UNTERPROGRAMME IN MMIX(AL)

Oft möchten wir eine Funktionalität mehrfach benutzen. Nun könnten wir den Code an jede Stelle kopieren, was aber eine Reihe von negativen Konsequenzen zur Folge hätte.
Copy & Paste – wie diese Programmier-Unkunst auch bezeichnet wird – hat aber viele Nachteile:

- Änderungen müssen an jeder Stelle im Code vollzogen werden.
- Das Ändern von kopiertem Code dauert länger und ist somit sehr unflexibel.
- Der Code ist weniger strukturiert und schwerer zu verstehen.

Das Ziel von Unterprogrammen ist es, eine spezielle Aufgabe zu lösen und eine genau spezifizierte Funktionalität anzubieten. Das Beispiel an dem sich dieser Text orientieren wird, ist das Interpretieren einer Zeichenkette im Speicher als Zahl.

Anmerkung: *Wir verwenden im Folgenden Unterprogramm, Funktion und Prozedur synonym.*

In Java würde eine solche Funktion zum Konvertieren eines Strings z.B. so aussehen:

```
Integer convertString(String stringToConvert) throws InvalidFormatException
{
        MAGIC;                  //konvertiere den String in eine Zahl
        return result;          //gib das Ergebnis zurück
}
```

Eine Funktion wird in einer höheren Programmiersprache durch folgende Eigenschaften definiert:

Parameter	Funktionen können Variablen übergeben werden. In unserem Beispiel ist dies der String stringToConvert.
Rückgabewert	Eine Funktion liefert in der Regel ein oder mehrere Ergebnisse zurück. In unserem Beispiel wäre dies eine Integer-Variable - nämlich die interpretierte Zahl.
Ausnahmen	In höheren Programmiersprachen werden so genannte Exceptions (dt. Ausnahme) benutzt, um Ausnahmesituationen bei der Ausführung zu kennzeichnen. In unserem Beispiel wäre dies z.B. der Fall, wenn eine „Nicht-Ziffer“ im String enthalten war und somit der String nicht konvertiert werden kann. Sollte ein Zeichen ungleich einer Ziffer im String vorhanden sein, so würde die Ausführung des Unterprogramms abgebrochen werden und eine InvalidFormatException geworfen werden. Exceptions können vom Aufrufer dann gezielt behandelt werden.

Parameter und Rückgabewert – und wenn vorhanden Ausnahmen – werden zusammen als Signatur bezeichnet. Mit dem Namen des Unterprogramms ist dies die Schnittstelle, über welche das Unterprogramm von außen aufgerufen werden kann. Mittels eines Kommentars sollte das Unterprogramm und die Bedeutung der einzelnen Parameter sowie der Rückgabewerte beschrieben werden.

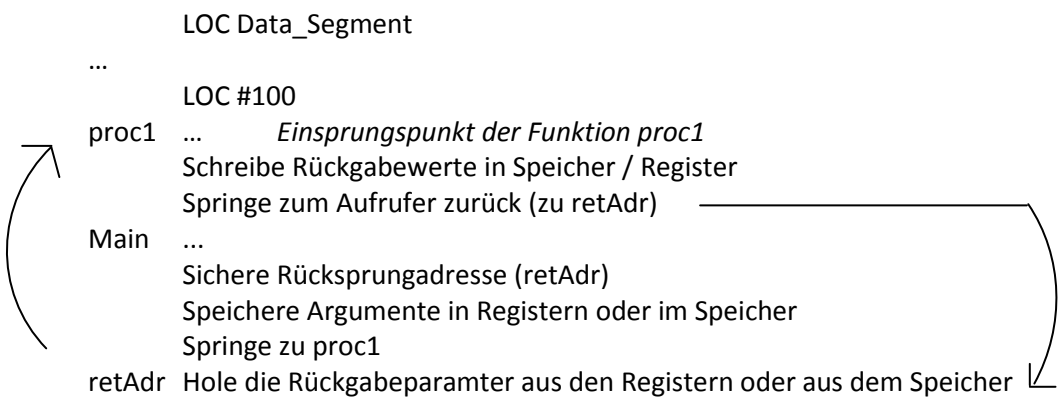
GRUNDSÄTZLICHE UMSETZUNG IN MMIX(AL)

Java oder ähnlich hohe Programmiersprachen sind Prozeduren oder Funktionen von Hause aus bekannt, d.h. es gibt in diesen Sprachen Konstrukte die eine solche Funktionalität anbieten. MMIXAL kennt als

Maschinensprache keine solchen Konstrukte, unterstützt den Programmierer aber durch spezielle Befehle. Wie man diese benutzt und welche Vorteile die einzelnen Varianten besitzen, wird im Folgenden vorgestellt.

Die Skizze weiter unten zeigt den theoretischen Ablauf eines Unterprogrammaufrufs. Die Prozedur proc1 wird von dem Hauptprogramm aus aufgerufen. Dazu müssen zuvor die Argumente im Speicher oder in Registern abgelegt werden. Zusätzlich muss die Rücksprungadresse gespeichert werden, damit später wieder zurückgesprungen werden kann.

Nachdem die Funktion aufgerufen wurde, d.h. der Programcounter auf die entsprechende Speicherstelle (Label proc1) gesetzt wurde, und die eigentliche Logik der Prozedur ausgeführt wurde, werden die Ergebnisse im Speicher oder in den Registern abgelegt und zum Hauptprogramm zurückgesprungen, d.h. der Programcounter wird auf die Anweisung gesetzt, welche in dem Hauptprogramm nach dem Prozeduraufruf gefolgt wäre. Das Hauptprogramm kann dann die Rückgabewerte auslesen und verwenden.



(Primitive) Umsetzung mit GO

Als erste Variante um Unterprogramme zu schreiben, betrachten wir die GO Anweisung.

