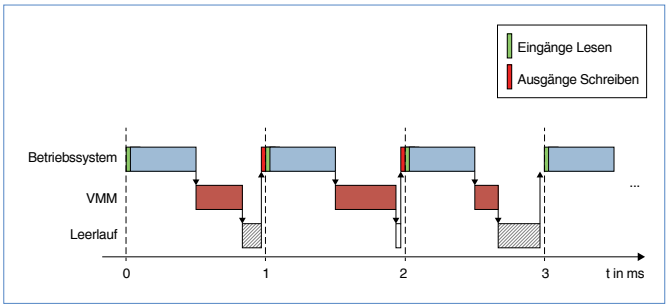
Applikationsspezifische Programmierung mit NanoJEasy 2.0

Neue Technologie für die zweite Generation der integrierten Programmiersprache

Für die zweite Generation unserer integrierten Programmiersprache haben wir die Technologie grundlegend überarbeitet, um Echtzeitfähigkeit, und eine höhere Ausführungsgeschwindigkeit zu erreichen. Während die Programme in der Version 1 von NanoJ als Bytecode in einer virtuellen Maschine parallel zur Motorregelung abliefen und deshalb keinem strikt deterministischen Timing unterlagen, ist in der neuen Version nun ein deterministischer 1 ms Zyklus implementiert. Dieser wird mittels „kooperativem Multitasking“ sichergestellt:



In jedem 1 ms-Zyklus werden zuerst die Daten aus dem Objektverzeichnis ausgelesen, das als zentrale Datenbasis alle Einstellungen und Zustandswerte des Reglers enthält. Anschließend läuft das „Betriebssystem“, also z.B. die übergeordneten Funktionen des Motorreglers und die Feldbuskommunikation.
Der eigentliche Regler läuft deutlich schneller mit 32KHz. Nachdem das Operating System alle zyklisch notwendigen Operationen durchgeführt hat, übergibt es die Ausführung an das Benutzerprogramm im VMM (Virtual Maschine Monitor).

Das Benutzerprogramm kann nun Werte aus dem Object Dictionary modifizieren, Berechnungen durchführen, usw., muss allerdings die Kontrolle vor Ablauf der 1 ms wieder an das Betriebssystem zurückgeben. Deshalb heißt es auch kooperatives Multitasking. Vor Ablauf des Zyklus werden dann die Ausgabewerte des User-Programms wieder in das Objektverzeichnis zurückgeschrieben und damit im darauffolgenden Zyklus vom Motorregler verarbeitet. Sie können z.B. auch wieder per Feldbus ausgelesen werden.

Ein Sandkasten für das Userprogramm

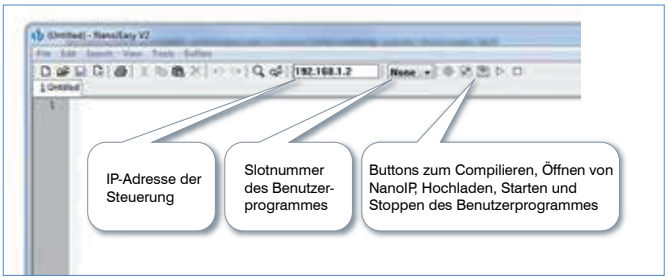
Der VMM stellt eine geschützte Ausführungsumgebung innerhalb der Firmware zur Verfügung. Durch eine sogenannte Sandbox, also die Beschränkung des User-Programms auf einen bestimmten Speicherbereich und bestimmte Systemressourcen, ist sichergestellt, dass das Benutzerprogramm niemals die eigentliche Firmware des Reglers zum Absturz bringen kann.

Auch einer Verzögerung der Regelung durch besonders rechenintensive User-Programme ist vorgesorgt: Wenn das User-Programm nicht kooperativ ist, die Kontrolle also nicht von sich aus wieder vor Ablauf des Zyklus an das Betriebssystem zurückgibt, wird es beendet. Es erscheint eine Fehlermeldung, wobei der Motorregler trotzdem ungestört und ohne Verzögerung weiterläuft.

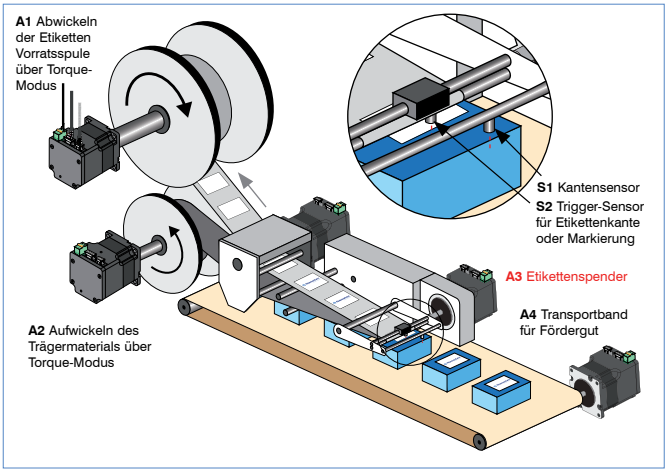
Diese Sandbox erlaubt auch die zweite große technologische Neuerung: Anstatt Bytecode wird nun direkt deutlich schnellerer Maschinencode ausgeführt.

