

Renesas R8C/13 mit mt-Multitasking

Dieser Beitrag ist so etwas wie eine Fortsetzung des Artikels *Mikrocontroller R8C/13 und Renesas HEW mit Emulator E8*. Zum Verständnis ist es unverzichtbar, auch den Teil 3 von *Modulares Mikrocontrollersystem mit AVR ATmega32* zu kennen. Während dort die Demonstration des mt-Multitasking anhand des Programms `mtdemo/main.c` im Mittelpunkt steht, geht es hier um die Übertragung (Portierung) dieses Programms auf den HEW von Renesas für den MC R8C/13. Es heißt in dieser Umgebung `mtdemo.c` und steht als Download zur Verfügung.

Relevante Unterschiede

Wenn beim HEW das Projekt `mtdemo` erzeugt wird, heißt das Hauptprogramm `mtdemo.c`. Es entsteht im Projektverzeichnis als Muster, das durch `mtdemo.c` aus dem Download zu ersetzen ist. Dieses Programm mit den dazu gehörenden Quellen, die ebenfalls in das Projekt zu kopieren sind, ist die portierte Variante von `mtdemo/main.c`.

Der R8C/13 hat im Gegensatz zum ATmega32 16 KB ROM und ist mit 20 MHz getaktet. Außerdem werden konstante Daten, die mit dem Qualifizierer `const` deklariert sind, vom Compiler ins ROM geschickt. Beim ATmega32 ist dieser einfache Weg nicht möglich, was an seiner Architektur liegt.

Die Portierung betrifft fast nur Hardwarezugriffe. An einer Stelle ohne Bezug zur Hardware (siehe Taskfunktion `play4`) reagieren die Compiler jedoch unterschiedlich, so dass ein Eingriff in die Software nicht zu vermeiden ist.

Die Bibliotheksfunktion `sqrt()` zur Berechnung der Quadratwurzel verbraucht beim R8C/13 sehr viel mehr Speicherplatz und Laufzeit als beim ATmega32, so dass sie praktisch nicht verwendet werden kann.

Bei Renesas sind zur Verwendung in C nicht nur die Registernamen definiert, sondern auch die meisten Bitnamen. Um z. B. das mit `xyz` definierte Bit zu setzen, genügt die Programmzeile `xyz = 1;`

Um das Makefile kümmert sich der HEW von Renesas und nicht der Programmierer. Logisch zufügen muss man die Quellen dem Projekt aber schon, und zwar mit `Project/Add Files`.

Konstanter Text, z.B. "Text", entspricht beim AVR-Compiler einem `char` Zeiger, der nur Zeigervariablen mit exakt diesem Typ zugewiesen werden kann. Bei `unsigned char` Zeigern wird gewarnt. Diese Strenge zeigt der Renesas Compiler nicht. Bei den Quelltexten `reac.c` und `play.c` spielt das eine Rolle.

Hardware

Ein zum LCD/Tastatur Modul aus AVR Teil 3 passendes R8C13-Modul gibt es zwar, wie auf dem Foto zu sehen ist. Aber auch die fliegende Verdrahtung ist nicht schwer, weil der R8C/13 auf seinem 32 Pin Carrier zum Einsatz kommt. Das Modul versieht den Carrier mit einer Reset-Schaltung, die nicht unbedingt nötig ist, und die LED kann ebenfalls als Provisorium angeschlossen werden.

Die richtige Verdrahtung und die sinnvolle Verwendung der Port-Leitungen wird durch den Signalplan auf der nächsten Seite sehr erleichtert.

Timer Y des R8C13 erzeugt die von mt benötigten Interrupts im Abstand von 1 ms, und Timer X erzeugt die Töne für die Musik.



Signalplan R8C

Projekt

R8C-Modul					Angeschlossene Module		
Pin Nr	Altern. Signal	Port Bit	K7 PinNr	IN/ OUT	Modul	K1 PinNr	Funktion / Zweck / Klemme
1	TxD10/RxD1	P3_7					
2	CNVss						
3	<u>Reset</u>						
4	Xout (1)	P4_7					
5	Vss (GND)						
6	Xin	P4_6					
7	Vcc (+5V)						
8	INT1/CNTR0	P1_7	12		Speaker		macht Musik, Ausgang Timer X
9	CLK0	P1_6	13				
10	RxD0	P1_5	14				
11	TxD0	P1_4	15				
12	KI3/AN11	P1_3	16				
13	KI2/AN10/CMPO2	P1_2	17				
14	KI1/AN9/CMPO1	P1_1	18				
15	KI0/AN8/CMPO0	P1_0	19				
16	<u>INT0</u>	P4_5	20	O	LED	-	Lebenszeichen
17	INT3/TCin	P3_3	11				
18	<u>INT2/CNTR1/CMP12</u>	P3_2	10				
19	Avcc/Vref						
20	Tzout/CMP11	P3_1	9				
21	AVss						
22	<u>CNTR0/CMP10</u>	P3_0	8				
23	IVcc (3)						
24	AN0	P0_7	7	O	Tast/LCD	9	LCD-D7 Pin 14
25	AN1	P0_6	6	O	Tast/LCD	8	LCD-D6 Pin 13
26	AN2	P0_5	5	O	Tast/LCD	7	LCD-D5 Pin 12
27	AN3	P0_4	4	O	Tast/LCD	6	LCD-D4 Pin 11
28	MODE						
29	AN4	P0_3	3	O	Tast/LCD	5	LCD-E Pin 6
30	AN5	P0_2	2	O	Tast/LCD	4	LCD-RS Pin 4
31	AN6	P0_1	1	I	Tast/LCD	3	Keyboard
32	AN7/TxD11	P0_0					

