Application note on REALIZER

Title:	How the generated code looks like
Date:	27 October 1998
Our ref.:	0332600
Subject:	Version 3.20
Author:	René Balvers

Introduction

This document informs users of Realizer how the program code that is generated by Realizer is built up. It describes the various properties of the parameters and algorithms that Realizer uses to generate the code.

Scope

This document assumes that the reader is either a user of Realizer or is interested in the working principles or Realizer. A basic knowledge about microcontrollers and the way they are programmed is mandatory. The reader is supposed to have elementary experience in programming in assembly.

All example schemes are for illustration purposes only. The resulting code shown is literally as generated by Realizer for the related schemes.

For reasons of convenience, all the code has been generated for STM's ST62 series of microcontroller. Please note that Realizer can generate code for more than just this controller architecture. For more information, please visit our web site: www.actum.com.

Traditional programming (hand-crafted)

If you would program an application by hand in assembly you would start with a main program that will check various conditions and upon certain conditions execute specific routines to perform specific tasks.

However, in embedded control applications "multitasking" is always *built in*. This is opposite to the way personal computer applications are programmed. There is a lot of sequential structure in a PC-application: First read the file, then search for the specific string, then change its value, then write back the data to the disk.

In embedded applications there are very often a lot of things to do at the same time. E.g.: compare values to switch an output on and check for time-outs on other variables as well ("at the same time"). This is essentially what makes embedded systems so hard to program.

Standard Realizer generated code

Realizer technology is based on the demands that have been identified in embedded applications. Lets assume we have the following application:

Inputs are: A, B, and C.
Outputs are: Go and Fault.

Go should be on when either C is on or when A and B are both on. Fault is the output signalling that C and B are both on too long (> 15 seconds).

The Schematic in Realizer would look like the one shown in figure 1.The code generated by Realizer is shown in table 1.

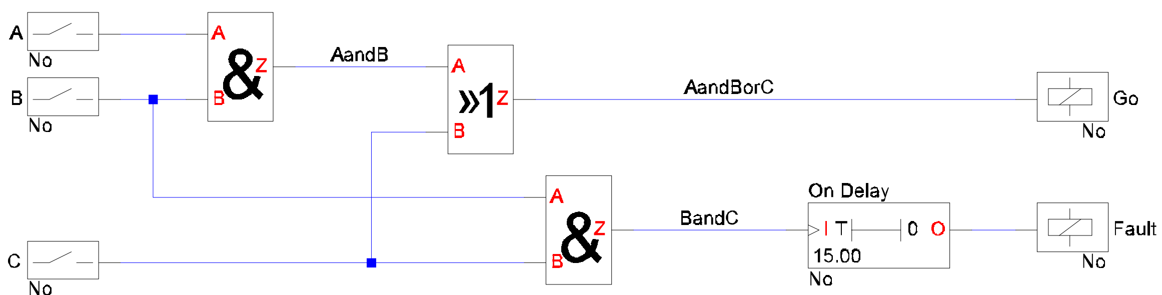

Figure 1 - Example scheme

Code	Remarks

Reset:
  CLR IOR
  LDI HWDR,0FFH
	Hardware reset entry point

BUDRA .DEF 089H,0FFH,0FFH BUDRB .DEF 08aH,0FFH,0FFH	

PortInit:
  LDI DDRA,004H
  LDI ORA,000H
  LDI DRA,000H
  LDI BUDRA,000H
  LDI DDRB,040H
  LDI ORB,040H
  LDI DRB,002H
  LDI BUDRB,002H
	Initialization of the devices I/O

AdcInit:	Analog inputs

Rtcinit:
TICK .DEF 08BH,0FFH,0FFH
RTICK .DEF 08CH,0FFH,0FFH
ARTICK .EQU 08CH
  LDI PSC,015H
  LDI TCR,0CFH
  LDI TSCR,06DH
  CLR TICK
  LDI IOR,010H
  RETI
	Initialization of time keeping device

