University of Twente

department of
Electrical Engineering

# 20-Sim code generation for PC/104 target

**Marcel Groothuis**

**Individual Design Assignment**

Supervisors J.F. Broenink
M.H. Schwirtz

July 2001

Report 009R2001
Control Laboratory
Electrical Engineering Department
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

## Summary

From version 3.2, 20-Sim will contain a new tool, called *C-code generation*. With this tool it will be possible to generate C code from a 20-Sim model. This tool works on basis of templates. For each target, a target specific template has to be made.

The goal of this project was to write a new 20-Sim template for a PC/104 target, a embedded Intel 80486 pc on the mobile robot Arty. It should to be possible to control the robot with a 20-Sim model. The template had to be written using the Phar Lap TNT Toolsuite Lite, a toolsuite for real-time embedded development. The written template is working and to demonstrate this, a 20-Sim model that controls the robot is made.

Another goal of this project was testing the code generation tool and doing some recommendations for improvement. From this project it followed that this tool works good, but that the following points have to be improved:

- Controlling the target hardware from a 20-Sim model is possible but only via a time-consuming workaround route.
- The generated code is large, because it contains many not-used functions and redundant variables and equations.
- The Runge Kutta routines fail to work in the existing templates when a model has no states.

## Samenvatting

Vanaf versie 3.2 zal 20-Sim een nieuw gereedschap bevatten, genaamd *C-code generatie*. Met behulp van dit gereedschap is het mogelijk om C code te genereren van een 20-Sim model. Dit gereedschap werkt aan de hand van templates. Voor elk systeem moet een specifieke template geschreven worden.

Het doel van dit project was om een nieuwe 20-Sim template te schrijven voor een PC/104 systeem: een embedded Intel 80846 pc op de mobiele robot Arty. Het moest mogelijk zijn om de robot te besturen met een 20-Sim model. De template moest gebruik maken van de Phar Lap TNT Toolsuite Lite, een pakket voor ontwikkeling van real-time embedded software. De geschreven template werkt en om dit te kunnen demonstreren is een 20-Sim model gemaakt dat de besturing van de robot regelt.

Het tweede doel van dit project was het testen van het code generatie gereedschap en het doen van aanbevelingen voor verbetering hiervan. Hieruit volgde dat dit gereedschap goed werkt, maar dat er nog een aantal punten zijn waarop het verbeterd kan worden:

- Het besturen van de hardware van het doelsysteem vanuit een 20-Sim model is mogelijk maar alleen via een omslachtige omweg.
- De gegenereerde code is groot, omdat het veel niet gebruikte functies bevat en overbodige variabelen en vergelijkingen..
- De Runge Kutta routines werken niet bij de bestaande templates als een model geen states bevat.

# Contents

# Preface

This project was a Individual Design Assignment (IOO in Dutch). This project was not possible without the assistance and support of a number of people.

I was not the only person, that was working with the code generation tool of 20-Sim. Joël Geerlings, Thijs Withaar and Hans van de Vis worked also with this tool. I wish to thank them for the discussions we  had on 20-Sim and the code generation tool and the valuable tips, they gave to  me.

I wish also to thank the people of Controllab Products BV (the producers of 20-Sim), especially Paul Weusting and Frank Groen, for their valuable tips and attempts to solve problems/bugs in 20-Sim a soon as possible. Without the assistance of them, this project was not completed yet.

And last but not least, I want to thank my supervisors Jan Broenink and Marcel Schwirtz for their cooperation and assistance. I want to thank also the remaining people that have assisted in the realization of this project.

Enschede, July 2001

Marcel Groothuis

# 1 Introduction

With the modeling and simulation package 20-Sim (http://www.20sim.com), it is possible to make models for various dynamic systems and design control systems to improve the dynamic behavior. The behavior of these models can be simulated with 20-Sim. But these are only simulations. For implementing a designed controller in a real (embedded) system, it must be build using the parameters used in the 20-Sim model.
It would be nice that it is possible to use the 20-Sim model to control a real system. Therefore, 20-Sim 3.1 Pro contains a code generation tool (Weustink *et al.*, 2001). It is possible to design a controller (or something else) in 20-Sim and to generate C-code from 20-Sim. Now, the generated code must be compiled for a target system and the controller is ready. See also Figure 1.

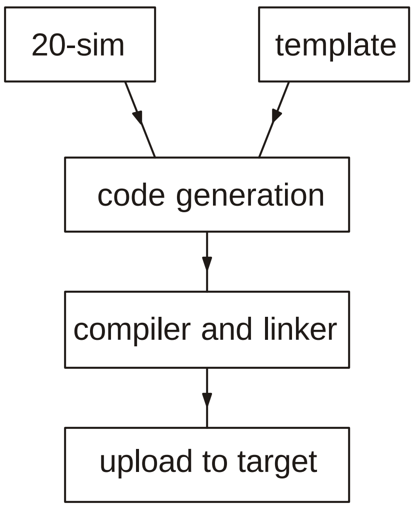


**Figure 1: 20-Sim code generation overview**

The possible targets of this tool are *Stand-alone C code,* a *C function* or a *Simulink S function*. The user has no possibility to influence the generated code, so the tool in version 3.1 Pro is not flexible. Therefore a new code generation tool will be included in the newest version of 20-Sim (version 3.2). This version of the code generation tool enables the user to customize the generated code for a specific target.
To generate code, 20-Sim 3.2 makes use of target specific templates. These templates are almost working C programs with all the functionality needed to control the target. The only code that is absent is the model-specific code. So for every target, that uses specific hardware 'drivers', a new template has to be written.

Purpose of this project was to write a C-code template and to make a 20-Sim model, used for code generation from 20-Sim. This template is specific for one target, namely Arty (Balkema *et al.*, 1999). Arty is a mobile robot with a PC/104 board with an Intel 486 processor. Part of the assignment was using Phar Lap's real-time ETS kernel (Grehan *et al.*, 1998) as host for the generated code. The findings about

Phar Lap will be used to check whether this software package is suitable for education purposes and real-time software development.
At this moment, the code generation tool of 20-Sim is still in beta phase. 20-Sim version 3.2 is not released yet. So, a part of the assignment is also testing of the 20-Sim code generation tool and giving recommendations for improvements of this 20-Sim tool and eventually new 20-Sim tools for facilitating the use of the 20-Sim code generation tool.

This report is organized as follows: in chapter 2, the 20-Sim code generation will be described. Chapter 3 contains an overview of the tools and utilities of the Phar Lap TNT toolsuite. The most relevant features of this toolsuite will be discussed. In chapter 4, the relevant features of the PC/104 target system will be discussed. In chapter 5, the written template and the 20-Sim testmodel will be discussed. Chapter 6 contains the tests and verifications of the template and testmodel.
Chapter 7 contains a discussion of the findings of the Phar Lap toolsuite and the code generation. Chapter 8 contains the final conclusion and recommendations.

# 2 20-Sim code generation

## 2.1 Introduction

This chapter contains an overview of the code generation functionality included in 20-Sim version 3.1.0.6ß. First, the code generation process is described. This is explained by an overview followed by a detailed description of the code generation process.

## 2.2 Code generation

### 2.2.1 Overview

The code generation tool is a new tool that will be released with the newest version of 20-Sim: 20-Sim 3.2. Because this version is not yet released at the moment, during this project 20-Sim version 3.1.0.6ß has been used. This version can be installed as an upgrade over version 3.1.0.5 (see appendix A). When the new version is installed the code-generation tool is available.

The process of code generation from 20-Sim consists of four steps (Weustink *et al.*, 2001). See in Figure 2.

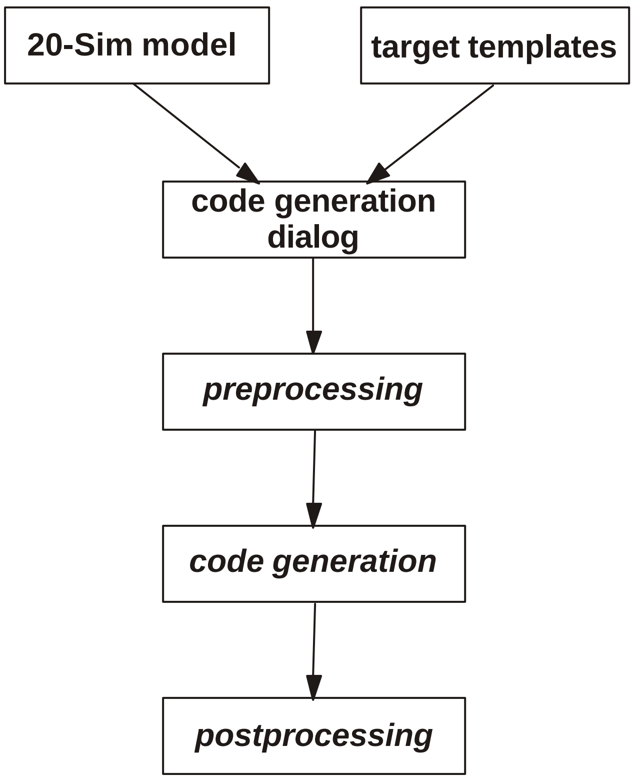
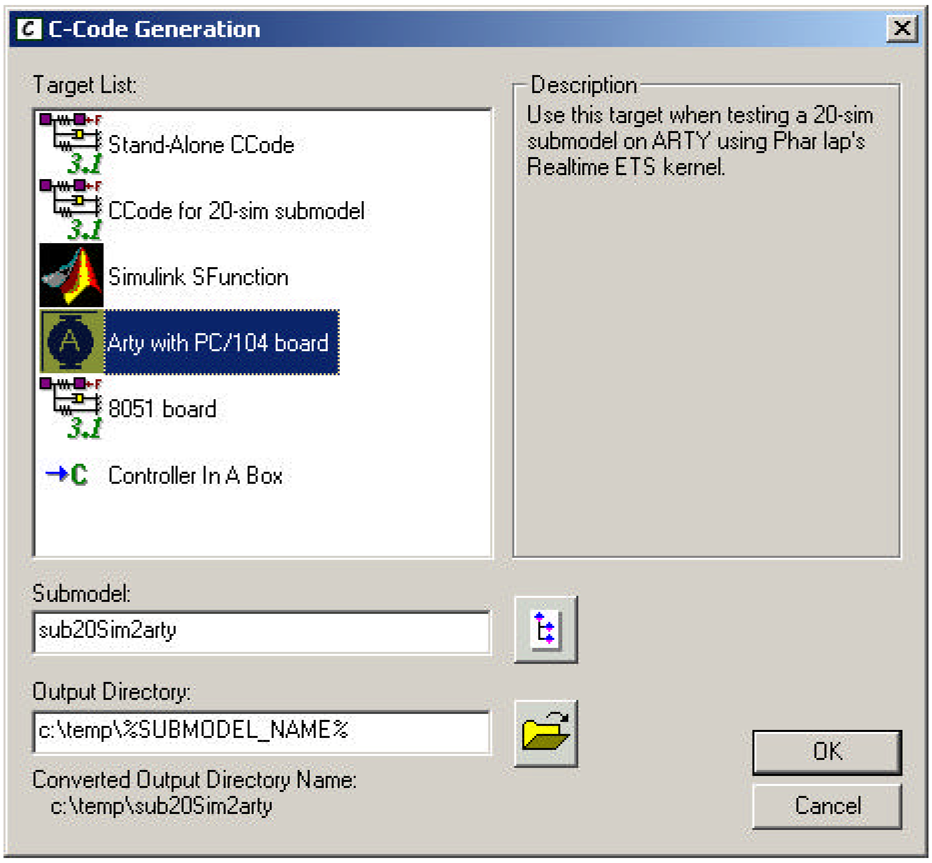


**Figure 2: Code generation process**

1. The process of code generation from 20-Sim starts with a model and a simulation in 20-Sim. After simulating the model and verification of the results, the user selects then the “Generate C-Code” tool from the “Tools” menu in the 20-Sim simulator. 20-Sim will now read a target configuration file (targets.ini; See for information Appendix C). This file contains information about the different available targets. 20-Sim will now open the code generation dialog (see Figure 3) and the user can select the preferred destination target, the output directory and, for submodel code generation, a submodel from the “code generation dialog”.
2. In the second step 20-Sim reads the target-specific template files specified in the target configuration file and starts (when necessary) a preprocessor that can process the template files before 20-Sim will use them.
3. The third step is the code generation step. 20-Sim processes the template files and the model and appends, at beforehand-defined places, the model specific code to the template code.
4. In the last step the adapted template files are put in the specified destination directory. Now 20-Sim can start an other program (a post processor) for further processing the files. For example the post processor can start a compiler, linker and an uploader for running the code on the specific target.



Description
Use this target when testing a 20-sim submodel on ARTY using Phar lap's Realtime ETS kernel.

Submodel: sub20Sim2arty

Output Directory: c:\temp\%SUBMODEL_NAME%

Converted Output Directory Name: c:\temp\sub20Sim2arty

**Figure 3: Code generation dialog**

## 2.3 The process of code generation

In the current situation, the code generation process is a adapted form of the 20-Sim simulation process (Figure 4) Before the simulation is started, the 20-Sim model will be converted to sequential model code.

This code will be adapted to make it suitable for the simulator. The simulator uses the simulation model code to calculate the model equations and presenting the result on the screen.

The same applies also to the code generation process. After the sequential model code is generated, it will be adapted to make it suitable for code generation by the model postprocessor. Then, the code generation tool uses this code and the code from the template to combine it to target specific model code, by replacing tokens in the template by the model code (see Appendix D for more detailed information).

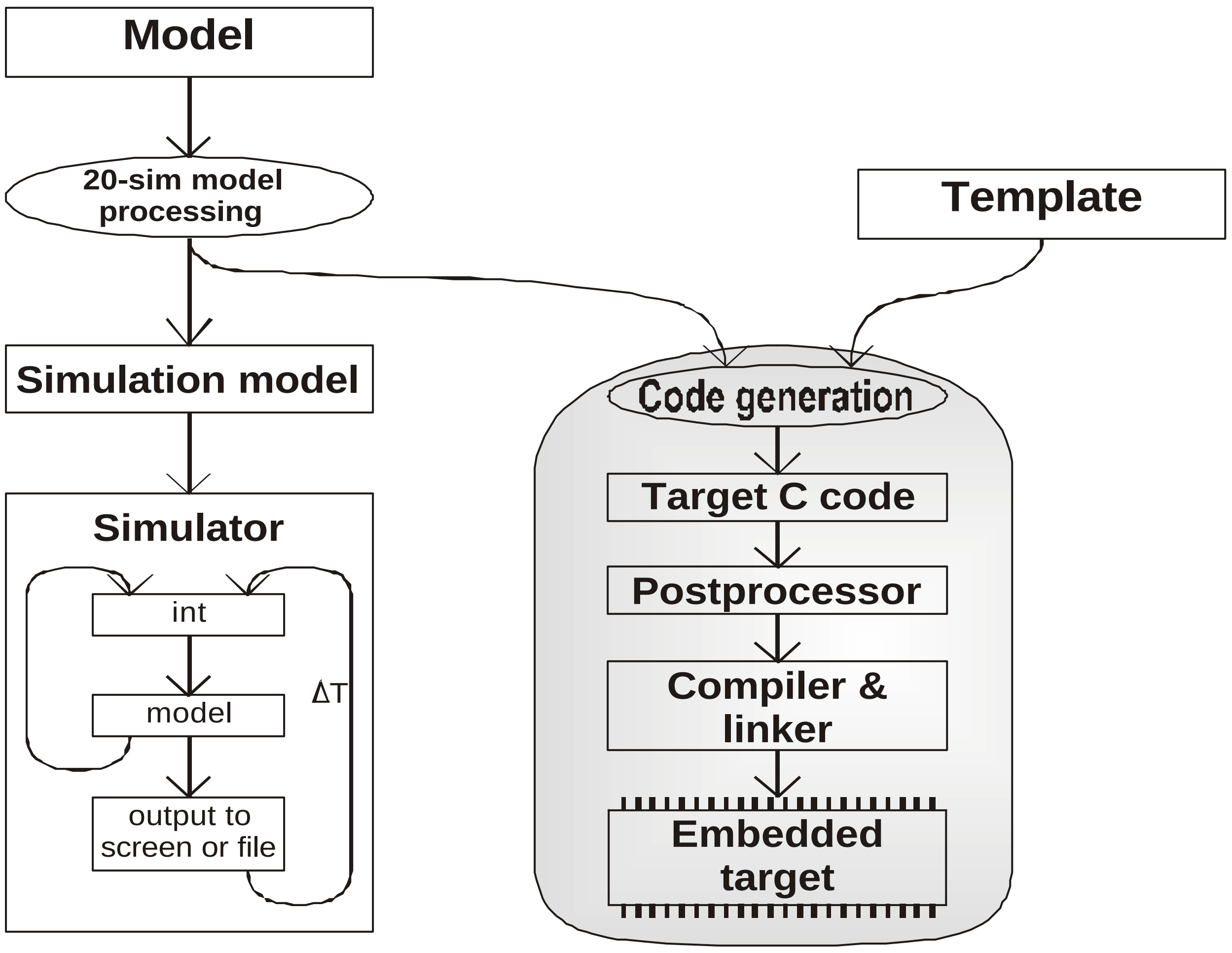


**Figure 4: Overview of the 20-Sim simulation and code generation**

The existing templates (e.g. Ansi C template) use the same execution structure as the simulator does. This execution structure is depicted in Figure 5. The initial equations of the model are equations that are calculated before anything else. The static equations of the model are only dependent from parameters and constants. The input equations of the model are dynamic equations that do not change in calls from the integration method. The dynamic equations are calculated by the integration method to calculate the new model states. The final equations of the model are calculated after all the calculations are performed. On each calculation step, the input variables, the output variables and the dynamic equations are calculated. The integration method determines the number of times the dynamic equations will be calculated during one calculation step (e.g. four times with Runge Kutta 4).

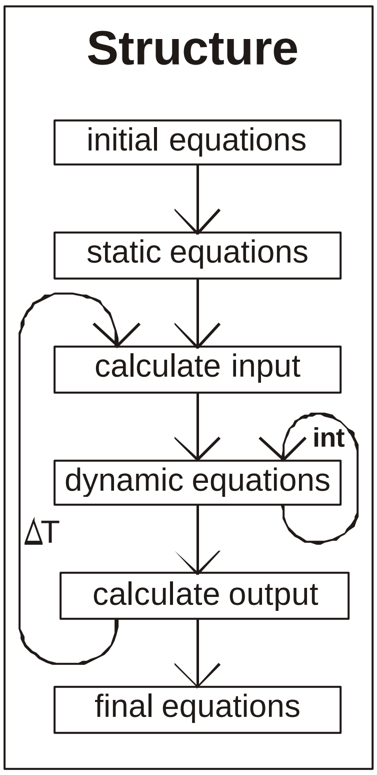


**Figure 5: Calculation structure**

When looking at Figure 4, code generation and simulation are two different ways that start with a model. Code generation is independent of simulation. Therefore the place to find the code generation tool (in the Tools menu of the 20-Sim simulator) is *not* a good place. Normally, the model is first simulated (verification) before generating code, but this is not necessary.

## 2.4 The template

Two types of templates exist, because there are two different ways of generating code. The first way is to generate code from the total 20-sim model the second is to generate only code of a submodel of the 20-sim model. When the generated code has to run on a target only the second way is of importance because only the contents of the target sub model (e.g. a controller) has to be uploaded to the target. This applies also for this project. In Appendix K both methods with the corresponding templates are described in detail.

## 2.5 Relation simulation and code-generation

The physical target contains several connections to the 'real world' such as ADC, DAC, etc. In 20-sim these functions have to be simulated. However, when code is generated for the target (for uploading into the target) the simulation code has to be replaced by functions calls. These functions now contain drivers for the ADC, DAC etc. This problem is not yet satisfactory solved in 20-sim. Fortunately for testing-purposes the problem is temporarily resolved by use of dlls. In the 20-sim submodel the target elements (ADC, DAC, etc) are represented by icons, which contain a dll statement like:

```
Y = dll(filename,functionname,X);
```

Where X is an input matrix with n by m elements and Y an output matrix with s by t elements.
After code generation this dll command will be replaced by a function call. The generated code looks like this:

```
/* Submodel\y = dll (Submodel\filename, Submodel\functionname,
Submodel\X); */

filename_functionname (outputmatrix, number_of_elements,
inputmatrix, number_of_elements, xx_major);
```

**Listing 1: Function call after code generation**

As can be seen from this, the called functionname consists of the dll name (without extension) and the used function. (The variable order is a bit strange. In future versions of 20-Sim the “output, input” order will be replaced by a more logical order “input, output”). This method enables the user to call target specific functions from within 20 Sim.

## 2.6 Preprocessing and postprocessing

After code generation, a directory with a number of source files exists. Most of the time this is not the endpoint. The source code is only a means to an end. When further processing has to be done, such as compilation, it is possible to add a post command to the target configuration file (See Appendix C). This command (program) will be executed when 20-Sim has completed the code generation process.
For the same purpose, a pre command tag is also available in the target configuration file, which enables the user to call a program before 20-Sim generates code. This can be useful for warning purposes e.g. “Switch on the target before continuing”.