Das Programm mtdemo.c

Nur die Unterschiede zu mtdemo/main.c werden beschrieben und als Programmcode dargestellt.

Hauptprogramm

Einrichtung des LED-Ausgangs

Mit drr6 und drr7 wird der Ausgang als LED Treiber konfiguriert, das heißt er wird höher belastbar.

```
//LED
#define LEDBIT      5
#define LEDPORT     p4

#define LEDBITINIT  pd4 |= 0x01<<LEDBIT ;\
                    drr6 = 1;\
                    drr7 = 1;

#define LEDON       LEDPORT |= 0x01<<LEDBIT ;
#define LEDOFF      LEDPORT &= ~(0x01<<LEDBIT) ;
```

Die Interrupt Service Routine

Sie wird nur aufgerufen, wenn auch die Datei sect30.inc dafür eingerichtet wurde (siehe unten).

```
//Interrupt Service Routine
#pragma interrupt timer_y
void timer_y (void){
    isrmt ();
}
```

Auszug aus der Vektorliste von Datei sect30.inc

Die Zeile `.if 0` muss in `.if 1` geändert werden, was nicht vergessen werden darf.

```
.if 1
..
..
.lword  dummy_int      ; vector 21
.lword  dummy_int      ; vector 22

.glb    _timer_y       ; vector 23
.lword  _timer_y       ; vector 23

.lword  dummy_int      ; vector 24
.lword  dummy_int      ; vector 25
..
..
```

main function

Im ersten Teil der main function wird vom internen Oszillator auf den externen Quarz-Oszillator gewechselt. Weil dieser Eingriff durch das Bit prc0 geschützt ist, muss der Schutz vorübergehend aufgehoben werden.

Dann wird Timer Y so eingerichtet, dass sein Timer Register im Takt von 250 kHz herunter gezählt wird, um dann einen Interrupt auszulösen und wieder von vorn.

```
//main function

int main(void) {
    asm( "FCLR I");    // Change on-chip oscillator clock to Main clock
    prc0 = 1;         // Protect off
    cm13 = 1;         // Xin Xout
```

```

cm15 = 1;          // XCIN-XCOUT drive capacity select bit : HIGH
cm05 = 0;          // Xin on
cm16 = 0;          // Main clock = No division mode
cm17 = 0;
cm06 = 0;          // CM16 and CM17 enable
asm("nop"); asm("nop"); asm("nop"); asm("nop");
ocd2 = 0;          // Main clock change
prc0 = 0;          // Protect on

//////////Init Timer Y 1kHz, CLOCK 20 MHz//////////
tymod0 = 0;        //Timer mode for Timer Y
tysc = 0;          //250 kHz / 250 = 1 kHz
typr = 250-1;
prey = 10-1;      //2,5 MHz / 10 = 250 kHz
tyck0 = 1;        //20 MHz / 8 = 2,5 MHz
tyck1 = 0;

tyic = 7;          // Interrupt-Level
ir_tyic = 0;

tyw = 0;
tys = 1;          // Timer on

////////// mt //////////////////////////////////////
LEDBITINIT
initmt ();
STARTMT           //Tasks starten
schedule ();      //läuft endlos
}

```

Datei mtsys.h

Die Definitionen für Assembler Befehle sind für den R8C zu aktivieren und für AVR zu einem Kommentar zu machen.

```

/* R8C */
#define DI asm ("\tFCLR I"); //disable interrupt
#define EI asm ("\tFSET I"); //enable interrupt
#define NOP asm("nop"); //No Operation

/* AVR
#define DI asm volatile ("cli"); //disable Interrupt
#define EI asm volatile ("sei"); //enable Interrupt
#define NOP asm volatile ("nop"); //No Operation
*/

```

Datei lcdsimple.h

Mit den Symbolen CLOCK und ADJUST wird die Zeiteinheit für die Verzögerung von Funktion delay_ms auf 1 ms justiert. Der R8/C kann etwa 20000 NOPs in einer Millisekunde ausführen und der ATmega 32 nur etwa 5000.

```

#define CLOCK 2000000 //delay_ms() Taktfrequenz des MC
#define ADJUST 20000 //delay_ms() Justierung

#define NIBBIT 4 //<= 4, LCDPORT Bit, bei dem Bit 0 des Nibbles steht
#define EBIT 3
#define RSBIT 2

#define INITPORTS \
prc2 = 1;\
pd0 |= (15<<NIBBIT);\
prc2 = 1;\
pd0 |= (1<<RSBIT) ;\
prc2 = 1;\
pd0 |= (1<<EBIT);
// (prc2 =1; Protect für Port p0 wegnehmen, wird autom. wieder hergestellt )