GO $X,label	Die Programmausführung wird an der Speicheradresse label fortgesetzt. Die Adresse des nächsten auszuführenden Befehls wird in Register X abgelegt. Bei dem Aufruf von GO wird eine absolute Adressierung benutzt. Es muss also ein globales Register mit der entsprechenden Speicheradresse in der Nähe des Einsprungspunkts der Prozedur vorhanden sein.
GO $X,$Y,$Z	Diese Version des GO-Befehls hat die gleiche Semantik wie die obige. Die Ausführung des Programms findet nach diesem Befehl an der Adresse ($Y+$Z) statt. Wieder wird in das Register X der nächste auszuführende Befehl geschrieben.

Mittels der GO Anweisung ist es möglich einfache Unterprogramme zu schreiben, wie die erste Implementierung unserer String → Zahl Konvertierung zeigt:

```
                LOC Data_Segment
                GREG @
strZahl         BYTE "31415",0
strError        BYTE "konvertierung nicht erfolgreich",#A,0
strOk           BYTE "konvertierung hat funktioniert",#A,0
strZahlIst      BYTE         "zahl ist 31415!",#A,0

                LOC #100
```

Unterprogramm

```
                GREG @
                PREFIX cSTN:
retJmpAdr       IS $10
strAdr          IS $11
retOk           IS $12
retNr           IS $13
char            IS $14
counter         IS $15
cmpR            IS $16
ascii0          IS '0'
ascii9          IS '9'
basis           IS 10
OK              IS 0
notOK           IS 1
                //cSTN
                //convertStringToNumber
                //konvertiert einen ASCII-String in eine Zahl
                //Parameter
                //$10 Rücksprungadresse (read-only)
                //$11 Adresse des Strings (read-only, wird als dezimal Zahl interpretiert)
                //Rückgabewert
                //$12 Ergebnis der Funktion:
                //    1 - erfolgreiche Ausführung
                //    2 - ein Fehler ist aufgetreten
                //$13 Konvertierte Zahl
                //Benutzte Register: $14, $15, $16
cSTN            SET retOk,OK //initialisiere Rückgabeparameter, Funktion grundsätzlich erfolgreich
                SET retNr,0 //Standardzahl ist 0
                SET char,0
                SET counter,0
cSTNloop        LDBU char,strAdr,counter //lade Byte
                BZ char,cSTNexit     //überprüfe ob 0-Byte ist
                CMP cmpR,char,ascii0         //überprüfe ob es eine gültige Ziffer (>='0') ist
                BN cmpR,cSTNerror
                CMP cmpR,char,ascii9     //überprüfe ob es eine gültige Ziffer (<='9) ist
                BP cmpR,cSTNerror
                SUBU char,char,ascii0        //extrahiere Ziffer
                MULU retNr,retNr,basis       //hornerschema zur Interpretation der Zahl
                ADDU retNr,retNr,char
                ADDU counter,counter,1
                JMP cSTNloop                //verarbeite nächstes Zeichen
cSTNerror       SET retOk,notOK
cSTNexit        GO retJmpAdr,retJmpAdr,0
```

Hauptprogramm

```
%Start des Hauptprogramms
        PREFIX :
cmpR    IS $0
strAdr  IS $1
Main    LDA :cSTN:strAdr,strZahl  //lade Adresse der Zahl (als AsciiCode)
        GO :cSTN:retJmpAdr,:cSTN:cSTN  //rufe die Funktion auf
        CMP cmpR,:cSTN:retOk,:cSTN:OK  //überprüfe, ob die Funktion erfolgreich ausgeführt wurde
        BP cmpR,convertError           //falls nicht, gib einen Fehler aus
        LDA $255,strOk
        JMP printStr
convertError LDA $255,strError
printStr TRAP 0,Fputs,StdOut            //Ausgabe..
        SET cmpR,31415
        CMP cmpR,:cSTN:retNr,cmpR      //Überprüfe, ob die richtige Zahl erkannt wurde
        BNZ cmpR,exit                  //falls nicht, beende das Programm
        LDA $255,strZahlIst            //ansonsten gib aus, dass die richtige Zahl zurückgegeben
wurde
        TRAP 0,Fputs,StdOut
exit    TRAP 0,Halt,0
```

GO - Analyse des Codes

Die Verwendung von GO hat einige schwere Nachteile:

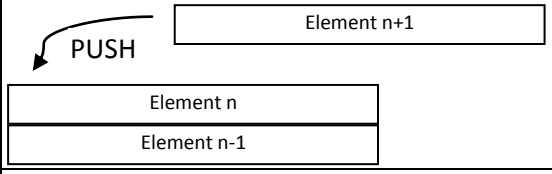
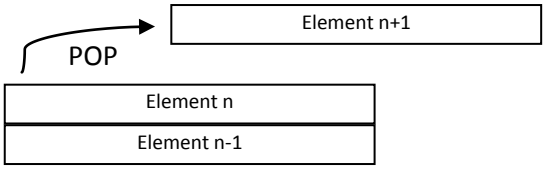
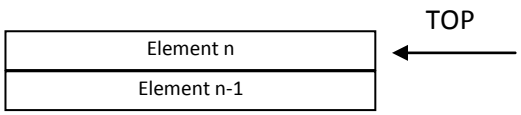
- Parameter und Rückgabewerte werden direkt über Register vereinbart. Dies führt dazu, dass bei größeren Programmen die Übersicht über die bereits verwendeten Register verloren geht. Dies führt z.B. dann zu Problemen, wenn eine Prozedur aus bestehendem Code herauskopiert wird um sie in neuem Code einzufügen, da beide Codeteile vielleicht die gleichen Register benutzen.