# 3 Phar Lap's TNT Embedded Toolsuite real-time Edition

A part of the assignment was to make use of the real-time ETS Kernel and tools from Phar Lap (Lite version). This software package is especially for building 32-bit real-time embedded applications. The lite version of this toolsuite that was supplied with the book "Real-Time Programming" (Grehan *et al.*, 1998) was used for this project. The TNT toolsuite consists of a real-time Embedded kernel, an add-in for the Microsoft Visual C++ IDE (Embedded StudioExpress) and a linker for building the programs. In this chapter each of these parts will be discussed.

## 3.1 Real-time ETS kernel

The real-time ETS kernel is a real-time operating system for running real-time embedded programs on a Intel 80x86 processor.
The kernel initializes in 32 bit protected mode, so the programmer does not have to worry about the real-mode barrier of 640 kB DOS memory. The kernel also enables interrupts and disables the processor's paging hardware during the initialization. Therefore, paging is not possible and the ETS kernel operates only on physical memory. The advantage of this approach is a better guarantee for timing, because paging costs time. Furthermore all addresses are physical addresses and embedded applications can directly access all physical memory. (Grehan, *et al.*, 1998 pages 60-63)

The kernel consists of several comprehensive API's for faster development of embedded applications:

- **C/C++ runtime libraries** Basic C++ functions, loaded at kernel setup.
- **Win32** Various Windows *like* (this is NOT Windows) functions. The functionality of the Win32 functions matches the Microsoft Windows 32-bit functions.
- **WinSock** Socket communication functions for networking.
- **ETS** Functions only defined for ETS. For example com-port routines, installing interrupt service routines and configuring/programming of the ETS kernel.

More information about the API's can be found in Grehan *et al.*(1998) and the document APIS.DOC in the directory `\TSLITE\DOCS` on the accompanying CD.

The kernel has also a number of insertable modules for customizing the kernel. The following modules are available:

- **real-time scheduler**. Kernel component that supports real-time multithreaded/multitasking applications
- **TCP/IP stack.** Adds TCP/IP functionality to the WinSock API.
- **MicroWeb server.** HTTP server and tools for building web pages on demand.
- **MS-DOS-compatible file system.** This makes file I/O to FAT16 and FAT32 file systems possible. File access is not only possible on the target system, but also on the host system when connected by cables.
- **Floating Point Emulation Library.** When the target has no FPU, the floating point emulator makes it possible to use floating point calculations, however this is slower than a real FPU.

- **DLL loader.** Makes it possible to use modules in the Pharlap programs. The toolsuite uses the same format as Windows, DLL (dynamic link library).

## 3.2 Embedded StudioExpress

Another part of the TNT toolsuite is an add-in for the Microsoft Visual C++ IDE: Embedded StudioExpress. With this add-in it is possible to write, compile and debug an embedded program within the Visual C++ IDE. The add-in consists of the following components (icons on the Embedded StudioExpress toolbar in Figure 6):

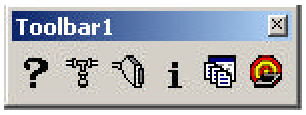


**Figure 6: Embedded StudioExpress toolbar**

- **Target Configuration Settings** . Tool for setting the download and communication settings.
- **Target Port Input/Output.** With this tool it is possible to access the I/O ports on the target directly, without running an embedded program (only the ETS monitor has to be activated). This tool is handy for use with Arty.
- **Target System Information.** This tool gives detailed information about the target and its status. This tool is useful for debugging. For example, it gives information about the used threads, the memory status, loaded modules, hardware, timer, kernel status, etc.
- **ETS Project Wizard.** This wizard helps creating new ETS projects. When using this wizard, Visual C++ automatically knows that the project is not a normal (console) C++ project, but an ETS project and uses therefore automatically the LinkLoc linker instead of the normal linker. With this wizard the kernel modules to be used can be specified.
- **Visual System Builder** (not available in the lite version) A Windows utility for easily configuring the ETS kernel for custom hardware.

## 3.3 Linker

To build projects made with the TNT toolsuite Lite, a special linker (LinkLoc) must be used, because the standard Visual C++ linker does not work with the ETS kernel.

The difference between normal programs and ETS programs is, that the kernel is linked with ETS programs to one *executable*. The generated executable can only be executed on the embedded target using the RUNEMB command on the host machine (Lite version). This program sends the executable to the target, executes it and starts the debug monitor when the Visual C++ debugger is used.

To send the program from the host PC to the target PC, two things are needed. First, the target must be booted with a special program, called ETS monitor, and second, two connection cables are needed between the host and the target. A serial nullmodem cable and a parallel LapLink cable (see Appendix E for the cable-layouts).

## 3.4 Difference between the full and the lite version

The lite version of the TNT Toolsuite has a number of restrictions with respect to the full version. In this section, these differences will be shortly described.

Restrictions of the lite version:

- Visual System Builder and CodeView debugger are not included
- A number of utilities are not incuded (386|ASM, 386|LIB, MAPEXE, MAKEHD, REBIND)
- No support for Inprise's C++ (builder)
- No support for older versions of Visual C++ than version 5.0
- Applications must be downloaded to the target from a host computer. Support for off disk loading, running from ROM and running without a host-target connection (standalone) is not available.
- No support for hard disks, RAM and ROM drives and flash file systems. The only drive supported by the file system is a floppy drive
- The amount of RAM available to the embedded applications is restricted to a maximum of 1 MB.
- The maximum program size is restricted to 1 MB.
- The only possible program load address is at 64K.
- The kernel has no support for PC cards, such as a network interface card (NIC) or a soundcard.
- Part of the flexibility with respect to customizing the kernel is removed. The ETS kernel cannot be customized for special hardware. The kernel source to all drivers and hardware-specific code is only included in the full product.

# 4 The target

The target used for this assignment is a PC/104 board (PC/104 consortium, 1996). This target is installed in a mobile robot, known as Arty. (Balkema *et al.*, 1999) Arty consists of the following parts:

- PC/104 board
- Two navigation wheels
- Two back up wheels (only for balancing)
- Two bumpers for detecting obstacles. Each bumper contains two microswitches
- Eight Polaroid ultrasonic sensors for detecting obstacles (bat radar)
- One beacon sensor for detection of a infrared beacon (IR sensor, angle sensor and motor)
- 6 V battery

See Figure 7 for a picture of Arty and Figure 8 for a block diagram.

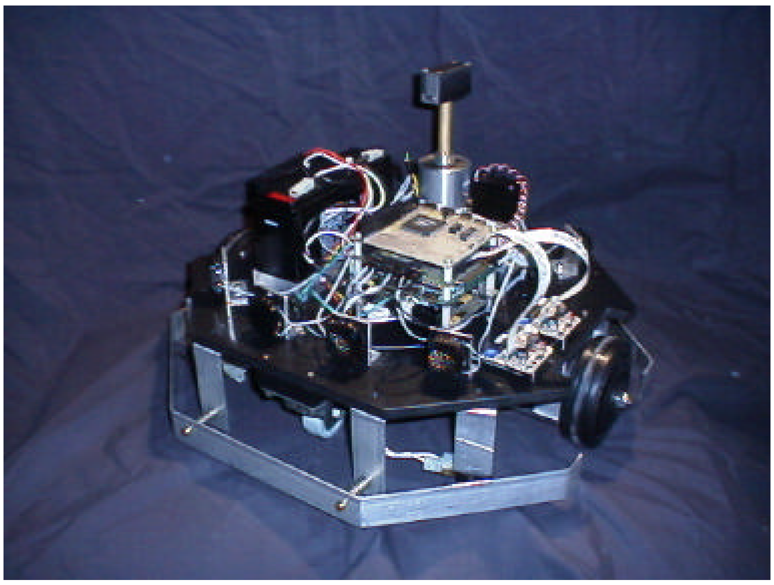


**Figure 7: Arty**

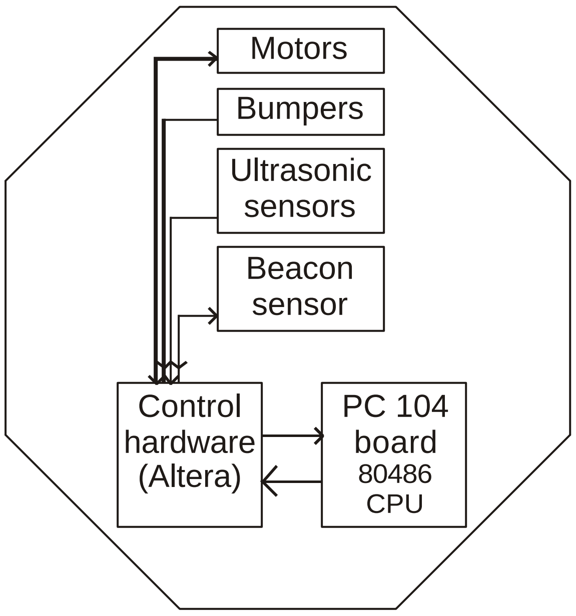


**Figure 8: Arty overview**

## 4.1 PC/104 board

The PC/104 board used, is a small-sized PC motherboard with a 486 processor and 4 MB onboard DRAM.
It has most of the features a normal motherboard also has, such as a floppy disk drive controller, a harddiskdrive controller, 2 serial ports, a parallel port and 16-bit PC/104 expansion bus.
For the exact specifications see Appendix F and Ampro (1997).

## 4.2 Navigation wheels (left and right)

Arty has two navigation wheels for moving and turning. Each wheel is connected with a motor and a velocity sensor. Controlling the left and right motor is done by writing a byte value to the I/O port corresponding with the motors. See Appendix G for the port values. A value from 0 to 7F (hex) means move forward (0 is stop, 7F is maximum speed) and a value from 80 (hex) to FF (hex) means move backward (FF is stop, 80 is maximum speed).
The hardware takes care for the real motor control and uses the build-in velocity sensors in combination with a digital PID controller for setting the correct speed. The values of the motor velocity sensor can be read via the same port as for setting the speed. Most of the time the sensor values and the established motor values are the same. Only with very low speed values (i.e. values from 0 to 10 and EF to FF) one gets back a 0, because the PID controller can’t set such a low speed due to the non-linearity of the motors.

## 4.3 Ultrasonic sensors

At the front side and the backside, Arty has four ultrasonic sensors (Polaroid 6500 Ultrasonic Ranging System). These sensors work like the ultrasonic ranging system that bats use to navigate as they fly. Bats send out various high-frequency ultrasonic waves and listen for their echoes. With this information they build a 3D representation of space. The sensors on Arty work the same way. The Polaroid ultrasonic system sends a chirp signal of 49.4 kHz; receives their echoes and determines the elapsed time. (Martin,

2001) This time will be converted by hardware to a value between 0 and 127 (the higher the value the bigger the distance). For each sensor, these values are send to the corresponding port addresses (see Appendix D). So, for determining the distances of the sensors to an obstacle, reading the byte values from the port addresses is sufficient.
In practice, these values lie between 20 and 120. A distance of 10 cm corresponds with a value around 20 and 70 cm with a value around 120. Shorter distances give unpredictable results

## 4.4 Bumper sensors

Arty has on the front side and on the backside, a bumper for detecting obstacles. Each bumper contains two microswitches, one on the left and one on the right side. The status of these four switches can be read from a I/O port. See Appendix G for the port values.

# 5 Design and Implementation of the PC 104 codegeneration

## 5.1 Introduction

In the chapters 2, 3 and 4, it is explained how 20-Sim code generation works, what Phar Lap can mean for this project and how Arty works. This chapter describes how this was used for the realization of a working template. First, the design choices are described. Then, the written template is described in detail. Also the used testmodel is described here.

## 5.2 Design choices

With respect to the Phar Lap TNT toolsuite, the following decision was made. For this project, the Lite version of the TNT toolsuite will be used. For Arty, the full version is desirable because of the ROMing support and to avoid a cable connection between Arty and the development PC. However, the goal of this project (a PC/104 template) is realizable with the Lite version.

For writing the new template, a existing template could be adapted or a complete new template could be written. Adapting the existing CFunction template was chosen, because this approach has the advantage, that starting with a working template, only the Arty specific functions and a function that enables the real-time processing capability of the ETS kernel have to be written. One disadvantage of this approach is, that the existing functions do not use the capabilities of the ETS kernel (if applicable).

## 5.3 20-Sim code template for PC/104

### 5.3.1 Introduction

In this section the structure of the 20-Sim code template will be described. The starting point for the PC/104 template is the Ansi CFunction template supplied with 20-Sim 3.106ß. Therefore, only the changes with respect to the CFuntion template will be discussed in detail The major changes to the template are extra functions for controlling the Arty, real-time functionality and some code for the postprocessing step. Further, some minor changes were made to correct some bugs (See Appendix I).

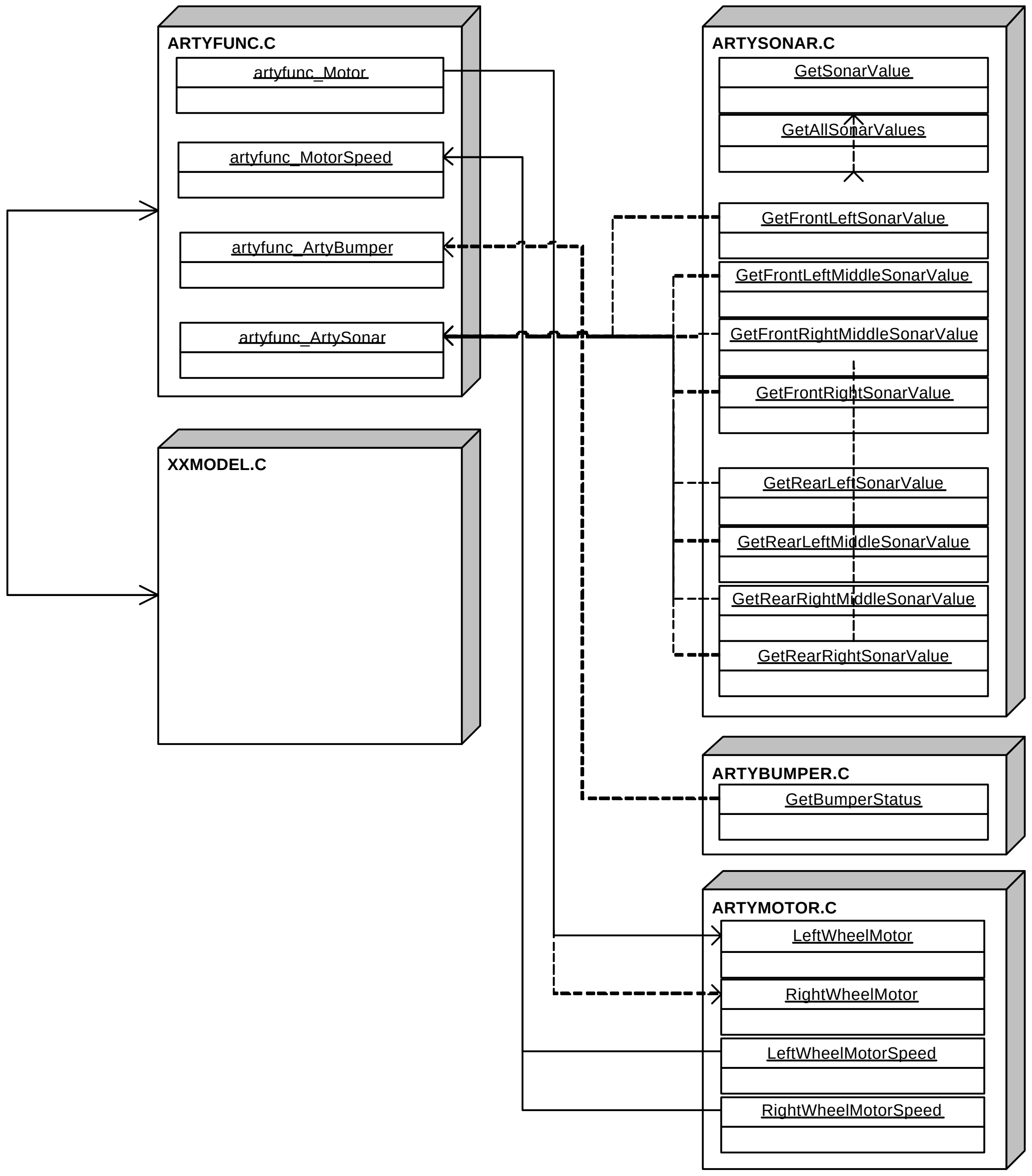


**Figure 9: Overview Arty specific functions (UML deployment diagram)**

### 5.3.2 Arty specific functions

#### Introduction

For controlling Arty from code, the template had to be extended by a number of functions for all the sensors and actuators (motors). To do this, this task was divided in two parts. The first part is the API (Advanced Programming Interface) part and the second part is the interface between the generated code and the API. Because the beacon for Arty was not available, there are no API functions for the beacon sensor.

Before discussing all files and functions a overview is given first. See **Error! Reference source not found.** for this overview.For clarity's sake, the connection between the existing template and the new files is also drawn.

### Motor API (ArtyMotor.c)

To control the Arty's motors, a number between 0 and 255 has to be written to one of the two motor ports. (see section 4.2). For speed measurement, the speed values must be read from these ports (these ports are defined in the headerfile artyport.h). This is not a convenient way for controlling the motors. To make this more convenient, a number of functions are written to handle the basic work. These functions are implemented in the file `ArtyMotor.c`. This file contains the following four functions LeftWheelMotor, RightWheelMotor, LeftWheelMotorSpeed and RightWheelMotorSpeed. Two actuator functions and two sensor functions.
The syntax for these functions is:

```
void LeftWheelMotor (int speed);
void RightWheelMotor (int speed);
int LeftWheelMotorSpeed ();
int RightWheelMotorSpeed ();
```

with speed, a value between –128 and +128.
A negative value corresponds with moving backward and a positive value corresponds with moving forward. Zero is stop and the higher the value, the higher the motorspeed. This is also valid for the two 'sensor' functions. A value outside this range will be interpreted as it's maximum value |128|

### Bumper sensor API (ArtyBumper.c)

To read the status for all four bumper switches, a function called GetBumperStatus is written. This function reads the status of all switches from port 306h and returns it. The syntax for this function is:

```
int GetBumperStatus();
```

The return value is a value between 0 and 16. To see whether a bumper is pressed, the returnvalue must be ANDed with the corresponding bumper mask. If this yields a value unequal to 0, the bumper is pressed.
The four bumpermasks and an example are given in Appendix J

### Ultrasonic sensors API (ArtySonar.c)

Each of the eight ultrasonic sensors returns a value between 0 and 255 to it's own port. To get the 'distance' of all the ultrasonic sensors, 8 port reads are necessary. To facilitate this task, a number of functions are written to accomplish this. These functions are: `GetSonarValue`, `GetAllSonarValues` and one function for each sonarsensor. So, a maximum flexibility with respect to reading out all the values is accomplished. The syntax for the functions is:

Functions for all sonarsensors:

```
      int GetSonarValue (unsigned char number);
      void GetAllSonarValues (struct SONARTYPE sv);
```

Functions for one sonarsensor:

```
      int GetFrontLeftSonarValue ();
      int GetFrontLeftMiddleSonarValue ();
      int GetFrontRightMiddleSonarValue ();
      int GetFrontRightSonarValue ();

      int GetRearLeftSonarValue ();
      int GetRearLeftMiddleSonarValue ();
      int GetRearRightMiddleSonarValue ();
      int GetRearRightSonarValue ();
```

All these functions return the 'distance' of one sonar sensor (in practice a value between 20 and 120) exept GetAllSonarValues. This function returns a structure containing all the sonar values. This structure consists of 8 byte values (chars). See Appendix J.

### DLL function code interface (Artyfunc.c)

The functions described in the three sections above are easy to call from C code, but not from 20-Sim generated code. Therefore, some interface functions are written (sort of hardware abstraction layer towards 20-Sim). The goal of these functions is to connect the '20-Sim DLL call function names' with the real API functions.
The DLL written for the 20-Sim testmodel (artyfunc.dll; see section 5.5.4) has four functions. One for each sensortype and one for driving the motor. The function names for these functions are `Motor`, `MotorSpeed`, `ArtyBumper` and `ArtySonar`. These functions can be called from within 20-Sim. Besides these four functions, two extra functions are necessary, namely `Initialize` and `Terminate` for initialization and termination.
The following functions are required:

```
void artyfunc_Motor (double *outarr, int outputs,
                         double *inarr, int inputs, int major);
void artyfunc_MotorSpeed (double *outarr, int outputs,
                         double *inarr, int inputs, int major);
void artyfunc_ArtyBumper (double *outarr, int outputs,
                         double *inarr, int inputs, int major);
void artyfunc_ArtySonar (double *outarr, int outputs,
                         double *inarr, int inputs, int major);

void artyfunc_Initialize ();
void artyfunc_Terminate ();
```

These functions are implemented in the file `artyfunc.c` and connect the generated 20-Sim model code with the hardware API. Both the Initialize and the Terminate function ensure that the motors are switched off for safety.

