Lösungsskizze 5 : Entwurf eines Disassembler

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Übersicht

Das folgende Handout skizziert den Entwurf eines Disassembler für einen 8-Bit-Mikroprozessor von der Art des 6502. Implementationssprache ist C/C++[1].

C und C++ sind relativ niedrige Sprachen. Statt eines Modulsystems kommen ein Präprozessor, Headerdateien und Funktionsprototypen zum Einsatz. Die Speicherstruktur der unterliegenden Maschine ist noch deutlich sichtbar und das Speichermanagement muss explizit vom Programmierer der Anwendung vorgenommen werden. C und C++ sind nicht die einzigen Sprachen mit diesen Eigenheiten, denn die wenigsten Sprachen aus dem zeitgenössischen industriellen Mainstream besitzen ein klares Modulkonzept, und das Problem der Notwendigkeit eines expliziten Speichermanagement plagt auch Sprachen der Pascal- und Modulafamilie, soweit sie nicht über eine Garbage-Collection wie Oberon-2 oder Modula-3 verfügen.

In solchen Fällen muss muss man sich Gedanken machen, wie man das ursprüngliche *ideale* Konzept etwa durch einen *Coding Style Guide* wieder zurück in die reale Sprache bringen kann. Oder anders ausgedrückt: Wie man das Vorgehen bei der Implementation in der realen Sprache so strukturieren kann, dass man nicht allzusehr von den Problemen geplagt wird, die sonst aus den Beschränkungen der Sprache resultieren.

Wir haben so etwas schon einmal getan, und zwar für das Modulkonzept: In dem hier verwendeten *C mit Namespaces* entspricht ein Modul genau einer Objektdatei in einer dedizierten Quelltextdatei, die den Namen des Moduls trägt, und deren Schnittstelle in einer analog benannten Headerdatei beschrieben ist (Siehe Implementation des FiFo in TeachSWT@Tü, 2002b, *Lösungsskizze 2)*. Um Namenskonflikten vorzubeugen – auch das war ein Anliegen des Modulkonzeptes – werden alle Deklarationen und Definitionen von Konstrukten in Header und Implementation in einen Namensraum gekapselt, der wiederum den Namen des Moduls trägt (siehe auch Abbildung 2). Dieses Vorgehen emuliert in weiten Teilen das, was in Sprachen mit einem vollständigen Modulkonzept (Ada, Modula, OCaml) der Compiler selbsttätig erledigt (siehe TeachSWT@Tü, 2002c, *Module und Namensräume)*. Der folgende Entwurf hält sich an das eben beschriebene Verfahren.

Es ist aber darüberhinaus noch notwendig sich über zwei weitere Fragen Gedanken zu machen. Zum Einen besteht ein Bedarf, die tatsächliche Implementation einer Datenstruktur vor dem Programmierer eines Klientenmoduls zu verbergen. Man nennt dies eine *opake Typdefinition*. Exportiert und in der Interfacedefinition sichtbar werden sollten im Idealfall nur die Typnamen, aber keine Information über die Struktur der Darstellung, um unbeabsichtigtes Einfließen von Wissen darüber in die Implementation der Klientenmodule zu verhindern. Wir werden in einem der folgenden Abschnitte abklären, inwieweit das überhaupt in der Sprache C möglich ist.

Zum Anderen sind in der idealen Welt der Spezifikation abstrakter Datentypen die Daten einfach Elemente von (Werte-) mengen, welche von Funktionen in andere Daten – wiederum Elemente aus Wertemengen – abgebildet werden. Der Typ steht dabei einfach für die Wertemenge aus der ein Wert in einer gegebenen Rolle stammen kann. In der realen Welt der meisten prozeduralen und objektorientierten Sprachen dagegen müssen Daten irgendwo gespeichert werden, und nur die Werte weniger größenbeschränkter und einfacher strukturierter Typen können eine vorübergehende Existenz als Zwischenergebnisse auf dem Stack führen. Für die Speicherung von komplexen Daten muss meist ein Speicher explizit belegt werden. Wir müssen uns also bei der Implementation von Operationen auf komplexeren Datentypen, wenn wir in C implementieren, immer Gedanken machen, welcher

[1] Um genau zu sein, eigentlich *C mit Namespaces*, die objektorientierten Konstrukte aus C++ kommen nicht zum Einsatz.

Speicher benötigt wird, wann man Speicher belegt und wann er wieder freigegeben werden muss um Speicherlecks zu vermeiden. Hierbei lassen sich einige wenige typische Arten, mit Speicher an der Schnittstelle von Operationen umzugehen, identifizieren. Auch dies wird in einem der folgenden Abschnitte dargelegt werden.

1.2 Opake Typdefinitionen in C

1.2.1 Motivation

Wenn opake Typdefinitionen von der Zielsprache unterstützt werden, können sie ein Mittel sein, einen recht häufigen Typ von Kontrakt zum Ausdruck zu bringen, der in etwa so lautet: „Das Providermodul stellt den Typ *X* und gewisse Zugriffsfunktionen (Attribute) und Operationen zur Verfügung. Die Manipulation von Daten des Typs *X* soll nur über Zugriffsfunktionen und Operationen erfolgen. Die Autoren des Providermoduls behalten sich vor, die (dem Providermodul interne) Darstellung von *X* jederzeit zu ändern."

Dies ist eigentlich nichts anderes als das Geheimnisprinzip. Die opaken Typdefinitionen in der Implementation entsprechen den abstrakten Datentypen der Spezifikation: Operationen stehen im Vordergrund, die tatsächliche Darstellung rückt in den Hintergrund und ist im extremen Fall auswechselbar. Man kann aber abstrakte Datentypen (das sind Konstrukte in der Spezifikation) ohne das Mittel opaker Typdefinitionen (das wären Konstrukte der Implementation) implementieren, indem man die Struktur der Daten zwar im Interface offenlegt – weil beispielsweise die Implementationssprache keine andere Möglichkeit lässt – jedoch *per Konvention* verbietet, von dieser Information auch Gebrauch zu machen, und fordert, immer die Zugriffsfunktionen zu verwenden. Deshalb kann die Darstellung trotzdem geändert werden, ohne dass der Quelltext der Klientenmodule geändert werden muss, zumindest solange sich der Programmierer an diese Vereinbarung hält.

Natürlich wäre es besser, wenn die Implementationssprache tatsächlich die Möglichkeit böte, die Information über die interne Darstellung in der Schnittstellenbeschreibung zu unterdrücken, denn dann könnte man den Kontrakt aus dem Quelltext der Schnittstellenbeschreibung ableiten und sogar vom Compiler automatisch prüfen lassen (wie das in Modula-2 und OCaml möglich ist). Auf diese Möglichkeit muss notgedrungen verzichtet werden, wenn die verwendete Implementationsprache dieses Vorgehen nicht gestattet, und man gezwungen ist, auf obige Konvention, d. h. eine informelle Vereinbarung zwischen den Programmierern des Providermoduls und denen des Klientenmoduls zurückzugreifen.

Die Situation in C ist genau die eben skizzierte: Es ist im allgemeinen Fall nicht möglich, die Struktur der implementierten Typen in den Headerdateien vollständig zu verbergen.

Oberstes Ziel einer jeden (per Sprachmechanismus oder Konvention) opaken Typdefinition und -implementation muss sein, die Illusion aufrecht zu erhalten, dass Werte (Elemente aus Mengen) manipuliert werden und dabei andere Werte (Elemente aus Mengen) erzeugt werden. Dabei muss soweit als möglich verborgen werden, dass dafür u. U. Speicher belegt werden muss. Dies ist eine Konsequenz des Geheimnisprinzips, da Informationen über die Speicherstrategien sehr viel über die interne Darstellung verraten, und ihre Änderung häufig eine Änderung des Klientenmoduls erfordern würde.

Es werden nun drei typische Anwendungsmuster gezeigt, wie dies in C geschehen kann.

Aus Gründen der Entwicklungsgeschichte diese Dokuments und um in Übereinstimmung zu bleiben mit dem abgedruckten Quelltext, werden opak definierte Typen in den folgenden Abschnitten häufig als „abstrakte Typen" bezeichnet. Dies sollte sich in einer der künftigen Ausgaben dieses Textes ändern. Bis dorthin besteht eine gewissen Verwechslungsgefahr mit den abstrakten Datentypen aus der Spezifikation.

1.2.2 Abstrakte Wertetypen

Im Idealfall kann ein definierter Typ *T*, wie die vordefinierten Typen *float*, *int* usw., eine temporäre Existenz als Zwischenergebnis auf dem Stack führen, mit dem Zuweisungsoperator „=“ in einer Variable gespeichert werden und mit *return* als Funktionsergebnis zurückgegeben werden, wie etwa in dem folgenden C-Quelltext-Fragment:

```
T foo( ... ){
  T t1;
        ...
  return t1;            /* return value */
}

void bar( ... ){
  T t2;
        ...
  t2 = foo( ... );      /* assign value */
        ...
  baz(t2, foo( ... ));  /* intermediate result, temporary value */
        ...
}
```

In C ist dies nur möglich, wenn entweder einer der erwähnten vordefinierten Typen redefiniert wird, oder die Darstellung von *T* durch eine *struct* erfolgt, die ihrerseits *keine* Zeiger enthält. Beispielsweise ist für komplexe Zahlen die folgende Definition eine solche Darstellung, wenn auch nicht die einzig mögliche:

```
typedef struct{
          float re;
          float im; } complex;
```

In dem Entwurf, der in den späteren Kapiteln vorgestellt werden wird, werden solche Typen als *abstrakte Werte* bezeichnet. In den Headerdateien sind sie durch ein Makro *ABSTRACT_VAL_* markiert,

```
typedef ABSTRACT_VAL_(
          struct{
            float re;
            float im; } ) complex;
```

um anzuzeigen, dass der Klientenprogrammierer die angegebene Struktur ignorieren soll. Das Makro selbst expandiert lediglich zu seinem Argument.

Dieses Makro und die in den beiden folgenden Abschnitten erwähnten Makros *ABSTRACT_VAR_* und *ABSTRACT_HEAP_* sind kein „Standard“, in welchem Sinn auch immer, sondern Teil des Disassembler-Projektes und in der Datei *MARKUP.h* definiert.

1.2.3 Abstrakte Variablentypen

Nicht immer können Typen so implementiert werden, dass die im vorangegangenen Abschnitt angesprochene „naive“ Verwendung möglich ist. Benötigt der Datentyp Zeiger für seine interne Darstellung, wird er also durch mehrere miteinander verzeigerte Datenblöcke dargestellt, wie z. B. die folgende Implementation einer Zeichenkette,

```
typedef  struct{
           char*   data;
           size_t  datalen;
           size_t  bufferlen; } mystring;
```

so würde eine Duplikation von Speicherinhalten mittels *return* oder dem Assignmentoperator zu einem Überlappen der Darstellungen führen. Die *data*-Zeiger von Original und Kopie würden auf denselben Speicherblock zeigen, da der Assignmentoperator in C wenig über die Struktur des Typs weiß und noch weniger über die Struktur der Abstraktion die damit dargestellt werden soll und blind den Speicherinhalt einer entsprechenden Scratch-Area auf dem Stack oder einer Variablen in den Speicherbereich der Zielvariablen kopiert (*shallow copy*). Was stattdessen angemessener gewesen wäre, wäre eine tiefe Kopie (*deep copy*) der Datenstruktur gewesen. Es hätte nicht der Zeiger *data* dupliziert werden sollen, sondern die Inhalte, auf die der Zeiger verweist.

Aber es kommt noch schlimmer: Bei dem – vom Compiler nicht verhinderten – Gebrauch eines Resultats dieses Typs als temporäres Zwischenergebnis würde ein Speicherleck entstehen, da natürlich nur die *struct* selbst auf dem Stack (in einer Scratch-Area) gelagert und auch wieder automatisch freigegeben würde, der Speicher auf den *data* verweist, aber beim Abbau des Stacks nicht berücksichtigt würde. Mit der nach dem Abarbeiten des Ausdrucks deallozierten Scratch-Area ginge auch der letzte Verweis auf diesen Speicher verloren.

In Konsequenz der Erkenntnis, dass es so nicht geht, muss für Typen, deren Darstellung Zeiger enthält, die Möglichkeit eines temporären Zwischenergebnisses auf dem Stack aufgegeben werden. Man muss fordern, dass Werte dieser Typen nur in Variablen (genauer gesagt *lvalues* vorkommen dürfen, also Speicherplatz für sie explizit durch Deklaration lokaler Variablen auf dem Stack oder durch Allokation auf dem Heap reserviert werden muss. Ehe eine solche Variable aus dem Scope geht, bzw. ehe der Heapspeicher, der einen solchen Wert enthält, freigegeben wird, muss er deinitialisiert werden, was dem Modul, das den Typ implementiert, die Gelegenheit gibt, den verborgenen allozierten, für den Programmierer des Klientenmodul nicht sichtbaren Speicher freizugeben.

Für diesen Zweck muss die Implementation eine Prozedur (etwa mit einem Namen wie *deinit*) zur Verfügung stellen. Da die Zuweisung mittels des Assignment-Operators aus den erläuterten Gründen nicht möglich ist, muss eine Prozedur zum Kopieren der (scheinbaren) Werte von einer Variable in die andere implementiert werden. Diese kann einen Namen wie *copy* tragen, und wird gegebenenfalls eine tiefe Kopie der Datenstruktur anfertigen, oder Referenzzähler für die überlappenden Teile der Darstellung verwalten. Darüberhinaus ist zu empfehlen, dass man eine weitere Prozedur (mit einem Namen wie *move*) zur Verfügung stellt, welche den Wert einer Variablen in eine andere Variable überträgt, und dabei die Ursprungsvariable uninitialisiert hinterlässt. Dies ist in vielen Fällen effizienter möglich, als die Anfertigung einer Kopie.

In einige Fällen legt die Pragmatik eines Typs (d. h. wie der Typ aus Konvention verwendet wird) aber auch nahe, dass es sinnlos ist, Werte dieses Typs zu duplizieren. Dies ist zum Beispiel der Fall für Stacks und FiFos, ganz allgemein immer für Typen, die als Container dienen – also eher für eine Speicher- und Zugriffsstrategie stehen – oder auch immer dann, wenn von „Zuständen“ (statt von Werten) die Rede ist. In all diesen Fällen wird man wohl keine *copy*- oder *move*-Prozeduren zur Verfügung stellen.

In einer Parameterübergabe beim Funktionsaufruf und durch die Anweisung *return* würden Werte, so wie beim Assignment-Operator, „naiv“ kopiert, d. h. es wird keine tiefe Kopie angefertigt, und es käme zur Überlappung der Darstellungen. Aus diesem Grunde dürfen Werte der gerade beschriebenen Art nur per Referenz an die Prozeduren übergeben werden und ebenso muss die Rückgabe von Ergebnissen so erfolgen, dass eine Referenz auf eine Variable (bzw. einen *lvalue*) geeigneten Typs übergeben wird, in der das Ergebnis dann hinterlegt wird. Eine Operation zum Hintereinanderhängen von Zeichenketten könnte also folgendermassen definiert und angewendet werden:

```
void append( string* source, string* dest );

void foo( ... ){
  string s1;
  string s2;
     ...
  append( &s1, &s2 );      /* means: s1' := s1 + s2 */
}
```

Der Initialisierungszustand von Variablen muss übrigens dann Teil der Vor- und Nachbedingungen werden, da daraus ja gewisse Verantwortungen für den Programmierer des Klientenmoduls resultieren.

