Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-26
Jänner 1992

Instrumentierung auf Sourcecode-Level

Thomas Hontsch, Ulrich Schmid

Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

Projekt VTA

Instrumentierung auf Sourcecode–Level

Thomas Hontsch, Ulrich Schmid

Description Version_0.0 29 Jan 92

FWF Projekt Nr. P8390–PHY

Inhaltsverzeichnis

1 Einleitung

Ziel der Instrumentierung ist es, die im Kontext der Targetsoftware auftretenden simple Events – auch Statement–Events genannt – dem Monitor zu melden; das Auftreten eines Statement–Events ist ja bekanntlich immer mit der Exekution eines bestimmten Sourcecode–Statements in der Target–Software verbunden. Der Monitor kann daraufhin alle einem Event zugeordeten Aktionen ausführen. Diese Aktionen betreffen meist ein Einholen von speziellen Informationen über den Systemstatus der Target–Software. So können dem Event etwa Werte von Sourcecode–Variablen zugeordet werden.

Das folgende Kapitel beschäftigt sich mit der Lösung der eben genannten Aufgabenstellung.

Die in diesem Kapitel entwickelten Methoden zur Instrumentierung sind also so konzipiert, daß die geforderte Unabhängigkeit vom konkret eingesetzten Target–Prozessor erreicht wird. Alle prozessorabhängigen Module sind im ProSP, alle sprachabhängigen Funktionen im LaSP zusammengefaßt. Dabei ergibt sich folgende Struktur: Das LaSP verwendet (geringfügig) Funktionen aus dem ProSP, aber nicht umgekehrt. In Bild 1 ist dies dargestellt, indem das LaSP (teilweise) auf dem ProSP aufsetzt. Als Input für das LaSP dient das Targetcodefile; der Output des ProSP ist der Code für die Instrumentierung eines Events. Dabei werden (durch Funktionen, die auf dem LaSP und ProSP aufsetzen) die Informationen, die im LaSP gewonnen werden, im ProSP zur Generierung der Instrumentierung herangezogen. Der durch die strichlierte Linie in Bild 1 dargestellte Informationsfluß soll diesen Umstand verdeutlichen.

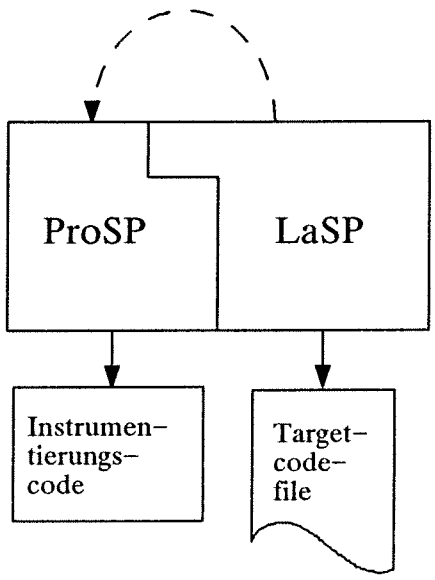

Bild 1: ProSP und LaSP

Alle Erklärungen und Beispiele in diesem Kapitel beziehen sich aber immer auf den 68000 von Motorola und den MCC68K C–Compiler von Microtec, für die das ProSP und LaSP entwickelt wurden.

2 Begriffe

Für das Verständnis der nächsten Abschnitte sind einige grundlegende Begriffe wichtig:

Event Handling Routine, Event Handler ist jene Funktion, die bei der Occurence eines Events abgearbeitet wird. Sie löst alle Aktionen aus, die die Target–CPU zur Event Recognition beitragen muß.

Statement Adresse ist eine Adresse im Targetmemory, auf die ein Event gesetzt werden soll.

Patched Code: Soll auf eine Statement Adresse im Targetcode ein Event gesetzt werden, so muß an dieser Stelle ein Sprung in die Event Handling Routine generiert werden. Patched Code ist nun jener Maschinenbefehl, der an dieser Stelle eigentlich steht. Er muß dort entfernt und von der Event Handling Routine an anderer Stelle exekutiert werden.

Patch Code ist der Maschinenbefehl, der auf der Statement Adresse "eingebaut" wird, um ein Event auftreten zu lassen. Dazu geeignet sind etwa ein Sprungbefehl oder eine Exception.

Event Code wird der Code der Event Handling Routine genannt.

Targetcodefile ist ein binäres File, dessen Code in das Targetmemory geladen werden kann. Es enthält neben dem Code auch noch die Debuginformationen der Software, die für die Instrumentierung wichtig sind.

Event Patcher ist das Programm, das ein Event zur Laufzeit der Target–Software im Memory installiert.

Event Mapping Syntax ist die Syntax für jene Ausdrücke, die beim Setzen eines Events die formalen Parameter der EDL–Definition an konkrete Variablen aus dem Target bindet.

3 ProSP – Processor Support Package: Die Erzeugung von Instrumentierungs–Code

Das ProSP stellt jene Mechanismen zur Verfügung, die imstande sind, für ein Statement–Event die notwendigen Maschinenbefehle für die Instrumentierung zu generieren. Dabei erhebt sich zuerst einmal die Frage, wie eine Instrumentierungs–Routine in der Target–Software "eingehängt" werden kann. Danach muß geklärt werden, welche Aktionen in der Instrumentierungs–Routine nun gebraucht werden, um all jene Informationen zu extrahieren, die das VTA–Target zur weiteren Bearbeitung des Events benötigt. Zuletzt muß auch sichergestellt werden, daß nach Verlassen einer Instrumentierungs–Routine durch die Target–CPU exakt jener Systemstatus wiederhergestellt wird, der vor der Abarbeitung der Instrumentierung vorgefunden wurde.

In den nächsten Abschnitten werden nun all diese Fragen beantwortet und das ProSP für den 68000 Prozessor entwickelt.

3.1 Prinzip der Instrumentierung

Erreicht die Targetsoftware während der Abarbeitung eine Programmstelle, an der ein Statement–Event gemeldet werden soll, so muß an dieser Stelle die Kontrolle kurz an den Monitor beziehungsweise an ein zum Monitor gehörendes Programmstück übergeben werden. Nur so hat der Monitor die Gelegenheit, das Event zu identifizieren, benötigte Informationen aus dem Systemstatus zu gewinnen und alle zum Event gehörenden Aktionen anzustoßen.

Die erste Forderung an einen guten Monitor ist, das Targetsystem so wenig wie möglich zu beeinflussen. Nachdem es sich bei dem nun vorgestellten Ansatz um Software–Instrumentierung handelt, gilt das Augenmerk bei der Entwicklung vorrangig der Minimierung dieses Problems.

Jedes Targetsystem erhält eine eigene VTA–Target Hardware. Diese besitzt grob umrissen einen Prozessor und ein Shared Memory. Das Shared Memory dient dabei als Interface zwischen dem VTA–Target und dem Target.

Der Ansatz zur Minimierung der Intrusiveness liegt nun in der Teilung der Aufgaben bei der Occurence und Recognition eines Statement–Events. Die Target–CPU hat dabei nur die Aufgabe, dem VTA–Target alle Informationen in einem Zwischenpuffer im Shared Memory zur Verfügung zu stellen. Das VTA–Target verwaltet diesen Puffer, holt sich die Informationen ab, wertet sie aus und generiert so letztlich das Statement–Event.

Die nächsten beiden Bilder verdeutlichen das Prinzip. Bild 2 zeigt die Targetsoftware, bevor ein Event gepatched wurde.

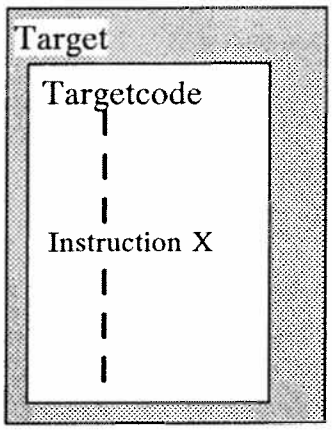

Bild 2: Targetcode vor Patch

Nun soll anstelle der Instruktion X ein Event aufgesetzt werden. Die sich daraus ergebende Situation erkennt man in Bild 3. An die Stelle der Instruktion X wurde der Patch gesetzt. Dieser übergibt die Kontrolle an den Event Handler, der ebenfalls von der Target–CPU ausgeführt wird. Er schreibt alle benötigten Informationen in den Shared–Memory–Buffer. Diese Informationen beinhalten eine eindeutige Eventnummer und die Werte der Target–Variablen. Die VTA–Target–CPU erkennt, daß ein neuer Event–Buffer beschrieben wurde und liest denselben aus, um die Informationen zu bearbeiten und daraus ein Event zu generieren.

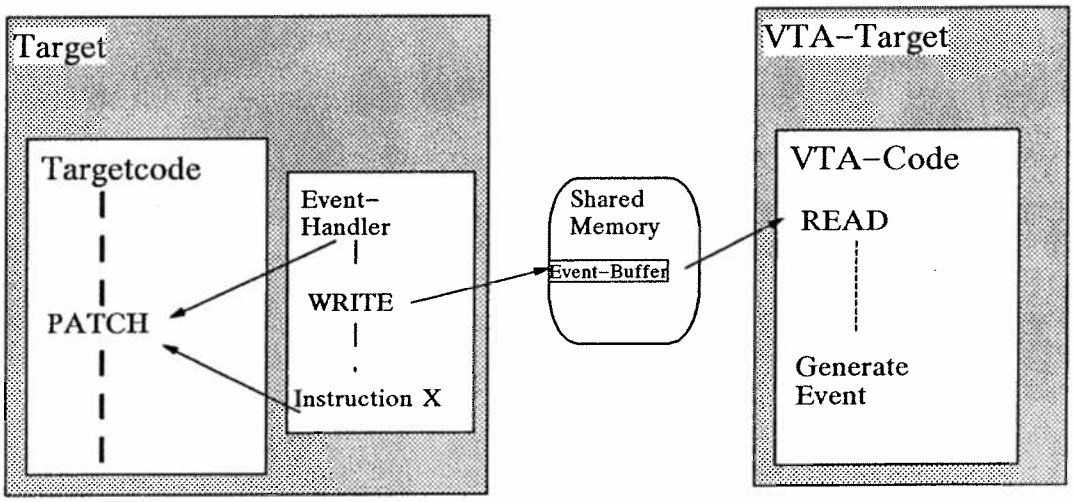

Bild 3: Targetcode nach dem Patch

3.1.1 Patching

Unter Patching im engeren Sinn wird hier jener Mechanismus verstanden, der die Kontrolle an den Eventcode übergibt. Der Programmfluß der Targetsoftware muß also geeignet angehalten und die Kontrolle an den Event Handler übergeben

werden.

Folgende Möglichkeiten der Kontrollflußunterbrechung beziehungsweise Kontrollflußübergabe stehen auf Assemblerebene zur Verfügung:

- o JSR: Jump Subroutine
- o JMP: Jump
- o BRA: Branch
- o TRAP: Excepti

Von der Targetsoftware aus gesehen stellt sich die Frage, welche Kontrollflußunterbrechungen unproblematisch einzubauen sind. Unproblematisch in dem Sinn, daß die Funktionalität durch die Abarbeitung eines Events nicht gefährdet wird. Dies richtet sich in erster Linie nach der Länge des betroffenen Maschinenbefehls (Patched Code). Jeder Maschinenbefehl hat eine bestimmte Länge. Diese reicht von 2 Bytes bis maximal 10 Bytes. Die an den Patchmechanismus zu stellende Forderung lautet:

Die Instruktion für den Patch darf nie größer sein als die betroffene Instruktion der Target–Software.

Wieso diese Forderung gestellt werden muß, zeigt das nächste Beispiel.

Angenommen, der Compiler für die Target–Software übersetzt eine Funktion in folgenden Assemblercode (Fragment):

```
                    :
0000004C 6020       BRA L3
0000004E 5382 L2:   SUBQ.L #1,D2
00000050 4A82       TST.L D2
                    :
00000074 6ED8       BGT L2
                    :
```

Will man nun ein Event auf der Adresse 0000004C patchen, so erkennt man, daß sich dort ein 2–Byte Befehl befindet. Setzt man nun dort als Patch einen Jump auf eine Long–Adresse ein, so ist dieser Jump 6 Bytes lang und überlagert somit auch noch die beiden nächsten Befehle. Nun könnte man meinen, daß es möglich sei, auch diese beiden Befehle an ihrer Originaladresse zu entfernen und, wie auch den ersten Befehl, an anderer Stelle (im Event Code) zu exekutieren. Nur hat man dabei übersehen, daß auf den zweiten Befehl ein Sprungziel angesetzt ist (L2). Würde man einen Jump einsetzen, so wäre an den Adressen 0000004E und 00000050 gerade die Sprungadresse des Jumps. Irgendwo im Targetprogramm gibt es aber sicherlich einen Sprung auf L2, der als Sprungziel plötzlich keinen exekutierbaren Befehl (und schon gar keinen, den die Targetsoftware erwarten würde) vorfindet.

Um auf jede Instruktion einen Patch setzen zu können, wird für 2 Byte Befehle demnach ein Befehl als Patch Code benötigt, der ebenfalls nur 2 Bytes lang ist. Hierfür bietet sich der TRAP an. Findet man einen Befehl vor, der mindestens 6 Bytes lang ist, so kann man natürlich problemlos einen Jump einbauen. Wie man bei der Verwendung des Jump–Befehls noch sehen wird, gibt es noch die Möglich-

keit des Branch, falls man einen 4 Byte Befehl erwischt. Voraussetzung dafür ist aber, daß das Sprungziel (der Event Code) im Offsetbereich des Branches liegt.