### 5.3.3 Real-time processing

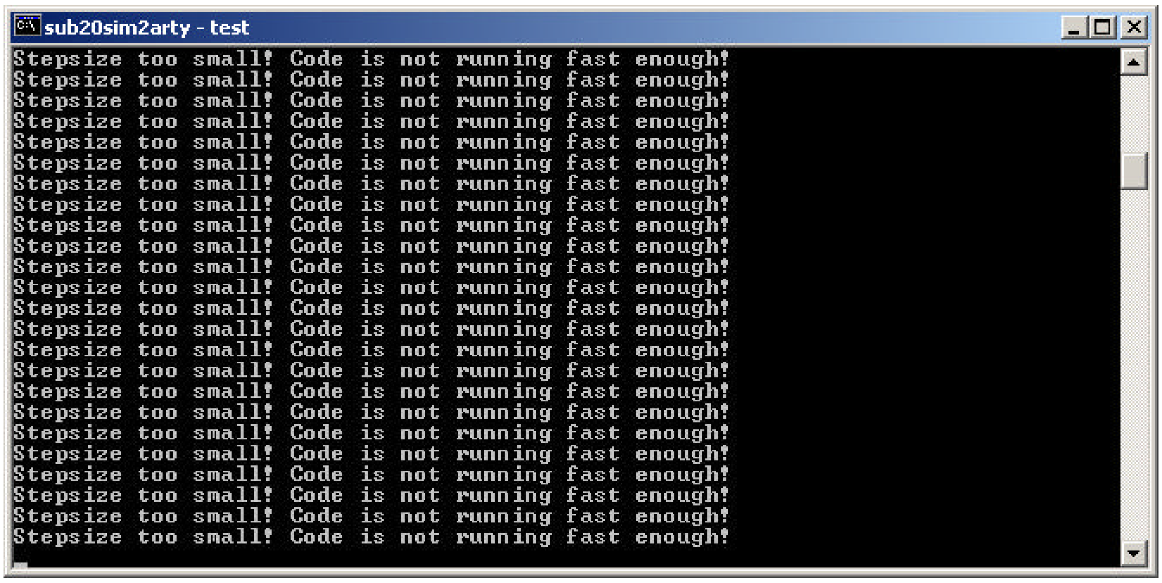
#### Introduction

To ensure that a model of 10 seconds also executes in 10 seconds, a routine is required to arrange this. This routine has to ensure that between every model calculation step, the execution pauses for a predefined time. This must happen at a manner that every `xx_step_size` seconds a calculation step is started. A Interrupt Service Routine for the timer interrupt is written to accomplish the real-time behaviour. This is a workaround for portraying the simulation execution model on the embedded system execution model.

#### Interrupt Service Routine

A interrupt service routine (ISR) is a piece of code, written for a specific interrupt, that will be executed every time this interrupt occurs. The ISR written for Arty is a routine that will be called on a Timer interrupt. To accomplish this, three functions (timer.c) are written that use the ETS kernel for hooking on the timer interrupt: `InitTimersys`, `TimerHandler` and `CleanupTimersys`.

When a timer interrupt occurs the function `TimerHandler` will be called. This function counts the elapsed time and when `xx_step_size` seconds are elapsed, the routine will increase the variable `timerint`. This happens parallel to the execution of the model code, using the multithreading facility of the ETS kernel. In the main routine (`xxmain.c`) the execution remains in a wait loop and continues only when the value of the variable `timerint` is greater than zero. As the execution continues with the calculation of the new states, the value of `timerint` will be reset to zero. When the calculation of the new states requires more time than `xx_step_size` seconds, the value of `timerint` will be larger than 1. This means that the code is not running on time any more. This will result in a warning on the screen (see Figure 10). To solve this, one has to enlarge the step size in 20-Sim before generating the code.



```
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
Stepsize too small! Code is not running fast enough!
```

**Figure 10: Console output when stepsize is too small**

### 5.3.4 Postprocessing

The code generated by 20-Sim contains a lot of unused functions. These functions (e.g. matrix manipulations) are present in the template because these functions could be required for some models. This overhead is not desirable, because it costs more memory and the downloading of the compiled code to Arty costs more time. Therefore, it would be nice that the generated code contains only the necessary functions and not all functions. 20-Sim has not yet possibilities to accomplish this, so the only way to do this is by using a postprocessor. This postprocessor has to remove all the unused functions from the generated code. But there is another way to do this. The C language has a feature, called "compiler directives" (commands that start with a #) by which it is possible to control the compiler from code. Telling the compiler which code it should compile and which code not can be done by using the directives `#DEFINE constant` and `#IFDEF constant … #ENDIF`. The preprocessor of the C-compiler compiles the code between the IFDEF and the ENDIF directives only when "constant" is defined.
To reduce the overhead and the size of the executable, all the functions in the template files are surrounded by #IFDEF … #ENDIF directives. See also Listing 2 The postprocessor searches for used functions and generates only the required #DEFINEs. One significant disadvantage of this approach is that the code will be larger and less readable (but the executable smaller) and a postprocessor always is required to process the code. The generated code can only be compiled after the postprocessor has generated the required #DEFINEs.
To make the postprocessor a optional feature and not a necessary tool, all the defines for the PC/104 template are collected in one file called `postproc.h`. So the code can be compiled without a Postprocessing step. If one needs small code, it is possible to use the postprocessor that produces a new `postproc.h` file with only the necessary defines and the code is smaller. The postprocessor written for this goal will be described in section 5.4.

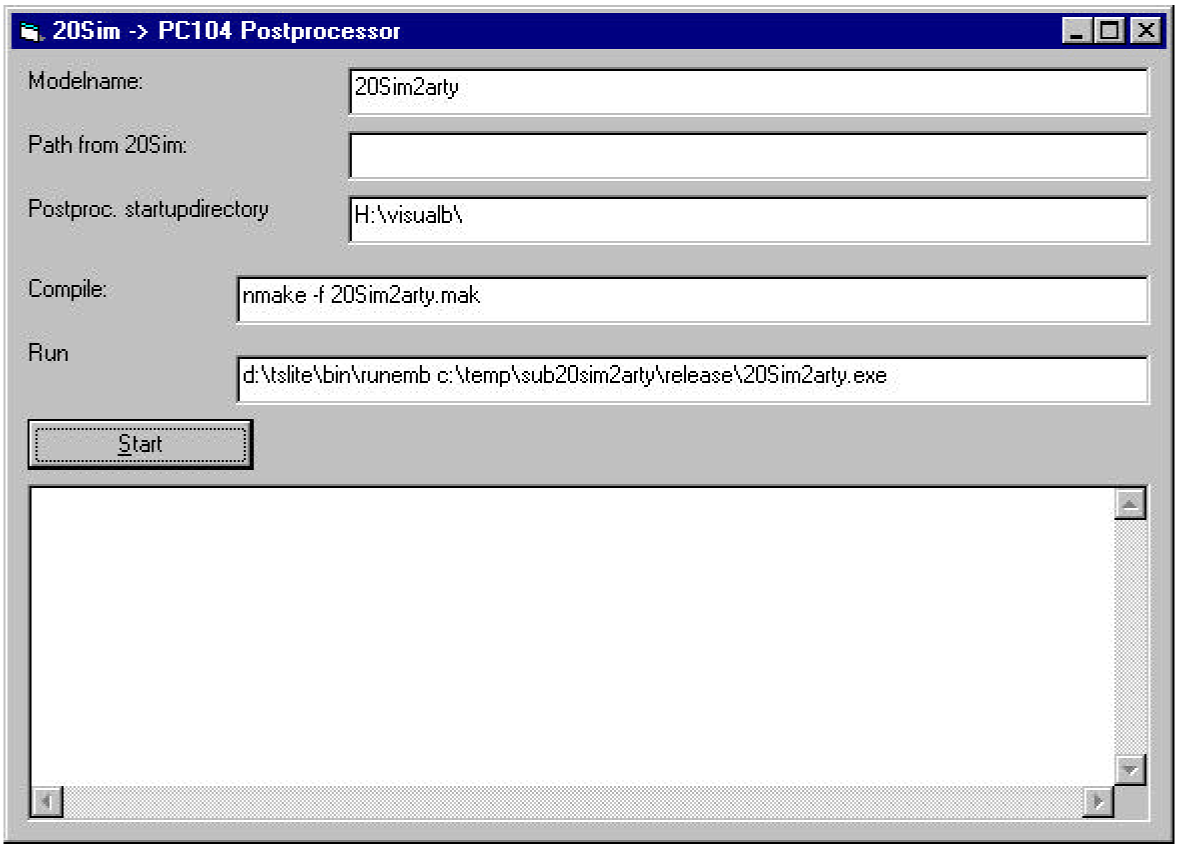
```
#define XXABSOLUTE

#ifdef XXABSOLUTE
XXDouble XXAbsolute (XXDouble argument);
#endif
```

**Listing 2: Illustration of the #define and #ifdef structure**

## 5.4 20-Sim postprocessor

### 5.4.1 Introduction

The PC/104 template comes with a own postprocessor. This postprocessor can be used for reducing the size of the generated code, for automatic compilation of the generated code and for automatic execution of the compiled code. The PC/104 postprocessor (`PC104PP.exe`) is a Windows program, written in Visual Basic 6. In this section the relevant aspects of this postprocessor will be described. See Figure 11 for a picture of the postprocessor.



Modelname: 20Sim2arty

Path from 20Sim:

Postproc. startupdirectory H:\visualb\

Compile: nmake -f 20Sim2arty.mak

Run d:\tslite\bin\runemb c:\temp\sub20sim2arty\release\20Sim2arty.exe

**Figure 11: 20-Sim PC/104 postprocessor**

### 5.4.2 Executable size reduction

One of the most important features of the PC/104 postprocessor is the size reduction of the generated executable. After code generation, the postprocessor will replace the existing postproc.h file with a new one containing only the required defines. The postprocessor uses two files to determine which functions are used. First, the postprocessor will read all the defines from the old `postproc.h` file (generated by 20-Sim). From this defines, the postprocessor knows for which functions it must look in the generated code. To simplify the searching for the postprocessor all the 20-Sim additions to the template (model specific code) will also be placed in the file `ppparse.def` by 20-Sim. The postprocessor uses this file to look for functionnames corresponding with one of the defines in the old `postproc.h` file. After completion of the search task, the postprocessor will rename the old `postproc.h` file to `postproc.old` and create a new `postproc.h` file with only the required defines uncommented. For an overview, see Figure 12.

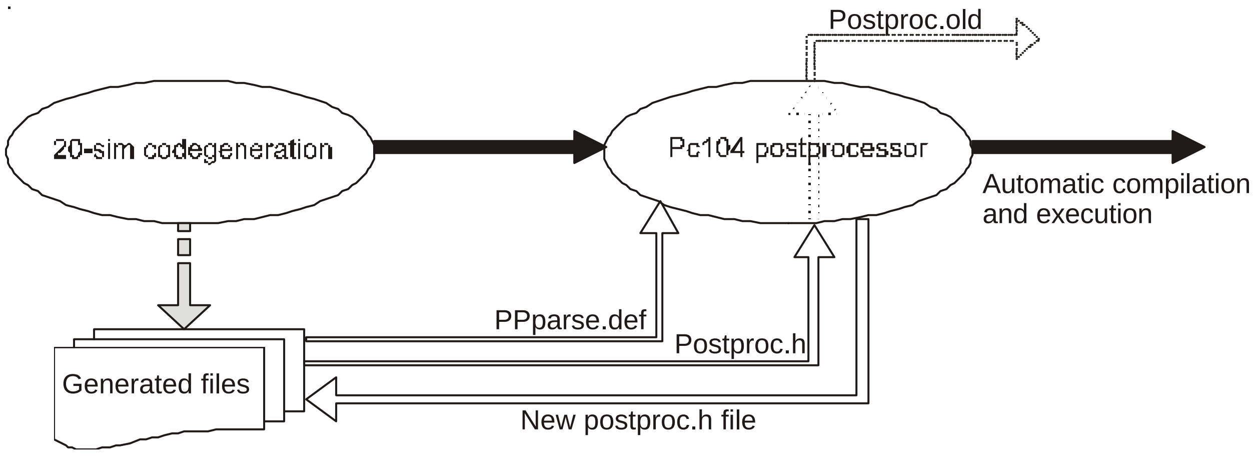


**Figure 12: Overview postprocessor**

### 5.4.3 Automatic compilation

The second feature available for the postprocessor, is automatic compilation of the generated code. This option can be enabled to edit the `postproc.ini` file. In this initialization file, it is possible to specify the compiler and where to look for the code. (Note: the current version of 20-Sim does not pass the modelcodepath to a postcommand. There is no possibility to do this. This will be corrected in future versions) Currently, the generated code is compiled by using a make file (`20SIM2ARTY.MAK`) and a linkloc link file (`20SIM2ARTY.LNK`) and the Visual C++ make command `nmake`.

### 5.4.4 Automatic execution

Because Arty is always connected to the host machine with cables, automatic execution of the compiled code is possible. Normally, the code will be executed by means of the RUNEMB command. This could also be done automatically. Therefore a option is build into the postprocessor for automatic execution. To enable this feature, the `postproc.ini` has to be edited.

## 5.5 20-Sim testmodel

### 5.5.1 Overview

To test the template, a simulation model of Arty is designed. This model is created by means of the building-block approach (Broenink *et al.*, 2000). This facilitates the reusability of the different parts, the model becomes more well-organized and it allows to check alternative solutions during design. The building block approach is a object-oriented approach. Each elementary part of Arty (such a motor or a bumper) is described by one block (submodel).

The Arty model consists of two main submodels, one with a 20-Sim description of the dynamics of Arty (simulation part) and the other contains the control system that drives Arty (control part). See Figure 13 for this division. These parts are both composed of more submodels. In the next part of this section, these both parts are described in more detail.

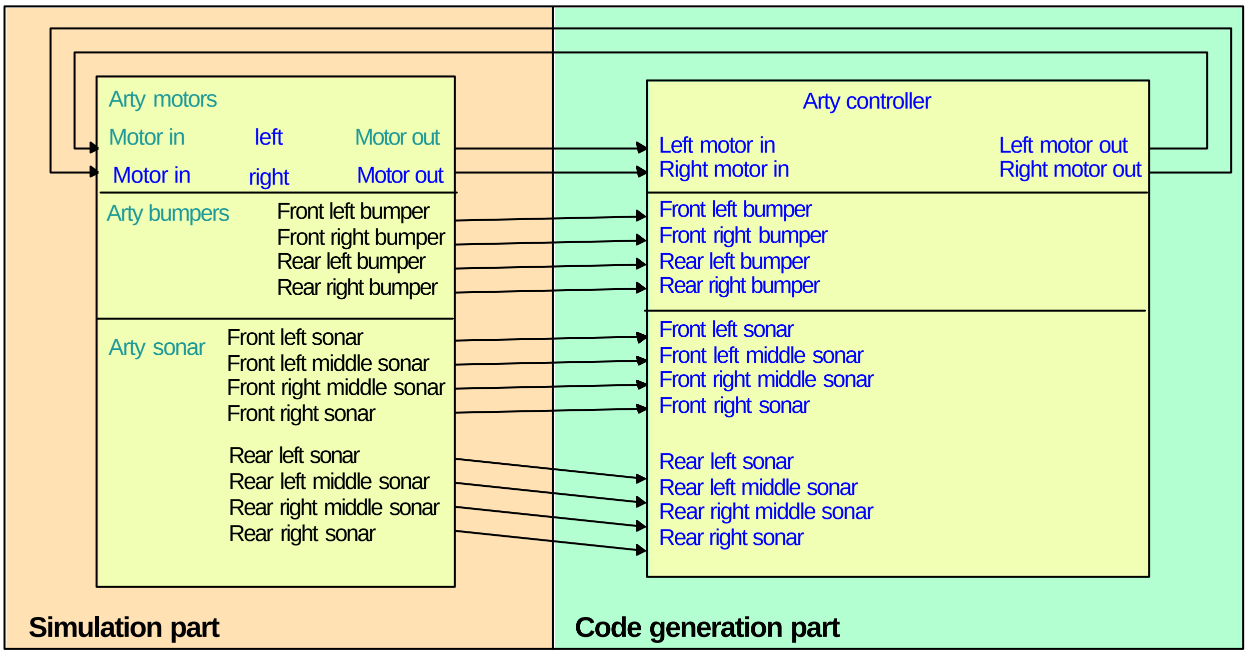


**Figure 13: Test model overview (20-Sim model ARTYMODEL)**

### 5.5.2 Simulation part (Arty submodel)

The simulation part of the testmodel is only for testing the code generation part. The simulation part contains some submodels for the bumper sensors, the ultrasonic sensors and the two motors. See Figure 14 for the inside of this part.

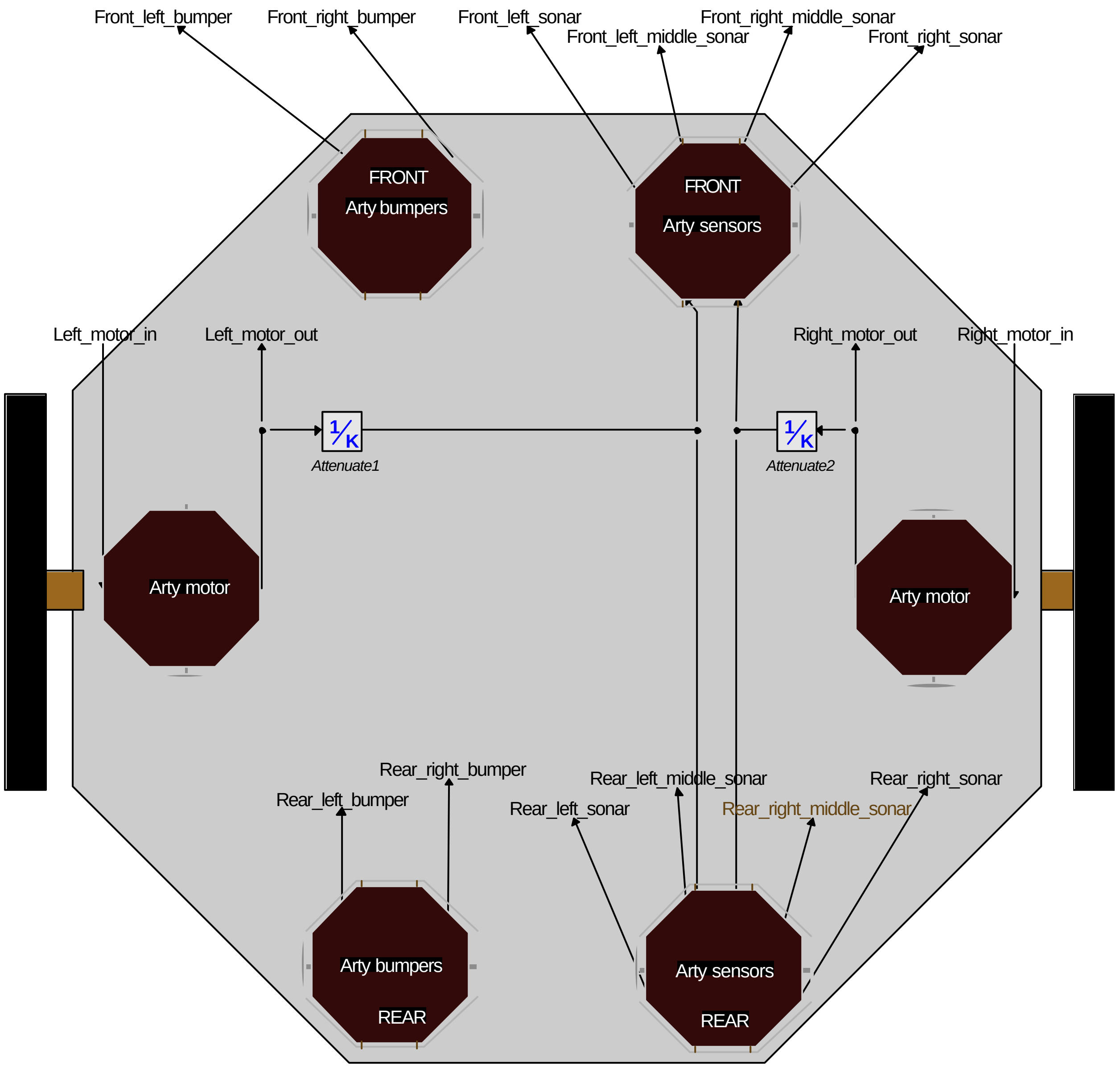


**Figure 14: Arty submodel (20-Sim drawing)**

In this figure, the labels near to the signal arrows are the submodel ports.

## Arty motor submodel

The submodel Arty motor contains a model for a Arty motor. From the documentation of Arty, it turned out that the motorspeed is controlled with a hardware PID controller. The specifications of this PID controller are unknown. Therefore, a ‘empty’ PID controller (submodel contains just a connection between input and output and no dynamics) is included into the 20-Sim model. In Figure 15 the inside of this submodel is depicted.

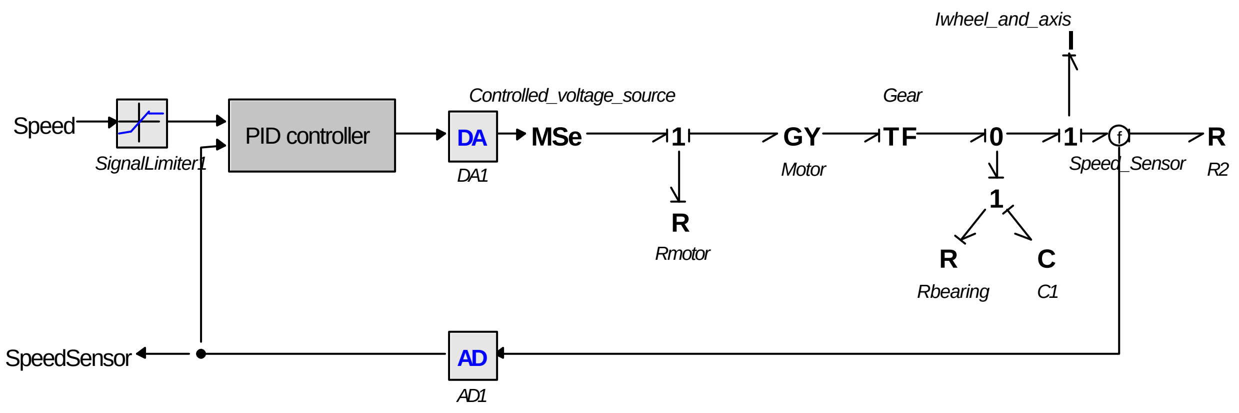


**Figure 15: Artymotor submodel**

To verify this submodel, a simulation is done with a block source connected to the speed input. See section 6.1.

