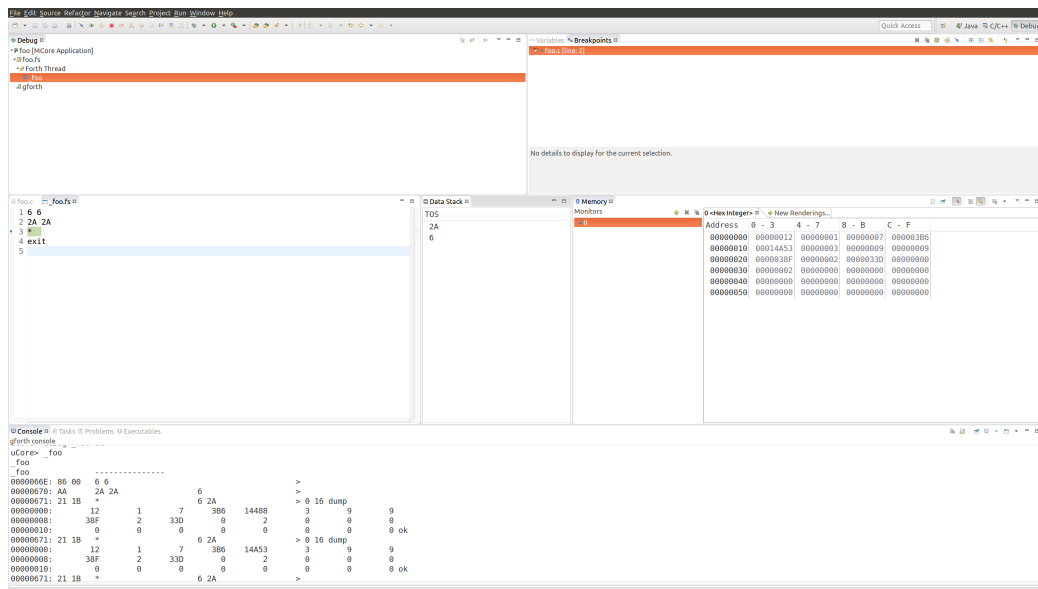


Bachelor-Thesis

Eclipse Entwicklungsumgebung für MicroCore

Benjamin Neukom

August 2015



Betreuer: Carlo Nicola

Inhaltsverzeichnis

1. Einleitung	6
2. Die Eclipse Plattform	7
2.1. Eclipse Plattform	7
2.2. Plugins	7
2.3. Extension Points	7
2.4. Eclipse basierte MCore Entwicklungsumgebung	8
2.4.1. JDT	8
2.4.2. Xtext	8
2.4.3. Dynamic Language Toolkit Framework	9
2.4.4. Eclipse C/C++ Development Tools	9
2.4.5. Verwendung für uCore Eclipse	9
3. Compiler Integration	10
3.1. Integration im Eclipse CDT	10
3.1.1. Configuration Beschreibung	10
3.1.2. Configuration Verwendung	10
3.1.3. Tool Beschreibung	10
3.1.4. Tool Verwendung	10
3.1.5. Toolchain Beschreibugn	10
3.1.6. Toolchain Verwendung	11
3.1.7. CDT Builder Beschreibung	11
3.1.8. CDT Builder Verwendung	11
3.1.9. Project Builder	11
4. Programm Launch	12
4.1. Launch	12
4.1.1. Run	13
4.1.2. Debug	13
5. Forth Kommunikation	14
5.1. Prozess Kommunikation	14
5.1.1. GDB/MI-Commands	14
5.1.2. Direkte Kommunikation mit dem Prozess	14
5.2. Kommunikation	15
5.3. API Design	15
5.4. Implementierung	15
5.4.1. Klassenbeschreibung	17

5.5. Testing	18
6. uForth Editor	19
6.1. Xtext Implementation des uForth Editor	19
6.2. uForth Editor	20
6.2.1. Wörter Dokumentation	21
6.2.2. Editor Outline	21
7. Debugger	22
7.1. Breakpoints	22
7.1.1. Per Konsole	22
7.1.2. Im Source Code	22
7.2. Konsolen basierter Debugger	23
7.3. Forth Debugger	23
7.3.1. CDT oder JDT Debugging Mechanismen	23
7.3.2. Debugger Aktionen	23
7.3.3. Stack View	26
7.3.4. Memory View	26
7.4. C-Debugger	27
8. Entwicklungsumgebungs Einstellungen	28
8.1. Umbilical	28
8.2. Loader	28
9. Optimierungen	29
9.1. Peephole Optimierung	29
9.1.1. Beispiele	30
9.1.2. Optimierungen	31
9.1.3. Automatische Generierung von Peephole Optimierungen	32
9.1.4. Constant Propagation	32
9.1.5. Resultate und Tests	32
9.1.6. Mögliche Erweiterungen	33
A. Literaturverzeichnis	34
B. Abbildungsverzeichnis	35
C. Tabellenverzeichnis	37
D. Installationsanleitung für Endanwender	38
E. Installationsanleitung für Entwickler	39
F. Forth Test-Funktionen	41
F.1. _Init	41

F.2. Hash	44
F.3. Update	45
F.4. Propagation	45
G. Ehrlichkeitserklärung	47

Im ersten Teil dieser Arbeit wird beschrieben, wie eine auf Eclipse basierende Entwicklungsumgebung implementiert werden kann. Es wird beschrieben, welche Features Eclipse bereitstellt um eine Entwicklungsumgebung darauf aufzubauen. Im zweiten Teil werden verschiedene Peephole Optimierungen, unter anderem der vom Davidson und Fraser beschriebene Optimierer PO, für den Compiler untersucht und mit der aktuellen Peephole Implementation des Forth-Cross Compilers verglichen.

1. Einleitung

MicroCore ist eine CPU mit Harvard-Architektur, die eine Teilmenge der Forth-Sprache als Maschinensprache benutzt. MicroCore wurde in Zusammenarbeit mit der Firma Send GmbH in Hamburg entwickelt. Für die Version 1.71 wurde ein C-Compiler mit SCC (Stack-C-Compiler) implementiert. Ziel dieser Arbeit ist, eine auf Eclipse basierte Entwicklungsumgebung zu entwickeln, welche den SCC als Compiler verwendet. Es sollte möglich sein, den ganzen Prozess, vom Quellcode bis zur Kompilierung und dem Herunterladen auf das HW-Target in der Entwicklungsumgebung durchführen zu können. Im ersten Kapitel „Die Eclipse Plattform“ wird beschrieben, was Eclipse als Plattform für Möglichkeiten bietet um eine Entwicklungsumgebung zu entwickeln. Im Kapitel „Compiler Integration“ wird erklärt, wie der Compiler integriert wurde. Im dritten Kapitel „Programm Launch“ wird beschrieben, wie das kompilierte uForth Programm aus der Entwicklungsumgebung gestartet werden kann und was dafür implementiert wurde. Im Kapitel „Forth Kommunikation“ wird beschrieben, wie die Kommunikation mit dem Forth Prozess, designt, implementiert und getestet wurde. Im nächsten Kapitel „uForth Editor“ wird beschrieben, wie Xtext dazu verwendet wurde um einen uForth Editor in der Entwicklungsumgebung zu integrieren. Im Kapitel „Debugger“ wird gezeigt, auf welche Arten der Debugger in die Entwicklungsumgebung eingebunden wurde und wie diese noch erweitert werden könnten. Im nächsten Kapitel „Entwicklungsumgebungs Einstellungen“ werden die Einstellungen, welche in der Entwicklungsumgebung vorgenommen werden können, beschrieben. Im Kapitel „Optimierungen“ werden einige Peephole Optimierungen für den Compiler vorgestellt und mit der aktuellen Peephole Optimierung des Forth-Cross-Compilers verglichen.

2. Die Eclipse Plattform

In diesem Kapitel wird gezeigt, was Eclipse für Möglichkeiten bietet, um eine moderne Entwicklungsumgebung zu implementieren. Die verschiedenen Möglichkeiten werden miteinander verglichen und die Vor- und Nachteile aufgezeigt. Auch werden die wichtigsten Eclipse Features, welche für das Entwickeln von Eclipse Rich Client Platform (RCP) Applikationen benötigt werden, beschrieben.

2.1. Eclipse Plattform

Eclipse RCP bietet eine Basis um beliebige, nicht zwingendermassen Entwicklungsumgebungen, Betriebssystem unabhängige, Applikationen zu entwickeln. Es bietet Mechanismen, wie Plugins und Extension Points, um modulares programmieren zu unterstützen und vereinfachen. Auch bietet das Framework Features wie Views, Editoren und Perspektiven welche häufig in Applikationen gebraucht werden.

2.2. Plugins

Eclipse Applikationen nutzen eine auf der OSGi Spezifikation basierten Runtime. Eine Komponente in dieser Runtime ist ein Plugin. Eine Eclipse RCP Applikation besteht aus einer Ansammlung dieser Plugins. [1] Plugins haben einen Lebenszyklus und können zur Laufzeit geladen oder entladen werden. Ein Eclipse Plugin kann Extension Points anderer Plugins ansteuern und eigene Extension Points oder APIs anbieten.

2.3. Extension Points

Um Plugins erweitern zu können, bietet Eclipse das Konzept von Extension Points an. Über Extension Points können Plugins eine bestimmte Funktionalität anbieten, welche von anderen Plugins angesteuert werden kann.

Eclipse bietet ein Plugin an, welches einen Extension Point für Views definiert. Andere Plugins können über diesen Extension Point, eine neue View erstellen und diese konfigurieren. [2]

In der Abbildung 2.1 ist zu sehen, wie über diesen Extension Point eine neue Eclipse View erstellt werden kann.