3.1.2 Patching mit einem JUMP

Wann immer man an einer Adresse im Target einen Befehl vorfindet, der 6 Bytes oder länger ist, ist als Patchcode ein Jump zulässig, mit dem an eine beliebige Adresse im gesamten Adressbereich des Prozessors gesprungen werden kann.

Die Verwendung eines Jumps hat mehrere Vorteile. Erstens ist der Jump wesentlich schneller als der Trap und zweitens wird der Stack nicht verändert.

3.1.3 Patching mit einem TRAP

Hat man als Patched Code nur einen 2 Byte Befehl, so ist man auf die Verwendung eines Traps angewiesen. Der 68000 hat eine Vielzahl an Möglichkeiten für das Exception–Handling anzubieten.

Für das Patching wird ein sehr selten verwendetes Feature des Prozessors ausgenutzt. Der 68000 hat drei Trapvektoren für alle nicht im Befehlssatz enthaltenen Hex–Codes.

1. unimplemented Instruction mit den Hex–Codes A000 – AFFF. Diese 4096 frei definierbaren Befehle sind für Emulationen oder eigene Erweiterungen des Befehlssatzes vorgesehen.

2. unimplemented Instruction mit den Hex–Codes F000 – FFFF. Diese werden für den Fließkomma–Koprozessor verwendet.

3. illegal Instruction Code: Alle Instruktionen, deren Code nicht identifiziert werden kann und nicht einem der beiden oben genannten Codes entspricht, erzwingen einen Trap mit diesem Vektor.

Die erste dieser drei Möglichkeiten bietet sich aus zwei Gründen ideal an:

1. Man hat die Möglichkeit, einen Trap zu generieren, ohne einen der Standard–Traps verwenden zu müssen. Diese werden nähmlich sehr oft von der Targetsoftware selbst gebraucht.

2. Man kann auf leichte Art und Weise dem Trap eine Eventnummer mitgeben. Von den 16 Bits des Instructionswords sind nur die vordersten 4 Bits mit hexadezimal 'A' fix; die restlichen 12 Bits können frei durchpermutiert werden.

Das zweitere Argument ist von besonderer Wichtigkeit. Im Gegensatz zu einem Jump, bei dem jedes Event auf einen anderen Eventcode springt, wodurch die Eventnummer fix kodiert werden kann, wird bei einem Trap immer die gleiche Adresse angesprungen. Der Traphandler (in unserem Fall also der "Eventhandler") muß demnach erst die Eventnummer identifizieren, um den eigenlichen Event Code

finden zu können. Dies wäre zwar auch über den am Supervisory Stack geretteten Programm Counter möglich, allerdings würde dies eine Hashtabelle oder ähnliches erfordern. Die Identifizierung der Eventnummer wäre also gegenüber der Eventbehandlung selbst überproportional zeitaufwendig. Bei der Verwendung der unimplemented Instructions bekommt man die Eventnummer sozusagen mitgeliefert.

Verwendet man einen Trap als Patchcode, so muß man sich auch darüber im klaren sein, wie der Prozessor diesen behandelt. Exekutiert ein Prozessor einen Trap–Befehl (in unserem Fall eine unimplemented Instruction), so werden im wesentlichen folgende Schritte durchgeführt.

- o Das S–bit im Statusregister wird gesetzt und bringt den Prozessor damit in den Supervisory Mode.
- o Der Program Counter wird auf den Supervisory Stack gepushed.
- o Das Statusregister wird auf den Supervisory Stack gepushed (mit dem ursprünglichen Wert des S–Bits).
- o Der neue Inhalt des Program Counters wird aus der Interrupt Vector Table geholt.
- o Die Programmexekution wird an der neuen Programmstelle fortgesetzt.

Springt man mit einem Jump direkt in die zugehörende Event Handling Routine, so kommt man mit einem Trap erst in die "Trap Handling Routine" und muß dort anhand der Eventnummer über eine Jumptable in die eigentliche Event Handling Routine für dieses Event springen. Der hierzu erforderliche Assemblercode* sieht folgendermaßen aus:

```
trap: MOVE.L A1,-(SP)           8 ; verwendetes Reg. retten
      MOVE.L 6(SP),A1          12 ; alten PC holen
      MOVE.W (A1),A1            8 ; unimpl.-Instr. nach A1
      SUB.W #A000,A1            8 ; A ausmaskieren
      MOVE.L tab(PC,A1.W),A1   14 ; Sprung berechnen
      JMP (A1)                  8 ; Sprung in Event Handler

tab:  DC.L event_0                ; Tabelle aller Event Handler
      DC.L event_1                ; Einsprungadressen
       :
      DC.L event_1023
                               ---
                                58
```

Dieser Code holt sich das Instructionword (also die unimplemented Instruction "Axxx") über den am Stack liegenden (alten) Program Counter nach A1 und sucht sich anhand der im Instructionword kodierten Eventnummer die Einsprungadresse des zugehörigen Event Handlers aus der Jump–Tabelle. Anhand dieser springt er dann in den Event Handler, wo das Event weiter bearbeitet wird. Der Rücksprung aus dem Trap (der Prozessor befindet sich auch noch nach dem Sprung in den

*Die Notation der Assemblerbefehle entspricht jener des Assemlers des Target–Entwicklungssystems (ASM68K).

Event Handler im "Trap") erfolgt aus Optimierungsgründen in jeder Event Handling Routine.

Bemerkung: Die Spalte mit den klein geschriebenen Zahlen gibt die Dauer des jeweiligen Befehls in Taktzyklen an. Die gesamte Dauer der Trap Handling Routine beträgt 58 Taktzyklen. Zuzüglich des Trap–Befehls selbst, der 34 Taktzyklen lang ist, ergibt sich eine Dauer von 92 Zyklen. Gegenüber einem einfachen Jump, der nur 14 Taktzyklen beansprucht, um die Event Handling Routine zu erreichen, immerhin der 7–fache Aufwand. Beim Setzen eines Events sollte also nach Möglichkeit ein Jump dem Trap vorgezogen werden.

Um die Adresse des Event Handlers korrekt aus der Jump–Tabelle zu lesen, muß im Instructionword die Eventnummer*4 stehen. Also ergibt sich etwa für das Event mit der Nummer 3 der Patch Code "A00C". Da somit nur noch ein Viertel der möglichen unimplemented Instructions verwendet werden können, sind 1024 Trap–Events möglich. Werden vom VTA mehr als 1024 Events benötigt, besteht die Möglichkeit, für Trap–Events und für Jump–Events zwei getrennte Nummernkreise zu verwalten. Der Nummernkreis für die Trap–Events ist dabei auf 0 bis 1024 eingeschränkt. Für Jump–Events können alle Nummern >= 1024 verwendet werden.

3.2 Event Handling Routine

An dieser Stelle konnte der Patch für ein Event erfolgreich im Targetcode gesetzt werden. Die Event Handling Routine hat die Kontrolle übernommen und muß nun alle notwendigen Informationen dem VTA–Target zur Weiterverarbeitung liefern. Dazu sind einige Aktionen durchzuführen, die in den nächsten Abschnitten erarbeitet werden.

3.2.1 Retten der Condition Codes

Da die meisten Maschinenbefehle (leider auch die meisten Moves) die Condition Codes verändern, ist es notwendig, vor der Abarbeitung des Event Codes das Statusregister zu retten und als letzte Aktion vor dem Ausführen des Patched Codes wieder zu restaurieren. Bei einem Trap–Event wird das Statusregister automatisch am Stack gerettet; bei einem Jump–Event muß dies durch einen eigenen Befehl erfolgen. Die hierzu notwendigen Befehle sind:

```
MOVE SR, -(SP)    ; Statusregister retten
MOVE (SP)+, CCR   ; Condition Codes wiederherstellen
```

3.2.2 Retten und Wiederherstellen verwendeter Register

Da der Inhalt der Register durch die Event–Behandlung nicht geändert werden darf, müssen alle bei der Abarbeitung der Event Handling Routine verwendeten Register gerettet werden. Insbesonders betrifft dies jene Register, die zum Kopieren der Sourcecode–Variablen in den Eventbuffer benötigt werden.

Der 68000 Prozessor hat zum Retten und Wiederherstellen von Registern eigene Befehle:

```
MOVEM.L reg_list, -(SP)   ; Reg. retten
MOVEM.L (SP)+, reg_list   ; Reg. wiederherstellen
```

Wobei "reg_list" eine Bitmaske ist.

Bei der Generierung des hierfür erforderlichen Codes muß berücksichtigt werden, ob der Patch durch einen Jump oder einen Trap erfolgte. Wird ein Trap verwendet, so ist zu beachten, daß bereits in der Trap Handling Routine das Register A1 gerettet wurde.

Beim Wiederherstellen der Register muß wiederum darauf geachtet werden, daß im Falle eines Traps das Register A1 ebenfalls restauriert werden muß. Es ist allerdings nicht immer möglich, dieses Register mit weiteren geretteten Registern zusammen wiederherzustellen, da dazu die Reihenfolge der am Stack geretteten Register richtig sein muß.

Ein Beispiel, bei dem ein Event durch einen Trap ausgelöst wird, verdeutlicht diesen Fall: In der Event Handling Routine wird zusätzlich zum Register A1 aus dem Trap Handler noch das Register A3 und D4 verwendet. Nach dem Retten dieser Register liegen sie nun in folgender Reihenfolge am Stack: A1, A3, D4. Damit diese 3 Register mit einem einzigen Befehl wiederhergestellt werden könnten, müßten sie in der Reihenfolge A3, A1, D4 am Stack liegen. Es müssen in diesem Fall daher leider 2 Befehle verwendet werden.

3.2.3 Anfordern des Event Buffers

Die Bereitstellung des Event Buffers ist Aufgabe des VTA–Targets. Die Verwaltung derselben obliegt ebenfalls dem VTA–Target und kann in diesem Stadium der Entwicklung noch nicht definiert werden. Für die Testumgebung der Instrumentierung ist ein eigener Algorithmus vorgesehen, der im Kapitel 7 vorgestellt wird.

Die Event Buffer Adresse des Event Buffers wird bei der Codegenerierung immer im Event Buffer Adressregister A1 erwartet. Bei der Bufferanforderung muß die vom VTA–Target bereitzustellende Adresse also in das Regsiter A1 kopiert werden. Bei jedem MOVE in den Event Buffer wird diese Adresse automatisch inkrementiert.

3.2.4 Kopieren der Sourcecode–Variablen in den Event Buffer

Um den Wert von Sourcecode–Variablen zu erhalten und in den Eventbuffer kopieren zu können, muß natürlich die Adresse der Sourcecode–Variablen bekannt sein. Konkret muß, abhängig von den zu kopierenden Sourcecode–Variablen, (irgendwann einmal) Code für den Zugriff darauf generiert worden sein. Diese Codegenerierung kann recht kompliziert sein; man denke dabei etwa an Struct– oder Array–Variablen sowie an Pointer und Indizierungen.

Um zu verhindern, daß andere Prozesse oder Exception Handling Routinen der Target–Software das Kopieren der Sourcecode–Variablen in den Event Buffer unterbrechen können (etwa durch Scheduling oder Interrupts), müssen während des

Kopiervorganges alle Interrupts gesperrt werden. Andernfalls kann nicht garantiert werden, daß die aus dem Status des Target–Systems extrahierten Informationen konsistent* sind! Selbstverständlich ist diese Maßnahme nur nötig, wenn zum Kopieren mehr als ein MOVE–Befehl notwendig ist; wird also nur eine Variable kopiert, die maximal ein Longword lang ist, so kann das Sperren der Interrupts unterbleiben.

Bei der Ausführung eines derartigen Variablenzugriffs innerhalb des Event Handlers sind einige (unerwünschte) Seiteneffekte beobachtbar, die berücksichtigt werden müssen. Dies sind:

- **Lesen eines Hardware–Registers:**

  Sourcecode–Variablen können auch direkt auf Register von Hardwarekomponenten der Target–Peripherie verweisen. Oft sind aber an das Lesen solcher Register implizit Nebeneffekte gebunden. So gibt es bespielsweise Bit–I/O–Karten, die so gestaltet sind, daß ein Lesen eines Inputregisters das neuerliche Abhorchen der Peripheriesignale startet. Wird ein Signalwechsel erkannt, wird dieser Abhorchmechanismus gestoppt und erst durch erneutes Lesen des zugehörenden Inputregisters wieder gestartet.

  Liest ein Event Code dieses Inputregister, so gehen der Target–Software unter Umständen wichtige Informationen verloren und stören so den Ablauf des Targetsystems.

  Sollen solche Variablen im Event Code gelesen werden, so muß sich der Anwender über die Seiteneffekte im klaren sein.

- **Lesen von Variablen in kritischen Abschnitten:**

  Target–Variablen, auf die mehrere Prozesse zugreifen, müssen bekanntlich durch Mutual Exclusion geschützt werden. Liest der Event Code eines Events, das außerhalb des kritischen Abschnitts gesetzt ist, Werte von geschützten Variablen, so muß sich der Anwender darüber im klaren sein, daß an dieser Programmstelle andere Prozesse auf die geschützten Variablen zugreifen und diese verändern können und das Event daher die Variablen in einem inkonsistenten Zustand lesen könnte.

3.2.5 Ausführen des Patched Codes und Exit

Wie man gleich sehen wird, ist die Ausführung des Patched Codes und der Rücksprung aus dem Event Handler zurück in den Targetcode nicht trivial.