## Arty bumper submodel

The Artybumper submodel simulates a bumper press. Since the bumper sensors are switches, this submodel contains only a block pulse source. During simulations, the duration of the bumper press can be set and when it will occur.

Because this submodel is so simple, both the front bumpers and both the rear bumpers are included in one submodel. See also Figure 16. Because this submodel contains only two standard submodels, verification by simulation is unnecessary.

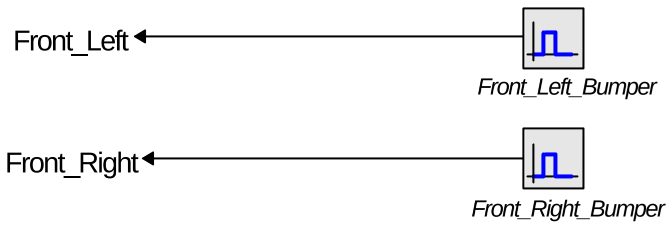


**Figure 16: Arty bumper submodel**

## Arty sonar submodel

This submodel contains a description of the behavior of the Polaroid ultrasonic sensors on Arty. The values of the ultrasonic sensors vary between about 20 and 120 (see section 4.3). A value of 20 corresponds with a distance around the 10 cm. A value of 120 corresponds with a distance of about 70 cm and more. Distances between 70 and 20 cm give values between 120 and 20. For simplicity, its assumed that this transfer is linear.

To make a model of this behavior, some data are required: the obstacle position, and the position of Arty. The position of the obstacle is modeled as a source with a constant value (ObstaclePosition). This is not a very flexible solution, but for testing purposes it satisfies. To determine the position of Arty, the speed values returned by the motors are used. As this value is a value between –127 and 127 and not a speed in

meters per second, a translation is needed. Therefore two attenuation factors 1/K are included in the simulation submodel (Figure 14).
The motor speed will be integrated to get (a estimation of) the position of Arty. The obstacle position and the position of Arty will be substracted and form, after attenuation and limiting, the sonar value. See Figure 17 for the blockdiagram of this submodel. The submodel for the sonar sensors on the rearside of Arty is a bit different. This, because these sensors are not connected to the hardware yet. Therefore, this submodel will always return the value 128 for all the four sensors. See also Figure 18.

The design of this submodel is not very flexible, because it assumes that Arty is moving forward (or backward) all the time. When Arty is turning, this submodel will not work any more because it does not take the direction other than forward or backward into account. To get a better model, this model should be extended with a rotation part, what is not yet done.
The current sonar submodel is just for testing the behavior of the control part of the model, so these restrictions do not limit the use; it's a competent model.

The testresults of this (the frontside) submodel are described in section 6.1.

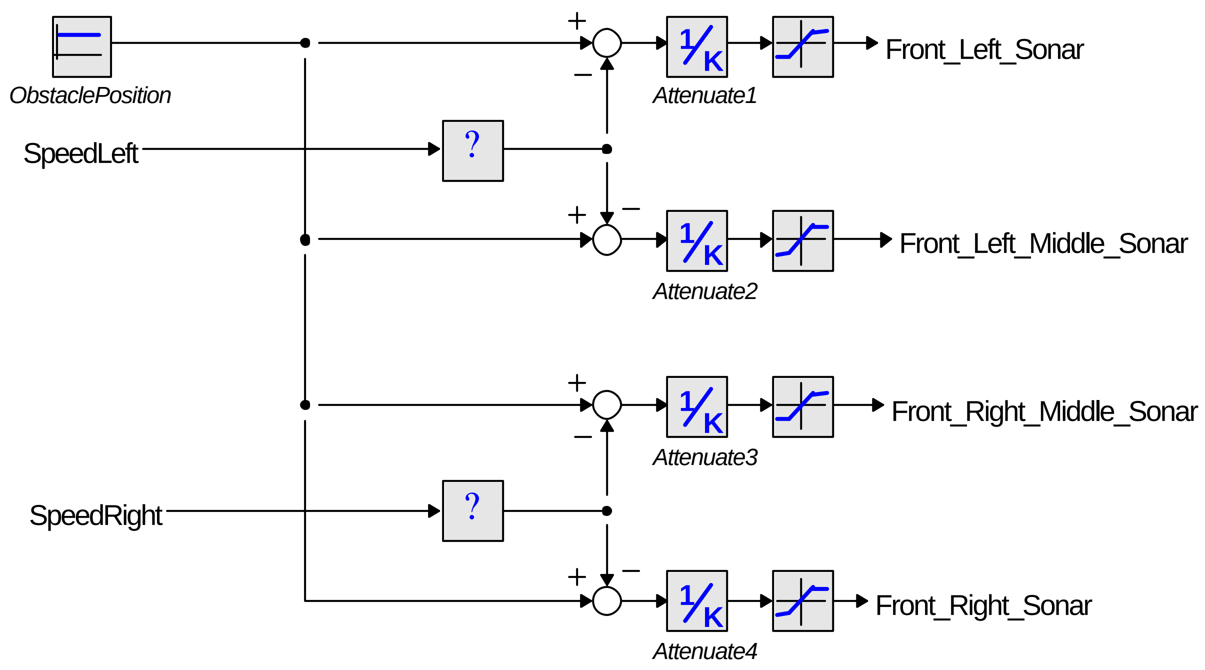


**Figure 17: Arty sonar submodel (Frontside)**

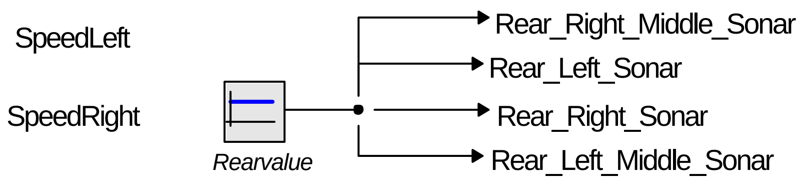


**Figure 18: Arty sonarsubmodel (Rearside)**

Now, all the submodels of the simulation part are discussed. Next, the submodels of the code generation part will be discussed.

### 5.5.3 Code generation part (Sub20Sim2Arty)

The code generation part of the Arty testmodel will be used for code generation. It contains the controller that controls the behavior of Arty. After code generation, this part has to communicate with the functions for the sensors and motors of Arty. For this purpose, the dll command, described in section 2.4, is used in the various dll call submodels. The contents of the code generation part are depicted in Figure 19. As can be seen from this figure, all the connections of this submodel to the outside world go trough a dll call submodel. Below, the controller submodel and the dll call submodel will be described in more detail.
The Sub20Sim2Arty submodel contains also four RC filters. Their use will be described in section 6.2.2

#### DLL call submodels

All connections between the simulation part and the code generation part are broken after code generation and have to be replaced by function calls. To realize this, without changing the 20-Sim model, all signals from and to the code generation part will go through a dll call submodel. This submodel establishes the connection between the functions in the template code and the code generated by 20-Sim. Each of the template functions in `artyfunc.c` has its own dll call submodel: `artyfunc_Motor`, `artyfunc_Motorspeed`, `artyfunc_ArtyBumper` and `artyfunc_ArtySonar`.
The SIDOPS code for such a submodel is given in Listing 3.

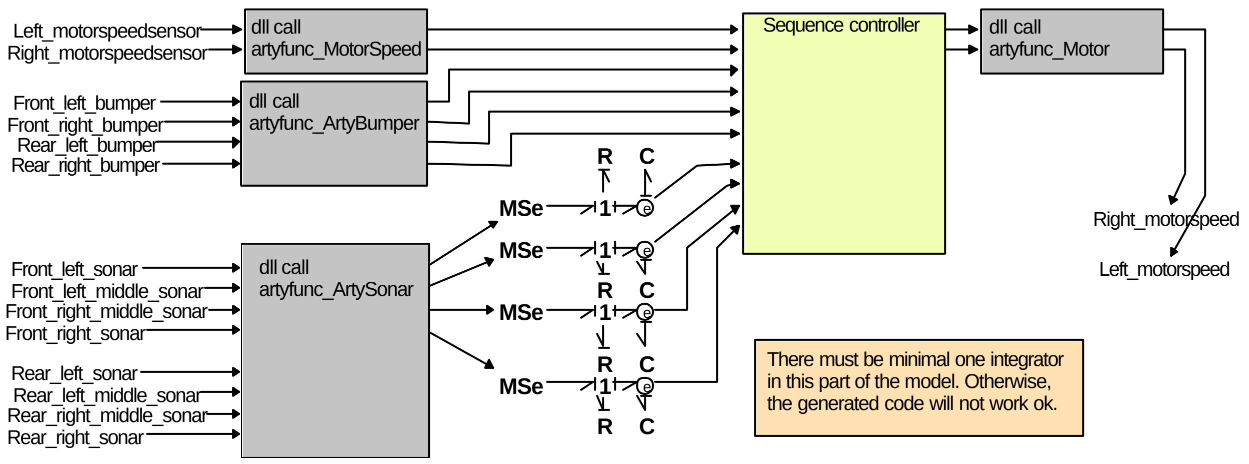


**Figure 19: Code generation part of the Arty testmodel Sub20Sim2Arty**

The first three rows of the SIDOPS code are commented because of a little bug in the code generation tool. The submodel code generation generates an error when using string variables in the dll call. This error does *not* occur when generating code for a complete model.

When simulating in 20-Sim, these submodels have to send on the input variables right to the output variables without modification. So the used dll has to do nothing but connecting the input variables to the output variables. The artyfunc.dll is described in Appendix H.

```
//parameters
//          string artyfunc = 'artyfunc.dll';
//          string amotorspeed = 'MotorSpeed';

variables
          real x[2],y[2];

equations

          x[1] = Left_in;
          x[2] = Right_in;

          //The dll  does nothing else in 20 Sim than send on the contents of
          //matrix x to matrix y
          //This method ensures that, after codegeneration, the correct function is called.
          y = dll('artyfunc.dll','MotorSpeed',x);

          Left_out = y[1];
          Right_out = y[2];
```

**Listing 3: SIDOPS code for one of the DLL call submodels**

## Sequence controller

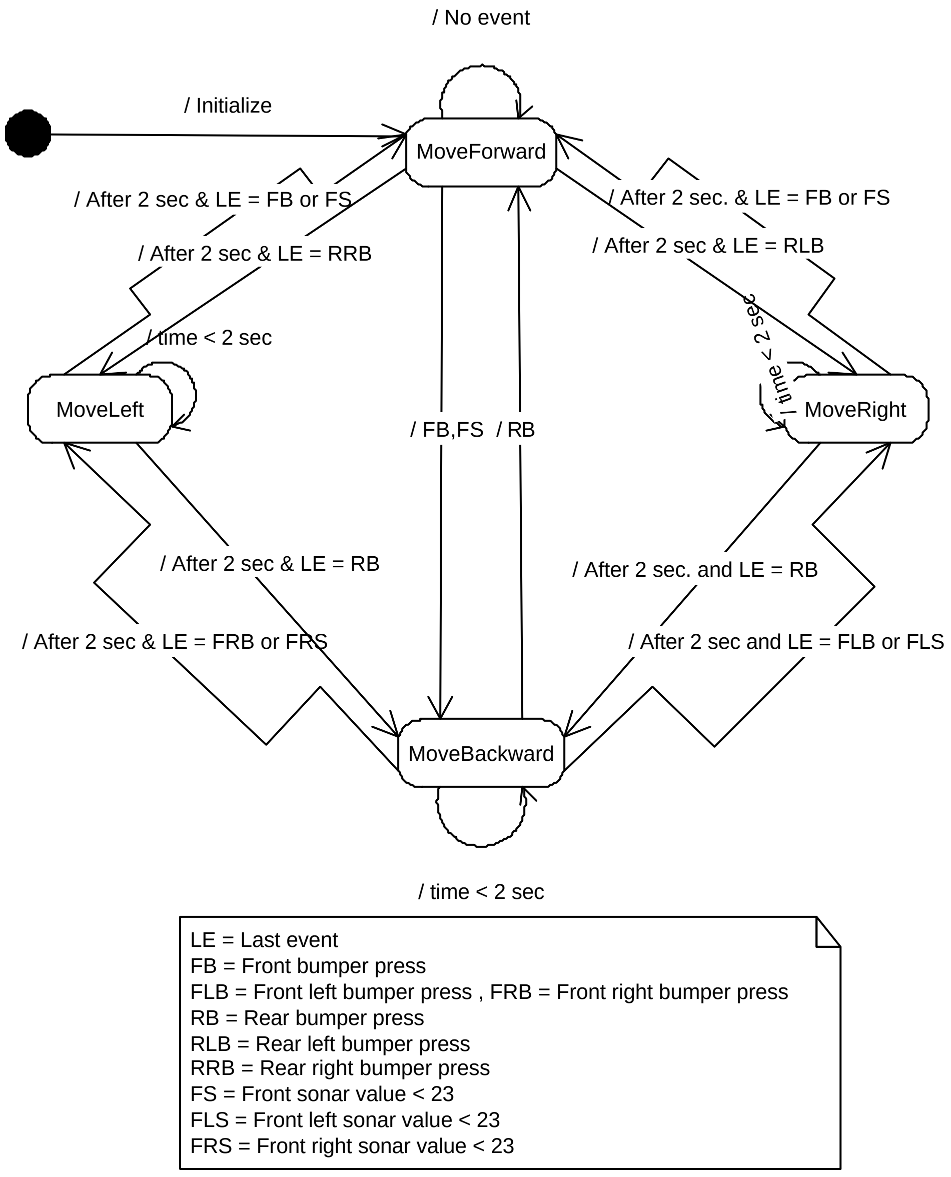
The sequence controller submodel contains the state machine that controls Arty. This is the part of the model that is important for code generation, because only the SIDOPS code in this submodel will be used for controlling Arty in real life. The sequence controller is implemented as a state machine. See Figure 20 for the statechart of this statemachine. The purpose of the "program" in this state machine is to avoid obstacles, just like the original DOS program for Arty did (Balkema *et al.*, 1999). A short description of the behavior in the statechart is given below.

On initialization, Arty begins to move forward. Arty will do this until one of the following events occurs: Front bumper press or Front sonar value < 23. When one of these events occurs, Arty moves backward for about 2 seconds and starts getting round the obstacle. When the event occurs on the left side (Front left bumper or Front left sonarsensors) Arty will turn right and otherwise Arty will turn left. After turning about 90 degrees (about 2 seconds) Arty will continue moving forward until the next event occurs. When moving backward, Arty responds to a rearbumper press and acts in the same way as for a front bumper press (of course the moving direction is inverted). The state chart does not contain a stop state because this state is redundant. When Arty is switched on, Arty has to search its way until Arty is switched off (corresponds with a infinite simulation in 20-Sim). Because switching off means power down, no stop state is required.

This description has to be build into the sequence controller submodel. Because 20-Sim has no possibilities to draw a statechart, just like a bondgraph, this controller must be implemented in SIDOPS code. It would be nice to do this with a CASE structure, such as in many programming languages, but this is not possible with SIDOPS code. The only way to realize this controller is by means of IF … THEN … ELSE commands.

Because 20-Sim is not prepared for programming a statechart, it is not possible to use a IF structure with more than 2 branches (like IF… THEN…ELSEIF…THEN…ELSE…). To accomplish this in SIDOPS code, nested IF structures are used. Another important point is that when using a IF branch, a ELSE

branch must be provided also. Otherwise, the code generation tool generates an error (simulation without a ELSE branch does not generate a error).
For the full SIDOPS code of this submodel see the 20-Sim testmodel `artymodel.em.` The testresults of this submodel are described in the next chapter.



LE = Last event
FB = Front bumper press
FLB = Front left bumper press , FRB = Front right bumper press
RB = Rear bumper press
RLB = Rear left bumper press
RRB = Rear right bumper press
FS = Front sonar value < 23
FLS = Front left sonar value < 23
FRS = Front right sonar value < 23

**Figure 20: State chart for the sequence controller**

### 5.5.4 Model dll

The testmodel needs a dll (Dynamic linking Library) when using the 20-Sim simulator. This dll must connect the input variables to the output variables.
Two types of 20-Sim dll are available, a static DLL and a dynamic DLL. A static dll is used here, since there are no statevariables involved in the dll C code (See also the 20-Sim helpfiles).

Almost every compiler that can build Windows applications, can also build dll-files containing dll-functions. The dll written for the testmodel is written in Microsoft Visual C++ 6.0. In general the dll-function called by the simulator has the following syntax in 20-sim:

```
Y = dll(filename,functionname,X);
```

When the 20-Sim simulator find this SIDOPS+ function, it will call the appropriate function in the user defined dll. The user-function in the dll must have the following prototype:

```
int myFunction(double *inarr, int inputs, double *outarr, int
                                              outputs, int major)
```

where

- **myFunction**: name of the function to be called from SIDOPS+ code. Corresponds with the name given to the stringvalue `functionname`.
- **inarr**: pointer to an input array of doubles, corresponding with `X` in the dll call. The size of this array is given by the second argument.
- **inputs**: size of the input array of doubles.
- **outarr**: pointer to an output array of doubles, corresponding with `Y` in the dll call. The size of this array is given by the fourth argument.
- **outputs**: size of the output array of doubles.
- **major**: boolean which is 1 if the integration method is performing a major integration step, and 0 in the other cases. For example Runge-Kutta 4 method only has one in four model evaluations a major step.
- **returnvalue** : 0 is success, otherwise error. An error causes the simulation to stop.

Besides the user-functions, the dll can contain also some additional functions for initialization and termination of the model. These functions are not used for the artyfunc dll because they are not necessary. For more detailed information about dlls, see the 20-Sim helpfiles and for the `artyfunc.dll` Appendix H.

# 6 Testing

In this chapter, the tests of the used 20-Sim model and the written template are described. The tests of the testmodel are described in section 6.1. This will be done on basis of the simulations of the various subsystems and some simulations of the full model. After this, the tests for the written template are described in section 6.2.

## 6.1 Testing of the testmodel

The tests of the testmodel will be described via the simulations of the various submodels, starting with the simulation part of the testmodel. Each of the simulations is done to verify if the various submodel are working.

### 6.1.1 Verification of the Artymotor submodel

This submodel is simulated by connecting a block pulse source to the Speed port and measuring the most important signals. The results of this simulation are depicted in Figure 21. One can see from this figure that this submodel is working. This submodel may not describe exactly the behavior of a Artymotor because the various model parameters are not validated by measuring the real parameters. For the purpose of the code generation part, this is not of concern.

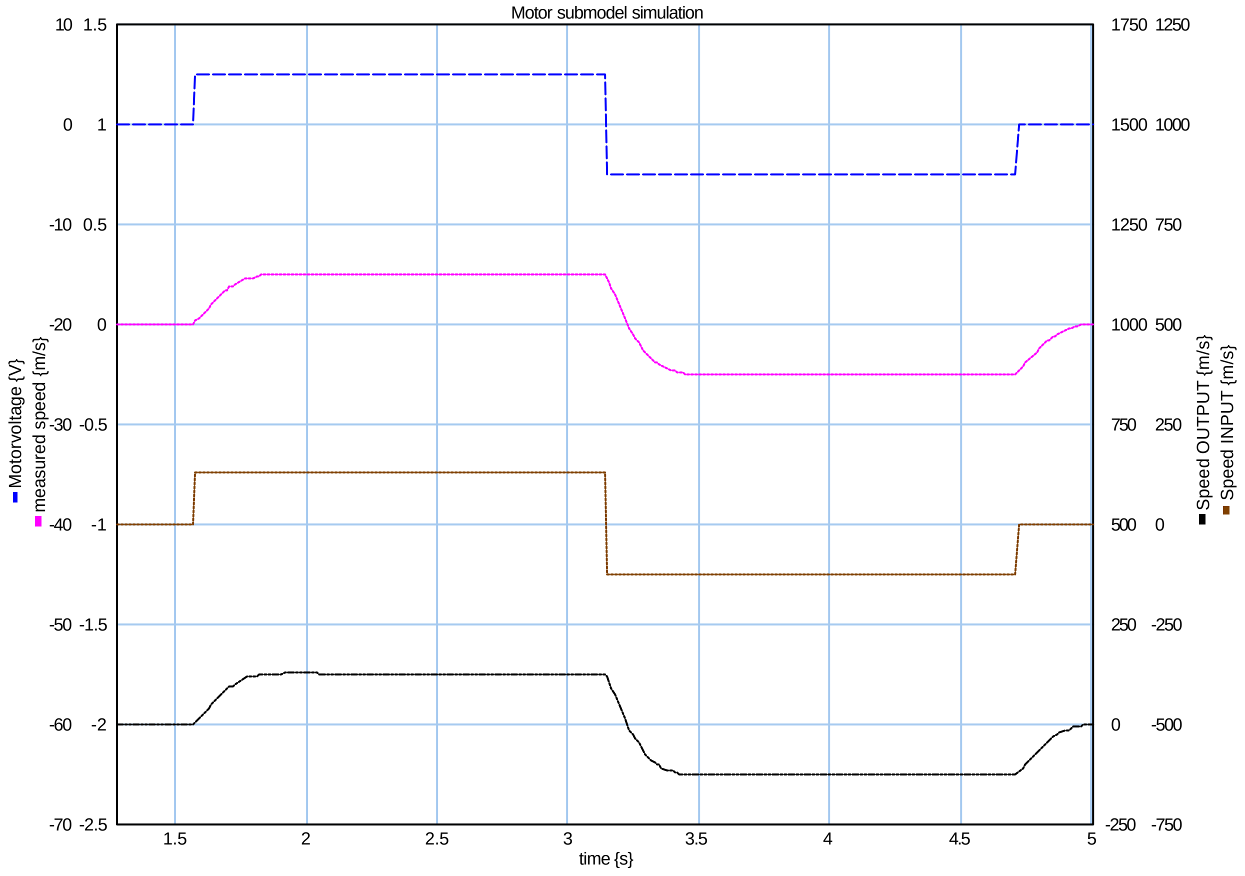


**Figure 21: Simulation result of the Artymotor submodel. From top to bottom: motorvoltage, measured speed, speed output and speed input**

### 6.1.2 Verification of the Artybumper submodel

Because this submodel consists of two standard submodels (Signalgenerator-pulse) it is not necessary to check it by means of a dedicated simulation. In Figure 24 it is shown that this submodel is working.