- Durch den ersten Punkt bedingt sind **Seiteneffekte** möglich. Als Seiteneffekt wird ein Verhalten genannt, dass durch die Schnittstelle der Funktion (und des Kommentars) nicht definiert sind. Dies wäre z.B. der Fall, wenn das Unterprogramm ein Register des Hauptprogramms überschreibt und somit die Daten des Hauptprogramms korrumpieren würde.
- Gemäß dieser primitiven Implementierung sind **keine rekursiven Funktionsaufrufe** möglich, da das Register, an der die Rücksprungadresse gespeichert wird, immer das gleiche ist und somit beim erneuten Aufruf die eigentliche Rücksprungadresse überschrieben werden würde.

Für kleinere, einfache Programme ist die Verwendung der GO-Anweisung akzeptabel, während sie bei komplexerem Code (inkl. Rekursiven Funktionsaufrufen) nicht praktikabel ist.

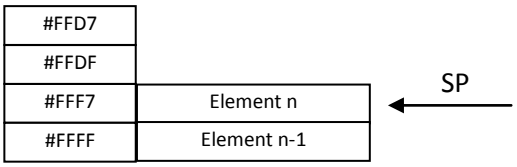
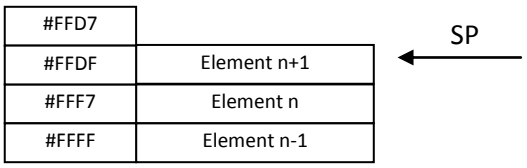
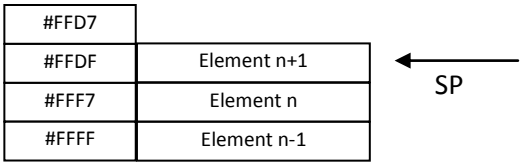
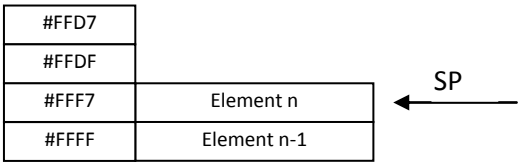
UMSETZUNG MIT EINEM STACK

Die heutigen Lösungen bei Funktionsaufrufen basieren immer auf einem so gennannten Stack (dt. Stapel oder Keller). Ein Stack ist eine Datenstruktur, die drei Operationen anbietet:

PUSH	Ein Element wird oben auf den Stapel gelegt.	
POP	Das oberste Element wird von dem Stack genommen.	
TOP	Es wird das oberste Elemente gelesen.	

Am besten kann man einen Stack anhand eines Tellerstapels darstellen. Wir stapeln mehrere Teller übereinander. Wir können dabei jeweils nur einen Teller auf den Stapel legen oder einen Teller vom Stapel entfernen. Zusätzlich können wir den „Inhalt“ des obersten Tellers einsehen. Wir können jedoch keinen Teller in die Mitte des Stapels schieben und auch keinen Teller aus der Mitte entfernen.

Der Stack wird mittels einem Zeiger auf das oberste Element im Speicher implementiert. Die konzeptionelle Implementierung ist in der folgenden Tabelle zu sehen:

Operation	Vor der Operation	Nach der Operation
PUSH		
POP		

In der obigen Tabelle sind zusätzlich die Speicherstellen dargestellt, an denen die Elemente exemplarisch abgelegt wurden.

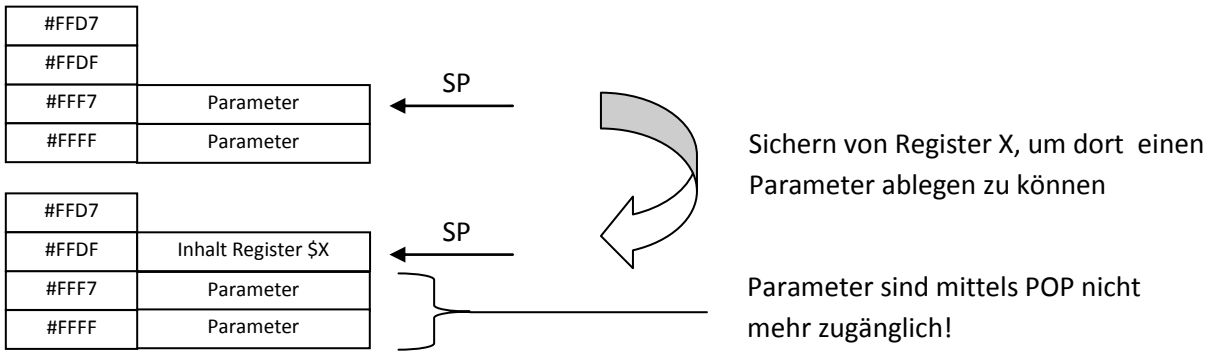
Funktionsaufruf mit einem Stack

Bei der Verwendung eines Stacks, wird im Regelfall wie folgt vorgegangen:

1. Das Hauptprogramm legt Parameter und Rücksprungadresse auf dem Stack.
2. Das Hauptprogramm ruft das Unterprogramm auf.
    a. Das Unterprogramm sichert die Werte der Register, die es benutzen wird, indem es sie auf den Stack legt.
    b. Das Unterprogramm verarbeitet die Parameter und speichert den / die Rückgabewert(e) auf dem Stack oder in Registern.
    c. Das Unterprogramm stellt die Inhalte der Register, die es auf dem Stack gespeichert hatte, wieder her und springt zum Hauptprogramm zurück.
3. Das Hauptprogramm wertet den / die Rückgabewert(e) auf dem Stack oder in den Registern aus.