Bevor der von seiner Statement Adresse entfernte Maschinenbefehl exekutiert werden kann, muß für Trap–Events und Jump–Events die gleiche Ausgangssituation geschaffen werden. Befindet man sich nach einen Jump im gleichen Mode des Prozessors wie vor dem Sprung, so gelangt man nach einem Trap auf jeden Fall in den Supervisory Mode. Daraus folgt, daß vor Exekution des Patched Codes der

*DMA kann auch durch Interrupt Disable nicht verhindert werden; dies ist aber kein Konsistenz–Problem, mit dem nicht auch das Target selbst zu kämpfen hätte!

Originalzustand des Prozessors hergestellt werden muß, was einen Abbau des Supervisory Stacks erzwingt.

Für ein Trap–Event werden dafür 3 zusätzliche Befehle benötigt:

```
MOVE.W (SP)+, (SP)   ; Statusreg. verschieben
MOVE.W (SP)+, (SP)   ; nochmals verschieben
MOVE (SP)+, SR       ; Statusreg. wiederherstellen
```

Diese Aktion scheint auf den ersten Blick unnötig aufwendig, bei näherer Betrachtung ist jedoch nur diese Befehlssequenz möglich. Grund dafür ist, daß beim Wiederherstellen des Statusregisters ein Wechsel des Prozessormodus vom Supervisory–Mode in den User–Mode erfolgen kann. Die beiden Modi haben aber getrennte Stacks. Im Supervisory–Mode zeigt der SP (SSP) auf den Supervisoy–Stack und im User–Mode zeigt der SP (USP) auf den User–Stack. Der SSP und der USP werden von der CPU intern umgeschaltet und beide über das Register A7 angesprochen. Würde nun zuerst das SR wieder hergestellt und danach der Stack fertig abgebaut werden, so würde der falsche Stack modifiziert werden (siehe unterhalb: Code (a)). Andererseits ist ein Abbau des Stacks mit nachträglichem Wiederherstellen des SR auch nicht korrekt (siehe unterhalb: Code (b)), weil zwischen den beiden Befehlen ein Interrupt den Stack modifizieren und somit den Wert des noch nicht wiederhergestellten SR zerstören könnte.

```
MOVE (SP)+, SR      ADDQ.L #6, SP
ADDQ.L #4, SP       MOVE 6(SP), SR
      (a)                 (b)
```

nicht korrekter Code zum Stackabbau

Auch ein Zwischenspeichern des SR in einem Register geht nicht, da dieses Register gerettet und nach dem Wiederherstellen des Statusregisters selbst wiederhergestellt werden müße. Aber leider liegt die der Wert des Registers nun am falschen Stack! Die erste Version des Stackabbaus ist also die einzig mögliche. Sie kopiert den Wert des SR am Stack auf die erste verwendete Adresse für den Stack–Frame des Trap–Handlers am Stack. Gleichzeitig wird der Stack bis zu dieser Adresse abgebaut. Letzter Schritt ist dann das Wiederherstellen des SR mit Hilfe des 'verschobenen' Wertes und der restliche Abbau des Stack–Frames.

Nun hat man sowohl für Traps als auch für Jumps die gleiche Ausgangssituation geschaffen und kann mit der Behandlung des Patched Codes beginnen. Leider gibt es neben vielen Maschinenbefehlen, die sich gegenüber einer Verschiebung im Memory neutral verhalten, auch eine größere Menge an Maschinenbefehlen, deren Auswirkung sich bei einem Verschieben im Memory stark ändert. Als einfachstes Beispiel sei hier einführend nur die Befehlsgruppe der Branches mit relativem Sprungziel genannt. In den nächsten Abschnitten werden nun all diese Befehle genau untersucht und jene Änderungen vorgestellt, die das ursprüngliche Verhalten wiederherstellen.

3.2.5.1 Branch conditional

Originalcode:

```
     Bcc L2
L1:  _______
      :
L2:  _______
```

Eventcode:

```
     Bcc L3
     JMP L1
L3:  JMP L2
```

3.2.5.2 Decrement and Branch conditional

Originalcode:

```
     DBcc L2
L1:  _______
      :
L2:  _______
```

Eventcode:

```
     DBcc L3
     JMP L1
L3:  JMP L2
```

3.2.5.3 Branch unconditional

Originalcode:

```
L1: BRA label
```

Eventcode:

```
JMP L1+label
```

Da die Adresse des Branch (ist die Adresse des Events) und der Label bekannt sind, kann das Sprungziel in eine absolute Adresse umgewandelt und statt des Branches ein Jump verwendet werden.

3.2.5.4 Jump

Bei einem Jump stehen mehrere Adressierungsarten zur Auswahl:

(An)	Register Indirect
d16(An)	Register Indirect with Displacement
d8(An,Ri)	Register Indirect with Index and Displacement
addr16	Absolute Short
addr32	Absolute Long
d16(PC)	PC–relative with Displacement
d8(PC,Ri)	PC–relative with Index and Displacement

Sind die ersten 5 Adressierungsarten unproblematisch und können unverändert verwendet werden, so machen die beiden letzten Adressierungsarten besondere Probleme. Der Sprung wird relativ zum Program Counter an der Statement Adresse im Targetcode berechnet. Der PC zeigt aber mitten in den Eventcode. Daher ist ein Umbau notwendig.

JMP d16(PC):

Da der Originalwert des PCs und das Displacement d16 bekannt sind, kann der Befehl durch einen Long–Jump ersetzt werden:

```
JMP StatementAdr+2+d16*
```

JMP d8(PC,Ri):

Dieser Befehl bereitet einige Probleme: Die Adressberechnung für das Sprungziel lautet "StatementAdr+2+Ri+d8". Dies ist kein statisch auswertbarer Ausdruck, da der Wert des Indexregisters nicht verfügbar ist. Da in sicherlich allen Fällen "StatementAdr+2+d8" relativ zum aktuellen PC größer als ein 8 Bit Offset sein wird, kann das Displacement d8 des Befehls auch nicht neu berechnet werden. Ein dynamisches Auswerten des Ausdrucks während der Eventbehandlung geht auch nicht, da dazu ein Register benötigt würde. Ein Wiederherstellen des Registers nach dem Befehl ist aber unmöglich, da ja mit dem Befehl selbst bereits wieder in den Targetcode "gejumped" wird.

Es bleibt nur die folgende, recht umständliche Lösung:

```
MOVE.L Ri, -(SP)
MOVE SR, -(SP)
ADD.L #StatementAdr+2+d8, 2(SP)
RTR
```

Dieser Code bedarf einiger Erklärung: Der erste MOVE–Befehl lädt den Inhalt des Indexregisters auf den Stack. Der Add–Befehl addiert den statisch berechenbaren Teil der Sprungadresse hinzu. Da dieser aber die Condition Codes verändert, muß zuvor noch das Statusregister auf den Stack gepushed werden. Somit hat der nun folgende "Return and Restore Condition Codes"–Befehl sowohl das Statusregister als auch die richtige Adresse am Stack, wenn er exekutiert wird. Der Stack wird dabei automatisch wieder abgebaut.

Leider ist diese Lösung aber noch immer nur die halbe Wahrheit. Der feine Unterschied liegt nämlich in der Verwendung des Indexregisters Ri. Es gibt die Möglichkeit, das Indexregister als Longword zu verwenden. Für diesen Fall funktioniert obige Lösung. Es existiert aber noch eine zweite Möglichkeit, vom Indexregister nur die Bits 0–15 zu verwenden (also als Word), wobei zur Berechnung der effektiven Adresse der Inhalt von Ri sign–extended wird. Und genau dieses Sign–Extend muß in diesem Fall durchgeführt werden, bevor der Inhalt von Ri zur Jump–Adresse addiert werden darf. Da hierzu aber ein Register benötigt wird, das gerettet und wiederhergestellt werden muß, wird der Umbau dieses Befehls schon sehr kompliziert und auch langsam:

*Bei JMP d16(PC) wird zur Berechnung der PC unmittelbar nach dem Instructionword des Befehls herangezogen. Also der PC des Befehls + 2 Bytes.

```
PEA #StatementAdr+2+d8
MOVE SR, -(SP)          ; Statusregister retten
MOVE.L D0, -(SP)        ; verwendetes Register retten
MOVE.W Ri, D0           ; Wert des Indexregister nach D0
EXT.L D0                ; sign-extend D0
ADD.L D0, 6(SP)         ; add. sign-extended Wert von Ri zu Adr.
MOVE.L (SP)+, D0        ; D0 wiederherstellen
RTR
```

Wird D0 als Indexregister verwendet, so muß hier statt D0 selbstverständlich ein anderes Register verwendet werden. RTR springt auf die Adresse, die am Stack bereit liegt und löscht diese und das Statusregister gleichzeitig vom Stack.

3.2.5.5 Jump Subroutine

Beim Sprung in eine Subroutine tritt ein weiteres Problem auf. Betrachten wir vorerst die primitive Lösung für den Eventcode:

```
JSR subr_adr      ; Aufruf der Subroutine
JMP target_code   ; Sprung zurück in den Targetcode
```

Im Normalfall wird die Subroutine ausgeführt, beendet und danach der Sprung zurück in den Targetcode durchgeführt. So weit, so gut. Nur besteht die Möglichkeit, daß während der Ausführung der Subroutine (und das kann lange dauern) das Event und mit ihm auch der Eventcode gelöscht werden (daß das Löschen von Events auch noch aus anderen Gründen für alle Instrumentierungen sehr kritisch ist, wird im Ausblick im Abschnitt 8 erläutert). Erfolgt jetzt ein Exit aus der Suboutine, so findet der Prozessor den eigentlich erwarteten Jump nicht mehr vor, sondern nur noch ein beliebiges (undefiniertes) Bitmuster. Daraus folgt, daß ein normaler Aufruf einer Subroutine nicht ratsam ist.

Die folgende Lösung erscheint wesentlich günstiger zu sein:

Originalcode:

```
JSR subr_adr
back: _______
```

Eventcode:

```
PEA #back
JMP subr_adr
```

Wird nun ein Exit aus der Subroutine durchgeführt, so findet diese die richtige Rücksprungadresse am Stack vor.

Da als Adressierungsarten für den JSR dieselben Möglichkeiten verwendet werden können wie für den JMP, bleiben die beiden Problemfälle des JMP mit PC–relativer Adresse aufrecht und müssen dementsprechend behandelt werden.

3.2.5.6 Branch Subroutine

Für den BSR gilt selbiges Problem wie für den JSR. Auch hier muß der Stackaufbau selbst durchgeführt werden. Der anschließende Branch muß in einen Long–Jump umgewandelt werden, falls der neu zu berechnende Offset die 16–Bit Grenze überschreitet.

3.2.5.7 Traps

Auch während der Ausführung eines Traps könnte (wie im Abschnitt über JSR erwähnt) das Event gelöscht werden. Daher ist auch hier,ähnlich wie beim JSR, die einfache Ausführung des Traps mit anschließendem Sprung in den Targetcode nicht günstig.

Da ein Trap selbst ein 1–Word Befehl ist, wird als Patch Code auch immer ein Trap ein gesetzt. Somit ist man innerhalb des Event Codes automatisch im Supervisory–Mode. Der Patch Code (zur Erinnerung: es wird eine unimplemented Instruction verwendet) baut den Stack auf ähnliche Art und Weise auf wie ein Trap. Dieser Stackaufbau wird für die "Emulation" des zu patchenden Traps verwendet. Dies ist recht günstig, da somit beim Return–From–Exception auch gleich der Originalzustand des Statusregisters wieder hergestellt wird (das SR wurde ja durch den Patch–Trap gesichert). Lediglich die Rücksprungadresse ist bei der unimplemented Instruction anders als beim Trap. Beim Trap wird als Rücksprungadresse immer der PC nach dem Trap–Befehl auf den Stack gepushed. Bei der unimplemented Instruction hingegen wird der PC der Instruction selbst auf den Stack gelegt. Da aber beide Befehle 1 Word lang sind, braucht zum PC am Stack nur um 2 erhöht werden, um die richtige Rücksprungadresse zu erhalten.

Der nächste Schritt nach dem ordnungsgemäßen Stackaufbau für den emulierten Trap ist, die Adresse des zugehörenden Trap Handlers aufgrund der Vektornummer zu bestimmen. Die Vektornummer kann dem gepatchten Befehl entnommen werden. Die Adresse, unter der die Adresse des Trap Handlers eingetragen ist, kann über die Basisadresse der Trap–Instruction–Table (siehe **(68000)**, Seite 36) und die Vektornummer berechnet werden. Die erhaltene Adresse des Trap Handlers wird auf den Stack gelegt und über ein Return–From–Subroutine angesprungen. RTS deswegen, weil dieser Befehl gleichzeitig die Sprungadresse wieder vom Stack entfernt.

```
ADDQ.L #2, 6(SP)
MOVE.L VectorTable+VectorNr*4, -(SP)
MOVE 10(SP), CCR*
RTS
```

Durch diese Emulation des gepatchten Traps ist es möglich, daß beim Return–From–Exception (RTE) des Traphandlers ein Rücksprung sofort in den Originalcode des Targets erfolgt. Dadurch wird das eingangs erwähnte Problem verhindert.

*Die Condition Codes müssen noch hergestellt werden, bevor in die Trap–Routine weiterverzweigt werden kann.

3.2.5.8 Instruktionen ohne Verzweigung aber mit Einbeziehung des Program Counters

PC–relative Adressierung kommt nicht nur bei JMP und JSR vor, sondern kann auch als Source–Adresse normaler Befehle, wie Move oder Add, verwendet werden.

Bei PC–relativer Adressierung mit Displacement kann die Adresse auf eine ähnliche Art und Weise wie beim Jump berechnet und durch eine absolute Adressierung ersetzt wer den.

