Manual
Clang für RTOS-UH

Dokument-Revision **1.0.0.0** vom 09.09.2026

Inhaltsverzeichnis

Revisionshistorie

Rev.	Datum	Name	Änderung
1.0	15.07.2026	HK	Erstellung

1 Einleitung

Diese Anleitung dient als kurzer Einstieg für den Einsatz des Clang Compilers unter RTOS-UH. Der Clang Compiler wurde in der Version 22 angepasst und kann Code ausschließlich für PowerPC-basierte Systeme erzeugen. 68k Systeme werden mit dem Clang nicht unterstützt. Die erzeugten Binärobjektdateien sind dabei zu unseren bestehenden Crest-erzeugten Objekten kompatibel. Diese zentrale Eigenschaft erlaubt es, bereits übersetzte Libraries oder Code mit Clang-erzeugtem Code zu kombinieren.

Wichtig: Es muss ein Crest-Paket mit **Version 2.83 oder höher** installiert sein. In diesem Paket wurden unsere Linker CLN und LNK erweitert, um Clang-erzeugte Objekte verarbeiten zu können. Der CCC Compiler wurde NICHT verändert.

Die Anleitung zum Linker CLN und CMake finden Sie daher wie gewohnt in dem entsprechenden Crest Manual[1]. Diese Anleitung ist als Ergänzung zu verstehen und ist wie folgt strukturiert:

- Kapitel 2 beschreibt die Einrichtung des Clang Compilers unter Windows
- Kapitel 3 beschreibt relevante Einschränkungen und Wissenswertes
- Kapitel 4 beinhaltet Tipps für die Migration eines bestehenden Crest Projekts
- Kapitel 5 beschreibt die aktuelle C++ Unterstützung
- Kapitel 6 enthält weiterführende Informationen und Links

Um gegebenenfalls Code unterschiedlich - je nach Einsatz von CCC oder Clang - zu unterscheiden, setzt der Clang Compiler die folgenden Makros/Defines:

1. `__clang__`
2. `__RTOSUH__` und `__rtosuh__`

Somit kann der Kontrollfluss jederzeit gezielt angepasst werden.

1.1 RTOS-UH Application Binary Interface (ABI)

RTOS-UH und das Crest Paket setzen auf ein spezielles Application Binary Interface, d.h. sie definieren wie genau z.B. Funktionsargumente, Strukturen oder auch generell die Register behandelt werden. Damit bereits geschriebene Software oder Libraries sowie das gesamte RTOS-UH mit neuer Software kompatibel bleibt, wurde Clang so angepasst, dass der erzeugte Code der RTOS-UH ABI folgt. Anwender müssen sich so also keine Sorgen machen, dass Code nicht kompatibel sein könnte. Es gibt jedoch Einschränkungen, wenn existierender Code nun mit Clang statt mit Crest übersetzt werden sollen. Kapitel 3 listet die zentralen Limitierungen und Kapitel 4 liefert Tipps für eine erfolgreiche Migration. Tipp: Ein Projekt kann auch gemixt werden, z.B. können spezielle Crest Aufrufe innerhalb einer .c Datei gekapselt werden, während der Rest des Projekts mit Clang übersetzt wird. Der Linker kümmert sich dann um den Rest.

Der Clang Compiler unterstützt modernes C und C++ (Details siehe Clang Compiler in Version

[1] `https://iep.de/de/download/SW/CrestManual.pdf`

22.1.4). Die entsprechenden Standardbibliotheken (Libraries) werden wie gewohnt von IEP zur Verfügung gestellt. Bei einem C Projekt können beispielsweise alle bekannten Standard- sowie Erweiterungsbibliotheken zu einer Clang-übersetzten Anwendung dazu gelinkt werden.

1.2 Mitgelieferte Tools

Die eigentlichen Compiler:

1. `rtclang.exe`: Clang C Compiler
2. `rtclang++.exe`: Clang C++ Compiler

Ein Präprozessor für C und C++ ist bereits in `clang.exe` sowie in `clang++.exe` integriert.

