Export Embedded C code

Blocks available for C code export

All blocks that are available for C code export can be found in Components/Blocks/ExportC
To select these blocks, select from the right tree view "Components", select "blocks" and open the ExportC section by selecting "ExportC". The blocks can be dragged onto the worksheet by clicking them with the left mouse button. Release the mouse button, (the component sticks to the mouse pointer), drag the mouse over the work-screen and click the left mouse button again to drop the component.

Build your control inside a library block

To make use of file classes and combine various file classes with different sampling times, the use of library blocks is recommended. For each library block a separate file class is exported. Variables declared inside a file class are only visible within the scope of the file class.

For export of the code of the library block, open the block by selecting the [Edit] button in the properties dialog box. After opening the library file, select the Embedded C Export button to export the c and h file.

Variables to interface with the code in file class

Inputs to the library block are exported as "extern" The external declaration can be found in the *.h file. The variable has to be declared as in the main project file.

Signals that have to be exported from the code have to be declared as external variables. They will be declared as external variables in the *.h file. In the properties of the block, the extern declaration can be defined.

Parameters for a block of code

Parameters can be set as numerical values inside the library block. In that case the user can only change those parameters by opening the library block and change the numerical value of the parameter.
By using inputs to the library block or MPAR blocks, the user can set the parameters of a library block without opening the library block.

Parameters for the code can be set using:

1. Constants defined in the library block. (The user has to open the library block)
2. Inputs to the library block (The parameters can vary duing the execution)
3. Parameters in the library block set using the "MPAR" block (The parameters can be set in the properties dialog box of the library block)

If a "MPAR" block is used within the library block, its value can be defined as default inside the "MPAR" block. In the properties dialog box of the library block, the parameter will be available for editing. In this way a user can easily insert a library block many times with different parameter sets.

Optimization of code

Block outputs from mathematical blocks that are not connected to more than one block are not exported by its own identifier, but are optimized inside the code. This means that the C code is condensed to one line.

Numerical integration methods

Numerical methods like Forward Euler, Runge Kutta 2 and Runge Kutta 4 are available for the code for the integrators. The values k1, k2, k3, k4 for the integrators are indicated by a preceding _k_

Runge Kutta 4 is set as default.

In case of Runge Kutta 4 for each output 4 variables are declared. This is done using the [4] array declaration.

After four times the evaluation of the integrator inputs for calculating k1, k2, k3 and k4 the outputs for the integrators are calculated and stored in xxxx[0]. This value is copied into xxxx[3] and finally all outputs are updated by recalculating the blocks once again based on the new value on the outputs of the integrators. If this is not implemented the output from an integrator differs from the other outputs.

Nested library blocks

Outputs from blocks inside a nested library block are exported as _prefixLibxx where the prefix is given by the Hungarian notation for the C type, and xx is an automatic generated number.

If a block is inside a nested library block the name of the block and the name of the library are commented behind the code. This enables the recognition of the blocks inside library blocks.

An output that is declared as extern in a nested library block is not exported as extern, but as static.

Remarks

If a remark is given inside a block, this remark is added as comment to the code using //
The remark is edited in the properties dialog box of the component.

Texas Instruments IQMATH library

The IQMATH library from Texas Instruments is added for export of functions using the qmath library. The qmath library is designed for fixed point calculations in the Texas Instruments C28x DSP .

Memory allocation and fragmentation

There are no memory allocation functions in the exported code. All declarations are static to prevent fragmentation of the memory. Even a proper allocation and de-allocation scheme that would loose only one byte during this process could give rise to a heap overflow and fragmentation. For a standard office application this is not a problem since these programs are loaded and removed from memory many times. An embedded system is loaded only once and will stay resident for a very long time.

Organization of the code in separate blocks.

The code is exported using file classes. This means that the scope of the variables and code are limited to within the c and h file in which they are exported.
On the other side this opens up the possibility to combine various file classes into a single project without overlapping of variables inside a file class. Only the variables that are declared as extern are visible outside the file class. This means that all communication between the code inside a file class is done via the externally declared variables.

What is all this File Classes about?

If only one block of C code is required, only one file class has to be exported. Only one library block has to be set up and the code of that block will be exported.

If more blocks of C code are required that can run independent on each other, multiple library blocks have to be inserted in the schematic. From each library block the code is exported in a unique file class and these file classes are combined in the main project file.

An example will clarify the advantage of the file classes.

A system would consist of two parts. The first part contains parameter declarations and some pre processing. The second part contains the control part that includes integrator blocks.
The code for the first block does not contain any integrators and this code has to be executed only once. The code for the second block contains integrator blocks and therefore, when applying Runge Kutta, the code is processed 2 or 4 times.

If for both parts the code would be exported as a single block of code, all he code has to be processed for each time step in the Runge Kutta numerical integration process. In practice this would mean that the code is executed 2 or 4 times. Unnecessary to say that the multiple executing of the parameter declarations and pre processing for each Runge Kutta step slows down the overall execution speed.

If for both parts separate blocks of code are exported, the C code of the first block is executed only once and the C code of the second part 2 or 4 times, depending on the Runge Kutta method.

To send the outputs from the first block of C code to the second block of C code, these variables are declared inside the main project.
The outputs from he first block are declared as extern and visible within the scope of the main project. The inputs of the second block are also declared as extern and visible within the scope of the main project.

During the execution, the first block of C code is executed once and the results are stored in the variables declared in the main project. The execution proceeds with the second part of C code that is executed 2 or 4 times. The inputs for the second block are the variables declared in the main project and due to the external declaration visible inside the second block of code.

Calling the code

The file class contains two interface functions. One interface function for the initialization and the second for the main body. During initialization of the main project code, the initialization functions of each file class have to be called. The user has to place calls to the initialization function in the main initialization code of the main project.
During the execution of the main project code, the main body of each file class has to be called. The user has to place calls to the main functions from each file class in the main body code of the main project. During the execution the main functions from the file classes are processed sequentially.
The sampling time is defined by the time delay with which the main body of the main project is called.

Sampling times

If multiple sample times are required, the main project code could include code to create various sampling times. The example code below shows two blocks of code with different sampling times.

```
int iEnable1=1;
int iEnable2=1;

caspocInit_Block1();
caspocInit_Block2();

While(true)
       {
       if(iEnable1>0)
              {
              caspocFunction _Block1();
              iEnable1=0;
              }
       if(iEnable2>9)
              {
              caspocFunction _Block2();
              iEnable2=0;
              }

       iEnable1++;
       iEnable2++;
       }
```

Block 1 is called each time in the loop, while block 2 is called only after calling block 1 ten times.