```

```
#define LCDPORT    p0
#define EPORT      LCDPORT
#define RSPORT     LCDPORT
```

Die mehrzeilige Definition INITPORTS ist Programmtext, der 6 Bits des Register p0 zu Outputs macht, denn dort ist das LCD angeschlossen. Bei jedem Zugriff auf pd0 muss der Schutz mit prc2 = 1 aufgehoben werden, weil er jedes mal automatisch wieder eingerichtet wird. Achtung: Bei den mehrzeiligen Definitionen darf in einer Zeile nach \ kein weiteres Zeichen, insbesondere kein Leerzeichen, folgen.

Datei reac.h

Das Bit 1 von Port p0 ist der Eingang für die analoge Tastatur. Nur die Auf-Taste ist für den Reaktionstester verwendbar, da sie als einzige auch digital abgefragt werden kann.

```
#define REACPORT   (unsigned char *) &p0    //Zeiger auf Port
#define REACBIT    1
```

Datei play.h

Die Definition TIMERINIT ist mehrzeilig und enthält Programmtext, der den Timer X in den Pulse Output Mode versetzt. Die Pulse entstehen an Bit 7 von Port 1, das ist Pin 8 des R8/C. Das Register prex bewirkt eine Frequenz von 125 kHz, mit der das Timer Register tx abwärts gezählt wird.

Auch die Definition SETTIMER enthält Programmcode, der das Timer Register entsprechend der jeweiligen Note setzt.

```
#define TIMERINIT \
    txmr = 0x01;\
    txck0 = 0; txck1 = 0;\
    prex = 160-1;
    // Pulse Output mode for Timer X , Frequenz an P1_7 Pin 8
    // Timer Count source = f1 = 1
    // Set Prescaler X register 20 MHz / f1 / 128 = 125 kHz

#define SETTIMER tx = SD [note] [TONE];
    // Set Timer X register z.B. 142 ergibt: 125 kHz / 142 = 880 Hz = Ton A 440 Hz

#define TIMERON txs = 1; //Timer X count start flag

#define TIMEROFF txs = 0;
```

Datei play.c

Die folgende Änderung ist die einzige, die wegen unterschiedlicher Arbeitsweisen der Compiler zwingend notwendig ist.

```
void play4 (void) {
    ..
    ..
    mtdelay ( (unsigned long) SD[note++] [DURA] * SD[0][0] ,play5 );
    ..
}
```

In Taskfunktion play4 werden zur Berechnung der Zeit für mtdelay() zwei Bytes multipliziert. Der AVR Compiler gibt dem Produkt einen größeren Typ, so dass das Ergebnis richtig dargestellt wird. Der Renesas Compiler macht das nicht. Er speichert das Ergebnis in einem Byte, wodurch es falsch wird, und das ohne zu warnen. Mit einer Typwandlung wird für den passenden Typ gesorgt.

Datei prim.c

Die Funktion sqrt aus der Bibliothek math.h (Quadratwurzel) unter Renesas vergrößert das Programm auf etwa 12400 Byte gegenüber etwa 7000 Byte, wenn als Ersatz eine Zahl benutzt wird. Die Quadratwurzel ist nicht unbedingt erforderlich. Sie minimiert die Zahl der nötigen Divisionen. Eine Zahl tut es auch, wenn sie größer als die Quadratwurzel aus MAXNUM ist, besser deutlich größer, weil bis zur nächsten Primzahl ab MAXNUM gerechnet wird.

Die Rechenzeit für den vorgegebenen Bereich beträgt mit `maxquot=sqrt(num)`; ca. 130 s und mit der Zahl 1100 ca. 70 s. `sqrt()` optimiert zwar den Algorithmus, ist aber wegen der langen Ausführungszeit kontraproduktiv.

Deshalb ist idle gleich 0, wenn `sqrt()` beibehalten wird. Die Ausführungszeit von `sqrt()` ist also größer 1 ms. Das sollte möglichst vermieden werden, weil es unkontrollierbare Verlängerungen von Reaktionszeiten geben könnte. Außerdem sind über 5 KB Programcode für diese Funktion bei 16 KB insgesamt völlig unangemessen. Deshalb wird `sqrt()` durch die Zahl 1100 ersetzt. Die Zahl idle nimmt dann etwa den Wert 28 an.

```
void prim2 (void) { //Init next prime test
    num++;
    quot= 2;
    maxquot = 1100; //sqrt(num);
    mtcoop (prim3);
}
```

Beim AVR Compiler ist von all dem kaum etwas zu merken. Weder ist idle so stark unterschiedlich, noch wächst die Programmlänge über Gebühr.

Allerdings ist offen, weshalb idle bei AVR ohne `sqrt()` etwa 10 beträgt und bei Renesas etwa 28, wo doch ohne Berechnung der Primzahlen bei beiden Umgebungen idle etwa 500 ist.