BCSC

VAX Procedure Calling Standard

Eine Einführung

B. Ulmann

15-OCT-1994

1. Allgemeines

Der VAX Procedure Calling Standard ist untrennbar verknüpft mit den Hardwaregegebenheiten der VAX Architektur[1].

Das Ziel der vorliegenden Kurzeinführung ist, die allgemeinen Aufrufmechanismen auf unterster Ebene darzustellen, um so auch dem Hochsprachenprogrammierer einen Einblick in die Hintergründe dieser Techniken zu geben.

Dieser Standard besitzt in den folgenden Fällen Gültigkeit:

- Interfaces von DIGITAL Standard-System-Software
- Intermodul-Aufrufe grösserer VMS-Komponenten
- Alle von Standard-DIGITAL-Compilern erzeugte externen Prozedur-Aufrufe

[2]

Der im folgenden beschriebene Standard umfaßt Mechanismen für folgende Formen der Parameterübergabe:

- Immediate Value [3]
- Reference
- Descriptor

[1] Selbstverständlich muß ein Großteil dieser Funktionalität auf der AlphaAXP-Architektur durch Emulation nachgebildet werden, da diese als reine RISC-Struktur keine derart ausgefeilten Konzepte von sich aus anzubieten in der Lage ist.

[2] Hingegen wird das in diesem Standard festgeschriebene Vorgehen nicht bei internen VMS-Calls eingehalten.

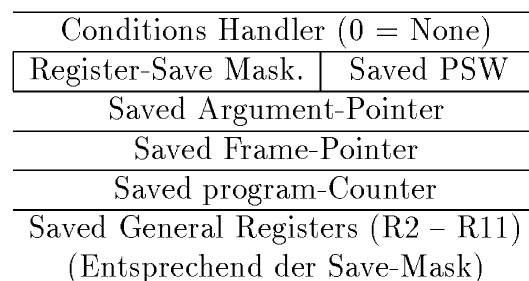
[3] Diese Übergabeform kann lediglich von MACRO bzw. BLISS aus verwendet werden.

2. Der Calling Mechanismus

Bei herkömmlichen CPU-Architekturen erwartet man als Parameter einer CALL -Instruktion zumeist die (virtuelle) Adresse der ersten ausführbaren Instruktion der betreffenden Routine. In dieser Hinsicht bildet die Architektur der VAX-CPU eine Ausnahme – hier zeigt dieser Parameter auf ein *entry-mask* genanntes Wort, das der eigentlichen Routine vorangestellt ist und als *register-save-mask* interpretiert wird. Aus ihm ist ersichtlich, welche Register von der aufzurufenden Routine verändert werden (können) und somit vor ihrer Ausführung gesichert bzw. beim Rücksprung restauriert werden müssen.

Diese Maske wird vom Mikrocode der `CALLG`-Instruktion [1] ausgewertet und führt zum automatischen Retten der betroffenen Register. Der nächste Schritt innerhalb eines Call-Statements besteht nun aus dem eigentlichen Sprung in den Code der Routine [2].

Die beiden VAX CALL-Statements `CALLS` bzw. `CALLG` [3] erzeugen bei ihrer Ausführung automatisch einen sogenannten *call frame* oder *stack frame* auf dem Stack, dessen allgemeiner Aufbau in Abbildung 2.1 dargestellt ist. Der aktuelle Framepointer



Conditions Handler (0 = None)	
Register-Save Mask.	Saved PSW
Saved Argument-Pointer	
Saved Frame-Pointer	
Saved program-Counter	
Saved General Registers (R2 – R11) (Entsprechend der Save-Mask)	

Abbildung 2.1: VAX Call-Frame

FP eines Call-Aufrufs zeigt stets auf das oberste Wort (*Condition-Handler*) des betreffenden *call frames*[4]. Das nächste Langwort dieser Struktur enthält die Kopie

[1] Call- Subroutine-with-General-Argument

[2] Wobei natürlich zusätzlich zu den in der *register-save-mask* angegebenen Registern der PC gesichert wird.

[3] Der Unterschied zwischen `CALLG` und `CALLS` ist darin zu sehen, daß letzteres die benötigte Argumentliste (siehe dort) auf dem Stack anlegt, `CALLG` hingegen in einem beliebigen Speicherbereich, was allerdings für die gerufene Routine insofern unerheblich ist, als sie die virtuelle Adresse der Argument-Liste stets im AP-Register übergeben bekommt.

[4] Zu Beginn einer Unterroutine entspricht er folglich wertmäßig dem Stackpointer **SP**.

der *register- save-mask* sowie des **PSW**. Die einzigen Register, die in den automatischen Sicherungsvorgang eines `CALLS`- bzw. `CALLG`- Statements einbezogen werden können, sind die Register **R2** bis **R11**[5].

Zu beachten ist lediglich noch, daß die Länge eines *call frames* nicht fix, sondern – je nach Anzahl zu rettender Register – Schwankungen unterworfen ist.

