanten Protokollen. Er ist Ressourcen schonend und hat einen guten Datendurchsatz. Viele kommerziellen Embedded Systeme nutzen für die IP-Kommunikation diesen Stack.
- Contiki ist ein Open-Source-Betriebssystem für Low-Cost-, Low-Power-Mikrocontroller für IoT-Anwendungen [3].

Dieser Artikel wird sich mit den Prinzipien und der Umsetzung der Protothreads beschäftigen, da so minimale Betriebssystemfunktionalität auch in kleinsten Embedded Systemen umgesetzt werden kann.

Grundfunktion neben läufiger Programme

Embedded-Programmierung könnte so einfach sein, wenn alles streng sequentiell und zyklisch ablaufen würde. Doch die Welt ist komplizierter. Je aufwendiger ein Prozessablauf ist, beispielsweise durch viele Kommunikationsschnittstellen, Displays und Eingabeeinheiten, desto komplizierter wird auch der Softwarecode. Was keiner haben möchte, ist dann zwangsläufig – ein undurchschaubarer Bandwurmcode. Eine Modularisierung und Trennung des Softwarecodes nach Aufgaben ist eine zwingend notwendige Konsequenz. Verschiedene Möglichkeiten stehen hierbei zur Verfügung:

- Entweder man entwickelt eine vollständige Zustandsmaschine für den gesamten Prozess, was in der Regel sehr komplex ist, oder
- man parallelisiert die Aufgaben, wie sie tatsächlich vorkommen, und versucht die Tasks quasi parallel ablaufen zu lassen. Nutzt man echtes Threadding mit Interrupt-Routinen, dann wird eine erhebliche Menge an Speicher benötigt.
- Alternative drei ist die konsequente Implementierung mit leichtgewichtigen Protothreads.

Welche Wahl man auch trifft, die Entscheidung ist nicht trivial. Eine komplexe Zustandsmaschine erfordert eine hohe Disziplin in der Implementierung. Der Prozessablauf muss in Automaten eingebettet werden, was in der Regel eine recht komplexe Aufgabe ist. Switch-Case-Anweisungen helfen hier, sind aber nicht immer einfach zu durchschauen.

Die Kapselung in Tasks macht dann insbesondere Sinn, wenn die zu bearbeitenden Aufgaben tatsächlich isoliert werden können. Das ist in vielen Fällen möglich. Dann unterscheiden sich die objektorientierte und die taskbasierte Vorgehensweisen nicht wesentlich. Betrachtet man zunächst die sequentielle Abarbeitung, ist hier eine Single-Core-Prozessorlösung weder störend noch