Bei PC–relativer Adressierung mit Index und Displacement gibt es wiederum nur die Möglichkeit, die Source–Adresse des Befehls am Stack zusammenzubauen. Ein Befehl XXX.x d8(PC,Ri.W),... kann auf folgende Art umgebaut werden:

```
MOVE.L Ri, -(SP)
MOVE SR, -(SP)
ADD.L StatementAdr+2+d8, 2(SP)
MOVE (SP)+, CCR
XXX.x (SP)+, ...
```

Wird das Indexregister als Word verwendet, so muß wieder ein Sign–Extend durchgeführt werden.

```
PEA #StatementAdr+2+d8
MOVE SR, -(SP)          ; Statusregister retten
MOVE.L D0, -(SP)        ; verwendetes Register retten
MOVE.W Ri, D0           ; Wert des Indexregister nach D0
EXT.L D0                ; sign-extend D0
ADD.L D0, 4(SP)         ; add. sign-extended Wert von Ri zu Adr.
MOVE.L (SP)+, D0        ; D0 wiederherstellen
MOVE (SP)+, CCR         ; Condition Codes herstellen
XXX.x (SP)+, ...
```

Befehle, die als Source–Adresse* eine PC–relative Adressierung zulassen, sind: ADD, AND, CHK, CMP, DIVS, DIVU, LEA, MOVE, MOVEM, MULS, MULU, OR, PEA und SUB.

Nach einem solchen Befehl wird natürlich ein JMP benötigt, um zurück in den Targetcode zu gelangen.

3.2.5.9 Return from Subroutine

Da der Originalzustand des Stacks bereits wieder hergestellt wurde, bereiten die Befehle RTS und RTR keine Probleme und können unverändert eingesetzt werden.

*Als Destination–Adresse ist diese Adressierungsart nicht zulässig.

3.2.5.10 Return from Exception

RTE funktioniert problemlos, da am Stack die Originaladresse des zugehörenden Traps steht (der Stack des Event–Traps wurde ja bereits abgebaut).

3.2.5.11 Instruktionen ohne Verzweigung und ohne Einbeziehung des Program Counters

Alle Maschinenbefehle, die weder verzweigen, noch den Program Counter verwenden oder verändern, sind am leichtesten handhabbar. Sie brauchen nur im Anschluß an den Code des Event Handlers eingefügt und abgearbeitet werden. Danach kann mit einem Jump zurück in den Targetcode gesprungen und mit der Ausführung des Targetcodes fortgefahren werden.

Als Beispiel sei das Patchen eines Add–Befehls vorgestellt:

Targetcode:	Eventcode:
: MOVE.L -24(A6), D0 (ADD.L #739, D0) => JMP ev_handle go_on: TST.L D0 :	: ADD.L #739,D0 JMP go_on

3.3 Aufbau des ProSP

Alle Funktionen, die speziell auf den jeweils verwendeten Target–Prozessor zugeschnitten sind (alle Routinen zur Generierung des Event Codes), sind im ProSP zusammengefaßt.

3.3.1 ProSP Interface

Das ProSP Interface besteht aus einer Menge von Prozeduren, die teilweise über eine interne Datenstruktur miteinander kommunizieren. Befehle, die den Wert einer Target–Variablen kopieren sollen, werden in dieser Datenstruktur abgelegt und bei der eigentlichen Generierung des Event Handlers in dessen Code eingefügt.

Die Kopier–Befehle (Moves) für Target–Variablenwerte und Konstante werden durch eigene Prozeduren (Make...) generiert. Wieso diese Vorgangsweise praktisch ist, wird im Abschnitt 6.2.2 klar werden.

Im Überblick enthält das ProSP folgende Routinen:

- **InitCodeGeneration**: initialisiert die interne Datenstruktur, die als Puffer für generierte Move–Befehle dient und von den Routinen Make...() beschrieben und von GenerateEventCode() gelesen wird.

- **GeneratePatchCode**: erzeugt den notwendigen Befehl für den Patch (also einen Jump, Branch oder Trap).

- **MakeReference**: erzeugt Befehle für die Dereferenzierung einer Target–Variablen, wenn diese vom Typ Pointer ist.

- **MakeMove**: erzeugt einen Kopier–Befehl in den Event Buffer. Dabei wird zuvor die Adresse der Targetcode–Variablen berechnet (Index und Offset sowie der Datentyp werden berücksichtigt; siehe auch bei dem Beispiel im Abschnitt 5)

- **MakeMoveIntConst**: erzeugt den Code, der notwendig ist, um eine Integer–Konstante in den Event Buffer zu kopieren.

- **MakeMoveFloatConst**: erzeugt den Code, der notwendig ist, um eine Float–Konstante in den Event Buffer zu kopieren.

- **GenerateEventCode**: erzeugt den Event Code. Die Prozedur rettet die verwendeten Register, fordert den Event Buffer an, kopiert die Eventnummer in denselben, fügt den Code zum Kopieren der Variablen aus der internen Datenstruktur ein, restauriert wieder die geretteten Register und setzt schließlich den PatchedCode ein, bevor der Rücksprung aus dem Event Handler generiert wird.

Eine weitere, kleine, aber sehr nützliche Routine enthält das ProSP noch. Bei der Generierung der Befehle (durch die verschiedenen Prozeduren) wird jeweils die Dauer der erzeugten Maschinenbefehle (in Taktzyklen) in einer Variablen der internen Datenstruktur aufsummiert. Nach Abschluß der Codegenerierung kann dieser Wert abgefragt werden, um so die Gesamtdauer der Instrumentierung für ein Event zu erfahren:

- **CPU_Time**: Retourniert die Dauer der Instrumentierung eines Events in Takt zyklen des Prozessors.

Schließlich noch eine letzte, aber wichtige Funktion:

- **GetInstructionLength**: liefert zu einem gegebenen (Instruction–)Word die Länge des gesammten Befehls in Bytes. Diese Funktion ist etwa notwendig, um zu erfahren, wie lange der Patched Code ist; wieviele Bytes also entfernt werden müssen, um einem Patch Platz zu machen.

Die exakten Aufrufe dieser Routinen mit all ihren Parametern sind dem Dokument "LaSP und ProSP: Funktionsbeschreibung" zu entnehmen.

3.3.2 MOVE Befehle

Um die Werte von Target–Variablen in den Event Buffer kopieren zu können, sind ausschließlich MOVE–Befehle notwendig, da die Adresse einer Variablen entweder statisch berechnet werden kann (Siehe Abschnitt 4.2), oder sich aus einer einfachen Dereferenzierung eines Pointers ergibt. Vorgreifend auf das LaSP

(Abschnitt 4) sind hier alle möglichen Adressierungsarten einer Target–Variablen aufgezählt, für die Code generierbar sein muß.

1. am Stack (mit Offset)
2. absolute Adresse
3. in einem Register
4. Pointervariable beinhaltet absolute Adresse einer Variablen

Daraus können jetzt folgende MOVE–Befehle abgeleitet werden (die Adresse des Event Buffers steht dabei immer im Register A1):

1. `MOVE d16(FP),(A1)+`
2. `MOVE addr32,(A1)+`
3. `MOVE Rn,(A1)+`

Da auf eine Pointervariable genauso zugegriffen werden muß, werden für eine Dereferenzierung MOVE–Befehle mit (fast) denselben Source–Adressen benötigt, wie für den MOVE einer Variablen. Die Destination ist jedoch immer das Adressregister A0.

4a `MOVE d16(FP),A0`
4b `MOVE addr32,A0`
4c `MOVE (An),A0`

Nach einer Dereferenzierung steht also die Adresse einer Variable im Register A0 bereit. Allerdings kann zusätzlich noch ein Offset beziehungsweise Index zu berücksichtigen sein (bespielsweise bei dem Ausdruck x–>y[2]). Es ergeben sich also noch die folgenden notwendigen MOVE–Befehle:

1. `MOVE (A0),(A1)+`
2. `MOVE d16(A0),(A1)+`

Da der 68000–er zwischen Byte–, Word– und Longword–Befehlen unterscheidet, muß über den Datentyp festgestellt werden, welcher Befehl nun konkret verwendet werden muß. Handelt es sich bei der Target–Variablen um eine "double float" Variable, so müssen sogar zwei MOVE.L hintereinander ausgeführt werden.

3.3.3 Variablenwerte aus Registern

Oft kommt es vor, daß der Wert einer Target–Variablen in einem Register gehalten wird. Soll der Wert dieser Variablen verwendet werden, so kann er aus dem Register in den Event Buffer kopiert werden. Probleme entstehen dabei aber, wenn gerade dieses Register vom Event Handler verwendet wird und daher am Anfang des Event Handler Codes am Stack gesichert wurde. Soll nun innerhalb des Event Handlers auf den ursprünglichen Wert des Registers zugegriffen werden, so muß man diesen vom Stack holen.

Konkret heißt dies, daß aus einer Register–Variablen für die Dauer der Event Handling Routine eine Stackvariable wird. Die Maschinenbefehle zum Kopieren

dieser Variablen müssen daher entsprechend umgebaut werden. Dazu sind drei Fälle zu unterscheiden:

```
1. MOVE.x Dn, ...          ⇒    MOVE.x d16(SP), ...

2. MOVE.x (An), ...        ⇒    MOVE.L d16(SP),A0
                                MOVE.x (A0), ...

3. MOVE.x d16(An), ...     ⇒    MOVE.L d16(SP),A0
                                MOVE.x d16(A0), ...
```

Der Stackoffset der geretteten Register kann statisch bei der Generierung des Event Codes berechnet werden.

4 LaSP – Language Support Package: Das Interface zum Targetcode–File

Um simple Events im Code der Target–Software einhängen zu können, benötigt man genaue Informationen über diese Software. So braucht man etwa die genaue Adresse des Maschinencodes eines Sourcecode–Statements. Spielen bei der Occurence des Events auch Variablen der Target–Software eine Rolle, so benötigt man auch alle Informationen über Speicherplatz und Datentyp derselben. Weiters muß beim Setzen eines Events überprüft werden, ob die gewünschten Variablen an der gegebenen Programmstelle überhaupt gültig sind. Hierzu müssen die Scoping Rules der zugrundeliegenden Programmiersprache der Target–Software herangezogen werden.

All das sind jedoch Fragen, mit denen auch der (normalerweise) für die Entwicklung der Target–Software verwendete symbolische Debugger konfrontiert ist. Aus diesem Grund enthalten loadable (executable) Object–Files (Targetcode–Files) üblicherweise zusätzliche Debug–Informationen, aus denen alle benötigten Informationen gewonnen werden können. Das Vorhandensein dieser Debug–Informationen setzt allerdings meistens die Compilation der Source–Files mit einer speziellen (Debug–)Option voraus.

Es wird also ein Interface benötigt, welches imstande ist, dem Targetcodefile alle zur Instrumentierung eines Statement–Events notwendigen Informationen zu entnehmen. Bevor dieses Interface im Detail vorgestellt wird, seien noch einmal alle benötigten Informationen im Überblick zusammengefaßt:

- die absolute Adresse eines bestimmten Statements einer Funktion in einem Sourcecode–Modul. Auf diese Adresse soll ja der Patch zur Eventbehand lung gesetzt werden.

- der Maschinenbefehl, der an einer gegebenen absoluten Adresse im Targetcode steht. Dieser muß für den Patch Platz machen und an anderer Stelle exekutiert werden.

- Adresse sowie Datentyp einer Sourcecode–Variablen. Da der Datentyp der Variablen natürlich beliebig komplex sein kann, werden auch all jene Informationen benötigt, die es ermöglichen, den zusammengesetzten Datentyp bis zu einem Basisdatentyp aufzulösen (Man denke etwa an Pointer, Arrays oder Records).

Weitere Komplikationen ergeben sich durch die Scoping Rules einer Programmiersprache. Will man etwa die Informationen zu einer Variablen abfragen, so muß zuvor spezifiziert werden, an welcher Programmstelle in welcher Funktion und in welchem Modul diese Variable verwendet werden soll. Bevor also Speicherplatz und Datentyp einer Target–Variablen (oder auch die Adresse eines Statements) abgefragt werden können, muß der Scope durch die Parameter Modul/Funktion/Statement eindeutig definiert sein. Da in einem Modul die Sourcecode–Zeilen fortlaufend numeriert werden, sodaß auf einzelne Funktionen keine Rücksicht genommen werden muß, genügt die Angabe einer Zeilennummer anstelle

von Function/Statement. Da in einer Zeile mehrere Statements stehen können, muß zur Identifizierung eines Statements innerhalb einer Zeile noch die Angabe einer Spaltennummer erfolgen.

Zusammenfassend müssen:

- das Targetcodefile,
- das Sourcecode–Modul,
- die Zeilennummer und
- eine Spaltennummer

eindeutig spezifiziert werden, um ein Statement oder eine Variable eindeutig zu identifizieren.

4.1 Die Adresse eines Soucrecode–Statements

Soll ein Statement–Event in der Target–Software auf ein Sourcecode–Statement gepatched werden, so muß die Adresse des ersten Maschinenbefehls des zu diesem Statement gehören den Codes festgestellt werden.

Für diesen Zweck (auch ein symbolischer Debugger benötigt selbige Informationen) enthält das Targetcodefile für jedes exekutierbare Statement, das durch eine Zeilen– und eine Spaltennummer eindeutig identifiziert wird, die (absolute) Adresse im Targetcode.

