SILworX

C++-Funktionsbaustein
Handbuch

SAFETY
NONSTOP

HIMA

Alle in diesem Handbuch genannten HIMA Produkte sind mit dem Warenzeichen geschützt. Dies gilt ebenfalls, soweit nicht anders vermerkt, für weitere genannte Hersteller und deren Produkte.

Alle technischen Angaben und Hinweise in diesem Handbuch wurden mit größter Sorgfalt erarbeitet und unter Einschaltung wirksamer Kontrollmaßnahmen zusammengestellt. Bei Fragen bitte direkt an HIMA wenden. Für Anregungen, z. B. welche Informationen noch in das Handbuch aufgenommen werden sollen, ist HIMA dankbar.

Technische Änderungen vorbehalten. Ferner behält sich HIMA vor, Aktualisierungen des schriftlichen Materials ohne vorherige Ankündigungen vorzunehmen.

Weitere Informationen sind in der Dokumentation auf der HIMA DVD und auf unserer Webseite unter `http://www.hima.de` und `http://www.hima.com` zu finden.

Kontakt

HIMA Adresse:

HIMA Paul Hildebrandt GmbH

Postfach 1261

68777 Brühl

Tel.: +49 6202 709-0

Fax: +49 6202 709-107

E-Mail: info@hima.com

Revisions-index	Änderungen	Art der Änderung	
		technisch	redaktionell
5.00	Erste Ausgabe	X	X
5.01	Kapitel 3.2 Erforderliche Prozessorbetriebssysteme	X	X
5.02	Kapitel 4.3.1 Stack-Bedarf	X	X

Inhaltsverzeichnis

1 Einleitung 5

1.1 Aufbau des Handbuchs 5

1.2 Zielgruppe 5

1.3 Darstellungskonventionen 5

1.3.1 Sicherheitshinweise 6

1.3.2 Gebrauchshinweise 6

2 Sicherheit 7

2.1 Bestimmungsgemäßer Einsatz 7

3 Produktbeschreibung 8

3.1 Lizenz 8

3.2 Erforderliche HIMax/HIMatrix Betriebssystemversionen 8

3.3 Überblick und Handhabung von C++-Funktionsbausteinen 8

3.3.1 Umgang mit C++-Funktionsbausteinen 8

3.3.2 Schnittstelle und Umgang mit Variablen 8

3.3.3 Hinweise und Einschränkungen 9

3.4 Prinzipieller Ablauf zur Erstellung von C++-Funktionsbausteinen 9

4 Konfiguration 10

4.1 C++-Funktionsbausteine in SILworX anlegen 10

4.2 Dialog Eigenschaften des C++-Funktionsbausteins 10

4.3 Editor der C++-Funktionsbausteine 13

4.3.1 Stack-Bedarf 13

4.3.1.1 Zusätzlichen Stack-Bedarf ermitteln und einstellen 13

4.3.2 Lokale Variablen im POE-Editor 14

4.3.3 Quelldateien 14

4.3.4 Export von Quelldateien aus einem C++-Funktionsbaustein 14

4.3.5 Import von Quelldateien in einen C++-Funktionsbaustein 14

5 C++-Quellcode erstellen 15

5.1 Einbindung des C++-Codes 15

5.1.1 Schnittstellen der C++-Funktionsbausteine für Funktionsbausteinsprache 15

5.1.2 C++-Funktionsbaustein in Funktionsbausteinsprache verwenden 15

5.1.3 Code generieren 15

5.2 Prüfung und Diagnose 15

5.3 C++-Sprachumfang 16

5.3.1 Allgemeine C++-Sprachelemente 16

5.3.2 Namensräume 16

5.3.3 Globale C++-Variablen 17

5.3.4 C++-Zeiger 17

5.3.5 Includes 18

5.3.6 Datentypen 19

5.3.7 Funktionen und Operatoren 19

5.3.7.1 Logische Operatoren 19

5.3.7.2 Bitweise Operatoren 20

5.3.7.3 Vergleichsoperatoren 20

5.3.7.4 Cast-Operatoren 20

5.3.7.5 Arithmetische Operatoren 21
5.3.7.6 Numerische Funktionen 21
5.4 Dateiformat und erlaubte Zeichen 22
5.5 Regeln für Dateinamen 22
5.6 Nicht erlaubte C++-Sprachelemente 23
5.6.1 Makros 23
5.6.2 Include 23
5.6.3 Datentypen 23
5.6.4 Funktionen 23
5.6.5 Variablen 23
5.6.6 Literale 24
5.6.7 Operatoren 24
5.6.8 Sonstige Syntax 24
5.6.9 Verwendung von globalen SILworX-Variablen 24
Anhang 25
Tabellenverzeichnis 25

1 Einleitung

Das vorliegende Handbuch beschreibt den Gebrauch und die Programmierung der C++-Funktionsbausteine. Das Handbuch enthält Informationen über die C++-Sprachelemente und die Konfiguration in SILworX.