Die oben genannten Funktionen PUSH und POP mit einem Stackpointer reichen jedoch leider nicht aus, um Funktionsaufrufe sinnvoll zu implementieren:

- Möchte das Unterprogramm auf die übergebenen Parameter zugreifen, muss es diese in Register laden. Bevor es jedoch Register selbst verwenden kann, müssen die Inhalte der Register (auf dem Stack) gesichert werden, um diese später wiederherzustellen. Da nun aber die Parameter unterhalb der gesicherten Werte der Register im Stack liegen, kann auf diese nicht mehr zugegriffen werden:



Um dieses Problem zu lösen, gibt es in modernen PC-Architekturen (u.a. x86) oft zwei Stackpointer:

- ✓ SP – Der Stackpointer zeigt immer auf das oberste Element des Stacks.
- ✓ BP – Der Basepointer zeigt auf das unterste Element des Stacks, welches von der aktuellen Prozedur auf den Stack gelegt wurde. Mittels des Basepointers können nun Elemente unterhalb und oberhalb dieses adressiert werden.

In der x86-Architektur gibt es für diese beiden Zeiger eigene Register innerhalb der CPU (ESP und EBP).

Beispiel: *C Function Call Convention*

Es gibt viele verschiedene Möglichkeiten Funktionsaufrufe mittels eines Stacks zu implementieren. Auf Grund dieser vielen Aufrufkonventionen und zur Verdeutlichung der Verwendung des SP und des BP werden wir nun die C Function Call Convention untersuchen und in MMIX nachbauen. Wir werden also per

Hand in MMIXAL den entsprechenden Code, den der Compiler bei einem normalen Funktionsaufruf in C erzeugen würde, erstellen.

Hinweis: *Da es sich bei MMIX um ein 64 Bit System handelt, definieren wir, dass PUSHes bzw. POPs den Stackpointer jeweils um 8 Byte verändern.*

Wir gehen davon aus, dass wir uns im Hauptprogramm befinden und der Stack noch nicht benutzt wurde. Der Basepointer sowie der Stackpointer zeigen somit auf das untere Ende des Stacks, auch Bottom Of Stack (BOS) genannt. In der unten stehenden Tabelle werden nun die einzelnen Schritte, die bei dem Funktionsaufruf, während der Ausführung des Unterprogramms und beim Rücksprung beachtet vollzogen werden, beschrieben.

<table>
<tr><td rowspan="2">Hauptprogramm</td><td>Der Stack vor dem Funktionsaufruf ist leer.</td><td><table><tr><td>BOS (Bottom of Stack)</td><td>←BP, SP</td></tr></table></td></tr>
<tr><td>Die aufrufende Funktion reserviert Speicher auf dem Stack für die Parameter, die Rücksprungadresse und die Rückgabewerte. Der Aufrufer speichert die Parameter, die Adresse für die Rückgabewerte und die Rücksprungadresse auf dem Stack und ruft die Funktion auf.</td><td><table><tr><td>Rücksprungadresse</td><td>←SP</td></tr><tr><td>Speicheradresse für Rückgabewert #1</td><td></td></tr><tr><td>...</td><td></td></tr><tr><td>Speicheradresse für Rückgabewert #k</td><td></td></tr><tr><td>Parameter #1</td><td></td></tr><tr><td>...</td><td></td></tr><tr><td>Parameter #n</td><td></td></tr><tr><td>Platz für Rückgabewerte</td><td></td></tr><tr><td>BOS (Bottom of Stack)</td><td>←BP</td></tr></table></td></tr>
<tr><td rowspan="2">Unterprogramm</td><td>Die aufgerufene Funktion sichert den Basepointer und richtet seinen eigenen Basepointer ein. Er kann so mittels einer relativen Adressierung bezogen auf den BP die Rücksprungadresse und die Parameter adressieren. Dieser Vorgang wird auch als Erzeugen seines „Stackframes“ bezeichnet.</td><td><table><tr><td>BP des Hauptprogramms</td><td>←BP,SP</td></tr><tr><td>Rücksprungadresse</td><td>BP + 8</td></tr><tr><td>Speicheradresse für Rückgabewert #1</td><td>BP + 16</td></tr><tr><td>...</td><td></td></tr><tr><td>Speicheradresse für Rückgabewert #k</td><td>BP + (k+1)*8</td></tr><tr><td>Parameter #1</td><td>BP + (k+2)*8</td></tr><tr><td>...</td><td>..</td></tr><tr><td>Parameter #n</td><td>BP + (k+n+1)*8</td></tr><tr><td>Platz für Rückgabewerte</td><td></td></tr><tr><td>BOS (Bottom of Stack)</td><td></td></tr></table></td></tr>
<tr><td>Das Unterprogramm sichert nun die Register, die es benutzen möchte.</td><td><table><tr><td>gesicherter Inhalt von Register $Y</td><td>←SP</td></tr></table></td></tr>
</table>

Unterprogramm

	gesicherter Inhalt von Register $X BP des Hauptprogramms ←BP Rücksprungadresse BP + 8 Speicheradresse für Rückgabewert #1 BP + 16 ... Speicheradresse für Rückgabewert #k BP + (k+1)*8 Parameter #1 BP + (k+2)*8 ... Parameter #n BP + (k+n+1)*8 Platz für Rückgabewerte BOS (Bottom of Stack)
Das Ergebnis der Berechnungen wird in den übergebenen Speicheradressen (BP+16 bis BP + (k+1)*8) gespeichert. Die Inhalte der Register (in diesem Beispiel $X und $Y) werden wiederhergestellt.	BP des Hauptprogramms ←SP, BP Rücksprungadresse BP + 8 Speicheradresse für Rückgabewert #1 BP + 16 ... Speicheradresse für Rückgabewert #k BP + (k+1)*8 Parameter #1 BP + (k+2)*8 ... Parameter #n BP + (k+n+1)*8 geschriebene Rückgabewerte BOS (Bottom of Stack)
Als letzte Vorbereitung für den Sprung zurück zum Hauptprogramm, wird der Basepointer des Hauptprogamms wieder an die richtige Stelle gesetzt. Somit wird der Stackframe des Hauptprogramms wiederhergestellt. Danach wird die Rücksprungadresse vom Stack gelesen und zurückgesprungen.	Rücksprungadresse ←SP Speicheradresse für Rückgabewert #1 BP + 16 ... Speicheradresse für Rückgabewert #k BP + (k+1)*8 Parameter #1 BP + (k+2)*8 ... Parameter #n BP + (k+n+1)*8 Rücksprungadresse BP + 8

