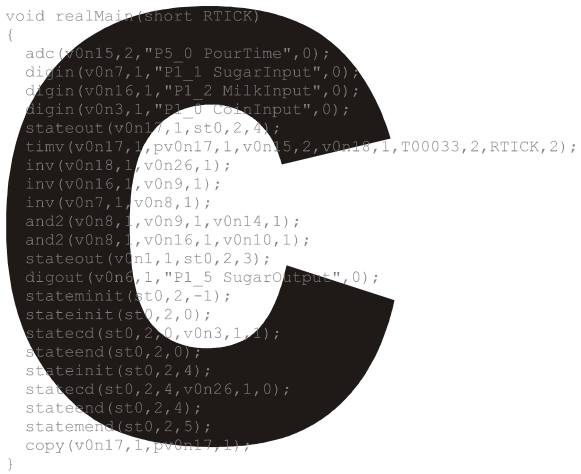
Application note on REALIZER

Title: Generating C-code
Subject: Generating C-code with Realizer
Date: 23 January 2001
Our ref.: 0373900
Author: Mark Bruijn

Introduction

This document describes how to generate C-code with Realizer.

With Realizer, you can select your microcontroller and Realizer will generate complete, ready-to-run code for this microcontroller. The generated code will take care of initialising the microcontroller and reading the inputs and setting the outputs of the microcontroller. It also keeps track of time-related variables that are used in oscillators and delays, etc. All this is invisible to the user: just download the generated code into your microcontroller and execute it.



When you generate C-code with Realizer, you do not select a microcontroller but you select the C-code generator instead. The generated C-code can not read inputs or set outputs for you, because Realizer does not know how to do that: this is specific for the hardware you are generating the C-code for. That's why you have to supply your own functions (written in C) for this. And that's why you have to supply your own "main" function too. Your "main" function uses the functions you supply by yourself (e.g. for reading and setting the inputs and outputs) as well as the functions Realizer generates for you.

This document contains an example of how these functions should look like. Change the functions accoording to your hardware and then you can use Realizer to generate C-code for your hardware too.

Disclaimer

The information in this document is believed to be accurate and reliable. However, Actum Solutions assumes no responsibility for any consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use.

No license is granted by implication or otherwise under any patent or patent rights of Actum Solutions.

Specifications mentioned in this document are subject to change without prior notice. This publication supersedes and replaces all information previously provided.

Actum Solutions products are not authorized for use as critical components in life support devices or systems without express written approval of Actum Solutions.

Realizer® is a registered trademark of Actum Solutions.

All other trademarks mentioned herein are the property of their respective companies.

Please visit our web site at http://www.actum.com for more information. You can also send us an e-mail at info@actum.com or contact us directly:

Actum Solutions
Industriestraat 9A
1704 AA Heerhugowaard
Netherlands
Tel. +31 (0) 72 576 2555
Fax. +31 (0) 72 576 2559

Overview of generating C-code

When writing a program in C-code for a certain microcontroller, you not only need to develop the C-code, but you need to compile and link your C-code into machinecode too. For the compiling and linking, you need to know about the compiler and linker you use, about make-files, etc.

When your program becomes larger, you probably use more files for your C-code. And maybe you have a library of functions that you use more often. In these cases, you have to take care of include-files and compile and link those include-files too.

Using Realizer to generate part of the C-code for you, does not change this: you have to take care of compiling and linking different files. A common project will have the following files:

beverage.c: the C-code Realizer generates for you, assuming your project is named "beverage".

beverage.h: an include-file Realizer generates for you, containing definitions used in beverage.c.

clib.h, cpers.h, cslp.h, clib.c, cpers.c, cslp.c: files that included by beverage.c, containing the definitions of the functions used in beverage.c. Those files are located in the library-directory (...\Realizer\Lib).

ros552.c: a file containing the C "main" function and functions for handling the I/O. This C-code in this file is compiler-dependent and you have to write similar C-code for your microcontroller and for your compiler.

ros552.bat: a batch-file containing all commands to make the hex-file out of all source files. You have to write this file and tell Realizer to use this file when analysing your project.

