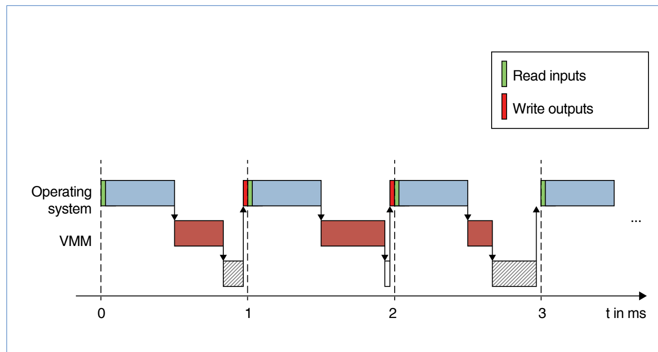
Application-specific programming with NanoJ Easy 2.0

New technology for the second generation of the integrated programming language

In the second generation of our integrated programming language, we have fully revised the technology to achieve real-time capability and a high execution speed. While the programs in version 1 of NanoJ ran as a byte code in a virtual machine parallel to the motor controller and therefore were not subject to a strictly deterministic timing, the new version now contains a deterministic 1-ms cycle. This is ensured by means of "cooperative multitasking":



In each 1-ms cycle, the data are first read out of the object directory, which contains all settings and state values of the controller as the central database. Then the "operating system" is executed, such as the high-level functions of the motor controller and the field bus communication.
The actual controller runs considerably faster at 32 kHz. After the operating system has executed all cyclically required operations, it transfers the execution to the user program in the VMM (Virtual Machine Monitor).

The user program can now modify values from the object dictionary, perform calculations, etc., but must return control to the operating system before 1 ms has expired. For this reason, this process is also referred to as cooperative multitasking. Before the cycle expires, the output values of the user program are written back to the object directory and are thus processed by the motor controller in the next cycle. They can also be read out again via the field bus, for example.

A sand box for the user program

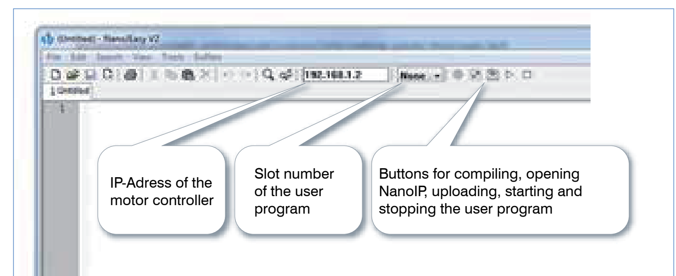
The VMM makes a protected execution environment available within the firmware. The so-called sand box restricts the user program to a certain memory area and certain system resources and thereby ensures that the user program can never cause the actual controller firmware to crash.

The system also offers protection against a control delay due to computationally-intensive user programs: If the user program is not cooperative, i.e. if it does not return control to the operating system before the cycle expires, the program is terminated. An error message appears during which the motor controller continues running smoothly and without delay.

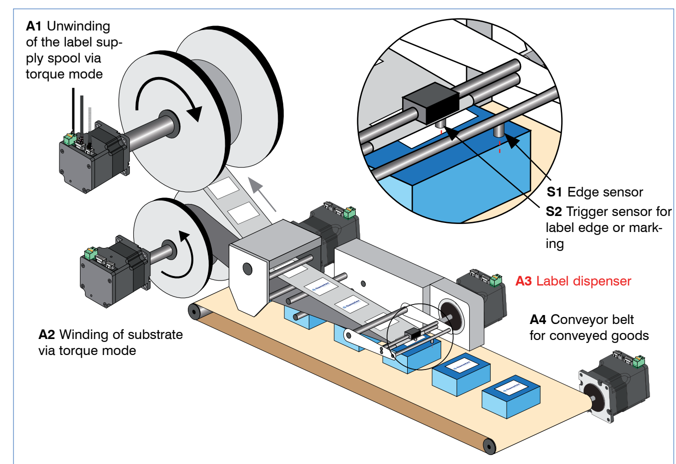
The sand box also enables the second major new feature: instead of a byte code, the markedly faster machine code is now executed directly.