		geschriebene Rückgabewerte BOS (Bottom of Stack) ← BP
Hauptprogramm	Das Hauptprogramm richtet den Stackpointer so aus, dass er auf die Rückgabewerte im Stack zeigt. Die übergebenen Parameter werden dadurch vom Stack gePOPt..Es kann nun auf die Ergebnisse der Funktion zugegriffen werden.	geschriebene Rückgabewerte ←SP BOS (Bottom of Stack) ← BP

Hinweis: *Der oben skizzierte Mechanismus ist eine leichte Abwandlung der richtigen C Calling Convention. Normalerweise gibt es in C genau einen Rückgabewert, welcher gemäß den primitiven Datentypen in C (32 Bit) nur 4 Byte groß ist (z.B. ein int). Dieser wird der Rückgabewert der Funktion in einem speziellen Register (EAX) abgelegt.*
Soll in C jedoch eine Funktion einen Wert zurückgeben, der größer als 4 Byte (bei 32 Bit-Systemen) ist, so wird – wie in unserem Beispiel – zuvor ein Speicherbereich reserviert, der groß genug ist und dann die Adresse dieses Bereichs als impliziter Parameter übergeben. Desweiteren werden in höheren Programmiersprachen alle Variablen im Speicher – und dabei oft auf dem Stack – angelegt und erst bei Verwendung in die Register geladen werden. Somit hat der Stack im Allgemeinen die Form:

gesicherter Inhalt von Register $Y	←SP
gesicherter Inhalt von Register $X	
Lokale Variable m	BP – m*8
...	
Lokale Variable 1	BP - 8
BP des Hauptprogramms	←BP
Rücksprungadresse	BP + 8
Speicheradresse für Rückgabewert #1	BP + 16
...	
Speicheradresse für Rückgabewert #k	BP + (k+1)*8
Parameter #1	BP + (k+2)*8
...	..
Parameter #n	BP + (k+n+1)*8
Platz für Rückgabewerte	
BOS (Bottom of Stack)	