1.1 Aufbau des Handbuchs

Das Handbuch ist in folgende Hauptkapitel gegliedert:

- Einleitung
- Sicherheit
- Produktbeschreibung
- Inbetriebnahme
- Betrieb
- Anhang

1.2 Zielgruppe

Dieses Dokument wendet sich an Planer, Projekteure und Programmierer von Automatisierungsanlagen sowie Personen, die zu Inbetriebnahme, Betrieb und Wartung der Geräte und Systeme berechtigt sind. Vorausgesetzt werden Kenntnisse der Programmierung in C++.

1.3 Darstellungskonventionen

Zur besseren Lesbarkeit und zur Verdeutlichung gelten in diesem Dokument folgende Schreibweisen:

Fett	Hervorhebung wichtiger Textteile. Bezeichnungen von Schaltflächen, Menüpunkten und Registern in SILworX, die angeklickt werden können
Kursiv	Systemparameter und Variablen
`Courier`	Wörtliche Benutzereingaben
RUN	Bezeichnungen von Betriebszuständen in Großbuchstaben
Kap. 1.2.3	Querverweise sind Hyperlinks, auch wenn sie nicht besonders gekennzeichnet sind. Wird der Mauszeiger darauf positioniert, verändert er seine Gestalt. Bei einem Klick springt das Dokument zur betreffenden Stelle.

Sicherheits- und Gebrauchshinweise sind besonders gekennzeichnet.

1.3.1 Sicherheitshinweise

Die Sicherheitshinweise im Dokument sind wie folgend beschrieben dargestellt.
Um ein möglichst geringes Risiko zu gewährleisten, sind sie unbedingt zu befolgen. Der inhaltliche Aufbau ist

- Signalwort: Warnung, Vorsicht, Hinweis
- Art und Quelle des Risikos
- Folgen bei Nichtbeachtung
- Vermeidung des Risikos

⚠ SIGNALWORT

Art und Quelle des Risikos!
Folgen bei Nichtbeachtung
Vermeidung des Risikos

Die Bedeutung der Signalworte ist

- Warnung: Bei Missachtung droht schwere Körperverletzung bis Tod
- Vorsicht: Bei Missachtung droht leichte Körperverletzung
- Hinweis: Bei Missachtung droht Sachschaden

HINWEIS

Art und Quelle des Schadens!
Vermeidung des Schadens

1.3.2 Gebrauchshinweise

Zusatzinformationen sind nach folgendem Beispiel aufgebaut:

i An dieser Stelle steht der Text der Zusatzinformation.

Nützliche Tipps und Tricks erscheinen in der Form:

TIPP An dieser Stelle steht der Text des Tipps.

2 Sicherheit

Sicherheitsinformationen, Hinweise und Anweisungen in diesem Dokument unbedingt lesen.

Der Anwender darf nicht alle Möglichkeiten nutzen, die ihm die C++-Funktionsbausteine technisch bedingt bieten, siehe Kapitel 3.3.3.

Dieses Dokument beschreibt, welche Regeln der Anwender bei der Erstellung eines C++-Funktionsbaustein einhalten muss, welche Sprachelemente von C++ er dafür nutzen darf, und welche Verantwortung ihm in dieser Hinsicht obliegt.

Die C++-Funktionsbausteine nur unter Beachtung aller Richtlinien und Sicherheitsrichtlinien einsetzen.

2.1 Bestimmungsgemäßer Einsatz

C++-Funktionsbausteine dürfen in Absprache mit der zuständigen Abnahmebehörde für sicherheitsgerichteten Betrieb eingesetzt werden!

⚠ WARNUNG

Der Anwender ist selbst für die Sicherheit des Anwenderprogramms gemäß IEC 61508 verantwortlich!

Ein Fehler in der Funktion eines C++-Funktionsbausteins kann sich unmittelbar auf jeden beliebigen Teil des aufrufenden Anwenderprogramms auswirken. Auswirkungen auf parallel ablaufende Anwenderprogramme oder andere Systemteile wie z. B. Hardware oder Protokolle sind nur indirekt über globale Variablen möglich.

3 Produktbeschreibung

Die C++-Funktionsbausteine bieten dem Anwender die Möglichkeit, eigene C++-Programme in die Funktionsbausteinsprache von SILworX (V.5.34 und höher) einzubinden. Dabei dienen die Eingangsvariablen und Ausgangsvariablen der C++-Funktionsbausteine als Schnittstelle zu den Funktionen des eingebundenen C++-Programms. Ein C++-Funktionsbaustein kann wie ein normaler Funktionsbaustein in SILworX verwendet werden.