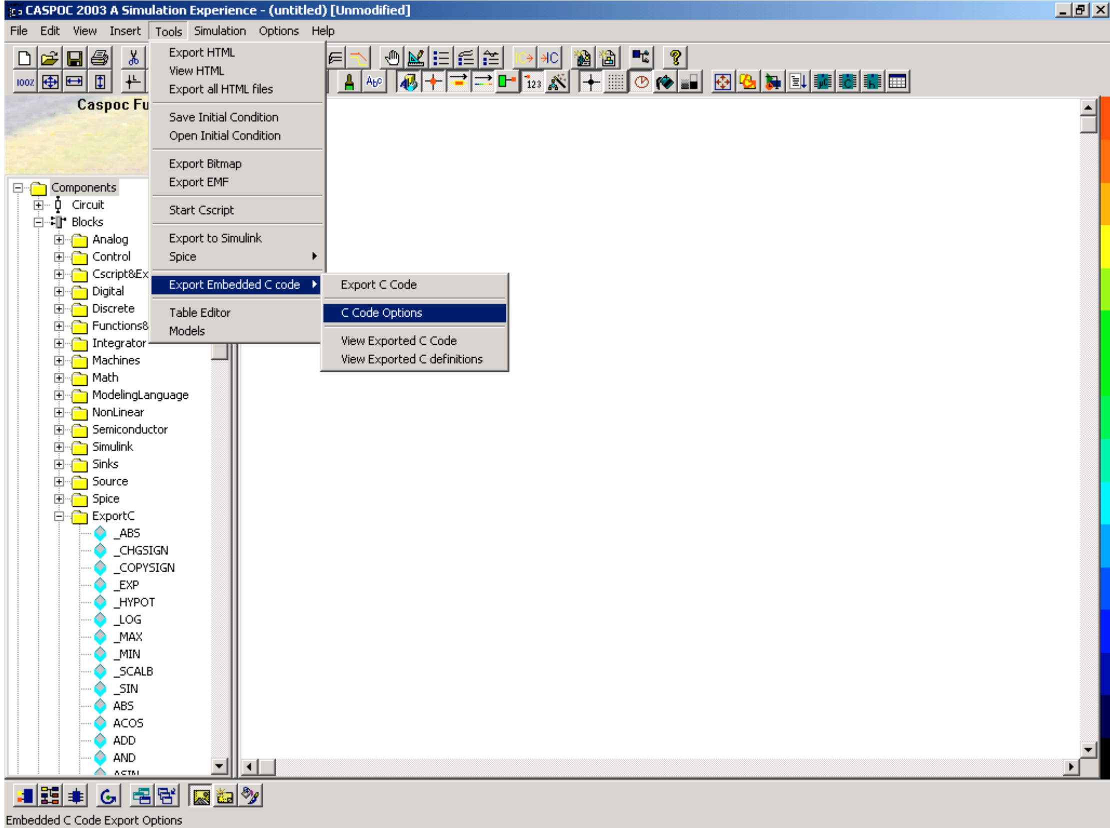
Start up code

The principle of different sampling times is also used for separating initialization and start up code from the rest of the code.
Suppose Block 1 would regulate the start up of the system, while after being signaled by a semaphore, block 2 would take over. Only the code one block has to be executed.

```
static int iSemaphore=0;

caspocInit_Block1();
caspocInit_Block2();

While(true)
       {
       if(iSemaphore==0)
              caspocFunction_Block1();

       if(iSemaphore==1)
              caspocFunction_Block2();
       }
```

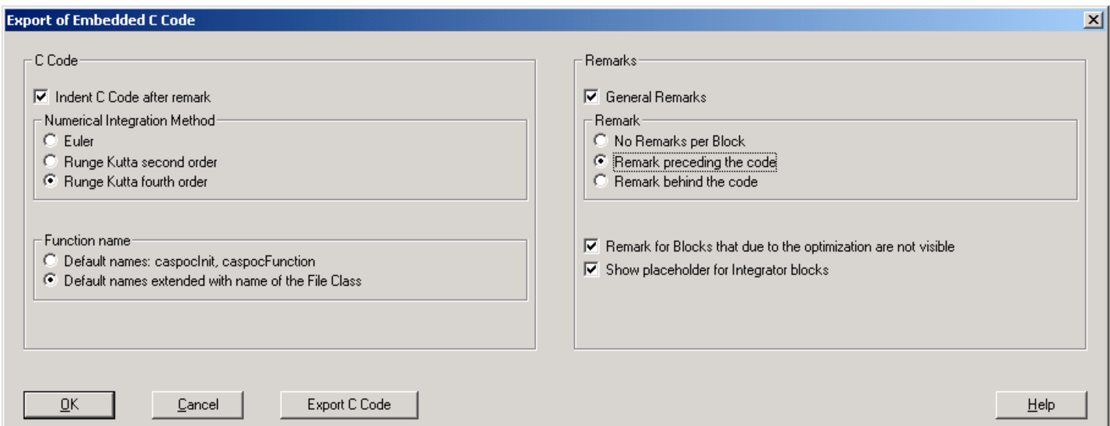
In the above example the variable iSemaphore is declared as extern in Block1. If the start up of Block 1 is finished this is indicated by setting the semaphore variable iSemaphore equal to 1 inside of Block 1.

Embedded C Code Export options



All Blocks from the section Components/Blocks/ExportC are available for export into C Code.

A number of options can be set before exporting the code. The numerical integration method can be set independent on the method used during the simulation. Also remarks can be toggled on or off.



C Code

Indent C Code after remark

Indent the C code with 2 spaces to make the code more readable

Numerical Integration Method

Choose the numerical integration method used in the exported C Code. Select between:

- Euler
- Runge Kutta second order
- Runge Kutta fourth order

Function name

The name of the functions in the File Class can be default or extended with the name of the Files Class. If only one File Class is exported, the default function name can be used.

Select [Default names: caspocInit, caspocFunction]

If more File Classes are exported, the functions in each File Class have to be identified with different names. Therefore the default function name is extended with the name of the File Class.

Select [Default names extended with name of the File Class]

Export C Code

Start the export of the C code

Remarks

General Remarks

Check this option to export the general remarks, like the structure of the code and the code organization.

Remark

For each block a remark can be exported. The remark can be defined in the Properties Dialog Box of the component. If no remark is specified in the Remark edit field of Properties Dialog Box, the remark is replaced by the full filename of the block and its type.
Blocks that are inside a library block can be identified by the name of the block.

- No Remarks per Block
- Remark preceding the code
- Remark behind the code

Remark for Blocks that due to the optimization are not visible

Blocks that, due to optimization, are not available as variable are not declared. By checking this option, the block is exported only as remark between the variable declarations

Show placeholder for Integrator blocks

Integrator blocks are not exported in a single line with code. Check this option to show remarks belonging to this integrator.