Gemäß der oben beschriebenen C Calling Convention haben wir das Beispiel der Konvertierung eines Strings in eine Zahl für die Verwendung eines Stacks umgeschrieben:

```
tmp      GREG 0
BOS      GREG #3000000000000000 //zeiger auf den "Boden" des Stacks (BOS - Bottom of Stack)
sp       GREG 0
bp       GREG 0
         LOC Data_Segment
         GREG @
strZahl  BYTE "31415",0
strError BYTE "konvertierung nicht erfolgreich",#A,0
```

```
strOk    BYTE "konvertierung hat funktioniert",#A,0
strZahlIst BYTE  "zahl ist 31415!",#A,0

        LOC #100
        GREG @
        PREFIX cSTN:
strAdr  IS $11
retOk   IS $12
retNr   IS $13
char    IS $14
counter IS $15
cmpR    IS $16
ascii0  IS '0'
ascii9  IS '9'
basis   IS 10
OK      IS 1
notOK   IS 2
        //cSTN
        //convertStringToNumber
        //konvertiert einen ASCII-String in eine Zahl
        //Parameter
        //sp    (bp+8)  zeigt auf Rücksprungadresse
        //sp+8  (bp+16) zeigt auf Adresse für des Hauptrückgabewert
        //sp+16 (bp+24) zeigt auf Adresse für die konvertierte Zahl
        //sp+24 (bp+32) zeigt auf Adresse des Strings

        //Rückgabewert
        //An der durch [bp+8] beschriebenen Speicherstelle befindet sich der
        //Hauptrückgabewert der Funktion:
        //     1 - erfolgreiche Ausführung
        //     2 - ein Fehler ist aufgetreten
        //In der Speicherstelle die bei Aufruf an Stelle [bp+16] übergeben worden war, befindet
        //sich, falls die Funktion erfolgreich war, die konvertierte Zahl.
cSTN    SUBU :sp,:sp,8
        STOU :bp,:sp   //Sichere basepointer von Hauptprogramm
        SET :bp,:sp    //basepointer=stackpointer

        SUBU :sp,:sp,40         //Sichere alle sonst benutzten Register auf dem Stack
        STO retOk,:sp,32
        STO retNr,:sp,24
        STO char,:sp,16
        STO counter,:sp,8
        STO cmpR,:sp

        SET retOk,OK //initialisiere Rückgabeparameter, Funktion grundsätzlich erfolgreich
        SET retNr,0 //Standardzahl ist 0
        SET char,0
        SET counter,0
        LDOU strAdr,:bp,32              //vor strAdr liegen 4 Parameter
cSTNloop  LDBU char,strAdr,counter      //lade Byte
          BZ char,cSTNexit              //überprüfe ob 0-Byte ist
          CMP cmpR,char,ascii0          //überprüfe ob es eine gültige Ziffer ist
          BN cmpR,cSTNerror
          CMP cmpR,char,ascii9
          BP cmpR,cSTNerror
          SUBU char,char,ascii0         //extrahiere Ziffer
          MULU retNr,retNr,basis        //hornerschema zur Interpretation der Zahl
          ADDU retNr,retNr,char
          ADDU counter,counter,1
          JMP cSTNloop                  //verarbeite nächstes Zeichen
cSTNerror SET retOk,notOK
cSTNexit  LDOU :tmp,:bp,16              //hole dir die Adresse des ersten Rückgabewerts
          STO retOk,:tmp                //speichere dort den Hauptrückgabewert
          LDOU :tmp,:bp,24              //hole dir die Adresse des zweiten Rückgabewerts
          STO retNr,:tmp                //speichere dort die eigentliche Zahl
          LDOU :tmp,:bp,8               //Sichere Rücksprungadresse in register

          LDO retOk,:sp,32              //stelle die oben gesicherten Register wieder her
          LDO retNr,:sp,24
          LDO char,:sp,16
          LDO counter,:sp,8
          LDO cmpR,:sp
          ADDU :sp,:sp,40

          LDOU :bp,:sp                  //stelle den Basepointer des Hauptprogramms wieder her
          ADDU :sp,:sp,8
          GO :tmp,:tmp,0                //springe gemäß der Rücksprungadresse zurück
```

```
        PREFIX :
cmpR    IS $0
strAdr  IS $1
retNr   IS $2
retOk   IS $3
retParamAdr1 IS $4
retParamAdr2 IS $5
retAdr  IS $6
Main    SET sp,BOS              //initialisiere Stackpointer
        SET bp,BOS              //sowie Basepointer
        SUBU sp,sp,16           //lege auf dem Stack 16 Byte für die Rückgabewerte der Funktion an
        SET retParamAdr1,sp     //speichere Adresse für ersten Rückgabewerte
        ADDU retParamAdr2,sp,8  //speichere Adresse für zweiten Rückgabewert (sp+8)
        LDA strAdr,strZahl      //lade die Adresse des Strings
        GETA retAdr,return      //lade die Rücksprungadresse

        SUBU sp,sp,32           //reserviere auf dem Stack Speicherplatz für 4 Parameter
        STOU strAdr,sp,24       //lege die Adresse des Strings auf dem Stack ab
        STOU retParamAdr2,sp,16 //lege die adresse für den zweiten Rückgabewert auf dem Stack ab
        STOU retParamAdr1,sp,8  //lege die adresse für den erst Rückgabewert auf dem Stack ab
        STOU retAdr,sp          //speichere die Rücksprungadresse auf dem Stack
        GO tmp,:cSTN:cSTN
return  LDO retOk,retParamAdr1  //nimm Ergebnis vom Stapel
        LDO retNr,retParamAdr2  //nimm Ergebnis vom Stapel
        ADDU sp,sp,32           //setze den Stackpointer richtig (auf den Bottom des Stacks)
        CMP cmpR,retOk,:cSTN:OK //überprüfe ob die Funktion erfolgreich ausgeführt wurde
        BP cmpR,convertError    //falls nicht, gib einen Fehler aus
        LDA $255,strOk
        JMP printStr
convertError LDA $255,strError
printStr TRAP 0,Fputs,StdOut    //Ausgabe
        SET cmpR,31415
        CMP cmpR,retNr,cmpR     //überprüfe ob die richtige Zahl erkannt wurde
        BNZ cmpR,exit          //falls nicht beende das Programm
        LDA $255,strZahlIst    //ansonsten gib aus, dass die richtige Zahl zurückgegeben wurde.
        TRAP 0,Fputs,StdOut
exit    TRAP 0,Halt,0
```

Weitere Informationen zur C Function Call Convention findet man unter anderem in den folgenden Quellen:

- ✓ Richard Chang, C Function Call Conventions and the Stack: http://www.cs.umbc.edu/~chang/cs313.s02/stack.shtml
- ✓ wikipedia, x86 calling conventions: http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl
- ✓ Nemanja Trifunovic, Calling Conventions Demystified: http://www.codeproject.com/KB/cpp/calling_conventions_demystified.aspx

Stack - Analyse des Codes

Werden Funktionsaufrufe mit Hilfe eines Stacks implementiert, so werden die meisten Nachteile der Implementierung mit GO beseitigt.

- ✓ Es können rekursive Funktionen geschrieben werden.
- ✓ Werden die Register, welche in dem Unterprogramm benutzt werden, richtig gesichert, so können keine Nebeneffekte mehr auftreten.

Der einzige wirkliche Nachteil bei der Verwendung eines speicherbasierten Stacks ist, dass alle Parameter und Rückgabewerte im Speicher abgelegt werden müssen. Dies benötigt im Vergleich zu dem Zugriff auf Register sehr viel Zeit. Desweiteren sind die vielen benötigten Operationen auf dem Stack für Beginner

unleserlich und sehr fehleranfällig. Um diese Nachteile zu umgehen, hat Knuth einen Registerstack in MMIX eingebaut.

UMSETZUNG MIT DEM MMIX-REGISTERSTACK

Bei einem Registerstack (bzw. register stack / registerstack) handelt es sich um einen Stack, der auf Registern basiert. Anstatt also Parameter und Rückgabewerte über einen Stack im Speicher zu übergeben, wird dies effizient über Register getan.

Für die Implementierung des Registerstacks werden Register in MMIX in folgende Kategorien unterteilt:

Register								
lokale Register			marginale Register			globale Register		
$0	...	$n	$n+1	...	$m-1	$m	...	$255
Register, welche derzeit in Verwendung sind			Register, welche derzeit nicht benutzt werden			Register die global (auch in Unterprogrammen) zur Verfügung stehen		

Bei lokalen Registern handelt es sich um diejenigen Register, die bisher benutzt wurden, d.h. in die ein Wert geschrieben worden ist. In dem speziellen Register rL wird die obere Grenze der lokalen Register gespeichert. Sofern Register $0 bis einschließlich Register $n lokal sind, wäre rL = n+1. Globale Register sind in der Regel nicht Teil des Registerstacks und werden über den Pseudobefehl GREG angelegt. Das spezielle Register rG enthält den Index des kleinsten globalen Registers. Ist rG = 250, so sind also die Register $250, $251, ..., $255 global.

