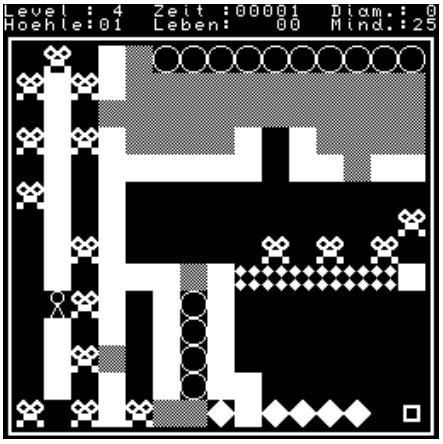
SDCC-Z1013-KC85 Dokumentation

Entwicklungsumgebung für Z1013, Z9001, HC900 sowie deren Nachfolgemodelle KC85/1 bis KC85/5



Inhaltsverzeichnis

Einführung

Voraussetzungen

OS: Linux, Windows (Cygwin) oder MacOS(Brew)

Tools: git, gmake, xserver (zum Testen von GUI-Programmen, Text/Grafik)

SDCC Version 3.6.5

Installation

```
$ git clone --recursive https://github.com/anchorz/sdcc-z1013-kc85.git
```

Der erste Grund eine bestimmte Version einzuchecken war, dass frühere Versionen vor 3.5.09 fehlerhaften Code bei einigen Beispielen erzeugten. Insbesondere `sample_07_kc85_appscreen` (mit seinen riesigen Funktionen) war davon betroffen.

Der zweite Grund ist, dass nach einem Update, hier z.B. auf 3.6.5, plötzlich Fehler gemeldet werden, bei Code der vorher als korrekt wahrgenommen wurde (Siehe weiter hinten `sample_02_compiler`)

Aus Gründen der Reproduzierbarkeit ist jetzt eine Version des SDCC als Submodul mit im Repository eingecheckt, mit der die Beispiele getestet wurden. Der folgende Schritt, sollte eigentlich nicht notwendig sein, wenn eine aktuelle Version des Compilers schon vorhanden ist, aber zur Sicherheit checken wir eine getestete Version noch mit aus.

```
$ sdcc-z1013-kc85/sdcc/sdcc
$ ./configure --disable-mcs51-port --disable-z180-port --disable-r2k-port
--disable-r3ka-port --disable-gbz80-port --disable-tlcs90-port --disable-ds390-
port --disable-ds400-port --disable-pic14-port --disable-pic16-port --disable-
hc08-port --disable-s08-port --disable-stm8-port
$ make
$ make install
$ sdcc -v
SDCC : z80 3.6.5 # (Mac OS X x86_64)
published under GNU General Public License (GPL)
$ _
```

Makefile

`SDCC_OBJECTS=`Dateien nur für SDCC-Ziel
`OBJECTS=`gemeinsame Dateien GCC-Host und SDCC-Ziel

Portierung

`putchar()`

Ist die erste und wichtigste Bibliotheksfunktion, die auf jeder Plattform implementiert werden muss. Intern wird die Ausgabe an die jeweilige Betriebssystemfunktion weitergeleitet. Das mag nicht besonders schnell sein, ist aber der kompatibelste Ansatz. Leider ist bei Bedeutung vom Steuerzeichen insbesondere `\r` und `\n` bei den verschiednen Plattformen jeweils

unterschiedlich. Um bessere Austauschbarkeit zu erzielen wird hier die Unix-Konvention als Standard für das Zeilenende angenommen. So sollte für eine Portierung z.B. jeweils die Funktion `putchar()` dementsprechend angepasst werden. Der Funktionsaufruf ist so etwas aufwändiger, aber für Quelltextkompatibilität sinnvoll. Will man die Zeichen statt dessen direkt ausgeben, kann man die jeweilige Betriebssystemfunktion nehmen, deklariert in `z1013.h`, `z9001.h` oder `kc85.h` .

Beispiel der Z1013 Implementierung

```
_putchar::
      ld    hl, #2
      add   hl, sp
      ld    a, (hl)
loop:
      cp    a, #UNIX_STYLE_NEW_LINE ; 0x0a
      jr    NZ, print_character
      ld    a, #VK_ENTER            ; Code ist 0x0d für eine neue Zeile
print_character:
      rst     0x20
      .db     OUTCH
      ret
```

Beispiele

sample_00_hello

Demo für die Funktion `printf()` und ein Beispiel für eine Host Kommandozeilenprogramm. Da `printf()` erst einmal keine besonderen Anforderungen an die Platform stellt, liegt es nahe das Programm auch auf dem Host zu kompilieren und zu testen.

Anmerkungen:

Intern wird die Ausgabe letztendlich über den Aufruf von `putchar()` durchgeführt. Ansonsten ist die Implementierung gleich auf allen Plattformen.

Verwenden sollte man allerdings die Funktion `printf()` auf 8-bit Rechner nie. Das liegt zum im wesentlich am Speicherverbrauch und der daraus resultierenden Knappheit von Speicher, als auch an der Geschwindigkeit. Die Funktion bringt eine Menge Ballast, der nicht immer verwendet wird, z.B. verwendet sie `signed` und `unsigned` Integer Divisionen.

Einen Kompromiss stellt die SDCC Implementierung dar, bei der die Darstellung von `float` und `double` entfernt wurden. Manche Programme verwenden eine Variante, die gezielt für das Programm „abgerüstet“ wurde, z.B. nur `%s` `%c` und `%04x` darstellen kann ohne jede andere Formatoptionen. Dazu kann man sich den Originalcode aus der Bibliothek hernehmen und jeweils den unbenutzten Teil auskommentieren. Dann wird der Speicherverbrauch im Bereich von 100 oder mehr Bytes liegen.

sample_02_compiler

Die neue Version 3.6.5 gibt eine Fehlermeldung aus, wenn man eine Funktion als

`__z88dk_callee` deklariert und versucht sie in C zu implementieren (Beispiel siehe unten „Alt:“)

src/main.c:90: error 9: FATAL Compiler Internal Error in file 'gen.c' line number '4587' : Unimplemented __z88dk_callee support on callee side

> sdcc is able to call smallc and z88dk_callee functions that are written in assembly language but it is not able to compile c functions...

Das Problem ist aber nicht sehr gross. Da aber das alte Beispiel schon in Assembler programmiert wurde, eigentlich genau aus dem Grund, dass man den Stack manuell korrigieren muss, muss man den Code der Funktion nur noch in eine eigene Assemblerdatei kopieren. Die Änderung hat aber auch den Vorteil, dass so Assemblercode aus dem C Quelltext entfernt werden und so etwas sauberer programmiert werden muss.

Alt: `main.c`

```
void OUTSTR_CALLEE(int c1, int c2, int c3) __z88dk_callee
{
    __asm__("pop iy"); //return address
    __asm__("pop hl");
    __asm__("call _put_char_int");
    __asm__("pop hl");
    __asm__("call _put_char_int");
    __asm__("pop hl");
    __asm__("call _put_char_int");
    __asm__("push iy");
    __asm__("ret");
}
```

Neu: `callee.s`

```
_OUTSTR_CALLEE::
        pop iy; //return address
        pop hl
        call _put_char_int
        pop hl
        call _put_char_int
        pop hl
        call _put_char_int
        push iy
        ret
```