INLINE ASSEMBLER

AVR GCC

-erklärt für extended asm statements-

Folie 1:

¿WARUM?

Erstellung von sehr kompakten, stabilen, und effizienten Code.

-

Nutzung von Anweisungen die über AVR-GCC einfach nicht direkt erreichbar sind.

Folie 2:

ERKLÄRUNG VON ASM ANWEISUNGEN ANHAND VON BEISPIELEN

- **MOV** – Führt eine Kopie vom rechten Argument zum Linken aus.
- **SBIW** – Ein Register Paar (Beispiel Pointer Register 16 Bit) erfährt eine Subtraktion.
- **In** – Ladet Daten vom I/O Space (Ports, Timers, Configuration Registers, etc.)

<u>Amerkung:</u>

Beim AVR ASSEMBLER-Dialekt liegt überwiegend ein Rechtsassoziatives Verhalten der Anweisungen vor.

Folie 3:

OPERANTEN

- Die wichtigsten, in richtig zu verwendender Reihenfolge, sind die Augänge und Eingänge.
- Falls einer der beiden nicht verwendet wird, so ist zumindest ein Doppelpunkte anzugeben.

:)

Folie 4:

MODIFIER + CONSTRAINTS

- Constraints werden den Operanten vorgestellt generell gilt folgendes Shema.
  - [[asmSymbolicName]] constraint (cvariablename)
- Sie sind sehr vielfältig und beschreiben Orte für das allozieren sowie die Verwendung des Operanten.

- Folgende Modifier sind als Prefix verwendbar.
- = Write Only gut für Ausgänge
- NVA Read Only
- & alloziertes Register sollte nur für den Output verwendet werden
- + Read/Write nur für Ausgang
- Es wird vom Compiler nicht angenommen das Eingänge überschrieben werden.
- (Jedoch möglich !)

Folie 5:

SYMBOLISCHE LOGIK VON ASM STATEMENTS

- Sofern man für seine Operanten keine symbolische Namen

[[asmSymbolicName]] constraint (cvariablename)
[port] "I" (_SFR_IO_ADDR(PORTD))

- verwendet, gilt, von oben nach unten, im Assembler Template, dasselbe was bei Arrays gilt: 0, 1, 2, 3 und steigend . Bedeutet, die Assembler-Anweisungen beziehen sich dann mit diesen Zahlen auf den ersten Operanten -0- bis zum Letzen, gebunden an die Definitions-Reihenfolge.
- Es gibt auch eine kombinierte Logik, welche verwendet wird um das Low Byte, dazwischenliegende Bytes oder das High Byte anzusprechen.
- Beispiel Low- bis High Byte für Operanten Eins, wenn Register 32 Byte groß ist:

%A0, %B0, %C0, %D0

Folie 6:

AUF WAS MAN ACHTEN SOLLTE

- Das der Compiler asm Positionen relativ zueinander neu anordnen kann. Und somit müssen Anweisungen, welche einer Reihenfolge bedürfen, in einem asm Statement geschrieben werden. (Künstliche Abhängigkeiten sind ein Ausweg – Fragt mich!)
- Der Compiler weiß nicht die Zusammenhänge von außenstehenden Referenzen und deren Benutzung in asm. Deshalb kann es passieren, dass außerhalb stehende Referenzen Optimierung erfahren. Somit kann es nötig werden, die außerhalb stehenden Referenzen ebenfalls mit volatile zu versehen, um dem zu entgehen.

Folie 7:

- GCC generiert Duplikationen oder entfernt solche möglicherweise im Rahmen der Optimierung. Aushilfe ist vor allem besser aufgebauter Code.

Folie 8:

- Extended asm kann nur innerhalb einer Funktion verwendet werden.
- Der & constraint qualifier ist für jene Ausgänge zu benutzen, wo sichergestellt sein soll, dass sich diese nicht mit den Eingängen überlappen. GCC alloziert eventuell den Ausgangs-

Operator mit dem selben Register, welches bereits für einen davon unabhängigen Eingangs-Operator Verwendung findet. GCC nimmt dabei an, dass die Eingänge vor den Ausgängen verarbeitet werden.

Weiter unten wird die Anwendung gezeigt.