Opake Datentypen, deren Implementation der eben vorgeschlagenen Systematik folgen, werden im später folgenden Entwurf als *abstrakte Variablen* bezeichnet, und in den Headerdateien durch das Makro *ABSTRACT_VAR_* gekennzeichnet. Auch hier ist es, wie schon bei *ABSTRACT_VAL_* so, dass ein wirkliches Verbergen der Implementation nicht möglich ist. Das Makro, das wiederum nur zu seinem Argument expandiert, dient auch hier nur als Hinweis an den Programmierer des Klientenmoduls.

Es sei hier noch erwähnt, dass die hier vorgestellte Systematisierung in der Implementation opaker Werte keineswegs Standardpraxis in C ist. Üblich ist, dass der Anwender einer Datenstruktur mit unnötigen Details über Zeiger in den von Anderen gelieferten Strukturen belästigt wird, und häufig nicht recht entscheiden kann, wo eine Struktur beginnt, und wo sie dann aufhört (d. h. ob sein Programm einem Zeiger folgen soll, oder nicht). Das vorgeschlagene Verfahren vereinfacht es jedoch sehr stark Speicherlecks zu vermeiden: Dazu müssen nur Variablen deinitialisiert werden, ehe sie aus dem Scope gehen, und dies wiederum ist eine Tatsache, von der man sich meist leicht durch eine statische Inspektion des Quelltextes überzeugen kann[2]. Darüberhinaus ist die dadurch entstehende Begrifflichkeit der in den formalen Methoden verwendeten ähnlicher, erleichtert also Spezifikation vor Implementation.

Dass das vorgeschlagene Verfahren nicht vollkommen aus der Luft gegriffen ist, erkennt man, wenn man einen Blick auf das Operator-Overloading in C++ wirft: Dort gibt es die Möglichkeit, Assignmentmethoden und Destruktoren für Objekte zu definieren, die dann auch automatisch aufgerufen werden, wenn eine Zuweisung mit dem Zuweisungsoperator „=“ erfolgt, bzw. wenn das Objekt aus dem Scope geht. Bis auf diese Automatik sind diese Methoden nichts anderes als die gerade vorgestellten *copy*- und *deinit*-Prozeduren. Ein

[2] Der entscheidende Punkt ist hier der restriktive Umgang mit Zeigern: Zeiger werden (sollen!) in Klientenprozeduren nur beim Aufruf im Stil Pass-by-Reference erzeugt werden. Prozeduren, die diese Zeiger erhalten, sind nicht für die Freigabe verantwortlich, da bekannt ist, dass es sich um Zeiger auf lokale Variablen aufrufender Prozeduren handelt. Die Verantwortung der Klientenprozeduren eines typimplementierenden Moduls beschränkt sich nur darauf, lokale Variablen vor dem Verlassen der Scope zu deinitialisieren, was eine verhältnismäßig leichte Verpflichtung ist. Dafür entsteht ein gewisser Aufwand in den typimplementierenden Modulen, da sie natürlich die Illusion unabhängiger Werte durchgehend und nach Möglichkeit auch noch effizient aufrechterhalten sollen. Aber genau dafür – für die Kapselung von Komplexität, und die entsteht häufig aus der Speicherverwaltung – ist die Modularisierung da.

Analog zur *move*-Prozedur fehlt allerdings, was vermuten lässt, dass Wertabstraktionen wie die Klasse *String* in C++ immer zu einer gewissen Ineffizienz verdammt bleiben werden.

Darüberhinaus tut die übliche Literatur zu C++ (allen voran das famose Buch von Bjarne Stroustrup: Stroustrup, 1998) ihr Bestes, den Sinn und Zweck dieser Mechanismen (eben: Wertabstraktion) bis zur Unkenntlichkeit zu verschleiern.

1.2.4 Abstrakte Heapvariablen

Eine Alternative zu den abstrakten Variablen ist es, den gewünschten Typ in Headerdateien als eine benannte *struct* zu definieren, die ihrerseits aber nur im Implementationsmodul definiert wird.

```
typedef struct _T_ T;
```

Das bedeutet, dass der Compiler beim Übersetzen der Klientenmodule keine Information über die Zusammensetzung von *struct _T_* erhält, also auch keine Informationen über die Größe von *struct _T_* hat. Eine solche Situation ist in C legitim, und kann dazu verwendet werden, wirklich opake Typen zu definieren, deren Struktur in der Headerdatei nicht offengelegt wird.

Dafür ist allerdings ein Preis zu entrichten: Wegen der fehlenden Größeninformation kann in den Klientenmodulen keine Variable des Typs T definiert werden, denn sowohl für die Erzeugung des Stackframe einer Prozedur als auch für die Unterbringung von Variablen im statischen Datensegment muss deren Größe zur Übersetzungszeit bekannt sein. Zeiger auf den Typ *T* können jedoch verwendet werden, aber es kann kein Speicher auf dem Heap alloziert werden, um Zeiger *T** zu erhalten – wiederum wegen der fehlenden Größeninformation.

Das Klientenmodul muss sich mit den Heapobjekten zufrieden geben, die das Providermodul in entsprechenden Prozeduren alloziert. Für diesen Zweck würde das Providermodul dann eine oder mehrere *new*-Prozeduren zur Verfügung stellen, ebenso müssen Prozeduren zur Freigabe der betreffenden Heapobjekte zur Verfügung gestellt werden. Letzteres kann das Klientenmodul nicht einfach durch einen Aufruf von *free*() erledigen, da zum einen möglicherweise implizit allozierter Speicher freigegeben werden muss, zum anderen das Providermodul auch eigene Allokatoren zur Reservierung des Speichers verwendet haben könnte.

Im Weiteren können opake Typen der eben skizzierten Art in ähnlicher Weise implementiert werden, wie die *abstrakten Variablen* des letzten Abschnitts: Es sollten *copy*- und *move*-Prozeduren zur Verfügung stehen, um den Inhalte aus einem Heapobjekt ins andere zu übertragen.

Der Vorteil der so definierten Typen ist, dass ihre Definition wirklich und nicht nur per Konvention opak ist, d. h. die Abschirmung des Klientenmoduls von Implementationsdetails durch den Compiler erzwungen wird. Der Nachteil andererseits ist, dass Zeiger eine viel herausragendere und weniger beschränkte Rolle spielen – mit allen Konsequenzen, die das für gewöhnlich so hat, d. h. die Verantwortlichkeit des Programmierers für die rechtzeitige Freigabe des Speichers, das Aliasing von Speicherobjekten und die Duplikation von Zeigern, was es – manchmal gewollt – erst möglich macht, dass die Lebensdauer von Daten nicht mehr eng an die Scopes des Programmes angelehnt ist.

Im vorliegenden Entwurf werden Typen, die in der gezeigten Art implementiert sind, als *abstrakte Heapvariablen* bezeichnet und in den Headerdateien mit dem Makro *ABSTRACT_HEAP_* gekennzeichnet, das in diesem Fall außerdem einen Namen für die zu implementierende *struct* generiert:

```
typedef ABSTRACT_HEAP_( T );
```

1.3 Umgang mit Speicher an den Schnittstellen

1.3.1 Das Problem.

Wenn an einer Schnittstelle (einem Prozeduraufruf) Referenzen auf Speicher übergeben werden, muss klargestellt sein, wo die Verantwortung für die Freigabe des Speichers liegt, und ob und unter welchen Umständen der Zeiger lesend oder schreibend dereferenziert werden darf: Wird hier etwa nur auf einen Speicherplatz verwiesen, dem Eingabedaten entnommen werden sollen? Wird hier auf Speicherplatz verwiesen, in den Resultate gespeichert werden sollen? Oder wird der Speicherplatz dem Providermodul gewissermassen übereignet, liegt die Verantwortung für die Freigabe bei einem Mechanismus des Moduls?

Wenn ein Zeiger zurückgegeben wird: Soll hier auf ein Ergebnis verwiesen werden? Oder wird der hinter dem Zeiger stehende Speicher an die aufrufende Prozedur übereignet und diese steht dann in der Verantwortung, diesen zur gegebenen Zeit freizugeben?

Es ist wohl angebracht, die möglichen Antworten auf diese Fragen etwas zu systematisieren. Im Großen und Ganzen lassen sich einige wenige und typische Arten des Umgangs mit Speicher an Schnittstellen identifizieren, die fast alle Anwendungsfälle abdecken.

1.3.2 Pass by Reference.

Der einfachste Fall ist der, dass der übergebene Zeiger lediglich auf Speicher verweist, der Eingabedaten enthält, und bzw. oder in den Ausgabedaten geschrieben werden sollen. Der Prozeduraufruf sähe in diesem Fall so aus:

```
some_procedure( &some_lvalue, ...);
```

Die aufgerufene Prozedur wird in solchen Fällen zwar Speicherinhalte lesen und möglicherweise verwenden – sie wird zu diesem Zweck auch den Zeiger dereferenzieren, aber sie wird (und darf) keine Notiz nehmen vom tatsächlichen Wert des übergebenen Zeigers, ihn duplizieren, oder irgendwohin speichern.

Dieser Fall der Übergabe von Eingaben durch Verweise und des Verweises auf Speicher für das Ablegen von Resultaten ist bekannt unter dem Namen *Pass by Reference*[3]. Er ist so häufig, dass dafür in den Programmiersprachen Pascal und Modula ein eigenes syntaktisches Konstrukt existiert – die Var-Parameterdeklaration. Da in diesen Sprachen ein Adressoperator wie „&“ nicht existiert und das Dereferenzieren des Verweises auf einen Var-Parameter in der aufgerufenen implizit ist, verhindert dieses Konstrukt, dass die aufgerufene Prozedur so programmiert wird, dass sie sich den übergebenen Zeiger explizit verschafft: Sie kann ihn nur (implizit) verwenden, aber nicht kopieren.

Im Fall von Pass by Reference wird man angeben müssen, ob und unter welchen Umständen der Speicher auf den verwiesen wird, vor oder nach dem Aufruf mit gültigen Daten (im Sinne einer bestimmten Abstraktion) initialisiert ist, und ob auf diesen Speicher lesend oder schreibend zugegriffen wird.

Der nächstkompliziertere Fall ist der, in dem Speicher außerhalb eines Moduls belegt, dann an das Modul übergeben wird, und dessen weiteres Schicksal in die Verantwortung des Moduls übergeht. Das kann im Wesentlichen 2 Fälle bedeuten: Einmal, dass das Modul die benötigten Daten aus dem betreffenden Speicher liest und dann den Speicher wieder freigibt, und das andere Mal, dass das Modul den übergebenen Speicher – d. h. eigentlich den Zeiger auf diesen Speicher – in Datenstrukturen integriert, die von diesem Modul verwaltet werden – dazu zählen dann auch die entsprechenden opaken Typen – und den Speicher

[3] Begrifflich komplementär dazu ist *Pass by Value*, wo tatsächlich die Daten selbst auf den Stack kopiert werden.

dann zur gegebenen Zeit – beim Deinitialisieren der Datenstrukturen – wieder freigeben muss.

1.3.3 Zeiger als Ergebnisse.

Schließlich gibt es noch die Fälle, in denen ein Modul einen Zeiger erzeugt oder aus einer von ihm verwalteten Struktur entnimmt, und an die aufrufende Prozedur zurückliefert. Hier kann die Verantwortung für die Freigabe des Speichers bei der aufrufenden Prozedur liegen, etwa, wenn das Modul Speicher alloziert, um ein Ergebnis zu speichern. Dies ist der einfachere Fall.

Der komplizierter Fall ist der, in dem das Modul als Ergebnis einen Zeiger auf Teile der von ihm verwalteten Datenstrukturen zurückliefert. Hier wäre es die bessere Lösung, eine Kopie der betreffenden Daten anzufertigen, aber wenn die Datenstrukturen recht umfangreich sind, ist das unter Umständen aus Gründen der Effizienz nicht möglich. Die aufrufende Prozedur wird auf einen solchen Zeiger nur lesend zugreifen dürfen. Weiterhin wird man, da das Modul die Datenstruktur in die der Zeiger verweist, irgendwann verändern wird, angeben müssen, wie lange der erhaltene Zeiger eine gültige Methode ist, das Ergebnis auszulesen. Die beiden häufigsten Fälle dürften sein, dass der Zeiger genau bis zum nächsten Aufruf einer Prozedur des Moduls gültig bleibt, oder dass er gültig bleibt, bis ein bestimmter Wert, eine bestimmte Datenstruktur verändert oder freigegeben wird. Wie man sehen kann, tritt im letzteren Fall eine Koppelung zwischen der Gültigkeit eines Zeigers und der Lebenszeit eines bestimmten Wertes auf.

Man kann nun zunehmend kompliziertere Fälle solcher Lebensdauerkoppelungen konstruieren (und damit dem Klientenmodul immer schwierigere Verpflichtungen für den Umgang mit den erhaltenen Zeigern aufbürden), aber das ist natürlich unerwünscht. Auch in dem eben genannten Fall möchte man die Koppelung eigentlich vermeiden, sie muss aber in einigen Fällen aus Effizienzgründen in Kauf genommen werden. Alle anderen Fälle von Verschränkung der Lebensdauern von Strukturen und der Gültigkeit von Zeigern können meines Erachtens (fast?) immer vermieden werden.

1.3.4 Zeiger als Tokens.

Schließlich und zuletzt gibt es noch den einfachen Fall, in dem ein Modul zwar Zeiger als anonyme Tokens verwaltet (also beispielsweise in eine Warteschlange stellt und wieder zurückliefert) ohne sie als Zeiger zu derefenzieren.

Aus dieser Fall muss natürlich dokumentiert werden, da Zeiger in der Schnittstellenbeschreibung erscheinen, und sich die Frage stellt, wo die Verantwortung für die Freigabe von Speicher bleibt: Hier liegt sie vor und nach dem Aufruf der Prozeduren beim Klientenmodul, das sich dann noch immer darüber im klaren sein muss, ob ein Zeiger noch im Providermodul gespeichert (dupliziert) ist, ehe es den Speicher, auf den der Zeiger verweist, freigeben kann.

1.3.5 Verallgemeinerungen.

Die Überlegungen der vorangegangenen Abschnitte lassen sich auf alles (alle Daten) verallgemeinern, die als Verweis auf eine anderswo oder extern allozierte Ressource dienen. Dateideskriptoren beispielsweise müssen nach Gebrauch geschlossen werden (da auch sie eine beschränkte Ressource sind), also muss auch, wenn Dateideskriptoren an eine Prozedur übergeben werden oder von einer Prozedur zurückgegeben werden, geregelt werden, wo die Verantwortung für das Schließen des Dateideskriptors bleibt.

Die Nichteinhaltung solche Regeln hat Lecks in den Programmressourcen zur Folge, bei längerer Laufzeit mit fatalen Ergebnissen. Die Beschreibung einer Schnittstelle kann erst als vollständig gelten, wenn eine statische Inspektion des Programms zusammen mit der Beschreibung *nur* der Schnittstelle eines Moduls geeignet ist, solche Fehler im Gebrauch der Programmressourcen aufzudecken. Daraus folgt, dass eine Schnittstellenbeschreibung auch Verantwortungen für den Umgang mit Ressourcen zu regeln haben, die diese Schnittstelle überqueren, oder präziser: Wenn Verweise auf Ressourcen dupliziert und über eine Schnittstelle weitergegeben werden.

