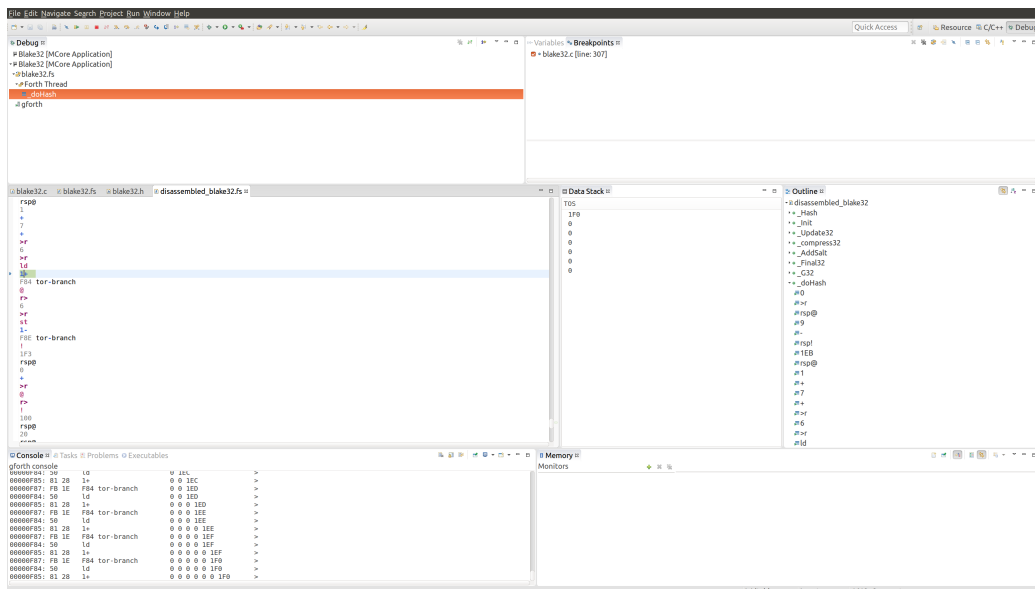


Bachelor-Thesis

Eclipse Entwicklungsumgebung für MicroCore

Benjamin Neukom

August 2015



Betreuer: Carlo Nicola

Inhaltsverzeichnis

1. Einleitung	5
2. Die Eclipse Plattform	6
2.1. Die Plattform	6
2.2. Plugins	6
2.3. Extension Points	6
2.4. Eclipse basierte MCore Entwicklungsumgebung	7
2.4.1. JDT	7
2.4.2. Xtext	7
2.4.3. Dynamic Language Toolkit Framework	8
2.4.4. Eclipse C/C++ Development Tools	8
2.4.5. Verwendung für uCore Eclipse	8
3. Integration des Compilers	9
3.1. Integration in Eclipse CDT	9
3.1.1. Configuration	9
3.1.2. Tool	9
3.1.3. Toolchain	9
3.1.4. CDT-Builder	10
3.1.5. Project Builder	10
4. Programm Launch	11
4.1. Launch	11
4.1.1. Run	12
4.1.2. Debug	12
5. Forth Kommunikation	13
5.1. Prozess Kommunikation	13
5.1.1. GDB/MI-Commands	13
5.1.2. Direkte Kommunikation mit dem Prozess	13
5.2. Kommunikation	14
5.3. API-Design	14
5.4. Implementierung	14
5.4.1. Klassenbeschreibung	16
5.5. Testing	17
6. uForth-Editor	18
6.1. Xtext-Implementation des uForth-Editors	18

6.2. uForth Editor	19
7. Debugger	21
7.1. Breakpoints	21
7.1.1. Per Konsole	21
7.1.2. Im Source Code	21
7.2. Konsolenbasierter Debugger	22
7.3. Forth Debugger	22
7.3.1. CDT oder JDT Debugging-Mechanismen	22
7.3.2. Debugger-Aktionen	22
7.3.3. Stack View	25
7.3.4. Memory View	25
7.4. C-Debugger	26
8. Entwicklungsumgebungs Preference Page	27
8.1. Umbilical Port	27
8.2. Loader	27
9. Optimierungen	28
9.1. Peephole-Optimierung	28
9.1.1. Beispiele	29
9.1.2. Optimierungen	30
9.1.3. Automatische Generierung von Peephole Optimierungen	31
9.1.4. Constant Propagation	31
9.1.5. Resultate und Tests	31
9.1.6. Mögliche Erweiterungen	32
A. Literaturverzeichnis	33
B. Abbildungsverzeichnis	34
C. Tabellenverzeichnis	36
D. Installationsanleitung für Anwender	37
E. Installationsanleitung für Entwickler	38
F. Forth Test-Funktionen	40
F.1. _Init	40
F.2. _Hash	43
F.3. _Update	44
F.4. _Propagation	45
G. Ehrlichkeitserklärung	46

Im ersten Teil dieser Arbeit wird beschrieben, wie eine auf Eclipse basierende Entwicklungsumgebung für MicroCore implementiert wurde. Im zweiten Teil werden verschiedene Peephole-Optimierungen, unter anderem der von 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 das Entwickeln einer auf Eclipse basierten Entwicklungsumgebung, die 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, welche Möglichkeiten Eclipse als Plattform bietet, um eine Entwicklungsumgebung zu entwickeln. Im Kapitel „Integration des Compilers“ 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, entworfen, 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 Preference Page“ werden die Einstellungen, die 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, welche Möglichkeiten Eclipse bietet, eine moderne Entwicklungsumgebung zu implementieren. Die Möglichkeiten werden miteinander verglichen und die Vor- und Nachteile aufgezeigt. Auch werden die wichtigsten Eclipse-Features, die für das Entwickeln von Applikationen auf der Eclipse Rich Client Platform (RCP) benötigt werden, beschrieben.

2.1. Die Plattform

Eclipse RCP bietet eine Basis, um betriebssystemunabhängige Applikationen zu entwickeln. Es bietet Mechanismen wie Plugins und Extension Points, um modulares Programmieren zu unterstützen und zu vereinfachen. Auch bietet das Framework Features wie Views, Editoren und Perspektiven, die häufig in Applikationen gebraucht werden.

2.2. Plugins

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

2.3. Extension Points

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

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

Die Abbildung 2.1 zeigt, wie über diesen Extension Point eine neue Eclipse View erstellt werden kann.

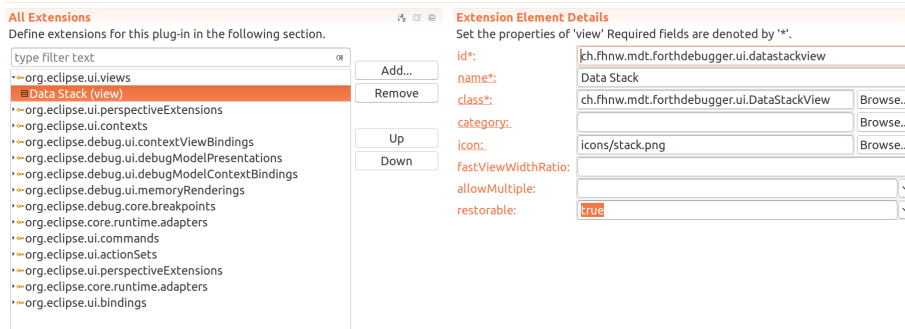


Abbildung 2.1.: Ein Plugin, das einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.