RamInit:
  LDI A,0
  LDI X,084H
  LDI Y,3
RamInit1:
  LD (X),A
  INC X
  DEC Y
  JRNZ RamInit1
	Initialization of data memory

v0n3 .DEF 084H,001H,001H v0n1 .DEF 084H,002H,002H v0n0 .DEF 084H,004H,004H v0n9 .DEF 084H,008H,008H pv0n9 .DEF 084H,010H,010H v0n8 .DEF 084H,020H,020H v0n2 .DEF 084H,040H,040H v0n4 .DEF 084H,080H,080H T00009 .DEF 085H,0FFH,0FFH	Variables declaration
RealMain:	Main entry point

Rtc:
  CLR IOR
  LD A,TICK
  CLR TICK
  LDI IOR,010H
  LD RTICK,A
	Keep track of time elapsed since previous loop started

diginb v0n3,0,1,DRB,3,1
  diginb v0n1,1,1,DRA,1,1
  diginb v0n0,2,1,DRB,1,1
RINPEND:
	Reading Bit inputs

and2bbb v0n1,1,1,v0n3,0,1,v0n9,3,1	AND (B, C)
delfonbbb v0n9,3,1,pv0n9,4,1,v0n8,5,1,1500,100,T 00009,4,RTICK,2	On Delay timer
digoutb v0n8,5,1,BUDRA,2,1	Write to output
and2bbb v0n0,2,1,v0n1,1,1,v0n2,6,1	AND (A, B)
or2bbb v0n2,6,1,v0n3,0,1,v0n4,7,1	OR (AandB, C)
digoutb v0n4,7,1,BUDRB,6,1 ROUTPEND:	Write Output
copybb v0n9,3,1,pv0n9,4,1	Save previous value of timer input.

LD A,BUDRA
  LD DRA,A
  LD A,BUDRB
  LD DRB,A
	extra instructions (See ST62xx datasheet)

LDI HWDR,0FFH	Trigger watchdog
JP RealMain	Start over again

Table 1 - Code generated for scheme of figure 1

Please note the following about the cod-listing:

1. The instructions listed are very often calls to macro's supplied with Realizer. Each macro is optimally implemented and extensively tested for the particular processor architecture by experienced assembly programmers: they already did the job for you.
2. The calculations of functions between the reading of the input and the completion of write operation of the outputs is encapsulated by the labels RINPEND[1] and ROUTPEND[2]. The location and meaning of these labels are important orientation points during the next discussions involving code generated by Realizer.
3. Behind the label AdcInit, there is no code. Since this application does not use any of such inputs, Realizer doesn't generate any code for that function. So optimal code is generated, with very little overhead.

In the code listing we can identify the following parts:

- Chip initialization
- I/O initialization
- Timer initialization
- Data memory (application) initialization
- Entrance of main loop
- Keep track of elapsed time
- Reading input data
- Calculation of output data
- Writing of output data
- Backup copies of variables (to detect edges)
- Update state machines
- Start over at main loop

Figure 2 shows the structure of the code in a graphical manner.

Realizer can apply various kinds of optimizations while analysing the design and generating the code. Among them are:

1. Variable optimization and
2. Sleeping code optimizations.

The next sections of this document describes each of them.

Future developments will lead to enhancements to the algorithms as well as new optimizations.