2 Zum Stand des Entwurfs

Der vorliegende Entwurf ist möglicherweise noch nicht vollständig und wurde keiner ausführli-chen Review unterzogen. Die letzte vorgenommene oberflächliche Prüfung hat einige (nicht sinnentstellende) Fehler und etliche Inkonsistenzen aufgedeckt, so dass zu erwarten ist, dass noch weitere enthalten sind. Notation und Namensgebung sind zum Teil uneinheitlich, und die verwendete Nomenklatur nicht selten ungeschickt. All dies kann erst in einer zukünftigen Version ausgebessert werden.

3 Abstrakte Datentypen: Ein Ansatz

Betrachtet man das Datenflussdiagramm des Disassemblers (Lösungsskizze 8), so findet man, dass sich sehr viele Verarbeitungsschritte um *Opcode[s]* zentrieren. In der Tat sind *Opcode[s]* und *Adr.-Mode[s]* die einzigen Daten, die als Schlüssel für die Suche in den Tabellen dienen, welche den Prozessor charakterisieren und die Disassemblierung treiben. Der Verdacht liegt nahe, dass die Tabellen *Eigenschaften* der Abstraktionen beschreiben, die hinter den Daten stehen, welche als Zugriffsschlüssel verwendet werden.

Dies ist der Ansatz den wir verfolgen werden: Wir werden *Opcode[s]* und *Adr.-Mode[s]* zu abstrakten Datentypen des Programms machen, und die Suche in den Tabellen als Funktionen definieren, welche Attribute der Werte des ADT zurückliefern.

Weitere Kandidaten für abstrakte Datentypen ergeben sich aus dem Bestreben über die Eingabedaten zu abstrahieren, d. h. die Darstellung der Daten auf dem Speichermedium und den genauen Zugriffsmechanismus für den eigentlichen Applikationskern zu verbergen. Analoges gilt für die Ausgabe: Auch diese sollte soweit abstrahiert werden, dass sie in Begriffen beschrieben wird, die dem Problem nahestehen, anstatt an die physikalischen Charakteristika des tatsächlich verwendeten Ausgabegeräts angelehnt zu sein.

Das Modul, welches die problemnahe Abstraktion der Ausgabeoperationen in gerätespezifische Operationen übersetzt, ist im weitesten Sinn des Wortes ein *Druckertreiber*.

Hätten wir ein Programm, dessen Aufgabe darin besteht, einen komplexen inneren Zustand zu verwalten (wie es zum Beispiel ein Editor tut, s. EFASS), dann wäre dieser innere Zustand ein weiterer Kandidat für die Abstrahierung und Zerlegung in abstrakte Datentypen.

4 Verwendungsbeziehungen zwischen Modulen

Bei kreativer Anwendung der oben erläuterten Prinzipien kann man zu der in Abbildung 1 gezeigten Modulstruktur kommen.

Die grundlegende Struktur wurde so angelegt, dass zuerst in einem Schritt die Eingabedaten eingelesen und als abstrakter Datentyp verfügbar gemacht werden. Es wird also bereits in diesem Schritt über die Eingabe abstrahiert (Eingabe-Tier). Im zweiten Teil des Programms ist die eigentliche datenverarbeitende Leistung – die Disassemblierung – lokalisiert. Die meisten Einzelschritte der Verarbeitung werden als Aufsuchen von Attributen abstrakter Entitäten (damit meine ich jetzt die Elemente abstrakter Datentypen) realisiert., lediglich im Modul *disassembly* sind offenkundig einige Abläufe beheimatet. Der Ausgabe-Tier schließlich stellt ein dem Problem angemessenes abstraktes Ausgabegerät zur Verfügung, abstrahiert also über die hostsystemspezifischen Mechanismen der Ausgabe.

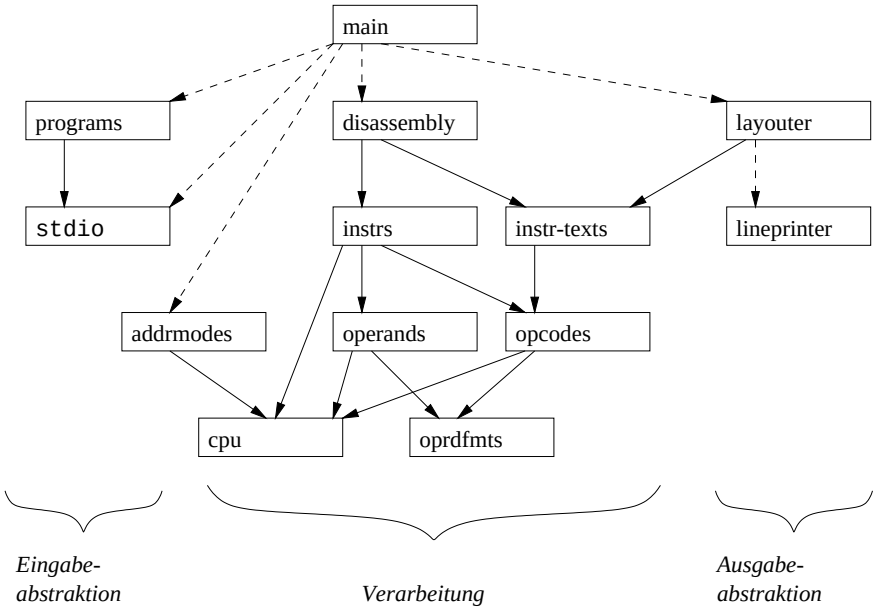

Abbildung 1: Modulstruktur des Disassemblers

Dem EVA-Prinzip ist insofern Rechnung getragen, als Ein- und Ausgabe sorgfältig zeitlich und nach Modulen getrennt [4] sind. Vor allem erfolgt nicht alle Verarbeitung stückchenweise in derselben Schleife, welche die Eingabedaten einliest – eine Versuchung, die für die vorliegende Aufgabe durchaus naheliegt, der man jedoch wiederstehen sollte, da dies das Programm wahrscheinlich in einen amorphen Brief spezialisierte Prozeduren verwandeln würde.

In Abbildung 1 sind zwei verschiedene Arten von Benutzungsbeziehungen unterschieden. Diejenigen, die bereits in den Schnittstellenbeschreibungen offenliegen, sind mit durchgezogenen Linien notiert. Die unterbrochenen Linien zeigen eine Benutzungsbeziehung an, von der jetzt schon absehbar ist, dass sie für die Implementation des betreffenden Klientenmoduls nötig oder gewollt ist. Das bedeutet, dass der nötige Import bzw. Einschluss von Schnittstellenbeschreibungen bzw. Headerdateien des Providermoduls in der Implementation des Klientenmoduls geschieht, nicht schon in seiner Schnittstellenbeschreibung. Die Benutzungsbeziehungen, die vom Hauptmodul ausgehen, zählen praktisch per Definition zu dieser Sorte, da das Hauptmodul keine Schnittstellendefinition hat.

Was die anderen durch die Implementation nötig werdenden Benutzungsbeziehungen betrifft, ist das Diagramm in Abbildung 1 alles andere als vollständig. Das hat durchaus seine Richtigkeit, da man in den ersten Phasen des Entwurfs nicht alle Aspekte regeln will. Viele dieser Benutzungsbeziehungen sind erst in späteren Phasen des Entwurfs abzusehen, wenn die Einzelheiten, Machbarkeit und Vorrausetzungen für die Implementation bestimmt werden. Darunter fällt zum Beispiel, dass für das Modul *lineprinter* kein Systemmodul angegeben ist, auf das sich jenes Modul abstützt, da diese Entscheidung bis hierher bewusst offen gelassen ist. Weiter ist zum Beispiel zu erwarten, dass in späterem Phasen des Entwurfs weitere Module dazukommen, die Mechanismen für die Implementation einer oder mehrerer Klientenmodule zur Verfügung stellen, beispielsweise Hash-Tabellen, auf denen die

[4] Das macht natürlich nur Sinn, wenn z. B. die Eingabetätigkeit auch mit einer entsprechenden Abstraktionsleistung verbunden ist, was aber meines Erachtens immer ratsam ist. Hätte das Eingabeverfahren nur aus einer expliziten Einleseschleife bestanden, statt eines abstrakten Datentyps, der die Details der Eingabe verbirgt, wäre es selbstverständlich sinnlos gewesen, sie in ein eigenes Modul auszulagern.

für die Attributfunktionen in den Modulen *operands* und *addrmodes* nötigen Zuordnungen implementiert werden könnten.

Die Namen von Systemmodulen sind in Abbildung 1 übrigens zur Unterscheidung in Schreibmaschinenschrift gehalten.

Bei alle dem muss betont werden, dass dieser Strukturierung der Anwendung ein gewisses Willkürelement innewohnt. Zum Beispiel ließen sich viele kleinere Module (*instrs*, *operands*, *opcodes* und *oprdfmt*) aus dem Verarbeitungs-Tier zu größeren Modulen zusammenfasse (zu einem hypothetischen Modul *instructions*). Dies würde zwar auf der Ebene der Modulbeziehungen die Komplexität etwas verringern helfen, aber auch die Gefahr einer unkontrollierten Diffusion von Funktionalität und Zuständigkeiten innerhalb des größeren Moduls bedeuten.

Besonders die Rolle des Moduls *disassembly* – das nur als Container für einige lose assoziierte Prozeduren dient, kann mit Recht kritisiert werden. Möglicherweise hätte man diese Funktionalität gleich ins Hauptmodul *main* integrieren sollen, oder als weitere Abstraktion auf den Eingabe-Tier aufgesetzen sollen[5].

Es ist die spezifische Leistung des Softwarearchitekten im Rahmen der angesprochenen Wahlfreiheit und den Regeln guter Praxis folgend, eine angemessene Struktur (Architektur) für ein Softwareprodukt auszuwählen, welche ergonomische und ökonomische Kriterien des Kunden und künftiger Maintainer gegeneinander balanciert. Es ist andererseits sein Versagen, wenn dies nicht gelingt.

[5] D. h. ein Programm wäre durch einen ADT als Folge von Instruktionen präsentiert worden.

```
#ifndef H_INCLUDED_foobar_HH
#define H_INCLUDED_foobar_HH

#include "MARKUP.h"

/* ... #include headers of all
                necessary provider modules */

namespace foobar {

  /* ... definitions and
         declarations of 'foobar' interface */

}

#endif /*  H_INCLUDED_foobar_HH */
```

Abbildung 2: Struktur des gemeinsamen Skeletts aller Headerdateien

5 Modulschnittstellen

5.1 Bemerkungen zum Vorgehen

Alle Module wurden als C-Headerdateien entworfen. Zu jeder Headerdatei wurde eine Dummy-Implementationsdatei erstellt, welche die Headerdatei einschließt, aber keine Prozedurimplementationen enthält (Das ist in C möglich).

Zum Abschluss wurden alle Dummyimplementationen mit dem C++-Compiler in Objektmodule übersetzt. Diese enthalten zwar keinen sinnvollen Objektcode (da die Implementationen fehlen), jedoch wird derart während der Übersetzung eine Syntax- und Typprüfung durchgeführt. Headerdateien, Makefiles und Dummy-Implementationen sind als disas65-2002-10-22-a von der Website zur Vorlesung erhältlich.

5.2 Gemeinsamkeiten

Alle Headerdateien sind durch ein *#ifdef* -Konstrukt vor doppelter Inklusion geschützt. Dies gilt als gute C/C++-Praxis. Die Deklarationen sind in einen Namensraum mit dem Namen des Moduls eingekapselt. Beide Konstrukte (s. Abb. 2) werden im Folgenden nicht mitabgedruckt, um die Übersichtlichkeit zu bewahren und müssen jeweils hinzugedacht werden. Im Anhang sind nochmal die vollständigen Headerdateien zur Referenz abgedruckt.

```
24    typedef ABSTRACT_HEAP_( program );
25    typedef ABSTRACT_HEAP_( segment );

27    program*
28    NEW_from_file(FILE* f);

30    const segment*
31    codeseg_of(program* p);

33    const segment*
34    dataseg_of(program* p);

36    void
37    FREE(program* p);

41    cpu::byte_count
42    length_of(const segment* s);

44    cpu::address
45    address_of(const segment* s);

47    const cpu::byte*
48    image_of(const segment* s);
```

Abbildung 3: Schnittstelle des Moduls *programs*

5.3 Modul *programs*

Das Modul *programs* abstrahiert über die Eingabe, indem es einen abstrakten Datentyp – *program* – anbietet, welcher die *logische* Struktur der Eingabedaten (statt ihrer physikalischen Darstellung) wiedergibt. Der eigentliche Vorgang des Einlesens (oder Parsens) wird in der Instantiierung eines Elements des Datentyps aus einem Dateideskriptor (welcher implizit ein Array von Bytes „enthält“) versteckt.

Datentyp *program*

Abstraktes Heap-Objekt

Dieser Datentyp abstrahiert die Struktur eines Programms des Mikrocontrollers. Ein Programm besteht aus 2 Segmenten: Code- und Datensegment. Datenobjekte dieses Typs sind nach ihrer Erzeugung konstant.

program

Prozedur *codeseg***()**

`const segment* codeseg_of(program* p);`

Gibt das Codesegment eines Programms zurück.

Ret.: *sp*.

Pre: *p* muss auf ein korrekt initialisiertes Speicherobjekt vom Typ *program* verweisen.

Acc.: **p* wird nur gelesen.

Post: *sp* zeigt auf ein initialisiertes Speicherobjekt vom Typ *segment*. Dies ist das Codesegment des Programms.

Ress.: Die Verantwortung für die Freigabe von **sp* bleibt beim Modul *programs*. Der Zugriff auf **sp* ist solange gültig, solange **p* existiert, also solange der Zugriff auf **p* gültig ist.

Bemerkung: Die Funktion, welche das Datensegment aus einem Programm holt, besitzt sinngemäß die gleichen Eigenschaften.

Datentyp *segment*

Abstraktes Heap-Objekt

Dieser Datentyp abstrahiert die Struktur eines Programmsegments des Mikrocontrollers. Ein Segment besitzt die Attribute Länge, Ladeadresse und Speicherabbild (Image, das ist die Folge von Bytes, aus denen das Segment besteht).

Prozedur *dataseg***()**

```
const segment* dataseg_of(program* p);
```

Gibt das Datensegment eines Programms zurück.

Ret.: *sp*.

Pre: *p* muss auf ein korrekt initialisiertes Speicherobjekt vom Typ *program* verweisen.

Acc.: **p* wird nur gelesen.

Post: *sp* zeigt auf ein initialisiertes Speicherobjekt vom Typ *segment*. Dies ist das Datensegment des Programms.

Ress.: Die Verantwortung für die Freigabe von **sp* bleibt beim Modul *programs*. Der Zugriff auf **sp* ist solange gültig, solange **p* existiert, also solange der Zugriff auf **p* gültig ist.

Prozedur *length_of***()**

```
cpu::byte_count length_of(const segment* s);
```

Gibt die Länge *l* von **s* zurück.

Ret.: *l*.

Pre: *s* muss auf ein korrekt initialisiertes Speicherobjekt vom Typ Segment verweisen.

Acc.: **s* wird nur gelesen.

Post: Der Rückgabewert *l* enthält die Länge des Segments **s*.

Ress.: Verantwortung für die Freigabe von **s* ändert sich nicht (bleibt also beim Modul *programs*).

Prozedur *address_of* **()**

```
cpu::address address_of(const segment* s);
```

Gibt die Ladeadresse des Segments **s* zurück.