### 6.1.3 Verification of the Artysonar submodel

To verify the Artysonar submodel, the complete simulation part of the Arty testmodel was used. Both motors were connected to one block pulse source. So in the simulations, Arty moves only forward or backward. When simulating this model, the results depicted in Figure 22 are obtained. From this figure one can see that when the distance to the obstacle (in the simulation set on 1 meter) drops below 0.71 meters (t=1.3), the sonarvalue begins to decrease. When the distance drops below 11 centimeters, the sonarvalue has reached it's minimum value of 20 (t=7). This submodel works for forward and backward movement. When Arty is turning, this submodel does not work any more. See Figure 23. In this simulation, the left motor is turning forward and the other is turning backward. This causes Arty to turn around its axis. One can see from this simulation that the calculation of the obstacle distance for the right sensors is increasing, while the distance for the left sensors is decreasing. In reality, this distance will not change much. So this submodel does not work for turning, as already indicated in section 5.5.2.

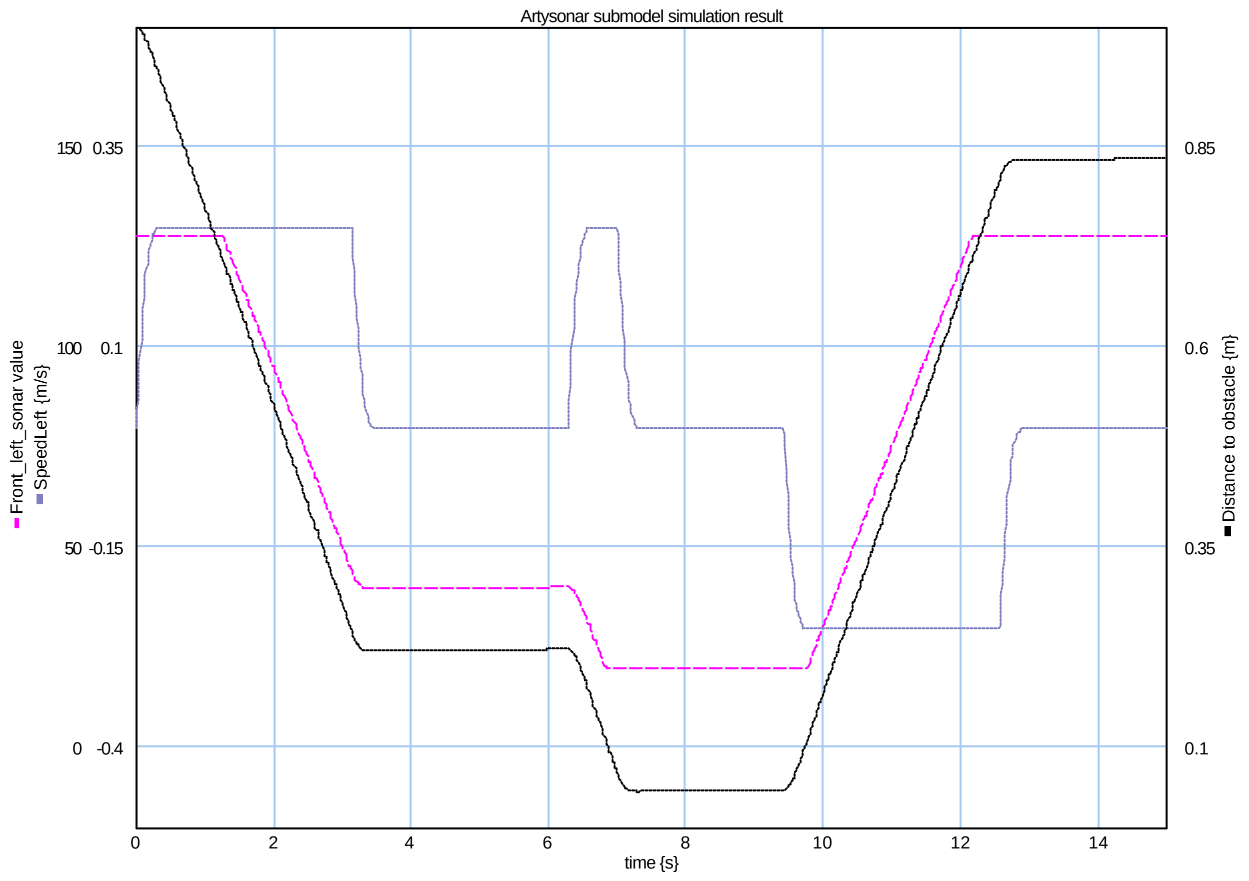


**Figure 22: Artysonar submodel test. From top to bottom: Front_Left_Sonar value, Distance to obstacle, SpeedLeft.**

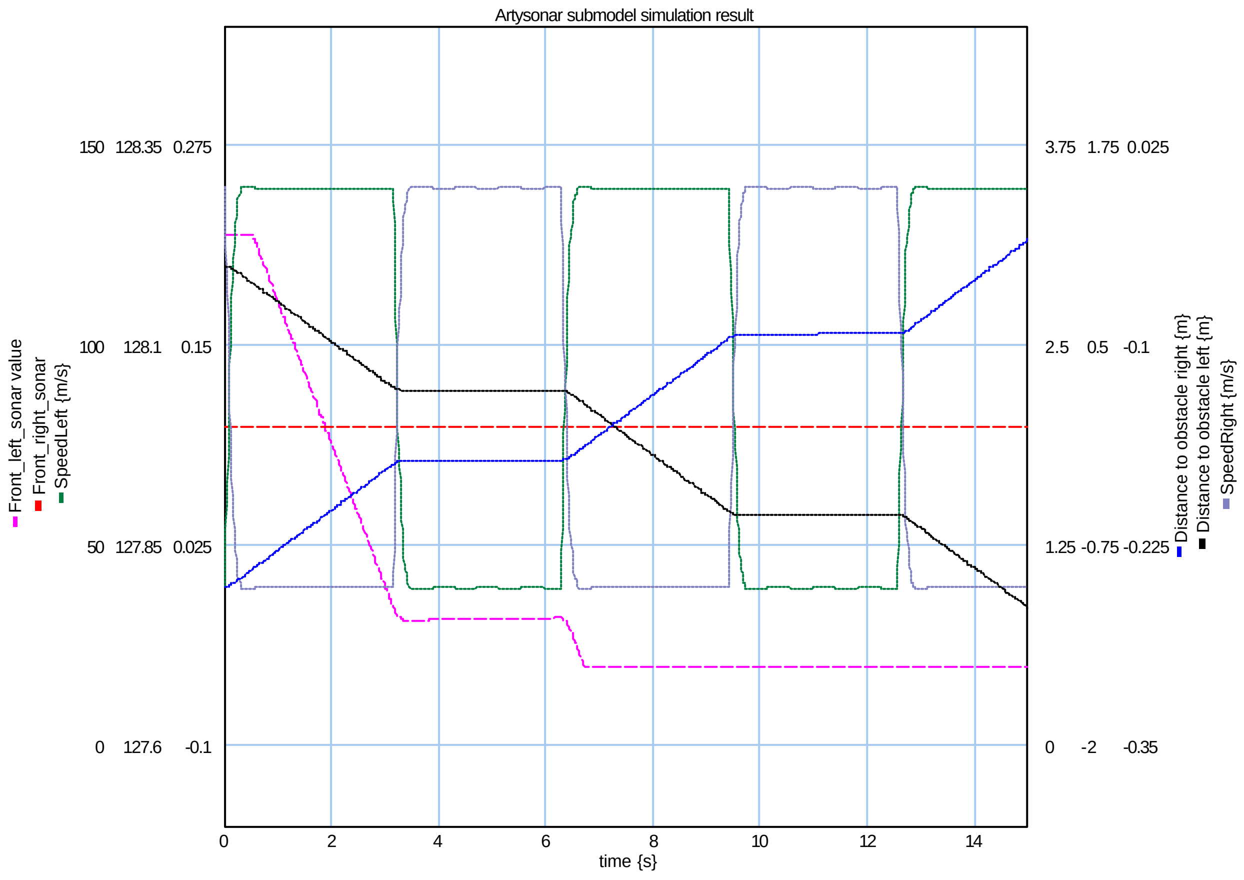


**Figure 23: Simulation of the Arty sonar submodel when Arty is turning around its axis. From top to bottom: Speed left and Speed right, Front left sonar value, Distance to obstacle left, Front right sonar value and Distance to obstacle right.**

### 6.1.4 Sequence controller submodel

To validate the correct working of the sequence controller submodel, the various input signals (events) have to be simulated. This can be done by connecting SignalGenerators to the various input port and checking the response of the sequence controller to these signals. These results would be almost the same as when the sequence controller submodel is tested by means of the complete Arty testmodel. Therefore, this submodel is at once tested in the complete model. See the next section for the results.

### 6.1.5 Validation of the complete testmodel

Because the complete testmodel is relatively big, it is almost impossible to validate the correct working with only one simulation. The result would be a long simulation and the output would contain many different signals. Therefore the complete model is checked by simulating the different events one-by-one.

#### Bumperpress event

To check the model for the bumperpresses, some simulations are done. The first simulation is for both bumpers on the frontside of Arty. See Figure 24 for the results.

The most upper line in this figure is the state of the statemachine. It should start with a value of 1 (Move forward) but it start with 2 (Move backward). So there is somewhere an error in the sequence controller code. This error proves to be very difficult to find and, therefore, is not corrected yet. This error has only consequences for the beginstate., the rest of the time this error has no consequences for the working of the sequence controller. So this error is no problem for verification of the correct working of the sequence controller, only the beginstate is not as expected.

Looking at Figure 24, one can see that when Arty is moving forward (state = 1) and a Front-left-bumperpress occurs, Arty moves backward (state = 2) for about 2 seconds (t=0..2) and then turns right (2 second only the leftmotor at maximum speed, state = 4, t=2..4). The reaction of the sequence controller to the right bumper (t=10) is also according to the statechart. So the reaction on both the bumpers on the frontside is good. Now the backside bumpers. These bumpers only work when Arty is moving backward, so now the beginstate error can be used as a beautiful starting point. See Figure 25 for the simulation results for rear bumper presses.

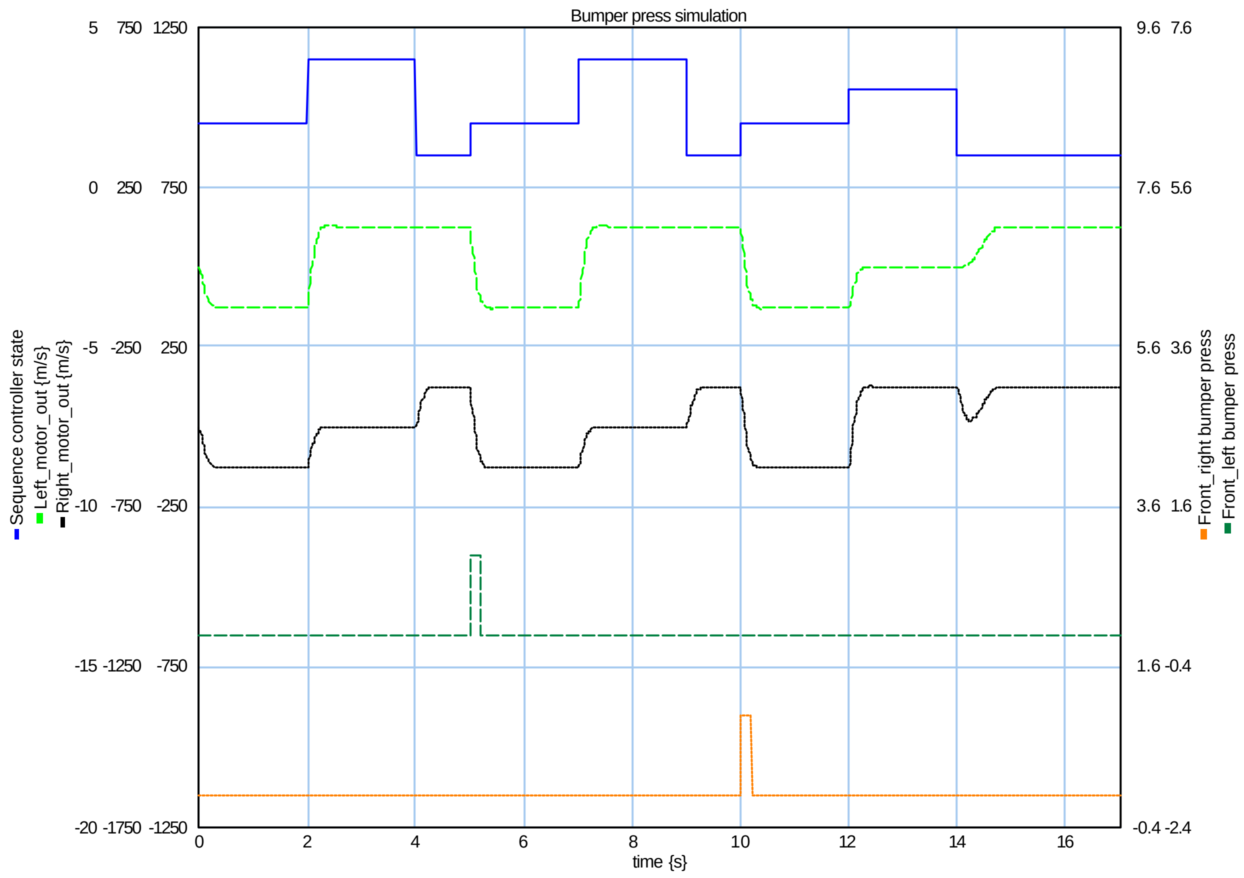


**Figure 24: Testing the sequence controller with front bumper presses. From top to bottom: Sequence controller state, Left motor output, Right motor output, Front right bumper and Front left bumper.**

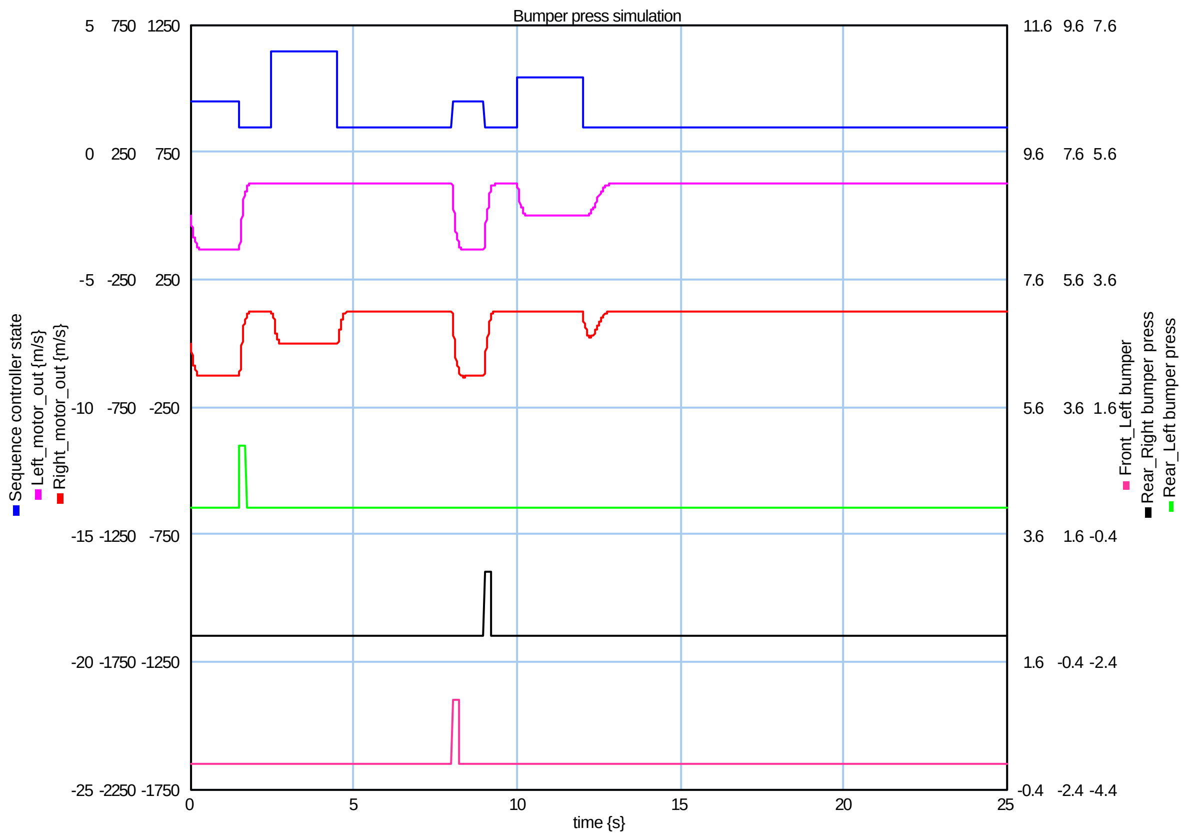


**Figure 25: Testing the rear bumpers: From top to bottom: Sequence controller state, Left motor output, Right motor output, Fromt left bumper, Rear right bumper and Rear left bumper.**

After 1 second, the rear-left-bumper is pressed and, as one can see from the figure, Arty will move forward for 1 second and then turn right for 2 seconds. After this Arty will again move forward. (1 second is chosen here for the purpose of maneuvering out of the awkward position that Arty stand between two obstacles (one on the frontside and one on the backside) that are placed very close together. It is visible in this figure that the reaction to the rear-right-bumper is also ok. (To get Arty again in the Move_Backward state (2) , the front-left-bumper is pressed at t=8).

### Low sonarvalue event

To test the response of the sequence generator to a low sonarvalue (less than 23), nothing special had to be done. Only the bumperpress events had to be switched off by means of changing the amplitude to 0. Further, one point for attention is the fact that the artysonar submodel does not work good when Arty is turning around its axis. See Figure 26 for the result. Because the beginstate is still 2 (Move Backward), Arty will turn and the response of the left and right sonar sensors will be different. Looking at the figure, one can see that around 7.6 seconds, the left sonarvalue drops below 23 and Arty starts to move backward. So the principle of responding to the sonar event is working, only the testing in this model is a bit difficult.