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

2.4. Eclipse basierte MCore Entwicklungsumgebung

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

2.4.1. JDT

Um die MCore-Entwicklungsumgebung zu implementieren, könnten die Standard Features von dem Eclipse Java Development Tools (JDT) verwendet werden. Dies wäre sehr aufwändig, 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, das das Programmieren einer auf Eclipse basierten Entwicklungsumgebungen erleichtert. Es ermöglicht, auf schnelle Art und Weise das Grundgerüst einer Entwicklungsumgebung mit folgenden Features zu generieren: [3]

- Editor mit Syntax Coloring
- Code Completion

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

Es muss lediglich eine ANTLR-ähnliche Grammatik für die Sprache definiert werden. [4] Eine Grammatik für den C-Editor zu definieren ist schwierig. Ausserdem müsste, für einen guten C-Editor, noch der Präprozessor mit einbezogen werden, was Xtext standardmässig nicht unterstützt.

Für den Forth-Editor ist es aber durchaus möglich, Xtext zu verwenden. 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, das das Grundgerüst für eine Entwicklungsumgebung generieren kann. Ursprünglich 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 dieselben Nachteile wie bei Xtext. Da C schwierig zu parsen ist, müsste trotz dem Framework noch viel 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

Eclipse C/C++ Development Tools (CDT) ist eine Eclipse Distribution mit Unterstützung für C und C++. CDT bietet viele Features für C, die schon vom JDT bekannt sind und stellt Extension Points zur Verfügung, um diese zu verwenden. So kann mit relativ wenig Aufwand ein neuer Compiler in die Entwicklungsumgebung eingebunden werden. Dies wurde auch schon mehrfach gebraucht, um verschiedene C/C++ Compiler in Eclipse CDT zu integrieren. Auf die Einbindung des Compilers wird im Kapitel 3 genauer eingegangen.

2.4.5. Verwendung für uCore Eclipse

Ich habe mich dazu entschieden, das Eclipse CDT als Target-Plattform zu wählen. Somit können alle Features vom Eclipse CDT verwendet werden. Xtext wird zusätzlich verwendet, um den uForth-Editor zu implementieren.

3. Integration des Compilers

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 Eclipse-CDT für das Editieren von C-Files zur Verfügung stehen.

3.1. Integration in Eclipse CDT

Das Eclipse-CDT stellt Extension Points zur Verfügung, die 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

Eine Konfiguration wird verwendet, um verschiedene standard Tools und Options bereitzustellen, die ein Projekt auf eine bestimmte Weise kompilieren. Normalerweise existieren für ein Projekt zwei Konfigurationen, eine Debug- und eine Releasekonfiguration. [6]

Verwendung

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

3.1.2. Tool

Ein Tool definiert ein Programm, wie zum Beispiel einen Compiler oder Linker, welches vom Buildprozess verwendet wird. [6]

Verwendung

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

3.1.3. Toolchain

Eine Toolchain ist eine Liste von Tools, die gebraucht werden, um den Output des Projekts zu generieren. [6]

Verwendung

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

3.1.4. CDT-Builder

Der CDT-Builder repräsentiert das Werkzeug, welches verwendet wird um den Build-Prozess zu steuern. Typischerweise ist es eine Variante von **make**. [6]

Verwendung

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

3.1.5. Project Builder

Der Project Builder wird nicht vom Eclipse CDT, sondern vom normalen Eclipse Build-prozess 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 fehlschlägt, wird der Build-Vorgang abgebrochen und die Fehler werden in der Problem View von Eclipse angezeigt. Die Abbildung 3.1 zeigt die Eclipse Problem View mit einem Fehler im Pfad.



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

In der Installationsanleitung im Anhang finden sich alle Schritte, die 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 Launch-Möglichkeiten unterschieden. In den nächsten Kapiteln werden die zwei Launch-Möglichkeiten erklärt.

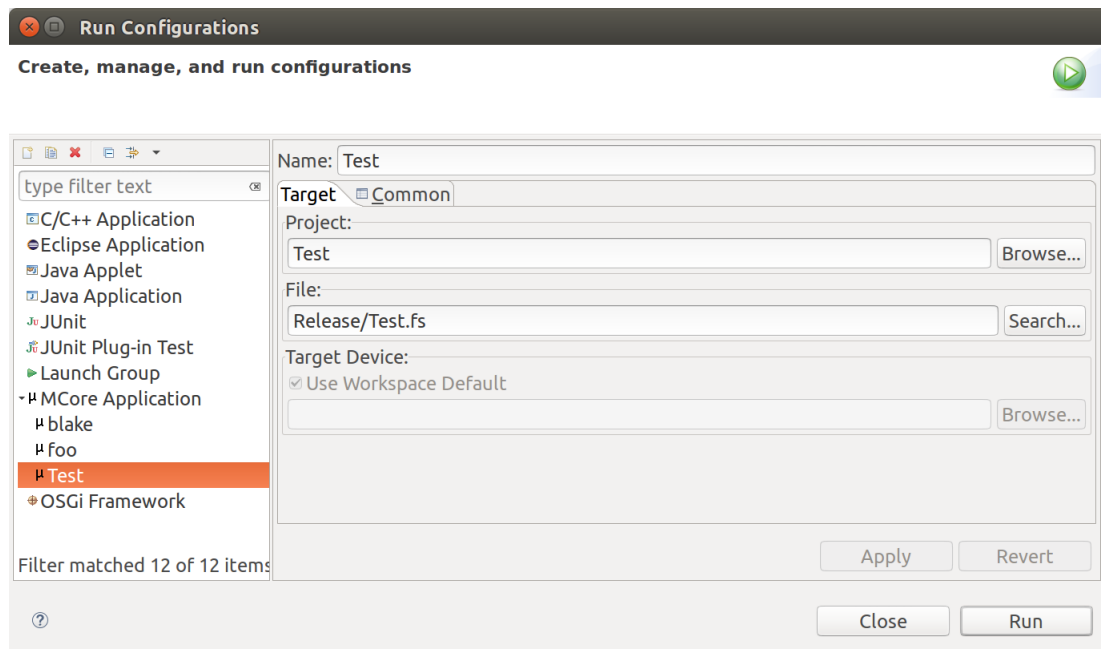


Abbildung 4.1.: Der Launch Configuration Dialog. In diesem Dialog kann eine neue MCore Launch-Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen, muss das Projekt und das Forth File, das gestartet werden soll, angegeben werden.

4.1.1. Run

Mit der Run-Konfiguration wird zuerst der Loader in den Forth Workspace kopiert, der 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 Preference Page). Falls der Umbilical Port ungültig ist, erscheint beim starten des Programms eine Fehlermeldung.