Die Programmierumgebung NanoJEasy

Die Programmierumgebung NanoJEasy ermöglicht die einfache Erstellung von Programmen in NanoJ. Mit dem integrierten Editor mit Syntax-Highlighting können die Programme entwickelt und dann kompiliert werden. Bei Steuerungen mit Ethernet-Schnittstelle wie der N5, kann das Programm auch direkt in die Steuerung übertragen, gestartet und gestoppt werden.



Ein Beispiel: Etikettenspender



Im Folgenden werden wir in zwei Schritten ein einfaches Programm entwerfen, dass die Funktion eines Etikettenspenders abbildet.
Die Aufgaben der anderen Motoren in der Applikation lassen sich sehr einfach umsetzen. Die Motoren zum Auf- und Abwickeln laufen im Drehmomentmodus, das Förderband im Drehzahlmodus, beides ist ohne Programmierung als Standardbetriebsart verfügbar.

Beim Spender dagegen brauchen wir ein kleines Programm. Unser Ziel ist es, den Motor zuerst auf eine konstante Geschwindigkeit zu beschleunigen, die der Förderbandgeschwindigkeit entspricht, und dann beim Erkennen der Etikettenkante eine definierte Strecke (die Etikettenlänge) abzufahren. Um das Beispiel auch einfach nachvollziehbar zu halten, arbeiten wir nur mit einem Trigger-Eingang.

In der realen Applikation müsste zumindest noch ein Starteingang hinzukommen, der den Motor startet. Wenn Sie das Beispiel verstanden haben, sollte es aber kein Problem sein, die Abfrage des Eingangs noch zu ergänzen.

Die erste einfache Version
Das folgende Programm ist unsere erste einfache Version, die zunächst nur den Positioniermodus aktiviert und den Motor startet:

```
1   //Flagposition-Mode
2
3   //mapping of frequently used SDO´s:
4
5   map U16 ControlWord as output 0x6040:00
6   map S16 ProfileVelocity as inout 0x6081:00
7   map S32 TargetPosition as inout 0x607A:00
8
9   #include "wrapper.h"
10
11  //starting the main-routine and settings
12
13  void user()
14  {
15      od_write(0x6060,0x00, 1);                   // Mode of Operation as Profile Position
16      InOut.ProfileVelocity = 200;                // setting the target velocity
17      InOut.TargetPosition = 1000000000;          // setting the target position (just as a limit)
18
19  //boot-up the state-machine
20
21      Out.ControlWord = 0x6;                      // enable voltage
22      yield();
23      Out.ControlWord = 0x7;                      // switched on
24      yield();
25      Out.ControlWord = 0x4F;                     // operation enable + target position "relative"
26      yield();
27      Out.ControlWord = 0x5F;                     // start
28      yield();
29
30      while(true)
31      {
32          yield();
33      }
34  }
```

Mapping von Einträgen des Objektverzeichnisses
In den Zeilen 5-7 erfolgt das Mapping, also die Zuordnung von Variablen im Programm zu Einträgen des Objektverzeichnisses. So besagt z.B. die Zeile 6, dass der Inhalt des Objektes 0x6081 mit jedem Zyklus in die 2 Byte große, vorzeichenbehaftete (S16) Variable „ProfileVelocity“ übernommen werden soll und nach dem Ende des Zyklus wieder zurückgeschrieben wird. Die Objektadressen bzw. Objekte entsprechen hierbei dem CAN-Standard DS402, 6081 ist also die Maximalgeschwindigkeit der Positionierung.
Wird das Mapping statt als „inout“ nur als „input“ oder „output“ deklariert, wird die Variable entsprechend nur am Anfang des Zyklus gelesen bzw. am Ende des Zyklus zurückgeschrieben, wie z.B. das Control-Wort in Zeile 6.
Die Präprozessoranweisung „include „wrapper.h““ in Zeile 9 braucht uns nicht weiter zu interessieren, sie muss als Anweisung für den Compiler in jedem NanoJ-Programm stehen.

