Übungen zu Organisation und Architektur von Rechnern
SS 2002
Serie 6 — Musterlösung

Aufgabe 16 (4 Punkte)

Der DUART-Dokumentation läßt sich entnehmen, daß die Initialisierung aus folgenden Schreiboperationen besteht:

a) `%00010000` an die Adresse `$effc01+4` (CRA) schreiben.

b) `%00000011` an die Adresse `$effc01+0` (MR1A) schreiben.

c) `%00000000` an die Adresse `$effc01+0` (MR2A) schreiben.

d) `%10111011` an die Adresse `$effc01+2` (CSRA) schreiben.

e) `%00000100` an die Adresse `$effc01+4` (CRA) schreiben.

Um diese `MOVE`-Befehle elegant hinschreiben zu können, bietet es sich an, mit Hilfe von `EQU`-Anweisungen Bezeichner für die Basisadresse des DUART (`$effc01`) und die erforderlichen Offsets zu definieren.

Aufgabe 17 (3 Punkte)

Die Funktion `PUT_CHAR` erwartet als Argument einen Byte-Wert auf dem Stack. Bei der Übergabe über den Stack ist jedoch zu berücksichtigen, daß beim MC68k der `SP` immer auf Wort-Adressen zeigen muß. Angenommen das Argument für `PUT_CHAR` wird im Byte-Format auf den Stack geschoben:

```
MOVE.B  #$21,-(SP)
JSR     PUT_CHAR                                        .
```

Die `MOVE`-Anweisung dekrementiert `SP` dann nicht nur um 1, sondern um 2, damit wieder eine legale Adresse herauskommt. Entsprechend muß am Ende von `PUT_CHAR` die Rücksprungadresse nicht nur um ein Byte, sondern um ein Wort verschoben werden und der `SP` entsprechend hochgesetzt werden:

```
MOVE.L  (SP),2(SP)      kill argument frame: move return address
ADDQ.L  #2,SP                                adjust SP         .
        RTS
```

Die Musterlösung umgeht dieses Problem, indem alle Argumente stets als Long-Werte auf den Stack geschoben werden. Innerhalb der Funktion werden von einem solchen Long-Wert bei Bedarf nur das niederwertige Byte oder Wort berücksichtigt.

Aufgabe 18 (3 Punkte)

Die Funktion `PUT_STR` erhält als Parameter die Startadresse einer Byte-Sequenz. Die an dieser und nachfolgender Adressen vorgefundenen Bytes werden mit Hilfe der Funktion `PUT_CHAR` solange ausgeben, bis an einer Adresse der Byte-Wert `$00` auftritt. Es ist darauf zu achten, daß dieses Endsymbol selbst *nicht* ausgegeben wird!

Aufgabe 19 (10 Punkte)

Die Funktion `HTOA` wurde so konzipiert, daß sie drei Argumente benötigt: Die zu konvertierende Hex-Zahl, sowie die niedrigste und die höchste Adresse des Speicherbereichs, in dem die zu generierende Zeichenkette abgelegt werden soll. Alternativ wäre es möglich, auf die Übergabe einer der beiden Adressen zu verzichten. Die Vorgehensweise der Musterlösung hat jedoch zwei Vorteile: Zum einen kann `HTOA` feststellen, ob der Speicherbereich eine ausreichende Größe hat. Zum anderen kann bezüglich des aufrufenden Kontextes offen bleiben, ob `HTOA` den Speicherbereich von vorne oder von hinten aus beschreibt.

Mit Hilfe der folgenden Assembler-Anweisungen ist es möglich auf elegante Art und Weise ein Byte-Feld der Größe 100 anzulegen, wobei die Label `BUFFER_F` und `BUFFER_L` die Adressen des ersten bzw. des letzten Bytes bezeichnen:

```
BUFFER_F        DS.B    99
BUFFER_L        DS.B    1                                               .
```

Achtung: Falls im Anschluß an diese Zeilen noch Instruktionen stehen, muß darauf geachtet werden, daß der reservierte Speicherbereich eine gerade Anzahl von Bytes umfaßt! Andernfalls würde beim Assemblieren der `LC` (location counter) für die nachfolgende Instruktion u. U. nicht mehr auf eine Wort-Adresse zeigen, und bei der Ausführung des Maschinencodes käme es zu einem Busfehler (falsches Alignment).