Enthält ein Sourcefile zum Beispiel auszugsweise folgende Programmzeilen:

```
117 while (i--)
118 {
119 c=getc(); c&=0x5F;
    ......
126 }
```

so sind für die Statements in Zeile 119 etwa folgende Informationen gepeichert:

Zeile	Spalte	Adresse
119	10	5014E
119	21	5015C

Bild 4–1:

Zwei Fragen müssen im Zusammenhang mit der Identifizierung eines C–Statements durch Zeile und Spalte geklärt werden:

1. Ein Statement kann sich über mehrere Zeilen erstrecken. Welche Zeilennummer wird in diesem Fall verwendet?

2. Welche Spalte wird zur Identifizierung eines Statements herangezogen?

Um bei der Adressensuche Mißverständnisse zu vermeiden, wurde folgende Vorgangsweise gewählt: Zur eindeutigen Identifizierung eines Statements werden jeweils zwei Paare von Zeile/Spalte herangezogen, die sowohl den Anfang als auch das Ende eines Statements (ungefähr) festlegen. Kann aufgrund dieser Informationen im Targetcodefile genau ein Statement gefunden werden, so ist die Suche erfolgreich.

4.2 Speicherplatz und Datentyp einer Sourcecode–Variablen

Beim Setzen eines Events müssen die formalen Parameter einer Event–Definition a' la EDL an konkrete Variablen der Target–Software gebunden werden. Die Aufgabe des LaSP ist es nun unter anderem, zu einer gegebenen Variablen den zugehörenden Speicherplatz sowie den Typ der Variablen zu finden.

Obwohl grundsätzlich nur Basisdatentypen (int, char, float, ...) zugelassen werden, müssen 2 Fälle unterschieden werden:

1. Die Variable hat einen Basisdatentyp (int, char, ...) oder
2. die Variable hat einen zusammengesetzten Ausgangs–Datentyp (char*, struct, [ ], ...)

Im ersteren Fall ist das Finden der Adresse des Speicherplatzes nicht besonders aufwendig. Insgesamt existieren drei Möglichkeiten der Adressierung:

1. Absolute Adresse: etwa bei globalen Variablen.
2. am Stack: lokale Variable sind mittels Offset zum Framepointer zu finden.
3. in einem Register: lokale Variablen, die in einer Funktion oft benötigt werden, können vom Compiler in einem Register gehalten werden. Weiters kann in C auch eine Registervariable durch das Keyword **register** forciert werden.

Zum Beispiel könnte die Suche nach den Variablen "file_index", "mode" und "counter" folgendes Ergebnis liefern:

Variable	Typ	Adresse
file_index	unsigned long	50F3A
mode	char	–32(FP)
counter	short	D3

Bild 4–2:

Im zweiteren Fall der zusammengesetzten Datentypen kommt zu den 3 beschriebenen Adressierungsarten noch einiges hinzu. Obwohl, wie bereits erwähnt, nur Basisdatentypen zugelassen sind, sind auch die Adressen dieser Variablen von Bedeutung. Bei einem Record muß etwa eine einzelne Komponente, bei einem Array ein einziges Element adressierbar sein, wenn diese einen Basisdatentypen ergeben. Vorerst sei einmal gesagt, daß auch diese Variablen nur auf eine der 3 Arten adressiert werden können.

Ein konkretes Beispiel: Gewünscht wird die Adresse und der Typ von "my_rec.elem[3]". Eine Recherche im Targetcodefile liefert:

Variable	Typ	Adresse
my_rec	struct	50100
elem	vector	+12
elem[3]	int	+3*sizeof(int)

Bild 4–3:

Die Adresse ergibt sich also aus der "Basisadresse" von "my_rec" zuzüglich des Offsets des Elements "elem" des Records und dem Index 3 (erweitert gemäß der Länge des Typs von "elem"). Dies ergibt im Falle von 4 Byte Integers eine absolute Adresse von 50124.

Im LaSP wird jedem Datentyp eine eindeutige Nummer zugeordnet. Dabei werden zwei Nummerkreise gebildet. Typnummern < 256 sind fix den Standarddatentypen zugeordnet (wobei nicht jede Nummer verwendet wird, da die Anzahl von Basisdatentypen sicher kleiner als 255 ist). Typnummern >= 256 sind Verweise auf zusammengesetzte Datentypen. Mit Hilfe einer Funktion GetType() können genaue Auskünfte über den gegebenen Datentyp eingeholt werden.

Beispiel: Angenommen, man hat auf die Suche nach der Variablen "vector" folgende Antwort erhalten: Adresse = -16(FP), Type = 270. Der Aufruf von GetType() liefert schließlich: Type 270 = Vector mit 10 Elementen vom Typ char.

Auf diese Art und Weise kann (gemäß der Syntax der Target-Programmiersprache) der zusammengesetzte Datentyp aufgelöst und die Adresse berechnet werden.

4.3 Aufbau des LaSP

Das LaSP ist eine prozedurale Schnittstelle zu einer beliebigen Anzahl von Targetcodefiles. Die Schnittstelle wurde so ausgelegt, daß sie von der zugrunde liegenden Programmiersprache der Targetsoftware unabhängig ist. Dies ermöglicht ein problemloses Austauschen des Interfaces, ohne die (restliche) VTA–Software, die dasselbe verwendet, ändern zu müssen. Die benötigten Informationen sind in

allen (prozeduralen) Programmiersprachen ähnlich und können so an das Format des LaSP–Interfaces angepaßt werden.

Das hier beschriebene Interface wurde für die Programmiersprache C, unter Verwendung des MCC68K Compiliers und des LNK68K Linkers von Mictrotec Research, geschrie ben.

Das LaSP ist in zwei Stufen aufgebaut:

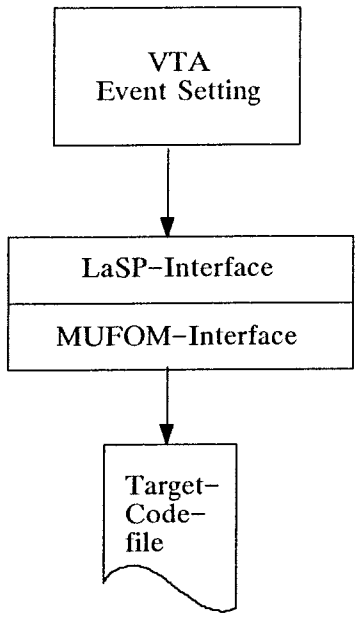

Bild 4: Aufbau des LaSP

Zwischen dem eigentlichen LaSP und dem Targetcodefile liegt noch das sogenannte MUFOM–Interface. Diese Schicht stellt primitive Routinen zum Lesen des Targetcodefiles zur Verfügung. Weiters verwaltet das MUFOM–Interface eine Datenstruktur, die ein effizientes Zugreifen auf gesuchte Informationen im Targetcodefile ermöglicht. Das Interface wird nach der Vorstellung des LaSP noch genau beschrieben.

4.3.1 LaSP Interface

Das LaSP Interface besteht aus folgenden Routinen:

- **OpenFile**: öffnet das gegebene Targetcodefile. Interne Datenstrukturen werden aufgebaut.
- **CloseFile**: schließt ein Targetcodefile und gibt die interne Datenstruktur frei.
- **CloseAllFiles**: schließt alle offenen Targetcodefiles. Die internen Datenstruk turen werden freigegeben.

- o **GetAddressOfLine**: sucht die Targetadresse zu einem gegebenen Sourcecode–Statement. Das Statement wird durch 2 Paare aus Zeilen– und Spaltennummer beschrieben, die den Beginn und das Ende des Statements spezifizieren.

- o **GetCodeOfAddress**: liefert zu einer Statement Adresse den Maschinenbefehl (und die Länge desselben).

- o **GetModuleOfFunction**: Diese Funktion liefert zu einer gegebenen Funktion das Modul (Sourcefile), in dem diese Funktion definiert ist.

- o **GetVariable**: liefert den Typ und die Adresse einer Sourcecode–Variablen.

- o **GetType**: Ist die Typnummer einer Variablen >255, so können mit dieser Routine die Attribute des zusammengesetzten Datentyps abgefragt werden.

- o **GetElement**: liefert zu einer Struct oder einer Union den Datentyp und den Offset eines Elements.

- o **SetActualFile**: erklärt das gegebene Targetcodefile als das nunmehr aktuelle. Alle oben beschriebenen Funktionen greifen somit auf dieses File zu.

- o **SetActualModule**: erklärt das gegebene Modul als das nunmehr aktuelle. Wie oben.

- o **SizeOf**: Liefert die Größe eines Standarddatentyps in Bytes.

Eine genaue Beschreibung der Funktionen ist dem Dokument "LaSP und ProSp: Funktionsbeschreibung" zu entnehmen.

4.3.2 MUFOM–Interface

Zwischen dem LaSP und dem Targetcodefile ist, wie schon erwähnt ein Low–Level Interface zwischengeschaltet. Dieses Interface erleichert und beschleunigt den Zugriff auf die Targetcodefiles. Der LNK68K Linker der verwendeten Entwicklungsumgebung für die Target–Software erzeugt ein File im IEEE–695 Format. Dieses Format wird auch MUFOM Format genannt (daher der Name des Interfaces). Das Format legt den Aufbau des Targetcodefiles genau fest. Aufgrund der Kenntnis dieses Formats können die notwendigen Informationen im Targetcodefile gelesen werden.

Um den Zugriff auf das Targetcodefile möglichst effizient durchführen zu können, legt das MUFOM–Interface beim Öffnen eines Targetcodefiles eine interne (memory–residente) Datenstruktur an. Über diese Datenstruktur kann das Interface in weiterer Folge relevante Informationen im Targetcodefile rasch finden.

4.3.2.1 Interne Datenstruktur

Das MUFOM–Format sortiert die in Form von Blöcken organisierten Debug informationen nach Modulen. Innerhalb eines Moduls existieren etwa Blöcke, die je eine Funktion des Moduls enthalten. Wird nun ein neues Targetcodefile geöffnet, so liest das MUFOM–Interface das File durch und merkt sich die Filepointer jedes Beginns eines neuen Blocks. Über diese Pointer kann die Lesemarke direkt auf jenes Modul oder jene Funktion gesetzt werden, von der Informationen benötigt werden.

Die Datenstruktur besteht aus drei einfach–verketteten Listen. Jedes neu geöffnete Targetcodefile bekommt einen Eintrag in der ersten Liste. Dieser Eintrag enthält globale Datenüber das File und eine Liste von Modulen (zweite Liste). Pro Modul im Targetcodefile wird ein Eintrag in dieser Liste erzeugt. Die Modulliste wiederum verweist auf die dritte Art von Listen, die Funktionsliste. Jede Funktion eines Moduls wird in ihrer zugehörigen Liste eingetragen.

Die Datenstruktur könnte also auszugsweise etwa folgendermaßen aussehen:

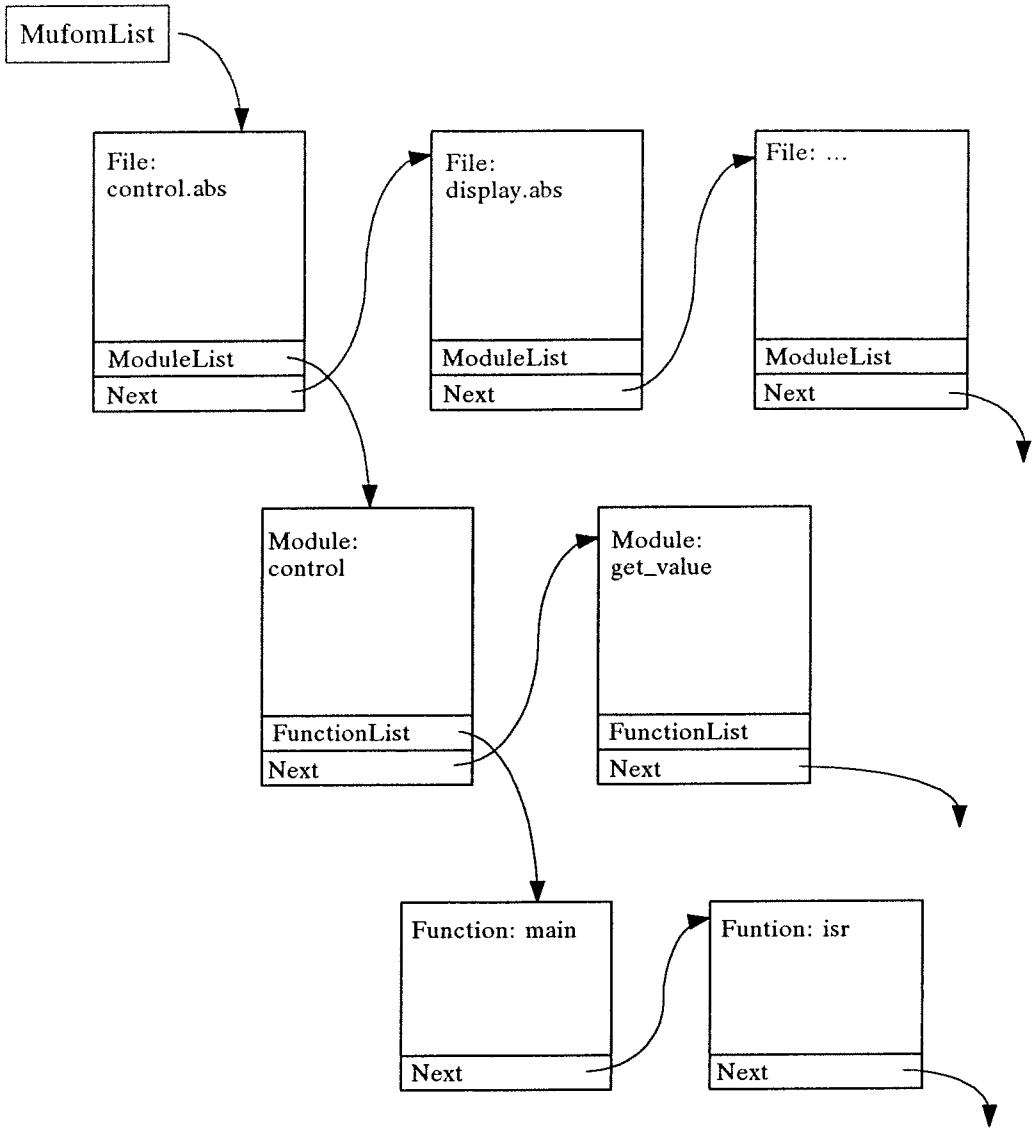

Bild 5: Aufbau des LaSP

Pro Targetcodefile werden folgende Daten verwaltet (MufomList):

