

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Danksagungen

asdf

asdf

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
1 Motivation	1
1.1 RETI-Architektur	2
1.2 Die Sprache PicoC	2
1.3 Eigenheiten der Sprache C	3
1.4 Gesetzte Schwerpunkte	4
1.5 Über diese Arbeit	5
1.5.1 Still der Schriftlichen Ausarbeitung	6
1.5.2 Aufbau der Schriftlichen Arbeit	6
Literatur	A

Abbildungsverzeichnis

1.1	Schritte zum Ausführen eines Programmes mit dem GCC	1
1.2	Stark vereinfachte Schritte zum Ausführen eines Programmes	1
1.3	README.md im Github Repository der Bachelorarbeit	5

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

1.1	Deklaration	3
1.2	Definition	3
1.3	Scope	3
1.4	Call by value	3
1.5	Call by reference	3
1.6	Imperative Programmierung	4
1.7	Strukturierte Programmierung	4
1.8	Prozedurale Programmierung	4

Grammatikverzeichnis

1 Motivation

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiell** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprache L_{Python} , welche als **interpretierte** Sprache bekannt ist, wird das in der Programmiersprache L_{Python} geschriebene Programm vorher zu **Bytecode** kompiliert, bevor dieser von der **Python Virtual Machine (PVM)** interpretiert wird.

Compiler, wie der **GCC**¹ oder **Clang**² werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert³ zu **Maschinencode** zu kompilieren.

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll, z.B. im Fall des **GCC** über `> gcc program.c -o machine_code`⁴. Als Ergebnis erhält man im Fall des **GCC** die mit der Option `-o` selbst benannte Datei `machine_code`, welche dann zumindest unter **Unix** über `> ./machine_code` **ausgeführt** werden kann, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

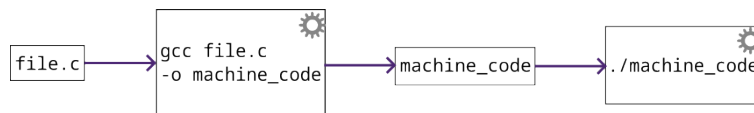


Abbildung 1.1: Schritte zum Ausführen eines Programmes mit dem GCC

Der ganze Kompilervorgang kann, wie er in Abbildung 1.2 dargestellt ist zu einer Box abstrahiert werden. Der Benutzer gibt ein **Programm** in der Sprache des Compilers rein und erhält **Maschinencode**, den er dann im besten Fall in eine andere Box hineingeben kann, welche die passende **Maschine** oder den passenden **Interpreter** in Form einer **Virtuellen Maschine** repräsentiert, der bzw. die den **Maschinencode** ausführen kann.

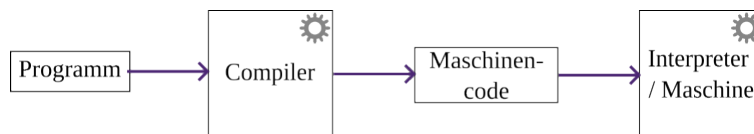


Abbildung 1.2: Stark vereinfachte Schritte zum Ausführen eines Programmes

¹GCC, the GNU Compiler Collection - GNU Project.

²clang: C++ Compiler.

³Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C .

⁴Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objectcode** (Definition ??) kompiliert wird. Das Kompilieren zu **Objectcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

Der Programmierer muss für das Vorgehen in Abbildung 1.2 **nichts** über die **Theoretischen Grundlagen des Compilerbau** wissen, noch wie der Compiler **intern** umgesetzt ist. In dieser Bachelorarbeit soll diese **Compilerbox** allerdings geöffnet werden und anhand eines eigenen im Vergleich zum **GCC** im Funktionsumfang **reduzierten Compilers** gezeigt werden, wie so ein Compiler **unter der Haube** stark vereinfacht funktionieren könnte.

Die konkrete **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC} , welche eine Untermenge der Sprache L_C ist⁵ in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zu Grunde liegt und im Unterkapitel 1.2 wird näher auf die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll.