Ret.: *ad*.

Pre: *s* muss auf ein korrekt initialisiertes Speicherobjekt vom Typ Segment verweisen.

Acc.: **s* wird nur gelesen.

Post: Der Rückgabewert *ad* enthält die Ladeadresse des Segments **s*.

Ress.: Verantwortung für die Freigabe von **s* ändert sich nicht (bleibt also beim Modul *programs*).

Prozedur *image_of* **()**

```
const cpu::byte* image_of(const segment* s);
```

Gibt einen Verweis auf das Speicherabbild des Segments **s* zurück.

Ret.: *ip*.

Pre: *s* muss auf ein korrekt initialisiertes Speicherobjekt vom Typ Segment verweisen.

Acc.: **s* wird nur gelesen.

Post: Der Rückgabewert *ip* zeigt auf ein Array mit *length_of*$(*s)$ Elementen des Typs *cpu*::*byte*, welche das Speicherabbild des Segments repräsentieren.

Ress.: Verantwortung für die Freigabe von **s* ändert sich nicht (bleibt also beim Modul *programs*).

Der Zugriff auf **ip* ist solange gültig, solange **sp* existiert, also solange der Zugriff auf **sp* gültig ist.

Bem.: Der Rückgabewert enthält selbst nur einen Zeiger, keine Information darüber, wieviele Elemente des durch *ip* referenzierten Speicherbereichs gültig sind. Diese Information muss mit *length_of*() ermittelt werden.

Prozedur *NEW_from_file***()**

```
program* NEW_from_file(FILE* f);
```

Liest ein Programm aus der Datei *f* ein.

Ret.: *p*.

Pre: Hinter dem augenblicklichen Dateizeiger von f muss eine Bytefolge in der Datei stehen, die sich in dem in der Anleitung erwähnten Sinn als Programm interpretieren lässt (d. h. vor allem, dass nach dem Header genügend Daten enthalten sein müssen, die sich als Code- und Datensegment lesen lassen).

Acc.: Die Datei *f** wird nur gelesen, aber der Inhalt der Datei nicht verändert.

Post: Der Zeiger *p* verweist auf ein initialisiertes Speicherobjekt vom Typ *program*. Sein Code- und Datensegment sind identisch mit den in der Datei gespeicherten, wenn man die Interpretation aus der Aufgabenstellung voraussetzt.

Ress.: Verantwortung für das Schließen der Datei geht an das Modul über. Die aufrufende Prozedur darf die Datei über den Dateideskriptor *f* nicht mehr lesen oder schreiben.

Excn.: Ist der Dateiinhalt inkonsistent, also die im Header angegebenen Segmentlängen größer als die Länge des Restes der Datei, wird ein 0-Zeiger zurückgegeben. Dasselbe geschieht bei Lesefehlern, dabei wird zusätzlich noch *errno* (siehe ANSI C Standard Bibliothek) gesetzt. Der Dateideskriptor wird dann sofort geschlossen.

Bem.: Die Vereinbarung, dass die Verantwortung für das Schließen der Datei an das Modul übergeht, erlaubt einen gewissen Spielraum der Implementation im Bezug darauf, wann Segmentinhalte tatsächlich eingelesen werden.

```
24    typedef ABSTRACT_VAL_(int)    opcode;
25    typedef struct{ char t[4] ;} mnemonic;

27    opcode          from_byte(cpu::byte b);

29    mnemonic
30    mnemonic_of(opcode oc);

32    oprdfmts::oprdfmt
33    oprd_fmt_of(opcode oc);

35    int
36    instr_len(opcode oc);

38    int
39    is_legal(cpu::byte b);

41    void
42    define_opcode(cpu::byte b,
43                  mnemonic m,
44                  oprdfmts::oprdfmt of,
45                  int instr_len);

47    void
48    init_module();

50    void
51    deinit_module();
```

Abbildung 4: Schnittstelle des Moduls *opcodes*

5.4 Modul *opcodes*

Das Modul *opcodes* implementiert *Opcode[s]* als abstrakten Datentyp. Datenverarbeitende Leistungen mit *Opcode[s]* – wie das Ermitteln des zugehörigen Mnemonik, der Länge von Instruktionen oder des zugehörigen Operandenformats, die mit dem betreffenden *Opcode* gebildet werden – werden als Attribute des Datentyps implementiert (*instr_len*(), *mnemonic_of*(), *oprd_fmt_of*()).

Da die Wertemenge des Datentyps und die zu den Werten gehörigen Attribute zur Übersetzungszeit des Moduls noch nicht feststehen, sondern erst beim Start des Programms von einer Beschreibungsdatei gelesen werden müssen, wird eine Prozedur zur Verfügung gestellt (*define_opcode*()), mit der sich die Attribute bei der Initialisierung des Moduls definieren lassen. Diese Prozedur sollte nur vor den Operationen auf dem ADT *opcode* verwendet werden, d. h. zur Initialisierung des Moduls.

Zustand: Dieses Modul besitzt einen inneren Zustand, der während der Initialisierungsphase mit der Funktion *define_opcode*() aufgebaut wird.

Ein geeignetes Modell des inneren Zustands des Moduls ist eine Tabelle, welche Werten des Typs *opcode* jeweils ein Operandenformat *operdfmts*::*oprdfmt*, einen Mnemonik *mnemonic*, eine Byte-Darstellung *cpu*::*byte* und einen Instruktionenlänge der mit diesem Opcode gebildeten Instruktionen zuordnet.

Datentyp *mnemonic*

`struct{ char t[4];}`

Textnotation einer Operation des Mikroprozessors.

Darstellungsinvarianten: Für *m* :*mnemonic* ist immer $m[3] == 0$.

Datentyp *opcode*

Abstrakter Wert

Die abstrakte Darstellung eines *Opcode* des Mikrocontrollers.

Invarianten: Die mit *instr_len*() ermittelte Instruktionslänge ist immer 0,1 oder 2.

Die folgenden Prozeduren erlauben den Zugriff auf die Attribute eines *opcode*.

Prozedur *mnemonic_of* **()**

`mnemonic mnemonic_of(opcode oc);`

Gibt den Mnemonik von *oc* zurück.

Post: s. Beschreibung.

Prozedur *oprd_fmt_of* **()**

`oprdfmts::oprdfmt oprd_fmt_of(opcode oc);`

Gibt das zu *oc* gehörende Operandenformat zurück.

Post: s. Beschreibung.

Rationale: Das (für den Ausdruck zu wählende) Operandenformat ist eigentlich durch die Adressart der Operation bestimmt. Andererseits ist die Adressart im Opcode kodiert, und in der Anwendung wird nur das Operandenformat wirklich benötigt. Ich habe mich deshalb entschlossen diesen zweistufigen Lookup zu verbergen und das Operandenformat direkt als Attribut des Opcode darzustellen. Es liegt in der Verantwortung des Prozesses, der mittels *define_opcode*() die Opcodetafel aufbaut, die Zuordnung zwischen Opcodes und Operandenformaten zu ermitteln. Siehe auch Modul *addrmodes*.

Prozedur *instr_len***()**

`int instr_len(opcode oc);`

Gibt die Länge zurück, welche Instruktionen haben, die mit *oc* gebildet werden.

Post: s. Beschreibung. Das Resultat ist > 1.

Die folgenden Prozeduren erlauben es, ein *cpu::byte* in einen Opcode umzuwandeln.

Prozedur *is_legal***()**

```
int is_legal(cpu::byte b);
```

Überprüft, ob das übergebene Byte *b* einen *opcode* kodiert. Ein *opcode* gilt als legal, wenn er zuvor mit *define_opcode*() definiert wurde.

Pre: Keine. Alle Werte aus *cpu::byte* sind erlaubt.

Post: Resultat ist $\neq 0$ wenn der Opcode sich in der Opcode-Tabelle des Moduls befindet, 0 sonst.

Rationale: Eine solche Funktion ist nötig, da der Disassembler beim Parsen des Codesegements das Byte am Anfang einer Instruktion dazu verwenden muss, um die Länge der Instruktion zu bestimmen (letzten Endes: Zu bestimmen, wo die nächste Instruktion beginnt). Dazu muss dieses Byte *b* erst mit *from_byte*() als *opcode* interpretiert werden. Diese Interpretation ist aber nur zulässig, wenn ein solches *b* zuvor als Opcode definiert wurde (und zwar in der Initialisierungsphase des Programms, wenn die Definitionstabellen geladen werden). Ob dies der Fall war, kann mit *is_legal*() festgestellt werden.

Prozedur *from_byte***()**

```
opcode from_byte(cpu::byte b);
```

Liefert den *opcode* mit der Byte-Darstellung *b* zurück, d. h. *b* wird in einen *opcode* umgewandelt

Pre: Es muss ein Opcode mit der Byte-Darstellung *b* definiert worden sein (sich in der Opcode-Tabelle des Moduls befinden).

Post: Resultat ist der *opcode*, der *b* als Byte-Darstellung besitzt.

Die folgenden Prozeduren dienen der Initialisierung des Moduls und der Definition von Attributen der Opcodes.

Prozedur *define_opcode***()**

```
void define_opcode(cpu::byte b, mnemonic m,
                                oprdfmts::oprdfmt of, int instr_len);
```

Definiert einen neuen Opcode mit den Attributen *b*, *m*, *of* und *instr_len*.

Mod.: Inneren Zustand des Moduls: Opcode-Tabelle.

Pre: Ein Opcode mit dem Byte *b* darf noch nicht definiert sein (sich in der Opcode-Tabelle des Moduls befinden), $instr\_len > 1$.

Post: Das Tupel *b*, *m*, *of* und *instr_len* befindet sich in der Opcode-Tabelle des Moduls. Das bedeutet, dass $is\_legal(b)$ gilt, und der Opcode $oc = from\_byte(b)$ gerade die hier genannten Attribute hat.

Prozedur *init_module***()**

```
void init_module();
```

Initialisiert das Modul. Diese Funktion muss vor dem ersten Aufruf irgendeiner Funktion des Moduls aufgerufen werden.

Mod.: Inneren Zustand des Moduls.

Pre: Modul muss noch uninitialisiert sein.

Post: Modul ist initialisiert, es sind keine Opcodes definiert.

Es ist möglich (im vorliegenden Fall sogar wahrscheinlich), dass *init_module*() eine Nulloperation ist – d. h. nichts tut. Die eigentliche Aufgabe solcher Initialisierungsfunktionen ist es, dem Modul die Gelegenheit zu geben, notwendigen Speicher für den inneren Zustand des Moduls zu allozieren, statisch Lookup-Tabellen zu berechnen, die nötigen Log- oder sonstige Dateien zu öffnen und so weiter. Da jedoch die Frage, ob ein konkretes Modul solche Initialisierungsaufgabe wahrnehmen muss, nur mit der konkreten Implementation und eher nicht bei der Definition des Interface entschieden werden kann, halte ich an dem Grundsatz fest, dass jedes Modul eine solche Funktion benötigt.

Prozedur *deinit_module***()**

```
void deinit_module();
```

Deinitialisiert das Modul. Diese Funktion muss nach dem letzten Gebrauch des Moduls aufgerufen werden.

Mod.: Inneren Zustand des Moduls (Opcode-Tabelle), Heap.

Pre: Modul muss initialisiert sein.

Post: Modul gilt als uninitialisiert.

Auch hier gilt sinngemäß derselbe Kommentar, wie zu *init_module*(). Die Aufgabe einer solchen Prozedur ist es, vor Programmende vom Modul verwaltete Ressourcen (Speicher, Dateideskriptoren usw.) wieder freizugeben. Da üblicherweise auf den meisten Betriebssystemen die Ressourcen mit der Beendigung des Programms automatisch wieder freigegeben werden, sind die meisten Deinitialisierungsprozeduren Nulloperationen. Allerdings gibt es auch Ressourcen, die vom Betriebsystem nicht wieder freigegeben werden: Dot-Locks und temporäre Dateien beispielsweise.

5.5 Bemerkung zur Initialisierung und Deinitialisierung von Modulen

In den folgenden Abschnitten werden Initialisierungs- und Deinitialisierungsprozeduren von Modulen nicht mehr explizit gemacht. Es empfiehlt sich jedoch, in der Implementation für jedes Modul solche Prozeduren vorzusehen, auch wenn es Nulloperationen sind.

Bei anderen Sprachen, wie etwa Modula oder Ada, kann jedes Modul einen Modulrumpf enthalten, der Initialisierungsanweisungen enthält. Trotzdem muss Deinitialisierung meist explizit geschehen, und zudem gibt es in einigen Sprachen kein Sprachmittel die Initialisierungsreihenfolge von Modulen zu regeln, obwohl das häufig nötig ist.

```
25    typedef ABSTRACT_VAL_(int)    addrmode;

28    addrmode from_string(const char* txtp);

30    oprdfmts::oprdfmt
31    oprd_fmt_of(addrmode am);

33    int
34    oprd_len(addrmode am);

39    addrmode
40    add_addrmode(oprdfmts::oprdfmt* ofp, int oprd_len, \
                                           const char* textrepr);

42    void
43    init_module();

45    void
46    deinit_module();
```

Abbildung 5: Schnittstelle des Moduls *addrmodes*

5.6 Modul *addrmodes*

Importiert öffentlich *oprdfmts*.

Das Modul *addrmodes* implementiert Adressarten als abstrakten Datentyp. Datenverarbeitende Leistungen mit Adressarten werden, wie schon im Modul *opcodes* als Attribute des abstrakten Datentyps implementiert.

Da die Wertemenge des Datentyps und die zu den Werten gehörigen Attribute zur Übersetzungszeit des Moduls noch nicht feststehen, sondern von einer Definitionsdatei gelesen werden sollen, wird eine Prozedur zur Verfügung gestellt (*define_addrmode*()), mit der sich diese bei der Initialisierung des Moduls definieren lässt. Diese Prozedur sollte nur vor den Operationen auf dem ADT *addrmode* verwendet werden, d. h. zur Initialisierung des Moduls.

Ein geeignetes Modell des inneren Zustands des Moduls ist eine Tabelle, welche Werten des Typs *addrmode* jeweils die nötigen Attribute zuordnet.

Datentyp *addrmode*

Abstrakter Wert

Die abstrakte Darstellung einer Adressart des Mikrocontrollers.

Inv.: Die Operandenlänge ist immer aus 0, 1, 2.

Prozedur *oprd_fmt_of* **()**

`oprdfmts::oprdfmt oprd_fmt_of(addrmode am);`

Ermittelt das zur Adressart gehörige Operandenformat.

Post: s. Beschreibung.

Prozedur *oprd_len***()**

```
int oprd_len(addrmode am);
```

Ermittelt die zur Adressart *am* gehörende Operandenlänge.

Post: s. Beschreibung.

Prozedur *from_string***()**

```
addrmode from_string(const char* txtp);
```

Ermittelt die Adressart mit der Textdarstellung **txtp*.

Ret.: *txt*.

Acc.: Der Zeiger *txtp* soll auf einen Speicherplatz zeigen, in dem ein textlicher Name für den Adressmode (d. h. „IM“ oder „AB“) als 0-terminierte Zeichenkette abgelegt ist.

Ress.: Verantwortlichkeit für (Eigentum am) den Speicher **txtp* verbleibt bei der aufrufenden Prozedur.

Pre: Eine Adressart mit der Textdarstellung *txtp* muss definiert sein.

Post: Resultat s. Beschreibung.