Available blocks

The following Blocks are available for Export of C Code

_ABS
_CHGSIGN
_COPYSIGN
_EXP
_HYPOT
_LOG
_MAX
_MIN
_SCALB
_SIN
ABS
ACOS
ADD
AND
ASIN
ATAN
ATAN2
BNG
CEIL
COMP
CON
COS
COSH
COUNTER
DIV
DSP
DT
EXP
FABS
FFL
FIX
FLOOR
FMOD
GAI
INL
INT
INTLIMIT
INTMOD
INTRESET
LABS
LDEXP
LIM
LOG10
MAX
MIN
MOD
MPAR
MUL
NAND
NOR
NOT
OFS
OR
PI
POL
POW
QATAN
QCOS
QCOSLT
QDIV
QFLOATTOQ
QINV1
QINV2
QLOG10
QLOGN
QSIN
QSINLT
QSQRT
QTOFLOAT
REL
ROTATE
SHL
SHR
SIN
SINH
SPL
SQRT
SQT
SUB
SUM
SUMW
TAN
TANH
TIME
TYPECAST
VSI
WEAK
XOR
Z1

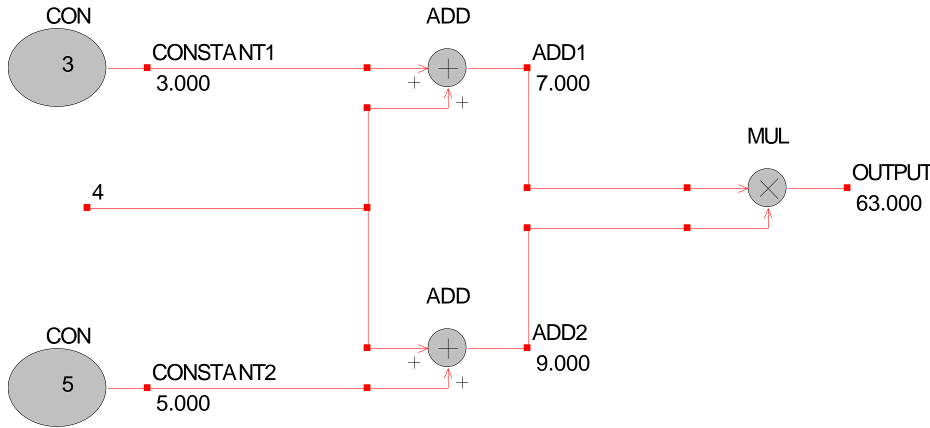
Hands-on

In this paragraph we will show some examples of how to export the code from the block diagram. In the first place we will make some simple example and show the different options when applying also numerical integration methods in the code. Also a container test application will be developed that will show you how to include the idea of file classes in your embedded software project.

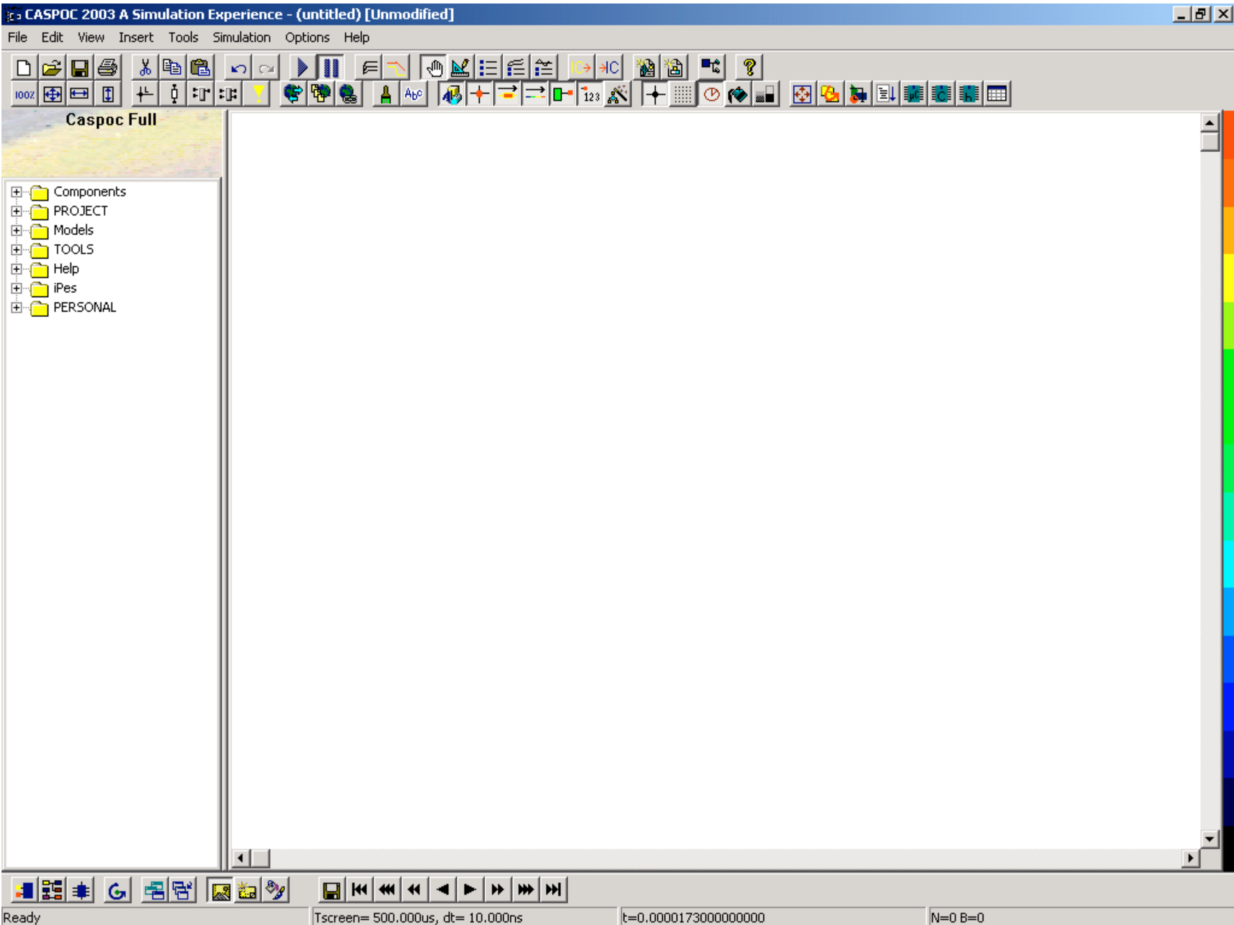
Simple example
Arithmetic example
Analog PI controller
Applying Qmath

Simple example

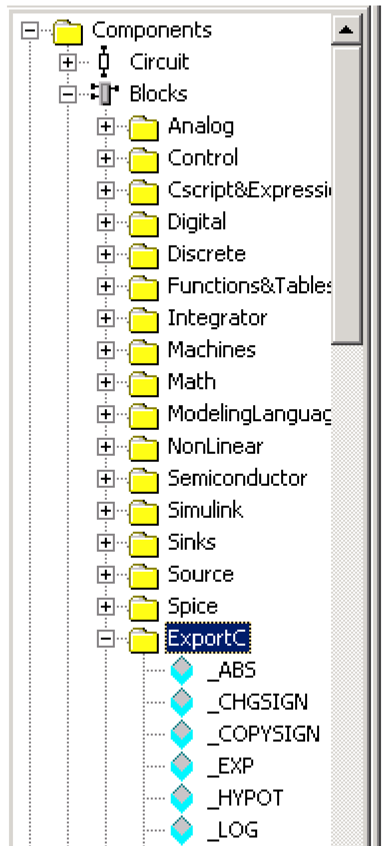
In this simple example the basics of the export of embedded C code are explained.
First we draw the block diagram as indicated below (simple.csi)



To do this we open Caspoc and we start with a blank work screen:



We select the blocks from the section Components/Blocks/ExportC



where we find the collection of all blocks that can produce C code.