1.1 RETI-Architektur

Die **RETI-Architektur** ist eine zu Lernzwecken für die Vorlesungen P. D. C. Scholl, „Betriebssysteme“ und P. D. C. Scholl, „Technische Informatik“ entwickelte **32-Bit** Architektur, die sich vor allem durch ihre einfache Zugänglichkeit kennzeichnet und deren **Maschinensprache** L_{RETI} als Zielsprache des **PicoC-Compilers** hergenommen wurde. In der Vorlesung P. D. C. Scholl, „Technische Informatik“ wird die **grundlegende RETI-Architektur** erklärt und in der Vorlesung P. D. C. Scholl, „Betriebssysteme“ wird diese Architektur erweitert, sodass diese mehr darauf angepasst ist, dass auch komplexere Konstrukte, wie ein **Betriebssystem**, **Interrupts**, **Funktionen** usw. auf nicht zu komplizierte Weise implementiert werden können.

Anmerkung

In dieser **Bachelorarbeit** wird bei der **Maschinensprache** L_{RETI} immer von der Variante, welche durch die **RETI-Architektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ umgesetzt ist^a ausgegangen.

^aWelche unter anderem eine **Memory Map** von **EPROM**, **UART** und **SRAM** nutzt usw.

Der **Aufbau** der **Maschinensprache** L_{RETI} ist in Grammatik ?? und Grammatik ?? zusammengefasst dargestellt. Für die **Bedeutungen** der einzelnen **Register** und **Maschinenbefehle**, sowie **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

Um den den **PicoC-Compiler** zu **testen** war es notwendig einen **RETI-Interpreter** zu implementieren, der genau die Variante der **RETI-Architektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ **simuliert**.

1.2 Die Sprache PicoC

Die Sprache L_{PicoC} ist eine Untermenge der Sprache L_C , welche

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/*` und `*/`
- die **Primitiven Datentypen** `int`, `char` und `void`
- die **Abgeleiteten Datentypen Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`)

⁵Die der **GCC** kompilieren kann.

- `if(cond){ }- / else{ }-Statements`⁶
- `while(cond){ }-` und `do while(cond){ };-Statements`
- **Arihmetische Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` und **unären Operatoren** `-`, `~` umgesetzt sind
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`) und **Funktionsaufrufe** (z.B. `fun(ar, st.var)`)

beinhaltet. Die ausgegrauten • wurden bereits für das **Bachelorprojekt** umgesetzt und mussten für die **Bachelorarbeit** nur an die **neue Architektur** angepasst werden.

Der **Aufbau** der **Programmiersprache** L_C ist in Grammatik ?? und Grammatik ?? zusammengekommen beschrieben.

1.3 Eigenheiten der Sprache C

Definition 1.1: Deklaration

a

^aP. D. P. Scholl, „Einführung in Embedded Systems“.

Definition 1.2: Definition

a

^aP. D. P. Scholl, „Einführung in Embedded Systems“.

Definition 1.3: Scope

a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.4: Call by value

a

^aBast, „Programmieren in C“.

Definition 1.5: Call by reference

a

^aBast, „Programmieren in C“.

⁶Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

Definition 1.6: Imperative Programmierung

Wenn ein Programm aus einer **Folge von Befehlen** besteht, deren **Reihenfolge** auch bestimmt in welcher **Reihenfolge** diese **Befehle** auf einer **Maschine** ausgeführt werden.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.7: Strukturierte Programmierung

Wenn ein Programm anstelle von z.B. `goto label`-Statements **Kontrollstrukturen**, wie z.B. `if(cond) { } else { }`, `while(cond) { }` usw. verwendet, welche dem Programmcode **mehr Struktur** geben, weil die **Auswahl** zwischen Statements und die **Wiederholung** von Statements eine **klare und eindeutige Struktur** hat, welche bei Umsetzung mit einem `goto label`-Statement **nicht so eindeutig erkennbar** wäre und auch **nicht** unbedingt immer **gleich aufgebaut** wäre.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.8: Prozedurale Programmierung