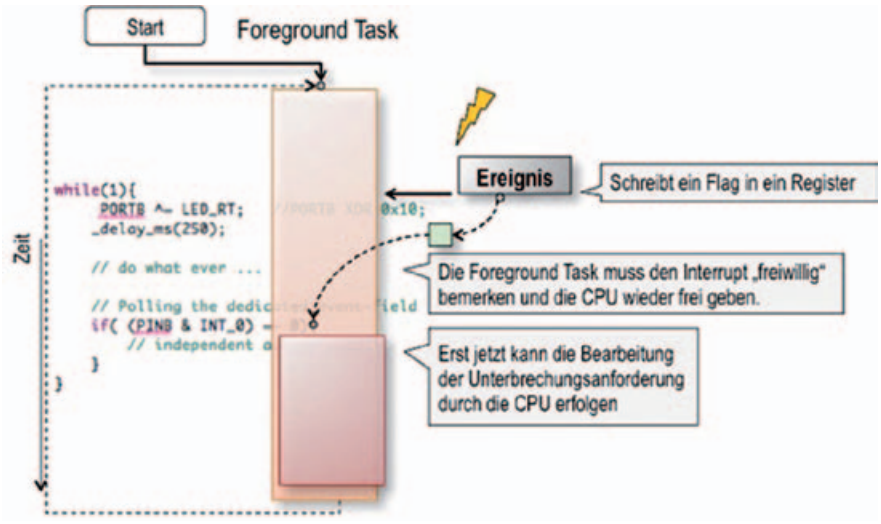


Bild 3: Nicht preemtives Multitasking durch Polling.

```
/* Interrupt Vectors */
/* Interrupt Vector 0 is the reset vector. */
#define INT0_vect            _VECTOR(1)   /* External Interrupt Request 0 */
#define INT1_vect            _VECTOR(2)   /* External Interrupt Request 1 */
#define PCINT0_vect          _VECTOR(3)   /* Pin Change Interrupt Request 0 */
#define PCINT1_vect          _VECTOR(4)   /* Pin Change Interrupt Request 0 */
#define PCINT2_vect          _VECTOR(5)   /* Pin Change Interrupt Request 1 */
#define WDT_vect             _VECTOR(6)   /* Watchdog Time-out Interrupt */
#define TIMER2_COMPA_vect    _VECTOR(7)   /* Timer/Counter2 Compare Match A */
#define TIMER2_COMPB_vect    _VECTOR(8)   /* Timer/Counter2 Compare Match A */
#define TIMER2_OVF_vect      _VECTOR(9)   /* Timer/Counter2 Overflow */
#define TIMER1_CAPT_vect     _VECTOR(10)  /* Timer/Counter1 Capture Event */
#define TIMER1_COMPA_vect    _VECTOR(11)  /* Timer/Counter1 Compare Match A */
#define TIMER1_COMPB_vect    _VECTOR(12)  /* Timer/Counter1 Compare Match B */
#define TIMER1_OVF_vect      _VECTOR(13)  /* Timer/Counter1 Overflow */
#define TIMER0_COMPA_vect    _VECTOR(14)  /* TimerCounter0 Compare Match A */
#define TIMER0_COMPB_vect    _VECTOR(15)  /* TimerCounter0 Compare Match B */
#define TIMER0_OVF_vect      _VECTOR(16)  /* Timer/Couner0 Overflow */
#define SPI_STC_vect         _VECTOR(17)  /* SPI Serial Transfer Complete */
#define USART_RX_vect        _VECTOR(18)  /* USART Rx Complete */
#define USART_UDRE_vect      _VECTOR(19)  /* USART, Data Register Empty */
#define USART_TX_vect        _VECTOR(20)  /* USART Tx Complete */
#define ADC_vect             _VECTOR(21)  /* ADC Conversion Complete */
#define EE_READY_vect        _VECTOR(22)  /* EEPROM Ready */
#define ANALOG_COMP_vect     _VECTOR(23)  /* Analog Comparator */
#define TWI_vect             _VECTOR(24)  /* Two-wire Serial Interface */
#define SPM_READY_vect       _VECTOR(25)  /* Store Program Memory Read */

#define _VECTORS_SIZE (26 * 4)
```

Bild 4: Interrupt-Vektoren ermöglichen die Reaktion auf Systemereignisse.