Prozedur *add_addrmode***()**

```
addrmode add_addrmode (oprdfmts::oprdfmt* ofp,
                                  int oprd_len, const char* textrepr);
```

Definiert eine neue Adressart mit den Attributen **ofp*, *oprd_len* und **textrepr* und liefert deren Darstellung zurück.

Ret.: *am*.

Mod.: Inneren Zustand des Moduls: Tabelle der Adressarten.

Acc.: Der Zeiger *textrepr* soll auf einen Speicherplatz zeigen, in dem ein textlicher Name für den Adressmode (d. h. „IM“ oder „AB“) als 0-terminierte Zeichenkette abgelegt ist.

Pre: *textrepr* muss ein 0-terminierte Zeichenkette sein, *oprd_len* muss aus 0,1,2 sein. Es darf noch keine Adressart mit der Textdarstellung *textrepr* definiert sein.

Post: Die Tabelle der Adressarten wird ist um eine neue Adressart ergänzt. Die Adressart *am* ist nach der Operation definiert, war vor der Operation nicht definiert und besitzt die hier als Parameter übergebenen Attribute.

Ress.: Die Verantwortlichkeit für die eventuelle Speicherfreigabe des Zeiger *textrepr* bleibt bei der aufrufenden Prozedur, *add_addrmode*() kopiert den Inhalt des Speichers auf den der Zeiger verweist. Dagegen geht das durch *ofp* referenzierte Speicherobjekt in die Verantwortlichkeit des Modul *addrmodes* über und soll danach lesend und schreibend ausschließlich von diesem verwendet werden: Der Speicher zu *ofp* muss vom Heap alloziert sein, und wird von *addrmodes* zur gegebenen Zeit wieder freigegeben.

Bemerkung zur Implementation: Werden zwei Adressarten mit derselben Textdarstellung definiert (d. h. gegen diesen Teil der Vorbedingung verstossen), wird das Resultat implementationsabhängig sein: Entweder bricht die Implementation mit einem Fehler ab, geht in einen undefinierten Zustand, ersetzt stillschweigend die vorangegangene Definition oder ignoriert (effektiv) die neue Definition.

```
22    typedef ABSTRACT_HEAP_( oprdfmt );

24    int
25    max_digits(oprdfmt* ofp);

27    int
28    max_len(oprdfmt* ofp);

30    void
31    format(oprdfmt* ofp, int, char* buf);

36    oprdfmt*
37    new_oprdfmt(const char* fmtstr);

45    void
46    free_oprdfmt(oprdfmt* ofp);
```

Abbildung 6: Schnittstelle des Moduls *oprdfmts*

5.7 Modul *oprdfmts*

Dieses Modul stellt Mechanismen zur Übersetzung von Operanden in eine Textdarstellung zur Verfügung. Letzten Endes charakterisiert ein *oprdfmt* eine Abbildung einer ganzen Zahl (hier vom *int*) in einen Text. Da die Definition, welche Operanden wie übersetzt werden müssen, letzten Endes in einer Konfigurationdatei vorgenommen wird, wird hier auch eine Funktion zur Verfügung gestellt, um Operandenformate (*oprdfmt*) aus einer speziellen Untermenge von Printf-Format-Strings zu erzeugen.

Datentyp *oprdfmt*

Abstraktes Heap-Objekt

Ein Operandenformat *oprdfmt* ist die opake Beschreibung einer Abbildung von ganzen Zahlen (*int*) in Text. Es besitzt die Attribute *max_digits* und *max_len*.

Dabei gibt *max_digits* an, wieviele Stellen der Zahl maximal dargestellt werden können und *max_len*, wie lang die resultierende Zeichenkette maximal werden kann.

Invarianten: $max\_digits \leq max\_len$

Prozedur *max_digits***()**

```
int max_digits(oprdfmt* ofp);
```

Ermittelt das Attribut *max_digits*, welches angibt, wieviele Stellen einer Zahl mit dem Operandenformat **ofp* maximal dargestellt werden können.

Prozedur *max_len***()**

```
int max_len(oprdfmt* ofp);
```

Ermittelt das Attribut *max_len*, welches angibt, wie lang eine mit dem Format **ofp* formatierte Zeichenkette maximal werden könnte.

Prozedur *format***()**

```
void format(oprdfmt* ofp, int, char* buf);
```

Eine Zahl *n* wird eine Zeichenkette übersetzt (formatiert) und in dem Puffer abgelegt, auf den *buf* zeigt.

Acc.: **ofp* wird nur gelesen, der Speicherplatz, auf den *buf* zeigt, wird mit dem Ergebnis beschrieben (aber nicht gelesen).

Pre: *buf* muss ein Verweis auf einen Speicherblock von mindestens $1 + max\_len(*ofp)$ Zeichen sein.

Post: Im Puffer *buf* befindet sich eine Textdarstellung von *n* gemäß dem Format **ofp*; für weitere Erläuterungen siehe auch *new_oprdfmt*(). Dabei gilt $strlen \leq max\_len(*ofp)$.

Ress.: Verantwortlichkeit für die Zeiger *buf* und *ofp* bleibt bei der aufrufenden Prozedur. Beide Zeiger werden nur während der Ausführung von *format*() verwendet.

Prozedur *new_oprdfmt***()**

```
oprdfmt* new_oprdfmt(const char* fmtstr);
```

Erzeugt auf dem Heap ein Speicherobjekt, dass ein Operandformat enthält, welches durch den *printf*-String *fmtstr* beschrieben wird.

Ret.: *ofp*.

Acc.: Der Speicher **fmtstr* wird nur gelesen.

Mod.: Den Heap.

Pre: *fmtstr* muss auf einen durch 0 terminierten Zeichenkette zeigen. **fmtstr* muss durch den regulären Ausdruck ^[^%\\]*%0[1-9]x[^%\\]*$ beschrieben werden (z. B. „$(%04x),Y“).

Post: Der Formatstring beschreibt im Sinne der in der C-Standard-Bibliothek implementierten Prozedur *printf*() ein Abbildung von Werten des Typs *int* auf Zeichenketten. (*ofp) beschreibt (so wie durch *format*() implementiert) genau dieselbe Abbildung.

Der Speicher **ofp* ist vom Heap alloziert.

Ress.: Die Verantwortlichkeit für den Speicher **fmtstr* bleibt bei der aufrufenden Prozedur. *fmtstr* wird nicht gespeichert.

Rationale: Das Modul *oprdfmts* bietet lediglich einen geschickten Wrapper um die Funktion *sprintf*() aus der C-Standardbibliothek, um genaue Information und Kontrolle darüber zu haben, welche Länge beim formatieren resultierende Zeichenkette hat, was bei der Formatierung mit *sprintf*() im allgemeinen Fall nicht nötig ist.

Die Einschränkungen für *fmtstr* wurden so gewählt, dass es vor allem ohne großen Aufwand möglich sollte, *max_digits* und *max_len* aus *fmtstr* zu ermitteln.

Prozedur *free_oprdfmt***()**

```
void free_oprdfmt(oprdfmt* ofp);
```

Gibt den Speicher für das Heapobjekt **ofp* wieder frei.

Mod.: Den Heap.

Pre: *ofp* muss auf einen heap-allozierten Speicherblock zeigen, der einen Wert vom Typ *oprdfmt* enthält (d. h. *ofp* muss ein Zeiger sein, der von *new_oprdfmt*() erhalten wurde).

Post: Die Referenz *ofp* ist nicht mehr gültig, der Speicher wurde an den Heap zurückgegeben.

Ress.: In einem gewissen Sinn übernimmt die Prozedur die „Verantwortung“ (und reserviert die exklusive Verwendung) für den Speicher *(*ofp). Die aufrufende Prozedur darf den Speicher nicht mehr beschreiben oder den Zeiger deferenzieren.

Rationale: Es mag auf den ersten Blick unnötig erscheinen, eine besondere Prozedur zur Freigabe eines Speicherobjektes vom Typ *oprdfmt* zur Vefügung zu stellen. Bei genauerem Hinsehen, stellt sich jedoch Folgendes heraus: *new_oprdfmt*() belässt die Verantwortlichkeit und Berechtigung für die Deallokation des Speichers **fmtstr* bei der aufrufenden Prozedur. Da die Zeichenkette **fmtstr* jedoch in der Prozedur *format*() später noch benötigt werden wird, wird sie wohl intern kopiert werden müssen. Hierfür muss durch den Heap oder anderweitig Speicher zur Verfügung gestellt werden, der beim Auflösen der Datenstruktur **ofp* wieder freigegeben werden muss. Dafür ist *free_oprdfmt*() verantwortlich – die einfache Verwendung von *free*() aus der Standard-Bibliothek würde entweder zu einem Speicherleck führen, oder uns zwingen, die innere Struktur von *oprdfmt* offenzulegen, um die dort gespeicherten Zeiger zum Zweck der Ausgabe auslesen zu können. Abgesehen einmal davon, dass eine solche Sichtbarkeit gegen das Geheimnisprinzip verstossen würde, würde sie sich auch zu sehr darauf verlassen, dass die Klientenmodule hier alles richtig machen, nämlich den richtigen Speicher zur richtigen Zeit freigeben.

```
24    typedef ABSTRACT_VAL_( struct{

26      int size;
27      int val;                  }) operand;

29    operand
30    from_bytes(int size, cpu::byte* b);

32    char *
33    format(oprdfmts::oprdfmt* ofp, operand od);
```

Abbildung 7: Schnittstelle des Moduls *operands*

5.8 Modul *operands*

Importiert öffentlich *cpu*, *oprdfmts*.

Dieses Modul stellt einen abstrakten Datentyp zur Verfügung, der die Operanden der Instruktionen des Mikroprozessor charakterisiert, sowie Prozeduren um Bytes in Operanden zu konvertieren und Textdarstellungen für Operanden zu erzeugen.

Datentyp *operand*

Abstrakter Wert

Werte des Typs *operand* beschreiben die abstrakte Sicht eines Operanden einer Instruktion des Mikrokontrollers. Ein Operand hat ein Länge (Größe), diese kann 1 oder 2 Byte betragen, und einen Wert, dieser ist eine ganze Zahl aus $\{0 \ldots 65535\}$

Prozedur *from_byte***()**

`operand from_bytes(int size, cpu::byte* b);`

Erzeugt einen Operanden aus *size* Elementen des Typs *cpu*::*byte*.

Acc.: Aus dem Speicher **b* werden die ersten *size* Elemente des Typs *cpu*::*byte* gelesen.

Pre: *size* muss 1 oder 2 sein. *b* muss auf ein Array von mindestens *size* korrekt initialisierten Elementen des Typs *cpu*::*byte* zeigen.

Post: Ein *cpu*::*byte* ist eine Zahl aus $\{0 \ldots 255\}$. Die übergebenen Parameter werden in einen Operanden der Größe *size* umgewandelt, dessen Wert, wenn $size = 1$ ist, $b[0]$ ist und wenn $size = 2$, $b[0] + 256 * b[1]$ ist (LSB first byte order).

Ress.: Die Verantwortung für die Freigabe des Speichers **b* bleibt bei der aufrufenden Prozedur. Der Zeiger *b* wird nicht gespeichert.

Prozedur *format***()**

`char* format(oprdfmts::oprdfmt* ofp, operand od);`

Wandelt einen Operanden gemäß dem Operandformat in eine Textdarstellung um.

Ret.: *cp*.

Acc.: **ofp* wird nur gelesen, der Zeiger nicht gespeichert.

Mod.: Heap.

Pre: Der Wert Operanden muss klein genug sein, um in $max\_digits(ofp)$ Stellen dargestellt werden zu können. Da die Zahlenbasis, zu der dargestellt wird, z. Z. nur 16 sein kann (s. *oprdfmts*), ist eine hinreichende Bedingung dafür, dass die Länge des Operanden maximal $max\_digits(ofp)/2$ ist.

Post: In **cp* befindet sich eine 0-terminierte Zeichenkette, welche im Sinne des Formats **ofp* eine Textdarstellung des Operanden ist (s. *oprdfmts*).

Ress.: Sei *cp* der Rückgabewert. Die Verantwortung für die Wiederfreigabe des Speichers **cp* liegt bei der aufrufenden Prozedur.

Excn.:

Rationale: Es werden kein Prozeduren zur Verfügung gestellt, mit denen aus einem Operanden dessen Attribute (Wert und Länge) ermittelt werden können. Der Disassembler benötigt diese Operationen nicht.

```
23    typedef ABSTRACT_VAL_( struct{

25      cpu::address      ad;
26      opcodes::opcode   oc;
27      bool              has_oprd;
28      operands::operand op;        })    instr;

31    instr
32    from_opcode(cpu::address ad, opcodes::opcode oc);

34    instr
35    with_operand(cpu::address ad, opcodes::opcode oc,     \
                                    operands::operand op);

37    opcodes::opcode
38    opcode_of(instr in);

40    operands::operand
41    operand_of(instr in);

43    cpu::address
44    address_of(instr in);

46    bool
47    has_oprd(instr in);
```

Abbildung 8: Schnittstelle des Moduls *instrs*

5.9 Modul *instrs*

Importiert öffentlich *cpu*, *opcodes*, *operands*.

Dieses Modul stellt einen Datentyp zur Verfügung, der die Instruktionen des Mikroprozessors beschreibt.

Datentyp *instr*

Art: Abstrakter Wert

Der Typ *instr* abstrahiert über die Instruktionen des Mikrocontrollers. Eine Instruktion besitzt eine Adresse, einen Opcode und einen optionalen Operanden.

Rationale: Dieser Typ ist wenig mehr als ein Container für einen Opcode und einen Operanden. Dies würde im Grunde dafür sprechen, den Typ nicht-opak anzulegen. Dadurch jedoch, dass der Operand optional ist, ist die Interpretation der Darstellung gewissen Regeln unterworfen, deren Einhaltung bei Verbergen der Darstellung leichter zu erzwingen und besser zu dokumentieren ist.

Die folgenden Prozeduren erlauben Zugriff auf die Attribute einer *instr*.

Prozedur *opcode_of* **()**

```
opcodes::opcode opcode_of(instr in);
```

Extrahiert den Opcode aus einer Instruktion.

Post: s. Beschreibung.

Prozedur *address_of* **()**

```
cpu::address address\_of(instr in);
```

Extrahiert die Adresse aus einer Instruktion.

Post: s. Beschreibung.

Prozedur *has_oprd* **()**

```
int has_oprd(instr in)
```

Ermittelt, ob eine Instruktion einen Operanden enthält.

Post: Rückgabewert ist 0, wenn die Instruktion keinen Operanden enthält, $\neq 0$ sonst.

Prozedur *operand_of* **()**

```
operands::operand operand_of(instr in);
```

Extrahiert den Operanden aus einer Instruktion.

Pre: Es muss $has\_oprd(in)$ gelten.

Post: s. Beschreibung.

Die folgenden Prozeduren können dazu verwendet werden um Werte des Typs *instr* zu konstruieren.

Prozedur *from_opcode* **()**

```
instr from_opcode(cpu::address ad, opcodes::opcode oc)
```

Erzeugt eine neue Instruktion, welche aus einem Opcode (ohne Operand) besteht.

Ret.: *in*.

Post: Es gilt $\neg has\_oprd(in)$.

Prozedur *with_operand* **()**

```
instr with_operand(cpu::address ad, opcodes::opcode oc,
                                                operands:operand op);
```

Erzeugt eine neue Instruktion, welche nur aus einem Opcode und einem Operanden besteht.

Ret.: *in*.