[5]Den Registern **R0** sowie **R1** kommt eine besondere Bedeutung zu, auf die im folgenden noch eingegangen wird.

3. Argumentübergabe

Alle Parameter, die einer aufzurufenden Routine übergeben werden sollen, werden mit Hilfe einer sogenannten *argument list* beschrieben und auf diese Art dem betreffenden Programmteil zur Verfügung gestellt.

Der allgemeine Aufbau einer solcher Liste ist aus Abbildung 3.1 leicht ersichtlich. Der oberste Wert enthält die Anzahl der Argumente, die sich im folgenden anschließen – Beachtung verdient hierbei die Tatsache, daß lediglich die untersten 8 Bit hierfür verwandt werden dürfen – die oberen 24 Bit müssen 0 sein! Alle in

Number of Arguments
arg_1
arg_2
...
arg_n

Abbildung 3.1: VAX Call-Frame

dieser Liste übergebenen Argumente sind sämtlich Langworte, die die folgenden drei Übergabetypen repräsentieren können:

- Immediate – hierbei stellt der Wert des Eintrags den Wert des Parameters selbst dar,
- By Reference – das betreffende Langwort enthält die Adresse des zu verwendenden Arguments,
- By Descriptor – [1] da ein Deskriptor selbst zu lang ist, um in nur einem Langwort Platz zu finden, wird in diesem Fall lediglich die Adresse des Descriptors übergeben.

Ein Deskriptor besteht stets aus einem Paar von Langworten, von denen das erste die Anzahl der Bytes enthält, die benötigt werden, um das eigentliche Objekt darzustellen, das zweite enthält die virtuelle Adresse, unter der das solcherart beschriebene Objekt abgelegt ist.

Innerhalb der aufgerufenen Routine wird über den sogenannten *argument pointer* **AP** auf die aktuelle Argument-Liste zugegriffen.

[1] Hierunter fallen auch String-Deskriptoren!

4. Register

Der VAX Procedure Calling Standard verwendet einige der VAX CPU-Register in genau festgeschriebener Form, so daß an dieser Stelle kurz auf die einzelnen Register und ihre Funktion eingegangen werden soll:

- **PC** – Programmzähler
- **SP** – Stackpointer
- **FP** – Framepointer – er zeigt auf den aktuellen *call frame*
- **AP** – Argumentpointer – Zeiger auf die aktuelle *argument list*
- **R1** – Dieses Register *kann* einen Environment-Wert enthalten, falls die gerufene Routine diesen erwartet – in allen anderen Fällen muß an dieser Stelle selbstverständlich nichts übergeben werden
- **R0, R1** – Funktionswert-Rückgabe-Register – hier sind folgende Fälle zu unterscheiden:
  1. Die Routine liefert einen bis zu 32 Bit langen Ergebniswert zurück – in diesem Fall wird lediglich **R0** für den Rückgabewert verwendet.
  2. Die Routine liefert einen Wert mit mehr als 32, jedoch weniger als 65 Bit zurück – dies ist der einzige Fall, in dem beide Register Anwendung finden.
  3. Das Ergebnis der Routine ist zu umfangreich, um entsprechend einer der beiden vorangegangenen Methoden behandelt zu werden. In diesem Fall ändert sich der Funktionsaufruf an sich dahingehend, daß der erste Argumenteintrag der *argument list* einen Verweis auf den zu erwartenden Funktionswert darstellt. Die folgenden Listeneinträge rücken entsprechend um eine Position nach unten.

5. Beispiele

5.1 Allgemeines

Der einfachste Aufruf einer Unterroutine ist sicherlich der ohne Argumente:

```
        CALLS   #0, procedure
```

Fast analog hierzu gestaltet sich nun ein etwas realistischerer Funktionsaufruf. Hierbei muß lediglich dafür Sorge getragen werden, daß alle erforderlichen Argument bereits auf dem Stack abgelegt sind, um für die eigentliche Routine zur Verfügung zu stehen:

```
        PUSH    arg1
        PUSH    arg2
        ...
        PUSH    argn
        CALLS   #n, procedure
```

5.2 Konkretes

Eine kürzeres FORTRAN-Beispiel soll zur Verdeutlichung des Aufbaus der Argument-Listen im folgenden gegeben werden:

```
        CALL SUB (x, %DESCR (x))
```

führt zu folgender Argument-Liste:

```
ARGLIST:.LONG   2    ; ANZAHL ARGUMENTE
        .ADDR   X    ; ADRESSE VON X, UEBERGABE PER REF
        .ADDR   L$1  ; ADRESSE DES DESCRIPTORS VON X

L$1:    .WORD   4    ; LAENGE
        .BYTE   10   ; TYP-CODE
        .BYTE   1    ; KLASSEN-CODE
        .ADDR   X    ; ADRESSE DES BESCHRIEBENEN OBJEKTS
```

Die Längenangabe im ersten Descriptor-Wort resultiert aus der Tatsache, daß das Objekt `x` vom Typ `REAL` ist und folglich 4 Byte belegt.