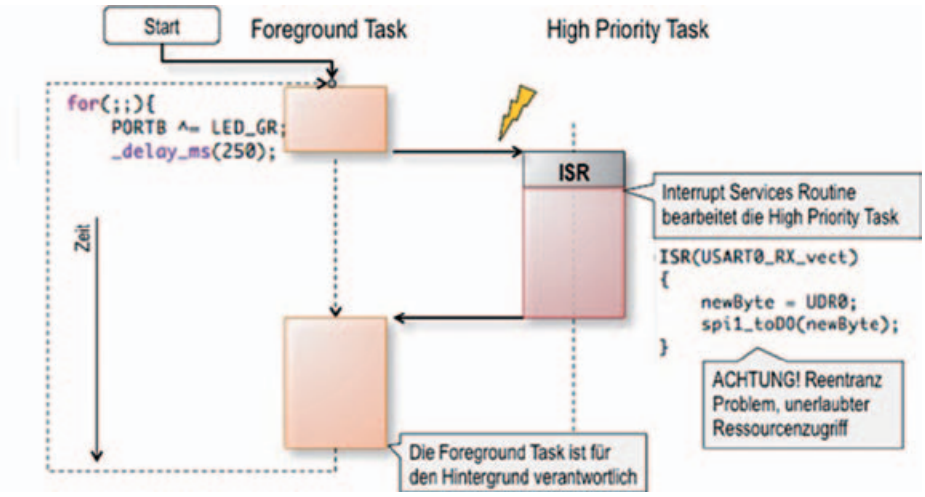


Bild 5: Preemtives Multitasking durch Unterbrecherbetrieb.

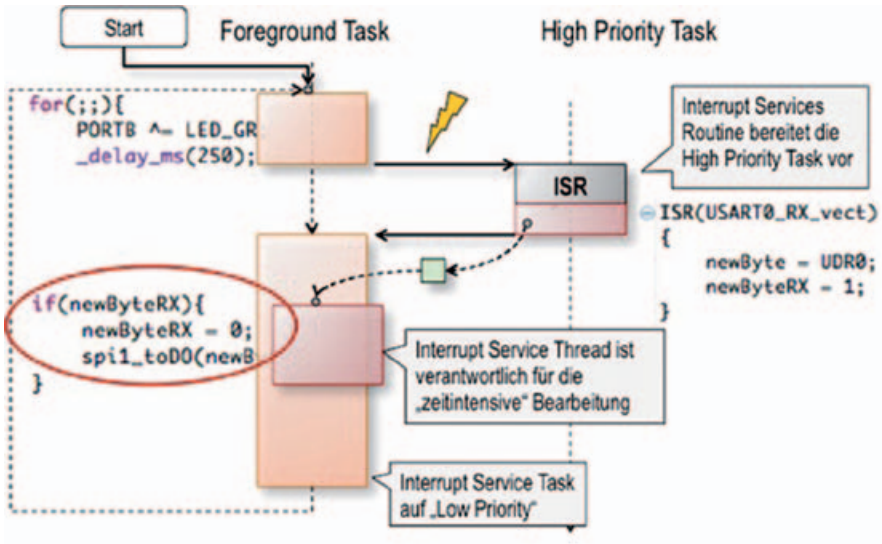


Bild 6: Multitasking und Trennung von ISR und IST sorgen für einen Betrieb mit kurzen Reaktionszeiten und minimalen Blockaden.

hilfreich. Anders wird es bei sogenannten nebenläufigen Programmen.

Als »nebenläufig« bezeichnet man die Fähigkeit eines Systems, tatsächlich Prozesse parallel (zeitgleich) abarbeiten zu können. Das ist keinesfalls selbstverständlich. Echte Parallelität bleibt Multi-Core-Prozessoren vorbehalten. In Single-Core-Systemen, was bei kleinen Mikrocontrollern üblich ist, kann das nicht realisiert werden. Hier spricht man dann von »quasi-parallelem« oder »quasi-nebenläufigem« Betrieb. Durch den Scheduler koordiniert werden die anstehenden Prozesse in der dafür vorgesehenen Reihenfolge bearbeitet. *Bild 2* macht diese Prozessbearbeitung deutlich – nur bei Multi- oder Many-Core-Prozessoren ist eine parallele Bearbeitung der anstehenden Tasks möglich. Andernfalls ist das verschränkte Arbeiten notwendig.

Unterbrecherbetrieb für nebenläufige Programme

Eine ereignisgesteuerte Verarbeitung wird durch externe Unterbrecheranforderungen (Interrupts), Timer-Interrupts oder Traps realisiert. Durch das externe Ereignis wird die Vordergrundtask unterbrochen und eine High-Priority-Task kann unmittelbar angestoßen werden.