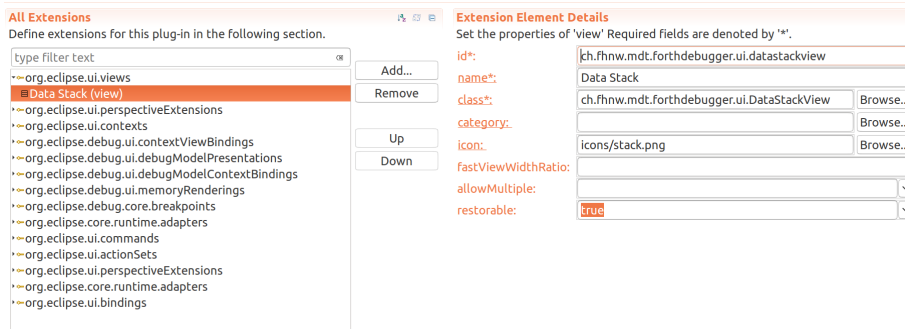


Abbildung 2.1.: Ein Plugin, welches einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.

Es ist auch möglich eigene Extension Points zu definieren, falls ein Plugin für andere Entwickler offen stehen soll für Erweiterungen.

2.4. Eclipse basierte MCore Entwicklungsumgebung

Eclipse als Grundlage für eine Entwicklungsumgebung zu verwenden eignet sich besonders gut, da Eclipse schon einiges an Funktionalität für eine Entwicklungsumgebung zur Verfügung stellt und schon viele Entwicklungsumgebungen (PHP, C/C++, Python und D unter anderem) basierend auf Eclipse entwickelt wurden. Auch existieren einige Tools, auf welche ich noch genauer eingehen werde, wie Xtext und DLTk, welche das entwickeln einer Entwicklungsumgebung weiter vereinfachen. In folgenden Kapiteln wird kurz auf die Möglichkeiten, welche von Eclipse bereitgestellten Technologien verwendet werden könnten um die Entwicklungsumgebung zu implementieren.

2.4.1. JDT

Eine Möglichkeit die MCore Entwicklungsumgebung zu implementieren wäre, die Standard Features von dem Eclipse Java Development Tools (JDT) zu verwenden. Dies wäre sehr Aufwendig, da alle Features für C, wie Syntax Highlighting, Code Completion oder eine Outline, neu implementiert werden müssten.

2.4.2. Xtext

XText ist ein Framework, welches es erleichtert, eine auf Eclipse basierte Entwicklungsumgebungen zu programmieren. Es ermöglicht auf schnelle Art und Weise das Grundgerüst einer Entwicklungsumgebung mit Features wie:

- Editor mit Syntax Coloring

- Code Completion
- Compiler Integration
- Java-basierter Debugger
- Outline
- Indexing

zu generieren. [3] Es muss lediglich eine ANTLR [4] ähnliche Grammatik für die Sprache definiert werden. Für den C-Editor eine Grammatik zu definieren ist schwierig, da dazu auch noch der Präprozessor mit einbezogen werden müsste. Für den Forth Editor kann Xtext verwendet werden. Im Kapitel uForth Editor wird beschrieben, wie Xtext verwendet wurde um einen Forth Editor zu implementieren.

2.4.3. Dynamic Language Toolkit Framework

Das Dynamic Language Toolkit (DLTK) ist ein weiteres Framework, welches das Grundgerüst für eine Entwicklungsumgebung generieren kann. Ursprünglicherweise war das Framework nur für dynamische Sprachen geeignet, es kann aber auch für statische Sprachen verwendet werden. Die D Entwicklungsumgebung wurde mit dem DLTK realisiert [5]. Es bestehen aber wieder dieselben Nachteile wie bei Xtext. Da C schwierig zu parsen ist, müsste trotz dem Framework noch viel selbst implementiert werden. Die Sprache D verwendet keinen Präprozessor, deshalb konnte für diese Entwicklungsumgebung das DLTK Framework verwendet werden.

2.4.4. Eclipse C/C++ Development Tools

Das Eclipse C/C++ Development Tools (CDT), ist eine Eclipse Distribution mit Unterstützung für C und C++. Das CDT bietet viele Features, welche schon vom JDT bekannt sind, für C an und stellt Extension Points zur Verfügung um diese zu verwenden. So kann mit relativ wenig Aufwand einen neuen Compiler in die Entwicklungsumgebung eingebunden werden. Dies wurde auch schon mehrfach gebraucht, um verschiedene C/C++ Compiler im Eclipse CDT zu integrieren. Auf die Einbindung des Compilers wird im Kapitel ?? genauer eingegangen.

2.4.5. Verwendung für uCore Eclipse

Ich habe mich dazu entschieden, das Eclipse CDT als Target Platform zu wählen. Somit können alle Features, welche das Eclipse CDT zur Verfügung stellt, für die C Entwicklung gebraucht werden. Xtext kann zusätzlich noch verwendet werden, um den Forth Editor zu implementieren.

3. Compiler Integration

In diesem Kapitel wird beschrieben, wie der Compiler in die Entwicklungsumgebung eingebunden wurde um ein C-File nach Forth zu übersetzen und welche Möglichkeiten mit dem CDT für das Editieren von C-Files zur Verfügung stehen.

3.1. Integration im Eclipse CDT

Das CDT stellt Extension Points zur Verfügung, welche gebraucht werden können um einen Compiler zu integrieren. Diese Extension Points werden in den nächsten Kapiteln erläutert und es wird erklärt wie der LCC dadurch in die Entwicklungsumgebung eingebunden wird.

3.1.1. Configuration Beschreibung

Eine Konfiguration wird verwendet, um verschiedene standard Tools und Optionen bereitzustellen, welche ein Projekt auf eine gewisse Weise zu kompilieren. Normalerweise existieren für ein Projekt zwei Konfigurationen, eine Debug- und eine Releasekonfiguration.

3.1.2. Configuration Verwendung

Für die Entwicklungsumgebung existiert nur eine Konfiguration für den Release Build. Debug spezifische Files werden von dem Launch generiert.

3.1.3. Tool Beschreibung

Definiert ein Tool, wie zum Beispiel ein Compiler oder Linker, welches vom Buildprozess verwendet wird.

3.1.4. Tool Verwendung

In der Entwicklungsumgebung wird nur ein Tool verwendet, welches den LCC Compiler aufruft und somit das Forth File generiert. Für dieses Tool wurde noch eine Option (-S-Q) definiert, welche den Peephole Optimizer des Compilers deaktiviert.

3.1.5. Toolchain Beschreibugn

Eine Liste von Tools welche gebraucht werden um den Output des Projekts zu generieren.

3.1.6. Toolchain Verwendung

Die von der Entwicklungsumgebung verwendete Toolchain, beinhaltet nur das vorhin beschriebene Tool, um den LCC Compiler aufzurufen.

3.1.7. CDT Builder Beschreibung

Repräsentiert das Werkzeug, welches verwendet wird um den Buildprozess zu steuern. Typischerweise eine Variante von `make`.

3.1.8. CDT Builder Verwendung

Für die uCore Entwicklungsumgebung wurde der Standard CDT Builder verwendet. Zusätzlich wurden aber die File-Endung der vom Compiler generierten Forth Files auf `.fs` geändert.

3.1.9. Project Builder

Der Project Builder wird nicht vom Eclipse CDT, sondern vom normalen Eclipse Buildprozess zur Verfügung gestellt. Für die uCore Entwicklungsumgebung wird der Project Builder verwendet, um Fehler in den Umgebungsvariablen zu finden. Der Project Builder überprüft:

- Ob das Programm `lcc-mcore` im Pfad zu finden ist.
- Ob eine Umgebungsvariable mit dem Namen `GFORTHPATH` existiert
- Ob der letzte Pfad der Umgebungsvariable `GFORTHPATH` zu einem gültigen Ordner zeigt

Falls einer dieser Schritte fehl schlägt, wird der Build abgebrochen und die Fehler werden in der Problems View von Eclipse angezeigt. In folgender Abbildung 3.1 wird ein Fehler im Pfad gezeigt.

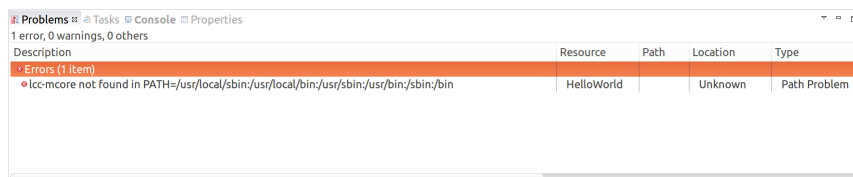


Abbildung 3.1.: Die Eclipse Problems View zeigt, dass das Program `lcc-mcore` nicht gefunden wurde im Pfad.

In der Installationsanleitung im Anhang befinden sich alle Schritte, welche notwendig sind um die Entwicklungsumgebung zu installieren.

4. Programm Launch

In diesem Kapitel wird beschrieben, wie das kompilierte uForth Programm aus der Entwicklungsumgebung gestartet werden kann und was dafür implementiert wurde.

4.1. Launch

In Eclipse wird zwischen zwei Launchmöglichkeiten unterschieden, deren Unterschiede in den nächsten Kapiteln aufgezeigt und erklärt werden.