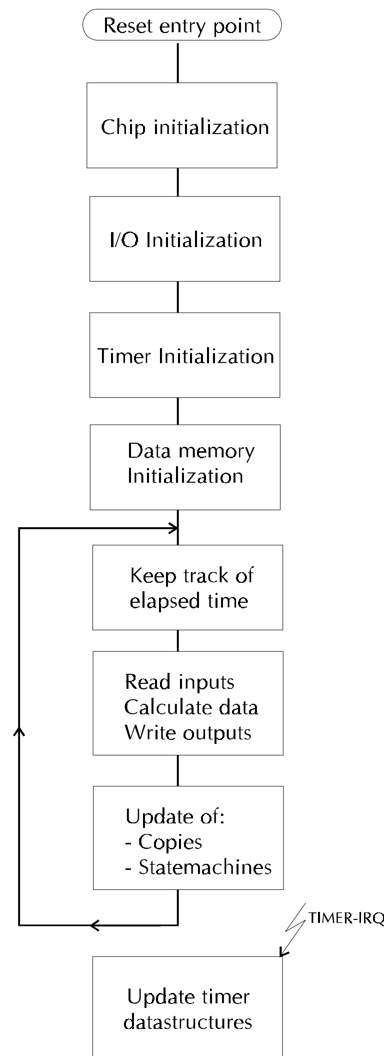

Figure 2 - Structure of Realizer generated code

[1] Realizer INPut reading ENDed
[2] Realizer OUTPut writing ENDed

© Actum Solutions 0332600

Variable optimization

This decreases the number of RAM variables that are needed. Normally each *net* (a collection of wires carrying the same signal) is stored in a separate memory location in RAM. Variable optimization decreases this number of memory locations by allocating more than one net to the same memory location.

The result is that larger applications can be implemented on a certain microcontroller.

In figure 1, the memory allocated for the net between input A and the And-gate can be re-used for storing the result of the And-operation. This can be cascaded throughout a scheme, saving a lot of valuable data space. In theory, this example can be calculated using 3 bit variables (excluding the overhead of input buffering and internal variables).

© Actum Solutions 0332600

Sleeping code optimization

This optimisation makes the application execute faster (and thus saving processor time).

The analyser checks for specific symbols like AND-gates, multiplexers and many more. When the analyser is confident that at a certain moment during execution, the resulting value of a branch of symbols is of no interest to the final result, it will skip the code execution of that branch.

In the example of figure 1 this would mean that the determination of the fact that input C is on, the branch connected to the other input of the Or-gate will not be calculate. An *if-then-else* construction is put around this part of the scheme. In a pseudo programming language:

if Input C = 1

then Go = 1

else Go = A & B

Before applying this optimization, the analyser will measure the size of each branch and generate the code for the shortest one first. Then it will insert the code for the *if-then* and then the code for the other branch.

Besides the information mentioned, it also (but not only) checks for timers and other symbols that have some kind of automatic, internal or "static data". This limits the extent of the optimization.

Note that the optimization is done at compile time, with the effect of saving execution time during the usage of the code.

The result of the sleeping code optimizations cannot easily be quantified, since it greatly depends on the schematic diagram. Tables 2 and 3 show the code generated of the example in figure 1.

Note that for the example of figure 1 the gain is not obvious.

Code	Remarks
RINPEND:	Normal code
and2bbb v0n1,1,1,v0n3,0,1,v0n9,3,1	And (B, C)
delfonbbb v0n9,3,1,pv0n9,4,1,v0n8,5,1,1500,100,T00009,4,RTICK,2	Calculate timer
digoutb v0n8,5,1,BUDRA,2,1	Output(Fault)
and2bbb v0n0,2,1,v0n1,1,1,v0n2,6,1	And (A, B)
or2bbb v0n2,6,1,v0n3,0,1,v0n4,7,1	Or (AandB, C)
digoutb v0n4,7,1,BUDRB,6,1	Output (Go)
ROUTPEND:	

Table 3 - Example Scheme <u>without</u> Sleeping code optimization

Code	Remarks
RINPEND:	
and2bbb v0n1,1,1,v0n3,0,1,v0n9,3,1	And (B, C)
delfonbbb v0n9,3,1,pv0n9,4,1,v0n8,5,1,1500,100,T00009,4,RTICK,2	Calculate timer
digoutb v0n8,5,1,BUDRA,2,1	Output (Fault)
cifnelsb v0n3,0,1,0	If not (C=1)
cifelsb v0n1,1,1,1	If B=1
copybb v0n0,2,1,v0n2,6,1	AandB = A
celse 1	Else
copybb v0n1,1,1,v0n2,6,1	AandB = B
cendif 1	Endif
copybb v0n2,6,1,v0n4,7,1	AandBorC = AandB
celse 0	Else
copybb v0n3,0,1,v0n4,7,1	AandBorC = C
cendif 0	Endif
digoutb v0n4,7,1,BUDRB,6,1	Write output
ROUTPEND:	

Table 2 - Example Scheme <u>with</u> sleeping code optimization

The scheme of figure 3 however, shows a quite different application, where the effect is astonishing.

Table 6 shows the effects of the optimization in terms of code size data size and the loop-time.