Bild 3 zeigt die Reaktion auf ein Eventflag. Die Bearbeitung der Unterbrecheranforderung hängt von dem freiwilligen Abgeben der Rechenzeit vom Vordergrundprozess und seinem Laufzeitverhalten ab. Das ist nicht deterministisch und erfüllt bestenfalls Best-Effort-Annahmen.

Soll eine unmittelbare Reaktion auf eine Unterbrecheranforderung erfolgen, dann kann dieses durch eine Hardware-Unterbrecheranforderung geschehen. Für den Fall eines Atmel-Prozessors sind für die unterschiedlichsten Systemereignisse Interruptvektoren für die unterschiedlichsten Systemereignisse definiert, wie es in *Bild 4* deutlich wird.

Selbst bei Low-end-8-bit-Mikrocontrollern wie dem Atmega 382p, wie er beispielsweise in der Arduino-Uno-Plattform eingesetzt wird, dauert ein Taskwechsel weniger als 10 µs. Damit können mit dem Interrupt-Betrieb auch anspruchsvolle Echtzeitaufgaben gelöst werden.

```
struct pt { unsigned short lc; };

#define PT_INIT(pt)             pt->lc = 0
#define PT_BEGIN(pt)            switch(pt->lc) { case 0:
#define PT_EXIT(pt)             pt->lc = 0; return 2
#define PT_WAIT_UNTIL(pt, c)    pt->lc = __LINE__; case __LINE__: \
                                if(!(c)) return 0
#define PT_END(pt)              } pt->lc = 0; return 1
```

Bild 7: Protothread Makro-Definition.

```
int a_protothread(struct pt *pt) {

PT_BEGIN(pt);

  PT_WAIT_UNTIL(pt, condition1);

  if(something) {

    PT_WAIT_UNTIL(pt, condition2);

  }
  PT_END(pt);
}
```

Bild 8: Protothread Code vs. Expandierte Makros.

Herausfordernd wird es, wenn die Dauer der Interrupt-Service-Routine im Verhältnis zur Vordergrundtask lange dauert. Einige 10 µs können durchaus problematisch werden, wenn zum Beispiel an einem Hardware-Eingang schnelle Signale gezählt werden müssen. In diesem Fall ist es sinnvoll, die Unterbrecherbehandlung zweistufig zu realisieren. Das wird auch bei vielen »großen« Betriebssystemen so gemacht. In *Bild 5* wird dieselbe Aufgabenstellung wie in *Bild 6* gelöst, nur mit dem Unterschied, dass nach *Bild 6* nur die zeitkritischen Aktivitäten in einer ISR (Interrupt Service Routine) realisiert werden und die zeitunkritischen Anwendungsteile in einen IST (Interrupt Service Thread) ausgelagert werden.

Diese Beispiele geben nur einen kleinen Einblick in die prinzipielle Herangehensweise bei der Programmierung von Eingebetteten Systemen. Schließlich soll nur ein wenig Verständnis für die unterschiedlichen Auswirkungen einer Task-basierten Programmierung gegeben werden. Hierbei wird deutlich, dass die Verwendung von Threads zu einer besser strukturierten und verständlicheren Programmabarbeitung – auch komplexer Sachverhalte – führt.

Was sind Protothreads?

Sogenannte Deep-Embedded-Systems verzichten aus Ressourcen- und Performance-Gründen häufig auf ein Betriebssystem. Möchte man dennoch nicht auf den Komfort einer Thread-Behandlung verzichten, bieten sich leichtgewichtige Protothreads an. Entgegen konventioneller Threads, die einen eigenen Stack benötigen, wird bei einem Protothread nur eine speichersparende Kontrollstruktur und ein gemeinsamer Stack für alle Threads gemeinsam verwendet. Adam Dunkels hat eine sehr schlanke API in ANSI C realisiert [4], die quasi auf allen Embedded-Systemen prinzipiell lauffähig ist, und damit eine Art universellen Nanokernel geschaffen. Protothreads werden in der Regel nicht preemtiv verwendet, haben eine FCFS-Bearbeitung (First Come First Serve) und können mit einem geringen Overhead (2 Byte pro Thread) auch auf minimalen Systemen eine virtuelle Parallelität realisieren. Die Kontrollstruktur eines Protothreads ist erschreckend klein und basiert auf einer Zustandsmaschine, die in ein paar Makros verborgen ist (*Bild 7*).