So the files you have to write yourself are ros552.c and ros552.bat. You can name these files as you like, just state the correct names in your batch-file. In this example the file is named ros552.c because it contains a Realizer Operating System (ros) for the 80552 controller. Included in this application note are ros552.bat, ros552.c, beverage.c and beverage.h.

The Realizer project

The example used in this application note is the beverage dispencer. The schematic is included in this application note. The application is a common coffee-machine and has the following features:

First you select if you want sugar and/or milk in your coffee (inputs "SugarInput" and "MilkInput"). After that you insert a coin in the machine (input "CoinInput"). The machine will drop a cup (output "CupOutput"), drops sugar and/or milk in it (outputs "SugarOutput" and "MilkOutput") and poors the coffee in it (output "PourOutput" will be active for a time that the service-engineer can set with the pour timer, input "PourTime").

A detailed description of this example can be found on our website, the application itself won't be discussed in this application note.

If you use Realizer to generate code for a certain microcontroller, you select the microcontroller (menu-item "Project-Hardware select"), and you connect the I/O symbols with a certain pin/device of your microcontroller (by double clicking the I/O symbols like "digin", "digout", etc.). When analysing your project, Realizer knows how to make the hex-file for your microcontroller: it knows how to call the correct compiler/linker.

If you use Realizer to generate C-code, you select the "C-code generator" instead of a certain microcontroller (menu-item "Project-Hardware select"). Now you can't connect the I/O symbols with the pins/devices of your microcontroller: Realizer has no data on what microcontroller to use. Before analysing your project, you have to tell Realizer how to compile/link the source files. You do this via menu-item "Project-Hardware settings", the setting for "Compiler". Change this into the name of your batch-file that makes the hex-file, "ros552.bat" for example. Do not check the setting "Complete ROS" for now.

Now you can analyse your project. Realizer will generate the .c and .h file for your project, for example "beverage.c" and "beverage.h". After that, Realizer will call your batch-file to make the hex-file out of all your source files. As stated before, you have to write this batch-file (e.g. "ros552.bat") by yourself. You also need to write a file (e.g. "ros552.c") containing the C "main" function and functions for handling the I/O.

Your C file

Realizer does generate C-code for your project, but you need to write your own C-code too. The C-code Realizer generates, defines two functions: “realMain” and “realInit”. You have to write C-code that uses those functions (the “main” function) and that defines functions for the I/O symbols (e.g. “digin”, “digout”, “adc”).

You have to do this only once for each type of microcontroller that you use: when you start a new project with the same microcontroller, the way you write to a digital output won’t change. And the “main” function won’t change too. So the C-code that you write is an interface between the C-code Realizer generates for you and the hardware that you use.

I/O functions

In this application note, “ros552.c” is included. This is an example of a file that you have to write yourself and is specifically written for the 80C552 microcontroller and the Tasking CC51 V4 C-compiler. You can change this file to suit your own platform. The file defines the following procedures that are used in the C-code that Realizer generates:

- digin
- adc
- digout
- dac

The procedures all have a similar interface, for example procedure “digin”:

```
void digin(long out, short typeOut,
char *ioPort, short typeIoPort)
```

out the Realizer variable (the outputpin of the digin-symbol) where you have to write the read input to. All Realizer variables are stored in an array of a certain type, for example:

- unsigned char realBit[15]
- unsigned char realByte[3]
- unsigned char realNonVolByte[1]

When accessing a certain variable, you must access the correct index in the array, for example: realBit[out]. The type of the variable (“typeOut”) determines which array you should use (realBit, realByte, realNonVolByte, etc).

typeOut the type of the Realizer variable. The following types are used (defined in clib.h):

- TypeNoCon 0
- TypeBit 1
- TypeUByte 2
- TypeSByte 3
- TypeUInt 4
- TypeSInt 5
- TypeLong 6
- TypeEvent 7
- TypeConst 100
- TypeConstBit 101

ioPort the name of the I/O port. This is the same name you used for the digin symbol. You can use names like “SugarInput” for this, but you can also use names like “P1_1 SugarInput”. You then can check for the first 4 characters to determine the port you want to read. In this way you can write an application independent C-file and have to write this file only once.

typeIoPort the type of the I/O port. See “typeOut” for a list of used types.