Folie 9:

- Die Benutzung vom Spezial-Register __tmp_reg__ ist ein sinnvoller Ausweg zu Clobbers (Fragt mich!).
- Allgemein ist die Benutzung von solchen Registern geeignet, um sehr effizienten Code zu generieren. So deutet zum Beispiel im __SREG__ das Bit **Zero Flag Z** das letze Rechenergebnis der ALU auf Null an. Dies ist sinnvoll, wenn man vor gehabt hätte, eine Variable auf null zu überprüfen.

Symbol	Register
__SREG__	Status register at address 0x3F
__SP_H__	Stack pointer high byte at address 0x3E
__SP_L__	Stack pointer low byte at address 0x3D
__tmp_reg__	Register r0, used for temporary storage
__zero_reg__	Register r1, always zero

Folie 10:

WANN VOLATILE ? (Wann nicht? Fragt mich!)

- Wenn es wahrscheinlich ist, dass der Optimierer keine zukünftige Verwendung für die Ausgangsvariabeln erkennt.
- Wenn der Optimierer meinen könnte das sich der Output gar nie ändert.
- Wenn Code erzeugt wird, der gewünschte Seiteneffekte erzeugen soll, welche vom Optimierer aber als ungünstig erkannt werden.

Folie 11:

ASSEMBLER TEMPLATE

- Besteht aus mindestens einer Zeichenkette, welche eine Assembler- Anweisung beinhaltet.

Beispiel sehr sehr einfach:

```
asm volatile("cli"::);
```

Beispiel sehr einfach:

```
asm("in %[retval], %[port]" :
[retval] "=r" (value) :
[port] "I"(_SFR_IO_ADDR(PORTD)) );
```

Beispiel einfach:

```
asm volatile("mov __tmp_reg__, %A0"
             "mov %A0, %B0"
             "mov %B0, __tmp_reg__"
             : "=r" (value)
             : "0" (value));
```

Extra: Hier wird mit "0" mitgeteilt das value zugleich als Ausgang und Eingang fungieren soll.

Der Code selbst ermöglicht das Vertauschen von High – und Low Byte.

Folie 12:

NUN ERFOLGT PRAXIS

```
asm volatile("in %0,%1"
             "out %1, %2" :
             "=&r" (input):
             "I" (_SFR_IO_ADDR(port)),
             "r" (output) );
```

BESCHREIBUNG

- Alle folgenden Operanten referenziern auf defines oder Deklarationen außerhalb des extended asm statements.
- Vom Eingabe Operant port (z.B. PORTB), wird in den Ausgangs Operant input (z.B. char character;) eingelesen. Es ist hier ein wichtiger Modifier gewähtl worden. Der mit "=&r" (input)definierte Ausgang wurde mit =& statt nur mit = modifiziert. Die Bedeutung ist wie bereits erwähnt, dass vom Compiler ein Register gewählt wird, welches ausschließlich für Ausgänge verfügbar ist.
- Nun zeigt sich der Bedarf:
- In der zweiten Anweisung wird der Eingang output, welcher ohne genannter Technik der bereits überschriebene Ausgang output wäre, auf den Eingang port geschrieben.
- ! Die Ausnahme, auf den Eingang port schreiben zu können, erfolgt durch das Compiler-Attribut _SFR_IO_ADDR !

Folie 13:

HILFREICHE TIPPS

- Wenn mehrere Constraints angegeben werden, so ist der Compiler befähigt, den effizientesten Ort für den Operanten, bezogen auf den aktuellen Kontext, auszuwählen. Besteht also Unsicherheit kann man zum Beispiel mit “=rm”, den Compiler entscheiden lassen ob ein Register oder der Memory ausgewählt wird.
- Es ist ratsam, sich zu informieren, welchen Assembler-Dialekt das System unterstützt.
- Im Handbuch “atmel 0856 avr instruction set” existiert eine komplette Übersicht von Assembler-Befehlen.
- Das Forum AVR-Freaks ist ein guter Anlaufpunkt für Nachforschungen.

Folie 14:

QUELLEN-ANGABE

- avr-libc-user-manual-1.8.1.pdf
- Neueste GCC Dokumentation
- atmel-0856-avr-instruction-set-manual.pdf
- AVR-Freaks Forum

Folie 15:

WEITERFÜHRENDER LINK

http://gcc.gnu.org/onlinedocs/

EOF