Select the block CON for the constant by clicking it with the left mouse button. Release the left mouse button and drag the mouse over the work screen. Click and release the mouse button once again will place the block on the work screen. If you click the CON block again from the list of components you can ad the second CON block. Proceed in the same way for the two adder blocks ADD and the multiplier block MUL. Place the blocks in a nice order and open each block by clicking it with the right mouse button. The dialog box is shown where you can enter the parameters for the block.
Give the block a nice name, like for Constant1

Name: CONSTANT1

Enter the constant 3 for the first constant block at the input field for the first parameter.

p1 3

and type some remark to identify the block in the exported C code

Remark
Constant nr 1

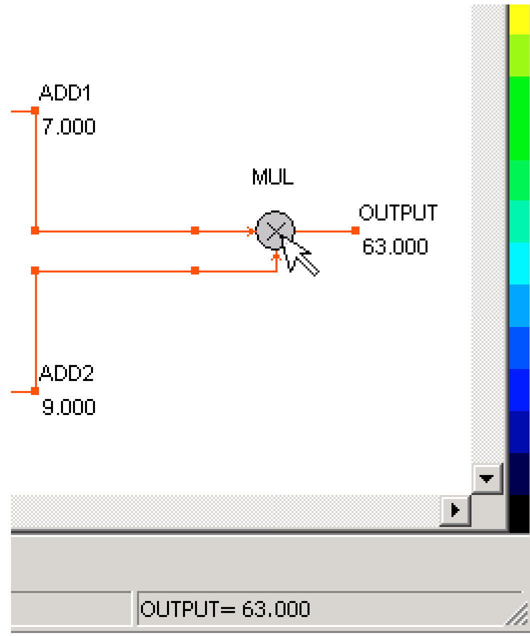
Repeat the same for the second constant block that has as name Constant2, as parameter p1=4, and as identifying remark Constant nr 2.

You can edit the remark of the multiplier block MUL and call it Multiplier 1.

You probably notices that the output nodes of the blocks changed to the name given inside the block properties dialog box..
Connect the blocks by drawing wires from the output of the constant blocks to the input of the two add blocks as indicated in the first schematic of the block diagram. Do this by clicking and releasing the output node of the block with the left mouse button. Drag the mouse to the node where the wires have to end and click an release the left mouse button once again. The wire will be shown. Draw all the remaining wires. The common input of the two

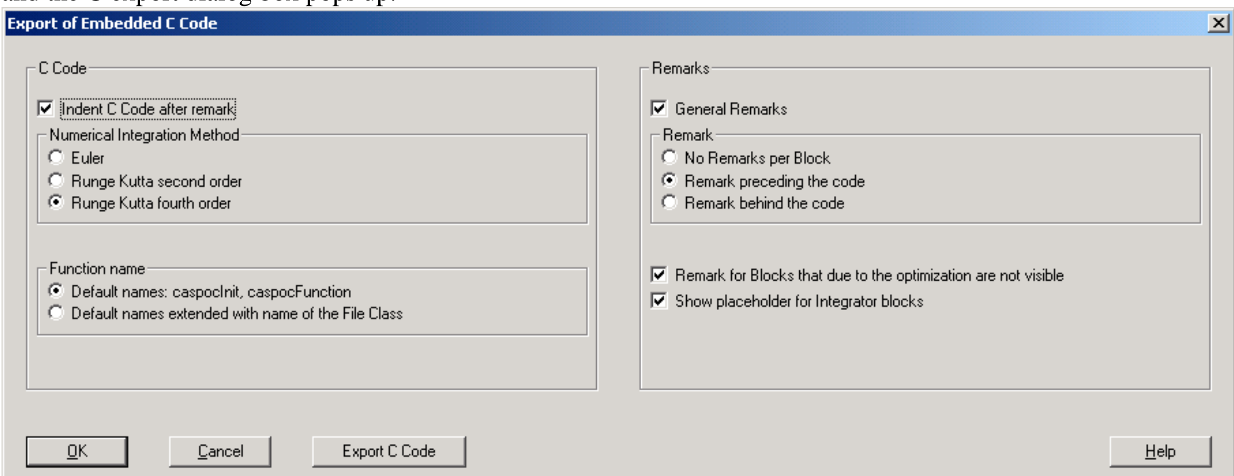
adders has to be set to the numerical value of 4. Do this by first drawing the wire and when the wire is finished, you can add a numerical value to the node. Note that when a node in the block diagram is numerical, its value is used in the simulation. Right clicking the node and input the value 4 in the edit field for the label. Closing the node properties dialog box shows the value of 4 at the node.

To check if the block diagram is performing correctly press the start button from the upper toolbar. The simulation starts and if all animation options are turned on, you will see the numerical result (3+4)*(4+5)=63 at the output of the block MUL.
If you would have your animation options turned off, you could check the result from the simulation by moving the mouse over the multiplier block.

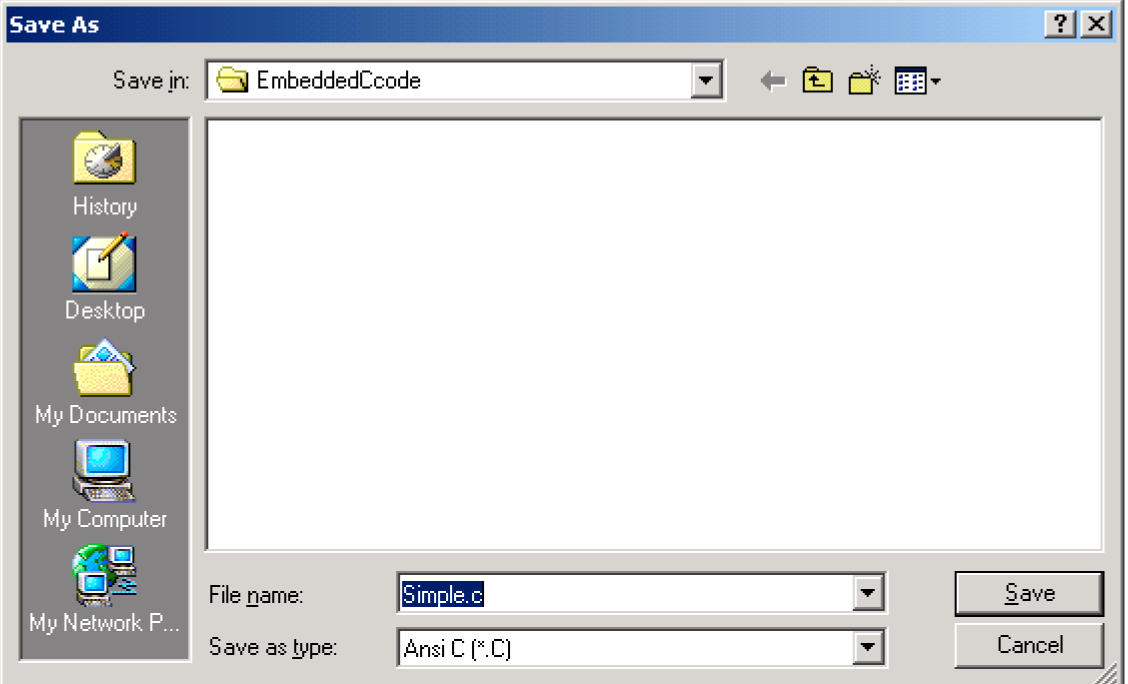


If this is working correctly you are ready to export the C code.
Save the sample under the name simple.csi. This file name will also be used for the file name of the exported c and h file.
Select the export C code button

and the C export dialog box pops up:



For this sample any option is ok. We leave most remark options checked and the numerrical integration method is not of interest, since no integrator blocks are used in this sample. Click the button [Export C Code] and store the file under the name of simple.c



the corresponding simple.h file is saved under the same file name simple.h
The c and h file are stored next to the simple.csi example

To view the exported c and h file click the buttons and the corresponding c or h file will open in your favorite text editor.