See the included “ros552.c” for the implementation of the procedures. Please note the following:

The names used for the I/O symbols in the Realizer schematic, includes the port of the microcontroller that the I/O symbol is connected to. For example “P1_1 SugarInput” instead of just “SugarInput”. By doing this you can check for the first 4 characters to determine which input to read (“P1_1” in this case). That why “digin” contains code like:

```
if (strncmp(ioPort, “P1_1”, 4) ==0)
realBit[out] = P1_1;
```

Timer function

The C-code you write not only contains definitions for procedures used by Realizer, it also provides a timer that is used by Realizer. Inside your Realizer schematic, you can use delays, oscillators and other time-related symbols. To implement these time-related symbols, Realizer needs access to a timer. See the definition and usage of the “realMain” procedure: the time passed (expressed in “ticks” of 10 msec) since the last execution of “realMain” is a parameter of “realMain”.

```
void realMain(short RTICK)
```

To generate the timerticks of 10 msec an interrupt routine is used that increments a variable each 10 msec. See the implementation of “tickIntHandler” in “ros552.c”.

Just before “realMain” is executed, the number of timerticks is read (copied into another variable) and reset. This gives us the number of timerticks since the last execution of “realMain”. See the implementation of “initTimerTick” and “readTimerTick” to see how this is done. For the usage of these functions, see the implementation of “main”.

Watchdog

Another function you might want to include in your C-code is the usage of the watchdog. To implement this, you have to set the watchdog each time “realMain” is executed. Setting the watchdog is implemented in procedure “setWatchdog”. See the implementation of “main” for the usage of this function.

Additional I/O routines

Sometimes it will be necessary to write additional routines to access the I/O of your microcontroller. In “ros552.c”, additional routines are necessary to scan and read the eight ADCs.

Scanning and reading the ADCs is done by an interrupt routine. The routine is executed after the AD conversion is finished and an interrupt is generated. The routine stores the value of the ADC in an array “adcValues”, and starts scanning the next ADC channel (by starting an AD conversion for the next channel). If all eight channels are read, the AD conversion is not started and the scanning and reading of the ADCs will stop. Scanning and reading of the 8 ADCs will start again when procedure “restartADC” is executed.

See procedure “ADCIntHandler” to see how the interrupt routine is implemented. See procedure “restartADC” and “main” for the implementation and usage of scanning and reading the ADCs.

The values that are stored in the array “adcValues” are used by the “adc” procedure. This procedure reads an analog value and writes this into the correct Realizer variable. See procedure “adc” for the implementation of this. See the explanation above for the interface of procedure “adc”.

This concludes the C-code you have to write yourself. It now contains everything you need to compile it into executable machinecode. See enclosed “ros552.c” for an example of the complete C-code you have to write for your microcontroller and your compiler. Please note that “ros552.c” is especially written for the 80C552 microcontroller and the Tasking CC51 V4 C-compiler. It contains instructions that are specific for this compiler.

Your batch-file

Now you’re ready to compile and link your C-files into machinecode. You can do that by hand, but you can also create a batch-file containing all the statements you need. These statements are dependent of the compiler and linker you use. Included “ros552.bat” is an example for the Tasking CC51 V4 C-compiler.

The batch-file takes care of:

- compiling “cstart.asm”. This is a file supplied by Tasking, necessary to set some options.
- compiling “ros552.c”. Your own C-file.
- compiling “beverage.c”. The C-file generated by Realizer.
- compiling “clib.c”. A file supplied by Actum Solutions, used by “beverage.c”.
- compiling “cpers.c”. A file supplied by Actum Solutions, used by “beverage.c”.
- linking. See enclosed “@ros552.lnk” to see the linker directives.
- generate a hex-file.

The batch-file contains statements specifically for this project (e.g. compiling “beverage.c”). So you have to change your batch-file for each new project.