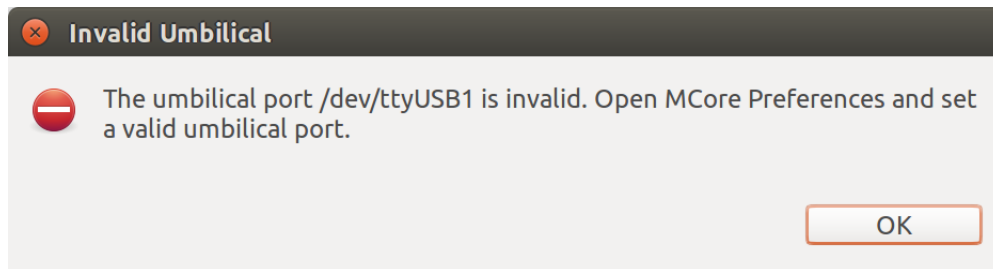


Abbildung 4.2.: Dialog, der erscheint, wenn ein Programm gestartet wird und der Umbilical Port ungültig ist. Der Port kann in der MCore Preference Page geändert werden.

4.1.2. Debug

In der Debug-Konfiguration werden zusätzlich alle Funktionen des gestarteten Forth Files disassembliert. 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 die Funktionalität der Entwicklungsumgebung davon abhängt, ob die Kommunikation stabil läuft. Es wird gezeigt, wie die Kommunikation entworfen, implementiert und getestet wurde.

5.1. Prozess Kommunikation

Um mit dem Prozess zu kommunizieren, gibt es die Möglichkeiten: ein Machine Interface zu implementieren, oder direkt mit dem Prozess zu kommunizieren. In den folgenden Kapiteln werden die beiden Möglichkeiten kurz beschrieben.

5.1.1. GDB/MI-Commands

Eine Möglichkeit, mit dem Prozess zu kommunizieren, ist der Gebrauch, eines Machine Interfaces (MI), wie es für den GDB implementiert wurde. GDB/MI ist ein linienbasiertes, maschinenorientiertes Text-Interface zum GDB. Es wurde entwickelt, um den GDB, als Debugger, in ein grösseres System einfacher einzubinden. [7] Ein MI ähnliches Interface wäre mit grossem Aufwand verbunden, da das Interface zuerst definiert werden müsste. Dafür könnte aber der CDT-Debugging-Mechanismus verwendet werden, da der CDT-Debugger auf dem GDB/MI basiert. Auch ist der CDT-Debugger sehr kompliziert [8] und es wäre ein grosser Einarbeitungsaufwand notwendig, den Forth Debugger darauf aufzubauen. [8]

5.1.2. Direkte Kommunikation mit dem Prozess

Eine weitere Möglichkeit besteht darin, die Befehle direkt an den Prozess zu senden und auf Antworten zu warten. Dafür müssen einige Klassen implementiert werden, um die Kommunikation zu vereinheitlichen und zu 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 oben genannten Klassen gelöst werden.

5.2. Kommunikation

Ich habe mich dafür entschieden, die Kommunikation nicht über ein MI-Interface zu implementieren. Der Aufwand, ein solches Interface zu entwerfen und zu 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 den folgenden Kapiteln besprochen.

5.3. API-Design

In einem ersten Schritt wurde ein API entworfen, das die Kommunikation mit dem Prozess möglichst einfach halten soll.

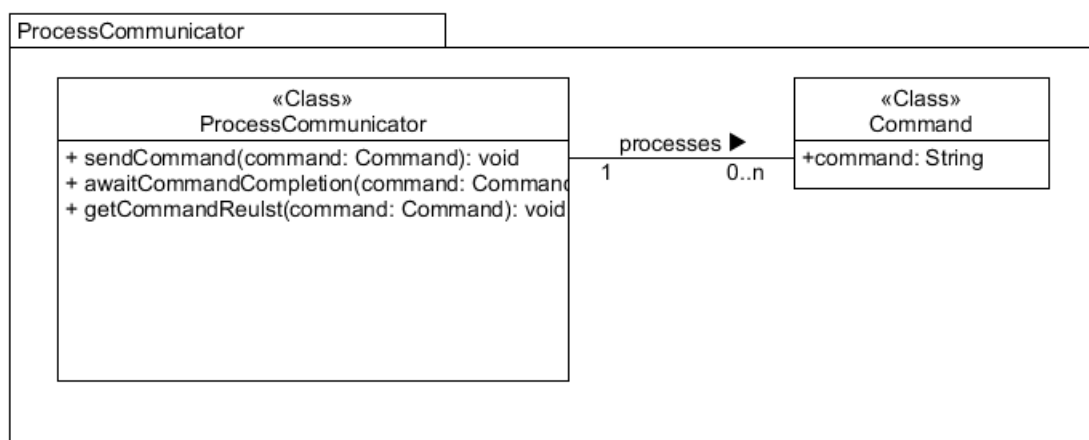


Abbildung 5.1.: Ein erstes Design des Prozess-API. Die zentrale Kommunikation geschieht über die Communicator-Klasse. Es können Befehle gesendet und Resultate abgewartet werden.

5.4. Implementierung

Bei der Implementierung kamen Änderungen hinzu. Im folgenden Klassendiagramm sind die Änderungen ersichtlich. Danach wird die Klassenstruktur genauer erläutert.