Lets have a look at the export C code.

Simple.c - Notepad

File Edit Format Help

```
/* Exported Code */
// static double CONSTANT1; // Remark = Constant nr 1
// static double CONSTANT2; // Remark = Constant nr 2
// static double ADD2; // Remark = Adder 2
// static double ADD1; // Remark = Adder 1
  static double OUTPUT; // Remark = Multiplier 1
/* Declaration Local Private Variables */

#include "Simple.h"
#include <stdlib.h>
#include <math.h>
#include <float.h>
void caspocInit()
{
}

void caspocFunction()
{
// static double OUTPUT(MUL             )Remark = Multiplier 1
  OUTPUT=((3)+4)*(4+(5));
  t=t+h;
}
```

Simple.h - Notepad

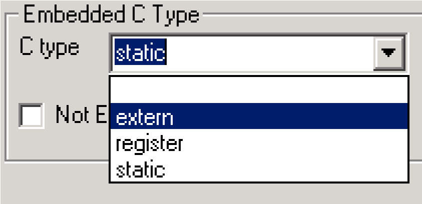
File Edit Format Help

```
/* Declaration External variables that are not outputs from the Block-diagram */
/* Declaration Global Public Variables */

void caspocInit()
void caspocFunction()
```

As you can see from the exported code, the output in the Caspoc function equals

OUTPUT=((3)+4)*(4+(5));

Which is correct according to our block diagram. The only problem is that the variable OUTPUT is declared as a static double with the simple.c file. This means that his scope is limited within the simple.c file and that it cannot be accessed from a c file that would include the compiled simple.obj file.
The variable OUTPUT has to be an external variable that is declared in the simple.h file. We do this by selecting the properties dialog box of the multiplier block MUL with the output label name OUTPUT. Right click this block with the right mouse button to open the properties dialog box and select the type of the variable.
Select the external type instead of the static declaration:



Close the properties dialog box and select the export C code button to export the code again

The C export dialog box pops up where you select the export button. Overwrite the previous simple.c file.

Simple.c - Notepad

File Edit Format Help

```
/* Exported Code */
// static double CONSTANT1; // Remark = Constant nr 1
// static double CONSTANT2; // Remark = Constant nr 2
// static double ADD2; // Remark = Adder 2
// static double ADD1; // Remark = Adder 1
/* Declaration Local Private Variables */

#include "Simple.h"
#include <stdlib.h>
#include <math.h>
#include <float.h>
void caspocInit()
{
}

void caspocFunction()
{
// extern double OUTPUT(MUL             )Remark = Multiplier 1
  OUTPUT=((3)+4)*(4+(5));
  t=t+h;
}
```

The declaration of the variable OUTPUT disapeared from the simple.c file, but appears now in the simple.h file.

Simple.h - Notepad

File Edit Format Help

```
/* Declaration External variables that are not outputs from the Block-diagram */
/* Declaration Global Public Variables */
extern double OUTPUT; // Remark = Multiplier 1

void caspocInit()
void caspocFunction()
```

Because the simple.h header file is included in the simple.c file, the variable OUTPUT is accessable in the simple.c

file. Because it is declared as an external in the header file simple.h , the variable can now be linked to your main applications main.c file.

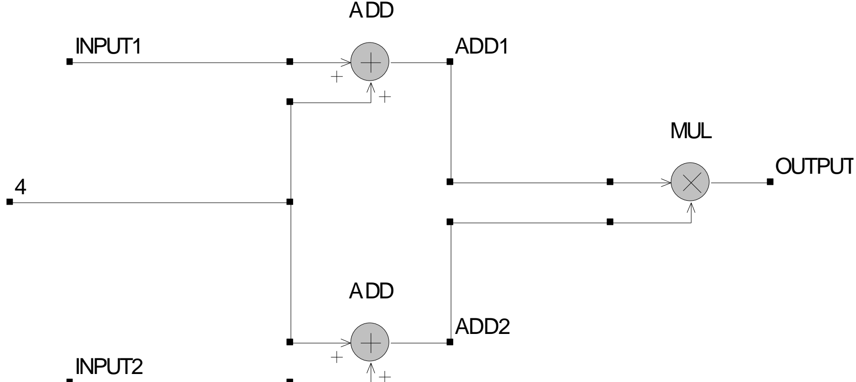
```
#include "simple.h"
main()
{
caspocInit();
caspocFunction();
printf("The result is %f",OUTPUT);
}
```