- o Unix–Filepointer auf das Targetcodefile
- o Name des Targetcodefiles; inklusive des Pfadnamens (mit dem das File geöffnet wurde)
- o Name des Prozessors, für den dieser Code generiert wurde
- o Pointer auf den Block mit den Vereinbarungen der globalen Variablen
- o Pointer auf den Beginn der Load–Section. Dort steht der Code, der ins Target zu laden ist.
- o Liste aller Module

Pro Modul werden folgende Daten verwaltet (ModuleList):

- o Filename des Moduls (Name des Sourcecodefiles)
- o Pfadname des Moduls
- o Pointer zur Typtabelle des Moduls. Diese enthält die Typvereinbarungen aller Datentypen des Moduls.
- o Pointer zu den globalen Variablen des Moduls.
- o Pointer zur Liste aller Sourcecode Statements. Diese Liste enthält für jede exekutierbare Sourcecodezeile die absolute Adresse des zugehörenden Codes im Target.
- o Liste aller im Modul definierten Funktionen

Schlußendlich werden pro Funktion folgende Daten verwaltet (FunctionList):

- o Name der Funktion
- o Datentyp des Returnvalues
- o Pointer zu den Informationen über die Funktion
- o Erste Adresse des Codes der Funktion
- o Letzte Adresse des Codes der Funktion

4.3.2.2 Routinen zur Verwaltung der Datenstruktur

Die drei eben vorgestellten Listen werden unsortiert verwaltet. Gemäß der Reihenfolge, in der die einzelnen Datenblöcke im File vorgefunden werden, werden neue Listeneinträge am Anfang der jeweiligen Liste erzeugt.

Nach einem Targetcodefile, einem Modul oder einer Funktion wird in der Liste sequentiell gesucht. Für jedes Event muß immer nur einmal nach dem Targetcodefile, dem Modul und der Funktion gesucht werden, in dessen Kontext das Event gesetzt werden soll. Es ist daher nicht erforderlich, die Liste zu sortieren oder einen schnellen Suchalgorithmus anzuwenden.

Pro Liste gibt es eine Routine zum

- o Neuanlegen einer Liste
- o Eintragen eines Elements
- o Suchen eines Elements
- o und zum Löschen einer Liste.

4.3.2.3 Filezugriffsroutinen

Die Spezifikation des MUFOM–Formats definiert einige elementare Syntaxelemente. Um diese bequem lesen zu können, wurden Leseroutinen für diese Basiselemente entwickelt.

Diese Basiselemente* sind:

- o Byte

*Siehe MUFOM–Syntax im Dokument "MUFOM – IEEE Standard for Microprocessor Universal Format for Object Moduls"

- o Hexnumber
- o String
- o MUFOM interne Variable
- o Adresse
- o TypeTableEntry
- o Kommando Identifier

Die genaue Beschreibung der Funktionen kann im Dokument "LaSP und ProSp: Funktionsbeschreibung" nachgeschlagen werden.

4.4 Fehlerbehandlung im LaSP

Bei der Fehlerbehandlung werden Fehler, die bei der Verwendung des LaSP entstehen, von fatalen Fehlern beim Lesen des Targetcodefiles unterschieden. Erstere entstehen beispielsweise, wenn versucht wird, ein bereits geöffnetes Targetcodefile nochmals zu öffnen, oder wenn das zu öffnende File nicht existiert.

Um die Fehlerbehandlung bequem an die Benutzeroberfläche anpassen zu können, wird für LaSP–Fehler eine eigene Routine aufgerufen, die an die jeweilige Oberfläche angepaßt werden kann. Fehler, die die Routine aufrufen, sind im Dokument "LaSP und ProSp: Funktionsbeschreibung" unter der jeweiligen LaSP–Routine zu finden.

Fatale Fehler im MUFOM–Interface (beim Lesen eines Files) können vorkommen, wenn die Datenstruktur in einem Targetcodefile nicht mit dem erwarteten Format übereinstimmt. Für diese Fehlerart wird eine eigene Fehlerroutine verwendet, um eine gesonderte Behandlung zu ermöglichen. Diese gibt derzeit eine genaue Meldung mit Ursache und Ort des Fehlers auf das Standard–Error Device aus.

5 Beispiel zur Verwendung von LaSP und ProSP

Zur Verbesserung des Verständnisses der Funktionsweise und der Verwendung von ProSP und LaSP soll ein Beispiel dienen, das aufzeigt, welche Aufrufe welcher Prozeduren in welcher Reihenfolge notwendig sind, um alle für ein Statement–Event notwendigen Informationen zu sammeln und die Maschinenbefehle für den Patch und die Event Handling Routine zu generieren:

In einem Node des Targets läuft die Software des Targetcodefiles "control.exe". Ein Statement–Event soll im Module "control.c" auf jenes Sourcecode–Statement gesetzt werden, das in der Zeile 17 von Spalte 4 bis (zirka) Spalte 21 reicht. Bei der Event–Occurence soll die Sourcecode–Variable "**regler7.soll wert[3]**" extrahiert werden.

Man benötigt also folgende Informationen aus dem Targetcodefile:

- o die absolute Adresse des C–Statements im Modul "control.c"
- o jenen Maschinenbefehl, der an dieser Stelle im Memeory des Targets geladen ist
- o die Adresse der Sourcecode–Variablen.

Anhand dieser Informationen ist es dann möglich, den Code für den Event Handler und den Patch zu generieren.

Zuerst aber die notwendigen Prozeduraufrufe im LaSP*:

OpenFile ("control.exe");
SetActualFile ("control.exe");
SetActualModule ("control");
GetAddressOfLine (17,4, 17,21, &StatementAddress);
⇒ StatementAddress= 503A4
PatchedCodeLength= **GetCodeOfAddress** (StatementAddress, PatchedCode, &KindOfPatch);
⇒ PatchedCode= "ADD.L #1000, D4" (hex: 0684 0000 03E8)
⇒ PatchedCodeLength= 6 Bytes
⇒ KindOfPatch= JUMP
GetVariable (StatementAddress, "regler7", &VarAddr, &AddrMode, &TypeNumber);
⇒ VarAddr= -324(FP)
⇒ AddrMode= VAR_ON_STACK
⇒ TypeNumber= 274
GetType (TypeNumber, &TypeAttributes);
⇒ TypeAttributes= STRUCT (und weitere Attribute der Struct)
GetElement (TypeAttributes, "sollwert", &TypeNumber, &Offset);
⇒ TypeNumber= 273
⇒ Offset= 6 Bytes
GetType (TypeNumber, &TypeAttributes);

*Die genaue Spezifikation der Prozeduren ist dem Dokument "LaSP und ProSP: Funktionsbeschreibung" zu entnehmen.

⇒ TypeAttributes= VECTOR mit 16 Elementen vom Type LONG_INT

Das Ergebnis kann nun etwa so interpretiert werden: Die Adresse des gewählten Statements in Zeile 17 ist 503A4. Der Maschinenbefehl an dieser Stelle ist 6 Bytes lang. Es wird ein Jump als Patch zum Event–Code vorgeschlagen. Da die Variable ein Element eines Arrays innerhalb eines Records (struct) ist, muß ihre Adresse wie folgt berechnet werden (wie wir gleich sehen werden, erfolgt diese Berechnung eigentlich automatisch im ProSP):

"**regler7**" hat die Adresse = –324(FP) und ist eine Struct. Das Element "**sollwert**" hat einen relativen Offset zum Beginn der Struct von 6 Bytes. Daher ist "regler7.sollwert" auf der Adresse = –318(FP) zu finden. Es wird aber das 4. Element von "sollwert" benötigt (**sollwert[3]**), daher muß zu –318(FP) noch 3*sizeof(LONG_INT) gezählt werden. Dies ergibt schlußendlich eine Adresse für "**regler7.sollwert[3]**" von –306(FP).

Um die Länge des Datentyps LONG_INT zu erfahren, wurde gerade intuitiv die Standard–C–Funktion sizeof()* angewendet. Da das LaSP aber unabhängig von der Target–Programmiersprache und der Target–CPU eingesetzt werden soll (beziehungsweise durch verschiedene Packages an diese anpaßbar sein soll), muß für die Berechnung der Länge von Basisdatentypen die LaSP–eigene Funktion SizeOf() verwendet werden. Korrekterweise erfährt man also die Größe von LONG_INT mit

VarSize= **SizeOf** (LONG_INT);
⇒ VarSize= 4

An dieser Stelle wurden nun alle Informationen zusammengetragen, die zur Code generierung benötigt werden. Zusammenfassend sind dies:

- o StatementAddress= 503A4
- o PatchedCode= "ADD.L #1000, D4"
- o PatchedCodeLength= 6
- o KindOfPatch= JUMP
- o Basisadresse der Variable = –324(FP); AddrMode= VAR_ON_STACK
- o Offset= 6
- o Index= 3
- o VarSize= 4

Der Codegenerierung steht also nichts mehr im Wege. Lediglich die Adresse, auf die der Event Handler geladen werden soll, und die Eventnummer für das neue Event (anhand derer das Event bei der Occurence eindeutig identifizierbar sein muß) müssen noch fest gesetzt werden. Nehmen wir als Adresse beispielsweise 58000 und als Eventnummer 23.

InitCodeGeneration();

*Eigentlich ist **sizeof** ein Keyword von Standard–C und keine Funktion im herkömmlichen Sinn.

MakeMove (VAR_ON_STACK, –324, 6, 3, 4);
⇒ "MOVE.L –306(FP), (A1)+" *
PatchCodeLength= **GeneratePatchCode** (&KindOfPatch, 58000, 503A4, 23, PatchCode);
⇒ KindOfPatch= BRANCH
⇒ PatchCode= "BRA $58000" (hex: 6000 7C6A)
⇒ PatchCodeLength= 4
EventCodeLength= **GenerateEventCode** (KindOfPatch, 503A4, 58000, 23, "ADD.L #1000, D4", 6, EventCode);
⇒ EventCode="

```
MOVEM.L A1, -(SP)          ;Register retten
MOVE.L $EventBuffer, A1    ;Eventbuffer anfordern**
MOVE.W #23, (A1)+          ;Eventnummer kopieren
MOVE.L -306(FP), (A1)+     ;extrahierte Var. kopieren
MOVE.L A1, $EventBuffer    ;Bufferrückgabe***
MOVEM.L +(SP), A1          ;Register restaurieren
ADD.L #1000, D4            ;PatchedCode ausführen
BRA $503AA"                ;Rücksprung in Targetcode
```

⇒ EventCodeLength= 42

InitCodeGeneration() initialisiert die ProSP–interne Datenstruktur. Danach kann der Code zum Kopieren der Variablen in den Event Buffer generiert werden. Als nächster Schritt erfolgt die Generierung des Patches selbst. Die Routine GeneratePatchCode() benötigt so wohl die Adresse des Statements als auch die Adresse des Event Handlers, um entscheiden zu können, ob der vorgeschlagene Jump durch einen Branch–Befehl ersetzt werden kann (der etwas schneller als der Jump ist). Wie man sieht, geht sich in diesem Beispiel ein Branch aus, um in den Event Handler springen zu können (als Label ist beim 68000er maximal ein signed Word möglich). Da im Falle eines Patches mit Trap die Eventnummer direkt in den Trap–Befehl kodiert wird, benötigt die Routine auch diesen Parameter.

Als letzter Schritt erfolgt nun noch die Generierung des Codes für den Event Handler. GenerateEventCode() stellt fest, welche Register gerettet werden müssen, fordert den Event Buffer an, kopiert die Eventnummer in denselben, fügt den Code zum Kopieren der Variable aus der internen Datenstruktur ein, restauriert wieder die geretteten Register und führt schließlich den PatchedCode aus, bevor der Handler verlassen und mit der Ausführung der Target–Software fortgesetzt wird.

---

*Der generierte Code zum Kopieren der Variable wird in der internen Datenstruktur des ProSP abgelegt.

**Derzeit erfolgt die Bufferanforderung laut Testumgebung (siehe Abschnitt 7).

***Nur für die Testumgebung notwendig.

6 Statement–Event Mapping Syntax

Wird ein Event im Target gesetzt, so müssen alle formalen Parameter der EDL–Definition an konkrete Variablen im Target geknüpft werden. Dieser Vorgang wird Event Mapping genannt.

Als Targetcode–Variablen kommen immer nur Variablen eines Basisdatentypes in Frage. Für die Programmiersprache C sind dies:

- o unsigned long
- o unsigned short
- o unsigned int
- o short
- o long
- o int
- o unsigned char
- o char
- o float
- o double

Natürlich müssen aber auch Variablen, die Element eines Records oder Arrays sind, oder über einen Pointer angesprochen werden, adressiert werden können. In der C–Syntax werden solche Konstrukte als Terme bezeichnet. Die Adresse der Variablen muß aber immer statisch berechenbar sein. Einige Beispiele im Anschluß verdeutlichen erlaubte und verbotene Terme. Übrigens sind auch Konstante als Term erlaubt.