Post: Es gilt $has\_oprd(in)$.

```
21    typedef struct{
22      const char*         addr_text;
23      opcodes::mnemonic   mn;
24      const char*         oprd_text; }    instr_text;

26    void
27    INIT_instr_text(instr_text* it,
28                    const char* addr_text,
29                    opcodes::mnemonic mn,
30                    const char* oprd_text);

32    void
33    DEINIT_instr_text(instr_text* it);
```

Abbildung 9: Schnittstelle des Moduls *instr-texts*

5.10 Modul *instr_text*

Importiert öffentlich *opcodes*.

Dieses Modul definiert den Datentyp *instr_text*. Es handelt sich dabei um einen Container für die Ergebnisse der Disassemblierung einer Instruktion: Den Mnemonik, der die Operation, und den Text, der den Operanden beschreibt.

Rationale: Die Ergebnisse der Disassemblierung (siehe Modul *disassembly*) müssen letzten Endes zeilenweise dem Modul *layouter* übergeben werden, welches diese in Zeilen und Spalten auf einer Seite anordnet. Das Ergebnis der Disassemblierung sind mehrere (hier: zwei) Zeichenketten, welche eine symbolischen Text für die Operation und eine Notation für Adressart und numerischen Wert des Operanden enthalten. Hier haben sich zwei Lösungen für die Darstellung der Ergebnisse der Disassemblierung angeboten: Zum einen hätte man das Ergebnis der Disassemblierung in *zwei* Zeichenketten (in zwei getrennten Variablen) darstellen können und beide gemeinsam, aber in getrennten Parametern an die betreffende Prozedur des Moduls *layouter* übergeben können.

Die andere Alternative bestand darin, zumindest ansatzweise über das *Ergebnis der Disassemblierung einer Instruktion* zu abstrahieren, und einen Datentyp zu definieren, der dieses Ergebnis enthält. Genau dies habe ich im vorliegenden Modul getan: Zwar entsteht dadurch ein höherer Aufwand, aber das Konzept wird klarer sichtbar.

Sowohl das Modul *disassembly* als auch das Modul *layouter* müssen Kenntnis vom Datentyp *instr_text* besitzen, der ja zum Austausch von Daten zwischen beiden Modulen dient. Andererseits ruft keines der beiden Module das andere direkt (es wird, wie man sehen wird Aufgabe der Prozeduren im Main-Modul sein, Daten aus *disassembly* zu ziehen und an *layouter* weiterzugeben) und es erschiene unangemessen, dass eines der beiden Module das jeweils andere importiert. Aus diesem Grund, muss die Struktur *instr_text* in einem eigenen Modul angesiedelt werden, das sowohl von *disassembly* als auch von *layouter* importiert wird.

Bemerkung: Im vorliegenden Beispiel erscheint es im Nachhinein fast einfacher, auf die Definition dieses speziellen Datentyps zu verzichten und stattdessen mit einzelnen Zeichenketten zu hantieren. Die Speicherverwaltung für das Clientmodul (die jetzt in *INIT_instr_text*() und *DEINIT_instr_text*() ausgelagert ist, würde dadurch allerdings etwas

komplizierter und damit auch fehlerträchtiger. Das Vorgehen im vorliegenden Fall demonstriert allerdings recht gut, wie zu diesem Zweck entworfene offene Datenstrukturen (also: nicht abstrakte) dem effizienten oder bequemen Datenaustausch zwischen zwei Modulen dienen können, und dass diese Strukturen in ein *drittes* Modul ausgelagert werden sollten, da sonst unsinnige und willkürliche Importbeziehungen zwischen den beiden in Frage stehenden Modulen resultieren.

Datentyp *instr_text*

```
struct {
       const char*       addr_text ;
       opcodes::mnemonic mn        ;
       const char*       oprd_text ; }
```

Dieser Datentyp dient als Container für die Ergebnisse der Disassemblierung einer Instruktion und für die Eingabedaten des Moduls *layouter*.

Prozedur *INIT_instr_text***()**

```
INIT_instr_text(instr_text* it, const char* addr_text,
                                opcodes::mnemonic mn, char* oprd_text);
```

Initialisiert eine Variable des Typs *instr_text* mit *addr_text*, *mn* und *oprd_text*.

Pre: *oprd_text* und *addr_text* müssen jeweils auf eine 0-terminierte Zeichenkette verweisen. **oprd_text* und **it* müssen vom Heap alloziert sein. Dabei muss **it* uninitialisiert sein.

Acc.: **it* wird beschrieben. *oprd_text* und *addr_text* sind *keine* Übergaben eines Parameters per Referenz, sondern die Übergabe einer Adresse (eines Pointers).

Post: **it* ist initialisiert, wobei

$$\begin{aligned} it{->}mn &= mn \\ it{->}oprd\_text &= oprd\_text \\ it{->}addr\_text &= addr\_text \end{aligned}$$

gelten wird.

Ress.: Die Zeiger *oprd_text* und *addr_text* werden in **it* gespeichert. Die Verantwortlichkeit für die Freigabe der Speicher **oprd_text* und **addr_text* geht an das Modul *instr_text* über. Die Verantwortung für die zeitige Deinitialisierung von **it* geht an die aufrufende Prozedur über.

Prozedur *DEINIT_instr_text***()**

```
INIT_instr_text(instr_text* it);
```

Deinitialisiert eine Variable des Typs *instr_text* und gibt assoziierten Speicher wieder frei.

Mod.: Heap.

Pre: **it* muss initialisiert sein.

Acc.: **it* wird gelesen und beschrieben.

Post: **it* gilt nun als uninitialisiert. Der Inhalt der Variablen darf nicht mehr interpretiert werden.

```
23    typedef ABSTRACT_HEAP_( job );

25    typedef struct{
26      const char* line_format;
27      int         pagelen;
28      int         topmargin;
29      int         headskip;
30      int         titleskip;
31      int         leftmargin; } layout_format;

33    job*
34    new_job(layout_format* lf, const char* headline);

36    void
37    print_title(job* jb,
38                const char* filename,
39                int         dsstart,
40                int         dslen,
41                int         csstart,
42                int         cslen      );

44    void
45    print_line(job* jb,
46               instr_texts::instr_text it);

48    int
49    end_job();
```

Abbildung 10: Schnittstelle des Moduls *layouter*

5.11 Modul *layouter*

Importiert öffentlich instr_text.

Verwendet intern line_printer.

Das Modul *layouter* stellt ein abstraktes Gerät zur Verfügung, mit dem die Ergebnisse der Disassemblierung formatiert ausgegeben werden können. Dieses Gerät ist intelligent genug, um den Seitenumbruch und das Zeilenlayout selbständig vornehmen zu können.

Bemerkung: Die ganze Struktur dieses Moduls ist – inklusive der *job*-Objekte welche den Status von Aufruf zu Aufruf mitführen – der des Moduls *lineprinter* sehr ähnlich.

Datentyp *layout_format*

```
struct {
       const char* line_format;
       int         pagelen;
       int         topmargin;
       int         headskip;
       int         titleskip;
       int         leftmargin; }
```

layout_format enthält die Parameter, die für das Layout während eines Druckvorgangs verwendet werden (s. Lösungskizze 8).

Invarianten: Es muss gelten $page\_len > topmargin + headskip + titleskip$.

line_format muss ein *printf*-Formatstring mit exakt drei „%s“ Spezifizierern, möglicherweise mit Angabe von Feldbreite und Padding, sein. Derüberhinaus dürfen keine Parameterspezifikationen im Formatstring vorkommen. Bezeichne l die Summe jeweils der Maxima aus der minimalen Feldlänge des ersten Spezifizierers und der maximalen Länge des Feldes *instr_text*–> *addr_text*, der minimalen Feldlänge des zweiten Spezifizierers und der maximalen Länge eines *menmonic* und der minimalen Feldlänge des dritten Spezifizierers und der maximalen Länge eines *instr_text*–> *oprd_text*. Bezeichne l' die Länge des Strings *line_format* ohne die Spezifierer. Dann muss $l + l' + leftmargin$ kleiner oder gleich der Zeilenlänge des physikalischen Ausgabegeräts in Zeichen sein.

Mit einem Wort: Unter keinen Umständen darf die Anwendung des Formatstrings *line_format* zu einem formatierten String führen, der länger ist als die Zeilenlänge des verwendeten Druckers.

Ein Beispiel für ein *line_format*: `"      %+4s   %-3s %-10s"`.

Prozedur *new_job***()**

```
job* new_job(layout_format* lf, const char* headline);
```

Beginnt einen neuen Layoutauftrag und gibt einen Zeiger auf das Statusobjekt des Auftrags zurück.

Ret.: *jbp*.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *lf* muss auf ein korrekt initialisiertes *layout_format* zeigen.

Acc.: **headline* und **lf* werden nur gelesen.

Post: *jbp* zeigt auf ein neues, gültig initialisiertes Layoutstatusobjekt.

Ress.: Die Verantwortung für die Freigabe des durch das Statusobjekt belegten Platzes liegt bei der aufrufenden Prozedur, darf aber nur durch *end_job*() erfolgen. Die Verantwortung für die Freigaben von **headline* und **lf* verbleibt bei der aufrufenden Prozedur, *new_job*() legt eine interne Kopie der benötigten Daten an.

Bem.: Es können durchaus mehrere Jobs parallel begonnen werden. Durch das Druck-Spooling des Host-Betriebssystems werden diese nach ihrem Abschluss hintereinander auf dem physikalischen Gerät ausgegeben.

Prozedur *print_line***()**

```
void print_line(job* jb, instr_texts::instr_text it);
```

Ausgabe und horizontales Layout einer Zeile.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *it* muss auf ein korrekt initialisierte Variable vom Typ *instr_texts*::*instr_text* verweisen.

jb muss auf ein korrekt initialisiertes Druckstatusobjekt verweisen.

Acc.: **it* wird nur gelesen, **jb* wird gelesen und verändert.

Post: *it−>mn.t*, *it−>addr_text* und *it−>oprd_text* werden mittels des Zeilenformats des Layoutauftrags zu einer Zeile arrangiert und ausgegeben.

Ist der bedruckbare Bereich der Seite zu Ende, wird automatisch umgebrochen. War die Seite zuvor leer, wird automatisch eine Kopfzeile generiert.

Ress.: Die Verantwortung für die Freigabe aller übergebenen Zeiger bleibt bei der aufrufenden Prozedur.

Excn.: Besitzt das Layoutstatusobjekt einen Fehlerstatus, so bleibt dieser erhalten und die Ausgabe unterbleibt.

Schlägt die Ausgabe fehl (etwa durch einen technischen Schaden am Gerät, das Fehlen von Papier, oder ein Problem mit dem Druckspooling des Hostsystems), so wird ein Fehlerstatus im Druckstatusobjekt gesetzt.

Prozedur *print_title***()**

```
void print_title(job* jb, const char* filename, int dsstart,
                                  int dslen, int csstart, int cslen);
```

Ausgabe des Titeltextes.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *filename* muss auf eine 0-terminierte Zeichenkette verweisen.

Acc.: **filename* wird nur gelesen, **jb* wird gelesen und verändert.

Post: Ein Titelabschnitt für das Listing in der Art, wie in der Aufgabenstellung gezeigt, wird ausgegeben.

Die Ausgabe wird intern wieder zeilenweise durchgeführt, so dass alle bei *print_line*() gemachten Bemerkungen über den automatischen Seitenumbruch sinngemäß gelten.

Ress.: Die Verantwortung für die Freigabe aller übergebenen Zeiger bleibt bei der aufrufenden Prozedur.

Excn.: Besitzt das Layoutstatusobjekt einen Fehlerstatus, so bleibt dieser erhalten und die Ausgabe unterbleibt.

Schlägt die Ausgabe fehl (etwa durch einen technischen Schaden am Gerät oder das Fehlen von Papier), so wird ein Fehlerstatus im Druckstatusobjekt gesetzt.

Prozedur *end_job***()**

```
int end_job(job* jb);
```

Eine eventuell noch unvollständige aktuelle Seite wird vervollständigt, der Layoutauftrag beendet und das Statusobjekt *jb* freigegeben.

Ret.: f.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *jb* muss auf ein korrekt initialisiertes Layoutstatusobjekt verweisen.

Acc.: **jb* wird gelesen.

Post: Die aktuelle Seite wird ausgeworfen, sofern sich der Druckkopf nicht gerade am Anfang der ersten Zeile dieser Seite befindet (d. h. ein Seitenauswurf unmittelbar voranging).

Der Layoutauftrag wird beendet, d. h. alle gepufferten Daten werden ausgegeben und der Druckauftrag auf dem unterliegenden Zeilendrucker geschlossen.

**jb* wird freigegeben. Danach verweist *jb* auf kein gültiges Layoutstatusobjekt mehr und darf von der aufrufenden Prozedur nicht mehr dereferenziert werden.

Der Rückgabewert f ist für gewöhnlich 1 um eine fehlerfreie Beendigung des Layouts anzuzeigen

Ress.: Die Verantwortlichkeit für die Freigabe von **jb* geht an das Modul layouter über.

Excn.: Besitzt das Layoutstatusobjekt einen Fehlerstatus oder treten beim Abschluss des Jobs Fehler auf, so ist der Rückgabewert $f = 0$.

Wie im Modul *line_printer* ist der Rückgabewert der Prozedur *end_job*() die einzige Art, auf die die Klientenprozedur erfahren kann, dass während des Layouts Fehler auftraten.

```
21    typedef ABSTRACT_HEAP_( job );

23    job*
24    new_job();

26    void
27    print_line(job* jb, const char* l);

29    void
30    next_page(job* jb);

32    int
33    end_job(job* jb);
```

Abbildung 11: Schnittstelle des Moduls *lineprinter*

5.12 Modul *lineprinter*

Verwendet intern *stdio, unistd*.

Dieses Modul definiert ein abstraktes Interface für den Drucker, welches einen einfachen Zeilendrucker emuliert. Ein solcher Drucker besitzt einen Druckkopf, welcher das Papier von oben nach unten zeilenweise bedruckt. Wird über den unteren Rand des Papiers hinausgedruckt, ist das Ergebnis undefiniert. Das Client-Modul muss also rechtzeitig einen Seitenvorschub auslösen. Die Länge einer Seite kann nicht abgefragt werden, das Client-Modul muss also die Kenntnis über die tatsächliche Länge einer Seite in Zeilen von anderswoher beziehen (z. B. aus einer Konfigurationsdatei).

Die Anzahl der Zeilen, die tatsächlich auf eine Seite ausgegeben werden können, wird im Folgenden als der *bedruckbare Bereich der Seite* bezeichnet.

Datentyp *job*

Abstraktes Heap-Objekt

Dieser Datentyp speichert den momentanen Zustand eines Druckauftrags, dazu gehört etwa die Position des (emulierten) Druckkopfes und Fehlerstatus. Dieser Zustand kann nicht direkt ausgelesen werden, er hat jedoch zum einen Einfluss auf die von den einzelnen Druckerprozeduren durchgeführten Operationen, zum anderen hängt von ihm das das Ergebnis von *end_job*() ab (siehe dort).

Prozedur *new_job***()**

`job* new_job();`

Beginnt einen neuen Druckauftrag und gibt einen Zeiger auf das Statusobjekt des Auftrags zurück.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Ret.: *jbp*.

Post: *jbp* zeigt auf ein neues, gültig initialisiertes Druckstatusobjekt. Der emulierte Druckkopf befindet sich am Beginn der obersten Zeile einer neuen Seite.