This concludes the process of generating C-code with Realizer and compiling it into ready-to-run machine-code. Besides generating machine-code, you can generate code for simulation purposes too. This involves generating machine-code for your PC instead of your microcontroller, so you need another compiler for this. A description of how to do this follows below.

Overview of simulating with C-code

Using Realizer, it is possible to simulate your project. When you select a microcontroller, then at analysing your project, Realizer generates code for this microcontroller. When simulating this project, Realizer uses an interpreter (a simulation DLL) to interpret the generated code, set breakpoints and read and set variables in the code. By doing this, Realizer can simulate your project using the same code that you download into your microcontroller.

When selecting the C-code generator instead of a microcontroller, the procedure is somewhat different. Because there is no machinecode generated and there is no matching interpreter, Realizer can not use a simulation DLL. Instead of this, Realizer compiles your project into machinecode for your PC. Realizer uses a known compiler for this, so it knows how to execute the compiler and what parameters to use. The result of compiling your project will be a DLL that has the same functionality as the simulation DLL that Realizer normally uses, so now it is possible to simulate your project.

As described above, Realizer needs a compiler to be able to simulate your project. **You need to obtain and install this compiler yourself**. The compiler you need is the Borland C++ 5.5 Compiler from Inprise. You can download this compiler for free from Inprise's website at http://www.borland.com. Select "Downloads" and then "The Borland C++ Compiler 5.5 Download" under "C++Builder". After downloading the file, do not install it in the default directory, but in the "Compiler" subdirectory of your Realizer directory (by default "C:\Program Files\Realizer Gold\Compiler").

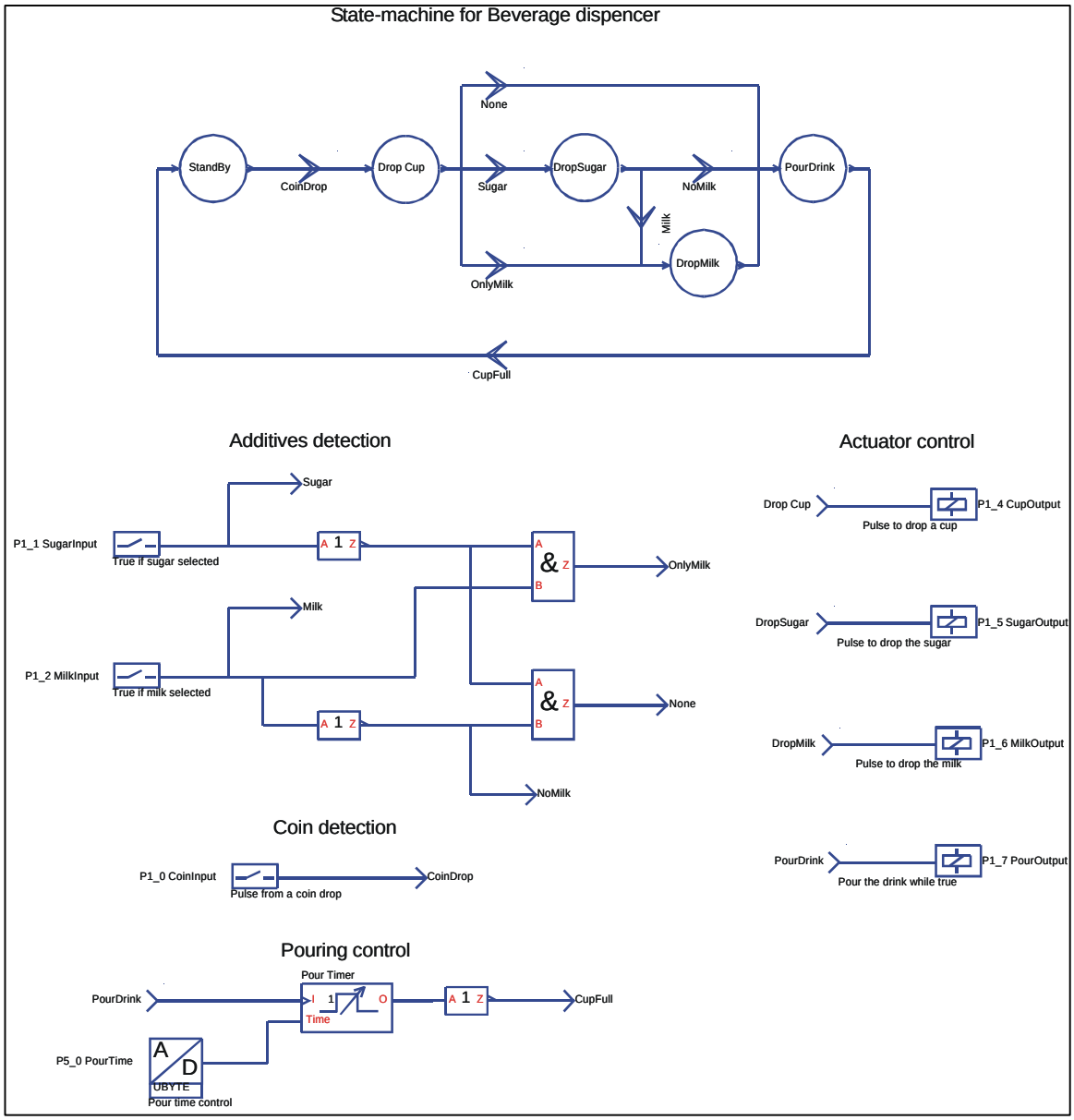
Realizer comes with a make-file that expects the Borland C++ 5.5 compiler in the specified directory. The make-file that Realizer uses is named "template.mak" and can be found in the compiler directory.