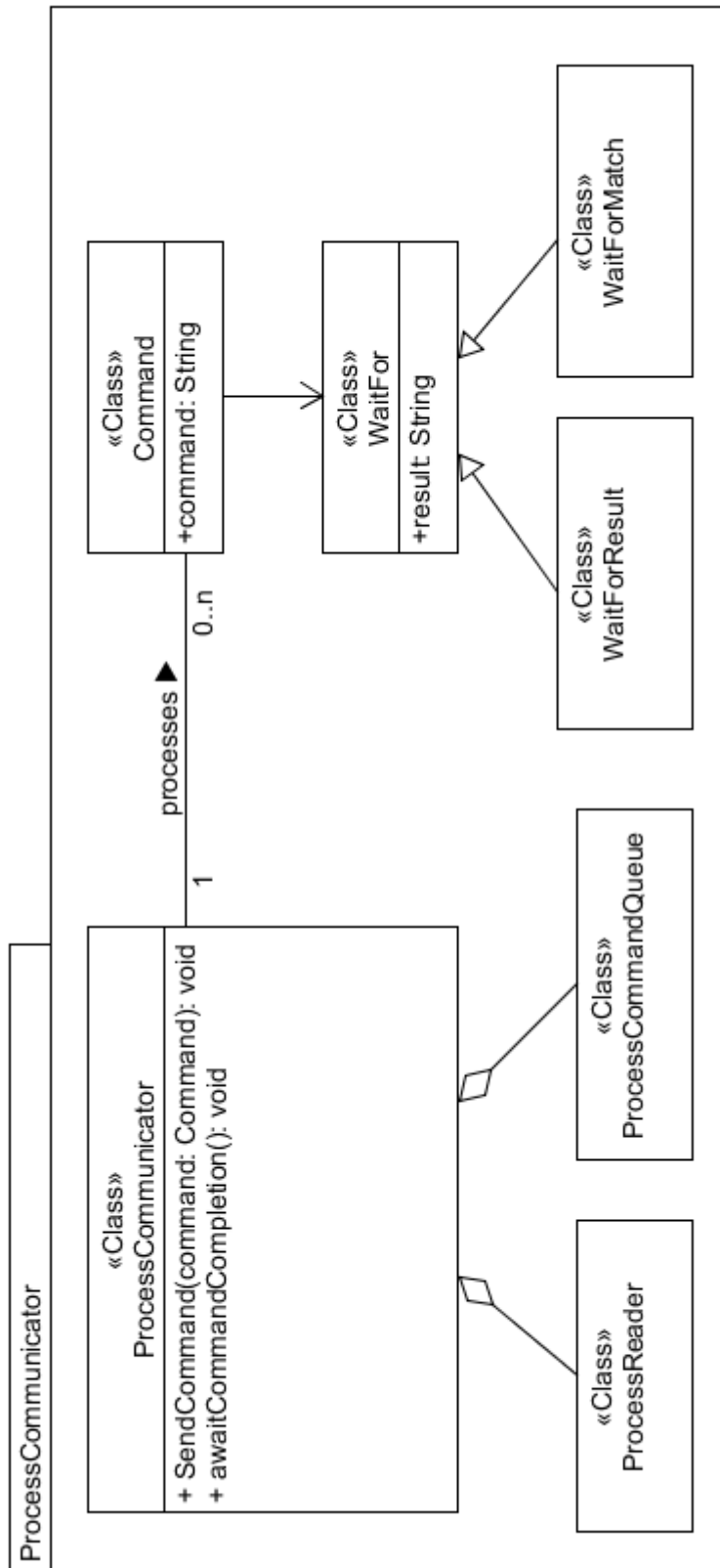


Abbildung 5.2.: Klassendiagramm des ProcessCommunicators.

5.4.1. Klassenbeschreibung

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

ProcessCommunicator

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

ProcessReader

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

ProcessCommandQueue

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

Command

Die `Command`-Klasse repräsentiert einen Befehl, der über den `ProcessCommunicator` an den Prozess gesendet werden kann. In einem `Command` kann ein `WaitFor` spezifiziert werden, falls der `Command` blockieren soll, bis ein Resultat vom Prozess geliefert wird.

WaitFor

Mit der abstrakten `WaitFor`-Klasse können Resultate des Prozesses abgewartet werden. Dafür wurden verschiedene Implementationen bereitgestellt. Mit der `WaitForResult` kann String-Resultate abgewartet werden. Mit der `WaitForMatch`-Klasse kann per Regex ein Resultat abgewartet werden. Im folgenden 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 1 und 9 schreibt. Für das Warten auf ein Resultat wurde ein Timeout implementiert. Falls der Prozess in der vorgegebenen Zeit keine Antwort gibt, wird eine `CommandTimeoutException` geworfen und der `ProcessCommunicator` wird heruntergefahren, um das Senden weiterer Befehle zu verhindern.

5.5. Testing

Der `ProcessCommunicator` wurde vor allem mit Unit-Tests getestet. Unter diesen Tests befindet sich ein Stresstest, welcher möglichst viele Befehle an den Prozess sendet.

6. uForth-Editor

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

6.1. Xtext-Implementation des uForth-Editors

Xtext ermöglicht es, das Grundgerüst einer Entwicklungsumgebung zu generieren. Dafür verwendet Xtext eine Extended Backus-Naur Form (EBNF) ähnliche Grammatik. Aus dieser Grammatik wird ein ANTLR-Parser und Klassen, die 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 keinen monolithischen Compiler und somit keine statische Grammatik besitzt, kann keine Grammatik für Xtext geschrieben werden, die die gesamte Forth-Sprache beschreibt. Es wurde deshalb eine Xtext-Grammatik verwendet, die ungefähr der folgenden EBNF-Grammatik entspricht.

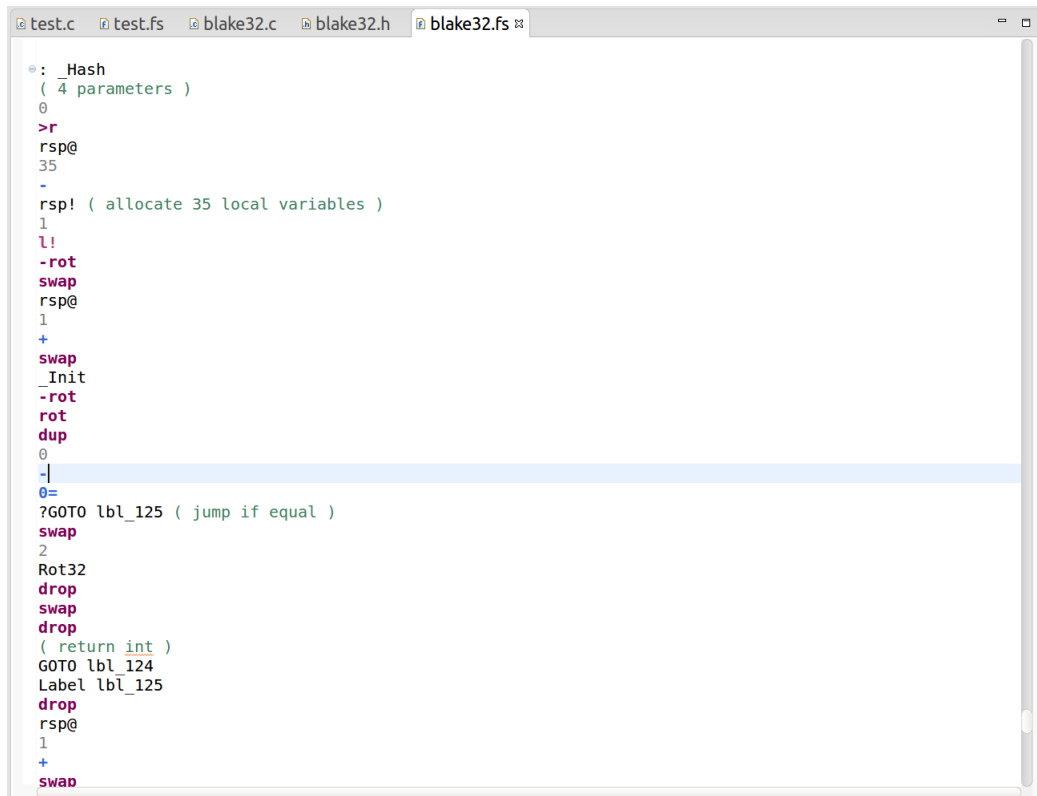
```
instructoin = create | function | word;  
function = ':', identifier, { Word }, ':';  
create = 'create', identifier, { literal ',', ' };  
word = identifier;
```