In *Bild 8* werden für die einfache Definition eines Protothreads die Makros expandiert. Und dann in die übliche Form umformatiert – schon sieht man die threadspezifische Statusmaschine.

Glücklicherweise braucht man sich nur wenig Gedanken über die Realisierung machen, da Adam Dunkels' Makros bewährte Technik sind und man sich auf eine ordent-

Literatur

[1] Dunkels, Adam: Programming Memory-Constrained Network Embedded Systems, Swedish Institute of Computer Sciene Doctoral Thesis, SICS Dissertation Series 47, February 2007.

[2] Dunkels, Adam: Protothreads, Simplifying event-driven Programming of memory constrained embedded systems. Proceedings oft he 4th International Conference on embedded networked Sensor Systems (ACMSenSys), Colorado, USA, 11.2006.

[3] Wollert, Jörg: Vorlesung »Embedded Systems« und »Systems Engineering«, FH Aachen.

[4] Wollert, Jörg: Rapid Application Development. DESIGN&ELEKTRONIK 2016, Heft 04, Seite 8ff.

[5] Clement, Jan: Example Protothread for Arduino, arduino.cc.

[6] http://www.contiki-os.org/


```
int a_protothread(struct pt *pt) {
  switch(pt->lc) {

  case 0:  pt->lc = 5;
  case 5:  if(!condition1)
              return 0;
           if(something) {
              pt->lc = 10;
           case 10: if(!condition2)
              return 0;
           }
  }
 return 1;
}
```

```
int a_protothread(struct pt *pt)
{
  switch(pt->lc) { case 0:

  pt->lc = 5; case 5:
  if(!condition1) return 0;
  if(something) {

    pt->lc = 10; case 10:
    if(!condition2) return 0;
  }
  } return 1;
}
```

Bild 9: Expandierte und umformatierte »Thread«-Zustandsmaschine.

```
#include <pt.h>   // include protothread library

#define LEDPIN 13  // LEDPIN is a constant

static struct pt pt1, pt2; // each protothread needs one of these

void setup() {
  pinMode(LEDPIN, OUTPUT); // LED init
  PT_INIT(&pt1);  // initialise the two
  PT_INIT(&pt2);  // protothread variables
}

void toggleLED() {
  boolean ledstate = digitalRead(LEDPIN); // get LED state
  ledstate ^= 1;   // toggle LED state using xor
  digitalWrite(LEDPIN, ledstate); // write inversed state back
}

/* This function toggles the LED after 'interval' ms passed */
static int protothread1(struct pt *pt, int interval) {
  static unsigned long timestamp = 0;
  PT_BEGIN(pt);
  while(1) { // never stop
    /* each time the function is called the second boolean
    *  argument "millis() - timestamp > interval" is re-evaluated
    *  and if false the function exits after that. */
    PT_WAIT_UNTIL(pt, millis() - timestamp > interval );
    timestamp = millis(); // take a new timestamp
    toggleLED();
  }
  PT_END(pt);
}
/* exactly the same as the protothread1 function */
static int protothread2(struct pt *pt, int interval) {
  static unsigned long timestamp = 0;
  PT_BEGIN(pt);
  while(1) {
    PT_WAIT_UNTIL(pt, millis() - timestamp > interval );
    timestamp = millis();
    toggleLED();
  }
  PT_END(pt);
}

void loop() {
  protothread1(&pt1, 900); // schedule the two protothreads
  protothread2(&pt2, 1000); // by calling them infinitely
}
```

Bild 10: So wird es einfach – zwei unabhängige Threads, durch die Loop schedult.

liche Funktion verlassen kann. Darüber hinaus sind Protothreads weit verbreitet und die Realisierung einer eigenen Anwendung reduziert sich auf die Anwendung der Protothread-Bibliothek. Das wird an einem praktischen Beispiel nachfolgend gezeigt.