You can't use another compiler by just changing this make-file. Realizer generates additional C-code (not just "realmain" and "realinit") to create a complete program that can be simulated. This additional C-code is compiler-dependant.

Simulating

Now that you've installed the compiler, you can simulate your project. First, you have to select the "C simulation engine" instead of the "C-code generator" (menu-item "Project-Hardware select"). Then analyse your project and after that you can execute the simulator. The pin-level simulation is not supported because Realizer doesn't know what microcontroller to use. However, you can use all the features of the functional simulation.

Realizer schematic of the beverage dispencer, beverage.sch:



Realizer Operating System for the 80552 controller, ros552.c:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *   (c) Copyright Actum Solutions 1990.. 2000. All Rights Reserved.
 *
 *   File:          ros552.c
 *   Language:      80552 C
 *   Compiler:      Tasking CC51 V4
 *   Description:   Realizer Operating System for a 80C552
 *
 *   Revision log:
 *         Ver.  Date    Id    Description
 *         1.00  960201  RPJK  First release
 *         1.01  000327  MBR   Names of functions
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "stdlib.h"
#include "string.h"

typedef unsigned  char byte;          /*  8 bit type for common use         */
typedef unsigned  int  word;          /* 16 bit type for common use         */

#define TRUE      1
#define FALSE     0

#define DOGTIME   0x00
#define Lo10ms    0x00
#define Hi10ms    0xDC

rom byte copyright[]  = "ROS552 [rev 1.01], (c) Copyright 1992,00 by Actum Solutions";

/*
 * External functions generated by the Realizer
 */

extern void realInit(void);
extern void realMain(short);
extern unsigned char realBit[];
extern unsigned char realByte[];
extern void setByteValue(long out, short typeOut, long value);
extern long convertByteValue(long in, short typeIn);

/*
 * Variables used by ROS
 */

word adcValues[8];      /* array to store the Analog input values */
byte chnl;              /* Current AD channel */
byte rlzTick;           /* variable to store the last 10 msec ticks */
byte tick;              /* variable to store the last 10 msec ticks */

/*************************************************************
 *
 *  void digin(long out, short typeOut, char *ioPort, short typeIoPort)
 *
 *  description:  Operating system specific interface function
 *                Interfaces with a DIGIN symbol by means of the NAME attribute
 *
 */
void digin(long out, short typeOut, char *ioPort, short typeIoPort)
{
  if (strncmp(ioPort,"P1_0",4) == 0)
    realBit[out] = P1_0;
  else if (strncmp(ioPort,"P1_1",4) == 0)
    realBit[out] = P1_1;
  else if (strncmp(ioPort,"P1_2",4) == 0)
    realBit[out] = P1_2;
  else if (strncmp(ioPort,"P1_3",4) == 0)
    realBit[out] = P1_3;
}
```

ros552.c, continued:

```
/************************************************************
 *
 *  void adc(long out, short typeOut, char *ioPort, short typeIoPort)
 *
 *  description:  Operating system specific interface function
 *                Interfaces with a ADC symbol by means of the NAME attribute
 *
 */
void adc(long out, short typeOut, char *ioPort, short typeIoPort)
{
  if (strncmp(ioPort,"P5_0",4) == 0)
    setByteValue(out,typeOut, adcValues[0]);
  else if (strncmp(ioPort,"P5_1",4) == 0)
    setByteValue(out,typeOut, adcValues[1]);
  else if (strncmp(ioPort,"P5_2",4) == 0)
    setByteValue(out,typeOut, adcValues[2]);
  else if (strncmp(ioPort,"P5_3",4) == 0)
    setByteValue(out,typeOut, adcValues[3]);
}

/************************************************************
 *
 *  void digout(long in, short typeIn, char *ioPort, short typeIoPort)
 *
 *  description:  Operating system specific interface function
 *                Interfaces with a DIGOUT symbol by means of the NAME attribute
 *
 */
void digout(long in, short typeIn, char *ioPort, short typeIoPort)
{
  if (strncmp(ioPort,"P1_4",4) == 0)
    P1_4 = realBit[in];
  else if (strncmp(ioPort,"P1_5",4) == 0)
    P1_5 = realBit[in];
  else if (strncmp(ioPort,"P1_6",4) == 0)
    P1_6 = realBit[in];
  else if (strncmp(ioPort,"P1_7",4) == 0)
    P1_7 = realBit[in];
}

/************************************************************
 *
 *  void dac(long in, short typeIn, char *ioPort, short typeIoPort)
 *
 *  description:  Operating system specific interface function
 *                Interfaces with a DAC symbol by means of the NAME attribute
 *
 */
void dac(long in, short typeIn, char *ioPort, short typeIoPort)
{
  if (strncmp(ioPort,"OUT0",4) == 0)
    PWM0 = convertByteValue(in,typeIn);
  else if (strncmp(ioPort,"OUT1",4) == 0)
    PWM1 = convertByteValue(in,typeIn);
}

/************************************************************
 *
 *  void setWatchdog(void)
 *
 *  description:  retrigger watchdog
 *
 */
void setWatchdog(void)
{
  PCON |= 0x10;
  T3 = DOGTIME;
}
```

ros552.c, continued:

```
/*************************************************************
 *
 *  void ADCIntHandler(void)
 *
 *  description:  ADC interrupt routine reads all eight adc channels
 *                stops after reading eight channels
 *
 */
interrupt 10 using 1 void ADCIntHandler(void)
{
  adcValues[chnl++] = ((byte) ADCH) * 4 + _rol((ADCON & 0xC0),2);
  ADCON &= 0xEF;              /* Reset ADC Interrupt flag           */

  if (chnl < 8)
  {
    ADCON = (ADCON & 0xF8) | chnl;
    ADCON |= 0x08;            /* Set ADC Start Conversion bit       */
  }
  else
    EAD = FALSE;              /* all eight channels read, disable adc interrupt
                                 and do not restart the conversion */
}

/*************************************************************
 *
 *  void restartADC(void)
 *
 *  description:  (re)starts scanning the 8 AD channels.
 *
 */

void restartADC(void)
{
  if (chnl == 8)
  {
    chnl = 0;
    ADCON = (ADCON & 0xF8) | chnl;      /* select current channel  */
    ADCON |= 0x08;                      /* Set ADCS bit            */
    EAD = TRUE;                         /* Enable ADC interrupts   */
  }
}