Das Hauptprogramm
In Zeile 13 beginnt dann mit der Funktion „user()“ das Hauptprogramm, die in NanoJ der Funktion „main“ in C bzw. Java entspricht und immer als erste Funktion ausgeführt wird. In Zeile 15 begegnet uns die zweite Möglichkeit, neben dem Mapping auf das Objektverzeichnis zuzugreifen, der Befehl „od_write“. Hiermit können im Programmablauf Objekte geändert oder mit „od_read“ gelesen werden, die nur einmal oder selten erforderlich sind und deshalb nicht in jedem Zyklus eingelesen werden müssen. Hier wird also durch Zugriff auf das CAN-Objekt 0x6060, „Modes of Operation“, der Positioniermodus aktiviert.
In den nächsten beiden Zeilen werden die gemappte Geschwindigkeit und eine Zielposition vorgegeben. Durch das Mapping als „inout“ reicht hier die Zuweisung eines Wertes zur Variablen aus, ein expliziter Schreibbefehl ist nicht notwendig. Schließlich wird über das gemappte ControlWord noch die Zustandsmaschine des DS402 auf den Zustand „ready to switch on“ geschaltet.
In der nächsten Zeile folgt dann das erste „yield()“, mit dem unser Programm die Kontrolle zurück an das Betriebssystem übergibt, den 1 ms Zyklus also abschließt. Wie man in den darauffolgenden Zeilen sieht, folgt nach jedem Übergang der Zustandsmaschine ein „yield()“, da jeder Zustand im Regler durchlaufen werden muss. Würde das „yield()“ in Zeile 22 entfallen, würde aus Sicht des Reglers direkt in den Zustand „switched on“ geschaltet. Das ist nach CAN-Standard nicht erlaubt und würde deswegen auch nicht funktionieren.
Nach dem fünften „yield()“ und damit nach 5 ms ist die Steuerung aktiv und der Motor beschleunigt auf die Geschwindigkeit von 200 Schritten/s. Die Einheit ist abhängig von den Einstellungen in den Objekten 0x2060-0x2062, in unserem Fall gehen wir davon aus, dass diese Objekte den Standardeinstellungen entsprechen.
Damit wären wir am Ende unserer ersten Programmversion angelangt. In den letzten Zeilen verhindert eine Endlosschleife, dass das Programm zu Ende kommt, da es in diesem Fall im nächsten Zyklus neu gestartet würde.

Trigger- und Analogeingang
Was uns jetzt noch fehlt, ist die Reaktion auf den Trigger-Eingang, der eine vorher definierte Strecke startet. Unsere erste Programmversion läuft ja (beinahe) endlos mit konstanter Geschwindigkeit, da eine Zielposition gewählt wurde, die faktisch nie erreicht wird. Das wäre eigentlich eine Anwendung für den Drehzahlmodus, den wir hier allerdings nicht verwenden, sondern durch die Vorgabe eines unerreichbaren Positionssollwertes simulieren. Dieser Trick erspart uns im nächsten Schritt die Umschaltung zwischen zwei Betriebsmodi. Es reicht, als Reaktion auf den Eingang die Sollposition neu zu setzen. Zusätzlich soll der Analogeingang verwendet werden, um eine einfache Einstellung der Etikettenlänge zu ermöglichen.

Um eine Änderung am Eingang im Programm zu sehen, müssen wir zuerst einmal das Objekt für die digitalen Eingänge und den Analogeingang mappen (Zeile 8-9):

```
7   map S32 TargetPosition as inout 0x607A:00
8   map U32 Inputs as input 0x60FD:00
9   map S32 ActualPosition as input 0x6064:00
10  map S32 AnalogInput as input 0x6401:01
```

Die Endlosschleife unserer ersten Version erweitern wir nun mit dem Code für die Abfrage des Eingangs und das Setzen der Zielposition:
Durch die „&“ (logisch „und“)-Verknüpfung des Objekts für die digitalen Eingänge in Zeile 37 mit der Bitmaske 0x10000 wird nur der Eingang 1 überwacht. Bei Änderung des Eingangs werden die Zielposition und die Geschwindigkeit geändert.
Die Zielposition wird dabei aus dem Wert des Analogeingangs (0 bis 1023) mit einem Multiplikator 10 berechnet. Wenn der Eingang gesetzt wird, wird die aktuelle Position ausgelesen, die aus dem Analogwert berechnete Zielposition aufaddiert und in Zeile 43 auf absolute Positionierung umgeschaltet. Dadurch ist sichergestellt, dass die Verzögerung von 2 ms, bis die Position nach Zeile 47 in den Regler übernommen ist, die ermittelte Zielposition nicht mehr beeinflusst.
Durch die Schleife am Ende wird gewartet, bis der Eingang nochmals seinen Zustand ändert. Anschließend würde der Motor nun stehenbleiben und mit jedem weiteren Schalten des Eingangs die durch den Analogeingang vorgegebene Strecke nochmals fahren.

```
36      while(true)
37      {
38          if((In.Inputs & 0x10000) == 0x10000)                        // check if Input 1 is active
39          {
40              InOut.TargetPosition = In.ActualPosition + (In.AnalogInput * 10); // calculate new target position
41              InOut.ProfileVelocity = 50;                             // set new target velocity
42              yield();
43              Out.ControlWord = 0x2F;                                 // operation enable abs. positioning
44              yield();
45              Out.ControlWord = 0x3F;                                 // start
46              yield();
47              while((In.Inputs & 0x10000) == 0x10000)                 // loop while Input 1 is still active
48              {
49                  yield();
50              }
51          }
52          yield();
53      }
```