Einziger Rückgabewert von `HTOA` ist die Startadresse der generierten Zeichenkette, es könnte also gleich im Anschluß die Funktion `PUT_STR` aufgerufen werden.

Beispiel: Die Funktion `HTOA` wird mit den beiden Adressen `$00001000` und `$0000100f` aufgerufen, der Speicherbereich umfaßt also 16 Bytes. Die gewählte Implementierung geht so vor, daß sie diesen Speicherbereich von hinten aus beschreibt. Die 10 Zeichen (`'$'`, acht Hex-Ziffern und `$00`) werden also auf den höchstwertigsten Adressen abgelegt und die Adresse `$00001006` zurückgeliefert.

Der **Assembler-Code** für die Aufgaben 16 – 19:

```
1                ORG     $0
2                DC.L    $8000           stack pointer value after a reset
3                DC.L    START           program counter value after a reset
4
5   START:       JSR     INIT_IO         initialize the DUART
6                BREAK
7                MOVE.L  #$1fa79ad9,-(SP)
8                JSR     PUT_HEX         call PUT_HEX
9                BREAK
10
11
12  ********************************************************************************
13  ********************************************************************************
14  *
15  * INPUT / OUTPUT stuff
16  *
17
18  DUART           EQU     $effc01
19  MR1A            EQU     0
20  MR2A            EQU     0
21  SRA             EQU     2
22  CSRA            EQU     2
23  CRA             EQU     4
24  TBA             EQU     6
25
26  BUFFER_F        DS.B    $ff             buffer containing $100 bytes with start
27  BUFFER_L        DS.B    1               ... and end address
28                  * caution: should be a multiple of word in order to avoid
29                  * alignment problems in subsequent statements!!!
30
31
32  ********************************************************************************
33  *
34  * function: void INIT_IO( )
35  *
36  * description:
37  *   Initializes the Dual UART Device
38  *
39  * registers affected: none
40  *
41
42  INIT_IO:        MOVEM.L A0,-(SP)                save registers on stack
43                  LEA     DUART,A0
44                  MOVE.B  #%00010000,CRA(A0)      reset MRxA pointer
45                  MOVE.B  #%00000011,MR1A(A0)     8 data bits
46                  MOVE.B  #%00000000,MR2A(A0)     normal mode
47                  MOVE.B  #%10111011,CSRA(A0)     set clock to 9600
48                  MOVE.B  #%00000100,CRA(A0)      enable Tx
49                  MOVEM.L (SP)+,A0                restore registers
50                  RTS
51
52
```

```
53  ********************************************************************************
54  *
55  * function: void PUT_CHAR( (SP))
56  *
57  * description:
58  *   Prints the character that is passed via the stack (lowest byte of the long)
59  *
60  * registers affected: none
61  *
62
63
64  PUT_CHAR:       LINK    A6,#0           store pointer to argument frame in A6
65                  MOVEM.L D0,-(SP)        save registers on stack
66                  MOVE.L  8(A6),D0        fetch character
67
68  PC_LOOP:        BTST    #2,SRA+DUART    check whether port is ready
69                  BEQ.S   PC_LOOP
70                  MOVE.B  D0,TBA+DUART    write data to output
71
72                  MOVEM.L (SP)+,D0        restore registers
73                  UNLK    A6              restore A6
74                  MOVE.L  (SP),4(SP)      kill argument frame: move return address
75                  ADDQ.L  #4,SP                               adjust SP
76                  RTS
77
78
79  ********************************************************************************
80  *
81  * function: void PUT_STR( (SP))
82  *
83  * description:
84  *   Prints the string (sequence of bytes) whose starting address is passed via
85  *   the stack (long)
86  *
87  * registers affected: none
88  *
89
90
91  PUT_STR:        LINK    A6,#0           store pointer to argument frame in A6
92                  MOVEM.L A0/D0,-(SP)     save registers on stack
93                  MOVE.L  8(A6),A0        fetch address of string
94
95  PS_LOOP:        MOVE.B  (A0)+,D0        fetch first character of string