/*************************************************************
 *
 *  void tickIntHandler(void)
 *
 *  description:  10 mSec interrupt routine
 *
 */

interrupt 1 using 2 void tickIntHandler(void)
{
  TL0 = Lo10ms;                       /* 10 ms (@ 11.0592 Mhz)             */
  TH0 = Hi10ms;
  TR0 = TRUE;                         /* Start timer 0                     */
  tick++;                             /* Increment Time                    */
}

/*************************************************************
 *
 *  void initTimerTick(void)
 *
 *  description: Initialize the timer to generate an interrupt every 10 msec.
 *
 */
void initTimerTick(void)
{
  ET0  = FALSE;
  TMOD = (TMOD & 0xF0) | 0x01;        /* timer 0 in mode 1                 */
  ET0  = TRUE;                        /* Enable timer 0 interrupt          */
  TL0  = Lo10ms;                      /* 10 ms (@ 11.0592 Mhz)             */
  TH0  = Hi10ms;
  TR0  = TRUE;                        /* Start timer 0                     */
}
```

ros552.c, continued:

```
/*************************************************************
 *
 *  void readTimerTick(void)
 *
 *  description: Copy the current 10 msec tick counter
 *
 */

void readTimerTick(void)
{
  ET0  = FALSE;         /* Disable timer interrupts        */
  rlzTick = tick;       /* Copy ROS-tick to Realizer-tick  */
  tick = 0;
  ET0  = TRUE;          /* Enable timer interrupts         */
}

/*************************************************************
 *
 *  void main(void)
 *
 *  description:
 *
 */

void main(void)
{
  chnl = 0;                         /* initialize variables */
  tick = 0;
  rlzTick = 0;

  initTimerTick();                  /* initialize 10 Msec timer */

  realInit();                       /* let Realizer init its variables */

  EA = TRUE;                        /* enable all interupts */

  while (TRUE)                      /* do forever */
  {
    setWatchdog();                  /* trigger watchdog */
    restartADC();                   /* restart AD reading */

    readTimerTick();                /* read 10 mSec timer tick */

    realMain(rlzTick);              /* let Realizer calculate one loop */
  }
}
```

make-file for ros552.c, ros552.bat:

```
@echo off
set dos4gpath=c:\apps\cc51_5\bin
set cc51inc=c:\apps\cc51_5\include

c:\apps\cc51_5\bin\mpp51 cstart.asm
if errorlevel 1 goto error
c:\apps\cc51_5\bin\asm51 cstart noprint
if errorlevel 1 goto error

c:\apps\cc51_5\bin\cc51 -C552 -Ic:\apps\cc51_5\include -Mr -t -err -s ros552.c
if errorlevel 1 goto error
c:\apps\cc51_5\bin\asm51 ros552 noprint
if errorlevel 1 goto error

c:\apps\cc51_5\bin\cc51 -C552 -Ic:\apps\cc51_5\include -Mr -t -err -s beverage.c
if errorlevel 1 goto error
c:\apps\cc51_5\bin\asm51 beverage noprint
if errorlevel 1 goto error

c:\apps\cc51_5\bin\cc51 -C552 -Ic:\apps\cc51_5\include -Mr -t -err -s clib.c
if errorlevel 1 goto error
c:\apps\cc51_5\bin\asm51 clib noprint
if errorlevel 1 goto error

c:\apps\cc51_5\bin\cc51 -C552 -Ic:\apps\cc51_5\include -Mr -t -err -s cpers.c
if errorlevel 1 goto error
c:\apps\cc51_5\bin\asm51 cpers noprint
if errorlevel 1 goto error

c:\apps\cc51_5\bin\link51 @ros552.lnk
if errorlevel 1 goto error
c:\apps\cc51_5\bin\ihex51 ros552.out -o ros552.hex
if errorlevel 1 goto error
:error
```

linker options used in ros552.bat,
@ros552.lnk:

```
cstart.obj,
ros552.obj,
beverage.obj,
clib.obj,
cpers.obj,
c:\apps\cc51_4\lib\c51r.lib
TO
ros552.out
RAMSIZE(100H)
MA
DS
DL
SB
PL
DP
```

generated header-file, beverage.h:

```
/* Realizer Gold (V4.00) : generated ANSI-C Code */
/* File           : C:\Realizer\Examples\ROS552\BEVERAGE.h */
/* Scheme Version : 1.13 */
/* Date           : Thu Mar 23 12:34:01 2000 */
/* Used variables : 16  */
/* Used functions : 40  */


#define PROJECTNAME "BEVERAGE"
#define PROJECTVERSION "1.13"

extern xdat unsigned char realBit[];
extern xdat unsigned char realByte[];
extern xdat unsigned char realNonVolByte[];
#define v0n15 0
#define v0n17 0
#define pv0n17 1
#define v0n18 2
#define v0n26 3
#define v0n7 4
#define v0n16 5
#define v0n9 6
#define v0n8 7
#define v0n14 8
#define v0n10 9
#define v0n1 10
#define v0n2 11
#define v0n5 12
#define v0n6 13
#define v0n3 14
#define T00033 1
#define st0 2
```

generated c-file, beverage.c:

```
/* Realizer Gold (V4.00) : generated ANSI-C Code */
/* File           : C:\Realizer\Examples\ROS552\BEVERAGE.c */
/* Scheme Version : 1.13 */
/* Date           : Thu Mar 23 12:34:01 2000 */
/* Used variables : 16  */
/* Used functions : 40  */


#include "BEVERAGE.h"

#include "clib.h"
#include "cpers.h"
#include "cslp.h"

xdat unsigned char realBit[15];
xdat unsigned char realByte[3];
xdat unsigned char realNonVolByte[1];

void realInit(void)
{
  short i;
  for (i=0;i<15;i++)
    realBit[i] = 0;

  for (i=0;i<3;i++)
    realByte[i] = 0;

}
```

beverage.c, continued:

```
void realMain(short RTICK)
{
  short i;

    /* Decrement 8 bit timers */
  for (i=0;i<1;i++)
  {
    if ( (convertByteValue(1+i,TypeUByte) & 0x80) == 0x00)
      setByteValue(1+i,TypeUByte,convertByteValue(1+i,TypeUByte)-RTICK);
  }
  adc(v0n15,2,"P5_0 PourTime",2);
  digin(v0n7,1,"P1_1 SugarInput",1);
  digin(v0n16,1,"P1_2 MilkInput",1);
  digin(v0n3,1,"P1_0 CoinInput",1);
  stateout(v0n17,1,st0,2,4);
  timv(v0n17,1,pv0n17,1,v0n15,2,v0n18,1,T00033,2);
  inv(v0n18,1,v0n26,1);
  inv(v0n16,1,v0n9,1);
  inv(v0n7,1,v0n8,1);
  and2(v0n8,1,v0n9,1,v0n14,1);
  and2(v0n8,1,v0n16,1,v0n10,1);
  stateout(v0n1,1,st0,2,3);
  digout(v0n1,1,"P1_6 MilkOutput",1);
  stateout(v0n2,1,st0,2,4);
  digout(v0n2,1,"P1_7 PourOutput",1);
  stateout(v0n5,1,st0,2,1);
  digout(v0n5,1,"P1_4 CupOutput",1);
  stateout(v0n6,1,st0,2,2);
  digout(v0n6,1,"P1_5 SugarOutput",1);
  stateminit(st0,2,-1);
  stateinit(st0,2,0);
  statecd(st0,2,0,v0n3,1,1);
  stateend(st0,2,0);
  stateinit(st0,2,1);
  statecd(st0,2,1,v0n7,1,2);
  statecd(st0,2,1,v0n14,1,4);
  statecd(st0,2,1,v0n10,1,3);
  stateend(st0,2,1);
  stateinit(st0,2,2);
  statecd(st0,2,2,v0n16,1,3);
  statecd(st0,2,2,v0n9,1,4);
  stateend(st0,2,2);
  stateinit(st0,2,3);
  state(st0,2,3,4);
  stateend(st0,2,3);
  stateinit(st0,2,4);
  statecd(st0,2,4,v0n26,1,0);
  stateend(st0,2,4);
  statemend(st0,2,5);
  copy(v0n17,1,pv0n17,1);
}
```