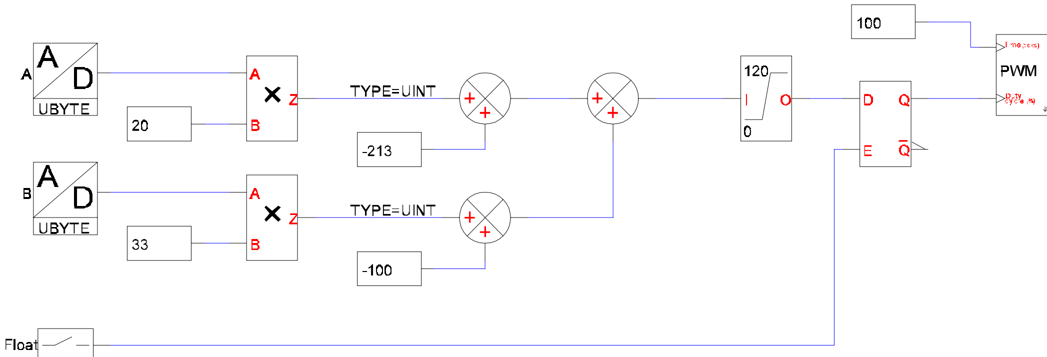

Figure 3 - Example for Sleeping code optimization

Code	Remarks

RINPEND:
  mulwww v0n11,2,v0n12,2,v0n16,4
  add2www v0n16,4,v0n18,3,v0n19,4
  mulwww v0n9,2,v0n10,2,v0n5,4
  add2www v0n5,4,v0n17,5,v0n6,5
  add2www v0n6,5,v0n19,4,v0n23,5
  limfww v0n23,5,v0n25,5,0,120
  dltchwbww v0n25,5,v0n0,0,1,v0n3,5,0,0
  pwmwwww v0n4,2,pv0n4,2,v0n3,5,pv0n3,5
 	
ROUTPEND:
	
	Scale first input
	
	Scale second one
	
	Add them together
	Apply limits
	Calculate D-Latch
	Write output

Table 4 - Example of figure 3 <u>without</u> Sleeping

Code	Remarks

RINPEND:
  cifb v0n0,0,1,0
  mulwww v0n11,2,v0n12,2,v0n16,4
  add2www v0n16,4,v0n18,3,v0n19,4
  mulwww v0n9,2,v0n10,2,v0n5,4
  add2www v0n5,4,v0n17,5,v0n6,5
  add2www v0n6,5,v0n19,4,v0n23,5
  limfww v0n23,5,v0n25,5,0,120
  dltchwbww v0n25,5,v0n0,0,1,v0n3,5,0,0
  cendif 0
  pwmwwww v0n4,2,pv0n4,2,v0n3,5,pv0n3,5
 	
ROUTPEND:
	
	if Float then
	   Scale first input
	
	   Scale second one
	
	   Add them together
	   Apply limits
	   Calculate D-Latch
	endif
	Update output

Table 6 - Example of figure 3 <u>with</u> sleeping code optimization code optimization

If the enable input of the D-type latch is not active (0), the value of the D-input is irrelevant.

From the tables 4 and 5, which show parts of the generated code of figure 3, we can see that during the low-state of the Float input, the only statements executed are the *if*-construction and the update of the PWM-output. Its obvious that the execution speed is strongly increased.

Loop-time and memory sizes	Sleeping Code Optimization	
Parameter	Without	With
Best case (μS)	2055	<u>**240**</u>
Worst case (μS)	3140	3146
Code (bytes)	612	617
Data (bytes)	33	33

Table 5 - Comparison of results of example of figure 3

Conclusion

You have seen the effect of various optimizations. They are all implemented in Realizer V3.20. With these optimizations, larger designs can be implemented on the samen microcontroller, or increase the performance of existing designs.

It is important to realize that the optimizations never interfere with the functionality of the design. So you can experiment freely to determine the right combination to trade off the various gains using the optimizations, either for code size, data memory size, or speed.

For more information please visit our web site: www.actum.com, send an e-mail to info@actum.com or fax us at: +31 (0)72 576 2555.