Beispiele für gültige Terme (sofern sie letztendlich immer einen Basisdatentyp ergeben):

a.b	a[2]	a[7].b.c[1][2][8]	123	.17e+5
a->b	(*a).b	(*(*(*a).b).c).d	'a'	137823.21
*a	**a.b	a->b->c[2]	0xFA3C	12.

Beispiele für ungültige Terme:

a[i]	a.b[3+c]	123*a	i++	*(ptr+24)

Um beim Setzen eines Statement–Events die formalen Parameter an Target–Variablen binden zu können, wird ein Ausdruck der Form

ev_name (parameter1, parmeter2, ...)

verwendet. Dabei ist ev_name der Instanzen–Name eines Statement–Events und darf nicht mit dem Klassennamen verwechselt werden. Die Event Mapping Syntax kann nun genau solche Ausdrücke erzeugen beziehungsweise analysieren.

Event Mapping Syntax:

Der Event Mapping Syntax wurde die gleiche Notation zugrundegelegt, die auch für die Programmiersprache C* verwendet wird. Dabei trennt | zwei Alternativen, { } umschließt einen Ausdruck, der beliebig oft wiederholt werden kann, [ ] umschließt optionale Teile und ( ) wird für die normale Klammerung verwendet. Alle **fett** geschriebenen Zeichen sind Terminalsymbole. Terminalsybole können auch in einfache Hochkommata eingeschlossen werden, um Mißverständnissen vorzubeugen.

```
ev_map:
        identifier '(' [ parameter { , parameter } ] ')'

parameter:
        term
        | const

term:
        identifier
        | '(' term ')'
        | term [ constant ]
        | term . identifier
        | term -> identifier
        | * term

identifier:
        ( letter | underscore ) { letter | digit | underscore }

constant:
        int_const
        | char_const
        | float_const
```

Die Syntaxdefinition für letter, underscore, digit, int_const, char_const und float_const sind einem C–Manual zu entnehmen (sollten allseits bekannt sein).

Für diese Syntax muß nun ein Übersetzer entwickelt werden. Dazu sind eine Lexikalische Analyse, eine Syntaxanalyse, eine Semantische Analyse und schließ lich ein Codegenerator notwendig.

6.1 Lexikalische Analyse

Die lexikalische Analyse liest die Zeichenfolge des gegebenen Ausdrucks und wandelt diese in eine Folge von Symbolen um, wobei ein Symbol der kleinsten semantischen Einheit des Ausdrucks entspricht. Einem Symbol können auch ein oder mehrere Attribute mitgegeben werden. Die Identifizierung eines Symbols erfolgt über eine Kodierung der semantischen Einheiten. Diese werden Token genannt. Ein Symbol besteht somit aus einem Token und optionalen Attributen.

*Kernighan, Ritchie, "The C Programming Language", 1977 Prentice Hall

Aus der Event Mapping Syntax werden folgende Symbole abgeleitet (Token werden immer **fett** geschrieben):

Token	Attribute
id	Id–Name
const	Value
*	
.	
–>	
(	
)	
[	
]	

Bild 6–1: Token der Event Mapping Syntax

Die Syntax der Event Mapping Syntax kann durch einen endlichen Automaten analysiert werden. Als Beispiel wird der Automat zur Erkennung der "int_const" und der "float_const" von C vorgestellt. Die Automaten der verbleibenden Konstrukte sind wesentlich einfacher und sehr ähnlich konstruierbar.

Als Darstellungsfom wurde das Zustandsdiagramm gewählt. Die Zustände werden durch die Knoten des Graphen dargestellt, dabei wird der Anfangszustand mit fettem Rand und jeder Endzustand mit doppeltem Rand gezeichnet. An den Kanten zwischen zwei Zuständen stehen die Eingabezeichen, mit denen ein Ü bergang von einem zum anderen Zustand stattfindet. Da die Lexikalische Analyse immer nach dem Prinzip des Longest Input Match arbeiten muß, werden solange Eingabezeichen gelesen, bis das Endezeichen des bearbeiteten Symbols auftritt. Leider haben aber die wenigsten Symbole ein eindeutiges Endezeichen. So muß mit dem Lesen aus der Eingabe solange fortgesetzt werden, bis ein Zeichen nicht mehr zum bearbeiteten Symbol gehört. Da dieses Zeichen aber bereits gelesen wurde, muß es entweder an die Analyse des nächsten Symbols übergeben werden oder in die Eingabe zurückgestellt werden. Da zweitere Lösung einfach zu implementieren ist wurde dieser Ansatz gewählt. Im Zustandsdiagramm drückt sich das Zurückstellen eines Eingabezeichens durch einen * im Knoten aus.

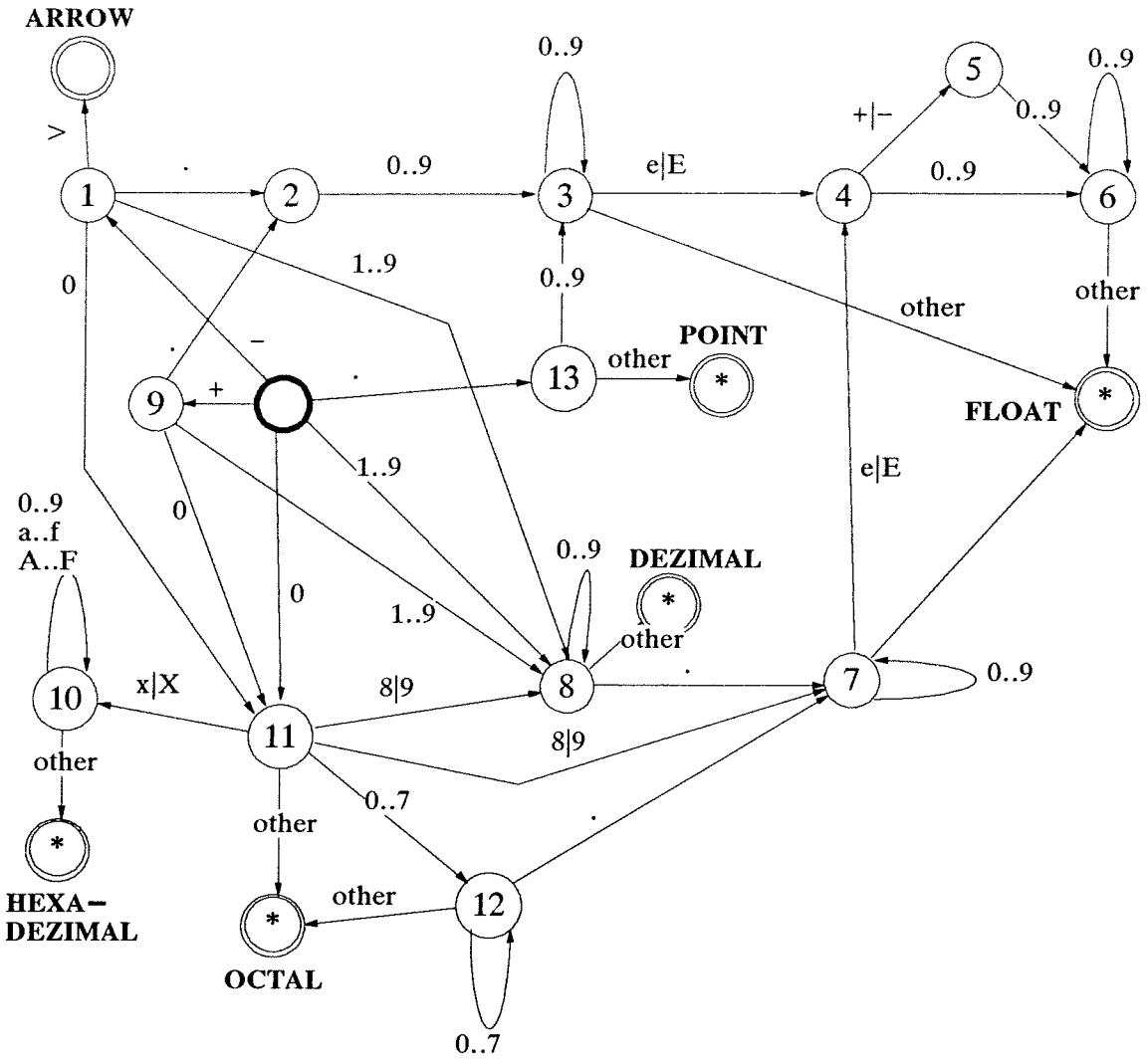

Bild 6: Automat für Float– und Integer–Zahlen

Für die Lexikalische Analyse wurde eine Prozedur `Lex(token)` geschrieben, deren Verwendung im Dokument "LaSP und ProSP: Funktionsbeschreibung" nachzulesen ist.

6.2 Syntaxanalyse

Die Syntaxanalyse untersucht, ob die von der Lexikalischen Analyse gelieferte Symbolfolge einen syntaktisch richtigen Ausdruck darstellt (ob sie der Grammatik der Sprache entspricht). Sie untersucht also die Struktur eines gegebenen Ausdrucks. Läßt sich der Ausdruck in der Grammatik nicht ableiten, so liegt ein Fehler vor.

Als Eingabe erwartet die Syntaxanalyse eine Folge von Token. Attribute der Token werden dabei nicht berücksichtigt; diese werden unverändert der Semantischen Analyse übergeben.

Übliche Syntax–Analysatoren setzen für gewöhnlich eine kontextfreie Grammatik voraus. Der nächste Schritt ist also die Entwicklung einer kontextfreien Grammatik für die Event Mapping Syntax. Zur besseren Lesbarkeit wurden die Nonterminalsymbole der Syntax für die kontextfreie Grammatik durch Großbuchstaben ersetzt. Die Token werden weiterhin **fett** geschrieben.

Kontextfreie Grammatik:

(1) S –> **id** (T)
(2) T –> AL | e
(3) L –> , AL | e
(4) A –> P | **const**
(5) P –> **id** | (P) | P [const] | P **.id** | P **–>id** | * P

Bevor diese Grammatik verwendet werden kann, müssen noch sämtliche Linksrekursionen entfernt werden. Da in der Grammatik aber nur direkte Linksrekursionen vorkommen, können diese leicht nach folgender Regel (semantikneutral) aufgelöst beziehungsweise in Rechtsrekursionen umgewandelt werden*:

Regeln der Form
$N \rightarrow N\alpha_1 \mid N\alpha_2 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \ldots$
werden durch die Regeln
$N \rightarrow \beta_1 N_r \mid \beta_2 N_r \mid \ldots$
$N_r \rightarrow \alpha_1 N_r \mid \alpha_2 N_r \mid \ldots \mid \varepsilon$
ersetzt.

Daraus ergibt sich eine LL(1)–Grammatik.

(1) S –> **id** (T)
(2) T –> AL | ε
(3) L –> , AL | ε
(4) A –> P | **const**
(5) P –> ***P** | **id** V | (P) V
(6) V –> **[const]** V | **.id** V | **–>id** V | ε

Als nächster Schritt werden die First– und Follow–Mengen für jedes Nonterminal und jede rechte Seite einer Regel bestimmt. Diese sind notwendig, um nachweisen zu können, daß die gegebene Grammatik auch wirklich eine LL(1)–Grammatik ist und somit die Syntaxanalyse widerspruchsfrei ablaufen kann. Weiters dienen die erhaltenen Mengen als praktische Unterstützung bei der Erstellung des Analyseprogramms.

*Bestehende Rechtsrekursionen brauchen dabei nicht umgewandelt werden.

Die Menge First(α) gibt an, mit welchen Terminalsymbolen die (Teil–)Satzform α beginnen kann beziehungsweise ob sie nach ε ableitbar ist. Die Menge Follow(N) gibt an, welche Terminalsymbole (inclusive $)* dem Nonterminalsymbol N in einer beliebigen Satzform folgen können**.

	First	Follow
S	id	$
T	* id const (ε	)
L	, ε	)
A	* id const (	,)
P	* id (	,)
V	[. -> ε	,)

Bild 6–2: First– und Follow–Mengen

Eine Grammatik ist genau dann LL(1), wenn für die Alternativen α_i der Regeln jedes Nonterminales N gilt***:

- o $First(\alpha_i) \cap First(\alpha_j) = \{\} \; \forall \; i,j \; i \neq j$
- o höchstens ein α_i läßt sich nach ε ableiten
- o falls $\alpha_i \Rightarrow^* \varepsilon$ dann muß gelten: $First(\alpha_j) \cap Follow(N) = \{\} \; \forall \; i \neq j$

Anhand der obigen Tabelle läßt sich die Gültigkeit dieser Regeln für die gegebene Grammatik leicht überprüfen.

Als Methode zur Syntaxanalyse wurde ein Top–Down Analyseverfahren mit rekursivem Abstieg gewählt. Bei diesem Analyseverfahren wird für jedes Nonterminal eine rekursive Prozedur gebildet. Der Kontroll–Algorithmus und das Steuerprogramm (die Grammatik) verschmelzen so zu einem Programm. Das Verfahren wurde gewählt, da sich hier die Semantische Analyse und die Übersetzung, nach dem Prinzip der Syntaxgesteuerten Übersetzung, leicht einbinden lassen.

Als Erleichterung bei der Programmierung der rekursiven Prozeduren wird zuvor die Top–Down Analysetabelle aufgestellt. Die Tabelle wird durch eine Matrix M[N,t] über den Indexbereichen Nonterminale (Zeilen) und Eingabesymbole (Spalten) dargestellt. Die Einträge bestehen aus den rechten Seiten der Produk-

*$ wird als (künstliches) Endezeichen des zu analysierenden Ausdrucks angenommen.

**Die Berechnung der First– und Follow–Mengen wurde dem Skriptum "Übersetzerbau" von Prof. Brockhaus entnommen.

***Dem Skriptum "Übersetzerbau" von Prof. Brockhaus entnommen.

tionsregeln N -> α, die angewendet werden, wenn sich das Programm in der rekursiven Prozedur N befindet und das nächste Eingabesymbol t ist.