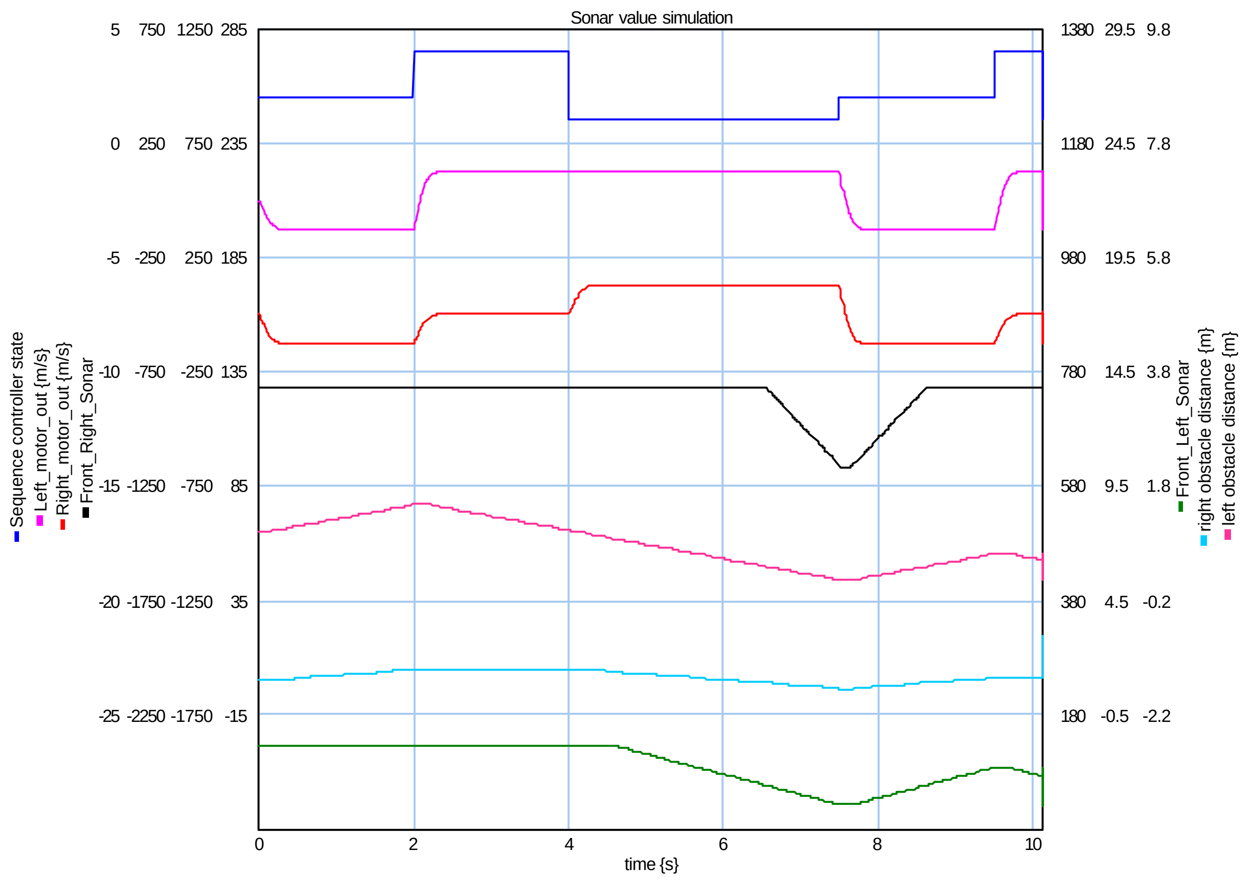


**Figure 26: Simulation results for the sonarevent test: From top to bottom: Sequence controller state, Left motor output, right motor output, Front right sonar, Front left sonar, Right obstacle distance and Left obstacle distance.**

## 6.2 Testing the template

The goal of this assignment was, to use the 20-Sim model to generate code that can be run on Arty. To see if the written template is working ok, the template is tested various times during the development by means of a very simple model with only a block source, a resistor and a capacitor connected to one of the wheels. This is done to make sure that the various parts of the template are working ok. This testing is partly done by using the Microsoft Visual C++ debugger in combination with Phar Lap to debug the template. So before testing the template with the testmodel, the template was already working with simple models.

### 6.2.1 Test setup

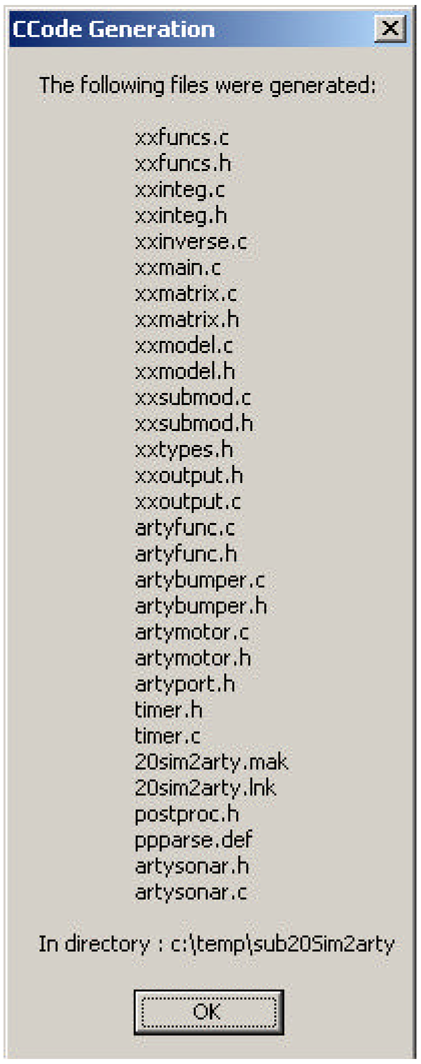
To test the template and the 20-Sim postprocessor, some preparations had to be done. First, the postprocessor (pc104pp.exe) has to stand somewhere where Windows can find it (e.g. the Windows directory). Second, the parameters in the postproc.ini file (like paths, compilers etcetera) must be set.
Third, Arty must be connected to the host pc by means of the parallel and serial cable and Arty has to be booted from a Phar Lap ETS monitor bootfloppy. And last, the PC 104 template has to be added to the target configuration file. After this preparations the code generation and postprocessing can be used.

To generate the code for the Arty testmodel, the simulator is started and the “Generate C-code” tool is selected from the tools menu. After selecting the correct template (PC 104 board submodel), the correct submodel (sub20sim2arty) and the output path, the code generation can be started.

### 6.2.2 Test results

After clicking Ok in the code generation dialog box, the code generator started with processing the model and generation of the source code. This gave no problems (Figure 27). Next, the postprocessor was started and generated a new postproc.h file. After that the postprocessor compiled the code and started the executable on Arty. So the code generation on its own is working.

To see if the generated code for the sequence controller is also working ok, Arty was placed between some obstacles and rerunning the generated code lead to the following result:
The reactions on the sonar sensors were a bit to strong, because of heavy fluctuations in the sonarvalues. To solve this some low pass (RC) filters were placed between the sequence controller and the sonarsensors (See Figure 19). (Better would be to use a integrator for this purpose…)
The response after this change was much better but still a bit to strong. Without the sonar sensors, the generated code is working perfectly. The response on bumperpresses is exactly as modeled. The response on sonarevents is reasonable with respect to the expected reaction. So the reaction on sonarevents could be improved in the future.


CCode Generation

The following files were generated:

xxfuncs.c
xxfuncs.h
xxinteg.c
xxinteg.h
xxinverse.c
xxmain.c
xxmatrix.c
xxmatrix.h
xxmodel.c
xxmodel.h
xxsubmod.c
xxsubmod.h
xxtypes.h
xxoutput.h
xxoutput.c
artyfunc.c
artyfunc.h
artybumper.c
artybumper.h
artymotor.c
artymotor.h
artyport.h
timer.h
timer.c
20sim2arty.mak
20sim2arty.lnk
postproc.h
ppparse.def
artysonar.h
artysonar.c

In directory : c:\temp\sub20Sim2arty

**Figure 27: Code generation is complete**

With respect to the real-time behavior, the following can be said: The generated code runs real-time on Arty when the stepsize is minimal 0.01 second. Using this stepsize, the message “`Stepsize too small! Code is not running fast enough!`” (see Figure 10) should not appear in the console window on the host machine. A smaller stepsize results in many of these warning messages, so 0.01 second is really the minimum value for the stepsize. This corresponds with 100 model calculations in 1 second, more than enough for controlling Arty.
So it is possible to run the code, generated of a relatively large 20-Sim model, real-time.

### 6.2.3 Performance test DOS/ETS

One of the major questions with respect to the Phar Lap ETS kernel was: “Is the ETS kernel faster or slower than DOS as OS”. This is very interesting to know, because almost none of the special features of the ETS kernel is used. To make a comparison between DOS and the ETS kernel, the generated code of the testmodel (without sonar response) was used. The code was adapted in such a way that it would run also under DOS, when compiling with a DOS C++ compiler. To accomplish this, the timer interrupt part had to be removed, because it uses the ETS kernel. (The other alternative, writing a DOS timer interrupt handler, costs too much time).

The next thing to do, was including some commands for measurement of the elapsed time. For this purpose, the ANSI C command `clock` was used. This command returns the number of clockticks since the execution of the program started. To measure the speed, the number of clockticks between the beginning of the execution and the end were measured. The used model, running 10 seconds, is calculated without looking to the stepsize (no pauses, continuous calculations).

The results of the calculations are written to the screen with `printf` commands. Because Arty has no screen this would be a problem (under Phar Lap, all the screen output is directed to the console window on the hostmachine). The solution for this problem is using the supplied terminal cable with the TVTerm program. With this program it is possible to view the screen output of Arty on another PC.

The results of the calculations are given below in Table 1 and Table 2.

## Phar Lap

| **Test 1 (fullscreen DOS with printf under Windows NT)** | | |
|---|---|---|
| **Attempt** | **Clockticks (1000 per second)** | **Elapsed time (sec)** |
| 1 | 7470 | 7.47 |
| 2 | 7490 | 7.49 |
| 3 | 7420 | 7.42 |
| 4 | 7590 | 7.59 |
| 5 | 7480 | 7.48 |
| | | |
| **Test 2 (normal DOS window with printf under Windows NT)** | | |
| **Attempt** | **Clockticks (1000 per second)** | **Elapsed time (sec)** |
| 1 | 5680 | 5.68 |
| 2 | 5740 | 5.74 |
| 3 | 5820 | 5.82 |
| 4 | 5820 | 5.82 |
| 5 | 5810 | 5.81 |
| 6 (100% processor load on the host machine) | 6920 | 6.92 |
| | | |
| **Test 3 (without printf output to the screen)** | | |
| **Attempt** | **Clockticks (1000 per second)** | **Elapsed time (sec)** |
| 1 | 610 | 0.610 |
| 2 | 611 | 0.611 |
| 3 | 611 | 0.611 |
| 4 | 610 | 0.610 |
| 5 | 611 | 0.611 |

**Table 1: Results for the speedtest with the ETS kernel as OS**

## DOS version

| **Test 1 (with printf output to the screen)** Full screen via TVTerm | | |
|---|---|---|
| **Attempt** | **Clockticks (18.2 per second)** | **Elapsed time (sec)** |
| 1 | 252 | 13.80 |
| 2 | 251 | 13.79 |
| 3 | 251 | 13.79 |
| 4 | 253 | 13.90 |
| 5 | 252 | 13.80 |
| | | |
| **Test 2 (normal DOS window under Windows NT)** | | |
| Not possible | | |
| | | |
| **Test 3 (without printf output)** Full screen via TVTerm | | |
| **Attempt** | **Clockticks (18.2 per second)** | **Elapsed time (sec)** |
| 1 | 14 | 0.77 |
| 2 | 14 | 0.77 |
| 3 | 13 | 0.71 |
| 4 | 13 | 0.71 |
| 5 | 14 | 0.77 |

**Table 2: Results for the speedtest under DOS**

Comparing these data with each other, one can see that the same code is executing faster under Phar Lap's ETS kernel than under DOS. The code used for the first test, with printf output to the screen (the elapsed time is printed) cannot even be used for a real-time program under DOS. Even without any timing code, the code is already running too slow. (10 seconds of the model, with a stepsize of 0.01 second, execute in almost 14 seconds).
Another important discovery is that the speed of the execution of the Phar Lap code on Arty is slightly dependent on the way of displaying the return-information on the screen. When running the code with RUNEMB from within a DOS box, the code executes a bit faster than in fullscreen mode.
A bit strange is the result of Phar Lap test 2, attempt 6. When the processor load on the host machine running Windows NT is 100%, the code on Arty executes slower. Obviously the ETS monitor, that communicates with the host machine during runtime, waits until the host machine has processed the received data. This is not very handy when writing a program that must run real-time. On the other hand, the Lite version of the TNT toolsuite is in principle for testing purposes, because it requires always a host-target connection.

Some notes have to be made with respect to this test:

- The number of clockticks per second differs under the ETS kernel and under DOS. To verify the calculated run time, a stopwatch was used to see if this time is correct. The results of this rough verification are that the results in the tables above seemed to be correct.
- Test 2 (running in a DOS window) is, of course, not possible under plain DOS. Running TVTerm under Windows NT would be a alternative, but this was not possible because TVTerm does not run under Windows NT.

# 7 Discussion

## 7.1 Introduction

This chapter contains a discussion of the findings of the Phar Lap toolsuite and the code generation tool.

## 7.2 20-Sim code generation

The 20-Sim code generation tool is very flexible with respect to different target and it's easy to add new templates. The user can generate code for a complete model or for only a submodel by choosing a template.
20-Sim has already some templates, from which the two general purpose ANSI C template form a good startpoint are for writing other templates.
The 20-Sim code generation tool reads a target configuration file with target specific information and, after selection of a target, 20-Sim will read all the template files and replaces the tokens in these files with the model specific information/code. It is possible to use a pre- or postprocessor that executes that performs additional tasks before/after code generation. For calling hardware specific functions (such as motorcontrol) from within 20-Sim, 20-Sim has a build-in SIDOPS command dll that can be used for this goal. This is only a trick that implements what is needed. There are also other possibilities to do this:

- By two different pieces of SIDOPS code for one submodel. One for simulation purposes and one for code generation.
- By a 20-Sim tool like the Linear systems editor that deals with the calling hardware specific functions after code generation.
- By a postprocessor that inserts some code to the generated files that connects a variable to a function.

During the project, some important limitations of the code generation tool were discovered:

- When a model has a continuous time part and a discrete time part, code generation is currently not possible.
- For code generation, a `if` statement must have an `else` branch, otherwise code generation generates an error (the simulator does not generate an error).
- A model must contain at least one state; otherwise, the generated code will not work with the existing templates.
- The generated code contains many different variables that correspond with one and the same value. Two submodels that are connected to each other and therefore have common port variables. After code generation, each connection between two submodels corresponds with a variable assignment like `V[2] = V[1]`. 20-Sim contains some processing options to remove these redundant equations, but these optimizations do *not* work when generating code. These optimizations cause incorrect array sizes.
- The 20-Sim code generation tool is based upon the simulation execution model. The simulation execution model has no real-time facilities, because for simulations, this is not necessary. To use

the generated code for a real-time program, the execution model in the templates had to be changed in such a way that real-time processing was possible. This could give some troubles with time critical applications. In discrete systems, values are only available at certain points in time. When simulating, this gives no problems, but when using the same execution model in real life one may run into troubles.

## 7.3 Phar Lap TNT Toolsuite

The Phar Lap Toolsuite Lite is not handy with respect to Arty, because Arty needs a extern floppydrive with power supply and two host-target cables. For a mobile robot, this is not acceptable. So, the full version of the Phar Lap TNT toolsuite is required for Arty to move freely. For the purpose of this project, writing a 20-Sim template, the Lite version was sufficient.

### 7.3.1 Comparison between Phar Lap and DOS

In this section, Phar Lap's Toolsuite Lite will be compared with DOS, because the existing software for Arty is a C++ program running under DOS (Balkema *et al.*, 1999).

Many differences exist between Phar Lap and DOS. DOS is a complete OS for normal tasks and Phar Lap's ETS kernel is in principle a mini OS dedicated to real-time programs. More advantages of the Phar Lap kernel are:

- Standard in protected mode (no 640 kB barrier)
- The kernel is customizable
- Less overhead (no DOS interrupt 21h and others)
- Boottime is lower (in comparison with DOS 6.2)
- real-time processing support

Besides this advantages, there are some more advantages with respect to software development under Phar Lap:

- Software development and testing for Arty can be done on a Windows machine with Microsoft Visual C++.
- Many APIs are available for faster development.
- Using download cables and the ETS monitor, floppies are not necessary for transporting the compiled program from the development machine to Arty every time something has to be tested.
- Running and debugging on Arty can be done from within the Microsoft Visual C++ IDE. The screen output of the embedded program appears on the host machine (when no standard video adapter is available), the variable content is available and it is also possible to see the memory content.

Phar Lap's Toolsuite has also some disadvantages. The Lite version in less flexible than DOS with respect to the necessary cables and Phar Lap programs must be written in C/C++. The complete Toolsuite uses no cables, so this version is a more flexible.

# 8 Conclusion and recommendations

## 8.1 Conclusion

The code generation from 20-Sim to the PC/104 target board in Arty is working. The written template contains hardware code for most of the Arty specific hardware and runs real-time for a stepsize >= 0.01 second. See section 6.2.2

The template uses the ETS kernel of the Phar Lap TNT toolsuite lite. Testresults show that the generated code runs faster under the ETS kernel than under DOS. So despite the host target connection with cables, the code runs faster under Phar Lap. See section 6.2.3

The generated code is relatively large and contains much overhead due to not-used functions. For decreasing this overhead, a postprocessor was written that decreases the code size by instructing the compiler which code to compile and which code not. This task of the postprocessor works good and enables a executable size reduction of almost 25 percent. Besides this task, the postprocessor compiles the code automatically and starts the execution of the code on Arty. So to run the 20-Sim sequence controller on Arty, starting the code generation is enough.

The 20-Sim testmodel contains a global description of the dynamics of Arty. Enough for testing the control part of the model. The 20-Sim sequence controller, designed for controlling Arty, works partly good. Only the response to sonar events is a bit unstable (section 6.2.2). The response to the rest of the sensors is good.

The behavior of Arty comes close to the behavior described by the second D1 projectgroup that developed the original controller software (Balkema *et al.*, 1999). The only behavior, not included in the 20-Sim model is beacon searching. The beacon sensor is not implemented in the template and the 20-Sim model, because the beacon was not available and the modeling of the response to the beacon sensor is difficult to do.

The overall conclusion for this project is, that the assignment for writing a code template for a PC/104 target is completed successfully.

## 8.2 Recommendations

The written template, the testmodel and the 20-Sim code generation tool are working, but all three are far from perfect. Below, some recommendations are given for improvements.

### 8.2.1 20-sim

The code generation tool on its own works rather good. Besides from some minor bugs, it is not difficult to use it. Some recommendations for future versions of this tool:

1. The place to find this tool in 20-Sim (in the simulator) *not* ok. It should be placed near the Simulation tool in the main 20-Sim window, because when looking at Figure 4, code generation and simulation are two different ways for processing a model. Code generation is independent of simulation.
2. The Runge Kutta routines in the existing templates fail to work correctly when a model has no states. This causes the generated code to stay in a endless loop. This should be corrected.

3. The order for the variables in the generated code for a dll call is not logical with respect to the order used in dll's. These orders should be equal.
4. Sometimes, when using the code generation tool, an error message with the text "An error has occurred. Please restart 20-Sim as soon as possible" will popup. When this occurs, the users knows, something is wrong with the model, but not what. For example when using string values in a dll call (only with submodel code generation) and when an ELSE branch lacks. The check for errors in the model should be extended and giving a error number, or even better, a clear error message.
5. Using tokens in the targets.ini file does not work with this version of the code generation tool. This is the only way to tell a postprocessor (e.g. a compiler) where the generated code files are. This should be corrected in future versions.
6. For this project, a postprocessor was written to reduce the overhead of the generated code by leaving out the not-used functions. 20-Sim can do that on its own. Do not copy functions from the template to the generated files that are not used by the SIDOPS code, such as most of the XXMatrix functions.
7. The principle for calling target specific functions from the generated code by means of the dll command works, but this a workaround route. The most easy way to do this, is by means of SIDOPS+ code. A solution would be that it is possible to specify for each submodel two different blocks of SIDOPS+ code, one that will be used when simulating the code and another when generating code. These blocks could be separated by special comment lines (just like the first comment line by a submodel generated by the Lineair Systems Editor). An other solution could be a tool, just like the Lineair Systems Editor, for specifying which variables should be connected to a template function after code generation.
8. The generated code contains many different variables that correspond with one and the same value. Two submodels that are connected to each other and therefore have common port variables. After code generation, each connection between two submodels corresponds with a variable assignment like `V[2] = V[1]`. Some optimization could be done to reduce the number variables and the number of lines with these assignments in the generated code, because each redundant assignment costs CPU time.

   20-Sim contains already some processing options to remove these redundant equations, but these optimizations do *not* work when generating code. The optimizations cause incorrect array sizes.

Some other things that should be implemented in 20-Sim:

1. An undo function in 20-Sim. Implementation of a undo function in 20-Sim would make 20-Sim more user-friendly. It is very annoying that it is not possible to undo changes made to a graph model or to the SIDOPS code.
2. It is very difficult to write a statemachine in 20-Sim with only IF and ELSE commands. Easier would be drawing a statemachine in a new tool called "State machine editor", just like the linear systems editor. Also a command like CASE would be very handy for this purpose. A disadvantage of this, could be the fact that this requires a more extensive error check, because in every case all the output and state variables in the submodel must get a value.
3. Replacing the explorer window for submodel browsing, with a docking window within the 20-Sim main window. Just like in Matlab Simulink 6.

4. When using 20-Sim simulations in a report like this, it is possible to select different line styles, but the in the legend, only the colors are shown, not the linestyles. This should be implemented, because otherwise it is almost impossible to see on a black-white print which line corresponds with a signal.

### 8.2.2 Template

The written template is not complete. For this project, the first step was, getting a working template. Therefore, not all sensors of Arty are implemented in the current template. The template has to be extended with code for the beacon sensor (and motor). Further, the template is written by extending the Ansi C submodel template. The template contains probably some unnecessary code that could be removed. The current template is very big and uses almost no special ETS functions.

### 8.2.3 Phar Lap

For the purpose of this project, the Phar Lap TNT toolsuite Lite was not suitable. The major disadvantages of this toolsuite (lite) are the host-target connection with cables, floppyboot and no support for ROMming. All these features are necessary for a mobile robot and not available in the Lite version.
Further, this Lite version is not suitable for real-time systems, because the host machine influences the speed of the target. This toolsuite is only interesting when the full version is available. The Lite version is only for testing purposes. Only for educational purposes and testing, this version satisfies.
The only positive feature of the Phar Lap toolsuite lite is the debugging facility from within Visual C++. The Phar Lap TNT toolsuite is only useful when writing complex programs that use non standard things like multithreading, TCP/IP connections and a webserver. So when Arty has a webcam, this toolsuite could be useful.
Before buying the full Toolsuite it would be useful to make a comparison between the features of this toolsuite and other real-time OSses like real-time Linux.