Eigenständiger Präprozessor:

1. `rtclangd.exe`: Standalone Preprocessor

Nützliche Tools für IDEs:

1. `rtclang-apply-replacements.exe`: Tool zum Refactoring
2. `rtclang-check.exe`: Tool zum Code Checking
3. `rtclang-format.exe`: Tool zur Code Formatierung
4. `rtclang-include-fixer.exe`: Tool zur Unterstützung von C/C++ Includes
5. `rtclang-move.exe`: Tool zum Refactoring von Funktionen/Klassen etc.
6. `rtclang-refactor.exe`: Tool zum Refactoring
7. `rtclang-reorder-fields.exe`: Tool zum Refactoring
8. `rtclang-tidy.exe`: Tool zur Prüfung des Codes in der IDE

Nützliche Tools für die Analyse von Clang-erzeugten Binärdateien:

1. `rtllvm-objdump.exe`: Tool zum Analysieren von erzeugten Binärdateien (z.B. Print des Maschinencodes)
2. `rtllvm-readelf.exe`: Tool zum Auslesen von erzeugten ELF-Binärdateien (falls objdump nicht ausreicht)

Die Dokumentation der einzelnen Tools können Sie dabei offiziellen Clang und LLVM Hompage (siehe weiterführende Literatur in Kapitel 6) entnehmen.

2 Setup

Das Clang Toolset wird aktuell nur für Windows zur Verfügung gestellt. Falls Sie Bedarf an einer Linux oder Mac OS Portierung haben, sprechen Sie uns gerne an. Das Crest Paket wird jedoch vorerst weiterhin exklusiv für Windows zur Verfügung stehen. Der Crest Linker CLN und LNK sind zwingend erforderlich, um eine RTOS-UH kompatible Anwendung zu erzeugen.

Das Toolset wird von IEP als .zip zur Verfügung gestellt. Die Zip Datei muss auf dem Rechner entpackt werden. Der Ordner (`clang`) in dem die .exe Dateien liegen, muss in die Umgebungsvariable `PATH` aufgenommen werden.

Testen Sie Ihre Konfiguration, indem Sie ein neues Terminal (z.B. cmd.exe oder Powershell.exe) öffnen und dort rtclang.exe aufrufen. Mit `rtclang -v` können Sie sich den Pfad anzeigen lassen, um sicherzustellen, dass Sie die richtige Clang Version ausführen (falls Sie bereits einen weiteren Clang Compiler installiert haben).

Der Clang Compiler erzeugt RTOS-UH kompatible Binärdateien für PowerPC Systeme. Aufgerufen werden kann der Clang Compiler wie folgt:

```
rtclang.exe INPUT -o OUTPUT -O2 -Wall -target powerpc-iep-rtosuh
```

Eine kurze Erläuterung der Argumente folgt im Anschluss.

Erforderliche Argumente:

1. `-target powerpc-iep-rtosuh` spezifiziert, dass der Clang Compiler Binärdateien für das Zielsystem RTOS-UH und PowerPC erzeugt.

Nützliche Argumente:

1. `-O2` gibt die Optimierungsstufe O2 an. Die genauen Auswirkungen können in der Clang Dokumentation nachgelesen werden. Falls Sie keine Optimierung wünschen, können Sie O0 wählen (Otto Null)
2. `-Wall` aktiviert alle Warnungen
3. `-Wno-invalid-source-encoding` deaktiviert alle Warnungen, die sich auf das Textencoding der Source Quelle beziehen. RTOS-UH setzt das Windows-1252 Encoding voraus, um deutsche Umlaute korrekt darzustellen. Clang warnt bei solchen Quellen vor einem unbekannten Encoding. Diese Warnung kann jedoch ignoriert werden, solange Sie Umlaute nur in Kommentaren oder als Werte in Zeichenketten einsetzen. Dort kopiert Clang nur Binärwerte und es kommt zu keinem Fehler.