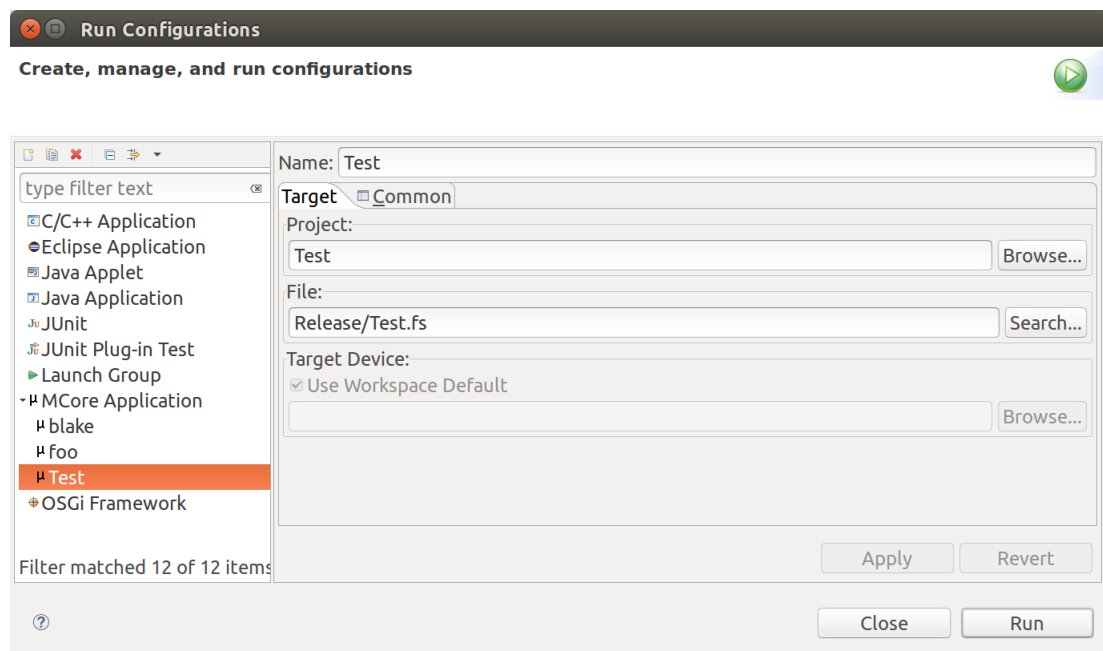


Abbildung 4.1.: Der Launch Configuration Dialog. Im Dialog kann eine neue MCore Launch Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen muss das Projekt und das Forth File, welches gestartet werden soll angegeben werden.

4.1.1. Run

Mit der Run Konfiguration wird zuerst der Loader in den Forth Workspace kopiert, welcher in den Umgebungsvariablen definiert sein muss. Danach wird der Forth Prozess gestartet und der Umbilical Port mittels `Umbilical: /dev/ttyUSBX` gesetzt. Der Umbilical Port und der Loader müssen in der MCore Preference Page gesetzt werden (siehe Kapitel Entwicklungsumgebungs Einstellungen). Falls der Umbilical Port ungültig ist, wird beim starten des Programmes eine Fehlermeldung gezeigt.

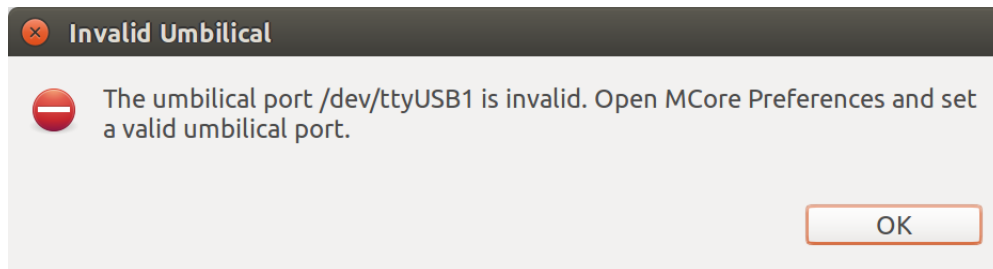


Abbildung 4.2.: Dialog, welcher angezeigt wird, wenn ein Program gestartet wird und der Umbilical Port ungültig ist. Der Port kann in der MCore Preference Page geändert werden.

4.1.2. Debug

In der Debug Konfiguration werden zusätzlich noch alle Funktionen des gestarteten Forth Files disassembled. Dies wird gemacht, damit der Debugger die Funktionen in einem File anzeigen kann. Es kann nicht der vom Compiler generierte Code genommen werden, da der Forth Cross-Compiler noch Optimierungen am Code vornimmt.

5. Forth Kommunikation

In diesem Kapitel wird beschrieben, wie die Entwicklungsumgebung mit dem Forth Prozess kommuniziert. Die Kommunikation mit dem Forth Prozess ist von zentraler Bedeutung, da viel der Funktionalität der Entwicklungsumgebung davon abhängt, dass die Kommunikation stabil läuft. Es wird gezeigt, wie die Kommunikation designt, implementiert und getestet wurde.

5.1. Prozess Kommunikation

Um mit dem Prozess zu kommunizieren, gibt es grundsätzlich zwei Möglichkeiten. Die erste ist ein Machine Interface zu implementieren, oder direkt mit dem Prozess kommunizieren. In folgenden Kapiteln werden die beiden Möglichkeiten kurz beschrieben.

5.1.1. GDB/MI-Commands

Eine Möglichkeit mit dem Prozess zu kommunizieren wäre, ein MachineInterface (MI) wie es für den GDB implementiert wurde, zu gebrauchen. GDB/MI ist ein Linien basiertes Maschinen orientiertes Text Interface zu dem GDB. Es wurde dazu entwickelt, damit der GDB als Debugger in ein grössers System einfacher einbindbar ist. [6] Eine MI ähnliches Interface wäre mit grossem Aufwand verbunden, da das Interface zuerst definiert werden müsste. Dafür könnte aber viel des CDT Debugging Mechanismus verwendet werden, da der CDT Debugger auch auf dem GDB/MI basiert. Auch ist der CDT Debugger sehr kompliziert [7] und es wäre ein grosser Einarbeitungsaufwand notwendig um den Forth Debugger darauf basieren zu können. [7]

5.1.2. Direkte Kommunikation mit dem Prozess

Eine weitere Möglichkeit wäre, direkt die Befehle an den Prozess senden und auf Antworten warten. Dafür müsste einige Klassen implementiert werden, um das Kommunizieren zu vereinheitlichen und vereinfachen.

Probleme bei der Kommunikation

Bei der Kommunikation mit dem Prozess können einige Probleme Auftreten. Es kann sein, dass der Prozess plötzlich keine Antworten mehr gibt. Auch weiss man nicht, wie lange es dauern wird, bis der Prozess Antwort gibt. Diese Probleme müssen mit den im vorigen genannten Klassen gelöst werden. TODO HOW?!

5.2. Kommunikation

Ich habe mich dafür entschieden, die Kommunikation nicht über ein MI Interface zu implementieren. Der Aufwand ein solches Interface zu designen und implementieren, sowie die Einarbeitungszeit für den CDT Debugger, wären zu gross. Die Kommunikation wird also direkt mit dem Prozess erfolgen. Das Design und die Implementation wird in folgenden Kapiteln besprochen.

5.3. API Design

In einem ersten Schritt wurde ein API designt, welches verwendet werden soll um die Kommunikation mit dem Prozess möglichst einfach zu halten.

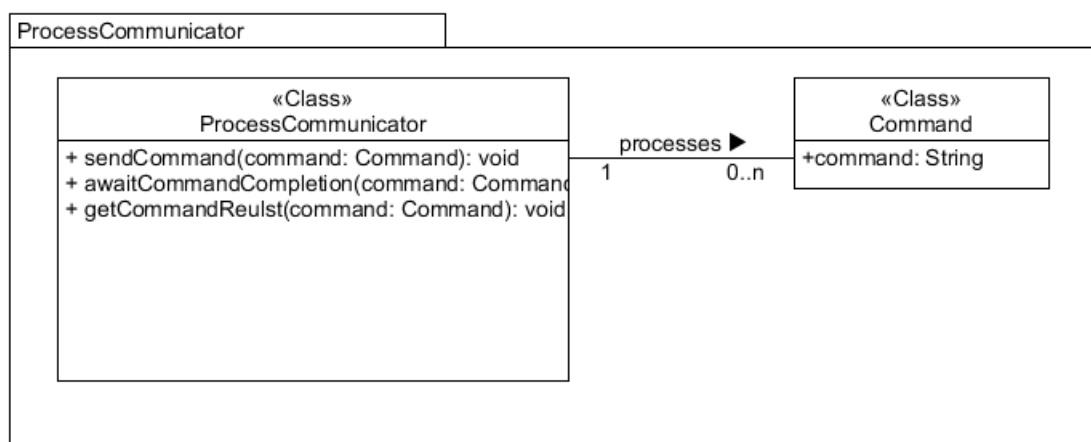


Abbildung 5.1.: Ein erstes Design des Prozess API. Die zentrale Kommunikation geschieht über die Communicator Klasse. Es können Commands abgesetzt werden und es kann auf Resultate gewartet werden.

5.4. Implementierung

Bei der Implementierung kamen dann einige Änderungen hinzu. In folgendem Klassendiagramm sind die Änderungen ersichtlich. Danach wird die Klassenstruktur genauer erläutert.