If you would compile and run the above program, it will respond

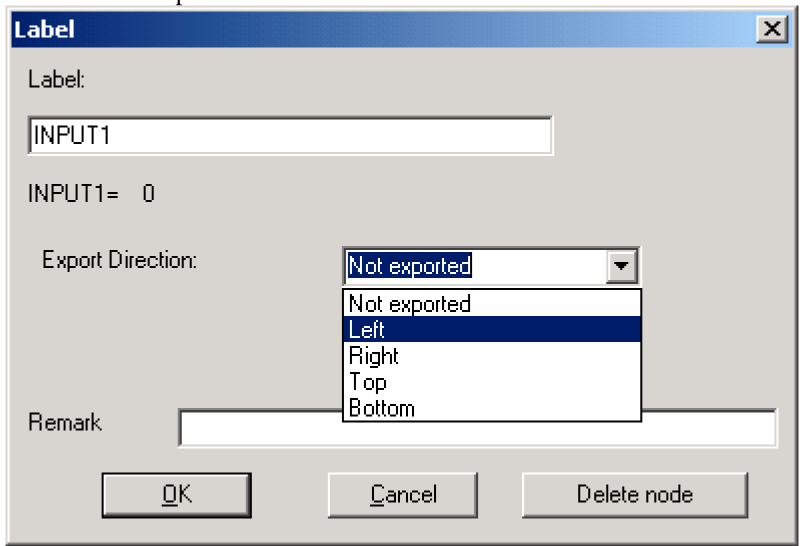
```
The result is 63
```

Arithmetic sample

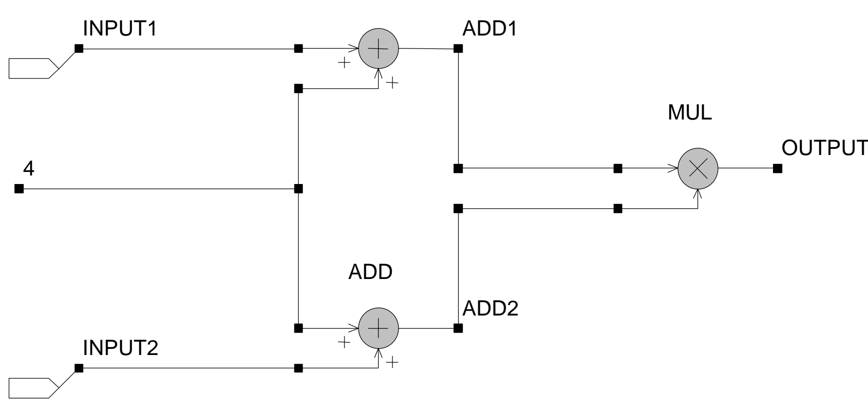
In the previous sample we exported a simple signal calculation, but the inputs to this system were constants. We would like to have time varying inputs like measured signals such as rotor speed or output voltage that has to be controlled.
We could remove the constant blocks CON, and consider these signal as inputs. Consider the inputs INPUT1 and INPUT2, where compared to the previous sample the blocks CON are removed. To make the inputs time variable, they have to be exported as external



However we can only define an external declaration inside the components properties dialog box. Another way to define the external declaration is to define the node as exported. To do this, select the node by clicking it with the right mouse button and select and export direction.



Do the same for the second input INPUT2 and in the schematic the export direction is shown as below.



Save the block diagram under the name of simple2.csi and if the block diagram is exported, the two inputs are exported as external.

Simple2.c - Notepad

File Edit Format Help

```
/* Exported Code */
// static double ADD1; // Remark = Adder 1
// static double ADD2; // Remark = Adder 2
/* Declaration Local Private Variables */

#include "Simple2.h"
#include <stdlib.h>
#include <math.h>
#include <float.h>
void caspocInit()
{
}

void caspocFunction()
{
// extern double OUTPUT(MUL              )Remark = Multiplier 1
  OUTPUT=(INPUT1+4)*(4+INPUT2);
  t=t+h;
}
```

The declaration of the two inputs appears in the simple2.h file.

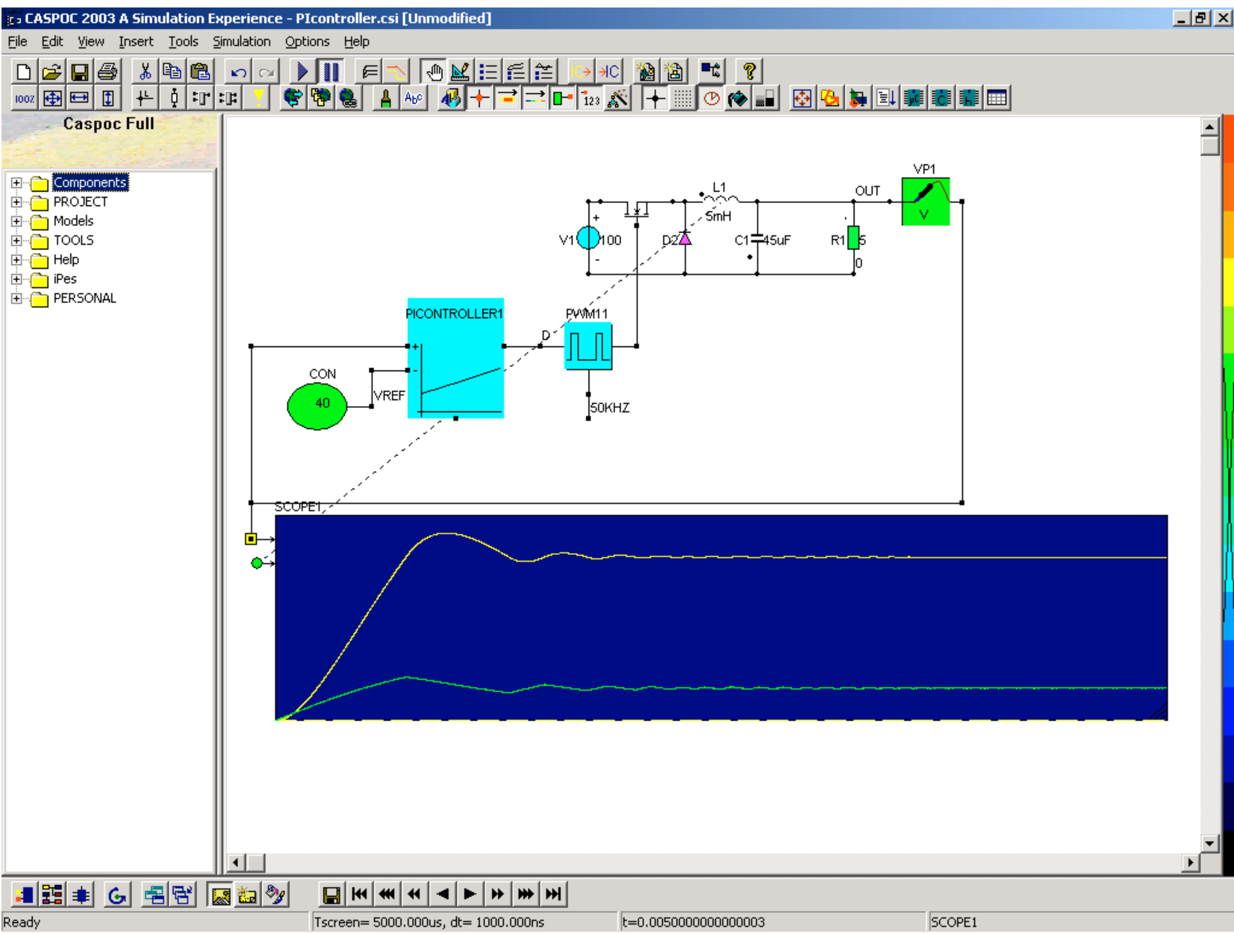
Simple2.h - Notepad

File Edit Format Help

```
/* Declaration External variables that are not outputs from the Block-diagram */
extern double INPUT1; // External input Block = ADD              // Remark = Adder 1
extern double INPUT2; // External input Block = ADD              // Remark = Adder 2
/* Declaration Global Public Variables */
extern double OUTPUT; // Remark = Multiplier 1

void caspocInit()
void caspocFunction()
```