96                  CMPI.B  #0,D0           is this the terminating 0?
97                  BEQ     PS_DONE
98                  MOVE.L  D0,-(SP)
99                  JSR     PUT_CHAR
100                 BRA     PS_LOOP
101
102 PS_DONE:        MOVEM.L (SP)+,A0/D0     restore registers
103                 UNLK    A6              restore A6
104                 MOVE.L  (SP),4(SP)      kill argument frame: move return address
105                 ADDQ.L  #4,SP                               adjust SP
106                 RTS
107
108
109 ********************************************************************************
110 *
111 * function: void PUT_HEX( (SP))
112 *
113 * description:
114 *   Prints the hex number that is passed via the stack (long)
115 *
116 * registers affected: none
117 *
118
119 PUT_HEX:        LINK    A6,#0           store pointer to argument frame in A6
120                 MOVEM.L D0,-(SP)        save registers on stack
121                 MOVE.L  8(A6),D0        fetch hex number
122
123                 MOVE.L  #BUFFER_F,-(SP) push pointer to start of buffer
124                 MOVE.L  #BUFFER_L,-(SP) push pointer to end of buffer
125                 MOVE.L  D0,-(SP)        push hex number
126                 JSR     HTOA            create ascii-rep in buffer and
```

```
127                                     * return ptr to result on stack
128                 JSR     PUT_STR         print buffer starting at returned ptr
129
130                 MOVEM.L (SP)+,D0        restore registers
131                 UNLK    A6              restore A6
132                 MOVE.L  (SP),4(SP)      kill argument frame: move return address
133                 ADDQ.L  #4,SP                                adjust SP
134                 RTS
135
136
137 ********************************************************************************
138 *
139 * function: (SP1) HTOA( (SP1), (SP2), (SP3))
140 *
141 * description:
142 *   Converts the unsigned hex number (SP3) into an ascii string which is put
143 *   into the buffer whose begin and end is pointed at by (SP1) and (SP2)
144 *   respectively. The starting address within the buffer is returned via the
145 *   stack (SP1).
146 *
147 * registers affected: none
148 *
149
150 HTOA:           LINK    A6,#0           store pointer to argument frame in A6
151                 MOVEM.L D0-D2/A0-A1,-(SP)
152                 MOVE.L  8(A6),D0        fetch hex number
153                 MOVE.L  12(A6),A1       fetch pointer to end of buffer
154                 MOVE.L  16(A6),A0       fetch pointer to start of buffer
155
156                 MOVE.B  #$00,(A1)       write terminating 0 into buffer
157                 MOVE.L  #8,D2           convert 8 digits
158
159 HTOA_LOOP:      MOVE.L  D0,D1
160                 ANDI.L  #$F,D1          separate last digit
161                 ADDI.B  #$30,D1         add '0' in order to get a char '0', ...
162                 CMPI.L  #$39,D1         is char > '9' ?
163                 BLE     HTOA2
164                 ADD.B   #$7,D1          add ('A' - ('9' + 1))
165 HTOA2:          CMPA.L  A1,A0           end of buffer reached?
166                 BEQ     HTOA_OOB
167                 MOVE.B  D1,-(A1)        write char into buffer
168                 LSR.L   #4,D0           shift addr 4 Bit right
169                 SUBI.L  #1,D2
170                 BNE     HTOA_LOOP
171
172                 CMPA.L  A1,A0           end of buffer reached?
173                 BEQ     HTOA_OOB
174                 MOVE.B  #$24,-(A1)      write '$' into buffer
175                 MOVE.L  A1,16(A6)       write result over buffer arg!
176                 JMP     HTOA_DONE
177
178 HTOA_OOB:       MOVE.L  12(A6),16(A6)   out of bounds: return empty string
179
180 HTOA_DONE:      MOVEM.L (SP)+,D0-D2/A0-A1
181                 UNLK    A6              restore A6
182                 MOVE.L  (SP),8(SP)      kill last two entries of arg.frame:
183                 ADDQ.L  #8,SP             move return address, adjust SP
184                 RTS
185
186
187 END
```