Programme werden z.B. mittels **Funktionen** in **überschaubare Unterprogramme** bzw. **Prozeduren** aufgeteilt, die **aufzurufbar** sind. Dies **vermeidet** einerseits **redundanten Code**, indem Code **wiederverwendbar** gemacht wird und andererseits erlaubt es z.B. Codestücke nach ihren Aufgaben zu **abstrahieren**, den Codestücken wird eine **Aufgabe zugeteilt**, sie werden zu **Unterprogrammen** gemacht und fortan über einen **Bezeichner aufgerufen**, was den Code deutlich **überschaubarer** macht. da man die Aufgabe eines Codestücks nun nur noch mit seinem **Bezeichner assoziieren** muss.^a

^aThiemann, „Einführung in die Programmierung“.

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinsprache L_{RETI} die **Syntax** und **Semantik** der Sprache L_C identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache L_{PicoC} im Vergleich zu z.B. dem **GCC**⁷ ohne merklichen Unterschied⁸ kompilieren können.

In **zweiter Linie** soll dabei möglichst immer so Vorgegangen werden, wie es die **RETI-Codeschnipsel** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ vorgeben. Allerdings sollten diese bei **Inkonsistenzen** bezüglich der durch sie selbst vorgegebenen **Paradigmen** und anderen **Umstimmigkeiten** angepasst werden, da der **erstere Schwerpunkt** überwiegt.

Des Weiteren ist die **Laufzeit** bei Compilern zwar vor allem in der Industrie **nicht unwichtig**, aber bei **Compilern**, verglichen mit **Interpretern** weniger zu gewichten, da ein Compiler bei einem fertig implementierten Programm nur **einmal** Maschinencode generieren muss und dieser Maschinencode danach fortan ausgeführt wird. Beim einem **Compiler** ist daher eher zu priorisieren, dass der kompilierte **Maschinencode** möglichst **effizient** ist.

⁷Da die Sprache L_{PicoC} eine **Untermenge** von L_C ist, kann der **GCC** L_{PicoC} ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinsprache L_{RETI} .

⁸Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinsprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter [Link⁹](#) zu finden. In der Datei **README.md** (siehe Abbildung 1.3) ist unter „**Getting Started**“ ein kleines **Einführungstutorial** verlinkt. Unter „**Usage**“ ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Daneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter [Link¹⁰](#) zu finden.



Abbildung 1.3: *README.md* im Github Repository der Bachelorarbeit

Die **Schriftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schriftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter [Link¹¹](#) zu finden. Die **PDF** der Schriftliche Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode**, welcher unter [Link¹²](#) veröffentlicht ist automatisch mithilfe der **Github Action** Nemec, *copy-file-to-another-repo-action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibliotheken** sind unter [Link¹³](#) zu finden¹⁴. Die Grafiken, die nicht mittels der Tikz Bibliothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**¹⁵ erstellt. Falls Interesse besteht **Grafiken** aus der Schriftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den **.svg**-Dateien von **Inkscape** im Ordner **/figures** zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibliotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der **README.md** unter „**References**“ bzw. direkt unter [Link¹⁶](#) zu finden.

⁹<https://github.com/matthejue/PicoC-Compiler>.

¹⁰<https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971>.

¹¹https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.

¹²<https://github.com/matthejue/Bachelorarbeit>.

¹³https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.

¹⁴Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.

¹⁵Developers, *Draw Freely* — **Inkscape**.

¹⁶https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

1.5.1 Still der Schriftlichen Ausarbeitung

In dieser **Schriftliche Ausarbeitung der Bachelorarbeit** sind die manche **Wörter** für einen besseren Lesefluss **hervorgehoben**. Es ist so gedacht, dass die **Hervorgehobenen Wörter** beim Lesen sichtbare **Ankerpunkte** darstellen an denen sich **orientiert** werden kann, aber auch damit der **Inhalt** eines vorher gelesener **Paragraphs** nochmal durch Überfliegen der Hervorgehobenen Wörter in **Erinnerung gerufen** werden kann.

