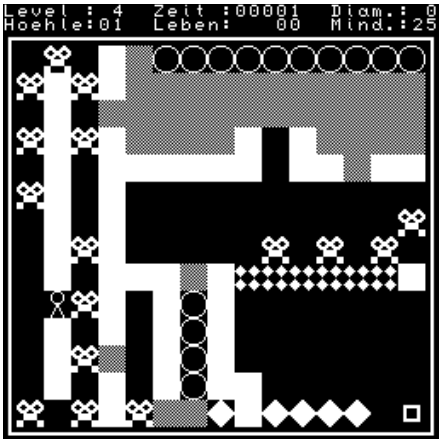
SDCC-Z1013-KC85 Dokumentation

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

Release Notes

`mk/rules.mk: --Werror` behandelt alle Warnungen als Fehler. Das erzwingt hoffentlich saubereren Code. Zusammen mit der Einstellung für den `gcc -Wall -pedantic -Werror` ergibt das hoffentlich auch einen portableren Quelltext.

`caos_version()` return 0x31, 0x41 oder 0x22

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

Speichermodel

Layout. Vergleich mit herkömmlichen C-Programmen. Heap.

Beispiele siehe header.s oder .z80 header – Wo ist das Programm? An welcher Stelle fängt der freie Speicher an?

ROM code – Initialisierung von Variablen, Tabelle im ROM,

RAM code

Stack – Eingeschränkt auf 1KByte. Genügt für die meisten Anwendungen.

`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` (siehe sample_03_z1013_hello weiter unten)

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.

sample_01_printf

Hier noch mal ein erweitertes Beispiel für die Verwendung von `printf()`. Nebenbei ist es auch ein Testprogramm für den Startupcode `crt0.s`.

Intern wird die Ausgabe letztendlich über den Aufruf von `putchar()` durchgeführt. Ansonsten ist die Implementierung von `printf()` 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.

Von besonderer Bedeutung in diesem Beispiel ist die Map-Datei z.B. `obj/kc85/printf.map`. Hier kann man sehen, dass initialisierte Variablen und uninitialisierte in verschiedenen Speicherbereich liegen.

```
_INITIALIZER                            0000150F    0000000E =          14. bytes
_DATA                                   00001544    00000002 =           2. bytes
00001544  _w1                                   main
_INITIALIZED                            00001546    0000000E =          14. bytes
00001546  _str                                  main
00001548  _w2                                   main
...
```

Die Programmdatei belegt in diesem Beispiel den Speicherbereich bis `0x1543`. Danach beginnt der eigentliche freie Speicher. Dieser muss vor dem Aufruf von `main()` initialisiert werden. Dafür ist der Startupcode in `crt0.s` verantwortlich, d.h. der Speicher für uninitialisierte Variablen, hier in dem Fall 2 Bytes groß, muss mit `0x00` gefüllt werden und der Speicher für initialisierte Variablen (`_INITIALIZED`) wird mittels einer Tabelle aus dem Programmspeicher (`_INITIALIZER`) gefüllt. Letzteres mag auf den ersten Blick ineffizient erscheinen, ich habe aber den original Startupcode verwendet, das man so die Programme auch im ROM abgelegen kann.

Das Beispiele wurde auch verwendet, um den Startup zu debuggen und nachzuschauen, wie die einzelnen Segmente initialisiert werden. In früheren Versionen von `crt0.s`, wurden Datensegmente `_DATA` der Größe 0 oder 1 entweder falsch initialisiert oder es ist zum Programmabsturz gekommen, einfach weil man vergessen hat richtig zu zählen.

sample_02_compiler

Beispiel für die Macros der verschiednen Platformen `__Z9001__`, `__Z1013__`, `__KC85__` und `__GNUC__` . Eine weitere Unterscheidung für KC85/2...5 wird vorerst von der Entwicklungsumgebung nicht angeboten. Die Idee dahinter ist, dass die Unterscheidung vom

Programm oder besser noch zur Laufzeit vorgenommen wird. Eine Ausnahme könnte man für die Bildschirmaufteilung machen. Es ist zwar kein Problem die Routinen zur Laufzeit anzupassen, aber dann hat man im Einzelfall immer noch zwei verschiedene Implementierung gemeinsam im Speicher, von der jeweils eine nie verwendet wird.

und für spezielle Erweiterungen des SDCC `__z88dk_fastcall`, `__z88dk_callee`, `__naked`, `__sfr` (8 und 16-bit IO-Zugriff!), `__at`. Die Dokumentation des SDCC ist in meinen Augen an der Stelle etwas mangelhaft. Deswegen will ich hier die mir bekannten Features auflisten und am Beispiel demonstrieren.

Es erscheint auf den ersten Blick etwas widersinnig hier auch noch eine GCC Version mitzuliefern, aber das Beispiel soll auch demonstrieren, wie man trotz der spezifischer Erweiterungen für den plattformunabhängigen Teil wiederverwenden kann.

Seit der Version 3.6.5 des SDCC gibt es eine Fehlermeldung, wenn man eine Funktion als `__z88dk_callee` deklariert und versucht sie in C zu implementieren (siehe ehemaliges Beispiel, 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
```

Das Problem ist aber nicht sehr dramatisch. Da 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 noch den Vorteil, dass so Assemblercode aus dem C Quelltext getrennt wird 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
```

sample_03_z1013_hello

Ein Beispiel für die Verwendung von Betriebssystemaufrufen zur Textausgabe. Der eigentliche Clou hier ist, das Macro PRST7(). In C werden Strings normalerweise im CONST DATA Segment abgespeichert und vor dem Funktionsaufruf die Adresse auf den Stack gelegt. Der Standartfunktionsaufruf würde dann etwa so aussehen:

```
ld hl,#ADR
push hl
call foo
```

Bei den meisten 8-Bit Recher ist dies nicht unbedingt notwendig. Das Betriebssystem hat Funktionen, die den Text gleich aus dem folgenden Code extrahieren.

```
call foo
.db  'my text',0
```

Das ist nicht unbedingt schneller, da man ja erst über den Stack die eigentliche Startadresse ermitteln muss, aber so spart man schon ein paar Bytes bei jeder Textausgabe.

Die zweite Eigenheit ist das Endekennzeichen beim Text, C-Konvention nimmt ein terminierendes Nullbyte. Die PRST7-Routine des Z1013 erwartet ein gesetztes Bit7. Noch interessanter ist die Tatsache, dass diese Konvention vom sdas Assembler unterstützt wird. Ich weiss von keinem anderen System, dass die gleiche Konvention benutzt. Der Assembler .ascis bietet. Schreibt man,

```
call foo
.ascis  'my text'
```

, so setzt der sdas Assembler beim letzten Zeichen das 8. Bit und wir haben noch ein Byte gespart. Jetzt muss man dem Compiler noch mitteilen, den Text in das Code Segment zu packen, gleich nach dem Funktionsaufruf.

```
#define PRST7(X) \
  __asm__("rst 0x20"); \
  __asm__(".db 2 ;PRST7"); \
  __asm__(".ascis "#X);
#endif
```

Sinngemäß funktioniert das auch für andere Rechner.

sample_04_kc85_modulescan

MODULE SCAN:
08 F6 0C 7B 10 EE
>

sample_05_kc85_graphic

Grafikdemo für Linien und Kreise. Diese Funktionen sind nicht auf allen CAOS Versionen vorhanden, deshalb gibt es hier noch eine Versionsabfrage:

```
if (caos_version()>=0x31)
{
    circle(graphic_widht2, graphic_heigth2, index, color);
}
```