	Nächstes Eingabesymbol:									
	id	**const**	(	)	*	[	.	**->**	,	$
S	**id**(T)									ε
T	AL	AL	AL	ε	AL					
L				ε					,AL	
A	P	**const**	P		P					
P	**id**V		(P)V		*P					
V				ε		**[const]**V	**.id**V	**->id**V	ε	

Bild 6–3:

Das Lesen der Eingabe wird durch die für die Lexikalische Analyse entwickelte Prozedur `Lex(token)` erledigt, die nach jedem Aufruf das nächste Token aus dem zu analysierenden Ausdruck bereitstellt. Das Programm wird hier in einer C–ähnlichen Pseudosprache beschrieben.

```
S()
{
    Lex (token);
    if (token == id);
    Lex (token);    /* (  */
    T();
}

T()
{
    if (token ∈ {*, id, const, (})
        A();
        L();
    elseif (token == ));
}

L()
{
    if (token == ,)
        Lex (token);
        A();
        L();
```

```
    elseif (token == ));
}

A()
{
    if (token ∈ {*, id, (})
        P();
    elseif (token == const)
        Lex (token);
    elseif (token ∈ {,, )}); }

P()
{
    switch (token)
        *:  Lex (token);
            P();
        id: Lex (token);
            V();
        (:  Lex (token);
            P();
            Lex (token);
            V();
        ,:
        ):;
}

V()
{
    switch (token)
        (:  Lex (token);
            if (token == const);
            Lex (token);
            if (token == ));
            Lex (token);
            V();
        .:  Lex (token);
            if (token == id);
            Lex (token);
            V();
        ->: Lex (token);
            if (token == id);
            Lex (token);
            V();
        ,:
        ):;
}
```

Der Start der Syntaxanalyse erfolgt durch Aufruf von:

```
Lex (token); S();
```

Das Programm stellt vorerst nur die reine Syntaxanalyse dar. Für einen ordnungsgemäßen Ablauf muß das Programm natürlich noch um Fehlermeldungen erweitert werden.

6.2.1 Semantische Analyse

In der semantischen Analyse wird der Typ eines formalen Parameters mit dem Typ einer analysierten Variablen aus der Syntaxanalyse verglichen. Die Syntaxanalyse übergibt der Semantikanalyse die laufende Nummer des analysierten Parameters, dessen Typ überprüft werden soll, und den zu prüfenden Typ. Aufgrund der laufenden Nummer muß die Semantikanalyse den Typ des formalen Parameters der EDL–Definition suchen und die Typ–Überprüfung vornehmen. Die Semantikanalyse ist derzeit immer "erfolgreich", da die Implementierung der EDL noch nicht verfügbar ist. Die Schnittstelle der Semantikanalyse ist im Dokument "LaSP und ProSP: Funktionsbeschreibung" beschrieben.

6.2.2 Syntaxgesteuerte Übersetzung und Codegenerierung

Bei der syntaxgesteuerten Übersetzung wird bereits während der Syntaxanalyse die Semantikanalyse und die Codegenerierung durchgeführt. Da die Syntaxanalyse durch rekursiven Abstieg ausprogrammiert ist, werden für die Semantikanalyse und die Codegenerierung einfach weitere Aktionen (in Form von Prozeduren) in die rekursiven Prozeduren der Syntaxanalyse eingebaut. Eine Übersicht gibt Aufschluß über die Aktionen, die bei der Abarbeitung durchgeführt werden müssen. Die linke Seite einer Produktion ist dabei äquivalent zur rekursiven Prozedur, in der sich die Syntaxanalyse gerade befindet. Die rechte Seite der Produktion ist äquivalent zu den Aktionen, die an dieser Programmstelle ausgeführt werden müssen. Ein Nonterminal entspricht einem weiteren Aufruf einer rekursiven Prozedur. In Klam mern werden die Namen der Prozeduren genannt, die die jeweiligen Aktionen ausführen. Im Anschluß daran werden diese Prozeduren dann näher beschrieben.

Produktion:	nach Abarbeitung von:	Aktionen:
S –> **id** (T)	**id**	Eventname merken
A –> P	P	Code für Variable generieren (ParameterOk)
A –> **const**	**const**	Code für Konstante generieren (ParameterOk)

Bild 6–4:

Produktion:	nach Abarbeitung von:	Aktionen:
P –> *P	P	Code für Dereferenzierung des Pointers generieren (Reference)
P –> **id** V	**id**	Variable im Targetcodefile suchen (Variable)
V –> **[const]** V	]	Index des Arrays berechnen (Indexing)
V –> **.id** V	**id**	Element der Struct oder Union suchen, Offset des Elements relativ zur Struct oder Union berechnen (PointElement)
V –> **–>id** V	**id**	Dereferenzierung des Pointers generieren, Element der Struct oder Union suchen, Offset des Elements relativ zur Struct oder Union berechnen (ReferenceElement)

Bild 6–4: (Fortsetzung)

- **Variable()**
  Variable() sucht im Targetcodefile nach dem Variablennamen, der von Lex(token) als Attribut zum Token **id** mitgeliefert wird. Die Prozedur stellt fest, ob die Variable vorhanden ist und liefert deren Adresse sowie deren Typ.
- **Reference()**
  Ist der Typ der zuletzt gelesenen Variablen (oder eines Elements einer Struct oder Union) ein Pointer, so muß der Codegenerator eine Dereferenzierung erzeugen. Die Funktion retourniert die Adresse und den Typ der Variablen, auf die der Pointer zeigt.
- **PointElement()**
  Ist der Typ der zuletzt gelesenen Variablen eine Struct oder eine Union, so wird nach dem Element mit dem Namen des Attibutes vom Token **id** gesucht. Retourniert wird der neue Typ und der berechnete Offset des Elements.
- **Indexing()**
  Ist der Typ der zuletzt gelesenen Variablen ein Array, so wird im Targetcodefile nach dem Typ der Elemente des Arrays und nach der Länge des Arrays gesucht. Ist der Index (Attribut vom Token const) kleiner als die Arraylänge, so wird der Index berechnet und zusammen mit dem neuen Typ des Elements retourniert.

- **ParameterOk()**
  Wurde eine Konstante als Parameter identifiziert, so muß der Codegenerator Code für einen Konstanten–Parameter generieren. Wurde hingegen ein Term als Parameter identifiziert und ist der zuletzt verwendete Typ ein Basisdatentyp, so ist die Semantikanalyse aufzurufen. Diese vergleicht den Basisdatentyp mit dem Typ des formalen Parameters aus der EDL–Definition. Stimmen die beiden berein (oder kann eine Typkonvertierung eine Übereinstimmung herstellen), so kann der Codegererator Code für einen Variablen–Parameter generieren. Er benötigt dazu den (eventuell in Indexing() errechneten) Index und den (eventuell in PointElement() ermittelten) Offset, sowie die Größe des Datentyps.

Im Abschnitt 3.3 wurde bereits erwähnt, daß für jeden Parameter der Parameterliste eines Statement–Events Maschinenbefehle generiert werden müssen, die den Wert einer Variablen in den Event Buffer kopieren. Der Codegenerator führt diese Aufgabe durch.
Drei Gruppen von Code müssen generiert werden:

1. Dereferenzierung
2. Move in den Event Buffer
3. Move Konstante in den Event Buffer

Eine Dereferenzierung ist notwendig, wenn auf eine Variable über einen Pointer zugegriffen wird. Soll zum Beispiel der Parameter "*counter" kopiert werden, so muß "counter" zuerst dereferenziert werden. Der Inhalt der Adresse, die sich daraus ergibt, kann dann mit einem Move in den Event Buffer kopiert werden. Im ProSP werden für die Generierung dieser Maschinenbefehle folgende Prozeduren zur Verfügung gestellt:

1. Zur Dereferenzierung: **MakeReference()**
2. Um den Wert einer Variablen in den Event Buffer zu kopieren: **MakeMove()**
3. Um eine Integer–Konstante zu kopieren: **MakeMoveIntConst()**
4. Um eine Float–Konstante zu kopieren: **MakeMoveFloatConst()**

Aufgerufen werden diese Prozeduren von **Reference()** und **ParameterOk()** der Syntax gesteuerten Übersetzung.

6.2.3 Syntax–Errors

Bei einem Syntax–Fehler wird eine Fehlerbehandlungsroutine aufgerufen, die an die jeweils verwendete Benutzeroberfläche angepaßt werden kann. Im derzeitigen Entwicklungsstadium wird eine Fehlermeldung an das Standard–Output Device gesendet. Die Schnittstelle der Fehlerbehandlungsroutine SyntaxError(), sowie alle möglichen Fehler sind im Dokument "LaSP und ProSP: Funktionsbeschreibung" aufgelistet. Ein Syntax–Fehler bricht die Analyse eines Event Mapping Ausdrucks ab und läßt diese scheitern.

7 Testumgebung

Um die Instrumentierung einer Targetsoftware mit Simple Statement Events testen zu können, wurde eine sehr einfache Testumgebung aufgebaut.

Ein – am VTA–Host – laufendes, interaktives Programm erlaubt die Eingabe von Ausdrücken gemäß der im Abschnitt 6 vorgestellten Event Mapping Syntax. Diese werden übersetzt und der resultierende Code auf ein ASCII–File geschrieben. Dieses ASCII–File dient dem Sourcecode–Level Debugger (XRAY+) der Target–Entwicklungsumgebung als Commandfile und enthält XRAY–Kommandos zum Setzen und Modifizieren von (Target) Memory–Adressen. Der Debugger liest das Commandfile und lädt damit den Event Code und den Patch Code ins Targetmemory. Ein weiteres Commandfile lädt den Code für den Traphandler und setzt den Trapvector auf diesen Handler. Danach wird noch ein kleines Programm als Minimalst–Version eines VTA–Targets geladen.

Nun kann die Targetsoftware gestartet werden. Alle auftretenden Events werden sequentiell in einem freien Memorybereich des Targets abgelegt. Ist die Exekution der Target–Software beendet, kann der Mini–Monitor gestartet werden, der die aufgetretenen Events an der Konsole des Targetsystems protokolliert. Da der Mechanismus der Event Buffer Anforderung sehr einfach ist, kann als Target programm nur ein sequentiell ablaufendes Programm verwendet werden.

8 Ausblick

Mit den vorgestellten Methoden ist es nun möglich, Simple Statement Events im Targetcode einzuhängen. Einige Punkte seien noch erwähnt, die bei der künftigen Verwendung des Instrumentierungs–Moduls für den VTA (insbesondere des ProSP) von Bedeutung sein werden.

1. Derzeit ist es nicht möglich, ein Event im Kontext eines bestimmten Tasks auftreten zu lassen. Ein Event kann immer nur in einem Node auf eine bestimmte Adresse (Statement) gesetzt werden. Ob ein Event im Kontext eines bestimmten Tasks auftritt, läßt sich nur dynamisch zur Laufzeit ermitteln. Ein Vorschlag zur Lösung dieses Problems (um der Target–CPU möglichst wenig Arbeit aufzubürden) wäre, bei Angabe eines Tasknamens (Task–ID) beim Setzen eines Events Befehle zu generieren, die die Task–ID in den Event Buffer kopieren, in dessen Kontext das Event nun tatsächlich auftrat. Das VTA–Target kann aufgrund dieser Task–ID feststellen, ob das Event nun im Kontext des gewünschten Tasks auftrat oder nicht.

2. Bei der Besprechnung über die Ausführung des Patched Code und dem Exit aus dem Event Handler wurde ein Problem angeschnitten, das bei einem JSR oder einem TRAP auftritt. Es geht dabei um Nebeneffekte, die beim Löschen eines Events auftreten können. Bei näherer Betrachtung weitet sich dieses Problem allgemein auf das Löschen von gesetzten Events aus. Einem Task, der gerade einen Event Code exekutiert, kann währenddessen (durch das Scheduling) die CPU weggenommen werden, wodurch er im Prinzip beliebig lange im Ready–Modus verweilen kann. Ein Event (respektive der Event Code) darf also nur gelöscht werden, wenn kein Task den Event Code gerade exekutiert. Die Überprüfung dieser Bedingung ist nicht trivial und muß noch gelöst werden.

3. Derzeit werden vom Event Handler die Register A0 und A1 für interne Be rechnungen verwendet. Nun werden diese Register aber auch vom Compiler sehr häufig dazu verwendet, Target–Variablen aufzunehmen. Sollen diese Variablen bei einem Event kopiert werden, so müssen ihre Werte vom Stack geholt werden (siehe dazu auch Kapitel 3.3.3). Es wäre also aus Gründen der Code–Optimierung günstiger, weniger oft verwendete Register (etwa A4 und A5) im Event Code zu verwenden. Somit müßte weniger oft auf die geretteten Werte am Stack zurückgegriffen werden.

4. Wenn die Sources der Target–Software mit der Option "Code–Optimierung" compiliert wurden, können unter anderem folgende Probleme auftreten:
   a. Durch die Optimierung kann der Compiler die Struktur von Schleifen verändern, ganze Statements wegoptimieren oder zusammenlegen, sodaß in manchen Fällen der Zugriff auf ein Statement nicht mehr wie gewohnt erfolgen kann.
   b. Wenn zur Optimierung Register herangezogen werden, so kann es vorkommen, daß eine Stackvariable kurzfristig (etwa in einer Schleife) in

ein Register kopiert wird und daher der Wert am Stack nicht erneuert wird. Ein Zugriff auf diese Variable über den Stack führt daher zu falschen Werten.