Das nächste Beispiel dient der Demonstration des Call-by-Reference- Übergabemechanismus in Verbindung mit FORTRAN sowie C:

```
      PROGRAM BCSC
      INTEGER I, INC
      WRITE (*, '(X, A, $)') 'WERT : '
      READ (*, *) I
      CALL INC (I)
      WRITE (*, *) I
      END

void inc (i)
int *i;
{   *i = *i + 1;
}
```

Da FORTRAN bei den Datentypen `REAL`, `INTEGER` etc. als Übergabemechanismus standardmäßig Call-by-Reference anwendet, ist in obigem Beispiel auf FORTRAN Seite nichts zu beachten. Anders liegt der Fall, wenn einer Routine ein Wert explizit durch Call-by-Value übergeben werden soll:

```
      PROGRAM BCSC
      INTEGER I, J, INC2
      WRITE (*, '(X, A, $)') 'WERT : '
      READ (*, *) I
      CALL INC2(%VAL (I), J)
      WRITE (*, *) J
      END

void inc2 (i, j)
int i, *j;
{   *j = i + 1;
}
```

Vergleichsweise kompliziert gestaltet sich ein solcher Unterprogrammaufruf, sobald Werte weitergereicht werden sollen, die durch einen Descriptor beschrieben werden, wie anhand des folgenden Beispiels deutlich werden dürfte, das den Aufruf einer System-Routine innerhalb eines C-Programms demonstriert, die ihren Parameter zwingend per Descriptor verlangt[1].

```
#include <ssdef.h>
#include <stdio.h>
#include <descrip.h>
int SYS$SETPRN ();
int main (void)
```

[1]FORTRAN stellt für die verschiedenen Übergabemodi die Möglichkeit eines `%VAL ( )`, `%REF ( )` bzw. `%DESCR ( )` zur Verfügung.

```
{   int ret;
        /* Struktur fuer Descriptor */
    struct dsc$descriptor_s name_desc;
    char *name = "GNLPF";
        /* Laenge setzen           */
    name_desc.dsc$w_length = strlen (name);
        /* Pointer auf Objekt      */
    name_desc.dsc$a_pointer = name;
        /* String Descriptor-Klasse */
    name_desc.dsc$b_class = DSC$K_CLASS_S;
        /* Daten-Type ASCII-String  */
    name_desc.dsc$b_dtype = DSC$K_DTYPE_T;
    ret = SYS$SETPRN (&name_desc);
    if (ret != SS$_NORMAL)
      fprintf (stderr,"Prozessname konnte nicht gesetzt werden !\n");
    exit (ret);
}
```

Wie man leicht sieht, gestaltet sich das explizite Erzeugen des gewünschten Descriptors eher umständlich, so daß der Wunsch nach einer einfacheren Lösung (wie sie beispielsweise in FORTRAN in Form von `%DESCR ( )` verfügbar ist) aufkeimt. Eine solche Möglichkeit bietet auch C, wie das folgende Beispiel zeigen wird, das im übrigen dieselbe Funktion wie das vorhergehende ausführt:

```
#include <descrip.h>
int SYS$SETPRN ();
main (void)
{       static $DESCRIPTOR (name_desc, "GNLPF");
        return SYS$SETPRN (&name_desc);
}
```

6. RET, JSB, RSB

Das Ende einer Unterroutine ist erreicht, wenn die Intruktion `RET` angetroffen wird – sie hat zur Folge, daß die CPU wieder auf den Status zurückgesetzt wird, den sie vor Eintritt in die Routine innehatte. Zusätzlich zu den Registern **AP**, **SP** und **FP** werden auch alle in der *register mask* angeführten Register auf ihren ursprünglichen Wert gesetzt[1].

Eine besondere Form des Unterprogrammaufrufs bzw. -Rücksprungs stellt das Instruktionspaar `JSB` bzw. `RSB` dar. Sie sind hier lediglich der Vollständigkeit halber erwähnt, da sie sich in keiner Weise an den hier beschriebenen Standard halten. Sie dienen beispielsweise Unterprogrammaufrufen innerhalb von VMS-Routinen.

`JSB` führt lediglich einen Sprung zu einer Routine aus, wobei nur der Programmzähler auf den Stack gesichert wird, `RSB` vollzieht den gegenteiligen Vorgang – der Programmzähler wird vom Stack restauriert, und entsprechend inkrementiert, so daß mit der Ausführung des rufenden Programmteils fortgefahren werden kann.

Parameter werden bei dieser Form des Aufrufs in den meisten Fällen in Registern übergeben – die Verantwortlichkeit für das Retten eventuell veränderter Registerinhalte obliegt der aufgerufenen Unterroutine.

[1] Das Restaurieren des Stackpointers **SP** entspricht dem Entfernen des *call frames* vom Stack – ebenso werden alle Werte, die im Verlauf der Routine auf ihm abgelegt wurden, entfernt.

Literaturverzeichnis

[1] Introduction to VMS System Routines

[2] INSIDE VMS: The System Manager's and System Programmer's Guide to VMS Internals