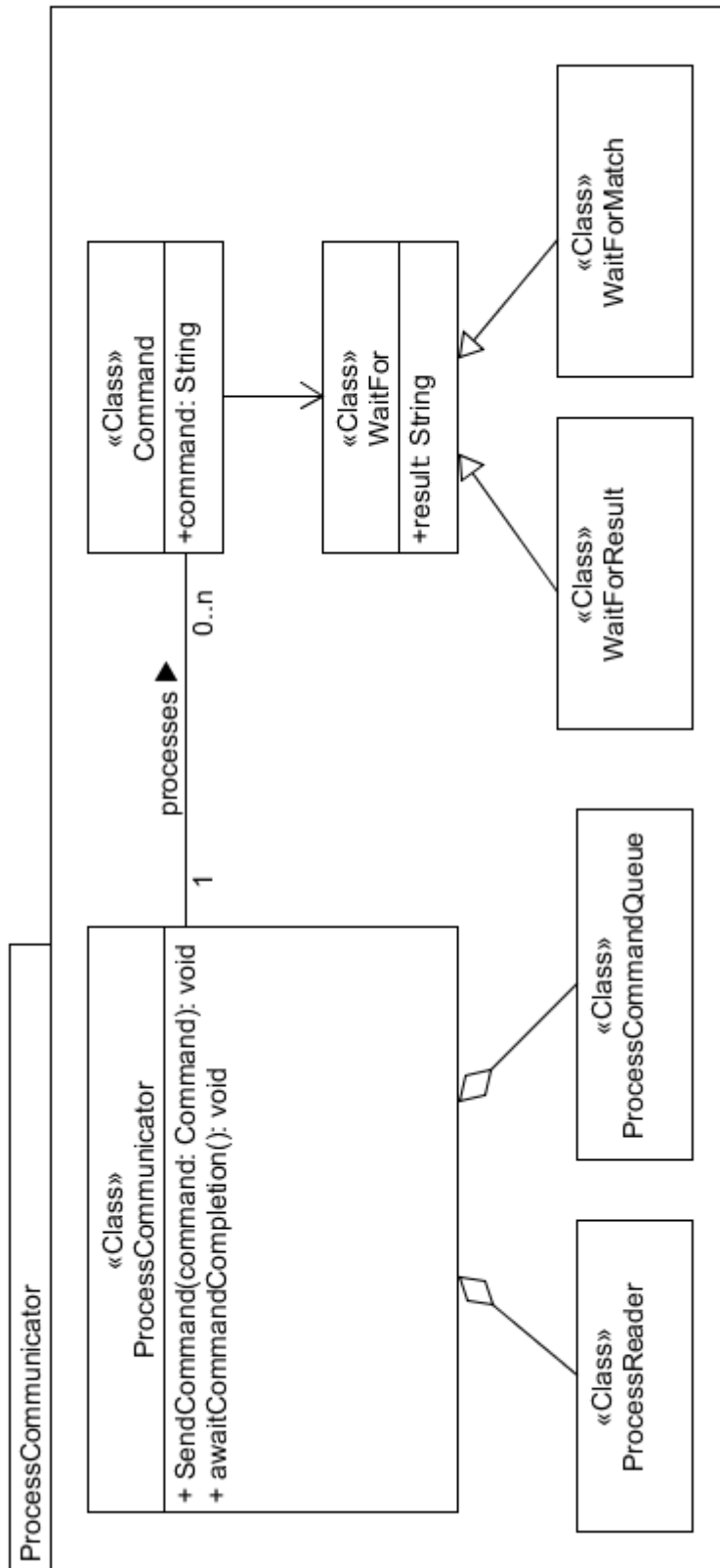


Abbildung 5.2.: Klassendiagramm der.

5.4.1. Klassenbeschreibung

Über das `ProcessCommunicator` API können Commands an den Prozess gesendet werden. Das API stellt blockierende und nicht blockierende Methoden für das senden von Commands an. So können Commands über die `sendCommand` Funktion einen Command abschicken, ohne zu blockieren. Falls auf ein bestimmtes Resultat gewartet werden muss, kann die Funktion `sendCommandAwaitResult` verwendet werden, welche den aufrufenden Thread blockiert, bis das Resultat eingetroffen ist.

ProcessCommunicator

Der `ProcessCommunicator` ist die zentrale Kommunikationsstelle. Über den `ProcessCommunicator` können die Commands gesendet werden und gleichzeitig auch auf die angeforderten Resultate warten.

ProcessReader

Der `ProcessReader` ist eine verschachtelte Klasse des `ProcessCommunicator`, welche als Thread im Hintergrund den Stream des Prozesses liest und verarbeitet. Der `ProcessReader` notifiziert alle Threads, welche auf ein Resultat warten, falls dies eingetroffen ist.

ProcessCommandQueue

Die `ProcessCommandQueue` ist eine verschachtelte Klasse des `ProcessCommunicator`, welche als Thread im Hintergrund Commands verarbeitet und an den Prozess sendet. Die `ProcessCommandQueue` blockiert, falls dies von einem Command spezifiziert wurde. Wenn die `ProcessCommandQueue` blockiert werden keine weiteren Commands mehr verarbeitet, bis das der `WaitFor` Klasse spezifizierten Resultat von dem Prozess geschrieben wurde.

Command

Die `Command` Klasse repräsentiert ein Command, welcher über den `ProcessCommunicator` an den Prozess gesendet werden kann. In einem Command kann eine `WaitFor` spezifiziert werden, falls der Command blockieren soll, bis ein Resultat von dem Prozess geliefert wird.

WaitFor

Mit der abstrakten `WaitFor` Klasse können auf Resultate des Prozesses gewartet werden. Dafür wurden verschiedene Implementationen bereitgestellt. Mit der `WaitForResult` können auf String Resultate gewartet werden. Mit der `WaitForMatch` Klasse kann per Regex auf ein Resultat gewartet werden. In folgendem Pseudocode wird die Funktionalität der `WaitForMatch` Klasse im Zusammenhang mit dem senden eines Commands gezeigt.

```
sendCommandAwaitMatch("foo", "[1-9]")
```

Der aktuelle Thread wird blockiert, bis der Prozess eine Ziffer zwischen 0 und 9 schreibt. Für das warten auf ein Resultat wurde ein Timeout von 10 Sekunden implementiert. Falls der Prozess in dieser Zeit keine Antwort gibt, wird eine `CommandTimeoutException` geworfen und der `ProcessCommunicator` wird heruntergefahren, das keine weiteren Commands mehr gesendet werden können.

5.5. Testing

Der `ProcessCommunicator` wurde vor allem mittels Unit-Tests getestet. Unter diesen Tests befindet sich ein Stresstest, welcher möglichst viele Commands an den Prozess sendet und ihn somit unter grosser Last testet.

6. uForth Editor

In diesem Kapitel wird beschrieben, wie das Framework Xtext funktioniert und wie es verwendet wurde, um Feature wie Syntax Highlighting, eine Outline und Dokumentations Popups für den Forth Editor zu implementieren.

6.1. Xtext Implementation des uForth Editor

Xtext ermöglicht es, das Grundgerüst einer Entwicklungsumgebung zu generieren. Dafür verwendet Xtext eine Extended Backus-Naur Form (EBNF) ähnliche Grammatik. Aus dieser Grammatik wird dann ein ANTLR-Parser und Klassen, welche für den Editor benötigt werden, generiert. Um Xtext zu konfigurieren wird die dependency injection (DI) Library, Guice von Google verwendet. Mit Guice können viele Teile von Xtext in einem zentralen Modul mittels DI ersetzt werden.

Da Forth kein monolithischen Compiler und somit keine statische Grammatik besitzt, kann keine Grammatik für Xtext geschrieben werden, welche die gesamte Forth Sprache beschreibt. Es wurde deshalb eine Xtext Grammtik verwendet, welche ungefähr der folgenden EBNF Grammtik entspricht.

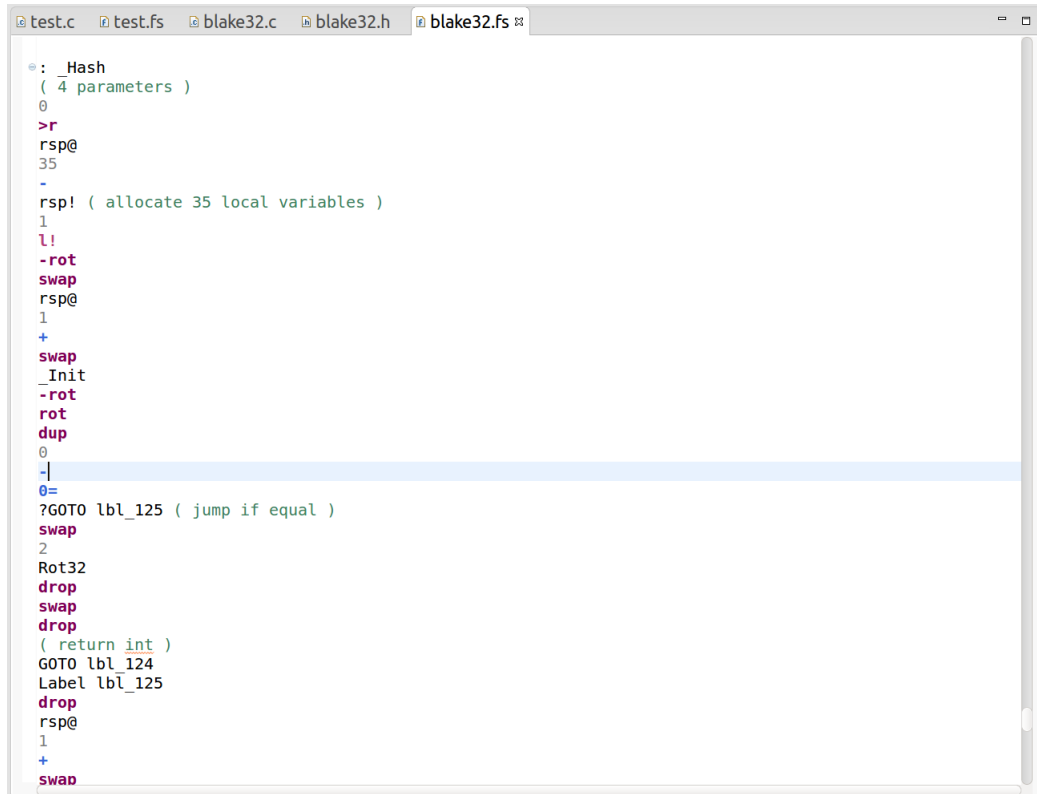
```
instructoin = create | function | word;  
function = ':', identifier, { Word }, ',';  
create = 'create', identifier, { literal ',', ' ' };  
word = identifier;
```

Wobei die `identifier` Regel alle gültigen Forth Identifier beschreibt und die `literal` Regel alle gültigen Integer und Double Werte beschreibt.