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

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-Wörter, Memory-Wörter und arithmetischewörter. Alle Gruppen haben eine andere Farbe, die in der uForth Preference Page geändert werden kann. Die Abbildung 6.1 zeigt den uForth-Editor.



```
test.c test.fs blake32.c blake32.h blake32.fs
: _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, der vor allem für den Debugger verwendet wird.

Es wurde eine Dokumentation für uForth-spezifischen Wörter, als Popup im Editor implementiert. Die Dokumentation wird dafür aus einer Textdatei gelesen. Die Abbildung 6.2 zeigt eine Popup-Dokumentation zum Wort `label`.

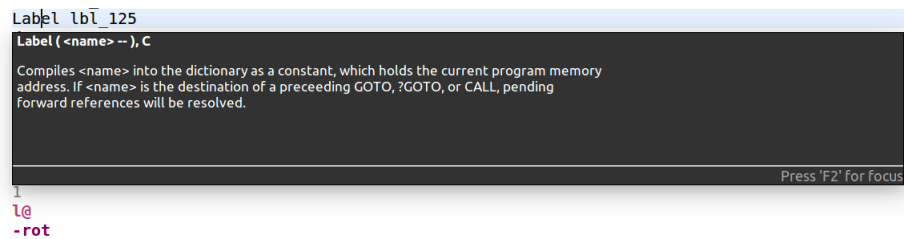


Abbildung 6.2.: Popup Dokumentation zum Wort `label`. Die Dokumentation wird angezeigt, falls der Cursor sich über einem Wort befindet, für das eine Dokumentation verfügbar ist.

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 Wörtern.

7. Debugger

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

7.1. Breakpoints

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

7.1.1. Per Konsole

Breakpoints können per Konsole gesetzt werden. Das Senden der Befehle funktioniert mit den im Kapitel 5 beschriebenen Klassen. In der Konsole kann der Befehl

```
debug _function
```

einggegeben werden, um einen Breakpoint zu setzen. Wobei `_function` für eine beliebige Forth Funktion steht.

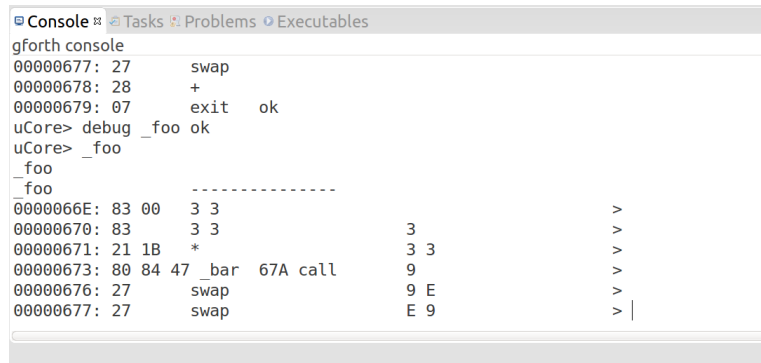
7.1.2. Im Source Code

Breakpoints können im C-File gesetzt werden. Dies wurde so realisiert, 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. Konsolenbasierter Debugger

Eine einfache Integration des Debuggers besteht in der Verwendung, des Konsolen Debugger von Forth im Eclipse. Dieser kann mit den im Kapitel 5 beschriebenen 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
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.: Konsolenbasierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden.

7.3. Forth Debugger

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

7.3.1. CDT oder JDT Debugging-Mechanismen

Eclipse stellt mehrere Möglichkeiten zur Verfügung, einen Debugger zu integrieren. Es können die Mechanismen vom Eclipse JDT (insbesondere das Plugin `org.eclipse.debug.core`) verwendet werden oder die erweiterten Mechanismen von CDT (insbesondere das Plugin `org.eclipse.cdt.debug.core`) mit denen C oder C++ Debugger integriert werden. Das vom CDT zur Verfügung gestellte Plugin wird vor allem dazu verwendet, um einen neuen 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, die Eclipse JDT Debugging-Mechanismen zu verwenden.

7.3.2. Debugger-Aktionen

Für den Debugger wurden einige Aktionen, die schon in Eclipse verwendet werden, implementiert und einige neue Forth-spezifische Aktionen hinzugefügt.

Resume

Mit der Resume-Aktion kann der Prozess während des Debuggens fortgeführt werden. Dafür wird ein **end-trace** Befehl an den Forth-Prozess gesendet. Der **end-trace** Befehl entfernt auch alle Breakpoints. Diese werden von der Resume-Aktion automatisch nach der Ausführung neu gesetzt.

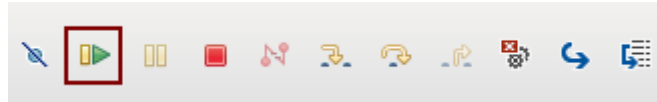


Abbildung 7.2.: Resume-Aktion.

Terminate

Mit der Terminate-Aktion kann der Prozess heruntergefahren werden, indem der **bye** Befehl 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.