Ress.: Die Verantwortung für die Freigabe des durch das Statusobjekt belegten Platzes liegt bei der aufrufenden Prozedur, darf aber nur durch *end_job*() erfolgen.

Bem.: Es können mehrere Jobs parallel begonnen werden. Durch das Druck-Spooling des Host-Betriebssystems werden diese nach ihrem Abschluss korrekt hintereinander auf dem physikalischen Gerät ausgegeben.

Prozedur *print_line***()**

```
void print_line(job* jb, const char* l);
```

Ausgabe einer Zeile.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *l* muss auf eine 0-terminierte Zeichenkette verweisen, die kürzer ist als die Zeilenbreite des Druckers, und die nur druckbare ASCII-Zeichen (mit Code aus $\{32, \ldots, 127\}$) enthält.

jb muss auf ein korrekt initialisiertes Druckstatusobjekt verweisen.

Der Druckkopf muss sich am Anfang einer Zeile des bedruckbaren Bereichs der Seite befinden.

Acc.: **l* wird gelesen, **jb* wird gelesen und verändert.

Post: Die Zeichenkette in **l* wird in der aktuellen Zeile ausgegeben. Der Druckkopf geht an den Anfang der nächsten Zeile. Dies bedeutet, dass er, wenn die aktuelle Zeile die letzte Zeile des bedruckbaren Bereichs war, den bedruckbaren Bereich verlässt.

Ress.: Die Verantwortung für die Freigabe von **l* und die Deinitialisierung von **jb* bleiben bei der aufrufenden Prozedur.

Excn.: Besitzt das Druckerstatusobjekt einen Fehlerstatus, so bleibt dieser erhalten und **l* wird nicht ausgegeben.

Schlägt die Ausgabe fehl (etwa durch einen technischen Schaden am Gerät oder das Fehlen von Papier), so wird ein Fehlerstatus im Druckstatusobjekt gesetzt.

Prozedur *next_page***()**

```
void next_page(job* jb);
```

Die aktuelle Seite wird ausgeworfen.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *jb* muss auf ein korrekt initialisiertes Druckstatusobjekt verweisen.

Acc.: **jb* wird gelesen und verändert.

Post: Die aktuelle Seite wird ausgeworfen. Der Druckerkopf befindet sich dann am Anfang der ersten Zeile einer neuen Seite.

Ress.: Die Verantwortung für die Deinitialisierung von **jb* bleibt bei der aufrufenden Prozedur.

Excn.: Besitzt das Druckerstatusobjekt einen Fehlerstatus, so bleibt dieser erhalten die aktuelle Seite wird nicht ausgeworfen.

Schlägt die Ausgabe fehl (etwa durch einen technischen Schaden am Gerät oder das Fehlen von Papier), so wird ein Fehlerstatus im Druckstatusobjekt gesetzt.

Prozedur *end_job***()**

```
int end_job(job* jb);
```

Eine noch unvollständige aktuelle Seite wird ausgeworfen, der Job beendet und das Statusobjekt *jb* freigegeben.

Ret.: f.

Mod.: Zustand der programmexternen Welt: Drucker oder Druckerqueue.

Pre: *jb* muss auf ein korrekt initialisiertes Druckstatusobjekt verweisen.

Acc.: **jb* wird gelesen.

Post: Die aktuelle Seite wird ausgeworfen, sofern sich der Druckkopf nicht gerade am Anfang der ersten Zeile dieser Seite befindet (d. h. ein Seitenauswurf unmittelbar voranging).

Der Job wird beendet, d. h. ein Epilog von Steuerkommandos an das physikalische Gerät geschickt oder die Pipe zum Druckspooler geschlossen.

**jb* wird freigegeben. Danach verweist **jb* auf kein gültiges Druckerstatusobjekt mehr und darf von der aufrufenden Prozedur nicht mehr referenziert werden.

Der Rückgabewert f ist für gewöhnlich 1 um eine fehlerfreie Beendigung des Jobs anzuzeigen

Ress.: Die Verantwortlichkeit für die Freigabe von **jb* geht an das Modul lineprinter über.

Excn.: Besitzt das Druckerstatusobjekt einen Fehlerstatus oder treten beim Abschluss des Jobs Fehler auf, so ist der Rückgabewert $f = 0$.

Der Rückgabewert der Prozedur *end_job*() ist übrigens die einzige Art, auf die die Clientprozedur erfahren kann, dass während des Ausdrucks Fehler auftraten.

```
24    typedef struct{

26      int           len;
27      instrs::instr ins[]; }  instrs_list;

29    cpu::byte_count
30    parse(const programs::segment* s, instrs_list* il);

32    void
33    DEINIT_instr_list(instrs_list* il);

35    void
36    disassemble(const char* af,
37                instrs::instr in,
38                instr_texts::instr_text* it);
```

Abbildung 12: Schnittstelle des Moduls *disassembly*

5.13 Modul *disassembly*

Importiert öffentlich *programs*, *instrs*, *instr-texts*, *cpu*.

Dieses Modul enthält die grundlegenden Algorithmen zur Disassemblierung: Zum einen die Zerlegung des Codesegment in eine Folge von Instruktionen, zum anderen die Übersetzung einer Instruktion in eine Textdarstellung.

Datentyp *instrs_list*

```
struct {
       int             len;
       instrs::instr ins[]; }
```

Diese Datenstruktur ist ein Container für eine Folge von Instruktionen.

Rinv.: Die Speicherobjekte $*ins$ bis $*(ins + len\text{-}1)$ müssen gültig alloziert sein (also für einen Zugriff gültig sein) und als *instrs*::*instr* initialisiert sein.

Prozedur *parse***()**

```
cpu::byte_count parse(const programs::segment* s, instrs_list* il);
```

Zerlegt ein Segment in eine Folge von Instruktionen.

Ret.: *errind* (error indication).

Pre: *s* muss auf ein initialisiertes Speicherobjekt vom Typ *programs*::*segment* zeigen. *il* muss auf auf ein uninitialisiertes Speicherobjekt vom Typ *instrs_list* zeigen.

Acc.: **s* wird nur gelesen, **il* nur beschrieben.

Post: In der Liste **il* werden die Instruktionen aus dem Segment in der Reihenfolge aufsteigender Adressen abgelegt. Das bedeutet, dass die Byte-Darstellungen dieser Instruktionen aneinandergehängt wieder das Speicherabbild des Segments *s* ergeben müssen.

Wenn kein Fehler auftritt ist der Rückgabewert *errind* = -1.

Ress.: Die Verantwortung für die Freigabe von *s bleibt unverändert. Die Verantwortung für die Freigabe von *il bleibt bei der aufrufenden Prozedur.

Excn.: Beim Parsen wird, beginnend vom Anfang des Codesegments, das jeweils nächste (noch nicht geparste) Byte als Opcode identifiziert, und aus dem Opcode die Länge der Instruktion ermittelt. Daraus erhält man wiederum den Anfang der nächsten Instruktion.

Dabei kann es geschehen, dass ein Byte, das eigentlich den Anfang einer Instruktion bilden sollte, keinen Opcode darstellt (nicht alle Werte des Typs *cpu*::*byte* werden vom Prozessor als Opcodes verstanden). In diesem Fall kann mit dem Parsen nicht fortgefahren werden. Als Fehlerindikation ist dann *errind* der Offset des betreffenden *illegalen Opcode* vom Segmentanfang – das Auftreten eines solchen Fehlers kann also daran erkannt werden, dass *errind* ≥ 0.

Prozedur *disassemble***()**

```
void disassemble(const char* af,
                      instrs::instr in, instr_texts::instr_text* it);
```

Disassembliert eine Instruktion (d. h. überführt die Instruktion in eine Textdarstellung).

Pre: *af* muss auf eine 0-terminierte Zeichenkette verweisen, die ein gültiges *printf*-Format zur Formatierung genau einer Ganzzahl ist. Ein gültiges Format wäre beispielsweise: `"%04x"`.

it muss auf einen uninitialisierten Speicherplatz vom Typ *instr_texts*::*instr_text* verweisen.

Acc.: **af* wird nur gelesen, **it* nur beschrieben.

Post: **it* ist initialisiert und enthält eine Textdarstellung von *in* entsprechend dem übergebenen Format *af* und den in den Eigenschaften von Opcodes festgelegten Darstellungsformaten. Konkreter: *it*−>*addr_text* resultiert aus der Anwendung de Formates *af* auf $address\_of(in)$, *it*−>*oprd_text* aus der Anwendung des dem Operanden der Instruktion zugeordneten Operandenformates

$$oprd\_fmt\_of(operand\_of(in))$$

auf den Operanden, falls einer existiert (oder bleibt leer, wenn kein Operand existiert), während $it{->}mn = menmonic\_of(opcode\_of(in))$ sein wird.

Ress.: Verantwortlichkeiten für die Freigabe von **af* und **it* bleiben unverändert.

Prozedur *DEINIT_instr_list***()**

```
void DEINIT_instr_list(instrs_list* il);
```

Deinitialisiert eine Variable (Speicher) vom Typ *instrs_list*.

Pre: **il* muss initialisiert sein.

Acc.: **il* wird gelesen und beschrieben.

Post: **il* ist uninitialisiert, der assoziierte Speicher freigegeben.

Bem.: Wohlgemerkt: **il* wird nicht freigegeben. Lediglich der indirekt allozierte Speicher, dessen Zeiger in **il* gespeichert sind.

Ress.: Verantwortung für die Freigabe von **il* bleibt unverändert (bei der aufrufenden Prozedur).

```
23  typedef struct{

25    const char*             opcodes_filename;
26    const char*             addrmodes_filename;
27    layouter::layout_format lf;                } configuration;

29  void
30  set_default_config(configuration* cfgp);

32  void
33  read_config_files(configuration* cfgp);

35  char*
36  parse_commandline(int argc, char** argv, configuration* cfgp);

38  void
39  build_tables(const char* opcodes_filename,    \
                 const char* addrmodes_filename);

41  void
42  disassemble_file(const char* filename);

44  int
45  main(int argc, char** argv);
```

Abbildung 13: Prozeduren im Modul *main*

5.14 Modul *main*

Importiert öffentlich *disassembly*, *layouter*.

Verwendet intern *programs*, *addrmodes*, *opcodes*, *instrs*, *instr-texts*.

Dieses Modul ist das Hauptmodul des Programms *disas65*. Als solches besitzt es kein Interface in dem Sinne, dass Konstrukte exportiert werden. Ich dokumentiere hier trotzdem die wesentlichen Prozeduren dieses Moduls, um die Struktur der Applikation verständlich zu machen.

Im Wesentlichen ist der in beim Aufruf durch dieses Modul ausgeführte Verarbeitungsprozess der folgende:

1. Lesen der Konfigurationdateien.
2. Interpretation der übergebenen Kommandozeilenargumente.
3. Einlesen der Prozessordefinitionen.
4. Öffnen der Programmdatei.
5. Einlesen des Programms mit Hilfe der Prozeduren aus *programs*. Holen des Codesegements.
6. Parsen des Codesegments mittels *disassembly*::*parse*().
7. In einer Schleife: Disassemblieren der einzelnen Instruktionen in der erhaltenen Instruktionenliste. Ausgabe über das Modul *layouter*.

Daraus ergeben sich dann fast schon automatisch die folgenden Prozeduren als tragende Teile von *main*:

Datentyp *configuration*

```
struct{
       const char*             opcodes_filename;
       const char*             addrmodes_filename;
       layouter::layout_format lf; }
```

Enthält die Konfiguration des Disassemblers – das sind im Parameter, die meist implizit, z. B. über Konfigurationsdateien, übergeben werden.

Prozedur *set_default_config***()**

```
void set_default_config(configuration* cfgp);
```

Füllt die Struktur **cfgp* mit Default-Konfigurationsdaten aus.

Mod.: Heap.

Pre: *cfgp* muss auf ein gültig alloziertes Speicherobjekt des Typs *configuration* zeigen. **cfpg* muss uninitialisiert sein.

Acc.: **cfgp* wird beschrieben.

Post: Die Felder in *cfgp* werden auf im Programm hart kodierte Defaultwerte gesetzt. Welche das genau sind, sollte in einem separaten Dokument erläutert werden (Beschreibung des Benutzerinterface).

**cfgp* ist initialisiert.

Ress.: Verantwortung für Deinitialisierung und Freigabe von **cfgp* verbleibt bei der aufrufenden Prozedur.

Bemerkung: Der Entwurf ist hier nicht ganz vollständig. Das angeführte „separate Dokument“ existiert bisher nicht.

Prozedur *read_config_files***()**

```
void read_config_files(configuration* cfgp);
```

Liest die Konfigurationsdateien */usr/local/etc/dias65.conf* und *$HOME/.disas65* (in dieser Reihenfolge) und überschreibt die Konfigurationsdaten in **cfgp* mit den dort vorgefundenen.

Mod.: Heap.

Pre: *cfgp* muss auf ein gültig alloziertes Speicherobjekt des Typs *configuration* zeigen. **cfpg* muss initialisiert sein.

Acc.: **cfgp* wird gelesen und beschrieben.

Post: s. Beschreibung.

Ress.: Verantwortung für Deinitialisierung und Freigabe von **cfgp* verbleibt bei der aufrufenden Prozedur.

Excn.: Im Falle eines Fehlers (nichtlesbare Konfigurationsdateien, Syntaxfehler in den Konfigurationsdateien) wird eine Fehlermeldung auf die Standardfehlerausgabe ausgegeben und das Programm abgebrochen: Die Prozedur kehrt nicht zurück.

Prozedur *parse_commandline***()**

```
char* parse_commandline(int argc, char** argv, configuration* cfgp);
```

Interpretiert die Kommandozeileargumente und überschreibt die Konfigurationsdaten in **cfgp* mit den dort vorgefundenen.

Mod.: Heap, *extern char *optarg*, *extern int optind*, *opterr*, *optopt*. Die externen Variablen sind in der Standardbibliothek global definiert und werden durch die Standard-Bibliotheksfunktion *getopt*() modifiziert.

Pre: *cfgp* muss auf ein gültig alloziertes Speicherobjekt des Typs *configuration* zeigen. **cfpg* muss initialisiert sein.

Acc.: **cfgp* wird gelesen und beschrieben. **argv* wird modifiziert (wie in *get_opt*(3) dokumentiert).

Post: s. Beschreibung. Da zum Parsen der Argumente *get_opt*() aus der Standardbibliothek verwendet wird, ergeben sich Seiteneffekte wie dort beschrieben (insbesondere: Umordnung von **argv*, Setzen von *optind*, *opterr* und *optopt*).

Nach der Abarbeitung der Prozedur zeigt *optind* auf das erste nicht-optionale Kommandozeilenargument (den Dateinamen). Es darf nur ein nicht-optionales Argument geben.

Ress.: Verantwortung für Deinitialisierung und Freigabe von **cfgp* verbleibt bei der aufrufenden Prozedur.

Bem.: Welche Kommandozeilenargumente möglich sind muss noch in einem separaten Dokument erläutert werden.

Excn.: Im Falle eines Fehlers (falsche Kommandozeilenargumente) wird eine Fehlermeldung auf die Standardfehlerausgabe ausgegeben und das Programm abgebrochen: Die Prozedur kehrt nicht zurück.

Prozedur *build_tables***()**

```
void build_tables(const char* opcodes_filename,
                                    const char* addrmodes_filename);
```

Liest die Definitionsdateien für den Befehlssatz des Mikrokontrollers ein und initialisiert damit die internen Tabellen der Module *addrmodes* und *opcodes*.