The NanoJ Easy programming environment

The NanoJ Easy programming environment is designed for the simple creation of programs in NanoJ. The programs can be developed and compiled using the integrated editor with syntax highlighting. In the case of motor controllers with an Ethernet interface such as N5, the program can also be transferred directly into the motor controller and started and stopped.



Example: label dispenser



Below, we will develop a simple program in two steps that reflects the function of a label dispenser.
The tasks of the other motors in the application are very easy to implement. The motors for winding and unwinding run in torque mode, the conveyor in speed mode, and both of these are available as standard operating modes without programming.

The dispenser, however, requires a small program. Our objective is to first accelerate the motor to a constant speed that equals that of the conveyor belt, and then to move a defined distance (the label length) when the label edge is detected. To keep the example simple, we have used only one trigger input here.

In real-life applications, there is at least an additional start input that starts the motor. After you have understood the example, it will be easy for you to add the input query.

The first simple version

The following program is our first simple version, which only activates the positioning mode and starts the motor:

```
 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 of object directory entries

Mapping, in which the variables in the program are assigned to the object directory entries, occurs in lines 5-7. Line 6, for example, specifies that the content of object 0x6081 should be adopted in the 2-byte, signed (S16) variable "ProfileVelocity" with each cycle and should be written back at the end of the cycle. The object addresses or objects correspond to the CAN standard DS402; 6081 is therefore the maximum positioning speed. If the mapping is declared not as "inout" but as "input" or "output", the variable is only read at the beginning of the cycle or is written back at the end of the cycle, such as the control word in line 6.
The preprocessor instruction "include "wrapper.h"" in line 9 is of no further interest to us here. It merely needs to be included as an instruction to the compiler in every NanoJ program.

The main program

In line 13, the main program begins with the "user()" function, which corresponds to the "main" function in C or Java in NanoJ and is always executed as the first function.
In line 15, we encounter the second possibility of accessing the object directory apart from mapping, namely the "od_write" command. With this command, objects that are required rarely or only once and that therefore do not need to be read in during every cycle can be changed or read in with "od_read" during the program sequence. Thus, the positioning mode is activated here by accessing the CAN object 0x6060, "Modes of Operation".
In the next two lines, the mapped speed and a target position are specified. On account of the mapping as "inout", the assignment of a value to the variable is sufficient here; an explicit write command is not required. Lastly, the DS402 final state machine is switched to the "ready to switch on" state via the mapped control word.

The next line then continues with the first "yield()" with which our program returns control back to the operating system, thereby closing the 1-ms cycle. As the subsequent lines show, a "yield()" follows every transition of the final state machine since every state needs to be run through in the controller. If there were no "yield()" in line 22, the program would switch directly into the "switched on" state from the point of view of the controller. This is not permitted by the CAN standard and would not function for this reason.
After the fifth "yield()" and thus after 5 ms, the motor controller is active and the motor accelerates to a speed of 200 steps/s. The unit depends on the settings in objects 0x2060-0x2062; in our case, we assume that these objects contain the default settings. Thus we have reached the end of the first program version. In the last lines, an infinite loop prevents the program from coming to an end since it would otherwise be restarted in the next cycle.

Trigger input and analog input

What we are now missing is the reaction to the trigger input, which starts a predefined path. Our first program version runs (almost) infinitely at a constant speed since a target position was selected that can in fact never be reached. This would normally be an application for the speed mode; however, instead of using this mode here, we will simulate it by specifying an unreachable position target value. This trick saves us the effort of having to switch between two operating modes in the next step. As a reaction to the input, it is sufficient to reset the target position. In addition, the analog input is used to easily set the label length.

To see a change at the input in the program, we first need to map the object for the digital inputs and analog input (lines 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
```

We now expand the infinite loop of our first version with the code for the input query and setting of the target position:
Due to the "&" (logic "and") link of the object for the digital inputs in line 37 with bit mask 0x10000, only input 1 is monitored. When the input changes, the target position and the speed are changed. The target position is calculated from the value of the analog input (0 to 1023) with a multiplier of 10. When the input is set, the current position is read out, the target position that is calculated from the analog value is added and absolute positioning is activated in line 43. This ensures that the delay of 2 ms until the position is adopted in the controller after line 47 no longer influences the determined target position.
The loop at the end is used to wait until the input changes its state again. Subsequently, the motor would now stop and every time the input switches, the motor would again travel the distance specified by the analog input.

```
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      }
```