Protothreads auf Arduino

Ein Arduino ist alles andere als ein Spielzeug [5]. Die Systematik ist übertragbar für fast alle anderen Mikrocontroller. Ein großer Vorteil der Arduino-Technologie ist die unglaubliche Übertragbarkeit auf unterschiedlichste Prozessorplattformen bei gleichzeitig minimaler Einstiegshürde.

Um einem Arduino-Multithreadding mit Protothreads beizubringen, reicht es aus, die Protothread-Bibliothek <pt.h> einzubinden. Hierbei handelt es sich um eine Standardportierung der Bibliothek von Dunkels. Diese enthält keinen Arduino-spezifischen Code und ist mit allen GNU-C-Compilern lauffähig.

In dem folgenden Codebeispiel (*Bild 9*) von Jan Clement [6] wird durch zwei Threads eine Leuchtdiode getoggelt, sodass ein quasi zufälliges Blinkmuster entsteht. Es ist das Standardbeispiel aus der Arduino-Protothread-Bibliothek.

Das Vorgehen in dem Programmcode ist offensichtlich. Es reicht aus, die beiden statischen Strukturen für die Thread-Verwaltung zu deklarieren und in der setup()-Methode die beiden Threads mit PT_INIT(&pt) zu initialisieren. Die &pt referenziert auf die Adresse der jeweiligen Verwaltungsstruktur.

Der Protothread selber ist nichts anderes als eine statische Funktion, bei der die Makros für den Start PT_BEGIN(pt) und das Ende PT_END(pt), sowie für die Schedul-Bedingung PT_WAIT_UNTIL(pt,Bedingung) implementiert werden. Jetzt muss nur noch der Thread in der Arduino-Loop-Methode aufgerufen werden. Fertig ist das Multithreadding auf einem 8-bit-Mikrocontroller. Der Arduino-Sketch wird nur verwendet, um in der Setup-Methode die Threads zu initialisieren und in der Loop-Methode die Threads zu schedulen. Der restliche Programmablauf wird den Protothreads überlassen.

Zusammenfassung

Low-end-Embedded-Plattformen stellen eine hohe Anforderung an die Entscheidungsfähigkeit der Entwickler: Zum nächstgrößeren Prozessor greifen und ein Betriebssystem benutzen oder doch besser auf das Betriebssystem verzichten? Die Frage lässt sich einfach beantworten: Einen Nanokernel verwenden und das Embedded-System mit einem minimalen Footprint realisieren. Adam Dunkels Protothreads sind eine ausgesprochen effiziente Art, Mikrocontroller gut strukturiert zu programmieren und gleichzeitig auf Overhead zu verzichten. So können auch mit kleinen 8-bit-Prozessoren anspruchsvolle Aufgaben in einem Thread-Modell bearbeitet werden. Man muss also nicht immer das Rad neu erfinden oder gleich auf Linux-basierte Systeme zurückgreifen. (fr)

PROF DR.-ING. JÖRG WOLLERT

Professor für Eingebettete Systeme und Mechatronik an der FH-Aachen. Nach etlichen Jahren in der Industrie ist er seit 1999 als Professor an verschiedenen Hochschulen tätig. Im Rahmen seiner Tätigkeit für diverse Einrichtungen berät er Unternehmen im Bereich verteilter Automatisierungssysteme, eingebetteter Systeme, insbesondere mit Funktechnologie und Industrie 4.0.