FUNKTIONSAUFRUFE MIT DEM REGISTERSTACK

Das Ziel des Registerstacks ist es, folgende Logik zu ermöglichen:

1. Bevor ein Unterprogramm aufgerufen wird, werden die Parameter in den höchsten lokalen Registern abgelegt.
2. Bei dem Aufruf des Unterprogramms gibt man an, ab welchem lokalen Register die Argumente für die Funktion zu finden sind.
    a. Die aufgerufene Funktion hat einen vollkommen „neuen“ Satz mit Registern zur Verfügung. Dabei findet eine Umbenennung der Register statt, so dass in den untersten ($0,..) die Parameter stehen. Die Register der aufrufenden Funktion, d.h. die Register, welche in dem Hauptprogramm vor den Parametern standen, sind nicht mehr zugänglich. Dadurch ist es dem Unterprogramm nicht möglich, Werte des Hauptprogramms zu verändern.
    b. Die aufgerufene Funktion speichert ihre Rückgabewerte in den ersten Registern ($0,..) und springt zum Aufrufer zurück.
3. Die Rückgabewerte stehen an der Stelle, an der zuvor die Parameter übergeben worden sind.

Das folgende Bild verdeutlicht diesen Sachverhalt:

Register des Hauptprogramms											
lokale Register						marginale Register			globale Register		
$0	..	$k-1	$k	..	$n	$n+1	..	$m-1	$m	..	$255

Parameter für Unterprogramm

Register des Unterprogramms direkt nach dem Aufruf durch das Hauptprogramm									
kein Zugriff	lokale Register			marginale Register			globale Register		
	$0	..	$n-k	$n-k+1	..	$m-1	$m	..	$255

Parameter

Bei dem Aufruf des Unterprogramms findet somit eine Verschiebung der Register statt. Das erste Register, das als Parameter übergeben werden sollte, ist Register $0, der zweite Parameter liegt in Register $1 usw.

Beim Rücksprung zum Hauptprogramm wird diese Verschiebung wieder rückgängig gemacht:

Register des Unterprogramms vor dem Rücksprung									
kein Zugriff	lokale Register				marginale Register		globale Register		
	$0	..	$i	..	$j+1	..	$m	.	$255

Rückgabewerte

Register des Hauptprogramms nach dem Rücksprung											
lokale Register						marginale Register			globale Register		
$0	..	$k-1	..	$k+i	..	$n+i+1	..	$m-1	$m	..	$255

Rückgabewerte

Die Befehle zur Verwendung des Registerstacks sind in der folgenden Tabelle zusammengefasst.

PUSHJ X,Label	Die Register $0 bis einschließlich $X werden auf den Registerstack gepusht und sind nicht mehr zugänglich. In dem Register $X wird die Anzahl an gepushten Registern gespeichert. Der Inhalt von Register $X wird also verändert. Es wird zur durch Label bezeichneten Speicheradresse gesprungen und die Ausführung des Programms dort fortgeführt. Dabei wird eine relative Adressierung verwandt (J steht für Jump). Die Rücksprungadresse wird in dem speziellen Register rJ gespeichert.
PUSHGO X,Label	Äquivalent zu PUSHJ. Es wird jedoch eine absolute Adressierung verwandt.
POP X,Y	Es wird an die in rJ gespeicherte Adresse zurückgesprungen. Der Wert X gibt an, dass in den Registern $0 bis einschließlich $(X-1) die Rückgabewerte stehen. Der vorherige Registersatz (vor dem PUSH) wird wiederhergestellt und die Register $0 bis $(X-2) werden stehen in den Registern, in denen zuvor die Parameter standen. Das Register $(X-1) wird in das Register vor den Parametern geschrieben.

Die Funktionsweise von PUSH und POP wird im folgenden gemäß unserem Standard-Beispiel gezeigt:

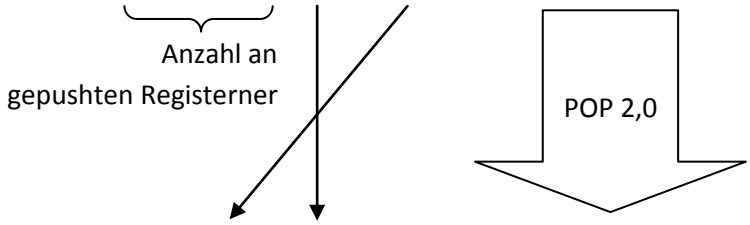
```
                LOC Data_Segment
                GREG @
strZahl         BYTE  "31415",0
strError        BYTE  "konvertierung nicht erfolgreich",#A,0
strOk           BYTE    "konvertierung hat funktioniert",#A,0
strZahlIst      BYTE        "zahl ist 31415!",#A,0

        LOC #100
        GREG @
        PREFIX cSTN:
retAdr  IS $0
strAdr  IS $1
```

```
retOk   IS $2
retNr   IS $3
char    IS $4
counter IS $5
cmpR    IS $6
ascii0  IS '0'
ascii9  IS '9'
basis   IS 10
OK      IS 1
notOK   IS 2
        //cSTN - Prozedur
        //convertStringToNumber
        //konvertiert einen ASCII-String in eine Zahl
        //Parameter
        //$0 Rücksprungadresse
        //$1 Adresse des Strings, welcher als Zahl interpretiert werden soll (mit 0 beendet)

        //Rückgabewerte nach Beendigung der Ausführung
        //$(0+X) - nullter Rückgabeparameter:
        //     1 - Funktion erfolgreich
        //     2 - Zahl konnte nicht erfolgreich gelesen werden
        //$(1+X) - erster Rückgabeparameter: der interpretierte Zahlenwert
cSTN    SET retOk,OK //initialisiere Rückgabeparameter, Funktion grundsätzlich erfolgreich
        SET retNr,0  //Standardzahl ist 0
        SET char,0
        SET counter,0
cSTNloop        LDBU char,strAdr,counter //lade Byte
                BZ char,cSTNexit     //überprüfe ob 0-Byte ist
                CMP cmpR,char,ascii0         //überprüfe ob es eine gültige Ziffer (>='0') ist
                BN cmpR,cSTNerror
                CMP cmpR,char,ascii9         //überprüfe ob es eine gültige Ziffer (<='9) ist
                BP cmpR,cSTNerror
                SUBU char,char,ascii0        //extrahiere Ziffer
                MULU retNr,retNr,basis       //hornerschema zur Interpretation der Zahl
                ADDU retNr,retNr,char
                ADDU counter,counter,1
                JMP cSTNloop                 //verarbeite nächstes Zeichen
cSTNerror SET retOk,notOK
cSTNexit  PUT :rJ,retAdr                 //schreibe die Rücksprungadresse nach rJ
          SET $0,retNr                   //zweiter Rückgabewert
          SET $1,retOk                   //Hauptrückgabewert
          POP 2,0                        //spring zurück


        PREFIX :
cmpR    IS $0
retOk   IS $3
retNr   IS $4
ArgReg  IS 19           //Gibt an ab welchem Register die Argumente folgen
arg0    IS $20            //nulltes Argument
arg1    IS $21            //erstes Argument
ret0    IS $19            //nullter Rückgabewert
ret1    IS $20            //erster Rückgabewert

%Start des Hauptprogramms
Main            LDA arg1,strZahl   //lade Adresse der Zahl (als AsciiCode) als erstes Argument
                GETA arg0,return        //lade die Rücksprungadresse
                PUSHGO ArgReg,:cSTN:cSTN  //rufe die Funktion auf
return          SET retOk,ret0            //hole die Ergebnisse aus den entsprechenden Registern
                SET retNr,ret1
                CMP cmpR,retOk,:cSTN:OK         //überprüfe, ob die Funktion erfolgreich ausgeführt
wurde
                BP cmpR,convertError    //falls nicht, gib einen Fehler aus
                LDA $255,strOk                  //ansonsten gib aus, dass alles Funktioniert hat
                JMP printStr
convertError    LDA $255,strError
printStr        TRAP 0,Fputs,StdOut     //Ausgabe..
                SET cmpR,31415
                CMP cmpR,retNr,cmpR     //Überprüfe, ob die richtige Zahl erkannt wurde
                BNZ cmpR,exit           //falls nicht, beende das Programm
                LDA $255,strZahlIst     //ansonsten gib aus, dass die richtige Zahl zurückgegeben
wurde
                TRAP 0,Fputs,StdOut
exit            TRAP 0,Halt,0
```

Besonderheit beim Aufruf von POP

Wie oben erwähnt wird der letzte Rückgabewert immer vor die Register, in denen die Parameter standen, geschrieben. Dies liegt daran dass in dem Register, dass bei dem Aufruf von PUSH X,Label in dem Register $X die Anzahl an gepushten Registern abgelegt wird. Um dies zu verstehen, betrachten wir den Registerstack vor und nach dem Rücksprung durch POP.

Registerstack vor dem Rücksprung mittels POP												
Register des Hauptprogramms (kein Zugriff)			lokale Register des Unterprogramms				marginale Register			globale Register		
$0	..	$19	$0	$1	..	$6	$7	..	$m-1	$m	..	$255



Registerstack nach dem Rücksprung mittels POP										
Lokale Register des Hauptprogramms					marginale Register			globale Register		
$0	..	$19	$20	$21	$22	..	$m-1	$m	..	$255

Rückgabewerte des Unterprogramms

Das Register $19, in das die Anzahl an gepushten Registern geschrieben worden ist, wird somit zusätzlich für Rückgabewerte benutzt, da man sich damit ein lokales Register für einen Rückgabewert sparen kann.

Registerstack - Analyse des Codes

Wie bei dem Stack, der auf dem Speicher basiert, werden mittels dem Registerstack Seiteneffekte bei der Ausführung von Unterprogrammen / Funktionen ausgeschlossen. Rekursive Aufrufe sind mittels der reinen Verwendung von PUSH und POP nicht vorgesehen, da die Rücksprungadresse immer in dem speziellen Register rJ gespeichert wird. Sofern man verschachtelte Aufrufe mittels dem Registerstack umsetzen möchte, so kann man jedoch die Rückgabeadresse einfach als zusätzlichen Parameter übergeben.

Weiterführende Informationen zum Registerstack

Knuth beschreibt in seinem *Fascile 1* in seiner Reihe *The Art of Computer Programming* ausführlich den Registerstack und die Umsetzung in MMIX. Desweiteren merkt Knuth an, dass das Konzept des Registerstacks auch in heutigen Prozessoren eingesetzt wird. Einen interessanten Artikel zur Umsetzung des Registerstacks in dem Intel Itanium Prozessor ist unter http://software.intel.com/en-us/articles/itaniumr-processor-family-performance-advantages-register-stack-architecture/ zu finden. In diesem Artikel werden die Vorteile der Verwendung eines Registerstacks im Vergleich zu der Verwendung eines speicherbasierten Stacks an Hand von Assemblercode verdeutlicht.

"Functions should do one thing. They should do it well. They should do it only."[1]

[1] Ein guter Ratschlag für alle Programmierer; entnommen aus Robert C. Martin, *Clean Code*, Prentice Hall