Ein Aufruf in einer bestehenden Cmake Datei wie z.B.

```
cccppc main.c main.obj
```

kann also wie folgt geändert werden:

```
rtclang main.c -o main.obj -target powerpc-iep-rtosuh
```

Die Crest Linker LNK oder CLN erkennen dann automatisch, ob es sich um eine Crest- oder Clang-erzeugte Binärdatei handelt und reagiert entsprechend. Ein Einsatz dieser Linker ist weiterhin zwingend erforderlich. Bei der Migration eines Projekts sind jedoch einige Einschrän-

kungen zu beachten, die in Kapitel 3 beschrieben werden. Ein Guide sowie hilfreiche Tipps für die Migration von einem Crest CCC Projekt finden Sie in Kapitel 4.

2.1 Neue Clang-builtins Library

Für einige Projekte ist ggf. der Einsatz spezieller builtin Funktionen von Clang notwendig. Diese Funktionen umfassen z.B. spezielle Shift oder Division Operatoren. Die Library `clang_builtins.lib` enthält genau diese Clang-spezifische builtin Ausdrücke und kann bei Bedarf gelinkt werden. Sie befindet sich im `clib-ppc` Ordner und ist unseren Linkern automatisch bekannt. Sie brauchen diese Library jedoch nur in seltenen Fällen (z.B. wenn Ihr Code hardware-spezifische Ausdrücke benötigt wie beispielsweise die mbedTLS 4.0 Library).

3 Einschränkungen und Wissenswertes

Der Clang Compiler verfügt im Vergleich zum Crest Compiler über die folgenden Einschränkungen, die im folgenden erläutert werden:

1. Keine direkte Pearl Unterstützung (Abschnitt 3.1)
2. Keine direkte Erzeugung von RTOS-UH Tasks (Abschnitt 3.2)
3. Ungewollte Optimierungen von Ausdrücken (Abschnitt 3.3)
4. Keine Unterstützung für Debugging (Abschnitt 3.6)
5. Keine direkte Unterstützung für spezifische FPUs (Abschnitt 3.7)

Zudem muss das Schlüsselwort `local` für Clang bekannt gemacht werden (Abschnitt 3.5) und das Padding sowie das Aligment von Strukturen kann sich zwischen Clang und Crest unterscheiden (Abschnitt 3.4).

3.1 Pearl Unterstützung

Von Clang übersetzte Codedateien können keine Pearl Funktionen aufrufen. Allerdings kann ein Projekt wie folgt strukturiert werden: Erstellen Sie eine `PearlWrapper.c` und eine `PearlWrappher.h`. Alle benötigten Pearl Aufrufe können Sie nun im `PearlWrapper.c` durch geeignete C Funktionen kapseln.

`PearlWrapper.c`:

```
pearl void SETPIX(int16_t xpos, int16_t ypos, color_t color);

void pearl_SETPIX(int16_t xpos, int16_t ypos, color_t color)
{
    SETPIX(xpos, ypos, color);
}
```

`PearlWrappher.h`:

```
void pearl_SETPIX(int16_t xpos, int16_t ypos, color_t color);
```

Sourcedateien, die mit Clang übersetzt werden sollen, können nun die C Funktion aus dem Pearlwrapper aufrufen. Wichtig ist nun, dass Sie den PearlWrapper weiterhin mit dem Crest Compiler übersetzen. Mit Hilfe dieses Tricks können Sie nun weiterhin Pearl Funktionen aufrufen.