Analog PI controller

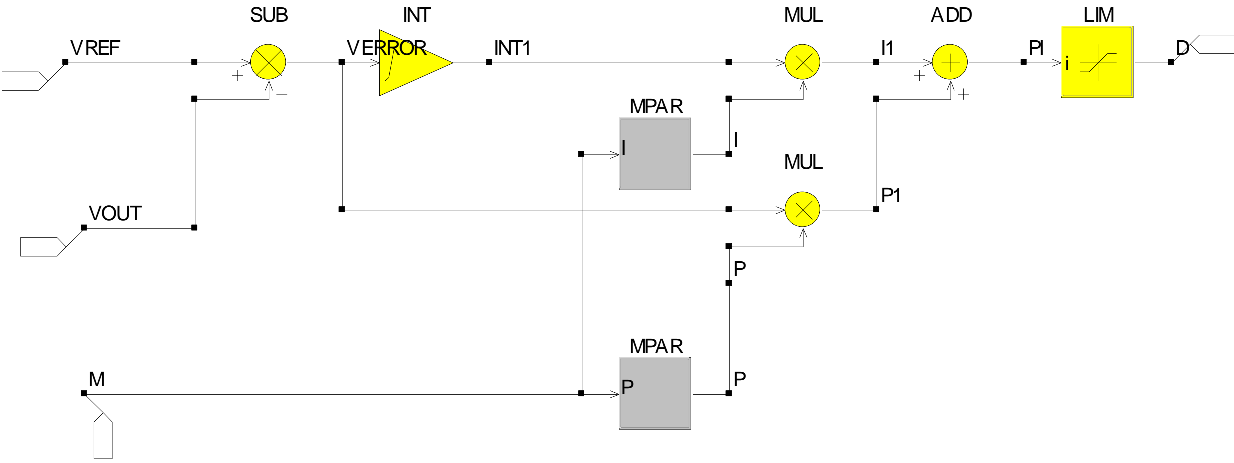
In this example we will export the C code for a simple analog PI controller for a Buck converter. In order to keep the sample simple, basic blocks are used to build to PI controller. More detailed parts are omitted such as the anti-wind up of the integrator. The schematic below shows the start-up of the buck converter and in the scope the yellow trace represents the output voltage on node OUT and the green trace shows the inductor current i(L1)



Our goal is to export the code for the PI controller that is modeled inside a library block. If we export only the contents of the PI controller block, other blocks are not included in the exported code, such as the reference voltage Vref=40volts and the PWM modulator. These components can be the standard components on the microprocesor board. We will interface to these components on the board by means of the externally declared variables representing the measured output voltage and refernce voltage from an ADC and the setting dutycycle signal for the PWM modulator on the board.

If we open the PI controller library block with a right mouse click and choose edit, the block diagram of the PI controller appears.

Notice in this block diagram that there is one integrator representing the integration function, the two multipliers that multiply the input with the given P and I and after the addition of both signals with the ADD block a limiter LIM block limits the duty cycle between 0.1 and 0.9. (Remember that for the sake of simplicity we didn't model the anti-windup in the integrator)



Because inside the library block the nodes are defined that are interface with the rest of the block diagram, they are marked with a flag showing the direction of the exxported label. The inputs VREF and VOUT are such nodes and appear on the left side of the library block. The output of the PI controller, the duty cycle D, appears on the right side of the library block. The input M is only used for reading parameters from a database and is not used in this sample.
Notice the use of the MPAR blocks that allow you to define constants that can be edited via the library block properties dialog box. The default values will be exported, so be sure to set these parameters to the desired value before exporting the schematic. In this sample they are both set to 10.
Export the C code for the PI controller and save the c file under PICONTROLLER.C

After the export is performed we can ope nthe piccontroller.h file to see which variables are declared as external.

PICONTROLLER.h - Notepad

File Edit Format Help

```
/* Declaration External variables that are not outputs from the Block-diagram */
extern double M; // External input Block = MPAR
extern double VREF; // External input Block = SUB
extern double VOUT; // External input Block = SUB
/* Declaration Global Public Variables */
extern double D;

void caspocInitPICONTROLLER(double t, double h)
void caspocFunctionPICONTROLLER(double t, double h)
```

All nodes that are exported from the library block for interfacing with the rest of the block diagram are declared as external variables as seen in the picontroller.h file.

To interface with this code we could create a main.c file that would look like

```
#include "picontroller.h"

static double VOUT; // Output voltage
static double VREF; // Reference voltage
static double D;    // Duty cycle for the PWM
static double t;    // This is the running time

main()
{
_InitYourBoard();                // Initialize your board settings
caspocInitPICONTROLLER(0,1e-6);  // Initialize the variables
                                 // in the PIcontroller



while(true)
  {
  VREF=_GetAnalogSignalFromChannel(1); // your ADC interface
  VOUT=_GetAnalogSignalFromChannel(2); // your ADC interface
  caspocFunctionPICONTROLLER(0,1e-6);  // call the PI controller
  _SetPWMDutyCycle(D);                 // set the PWM duty cycle
  }
}
```

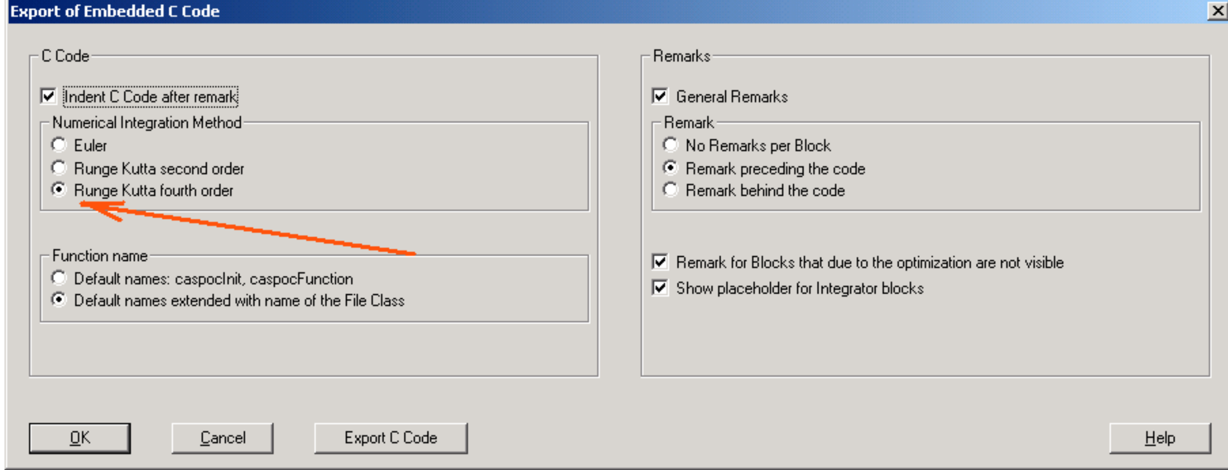
First the external variables from the header file PIcontroller.h are declared as static double. Inside the main function, the initialization of the board and a call to caspocInit() to initialize the variables used in the PI controller are done. The while loop represents a continuous loop that calls the PI controller continuously. (Be aware that in a real application this loop has to be synchronized with the real time.)
The analog signals representing the output voltage from the buck converter and the reference voltage are read from the AD converters on the board. As they are declared as externals, their value is also know inside the caspocFunctionPICONTROLLER function. The duty cycle calculated is stored in the external variable D inside the caspocFunctionPICONTROLLER function. You use this variable to set the duty cycle in your PWM modulator on the board.

```
/* Exported Code */



  static double INT1;
  static double VERROR;
/* Declaration Local Private Variables */

#include "PICONTROLLER.h"
#include <stdlib.h>
#include <math.h>
#include <float.h>
void caspocInitPICONTROLLER(double t, double h)
{
  INT1=0;
}

void caspocFunctionPICONTROLLER(double t, double h)
{
  // integrator:INT1
  VERROR=VREF-VOUT;
  D=1*((INT1*(10))+(VERROR*(10)));if(D> 9.00000000000000E-0001)D= 9.00000000000000E-0001;if(D< 1.0000

// integrators
  INT1=INT1+VERROR*h; //  yn+1=yn+h*input+O(h^2)

//Final results
  INT1=INT1;
  // integrator:INT1
  VERROR=VREF-VOUT;
  D=1*((INT1*(10))+(VERROR*(10)));if(D> 9.00000000000000E-0001)D= 9.00000000000000E-0001;if(D< 1.0000
  t=t+h;
}
```