### 8.2.4 Testmodel

The used testmodel for this project is for testing purposes a competent model. It contains everything, needed for testing the sequence controller, but the model is far from perfect. The dynamic behavior of Arty does not correspond perfectly to the real world, because none of the parameters were measured and possible not all the dynamic behavior is modeled. Especially the Artysonar submodel is not complete. This submodel works only good when Arty does not turn. This could be improved by measuring the rotation (for instance by determining a angle) and a better definition for the place of an obstacle (not only a distance, but maybe coordinates).
The sequence controller could be improved as well. For example, the controller responds only to the rear bumpers when Arty is driving backward.

# Appendix A: How to install 20-Sim 3.106b

The following steps are required to upgrade 20-Sim 3.105 to version 3.106b.

1. Ensure that 20-Sim 3.105 is installed. (Look at the Help -> About dialog)
2. Open the zipfile 31.06beta.zip and extract the files to the 20-Sim directory.
3. Copy the file 20simext31.dll from the directory \BIN\IN SYSTEM32\ (can be found in the 20-Sim directory) to the Windows system directory (SYSTEM under Windows 9x and SYSTEM32 under Windows NT/2000)

# Appendix B: Installation of the Phar Lap TNT Toolsuite Lite

This appendix describes some points to take care of while installing the TNT Toolsuite. It does not describe the whole installation procedure already given in Grehan (Grehan *et al.*, 1998 pages 64-74)

To get started with the Toolsuite Lite, two computers are needed. One development host and the other is the embedded target (Arty). The requirements for the development host are minimal a Intel 80486 (Grehan *et al.*, 1998, page 64) but because Microsoft Visual C++ 5 or 6 is required, a Intel Pentium 166 MHz is recommended.
Further, the host pc must have one free parallel port and one free serial port (or just two free serial ports) for the host to target connection. The same applies also to the target pc. This means for Arty that the parallel port adapter cable and the serial port adapter cable are needed.

## Installation of the ToolSuite Lite on the host machine

For installation of the ToolSuite Lite on the host PC, just follow the instructions on pages 65-71 (Grehan *et al.*, 1998). Very important is adding the directories ToolSuite directories to Visual C++ (page 67) and to the Windows environment variables (page 71). The latter can be found under Windows NT/2000 by right clicking on "My Computer". Choose "Properties" and then on the tab "Advanced" the button "Environment variables. To change this, administrator rights are needed.

## Building a Boot Disk

For building the Phar Lap bootdisk with the ETS monitor, follow the instructions on pages 71-73. The bootdisk uses standard com-port 1 for the host-target communication. For Arty it is better to use com-port 2, because only com-port 1 can be used as terminal port for the TVTERM program (see below). To do this use: `cfgkern –com 2 a:diskkern.bin`

To check if the boot disk is working, try it first on a PC with monitor. If nothing appears on the screen after booting, the disk is not working (Does sometimes happen). Just repeat the steps above to correct this.

To use the bootdisk on Arty, the floppydrive has to be connected and the PC/104 BIOS must boot from floppy first. To change the BIOS settings, the supplied terminal cable (serial nullmodemcable with a switch on one of the connectors) has to be connected to the first serial port of the PC/104 board and another computer running MSDOS and the TVTERM program, supplied with the PC/104 board.
The TVTERM program is a terminal program that gives the user the ability to interact with the PC/104 using the screen and keyboard of a normal PC. With this program it is possible to see the bootprocess and to change the settings of the PC/104 BIOS.

# Appendix C: Target configuration file

An important file for the code generation tool is the target configuration file (or target description file) “targets.ini”. This file contains all the information needed for 20-Sim to generate code for a specific target. So, when a user wants to define a new target, the “targets.ini” file has to be adjusted. In this file each target has its own section in which: name, icon, description, directory, template files, target directory, precommand and postcommand are specified. Also the type of model can be selected (submodel or not) and the new line character (e.g. for Unix files).
To add a new target to the target configuration file, first, a tag has to be added to the targets section. The next step is to add a new section to the file with the same name. In this section, the following keywords can be used to add the target specific information (At the end of this appendix, the settings for the PC/104 target are given as an example).

**targetName=**
the name that will appear in the code generation dialog in 20-sim

**iconFile=**
the name of a icon file (.ico) which contains an icon to appear in the 20-sim code generation dialog.

**description=**
the string that will appear in the description field in the code generation dialog

**submodelSelection=**
values: TRUE (default)
FALSE
Determines whether c-code is generated for the complete 20-sim model or that a submodel selection is required

**preCommand=**
a Command which will be executed in the target directory before that the c-code will be generated.

**templateDirectory=**
here the pathname where the template files for the c-code can be found can be specified. The default name is the target name in the "ccode" directory of 20-sim. If no full path is specified then as offset the ccode directory in 20-sim is taken. The name may be in double-quotes, but this is not necessary.

**templateFiles=**

Specifies a list of files, ';'-separated, which specify what files are generated in the targetDirectory. A find/replace of keywords is done by 20-sim. Names may be in double-quotes, but this is not necessary.

**targetDirectory=**

this holds the default target directory where the files will be generated. this will appear in the 20-sim dialog box when c-code is generated and can can be overruled. Names may be in double-quotes, but this is not necessary.

**postCommand=**

a Command which will be executed in the target directory after that the c-code has been generated. For example a "mex" command for Simulink. But also the name of a batch-file(.bat) can be specified, so that a make command can be invoked.

**newLineCharacter=**

0 = CRLF (0x0d0a = DOS Standard)
1 = CR   (0x0d = Macintosh Standard)
2 = LF   (0x0a = Unix Standard)

Settings for the PC/104 target:

```
; Possible targets for 20-sim C-Code Generation
[targets]
StandAloneC
CFunction
Simulink
PC/104BoardSub

;
; … Code for the other targets here …
;
; Generate C-code for a PC/104 board (ARTY) using Phar lap's real-time ETS
; kernel (for submodel)
[PC/104BoardSub]
targetName="PC/104 Board  (submodel)"
iconFile="pc104.ico"
description="Use this target when testing a 20-sim submodel on ARTY using
Phar lap's real-time ETS kernel."
SubmodelSelection=TRUE
templateDirectory="PC/104Sub"
templateFiles=xxfuncs.c;xxfuncs.h;xxinteg.c;xxinteg.h;xxinverse.c
templateFiles=xxmain.c;xxmatrix.c;xxmatrix.h;xxmodel.c;xxmodel.h
templateFiles=xxsubmod.c;xxsubmod.h;xxtypes.h;xxoutput.h;xxoutput.c;
templateFiles=artyfunc.c;artyfunc.h;artybumper.c;artybumper.h
templateFiles=artymotor.c;artymotor.h;artyport.h;timer.h;timer.c
templateFiles=20sim2arty.mak;20sim2arty.lnk;postproc.h;ppparse.def
templateFiles=artysonar.h;artysonar.c
targetDirectory="c:\temp\%SUBMODEL_NAME%"
postCommand=pc104pp.exe
```

# Appendix D: 20-Sim tokens

## Overview:

The adaptation of the template files, to create files containing the model, is done by use of *tokens*. The template files contains the global framework of the simulation model. The parts of the source code that are model specific are represented by tokens. When 20-sim generates the code these tokens are replaced by model specific information code. The reason for using this structure is that this code is not yet known and is dependent on the model. A token consists of a % sign followed by a variable name and is ended with %. When 20-sim detects these tokens they are automatically replaced by standard ANSI C-code. Therefore the template also has to be written in ANSI-C. Several tokens are available for use in a template. These tokens are included below.

## Tokens:

In this appendix, all the current available tokens for the template source code are described. Future versions of 20-Sim may have more tokens. All the available tokens are written in the file ‘keywords.txt’ (in the CCODE directory in the 20-Sim directory)

**%NUMBER_CONSTANTS%**
**%NUMBER_PARAMETERS%**
**%NUMBER_VARIABLES%**
**%NUMBER_STATES%**
**%NUMBER_DEPSTATES%**
**%NUMBER_ALGLOOPS%**
**%NUMBER_CONSTRAINTS%**
**%NUMBER_IMPORTS%**
**%NUMBER_EXPORTS%**
**%NUMBER_INPUTS%**
**%NUMBER_OUTPUTS%**
**%NUMBER_MATRICES%**
**%NUMBER_UNNAMED%**

An integer that gives the amount of variables etc. that appear in the model equations. Typically used in memory allocation parts and loops.

*Example*:	double p [%NUMBER_PARAMETERS% + 1];

**%WORK_ARRAY_SIZE%**

An integer defining the amount of doubles needed for the work array. This work array is used in certain matrix operations.

**%CONSTANT_NAMES%**
**%PARAMETER_NAMES%**
**%VARIABLE_NAMES%**
**%STATE_NAMES%**

**`%RATE_NAMES%`**
**`%DEPSTATE_NAMES%`**
**`%DEPRATE_NAMES%`**
**`%ALGLOOP_NAMES%`**
**`%CONSTRAINT_NAMES%`**
**`%IMPORT_NAMES%`**
**`%EXPORT_NAMES%`**
**`%INPUT_NAMES%`**
**`%OUTPUT_NAMES%`**
**`%MATRIX_NAMES%`**

The names of the variables etc. that appear in the model as a list of comma-separated static strings. Typically used in user-interface routines to edit parameters or plot variables later on.

*Example*:  char *pnames [] = { %PARAMETER_NAMES %  NULL};

**`%INITIALIZE_CONSTANTS%`**
**`%INITIALIZE_PARAMETERS%`**
**`%INITIALIZE_MATRICES%`**
**`%INITIALIZE_STATES%`**
**`%INITIALIZE_DEPSTATES%`**
**`%INITIALIZE_ALGLOOPS%`**
**`%INITIALIZE_CONSTRAINTS%`**
**`%INITIALIZE_INPUTS%`**
**`%INITIALIZE_OUTPUTS%`**

Equations that set initial values of parameters etc. Typically used at the start of a routine to set correct values.

*Example*:

```
P[0] = 3.185;        /* initial gain */
P[1] = -0.2;         /* feedback */
s[0] = 120.8;        /* x0 */
```

**`%INITIAL_EQUATIONS%`**
**`%STATIC_EQUATIONS%`**
**`%INPUT_EQUATIONS%`**
**`%DYNAMIC_EQUATIONS%`**
**`%OUTPUT_EQUATIONS%`**
**`%FINAL_EQUATIONS%`**

Equations that form the main calculation part of the model itself. This will be the bulk of the generated code.

**`%INPUT_TO_VARIABLE_EQUATIONS%`**
**`%VARIABLE_TO_OUTPUT_EQUATIONS%`**

Equations for submodel calls that perform a mapping from an input and output vector argument (u and y) to the internal variables, states etc.

**%INTEGRATION_METHOD_NAME%**

A string representing the name of the selected integration method, for now only Euler, RungeKutta4 and Discrete are available.

**%START_TIME%**
**%FINISH_TIME%**
**%TIME_STEP_SIZE%**

The values (in floating point notation) of the simulation parameters.

*Example* double t_start = %START_TIME%;
double t_end = %FINISH_TIME%;

**%MODEL_IS_DISCRETE%**

Either TRUE or FALSE, depending if the model is discrete in time or continuous in time.

**%FILE_NAME%**
**%MODEL_FILE%**
**%MODEL_NAME%**
**%SUBMODEL_NAME%**
**%EXPERIMENT_NAME%**

Strings that represent the name of the current file, model, submodel and experiment. Typically used in comments and in the target.ini file.

**%GENERATION_TIME%**
**%GENERATION_DATE%**
**%GENERATION_BUILD%**
**%GENERATION_DIR%**
**%USER_NAME%**
**%COMPANY_NAME%**
**%20SIM_DIR%**

String that represent additional information on the code generation process, the user and 20-sim itself. Typically used in comments and in the target.ini file.

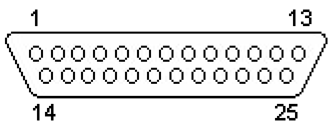
# Appendix E: Cable layouts

This appendix contains the cable layouts for the cables needed for programming Arty with the TNT Toolsuite lite.

## Parallel cable

This cable is used for sending the compiled code to the target. This cable will work with:

- TNT Toolsuite
- LapLink
- Microsoft MS-DOS InterLink
- Microsoft Windows Direct Cable connection
- Symantec Norton Commander v4.0 & v5.0



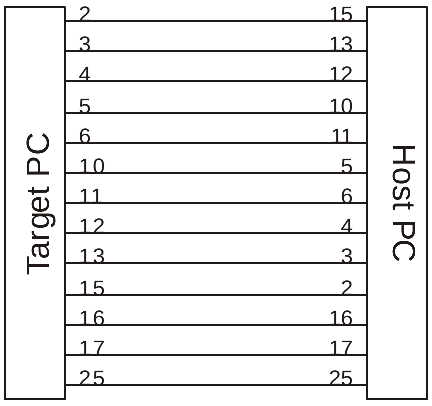


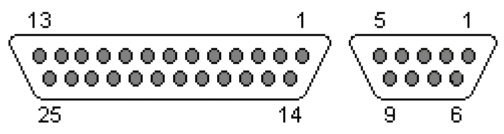
**Figure 28: Laplink cable connector pinning & cablelayout**

*Note*: Signal names are the standard pin names, not the names for a parallel Laplink cable.

| **25 PIN D-SUB MALE to Host** | | **25 PIN D-SUB MALE to target** | |
|---|---|---|---|
| **Name** | **Pin** | **Pin** | **Name** |
| Data Bit 0 | 2 | 15 | Error |
| Data Bit 1 | 3 | 13 | Select |
| Data Bit 2 | 4 | 12 | Paper Out |
| Data Bit 3 | 5 | 10 | Acknowledge |
| Data Bit 4 | 6 | 11 | Busy |
| Acknowledge | 10 | 5 | Data Bit 3 |
| Busy | 11 | 6 | Data Bit 4 |
| Paper Out | 12 | 4 | Data Bit 2 |
| Select | 13 | 3 | Data Bit 1 |
| Error | 15 | 2 | Data Bit 0 |
| Reset | 16 | 16 | Reset |
| Select | 17 | 17 | Select |
| Signal Ground | 25 | 25 | Signal Ground |

## Serial cable (full null modem cable)

This cable is for the communication from the target to the host machine. The TNT Toolsuite also uses this cable for debugging the running application on the target.



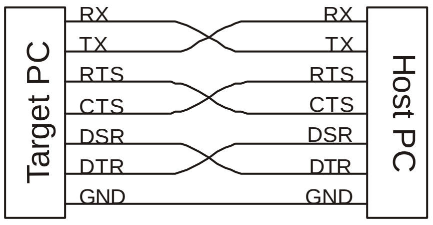


**Figure 29: Nullmodem cable: connector pinning & cable layout**

| Signal (host) | 25-pin Female | 9-pin Female | Signal (target) | 25-pin Female | 9-pin Female |
|---|---|---|---|---|---|
| GND | 7 | 5 | GND | 7 | 5 |
| TX | 2 | 3 | RX | 3 | 2 |
| RX | 3 | 2 | TX | 2 | 3 |
| RTS | 4 | 7 | CTS | 5 | 8 |
| CTS | 5 | 8 | RTS | 4 | 7 |
| DSR | 6 | 6 | DTR | 20 | 4 |
| DTR | 20 | 4 | DSR | 6 | 6 |

# Appendix F: PC/104 board specifications

These data are originating from the Arty report (Balkema, *et al.*, 1999)

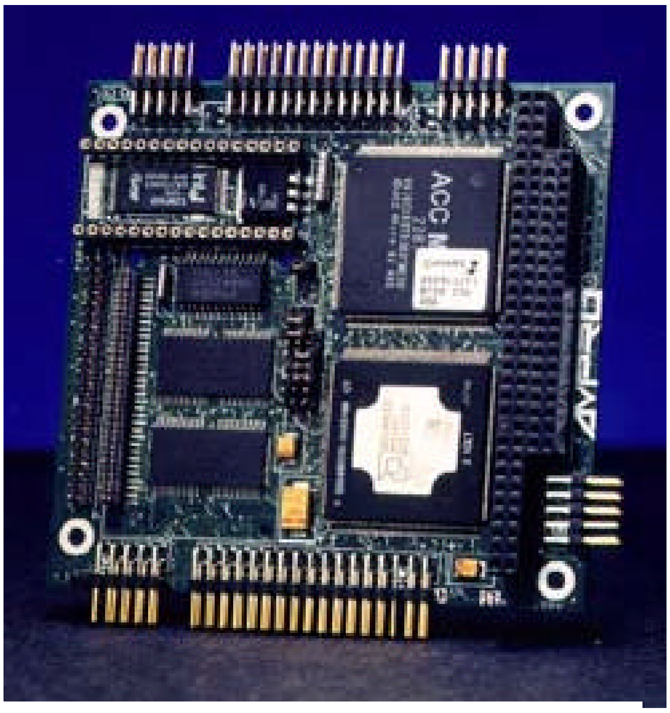


**Figure 30: AMPRO PC/104 board**

## CoreModule/4Dxi

- Powerful 66, 100 or 133 MHz 486DX4 processor
- Up to 52M bytes onboard DRAM
- Fast IDE controller
- Floppy disk controller
- Dual FIFO- buffered serial controllers
- Bi-directional parallel port
- 16 bit PC/104 expansion bus
- Optimized for embedded applications
- Wide operating temperature
- Enhanced embedded PC-BIOS
- Socket for bootable "Solid State Disk"
- Watchdog timer
- Advanced Power Management
- Low power consumption
- Batteryless boot

# Appendix G: Arty control ports

Ports can address all of the sensors and actuators that are connected with the PC/104 board on Arty. This appendix contains the port numbers and control values for the various sensors/actuators. These portnumbers are given here because they lack in the Arty report (Balkema *et al.*, 1999)

| I/O Port (hexadecimal) | Actuator/sensor | Description |
|---|---|---|
| **Left motor** | | |
| 300h | Left motor status port | |
| 302h | Left motor speed sensor port | Contains the left motor speed (value between 00 and FF) |
| 302h | Left motor control port | Sets the left motor speed (value between 00 and FF) |
| **Right motor** | | |
| 300h | Right motor status port | |
| 304h | Right motor speed sensor port | Contains the right motor speed (value between 00 and FF) |
| 304h | Right motor control port | Sets the right motor speed (value between 00 and FF) |
| **Bumper** | | |
| 306h | Bumper output port | Bit 0: front left bumper switch<br>Bit 1: front right bumper switch<br>Bit 2: rear left bumper switch<br>Bit 3: rear right bumper switch |
| **Sonar** | | |
| 310h | Front left sonar output port | Contains the front left sonar output value |
| 312h | Front left middle sonar output port | Contains the front left middle sonar output value |
| 314h | Front right middle sonar output port | Contains the front right middle sonar output value |
| 316h | Front right sonar output port | Contains the front right sonar output value |
| 318h | Rear left sonar output port | Contains the rear left sonar output value |
| 31Ah | Rear left middle sonar output port | Contains the rear left middle sonar output value |
| 31Ch | Rear right middle output port | Contains the rear right middle sonar output value |
| 31Eh | Rear right sonar output port | Contains the rear right sonar output value |
| **IR beacon** | | |
| 320h | IR sensor front port | |
| 322h | IR sensor rear port | |
| 324h | IR motor status port | Current angle of the rotation sensor |
| 326h | IR motor control port | Sets the motor speed (value between 00 and FF)<br>*Do not turn more than 180 ° because of cables in the sensorbar!* |

# Appendix H: artyfunc.dll

The dll used by the testmodel is `artyfunc.dll`. The exported dll functions are:

- `Initialize`
- `Terminate`
- `Motor`
- `MotorSpeed`
- `ArtyBumper`
- `ArtySonar`

When the 20-Sim simulator has to call the user defined dll for the first time the dll is linked and if the one of the initialize functions exist, this function will be called by 20-Sim before is calls the appropriate user defined function. The Initialize function will be called only when 20-Sim links to the dll and the terminate function when the simulation is ready/stopped.