Der daraus generierte Parser kann alle vom LCC generierten Forth Files parsen und reicht somit für die Entwicklungsumgebung aus.

6.2. uForth Editor

Der Editor unterstützt Syntax Highlighting für vordefinierte Forth Wörter, Literals und Kommentare. Die Wörter werden in drei Gruppen unterteilt. Stack-, Memory- und Arithmetische-Wörter. Alle Gruppen haben eine andere Farbe, welche in der uForth Preference Page geändert werden kann. In folgender Abbildung 6.1 ist der uForth Editor zu sehen.



```
@ test.c  test.fs  blake32.c  blake32.h  blake32.fs  x
: _Hash
  ( 4 parameters )
  0
  >r
  rsp@
  35
  -
  rsp! ( allocate 35 local variables )
  1
  l!
  -rot
  swap
  rsp@
  1
  +
  swap
  _init
  -rot
  rot
  dup
  0
  -|
  0=
  ?GOTO lbl_125 ( jump if equal )
  swap
  2
  Rot32
  drop
  swap
  drop
  ( return int )
  GOTO lbl_124
  Label lbl_125
  drop
  rsp@
  1
  +
  swap
```

Abbildung 6.1.: uForth Editor, welcher vorallem für den Debugger verwendet wird.

6.2.1. Wörter Dokumentation

Eine Dokumentation wurde für uForth spezifischen Wörter als Popup implementiert. Die Dokumentation wird dafür aus einer Textdatei gelesen. In folgender Abbildung 6.2 ist eine Popup Dokumentation zu dem Wort `label` zu sehen.

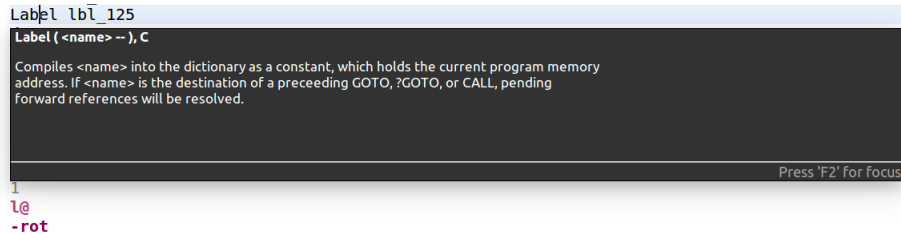


Abbildung 6.2.: Popup Dokumentation zu dem Wort `label`. Die Dokumentation wird angezeigt, falls der Cursor sich über einem Wort befindet, für welches eine Dokumentation verfügbar ist.

6.2.2. Editor Outline

Für den Editor steht zudem eine Outline zur Verfügung. Die Outline gliedert das Forth File nach den von der Grammatik spezifizierten Regeln. Für den uForth Editor bedeutet das, dass das File nach Funktionen und Wörtern gegliedert wird.



Abbildung 6.3.: Outline zu einem Forth File. Die Outline gliedert das File in Funktionen und Wörter.

7. Debugger

In diesem Kapitel wird beschrieben, wie der Debugger in Eclipse integriert wurde. Es wird aufgezeigt, was für Möglichkeiten existieren einen Debugger in Eclipse zu integrieren und welche implementiert wurden.

7.1. Breakpoints

Als erstes müssen für den Debugger Breakpoints gesetzt werden können. Dafür stehen zwei Möglichkeiten zur Verfügung, welche in den nächsten Kapiteln erläutert und verglichen werden.

7.1.1. Per Konsole

Die erste Möglichkeit ist, die Breakpoints per Konsole zu setzen. Das Senden der Commands funktioniert mit den im Kapitel ?? beschriebenen Klassen. In der Konsole kann der Command

```
debug _function
```

abgesetzt werden, um einen Breakpoint zu setzen.

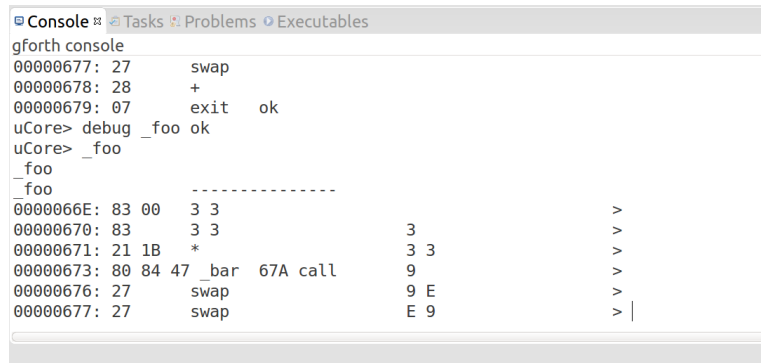
7.1.2. Im Source Code

Eine weitere Möglichkeit ist, Breakpoints im C-File zu setzen. Dies wurde so umgesetzt, dass der Breakpoint nur auf eine Funktionsdefinition gesetzt werden kann. Alle anderen Zeilen des C-Source Codes können nicht direkt auf den übersetzten Forth Code abgebildet werden und sind deshalb nicht erlaubt für die Breakpoints.

Eclipse CDT stellt den Abstract Syntax Tree (AST) des C-Files zur Verfügung. Mit Hilfe des AST kann überprüft werden, ob sich der Breakpoint wirklich auf einer Funktionsdefinition befindet.

7.2. Konsolen basierter Debugger

Eine einfache Integration des Debuggers ist, den schon existierenden Forth Konsolen Debugger im Eclipse zu verwenden. Dieser kann mit dem im Kapitel ?? beschriebenen Prozess Kommunikationsmitteln angesteuert und in einer Eclipse Console View angezeigt werden.



```
gforth console
00000677: 27      swap
00000678: 28      +
00000679: 07      exit    ok
uCore> debug _foo ok
uCore> _foo
foo
foo
-----
0000066E: 83 00    3 3
00000670: 83      3 3          3
00000671: 21 1B    *          3 3
00000673: 80 84 47 _bar 67A call 9
00000676: 27      swap        9 E
00000677: 27      swap        E 9
```

Abbildung 7.1.: Konsolen basierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden.

7.3. Forth Debugger

Eine weitere Möglichkeit ist, das Debug User Interface von Eclipse zu verwenden um den Debugger zu steuern. Dies ist für den Endanwender angenehmer, da alle Informationen des Debuggers in einem User Interface ersichtlich sind.

7.3.1. CDT oder JDT Debugging Mechanismen

Das Eclipse JDT stellt mehrere Möglichkeiten zur Verfügung, wie ein Debugger integriert werden kann. Es können die vom Eclipse JDT verwendeten Mechanismen (vorallem das Plugin `org.eclipse.debug.core`), oder die vom CDT erweiterten Mechanismen (vorallem das Plugin `org.eclipse.cdt.debug.core`), welche verwendet werden um einen C oder C++ Debugger zu integrieren. Das vom CDT zur Verfügung gestellte Plugin wird vorallem dazu verwendet, um ein neuer C oder C++ Debugger zu integrieren, da es sich aber um einen Forth Debugger handelt, werden diese Erweiterungen nicht gebraucht. Ich habe mich deshalb dazu entschieden, das JDT Debugging zu verwenden.

7.3.2. Debugger Aktionen

Für den Debugger wurden einige Aktionen, welche schon im Eclipse verwendet werden, implementiert und einige neue Forth spezifische Aktionen hinzugefügt.

Resume

Mit der Resume Aktion kann der während dem Debuggen fortgeführt werden. Dafür wird ein `end-trace` Command an den Forth Prozess gesendet. Der `end-trace` Command entfernt auch alle Breakpoints, diese werden automatisch nach der Ausführung neu gesetzt.

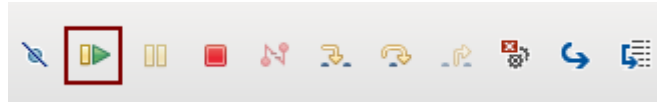


Abbildung 7.2.: Resume Aktion.

Terminate

Mit der Terminate Aktion kann der Prozess heruntergefahren werden, in dem der `bye` Command gesendet wird. Falls der Prozess nicht mehr reagiert, sollte die Kill Aktion verwendet werden. Die Terminate Aktion unterscheidet sich von der Kill Aktion, in dem sie versucht den Prozess auf normale Weise zu beenden und nicht über ein Kill Signal.

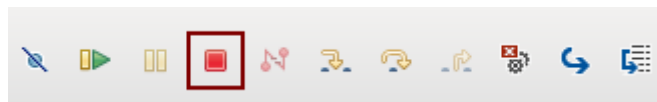


Abbildung 7.3.: Terminate Aktion.

Step Into

Mit der Step Into Aktion kann in eine Funktion gesprungen werden, falls die nächste Zeile ein Funktionsaufruf ist. Dafür wird ein `nest` Command an den Forth Prozess gesendet.