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas genau so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht¹⁷ die Frage des „**warum**“.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**¹⁸ verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **konsistent** sind, wie auch das in Ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernommen, so wurde die Definition oder der entsprechende Teil in „**Anführungszeichen**“ gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox**, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

1.5.2 Aufbau der Schriftlichen Arbeit

Die **Schriftliche Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: **Motivation**, **??**, **??** und **??**.

Im momentanen Kapitel **Motivation**, wurde ein kurzer **Einstieg** in das Thema **Compilerbau** gegeben und die **zentrale Aufgabenstellung** der Bachelorarbeit erläutert, sowie auf **Schwerpunkte** und kleinere **Teilprobleme**, die eines **besonderen Fokus** bedürfen eingegangen.

Im Kapitel **??** werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische Vorgehen**, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen in Bezug zu Compilern und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

Des Weiteren wurden für **T-Diagramme** und **Formale Sprachen** eigene Unterkapitel erstellt. Für **T-Diagramme** wurde ein eigenes Unterkapitel erstellt, da sie häufig in der Schriftlichen Ausarbeitung verwendet werden und die **T-Diagramm Notation** nicht allgemein bekannt ist. Für **Formale Sprachen** wurde ein eigenes Unterkapitel erstellt, da für den Gutachter Prof. Dr. Scholl das Thema **Formale Sprachen** eher **fachfremd** ist, aber dieses Thema einige zentrale und wichtige Fachbegriffe besitzt, bei denen es wichtig ist die genaue **Definition** zu haben. Generell wurde im Kapitel **Einführung** versucht an Erklärungen nicht zu sparen, damit aufgrund dessen, dass das Thema eher fachfremd für Prof. Dr. Scholl ist für das Kapitel **Implementierung** keine wichtigen Aspekte unverständlich bleiben.

Im Kapitel **??** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers**, unterteilt in die verschiedenen **Phasen der Kompilierung** nach denen das Kapitel **Einführung** ebenfalls unterteilt ist erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähnliche Kapiteileinteilung**

¹⁷Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.

¹⁸Das ist ein **Beispiel** für eine **Fußnote**.

haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständnis eines Aspekts in Kapitel **Implementierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteileinteilung** die entsprechende Definition analog im Kapitel **Einleitung** gefunden werden.

Im Kapitel ?? wird ein **Überblick** über die **wichtigsten Funktionalitäten** des PicoC-Compilers gegeben, indem anhand **kleiner Anleitungen** gezeigt wird wie man diese verwendet. Des Weiteren wird darauf eingegangen, wie die **Qualitätsicherung** für den **PicoC-Compiler** umgesetzt wurde, also wie gewährleistet wird, dass der **PicoC-Compiler** funktioniert. Zum Schluss wird noch auf **weitere Erweiterungsideen** eingegangen, die auch interessant zu implementieren wären.

Im Kapitel ?? werden einige **Definitionen** und **Themen** angesprochen, die bei **Interesse** zur **weiteren Vertiefung** da sind und **unabhängig** von den anderen Kapiteln sind. Diese **Themen** und **Definitionen** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Diese **Themen** und **Definitionen** passen nicht ins Kapitel ??, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tun haben und auch nichts ins **Kapitel ??**, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in der Schriftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition ??) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

Literatur

Online

- *clang: C++ Compiler*. URL: <http://clang.org/> (besucht am 29.07.2022).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: <https://inkscape.org/> (besucht am 03.08.2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).

Bücher

- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand*. 1. Aufl. Wiley, 20. Nov. 2012.

Vorlesungen

- Bast, Prof. Dr. Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Scholl, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03.08.2022).
- Scholl, Prof. Dr. Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Prof. Dr. Peter. „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).

Sonstige Quellen

- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03.08.2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: <https://github.com/tueda/makefile4latex> (besucht am 03.08.2022).