Bem.: Das Format der Opcode-Definitionsdatei ist in der Anleitung dokumentiert. Das Format der Adressart-Definitionsdatei ist im Anschluss dokumentiert.

Pre: *opcodes_filename* und *addressmodes_filename* müssen auf 0-terminierte Zeichenketten zeigen. Beide Zeichenketten sollten die Namen von existierenden Dateien sein, welche die Definitionen im dokumentierten Format enthalten.

Acc.: **opcodes_filename* und **addressmodes_filename* werden nur gelesen.

Post: s. Beschreibung.

Ress.: Verantwortung für die Freigabe von **opcodes_filename* und **addressmodes_filename* bleibt bei der aufrufenden Prozedur.

Excn.: Im Falle eines Fehlers (Beschreibungsdateien existieren nicht oder haben nicht das richtige Format) wird eine Fehlermeldung auf die Standardfehlerausgabe ausgegeben und das Programm abgebrochen: Die Prozedur kehrt nicht zurück.

Prozedur *disassemble_file***()**

```
void disassemble_file(const char* filename);
```

Die Datei mit dem Namen **filename* wird eingelesen, disassembliert und auf den Drucker ausgegeben.

Pre: *filename* muss auf eine 0-terminierte Zeichenkette zeigen.

Acc.: **filename* wird nur gelesen.

Post: s. Beschreibung.

Ress.: Verantwortung für die Freigabe von **filename* bleibt bei der aufrufenden Prozedur.

Excn.: Im Falle eines Fehlers (Datei existiert nicht, Lesefehler, Fehler beim beim Parsen oder der Disassemblierung des Codesegments) wird eine Fehlermeldung auf die Standardfehlerausgabe ausgegeben und das Programm abgebrochen: Die Prozedur kehrt nicht zurück.

Format der Adressart-Definitionsdatei: Diese Datei besteht aus aufeinanderfolgenden Zeilen, welche jeweils einen Datensatz enthalten, der eine Adressart beschreibt. Die 3 Felder des Datensatzes sind voneinander durch Kommas getrennt, das dritte Feld kann Kommas enthalten. Im einzelnen sind die Felder:

1. Symbolische Bezeichnung der Adressart (wie in der Opcode-Definitionsdatei). Genau zwei Zeichen.
2. Länge des Operanden in Bytes. Genau 1 Zeichen, und zwar „0“, „1“ oder „2“.
3. Der Formatstring des Operandenformats. Zu Einschränkungen siehe die Dokumentatation von *oprdfmts*::*new_oprdfmt*.

Alle Datensätze müssen sich im ersten Feld (symbolische Bezeichnung der Adressart) unterscheiden.

Beipiel:

```
IY,2,$(%4x),Y
AC,0,
IM,1,#%2x
```

```
21    typedef unsigned char byte;
22    typedef int address;
23    typedef int byte_count;
```

Abbildung 14: Schnittstelle des Moduls *cpu*

5.15 Modul *cpu*

Dieses Modul definiert Datentypen, welche Charakteristika der Mikrocontroller-CPU beschreiben.

Datentyp *byte*

unsigned char

Dieser Typ stellt ein Byte der Mikrocontroller CPU dar.

Rinv.: Wertebereich $\{0,\ldots,255\}$.

Datentyp *address*

int

Dieser Typ kann eine Adresse der Mikrokontroller-CPU enthalten

Rinv.: Wertebereich $\{0,\ldots,65535\}$.

Datentyp *byte_count*

int

Dieser Type kann die Differenz zwischen zwei Adressen enthalten, gibt also die Länge eines Speicherabschnitts des Mikrokontrollers an.

Rinv.: Wertebereich $\{0,\ldots,65535\}$.

Rationale: Diese Typen können sinnvollerweise nicht abstrakt gemacht werden. Die meisten Klienten dieses Moduls verlassen sich darauf, dass *byte*, *address* und *byte_count* eine Interpretation als Ganzzahl haben. Zwar könnte mit jedem dieser Typen eine Prozedur definiert werden, mit der diese Konversion erledigt werden kann (und die Typen dann abstrakt definiert werden), allerdings stellt sich sofort die Frage nach der mindestens nötigen Größe des Zieltyps bei der Konversion, womit wir wieder am Anfang angelangt wären: Nämlich, dass es, da diese Typen in eine Ganzzahl überführt werden sollen, nötig ist, eine ober Schranke für die Größe dieser Ganzzahl anzugeben.

Ich habe mir hier diese Komplikationen erspart, und *byte*, *address* und *byte_count* als Typ-Aliases definiert, deren wesentlicher Zweck darin besteht, in den Klientenmodulen zu dokumentieren, in welcher Rolle ein Wert verwendet wird. Selbstverständlich müssen bei einer Änderung dieser Typen zu einem größeren Typ hin alle Module, die *cpu* importieren, überprüft werden, ob dort die Aufweitung verwendeter Ganzzahltypen nötig ist.

A Modulschnittstellen / Header-Dateien

Die folgenden Abschnitte geben die vollständigen Headerdateien dieses Entwurfs in alphabetischer Reihenfolge wieder. Die Zeilennummern entsprechen denen des Release *disas65-2002-10-22-a*. Leerzeilen wurden nicht nummeriert und mehrere aufeinanderfolgende Leerzeilen zu einer zusammengefasst. Überlange Zeilen wurden manuell umgebrochen, aber die jeweiligen Folgezeilen nicht nummeriert.

A.1 addrmodes.hh

```
14  #ifndef H_INCLUDED_addrmodes_HH
15  #define H_INCLUDED_addrmodes_HH

17  #include "MARKUP.h"

19  #include "oprdfmts.hh"

23  namespace addrmodes {

25    typedef ABSTRACT_VAL_(int)    addrmode;

28    addrmode from_string(const char* txtp);

30    oprdfmts::oprdfmt
31    oprd_fmt_of(addrmode am);

33    int
34    oprd_len(addrmode am);

37    /* Module / ADT initialization */

39    addrmode
40    add_addrmode(oprdfmts::oprdfmt* ofp, int oprd_len, \
                                           const char* textrepr);

42    void
43    init_module();

45    void
46    deinit_module();
47  }

49  #endif /*  H_INCLUDED_addrmodes_HH */
```

A.2 cpu.hh

```
14  #ifndef H_INCLUDED_cpu_HH
15  #define H_INCLUDED_cpu_HH

17  #include "MARKUP.h"

19  namespace cpu {

21    typedef unsigned char byte;
22    typedef int address;
23    typedef int byte_count;
24  }

26  #endif /*  H_INCLUDED_cpu_HH */
```

A.3 disassembly.hh

```
14  #ifndef H_INCLUDED_disassembly_HH
15  #define H_INCLUDED_disassembly_HH

17  #include "programs.hh"
18  #include "instrs.hh"
19  #include "instr-texts.hh"
20  #include "cpu.hh"

22  namespace disassembly {

24    typedef struct{

26      int           len;
27      instrs::instr ins[]; }  instrs_list;

29    cpu::byte_count
30    parse(const programs::segment* s, instrs_list* il);

32    void
33    DEINIT_instr_list(instrs_list* il);

35    void
36    disassemble(const char* af,
37                instrs::instr in,
38                instr_texts::instr_text* it);

40  }

42  #endif /*  H_INCLUDED_disassembly_HH */
```

A.4 instr-texts.hh

```
14  #ifndef H_INCLUDED_instr_texts_HH
15  #define H_INCLUDED_instr_texts_HH

17  #include "opcodes.hh"

19  namespace instr_texts {

21    typedef struct{
22      const char*         addr_text;
23      opcodes::mnemonic   mn;
24      const char*         oprd_text; }    instr_text;

26    void
27    INIT_instr_text(instr_text* it,
28                    const char* addr_text,
29                    opcodes::mnemonic mn,
30                    const char* oprd_text);

32    void
33    DEINIT_instr_text(instr_text* it);

35  }

37  #endif /*  H_INCLUDED_instr_texts_HH */
```

A.5 instrs.hh

```
14  #ifndef H_INCLUDED_instrs_HH
15  #define H_INCLUDED_instrs_HH

17  #include "cpu.hh"
18  #include "opcodes.hh"
19  #include "operands.hh"

21  namespace instrs {

23    typedef ABSTRACT_VAL_( struct{

25      cpu::address      ad;
26      opcodes::opcode   oc;
27      bool              has_oprd;
28      operands::operand op;        })    instr;

31    instr
32    from_opcode(cpu::address ad, opcodes::opcode oc);

34    instr
35    with_operand(cpu::address ad, opcodes::opcode oc,     \
                                    operands::operand op);
```

```
37    opcodes::opcode
38    opcode_of(instr in);

40    operands::operand
41    operand_of(instr in);

43    cpu::address
44    address_of(instr in);

46    bool
47    has_oprd(instr in);

50  }

52  #endif /*  H_INCLUDED_instrs_HH */
```

A.6 layouter.hh

```
14  #ifndef H_INCLUDED_layouter_HH
15  #define H_INCLUDED_layouter_HH

17  #include "MARKUP.h"

19  #include "instr-texts.hh"

21  namespace layouter {

23    typedef ABSTRACT_HEAP_( job );

25    typedef struct{
26      const char* line_format;
27      int         pagelen;
28      int         topmargin;
29      int         headskip;
30      int         titleskip;
31      int         leftmargin; } layout_format;

33    job*
34    new_job(layout_format* lf, const char* headline);

36    void
37    print_title(job* jb,
38                const char* filename,
39                int         dsstart,
40                int         dslen,
41                int         csstart,
42                int         cslen      );

44    void
45    print_line(job* jb,
46               instr_texts::instr_text it);
```

```
48    int
49    end_job();

52  }

54  #endif /*  H_INCLUDED_layouter_HH */
```

A.7 lineprinter.hh

```
14  #ifndef H_INCLUDED_lineprinter_HH
15  #define H_INCLUDED_lineprinter_HH

17  #include "MARKUP.h"

19  namespace lineprinter {

21    typedef ABSTRACT_HEAP_( job );

23    job*
24    new_job();

26    void
27    print_line(job* jb, const char* l);

29    void
30    next_page(job* jb);

32    int
33    end_job(job* jb);

35  }

37  #endif /*  H_INCLUDED_lineprinter_HH */
```

A.8 main.cc

```
14  #include "layouter.hh"
15  #include "disassembly.hh"

17  #include "programs.hh"
18  #include "addrmodes.hh"
19  #include "opcodes.hh"
20  #include "instrs.hh"
21  #include "instr-texts.hh"

23  typedef struct{

25    const char*             opcodes_filename;
26    const char*             addrmodes_filename;
27    layouter::layout_format lf;                } configuration;
```

```
29  void
30  set_default_config(configuration* cfgp);

32  void
33  read_config_files(configuration* cfgp);

35  char*
36  parse_commandline(int argc, char** argv, configuration* cfgp);

38  void
39  build_tables(const char* opcodes_filename,    \
                 const char* addrmodes_filename);

41  void
42  disassemble_file(const char* filename);

44  int
45  main(int argc, char** argv);
```

A.9 opcodes.hh

```
14  #ifndef H_INCLUDED_opcodes_HH
15  #define H_INCLUDED_opcodes_HH

17  #include "MARKUP.h"

19  #include "cpu.hh"
20  #include "oprdfmts.hh"

22  namespace opcodes {

24    typedef ABSTRACT_VAL_(int)    opcode;
25    typedef struct{ char t[4] ;} mnemonic;   \
                                               \
                                /* mnemonic.t[3]==0 always */

27    opcode          from_byte(cpu::byte b);

29    mnemonic
30    mnemonic_of(opcode oc);

32    oprdfmts::oprdfmt
33    oprd_fmt_of(opcode oc);

35    int
36    instr_len(opcode oc);

38    int
39    is_legal(cpu::byte b);
```

```
41    void
42    define_opcode(cpu::byte b,
43                  mnemonic m,
44                  oprdfmts::oprdfmt of,
45                  int instr_len);

47    void
48    init_module();

50    void
51    deinit_module();
52  }

54  #endif /*  H_INCLUDED_opcodes_HH */
```

A.10 operands.hh

```
14  #ifndef H_INCLUDED_operands_HH
15  #define H_INCLUDED_operands_HH

17  #include "MARKUP.h"

19  #include "cpu.hh"
20  #include "oprdfmts.hh"

22  namespace operands {

24    typedef ABSTRACT_VAL_( struct{

26      int size;
27      int val;                  }) operand;

29    operand
30    from_bytes(int size, cpu::byte* b);

32    char *               /* freshy malloc'ed */
33    format(oprdfmts::oprdfmt* ofp, operand od);

36    /* you don't need these, my friend

38    int
39    val_of(operand od);

42    int
43    size_of(operand od);

45    */

47  }

50  #endif /*  H_INCLUDED_operands_HH */
```

A.11 oprdfmts.hh

```
14  #ifndef H_INCLUDED_oprdfmts_HH
15  #define H_INCLUDED_oprdfmts_HH

17  #include "MARKUP.h"

20  namespace oprdfmts {

22    typedef ABSTRACT_HEAP_( oprdfmt );

24    int
25    max_digits(oprdfmt* ofp);

27    int
28    max_len(oprdfmt* ofp);

30    void
31    format(oprdfmt* ofp, int, char* buf);

33    /* ^ for this to work, every value \
         (read: operand) formatted must
34       fit into an int */

36    oprdfmt*
37    new_oprdfmt(const char* fmtstr);

39    /* ^ must be a 0-terminated string of the format

41       ^[^%\\]*%0[1-9]x[^%\\]*$

43    */

45    void
46    free_oprdfmt(oprdfmt* ofp);

48  }

51  #endif /*  H_INCLUDED_oprdfmts_HH */
```

A.12 programs.hh

```
14  #ifndef H_INCLUDED_programs_HH
15  #define H_INCLUDED_programs_HH

17  #include "MARKUP.h"
18  #include <stdio.h>

20  #include "cpu.hh"

22  namespace programs {
```

```
24    typedef ABSTRACT_HEAP_( program ); /* immutable */
25    typedef ABSTRACT_HEAP_( segment ); /* immutable */

27    program*
28    NEW_from_file(FILE* f);

30    const segment*
31    codeseg_of(program* p);

33    const segment*
34    dataseg_of(program* p);

36    void
37    FREE(program* p);

39    /* segments */

41    cpu::byte_count
42    length_of(const segment* s);

44    cpu::address
45    address_of(const segment* s);

47    const cpu::byte*
48    image_of(const segment* s);

50  }

52  #endif /*  H_INCLUDED_programs_HH */
```

Literatur

[Stroustrup 1998] STROUSTRUP, Bjarne: *Die C++-Programmiersprache*. Dritte Auflage. Addison-Wesley-Longman, 1998. – Dt. Übersetzung aus dem Englischen. – ISBN 0-201-88954-4

[TeachSWT@Tü 2002a] Wilhelm Schickart Institut (Veranst.): *Übungsmaterial, Fallbeispiele und Ergänzungen zur Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `teachswt-tue-2002-a.tar.gz`

[TeachSWT@Tü 2002b] LEYPOLD, M E.: Lösungsskizze 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002a). – `handouts/loesung-02.vdm`

[TeachSWT@Tü 2002c] LEYPOLD, M E.: Module und Namensräume. In: (TeachSWT@Tü, 2002a). – Quelle `handouts/modules-and-namespaces.tex`