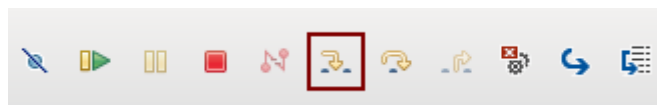


Abbildung 7.4.: Step Into Aktion.

Step

Mit der Step Aktion kann ein normaler single step ausgeführt werden. Dafür wird ein `CR` Command an den Forth Prozess gesendet.



Abbildung 7.5.: Step Aktion.

Kill

Mit der Kill Aktion kann der Prozess terminiert werden, in dem das Kill Signal gesendet wird. Im Normalfall sollte die Terminate Aktion verwendet werden, da sie den Forth Prozess über den `bye` Command beendet.

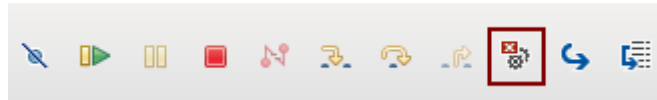


Abbildung 7.6.: Kill Aktion.

After

Die After Aktion setzt den Breakpoint nach der nächsten Instruktion, falls es sich um einen Rückwärts Sprung handelt, welcher möglicherweise von einer `UNTIL`, `REPEAT`, `LOOP` oder `NEXT` Instruktion kompiliert wurde. Der Rest der Schleife wird deshalb ohne Unterbruch ausgeführt.

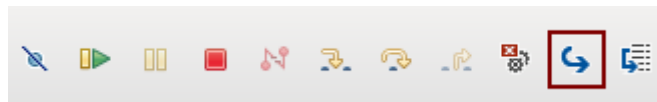


Abbildung 7.7.: After Aktion.

Jump

Mit der Jump Aktion kann über die nächste auszuführende Instruktion gesprungen werden. Dafür wird ein `jump` Command an den Forth Prozess gesendet.

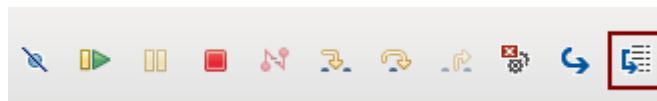


Abbildung 7.8.: Jump Aktion.

7.3.3. Stack View

In der Stack View wird der aktuelle Data-Stack angezeigt. Die Stack View wird automatisch nach jedem stepping des Debuggers aktualisiert.

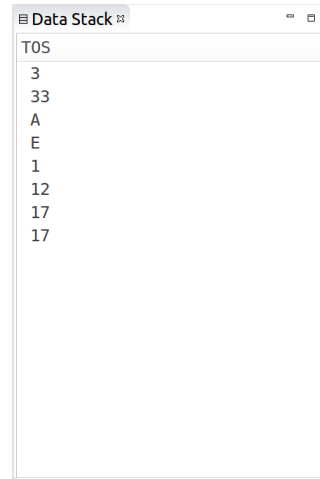


Abbildung 7.9.: Stack View mit aktuellem Dstack Inhalt. Der Top Of Stack (TOS) ist zuoberst in der Liste.

7.3.4. Memory View

In der Memory View kann ein Memory Dump, welcher mit dem `dump` Befehl von uForth abgefragt werden kann, angezeigt werden. Der Memory Dump wird automatisch nach jedem stepping des Debuggers aktualisiert.

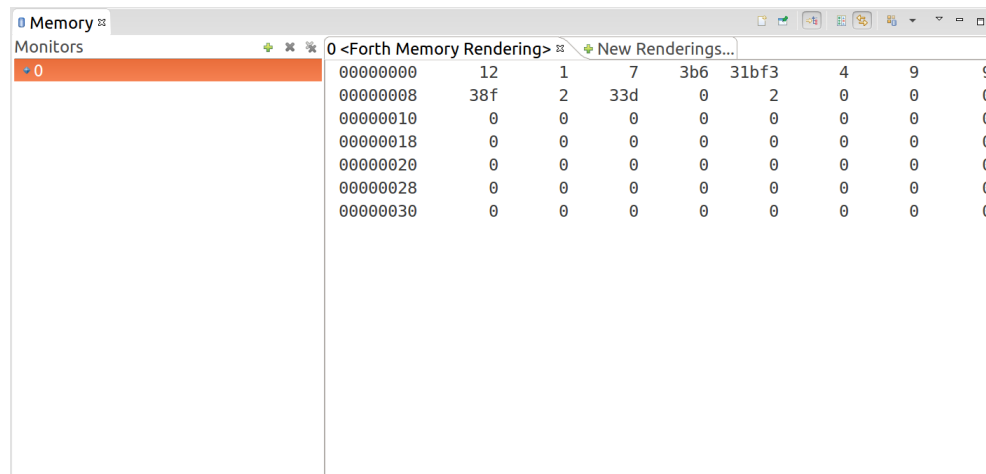


Abbildung 7.10.: Eine Memory Dump, welcher

7.4. C-Debugger

Eine mögliche Erweiterung, wäre den Debugger so zu integrieren, das er direkt auf dem C-Source Code arbeitet (nicht wie bis jetzt, auf dem generierten Forth Code). Dies konnte nicht umgesetzt werden, da Debug Informationen des Compilers fehlen. Der C-Source Code kann nicht auf den entsprechenden generierten Forth Source Code abgebildet werden.

8. Entwicklungsumgebungs Einstellungen

Für die Entwicklungsumgebung wurde eine Eclipse Preference Page erstellt. In diesem Kapitel wird erklärt, was in dieser Preference Page für Einstellungen vorgenommen werden können und was diese für Auswirkungen haben. Die Einstellungen sind im Eclipse Preference Dialog unter MCore zu finden.

8.1. Umbilical

Der Umbilical Port kann über einen Dialog, welcher alle möglichen Ports anzeigt, ausgewählt werden.

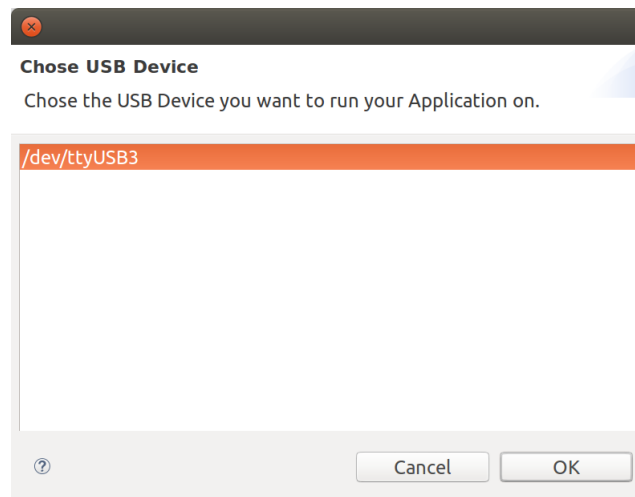


Abbildung 8.1.: Dialog über welchen der Umbilical Port gesetzt werden kann.

Der Port wird, wenn das Programm ausgeführt wird, automatisch gesetzt.

8.2. Loader

Der Loader ist das File, welches von gforth mittels dem Befehl:

```
gforth ./loader.fs
```

gestartet wird. Im Loader befindet sich ein Platzhalter `$INPUT_FILE`, welcher bei der Ausführung des Programms mit dem Namen des Files ersetzt wird.

9. Optimierungen

In diesem Kapitel wird beschrieben, was für Optimierungen für den Compiler implementiert wurden. Im Forth Cross-Compiler sind schon einige Peephole Optimierungen implementiert. In diesem Kapitel werden die neu implementierten Optimierungen mit denen des Cross-Compilers verglichen und gezeigt wo noch bessere Optimierungsstrategien verwendet werden können.

9.1. Peephole Optimierung

Peephole Optimierungen ist eine Art von Optimierung, welche auf einer kleinen Sequenz von Instruktionen durchgeführt wird. Dieses Sequenz wird Peephole oder auch Window genannt. Die Peephole Optimierung versucht Sets von Instruktionen durch kürzere oder schnellere Instruktionen zu ersetzen. [8] Peephole Optimierungen können die grösse des Codes um 15–40 Prozent verkleinern und sind heute in allen gängigen Compilern implementiert. [9] Zu den Peephole Optimierungen gehören unter anderen folgende Arten von Optimierungen:

- Constant Folding - Konstante Expressions auswerten
- Constant Propagation - Konstante Werte in Expressions substituieren
- Strength Reduction - Langsame Instruktionen mit äquivalenten schnellen Instruktionen ersetzen.
- Combine Operations - Mehrere Operationen mit einer äquivalenten ersetzen
- Null Sequences - Unötige Operationen entfernen [8]

9.1.1. Beispiele

Folgend einige Peephole Optimierungs Beispiele anhand von Forth Code.

Constant Propagation

Folgende Instruktionen

```
1
2
swap
+
dup
```

können durch:

```
2
2
```

ersetzt werden. Die Instruktionen swap, + und dup können schon zur Kompilierzeit durchgeführt werden.

Combine Operations

Folgende zwei Instruktionen

```
rot
rot
```

können durch

```
-rot
```

ersetzt werden. Die zwei rot Instruktionen sind äquivalent zu einer -rot Instruktion. Oder die folgenden zwei Instruktionen

```
dup
drop
```

können durch

```
nop
```

ersetzt werden. Die zwei Instruktionen heben sich auf und können somit entfernt werden.