3.2 Tasking

Mit Crest war es möglich, Tasks in C wie folgt zu definieren:

```
void read_touch_i2c()
{
    printf("Hallo World\n");
}

#pragma TASK NO_TASKSTART USE_FUNCTIONNAME
Task *TASK_READ_TOUCH_I2C(void)
{
    read_touch_i2c();
}

void start_read_touch_i2c_task(void)
{
    Task *i2c_task = TASK_READ_TOUCH_I2C();
    rt_enable_event(GPIO1_MASK);
    rt_event_activate_quick(i2c_task, TASK_PRIORITY_TOUCH_I2C, GPIO1_MASK);
}
```

Der Clang Compiler unterstützt solche #pragma Ausdrücke nicht und kann deshalb keine geeigneten Instruktionen erzeugen. Sie können jedoch einen ähnlichen Trick wie bei der Unterstützung von Pearl Funktionen anwenden. Bewegen Sie das Erzeugen aller Tasks in eine extra dafür vorgesehene Datei wie z.B. `Tasks.c` und `Tasks.h`.

`Tasks.c`:

```
#pragma TASK NO_TASKSTART USE_FUNCTIONNAME
Task *TASK_READ_TOUCH_I2C(void)
{
    read_touch_i2c();
}

void start_read_touch_i2c_task(void)
{
    Task *i2c_task = TASK_READ_TOUCH_I2C();
    rt_enable_event(GPIO1_MASK);
    rt_event_activate_quick(i2c_task, TASK_PRIORITY_TOUCH_I2C, GPIO1_MASK);
}
```

`Tasks.h`:

```
void start_read_touch_i2c_task(void);
```

Die eigentliche Funktion `read_touch_i2c()` kann in einer Datei definiert und deklariert werden, die von Clang übersetzt wird. Mit Hilfe dieses Tricks können also weiterhin Tasks erzeugt werden. Wichtig ist, dass die Datei `Tasks.c` weiterhin mit Crest übersetzt wird.

3.3 Ungewollte Optimierungen (`volatile`)

Der Clang Compiler verfügt über zahlreiche Optimierungsstrategien (besonders wenn Sie Code mit `O2` übersetzen). In folgendem Codebeispiel wird die Addresse einer Funktion in die Vector Table eingetragen:

```
void I2C_irq( void )
{
    printf("Hello World\n");
}

void initialize_touch_interrupt_i2c(void)
{
    /* setup own Vector in Table */
    *(ULONG *) 0x00004258 = (ULONG)(void *)I2C_irq;
}
```

Ein solcher Aufruf wird ggf. von Clang vollständig entfernt, da der Ausdruck

```
*(ULONG *) 0x00004258 = (ULONG)(void *)I2C_irq;
```

keinen Einfluss auf den Programmfluss hat. Die Adresse `0x00004258` liegt außerhalb der von Clang verfolgten Variablen. Ein solcher Ausdruck wird in einer Optimierung einfach entfernt. Der C Standard sieht nämlich vor, dass Statements ohne observierbaren Nebeneffekt als optional angesehen werden dürfen. Hier greift dann eine as-if Regel, die ein solches Statement einfach entfernt, da er scheinbar keinen Einfluss auf das Programmverhalten hat.

Der C Standard sieht als Lösung des Problems das Schlüsselwort `volatile` vor. Ändern Sie obigen Ausdruck wie folgt:

```
void I2C_irq( void )
{
    printf("Hello World\n");
}

void initialize_touch_interrupt_i2c(void)
{
    /* setup own Vector in Table */
    *(volatile ULONG *)0x00004258 = (ULONG)(void *)I2C_irq;
}
```

Nun erkennt der Clang Compiler, dass er die Zuweisung nicht entfernen darf, da sie mit `volatile` gekennzeichnet wurde. Das Schlüsselwort `volatile` signalisiert dem Compiler, dass die Variable nicht gecached werden darf. Das heißt, dass jedes Lesen (Read), jedes Schreiben (Write) oder jeder Zugriff explizit ausgeführt werden muss. Daher gibt es in diesem Fall einen observierbaren Nebeneffekt und der Code darf nicht wegoptimiert werden.