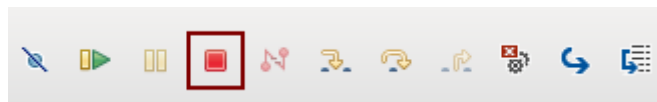


Abbildung 7.3.: Terminate-Aktion.

Step Into

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

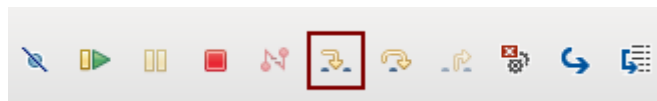


Abbildung 7.4.: Step-Into-Aktion.

Step

Mit der Step-Aktion kann ein Single Step ausgeführt werden. Dafür wird ein `CR` Befehl 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` Befehl beendet.

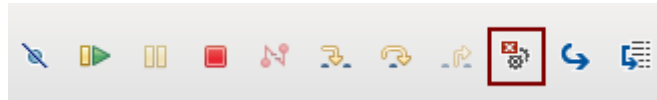


Abbildung 7.6.: Kill Aktion.

After

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

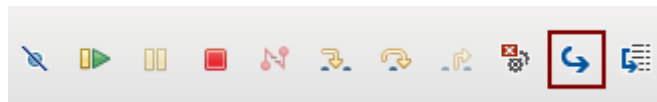


Abbildung 7.7.: After-Aktion.

Jump

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

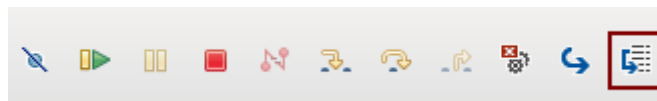


Abbildung 7.8.: Jump-Aktion.

7.3.3. Stack View

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

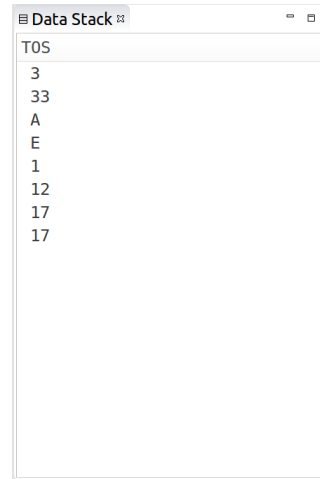


Abbildung 7.9.: Stack View mit aktuellem Inhalt des Dstacks. Der Top Of Stack (TOS) befindet sich zuoberst in der Liste.

7.3.4. Memory View

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

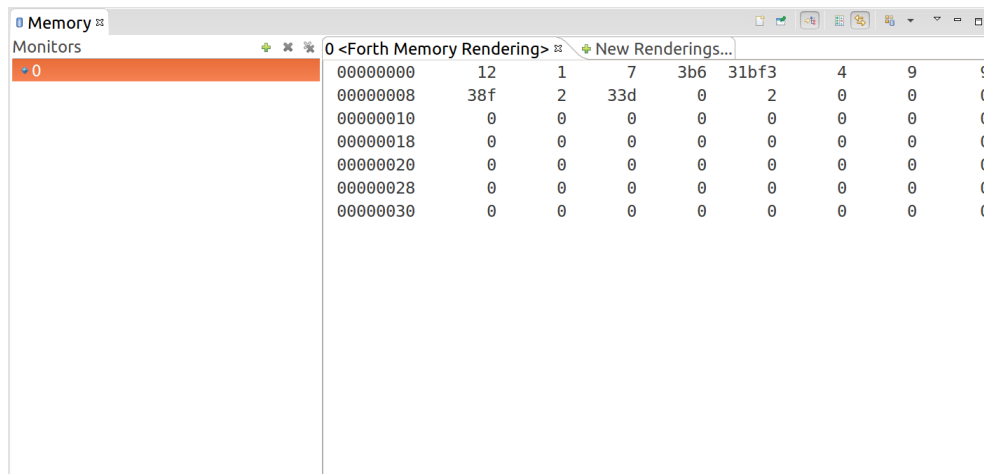


Abbildung 7.10.: Die Memory View mit dem Inhalt eines dump-Befehls.

7.4. C-Debugger

Der Debugger könnte so erweitert werden, dass er nicht wie bisher auf dem generierten Forth-Code arbeitet, sondern direkt auf dem C-Source Code. Dies konnte noch nicht umgesetzt werden, weil Debug-Informationen des Compilers fehlen. Der C-Source Code kann nicht auf den entsprechenden Forth Source Code abgebildet werden.

8. Entwicklungsumgebungs Preference Page

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

8.1. Umbilical Port

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

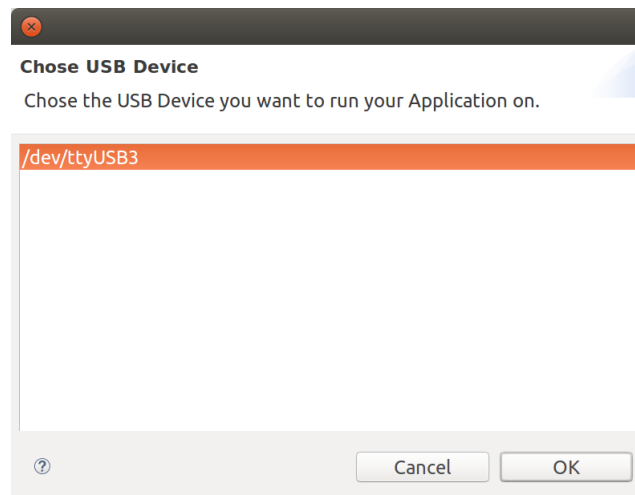


Abbildung 8.1.: Dialog, über welchen der Umbilical-Port gesetzt werden kann.

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

8.2. Loader

Der Loader ist das File, das von Forth mit dem Befehl

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

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

9. Optimierungen

In diesem Kapitel wird beschrieben, welche 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 es wird gezeigt, wo noch bessere Optimierungsstrategien verwendet werden können.

9.1. Peephole-Optimierung

Peephole-Optimierungen sind Optimierungen, die auf einer kleinen Sequenz von Instruktionen durchgeführt werden. Diese Sequenz wird Peephole oder auch Window genannt. Die Peephole-Optimierung versucht, Paare von Instruktionen durch kürzere oder schnellere Instruktionen zu ersetzen. [9] Peephole-Optimierungen können den Code um 15–40 Prozent verkleinern und sind heute in allen gängigen Compilern implementiert. [10] Zu den Peephole Optimierungen gehören unter anderen folgende Arten von Optimierungen:

- Constant Folding - konstante Ausdrücke auswerten
- Constant Propagation - konstante Werte in Ausdrücken substituieren
- Strength Reduction - langsame Instruktionen durch äquivalente schnelle Instruktionen ersetzen
- Combine Operations - mehrere Operationen durch eine äquivalenten Operation ersetzen
- Null Sequences - unnötige Operationen entfernen [9]

9.1.1. Beispiele

Einige Beispiele von Peephole-Optimierungen in Forth Code.

Constant Propagation

Die Instruktionen