The picontroller.c file shows that, due to optimization, only the variables VERROR and INT1 are declared as variables inside the caspocFunctionPICONTROLLER function. The value of the variable D is limited between 0 and 0.9.

The used numerical integration method is Backward Euler.

More precise results can be achieved by using the Runge Kutta fourth order integration method. This option can be set in the options dialog box for the export of C code.



Export the C code again and the picontroller.c file will show that now the code is including the numerical Runge Kutta fourth order integration method.

```
// Runge Kutta Step: 1
  // integrator:INT1[0]
  VERROR[0]=VREF[0]-VOUT[0];
  D[0]=1*((INT1[0]*(10))+(VERROR[0]*(10)));if(D[0]> 9.00000000000000E-0001)D[0]= 9.0000000
  _k_INT1[0]=h*VERROR[0]; // Evaluate k1
  INT1[1]=INT1[0]+_k_INT1[0]/2; // integrator output with k1
```

Texas Instruments IQMath library

Texas Instruments provides a special library with scaling to fixed point and tabulated functions for their DSP. You can take advantage of this library by using fixed point calculations in Caspoc and exporting the IQMath library functions from the block diagram.
All the functions from the IQMath library are modelled in Caspoc. Their implementation in the library is modelled in the same way in Caspoc, meaning that the functionality during simulation in Caspoc is equal to the functionality in the final C code. For example the sinusoidal functions are represented by their polyninomial approximations. For more info on the IQMath library from Texas Instruments we recommend you to visit the TIO web site and to download the manual for appling the IQMath on their DSP.

To use Fixed Point simulations, the analog signals have to be scaled to fxed point data types. For each block you can set the C declaration type for the variable. For example if you would only use 8 bits, you could set the C type to _int8. In general the scaling is depending on the amplitude of signals and it varies per control system.

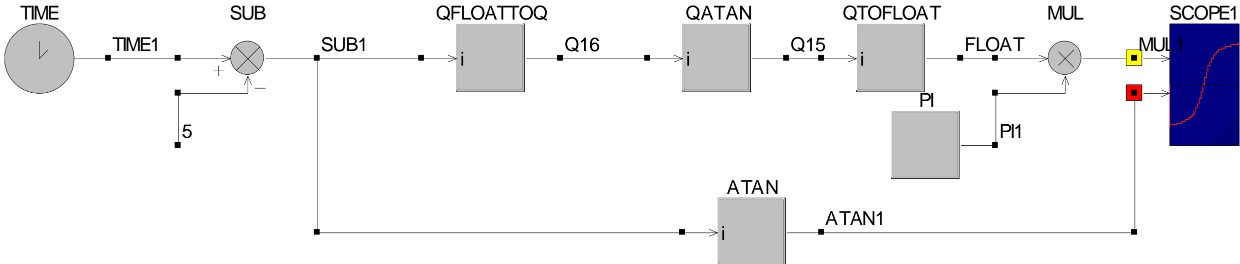
For use with the IQMath library the scaling has to be done using the scaling functions from the IQMath library. The used scaling is inidcated by the Q format that defines which scaling is applied. For each block the Q format is defined and the output from the block can have another Q format. Therefore use the block QFLOATTOQ and QTOFLOAT to scale from floating point into the fixed point Q format and back into the floating point data format. For example the QLog10 block has a Q16.16 format input and the output is in the Q4.12 format.

C28x QMATH LIBRARY BENCHMARKS

Function Name	Execution Cycles	Accuracy	Program Memory	Input format	Output format
QSIN	33 cycles	15 bits	31 words	Normalized Q1.15	Signed Q1.15
QSINLT	25 cycles	14 bits	17 words	Normalized Q1.15	Signed Q1.15
QCOS	33 cycles	15 bits	32 words	Normalized Q1.15	Signed Q1.15
QCOSLT	25 cycles	14 bits	19 words	Normalized Q1.15	Signed Q1.15
QATAN	65 cycles	15 bits	50 words	Signed Q16.16	Normalized Q1.15
QSQRT	45 cycles	14.5 bits	44 words	Unsigned Q16.16	Unsigned Q8.8
QLOG10	38 cycles	15.5 bits	38 words	Unsigned Q16.16	Signed Q4.12
QLOGN	38 cycles	15.5 bits	38 words	Unsigned Q16.16	Signed Q5.11
QINV1	51 cycles	32 bits	15 words	Signed Q(x)	Signed Q(31-x)
QINV2	35 cycles	16 bits	14 words	Signed Q(x)	Signed Q(15-x)
QDIV	54 cycles	32 bits	16 words	Signed Q(x), Q(y)	Signed Q(16+x-y)

Notes:
1. Execution Cycles mentioned in the table includes the CALL and RETURN (LCR + LRETR).
2. QCOSLT and QSINLT functions use 256-point look-up table.
3. Branch instruction is totally eliminated in all the above functions.

The sample below shows the simulation of the Arctan function. The Arctan function is modeled using the IQMath library and also using the regular ATAN block from Caspoc. The results are compared in the scope. Note that the output from the QATAN block is scaled by 1/p. Therefore the output is multiplied with p. During the simulation the input (the running time t-5) is scaled into the Q16.16 format and after calculation by the QATAN block scaled back from the Q1.15 format into floating point. The output from the QTOFLOAT block is multiplied with p to account for the internal scaling of the QATAN block.



If we export the block diagram into C code, the scaling into the Q format is performed by the ANSI-C ldexp function. Due to optimization, the output from the QATAN function is removed from the code and integrated in the line for the MUL1 output.

aTan.c - Notepad

File Edit Format Help

```
/* Exported Code */
  static double SUB1;
  static double ATAN1;
  static double MUL1;
/* Declaration Local Private Variables */

#include "aTan.h"
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <qmath.h>
void caspocInitaTan(double t, double h)
{
}

void caspocFunctionaTan(double t, double h)
{
  SUB1=(t)-5;
  ATAN1=atan(SUB1);
  MUL1=(ldexp((qatan((ldexp(SUB1,16)))),-15))*(pi);
  t=t+h;
}
```

Notice the include of the <qmath.h> header file that is required when compiling for the DSP. The qmath.h header and qmath assembly file are supported by Texas Instruments.