3.4 Alignment und Padding von Strukturen

Beachten Sie, dass sich das Alignment und Padding von Strukturen zwischen Clang und Crest unterscheiden kann. Wenn Sie die Struktur nur jeweils im Clang oder Crest Teil exklusiv nutzen wollen, so entstehen keine Probleme. Wollen Sie die gleiche Struktur jedoch aus beiden Teilen nutzen, so stellen Sie das gleiche Alignment und Padding sicher. Wenn Sie kein Padding explizit angegeben haben, so ist das Clang und Crest Alignment und Padding kompatibel.

Wenn Sie sich jedoch beispielsweise im Crest auf ein 68K Padding festgelegt haben, so müssen Sie diese Entscheidung auch Clang mitteilen. Die folgenden Ausdrücke sind für das PowerPC System gleichwertig:

```
#ifdef __clang__
#pragma pack(push, 1)
#else
#pragma MEMBER_PADDING_68K
#pragma STRUCT_SIZE_WORD
#endif
typedef struct tMACADR
{
    unsigned char ad[6];
}
tMACADR;

typedef struct tGETPHY
{
    unsigned short cmd;
    unsigned short automatic;
    unsigned short speed;
    unsigned short duplex;
    unsigned short link;
}
tGETPHY;

#ifdef __clang__
#pragma pack(pop)
#else
#pragma MEMBER_PADDING_PPC
#pragma STRUCT_SIZE_LONG
#endif
```

3.5 Local

Das Schlüsselwort `local` ist Clang nicht direkt bekannt. Jedoch definiert `local` nur, wo die Variable später im Speicher liegt. Wenn Sie eine Deklaration wie folgt durchführen möchten:

```
extern local FILE  *stdin  ;
extern local FILE  *stdout ;
extern local FILE  *stderr ;
```

Dann können Sie entweder das Attribut `.local` für Clang verwenden:

```
__attribute__((section(".local")))
extern FILE* stdin;
__attribute__((section(".local")))
extern FILE* stdout;
__attribute__((section(".local")))
extern FILE* stderr;
__attribute__((section(".local")))
extern int    errno;
```

oder alternativ das Schlüsselwort `local` wie folgt definieren:

```
#define local __attribute__((section(".local")))
```

Clang wurde um eine Unterstützung angepasst, `local` Variablen richtig zu adressieren, sodass zur Laufzeit keine Probleme auftreten sollten. Der Rest ist Aufgabe des Linkers und des Loaders.

3.6 Debugging

Zum aktuellen Zeitpunkt unterstützten wir kein Debugging von Clang-erzeugten Objectfiles. Sie können also ausschließlich Code Debuggen, der mit Crest erzeugt wurde. Mit Hilfe der Tools aus der Clang Suite können Sie sich jedoch z.B. die Maschineninstruktionen anzeigen lassen (via `llvm-objdump.exe`).

3.7 FPU (Floating Point Units)

Aktuell unterstützten wir keine Codeerzeugung für FPUs mit dem Clang Compiler. Floats und Doubles werden mit dem Clang in Software berechnet. Wenn Sie die Nutzung einer FPU wünschen, so müssen Sie entsprechende C Funktionen schreiben und in einer Datei kapseln, die sie weiterhin mit Crest übersetzen; ähnlich zu Pearl und dem Tasking. Die Funktionen können dann aus dem Code, der mit Clang übersetzt wird, aufgerufen werden.

4 Migration eines Crest Projekts

Für die Migration eines Crest Projekts befolgen Sie die Hinweise aus Kapitel 3. Folgende Schritte sollten Sie beachten:

1. **Pearl:** Wenn Sie Pearl Funktionen in ihrem C Code aufrufen wollen, so kapseln Sie jegliche Pearl Funktionen durch geeignete C Funktionen in einem `PearlWrapper.c` und `PearlWrappher.h` (siehe Abschnitt 3.1).
2. **Tasking:** Wollen Sie Tasking verwenden, so reicht es nur die Erzeugung der Tasks in einer `Tasks.c` und `Tasks.h` Datei zu kapseln (siehe Abschnitt 3.2).
3. **Volatile:** Denken Sie an folgende Faustregel: Wollen Sie eine Speicheraddresse irgendwo im Speicher einfach beschreiben, so kennzeichnen Sie diese Ausdrücke mit dem Schlüsselwort `volatile`, sodass Clang die Ausdrücke nicht optimiert (siehe Abschnitt 3.3).
4. **local:** Möchten Sie das Schlüsselwort `local` verwenden, so müssen Sie es vorher für Clang definieren (als explizites Sectionattribute oder als Define; siehe Abschnitt 3.5)
5. **Structs:** Beachten Sie, dass ihr Alignment und Padding von Strukturen zwischen Crest und Clang Dateien kompatibel ist (siehe Abschnitt 3.4).

Ein geeignete Cmake Datei könnte also wie folgt aussehen:

```
NAME = Test
PTH=SP$(S)

CLANGOPTIONS = -O2 -Wall -target powerpc-iep-rtosuh
    -Wno-invalid-source-encoding
COPTIONS  = -L -P=2 -x --fpu -D=1 -C=1 -q
LOPTIONS  = -M -P=2 -N=$(Name) --fpu -#PTH=$(PTH)

$(PTH)$(NAME).SR:            \
    $(PTH)main.obj           \
    $(PTH)core.obj           \
    $(PTH)network.obj        \
    $(PTH)IO.obj             \
    $(PTH)Tasks.obj          \
    $(PTH)PearlWrapper.obj   \

   clnppc Test.dol $(PTH)$(NAME).SR $(LOPTIONS)

$(PTH)*.obj : *.c

   rtclang *.c -o $(PTH)*.obj  $(CLANGOPTIONS)

$(PTH)PearlWrapper.obj : PearlWrapper.c

   cccppc PearlWrapper.c $(PTH)PearlWrapper.obj $(COPTIONS)

$(PTH)Tasks.obj : Tasks.c

   cccppc Tasks.c $(PTH)Tasks.obj $(COPTIONS)
```

Das entsprechende Linkfile `Test.dol` könnte wie folgt aussehen:

```
sstart.obj
$(PTH)main.obj
$(PTH)core.obj
$(PTH)network.obj
$(PTH)IO.obj
$(PTH)Tasks.obj
$(PTH)PearlWrapper.obj
fpupfast.lib
```

In diesem Beispiel wird `Tasks.c` und `PearlWrapper.c` mit Crest übersetzt. Das restliche Projekt mit Clang. Mit Hilfe dieses Tricks können Sie also bestehende und Crest-spezifische Ausdrücke weiterhin mit Crest übersetzen und in einer Applikation nutzen, die sonst größtenteils mit Clang übersetzt wird. Sie Namen `Tasks` und `PealWrapper` sind natürlich frei gewählt. Sie können diese Namen beliebig anpassen oder beliebig viele weitere Dateien mit Crest übersetzen.

Denken Sie daran, dass der Clang Compiler folgende Defines setzt:

1. `__clang__`
2. `__RTOSUH__`

So können Sie jederzeit kontrollieren, welche Deklaration von welchem Compiler gesehen wird. Beispielsweise so:

```
#if defined ( __clang__ )
__attribute__((section(".local")))
extern FILE* stdin;
__attribute__((section(".local")))
extern FILE* stdout;
__attribute__((section(".local")))
extern FILE* stderr;

__attribute__((section(".local")))
extern int    errno;

#else
extern local FILE  *stdin  ;
extern local FILE  *stdout ;
extern local FILE  *stderr ;

extern local int    errno  ;
#endif
```

5 Unterstützung von C++ (Experimentell)

Achtung: Die C++ Unterstützung befindet sich derzeit noch in der Entwicklung und ist als experimentell anzusehen.

Mit Clang folgt nun auch eine C++ Unterstützung für RTOS-UH. Im ausgelieferten Paket befindet sich der C++ Compiler `rtclang++`, der über die Kommandozeile aufgerufen werden kann.