```
1
2
swap
+
dup
```

können durch

```
3
3
```

ersetzt werden. Die Instruktionen `swap`, `+` und `dup` können schon beim Kompilieren durchgeführt werden.

Combine Operations

Die Instruktionen

```
rot
rot
```

können durch

```
-rot
```

ersetzt werden. Die zwei `rot` Instruktionen sind äquivalent zu einer `-rot`-Instruktion.

Die Instruktionen

```
dup
drop
```

können durch

```
nop
```

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

9.1.2. Optimierungen

Für den Compiler wurden zwei Peephole-Optimierungs Prototypen 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:

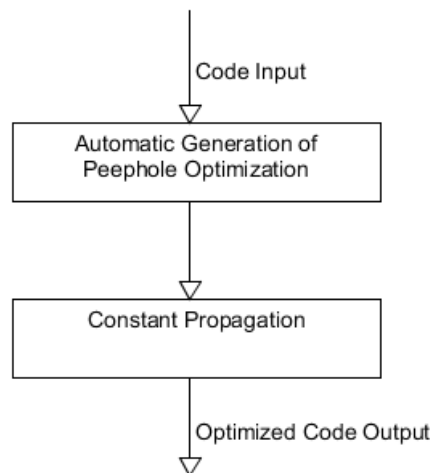


Abbildung 9.1.: Die zwei Phasen des Optimizers.

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

1. `<lit> + ld`, `<lit> + st`, `<lit> + @`, and `<lit> + !` werden durch automatisch inkrementierende Speicherzugriffsinstruktionen ersetzt, wenn `<lit>` sich zwischen `-4` und `3` befindet
2. Die Patterns
 $swap, over \rightarrow tuck$
 $over, swap \rightarrow under$
 $swap, drop \rightarrow nip$
 $r >, > r \rightarrow nop$
werden vom Forth-Cross-Compiler erkannt und optimiert

Für eine komplette Liste siehe „uForth real time, object oriented with debugger“ [11].

9.1.3. Automatische Generierung von Peephole Optimierungen

Klassische Peephole Optimizer versuchen häufig, einige maschinenspezifische Patterns zu korrigieren. Der von Davidson und Fraser [10] beschriebene Algorithmus (PO) verwendet eine Machine Description, simuliert benachbarte Instruktionen und versucht, diese durch äquivalente, schnellere Instruktionen zu ersetzen. Für den Forth-Optimizer wurde ein Teil von PO objektorientiert implementiert.

9.1.4. Constant Propagation

Unter Constant Propagation versteht man das Vorwärtssubstituieren von Konstanten im Code. Dies kann zur Folge haben, dass mehrere Instruktionen schon beim Kompilieren ausgewertet werden können, wie die Beispiele 9.1.1 zeigen.

9.1.5. Resultate und Tests

Die Resultate des Optimizers wurden mit verschiedenen Forth-Funktionen getestet und 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, die neu implementiert wurde, konnte keine durchgeführt werden. PO konnte Instruktionspaare finden, die vereinfacht werden können.
_Update	8	8	4	PO konnte zwei Instruktionspaare vereinfachen, welche von der Referenzimplementierung nicht optimiert werden.
_Hash	65	60	60	Die Referenzimplementierung produziert wegen der Speicherzugriffsoptimierung weniger Code. PO konnte Instruktionspaare finden, welche vereinfacht werden können.
_Propagation	11	11	4	Bei dieser Funktion konnte vor allem Constant Propagation durchgeführt werden. Die Referenzimplementierung führt keine Constant Propagation durch und konnte somit nichts vereinfachen.

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 die Constant Propagation effizienter zu gestalten.

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

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

Diese Regeln könnten in dem Forth-Cross-Compiler integriert werden. Im nächsten Kapitel werden Änderungen für den PO vorgeschlagen, damit dieser mehr Instruktionspaare, die vereinfacht werden können, erkennt.

9.1.6. Mögliche Erweiterungen

Im Moment werden vom PO nur Instruktionen simuliert, die einen Einfluss auf den Daten-Stack haben. PO könnte erweitert werden, indem auch andere Instruktionen simuliert werden, um mögliche Vereinfachungen zu finden. Eine weitere Möglichkeit wäre, den von Bansal und Aiken beschriebene Superoptimizer zu implementieren. Dieser Superoptimizer verwendet Bruteforce-Optimierungen mit tausenden von Regeln. Diese Regeln werden von Trainingsprogrammen inferiert und in einer Datenbank gespeichert. [12]

Die Constant Propagation könnte so erweitert werden, dass sie auch mit Branches umgehen kann. Somit könnten unnötige If-Statements und Schleifen entfernt werden. Um dies zu implementieren müsste der Code zuerst in eine Static Single Assignment Form (SSA) transformiert werden. [13] Es wäre dann jedoch 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] Managed build definitions. <http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fextension-points%2Findex.html>.
- [7] The gdb/mi interface. https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.
- [8] Understanding the gnu debugger machine interface (gdb/mi). <http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug2/>.