9.1.2. Optimierungen

Für den Compiler wurden Prototypen mässig zwei Optimierungen in Java implementiert. Die erste Optimierung versucht benachbarte Instruktionen zu vereinfachen. Die zweite Optimierung ist eine einfache Constant Propagation. Die beiden Optimierungen werden in den nächsten Kapiteln genauer beschrieben. Der neue Optimizer wird in zwei Phasen durchgeführt:

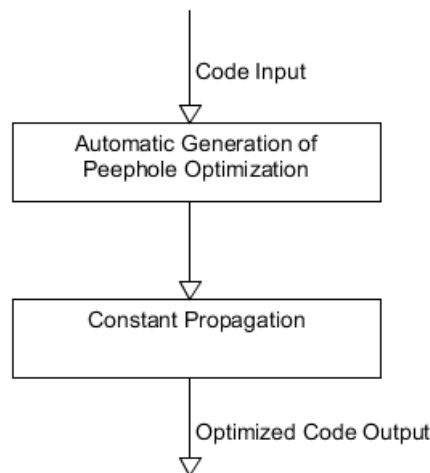


Abbildung 9.1.: Die zwei Phasen des Optimizers.

Im Cross-Compiler wurden unter anderem folgende Peephole Optimierungen schon implementiert.

1. `<lit> + ld`, `<lit> + st`, `<lit> + @`, and `<lit> + \!` werden mit automatisch inkrementierenden Speicherzugriff Instruktionen ersetzt, wenn `<lit>` sich zwischen 4 und 3 befindet
2. Folgende Stack Operationen:
 $swap, swap \rightarrow nop$
 $-rot, rot \rightarrow nop$
 $swap, + \rightarrow +$

Für eine komplette Liste siehe "real time, object oriented with debugger" [10].

9.1.3. Automatische Generierung von Peephole Optimierungen

Klassische Peephole Optimizer versuchen häufig einige Maschinenspezifische Patterns zu korrigieren. Der von Davidson und Fraser [9] beschriebene Algorithmus (PO) verwendet eine Machine Description, simuliert benachbarte Instruktionen und versucht diese mit äquivalenten, schnelleren Instruktionen zu ersetzen. Für den Forth Optimizer wurde ein Teil des PO objektorientiert implementiert.

9.1.4. Constant Propagation

Unter Constant Propagation versteht man, dass vorwärts substituieren von Konstanten im Code. Dies kann zur Folge haben, dass mehrere Instruktionen schon zur Kompilierzeit ausgewertet werden können, wie bei den Beispielen 9.1.1 zu sehen ist.

9.1.5. Resultate und Tests

Die Resultate des Optimierers wurden mit verschiedenen Forth Funktionen getestet und die Resultate mit dem Peephole Optimizer des uForth Cross-Compilers verglichen. Der Source-Code zu den getesteten Funktionen befindet sich im Anhang.

Funktion	Orig	Ref	Neu	Kommentar
_Init	126	112	119	Die Referenzimplementierung produziert vor allem wegen der Speicherzugriffsoptimierung kürzeren Code. Constant Propagation, welche neu implementiert wurde, konnte keine durchgeführt werden. PO konnte Instruktionspaare finden, welche weg optimiert werden können.
_Update	8	8	4	PO konnte zwei Instruktionspaare weg optimieren, welche von der Referenzimplementierung nicht optimiert werden.
_Hash	65	60	60	Die Referenzimplementierung produziert wieder wegen der Speicherzugriffsoptimierung weniger Code. PO konnte wieder Instruktionspaare finden, welche weg optimiert werden können.
_Propagation	11	11	4	Bei dieser Funktion konnte vor allem Constant Propagation durchgeführt werden. Die Referenzimplementierung führt keine Constant Propagation durch.

Tabelle 9.1.: Resultate des neuen Optimizers verglichen mit dem Cross-Compiler Optimizer.

Es hat sich herausgestellt, dass bei allen Beispielen die Constant Propagation nur wenig Code optimieren konnte. Dies ist vermutlich der Fall, weil konstante Ausdrücke schon vom Compiler optimiert werden und die implementierte Constant Propagation zu primitiv ist. Im nächsten Kapitel werden einige Erweiterungen vorgeschlagen, um diese effizienter zu gestalten.

PO konnte Regeln finden, welche vom Forth-Cross Compiler noch nicht erkannt werden. Unter anderem folgende:

```
swap, swap → nop  
-rot, rot → nop  
rot, -rot → nop  
1, + → 1+  
swap, + → +
```

Diese Regeln könnten im Forth Cross-Compiler integriert werden. Im nächsten Kapitel werden Änderungen für den PO vorgeschlagen, damit dieser mehr mögliche Instruktionketten erkennen könnte, welche vereinfacht werden können.

9.1.6. Mögliche Erweiterungen

Im Moment werden vom PO nur Instruktionen, welche einen Einfluss auf den Daten Stack haben simuliert. PO könnte erweitert werden, indem weitere Instruktionen auch simuliert werden. Eine weitere Möglichkeit wäre, der von Bansal und Aiken beschriebene Superoptimizer zu implementieren. Dieser Superoptimizer verwendet Bruteforce Optimierungen mittels tausenden von Regeln. Diese Regeln werden von Trainings Programmen inferiert und in einer Datenbank gespeichert. [11]

Die Constant Propagation könnte noch so erweitert werden, dass sie auch mit Branches umgehen kann. Somit könnten unnötige If-Statements, sowie Schleifen entfernt werden. Um dies zu implementieren müsste der Code zuerst in eine static single assignment form (SSA) transformiert werden. [12] Es wäre somit keine Peephole optimierung mehr.

A. Literaturverzeichnis

- [1] John Arthorne. Faq what is a plug-in? https://wiki.eclipse.org/FAQ_What_is_a_plug-in%3F, 2011.
- [2] Vogella. Eclipse extension points and extensions - tutorial. <http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html>, 2013.
- [3] Xtext. <https://eclipse.org/Xtext/>, 2013.
- [4] Antlr. <http://wwwantlr.org/>.
- [5] Bruno Medeiros. D development tools. <https://github.com/bruno-medeiros/DDT>.
- [6] The gdb/mi interface. https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.
- [7] Understanding the gnu debugger machine interface (gdb/mi). <http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug2/>.
- [8] Peephole optimization. https://en.wikipedia.org/wiki/Peephole_optimization.
- [9] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. In *ACM Trans. Program. Lang. Syst.*, pages 191–202, 1980.
- [10] Klaus Schleisiek. uforth real time, object oriented with debugger, 2013.
- [11] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ACM SIGPLAN Not.*, pages 394–403, 2006.
- [12] Static single assignment form. https://en.wikipedia.org/wiki/Static_single_assignment_form.
- [13] Xtext download. <https://eclipse.org/Xtext/download.html>.

B. Abbildungsverzeichnis

2.1. Ein Plugin, welches einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.	8
3.1. Die Eclipse Problems View zeigt, dass das Program lcc-mcore nicht gefunden wurde im Pfad.	11
4.1. Der Launch Configuration Dialog. Im Dialog kann eine neue MCore Launch Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen muss das Projekt und das Forth File, welches gestartet werden soll angegeben werden.	12
4.2. Dialog, welcher angezeigt wird, wenn ein Program gestartet wird und der Umbilical Port ungültig ist. Der Port kann in der MCore Preference Page geändert werden.	13
5.1. Ein erstes Design des Prozess API. Die zentrale Kommunikation geschieht über die Communicator Klasse. Es können Commands abgesetzt werden und es kann auf Resultate gewartet werden.	15
5.2. Klassendiagramm der.	16
6.1. uForth Editor, welcher vorallem für den Debugger verwendet wird.	20
6.2. Popup Dokumentation zu dem Wort label. Die Dokumentation wird angezeigt, falls der Cursor sich über einem Wort befindet, für welches eine Dokumentation verfügbar ist.	21
6.3. Outline zu einem Forth File. Die Outline gliedert das File in Funktionen und Wörter.	21
7.1. Konsolen basierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden.	23
7.2. Resume Aktion.	24
7.3. Terminate Aktion.	24
7.4. Step Into Aktion.	24
7.5. Step Aktion.	25
7.6. Kill Aktion.	25
7.7. After Aktion.	25
7.8. Jump Aktion.	25
7.9. Stack View mit aktuellem Dstack Inhalt. Der Top Of Stack (TOS) ist zuoberst in der Liste.	26
7.10. Eine Memory Dump, welcher	26

8.1. Dialog über welchen der Umbilical Port gesetzt werden kann.	28
9.1. Die zwei Phasen des Optimizers.	31
E.1. Mit der Target Definition kann die Target Platform gesetzt werden.	39
E.2. Der MWE2 Workflow generiert alle notwendigen Klassen für den Forth Editor.	40
E.3. Die Entwicklungsumgebung kann im plugin.xml des Plugins ch.fhnw.mdt.ui gestartet werden.	40

C. Tabellenverzeichnis

9.1. Resultate des neuen Optimizers verglichen mit dem Cross-Compiler Optimizer.	32
--	----

D. Installationsanleitung für Endanwender

Zuerst muss lcc-mcore und gforth nach der Anleitung in der jeweiligen Dokumentation installiert werden. lcc-mcore muss im `PATH` und `GFORTHPATH` muss gesetzt sein. Falls eine dieser Einstellungen nicht korrekt ist, wird die Entwicklungsumgebung das Projekt nicht komplette builden. Die Entwicklungsumgebung braucht Superuser Rechte, das heisst sie muss mit `sudo` gestartet werden. Damit der richtige `PATH` verwendet wird muss dieser dem `sudo` Befehl übergeben werden. Die Entwicklungsumgebung kann mit dem Befehl `sudo env PATH=$PATH ./eclipse` korrekt gestartet werden.