Below, the source code for the dll is included for completeness. Each of the Arty functions checks first the array dimensions before it copies the values of the input array to the output array. When these dimensions are different, the dll will stop the simulation by returning a returnvalue unequal to zero. Further, the initialization and termination functions are implemented in the dll. The only thing they do here is telling 20-Sim that the initialization/termination was successful.

```
/* Dll for 20Sim codegeneration for calling Arty functions after codegeneration
   Marcel Groothuis */

#include <windows.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define DllExport __declspec( dllexport )

extern "C"
{

DllExport int Motor(double *inarr, int inputs, double *outarr, int outputs, int major)
{

 // this function connects the Arty motor function with the 20-Sim model

	if(inputs != 2)			// Check if array size is correct
		return 1;
	else if (outputs != 2)	// Check if array size is correct
		return 1;
	else
	{
		if (inarr[1] == 0)
		{
			outarr[0] = inarr[0];  //Output = Input
		}
		else if (inarr[1] == 1)
```

```
                {
                        outarr[0] = inarr[0];
                }
                else
                        return 0;
        }

        return 0; // return successful
}

DllExport int MotorSpeed(double *inarr, int inputs, double *outarr, int outputs, int major)
{
        if ((outputs != 2) || (inputs != 2))
        {
                outarr[0] = inarr[0];
        }
        else
        {
                outarr[0] = inarr[0];
                outarr[1] = inarr[1];
        }
        return 0; // return succesfull
}

DllExport int ArtySonar(double *inarr, int inputs, double *outarr, int outputs, int major)
{
        if (outputs != inputs)
                return 1; //Incorrect array size
        else
        {
                int i;
                for (i=0; i<=inputs; i++);
                {       outarr[i] = inarr[i];}
                return 0; // return succesfull
        }
}

DllExport int ArtyBumper(double *inarr, int inputs, double *outarr, int outputs, int major)
{
        if ((outputs != inputs) || (outputs !=4))
                return 1;
        else
        {
                int i;
                for (i=0; i<=inputs; i++);
                {       outarr[i] = inarr[i];}
                return 0; // return successful
        }
}

DllExport int Initialize()
{
        // no initializations necessary.
        return 0; // Indicate that the dll was initialized successfully.
}

DllExport int Terminate()
{
        // no cleaning necessary
        return 0; // Indicate that the dll was terminated successfully.
}
}
```

**Listing 4: Part of the artyfunc.dll source**

# Appendix I: Bugs in the ANSI C template

Before changing the CFunction template to the PC104SUB template for Arty, The CFunction template was tested first, to check if it works. During these tests some bugs were discovered in the existing CFunction template.

### Step size/simulation time bug:

A simple 20-Sim model was used with only a pulse generator and a resistor to test the code generation tool. After code generation, it turned out that the code was not working. The two variables below were not declared in `xx.main.c`.

```
XXDouble xx_step_size;
XXDouble xx_simulation_time;
```

### No states bug:

After this correction, it turned out that the code was still not working correct. The file `xxinteg.c` also contains a bug. When the model has no states, the Runge Kutta integration method is not called and the code gets in a endless loop. This bug is corrected in the PC104SUB template by adding a else branch to the if statement in the `XXRungeKutta4Step` function. See also below in Listing 5:

```
/* check if we have states at all */
if (xx_number_of_states > 0)
{
        ………existing code………
}
else
{
        xx_simulation_time = xx_simulation_time + xx_step_size;
}
```

**Listing 5: Bug correction for the no states case**

### DLL call conversion-to-function bug:

During some tests with the 20-Sim dll call, Thijs Withaar discovered that the variable order for calling the 'dll function' in the template was a bit strange: function(output, input). More logical is first the input variables and then the output variables, also used when writing DLL's. See also section 2.4 about the The template
Two types of templates exist, because there are two different ways of generating code. The first way is to generate code from the total 20-sim model the second is to generate only code of a submodel of the 20-sim model. When the generated code has to run on a target only the second way is of importance because only the contents of the target sub model (e.g. a controller) has to be uploaded to the target. This applies also for this project. In Appendix K both methods with the corresponding templates are described in detail.
.

During this test it became clear that the variable `xx_major`, required for the function call was nowhere defined. This variable indicates whether a integration step is a major one or a intermediate step. This lack is corrected in the PC104SUB template.

# Appendix J: Arty API

## Motor API

### Functions:

```
void LeftWheelMotor (int speed);
void RightWheelMotor (int speed);
int LeftWheelMotorSpeed ();
int RightWheelMotorSpeed ();
```

with speed, a value between –128 and +128.

## Bumper sensor API

### Function:

```
int GetBumperStatus();
```

Returnvalue must be ANDed with one of the following masks:

| **Mask** | **Value** |
|---|---|
| FRONTLEFTBUMPER | 1 |
| FRONTRIGTBUMPER | 2 |
| REARLEFTBUMPER | 3 |
| REARRIGHTBUMPER | 4 |

**Table 3: Bumpermask constants**

Example:

```
//Check whether the Front left bumper is pressed:
if ((returnvalue & FRONTLEFTBUMPER) == FRONTLEFTBUMPER) …
```

## Ultrasonic sensor API

### Functions for all sonarsensors:

```
int GetSonarValue (unsigned char number);
void GetAllSonarValues (struct SONARTYPE sv);
```

### Functions for one sonarsensor:

```
int GetFrontLeftSonarValue ();
int GetFrontLeftMiddleSonarValue ();
int GetFrontRightMiddleSonarValue ();
int GetFrontRightSonarValue ();
int GetRearLeftSonarValue ();
int GetRearLeftMiddleSonarValue ();
int GetRearRightMiddleSonarValue ();
```

```
      int GetRearRightSonarValue ();
```

All functions return the distance of one sonar sensor (in practice a value between 20 and 120) exept GetAllSonarValues. This function returns a structure containing all the sonar values. This structure consists of 8 byte values (chars). See below:

```
      struct SONARTYPE
      {
            char fl;    //Front left sonar
            char flm;   //Front left middle sonar
            char frm;   //Front right middle sonar
            char fr;    //Front right sonar
            char rl;    //Rear left sonar
            char rlm;   //Rear left middle sonar
            char rrm;   //Rear right middle sonar
            char rr;    //Rear right sonar
      };
```

**Listing 6: Struct definition of SONARTYPE**

To facilitate the use of the function GetSonarValue, eight constants, corresponding with the sonar sensors, are defined. See Table 4

| **Constant** | **Value** |
|---|---|
| FRONTLEFT | 1 |
| FRONTLEFTMIDDLE | 2 |
| FRONTRIGHTMIDDLE | 3 |
| FRONTRIGHT | 4 |
| REARLEFT | 5 |
| REARLEFTMIDDLE | 6 |
| REARRIGHTMIDDLE | 7 |
| REARRIGHT | 8 |

**Table 4: Sonar constants**

# Appendix K: Template structure

First the code generation for complete model is explained as a sort of introduction to the code generation for submodels. This is done because the code generation for complete models is more orderly and the code generation for submodels can be seen as an extension of it.

## Code generation for complete models

When the option Stand-alone ANSI-C is selected in the code generator, 13 files are generated. These files with their description are represented in Table 5

| | |
|---|---|
| xxfunc.c | This file describes the additional functions that may appear in the 20-Sim model and that are not part of the ANSI-C language. For example `ArcTangentHyperbolic` |
| xxinteg.c | This file describes the integration methods that are supplied for computation (Euler or Runge Kutta 4). |
| xxinverse.c | This file contains the functions necessary for calculating the inverse of a matrix |
| xxmodel.c | This files contains the C-code that describes the 20-sim model. |
| xxmain.c | This file contains the main program that runs the model. |
| xxmatrix.c | This file implements the functions necessary for matrix operations |
| xxoutput.c | This file describes how the simulation output should be printed on screen. |
| xxfunc.h | Header file for xxfunc.c. |
| xxinteg.h | Header file for xxinteg.c. |
| xxmatrix.h | Header file for xxmatrix.c |
| xxmodel.h | Header file for xxmodel.c |
| xxoutput.h | Header file for xxoutput.c |
| xxtypes.h | This file describes the typedefs that are used for integers and doubles. All the generated code uses these typedefs to enhance flexibility (in case other compiler are used than Microsoft Visual C). |

**Table 5: Generated files for complete model code generation**

The files xxinverse.c and xxmain.c do not have a header file.
In xxtypes.h the following types are defined:

```
typedef double XXDouble;
typedef int XXInteger;
typedef char XXCharacter;
typedef char XXBoolean;
```

The compiler now fills in the size of double, int, etc. However 20-sim expects the following sizes:

- XXDouble is 8 bytes (64 bits) with a range of [-1.7E+308, 1.7E+308]
- XXInteger is 4 bytes (32 bits) with a range of [-2147483648, 2147483647]
- XXCharacter is 1 byte (8 bits) with a value of [0, 255]
- XXBoolean is 1 byte (8 bits) with a value of 0 or 1

Therefore if a compiler reduces the number of bytes for a variable, the results on the target may differ from the ones of the 20-sim simulation.

The file xxmain.c contains the main loop, which runs with a step-size until the simulation-time is reached. See Figure 31 for an overview of the main structure of the template.

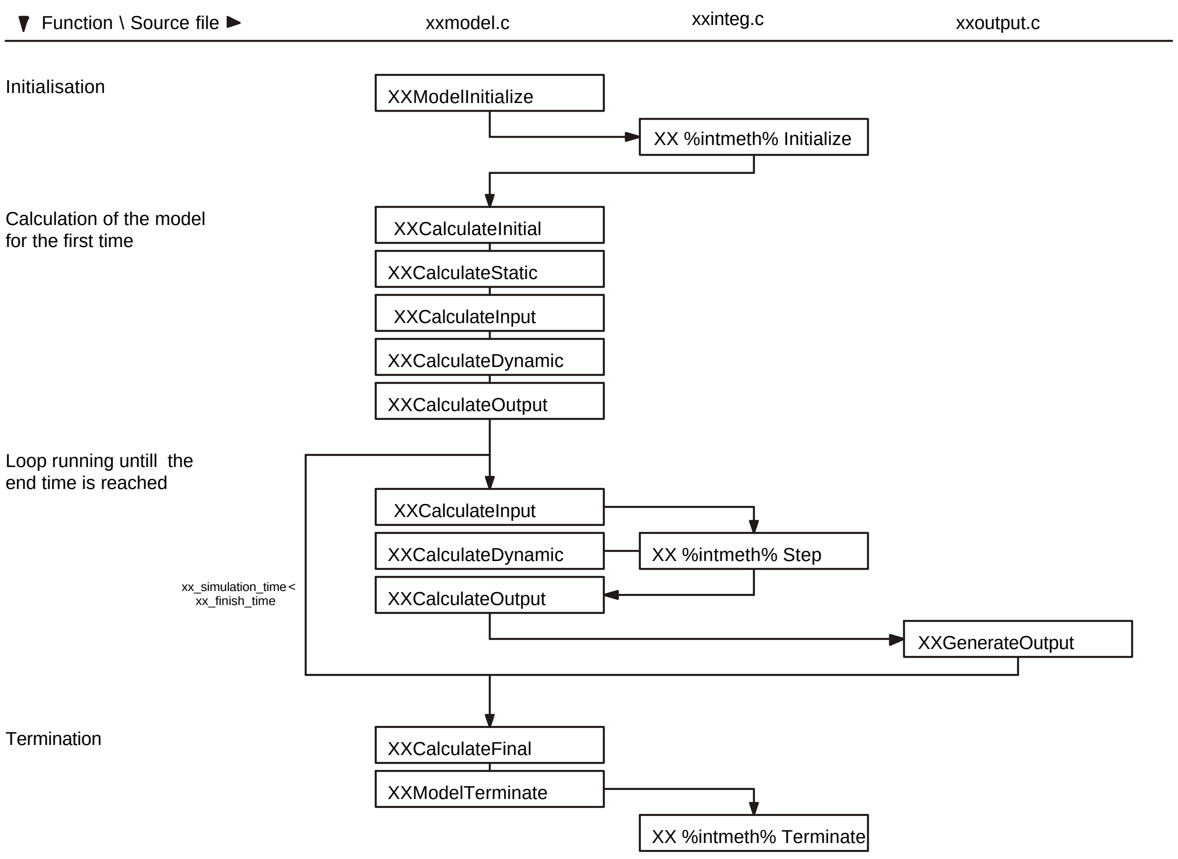


**Figure 31: Flowchart of the template for a complete model**

This diagram represents the xxmain.c file, the blocks are functions that are called in xxmain.c. The functions are placed in columns. Above each column the c-file in which the functions are located can be seen. The flowchart describes the following path:

- Initialisation phase:
  The initialisation function 'XXModelInitialize' creates the array structures and the pointers needed in the other functions. The second initialisation that has to be done is the initialisation of the integration method. This function is called with 'XX %intmeth% Initialize' where %intmeth% is a token for the integration method used. For now only Euler and Runga Kutta 4 are available. If a new integration method is required an extra function for this method can be added to 'xxinteg.c'.

- Calculation of the model for the first time:
  In the next five functions the model is calculated for the first time. All these functions are located in the file xxmodel.c.

- Loop:
  After this stage a loop is entered which runs until the finish time is reached. In this loop first the input is calculated ‘XXCalculateInput’ then ‘XX %intmeth% Step’ is called to calculated the next step. After that the output variables are calculated. Although not implemented in the original template, after calculation of the output, this output could be used to generate output to the outside world with use of ‘XXGenerateOutput’. This output could be used for example for debugging.

- Termination phase:
  When the finish time of the simulation is reached, the loop is ended. Now the last calculation is done ‘XXCalculateFinal’ and the model is terminated with ‘XXModelTerminate’ and ‘XX %intmeth% Terminate.

So xxmain.c contains the calls to all the other files. The file xxmodel.c holds the information about the model, simulation data etc. The files xxinteg.c and xxoutput.c don’t contain any tokens so they are not adapted during the code-generation process. So if we look back at figure 1 we see that xxmodel.c is mainly filled by information generated by 20-sim and xxinteg.c and xxoutput.c is filled only with information already included in the template.

During the calculation of the model the following arrays are used:

| **Array** | **Pointer to array** | **Description** |
|---|---|---|
| Xx_constants | *c | This array contains all constants used in the 20-Sim model |
| Xx_parameters | *P | This array contains all used parameters |
| Xx_variables | *V | This array contains all used variables (e.g. `p.e`, `p.f`, `output`) |
| Xx_states | *s | This array contains all state variables (e.g. `state`) |
| Xx_rates | *R | This array contains all the rate variables (e.g. `p.f` by a C element and `p.e` by a I element) |
| Xx_matrices | *M | This array contains all the used matrixes |
| Xx_unnamed | *U | This array contains all unnamed constants |
| Xx_workarray | | Not used in the current templates |

**Table 6: Defined arrays**

These arrays are created during the initialization and the occupied memory is freed in the termination phase. Elements of these arrays could be used in the ‘XXGenerateOutput’ function to send output to the user as a check. For convenience, the original names of the variables, parameters, etcetera, corresponding with the arrays above, are saved in the arrays `xx_constant_names`, `xx_parameter_names`, `xx_variable_names`, `xx_state_names`, `xx_rate_names`, `xx_matrix_names`.

## Code generation for submodels

When the option ANSI-C Function is selected in the Code Generator, the files in Table 7 are generated.

| | |
|---|---|
| xxfuncs.c | This file describes the additional functions that may appear in the model and that are not part of the ANSI-C language. |
| xxinteg.c | This file describes the integration methods that are supplied for computation (Euler or Runge Kutta 4). |
| xxinverse.c | This file contains the functions necessary for calculating the inverse of a matrix |
| xxmain.c | This file contains a simple SISO demonstration of how the ANSI-C function can be used. |
| xxmatrix.c | This file implements the functions necessary for matrix operations |
| xxmodel.c | This file contains the actual C-code that describes the 20-sim submodel. |
| xxsubmod.c | This file contains the input / output functions that describe the 20-sim submodel. (ports in 20-Sim) |
| xxfuncs.h | Header file for xxfuncs.c. |
| xxinteg.h | Header file for xxinteg.c. |
| xxmatrix.h | Header file for xxmatrix.c |
| xxmodel.h | Header file for xxmodel.c |
| xxsubmod.h | Header file for xxoutput.c |
| xxtypes.h | This file describes the typedefs that are used for integers and doubles. All the generated code uses these typedefs to enhance flexibility in case other compiler are used than Microsoft Visual C or another target platform like Unix is chosen. |

**Table 7: Generated files for submodel code generation**

Most of the template files are the same as for a complete model except for xxsubmod.c, xxsubmod.h and xxmain.c. The flowchart of xxmain.c can be seen in Figure 32. This file only contains the main loop and calls for the functions described in xxsubmod.c. This file, on its turn, calls the functions in xxmodel.c and xxinteg.c.

The main difference between the template for complete models and the template for submodels are the functions XXCopyInputsToVariables and XXCopyVariablesToOutput. Because the submodel is part of a complete model, the submodel contains some input and output ports to the rest of the model. The two functions mentioned above take care of the exchange between input/output variables and the input/output vector. However because the submodel will run on the target, input data will be inserted after the ports (to complete model). Therefore in this project the input value’s can be set to zero. The real input value’s will come from the I/O of target

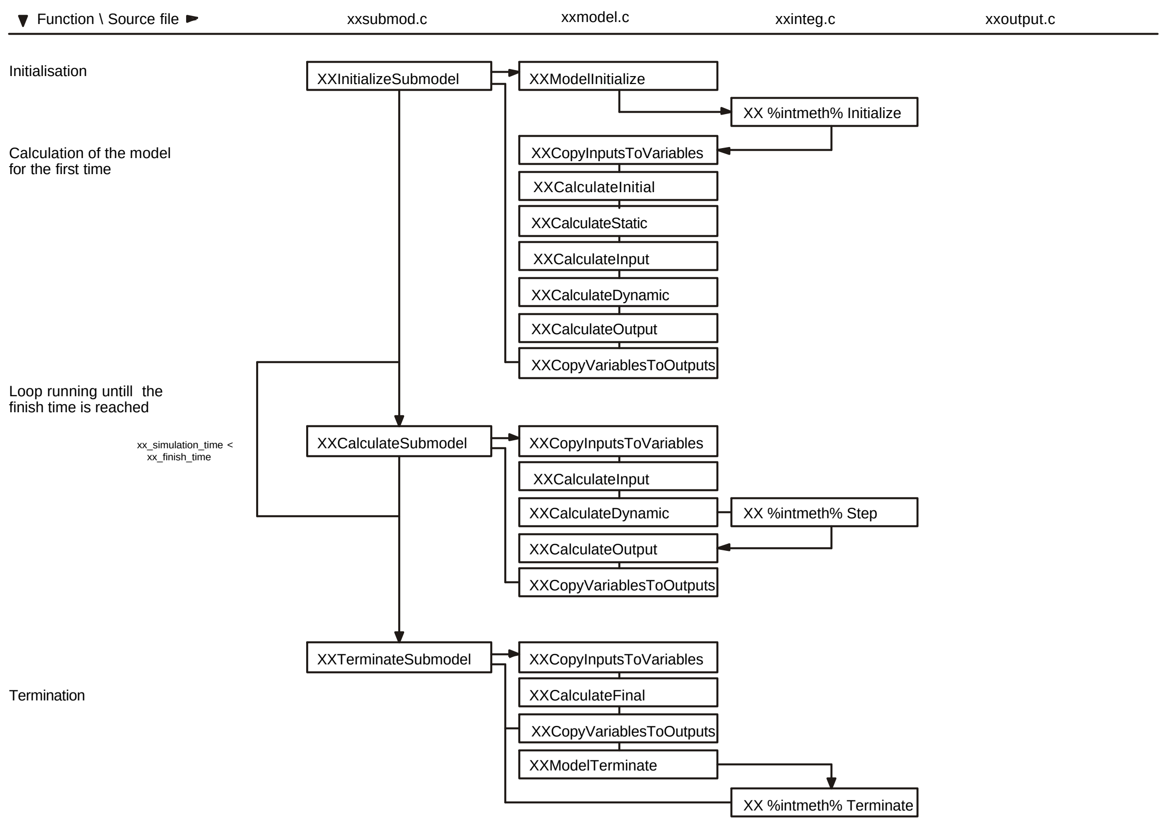


**Figure 32: Flowchart of the template for a submodel**

# References

Amerongen, J.van and T.J.A.de Vries (1998), *Digitale Regeltechniek,* Universiteit Twente, Enschede

Ampro (1997), *CoreModule ä /4DXi Technical Manual,* Ampro Computers Incorporated, San José,

Balkema, W. *et al* (1999), *Arty*, Enschede

Breedveld, P.C. and J.v. Amerongen (1994), *Dynamische Systemen: modelvorming en simulatie met bondgrafen*, Open Universiteit, Heerlen, 90 358 1302 2.

Broenink, J.F. and G.H. Hilderink (2001), Building blocks for control system software, *Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001,* Enschede, Netherlands, (Ed.),

Broenink, J.F. and G.H. Hilderink (2001), A structured approach to embedded control systems implementation, *2001 IEEE Conference on Control Applications, Sept 5-7,* Mexico, (Ed.),

Broenink, J.F., G.H. Hilderink and A.W.P. Bakkers (2000), Building Blocks for Embedded Control Systems, *PROGRESS 2000 Workshop on Embedded Systems,* Utrecht, (Ed.), 27-34.

Douglass, B.P. (1998), *Real-Time UML: developing efficient objects for embedded systems,* Addison Wesley Longman, 0-201-32579-9.

Grehan, R., R. Moote and I. Cyliax (1998), *Real-Time Programming: a guide to 32-bit Embedded Development,* Addison Wesley Longman, Reading, Massachusetts, 0-201-48540-0.

Martin, F.G. (2001), *Robotic Explorations,* Prentice Hall, Upper Saddle River, New Yersey, 0-13-089568-7.

PC/104 Consortium (1996), PC/104 Specification, Version 2.3 June 1996, http://www.rtdusa.com/pdf/pc104-23.pdf

Steehouder, M., C. Jansen and K.M. e.a. (1999), *Leren Communiceren,* Wolters Noordhoff bv, Groningen, 90-01-80826-3.

Weustink, P. and F. Groen (2001), Generating ANSI C-Code for user targets with 20-Sim 3.2 Pro, Controllab Products B.V., Enschede.