- [9] Peephole optimization. https://en.wikipedia.org/wiki/Peephole_optimization.
- [10] 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.
- [11] Klaus Schleisiek. uforth real time, object oriented with debugger, 2013.
- [12] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ACM SIGPLAN Not.*, pages 394–403, 2006.
- [13] Static single assignment form. https://en.wikipedia.org/wiki/Static_single_assignment_form.
- [14] Xtext download. <https://eclipse.org/Xtext/download.html>.

B. Abbildungsverzeichnis

2.1. Ein Plugin, das einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.	7
3.1. Die Eclipse Problem View zeigt, dass das Program lcc-mcore nicht im Pfad gefunden wurde.	10
4.1. Der Launch Configuration Dialog. In diesem Dialog kann eine neue MCore Launch-Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen, muss das Projekt und das Forth File, das gestartet werden soll, angegeben werden.	11
4.2. Dialog, der erscheint, wenn ein Programm gestartet wird und der Umbilical Port ungültig ist. Der Port kann in der MCore Preference Page geändert werden.	12
5.1. Ein erstes Design des Prozess-API. Die zentrale Kommunikation geschieht über die Communicator-Klasse. Es können Befehle gesendet und Resultate abgewartet werden.	14
5.2. Klassendiagramm des ProcessCommunicators.	15
6.1. uForth-Editor, der vor allem für den Debugger verwendet wird.	19
6.2. Popup Dokumentation zum Wort label. Die Dokumentation wird angezeigt, falls der Cursor sich über einem Wort befindet, für das eine Dokumentation verfügbar ist.	20
6.3. Outline zu einem Forth File. Die Outline gliedert das File in Funktionen und Wörter.	20
7.1. Konsolenbasierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden.	22
7.2. Resume-Aktion.	23
7.3. Terminate-Aktion.	23
7.4. Step-Into-Aktion.	23
7.5. Step-Aktion.	24
7.6. Kill Aktion.	24
7.7. After-Aktion.	24
7.8. Jump-Aktion.	24
7.9. Stack View mit aktuellem Inhalt des Dstacks. Der Top Of Stack (TOS) befindet sich zuoberst in der Liste.	25
7.10. Die Memory View mit dem Inhalt eines dump-Befehls.	25

8.1. Dialog, über welchen der Umbilical-Port gesetzt werden kann.	27
9.1. Die zwei Phasen des Optimizers.	30
E.1. Mit der Target Definition kann die Target-Platform gesetzt werden.	38
E.2. Der MWE2 Workflow generiert alle notwendigen Klassen für den Forth- Editor.	39
E.3. Die Entwicklungsumgebung kann im plugin.xml des Plugins ch.fhnw.mdt.ui gestartet werden.	39

C. Tabellenverzeichnis

9.1. Resultate des neuen Optimizers verglichen mit dem Cross-Compiler Optimizer.	31
--	----

D. Installationsanleitung für Anwender

Zuerst muss lcc-mcore und gforth nach Anleitung der jeweiligen Dokumentation installiert werden. lcc-mcore muss sich im `PATH` befinden und `GFORTHPATH` muss gesetzt sein. Falls eine dieser Einstellungen nicht korrekt ist, wird die Entwicklungsumgebung das Projekt nicht komplett bilden. 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 wird mit dem Befehl `sudo env PATH=$PATH ./eclipse` korrekt gestartet.

E. Installationsanleitung für Entwickler

Zuerst müssen alle Schritte, des Kapitels „Installationsanleitung für Anwender“ durchgeführt werden. Für die Entwicklung wird eine Eclipse RCP-Distribution mit Xtext-Unterstützung benötigt. Diese kann auf der offiziellen Xtext-Webseite [14] heruntergeladen werden. Zuerst müssen alle Projekte in Eclipse importiert werden. Dann muss die Target Platform gesetzt werden. Dies kann im File `cdt-8.5.target` erfolgen, wie die Abbildung E.1 zeigt.

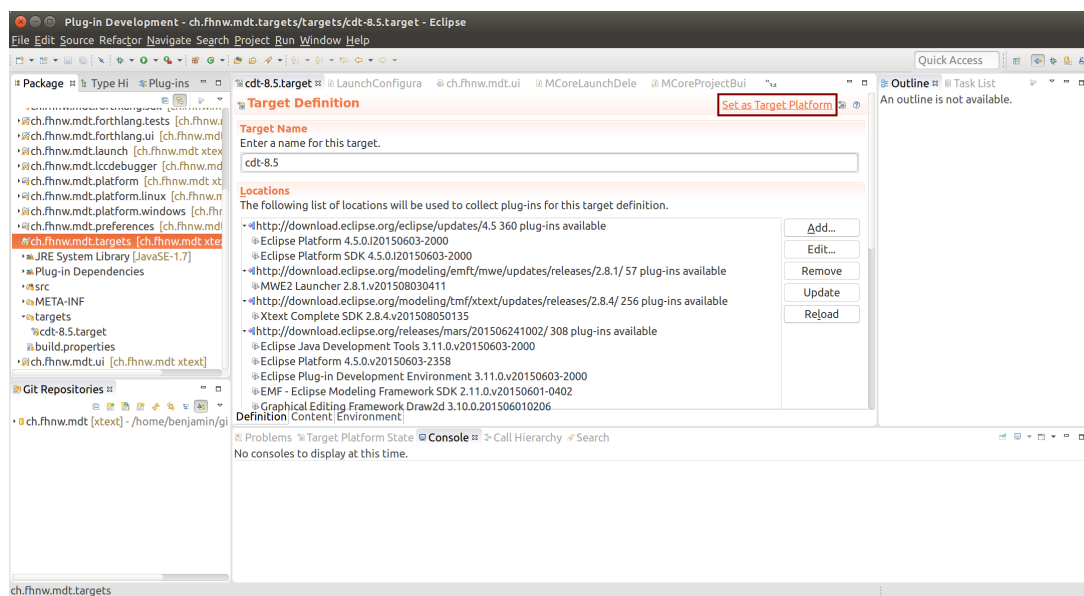


Abbildung E.1.: Mit der Target Definition kann die Target-Platform gesetzt werden.

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 Plugin `ch.fhnw.mdt.forthlang` gestartet werden, wie die Abbildung E.2 zeigt.

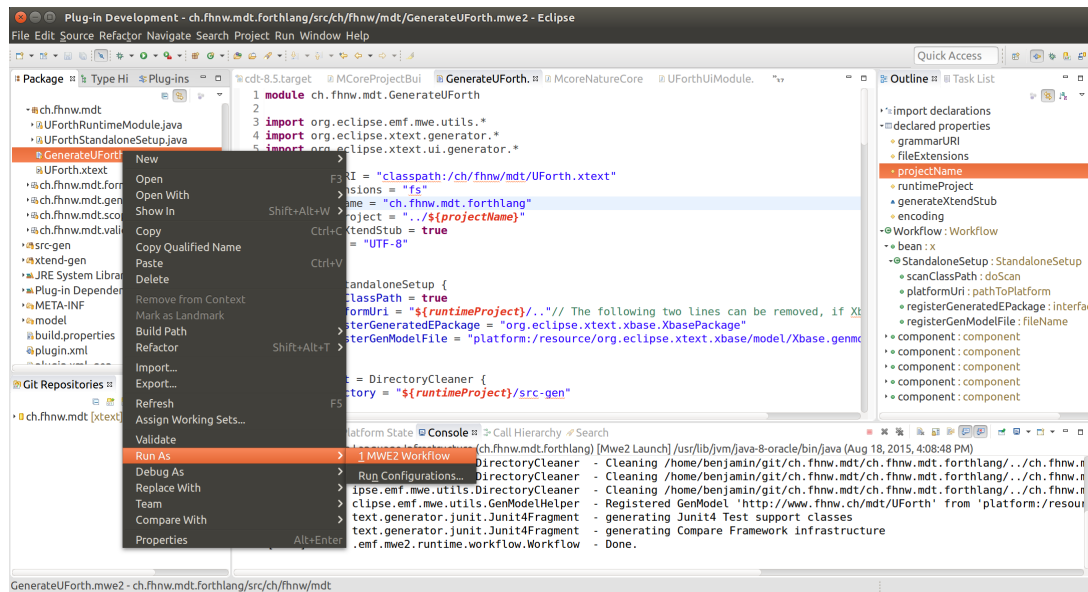


Abbildung E.2.: Der MWE2 Workflow generiert alle notwendigen Klassen für den Forth-Editor.

Die Entwicklungsumgebung kann danach im plugin.xml des Plugins `ch.fhnw.mdt.ui`, wie in der Abbildung E.3 zu sehen ist, gestartet werden.

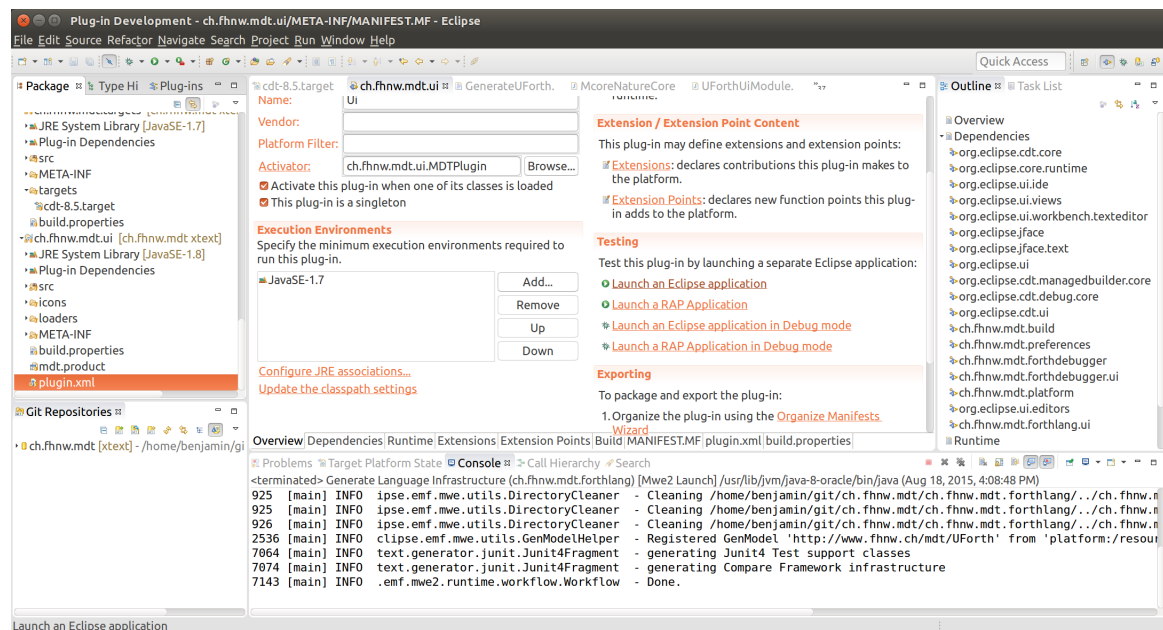


Abbildung E.3.: Die Entwicklungsumgebung kann im plugin.xml des Plugins `ch.fhnw.mdt.ui` gestartet werden.

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 bestätigt der Autor, diese Arbeit ohne fremde Hilfe und unter Einhaltung der gebotenen Regeln erstellt zu haben.

Benjamin Neukom

Ort, Datum

Unterschrift