Unsere C++ Unterstützung umfasst dabei:

1. Den Clang Compiler in Version 22.1.4. Eine Unterstützung der C++ Features können Sie unter folgendem Link finden[2].
2. Eine minimale C++ Runtime für RTOS-UH. Für leichtgewichtige Projekte, ohne Abhängigkeiten zur C oder C++ Standard Library.
3. Eine Portierung der Open Source C++ Standard Library libcxx[3] in Version 22[4] für RTOS-UH. Es werden jedoch nicht sämtliche Funktionen unterstützt. Ausgenommen sind aktuell Threading, Tasks und Prozesse. Diese müssen weiterhin über einen Crest C Wrapper abgebildet werden.

Wichtige Einschränkungen:

1. Es wird aktuell kein Exception Handlung für C++ auf PowerPC unterstützt. Der Clang Compiler übersetzt also try/catch Ausdrücke nicht und gibt einen Fehler aus . Dazu muss das Compiler Flag `-fno-exceptions` gesetzt werden.

Um die C++ Unterstützung vollständig zu nutzen, müssen Sie die den Clang Compiler wie in Schritt 2 einrichten. Wichtig ist, dass die RTOS UH C++ Libraries `cxxstd.lib`, `cxxabi.lib`, `clang_builtins.lib` und `cxx_runtime_support.cpp.obj` in den Ordner `clib-ppc` des Crest Pakets aufgenommen werden. Damit werden Sie unserem Linker `cln` bekannt gemacht und stehen von nun an zur Verfügung. Alternativ können Sie natürlich die Libraries auch direkt in ein Projekt übernehmen.

1. `cxx_runtime_support.cpp.obj` enthält ein minimales Objekt (ohne jegliche Standard Library), dass die Unterstützung von C++ auf RTOS-UH ermöglicht.
2. `clang_builtins.lib` enthält Clang-spezifische builtin Ausdrücke.
3. `cxxabi.lib` umfasst ein vollständiges Application Binary Interface. Diese Library enthält die `clang_builtins.lib` und `cxx_runtime_support.cpp.obj`.
4. `cxxstd.lib` umfasst die libcxx Library. Diese Library enthält die Libraries `cxxabi.lib`, `clang_builtins.lib` und `cxx_runtime_support.cpp.obj`.

Möchten Sie also nur eine C++ Unterstützung, so nehmen Sie die `cxxabi.lib` Library in Ihr

[2] `https://releases.llvm.org/22.1.0/tools/clang/docs/ReleaseNotes.html`
[3] `https://libcxx.llvm.org/`
[4] `https://releases.llvm.org/22.1.0/projects/libcxx/docs/ReleaseNotes.html`

Linkfile mit auf. Möchten Sie eine C++ Standard Library und ein Interface zur C Standard Library, so nehmen Sie die `cxxstd.lib` Library in Ihr Linkfile mit auf. Möchten Sie C und C++ Code mischen, so achten sie darauf, dass sie alle notwendigen Bibliotheken einbinden.

5.1 Beispiele

Simpler Aufruf des C++ Compilers über die Kommandozeile:

```
rtclang++ main.cpp -o main.o -target powerpc-iep-rtosuh -fno-exceptions
```

Simples CMake File:

```
PTH=SP$(S)
COPTIONS = -O2 -Wall -target powerpc-iep-rtosuh -fno-exceptions

$(PTH)CxxTest.SR:            \
        $(PTH)main.o         \

    clnppc main.link $(PTH)CxxTest.SR -M -P=2 -N=CxxTest -#PTH=$(PTH)

$(PTH)*.o : *.cpp

   rtclang++ *.cpp -o $(PTH)*.o $(COPTIONS)
```

Dazugehöriges Linkfile `main.link`:

```
sstart.obj
cxxstd.lib
$(PTH)main.o
```

6 Weiterführende Information

1. Clang Manual: `https://clang.llvm.org/docs/UsersManual.html`
2. Crest Manual: `https://iep.de/de/download/SW/CrestManual.pdf`