E. Installationsanleitung für Entwickler

Zuerst müssen alle Schritte, welche im Kapitel Installationsanleitung für Endanwender beschrieben werden, durchgeführt werden. Für die Entwicklung wird eine Eclipse RCP Distribution mit Xtext Unterstützung benötigt. Diese kann auf der offiziellen Xtext Webseite [13] heruntergeladen werden. Zuerst müssen alle Projekte im Eclipse importiert werden. Dann muss die Target Platform gesetzt werden. Dies kann im File `cdt-8.5.target` gemacht werden, wie in folgender Abbildung E.1 zu sehen ist.

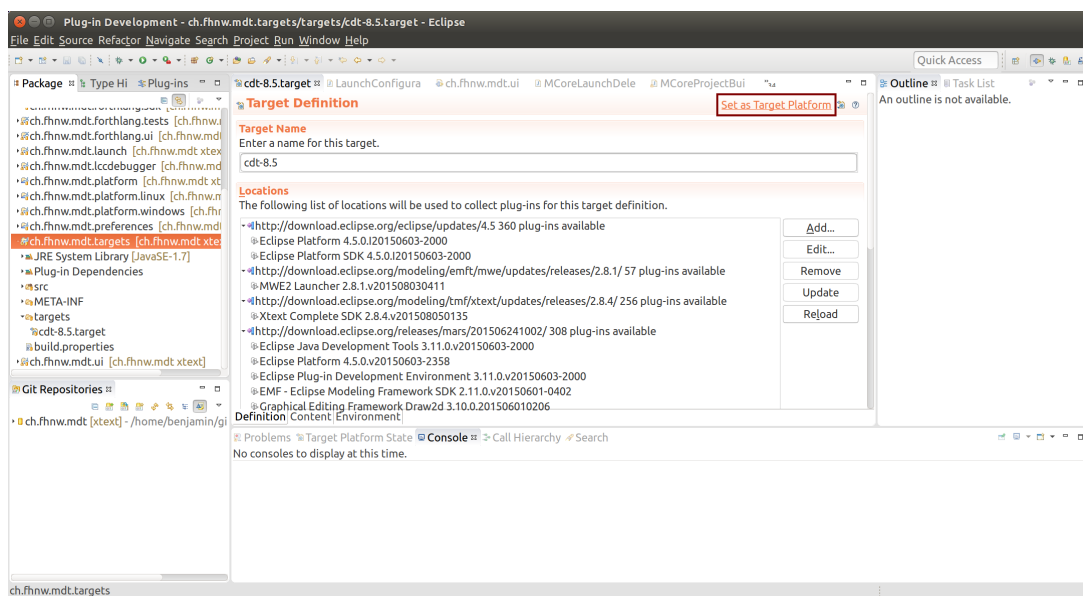


Abbildung E.1.: Mit der Target Definition kann die Target Platform gesetzt werden.

Danach muss der Xtext Workflow gestartet werden. Dieser generiert alle notwendigen Klassen, die für den uForth Editor gebraucht werden. Der Workflow kann im File `GenerateUForth.mwe2` im `ch.fhnw.mdt.forthlang` Plugin gestartet werden, wie in folgender Abbildung E.2 zu sehen ist.

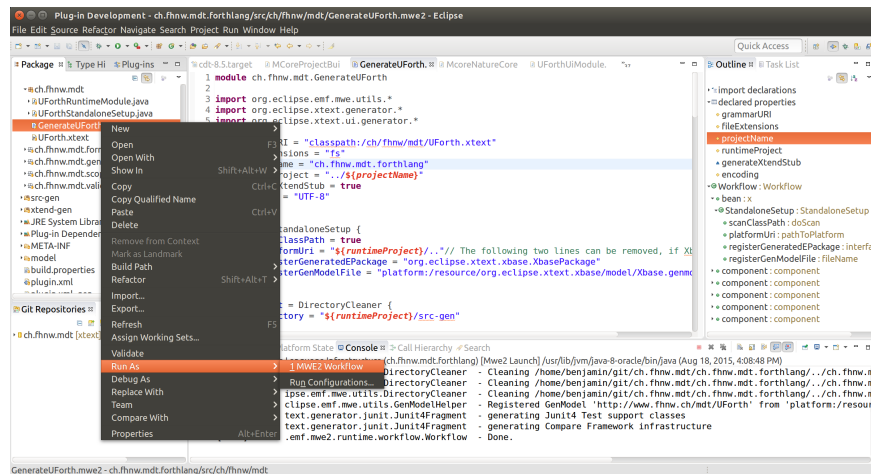


Abbildung E.2.: Der MWE2 Workflow generiert alle notwendigen Klassen für den Forth Editor.

Die Entwicklungsumgebung kann danach im plugin.xml des Plugins `ch.fhnw.mdt.ui`, wie in folgender Abbildung [?] zu sehen ist, gestartet werden.

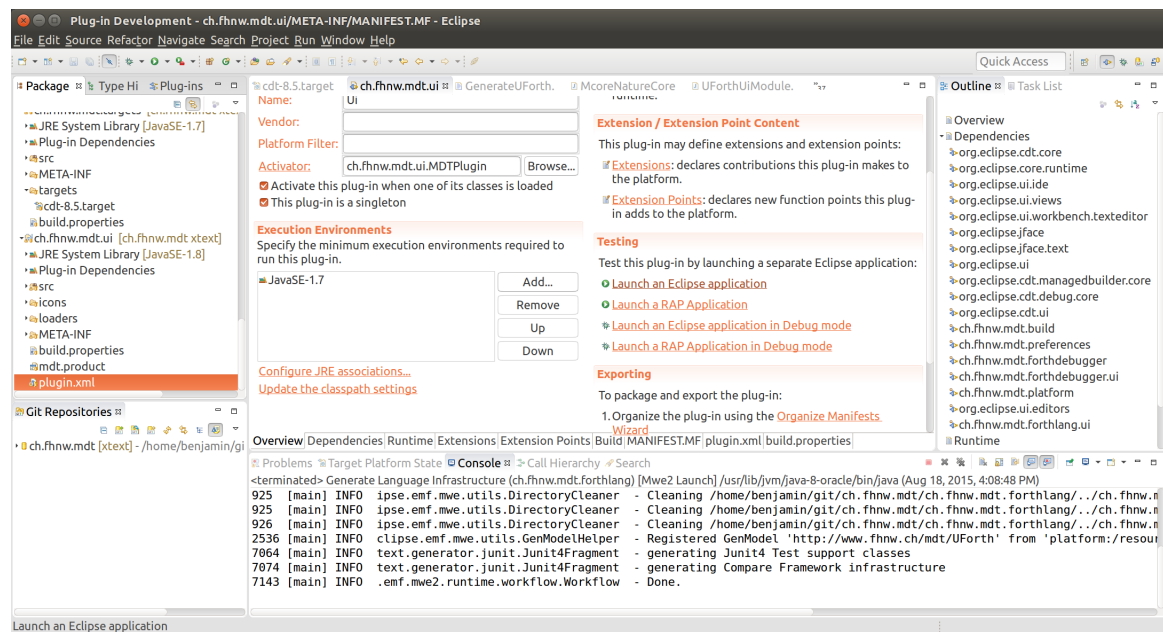


Abbildung E.3.: Die Entwicklungsumgebung kann im plugin.xml des Plugins `ch.fhnw.mdt.ui` gestartet werden.

F. Forth Test-Funktionen

F.1. _Init

```
: _Init
swap
over
256
-
?GOTO lbl_72 ( jump if not equal )
0
GOTO lbl_77
Label lbl_74
swap
over
2
Rot32
dup
_IV256
2
Rot32
+
-2
Rot32
@
swap
2over
nip
4
+
2
Rot32
+
-2
Rot32
!
Label lbl_75
swap
1+
```

```

swap
swap
Label lbl_77
dup
8
u<
?GOTO lbl_74 ( jump if less )
drop
0
over
12
+
!
0
over
13
+
!
0
GOTO lbl_81
Label lbl_78
swap
0
2over
nip
2over
nip
14
+
+
!
Label lbl_79
swap
1+
swap
swap
Label lbl_81
dup
16
u<
?GOTO lbl_78 ( jump if less )
drop
0
over

```

```

30
+
!
0
over
31
+
!
0
over
32
+
!
0
over
33
+
!
GOTO lbl_73
Label lbl_72
2
Rot32
drop
drop
2
( return int )
GOTO lbl_71
Label lbl_73
swap
over
!
0
over
1+
!
1
over
2
+
!
0
swap
3
+

```

```
!  
0  
;
```

F.2. _Hash

```
: _Hash  
0  
>r  
rsp@  
35  
-  
rsp! ( allocate 35 local variables )  
1  
l!  
-rot  
swap  
rsp@  
1  
+  
swap  
_Init  
-rot  
rot  
dup  
0  
-  
0=  
?GOTO lbl_125 ( jump if equal )  
swap  
2  
Rot32  
drop  
swap  
drop  
GOTO lbl_124  
Label lbl_125  
drop  
rsp@  
1  
+  
swap  
rot
```

```

_Update
dup
1
l@
-rot
0
-
0=
?GOTO lbl_127 ( jump if equal )
swap
2
Rot32
drop
GOTO lbl_124
Label lbl_127
drop
rsp@
1
+
swap
_Final
Label lbl_124
rsp@
35
+
rsp!
rdrop ( deallocate 35 local variables )
;

```

F.3. _Update

```

: _Update
-rot
swap
swap
rot
_Update32
Label lbl_109
;

```

F.4. `_Propagation`

```
: _propagation
5
2
+
1+
3
*
7
*
Label lbl_1
;
```

G. Ehrlichkeitserklärung

Hiermit bestätigen die Autoren, diese Arbeit ohne fremde Hilfe und unter Einhaltung der gebotenen Regeln erstellt zu haben.

Benjamin Neukom

Ort, Datum

Unterschrift