Die C++-Funktionsbausteine ermöglichen dem Anwender, flexibler zu programmieren und eine höhere Performance zu erzielen als mit der Funktionsbausteinsprache. Zudem kann ein C++-Quellcode aus Fremdtools (z. B. für Mathematik- und Simulationssoftware) mithilfe der C++-Funktionsbausteine in die Funktionsbausteinsprache von SILworX eingebunden werden.

3.1 Lizenz

Für das Erstellen und Ändern von C++-Funktionsbausteinen wird ein Freischaltcode mit der entsprechenden Lizenz-Option benötigt. Die Verwendung von Archiven und Projekten mit fertigen C++-Funktionsbausteinen ist auch ohne diese Lizenz-Option möglich. Der Freischaltcode ist in SILworX in der Lizenzverwaltung einzugeben.

3.2 Erforderliche HIMax/HIMatrix Betriebssystemversionen

C++-Funktionsbausteine sind nur auf Steuerungen mit dem passenden Prozessorbetriebssystem einsetzbar, siehe nachfolgende Tabelle:

System	Prozessorbetriebssystem
HIMax	ab CPU BS V4.14
HIMatrix Layout 2	ab CPU BS V8.10
HIMatrix Layout 3	ab CPU BS V8.14

Tabelle 1: Erforderliche Prozessorbetriebssysteme

3.3 Überblick und Handhabung von C++-Funktionsbausteinen

Diese Kapitel gibt einen Überblick über die Besonderheiten und den Umgang mit C++-Funktionsbausteinen.

3.3.1 Umgang mit C++-Funktionsbausteinen

- C++-Funktionsbausteine können mit SILworX erstellt werden.
- C++-Funktionsbausteine können im Anwenderprogramm und in Funktionsbausteinen verwendet werden.
- Online-Änderungen (Reload) eines Programms mit C++-Funktionsbausteinen sind möglich.
- Die SILworX-Funktionen wie Archivieren, Wiederherstellen, Kopieren, Einfügen, Löschen, Drucken oder Aktivieren des Schreib- oder Know-how-Schutzes für C++-Funktionsbausteine können verwendet werden.
- Die C++-Funktionsbausteine können in SILworX nur als Blackbox über die Eingangs- und Ausgangswerte getestet werden, siehe Kapitel 5.2.
- Anwender können Archive importieren, die C++-Funktionsbausteine enthalten.
- Anwender können C++-Funktionsbausteine löschen.

3.3.2 Schnittstelle und Umgang mit Variablen

- Die Variablentypen VAR_Input, VAR_Output und VAR werden für die Schnittstelle des C++-Funktionsbausteins verwendet.
- Der Wert der Variablen vom Typ VAR und VAR_Output bleibt von einem Anwenderprogrammzyklus bis zum nächsten erhalten.

- Nur elementare Datentypen (keine Arrays, Strukturen oder Zeiger) sind für VAR_Input, VAR_Output und VAR zulässig.
- Der C++-Bezeichner für eine Variable setzt sich aus einem Präfix und dem in SILworX eingegebenen Variablennamen zusammen. Wenn der Variablenname Zeichen enthält, die ein C++-Bezeichner nicht enthalten darf, werden diese ersetzt. Unterstriche ebenso. Der Ersatz besteht aus einem Unterstrich gefolgt von ein bis drei Buchstaben und Ziffern.
- Zusätzliche C++-Quellcode- und Header-Dateien können hinzugefügt werden. Die definierten Elemente in diesen Dateien können innerhalb der C++-Funktionsbausteine verwendet werden.

3.3.3 Hinweise und Einschränkungen

Es folgt eine Übersicht der wichtigsten Einschränkungen:

- Die Verwendung von globalen SILworX-Variablen in C++-Funktionsbausteinen ist nicht zulässig.
- Die Verwendung von SILworX-Funktionsbausteinen in C++-Funktionsbausteinen ist nicht zulässig.
- Online-Test von C++-Quellcode ist nicht möglich.
- Offline-Simulation von C++-Quellcode ist nicht möglich.
- Die Klassenstruktur der C++-Funktionsbausteine kann nur mit der SILworX-Interface-Deklaration von VAR_Input, VAR_Output, VAR geändert werden.

3.4 Prinzipieller Ablauf zur Erstellung von C++-Funktionsbausteinen

Die folgenden Schritte zeigen den prinzipiellen Ablauf zur Erstellung eines C++-Funktionsbausteins in SILworX.

1. C++-Funktionsbaustein in SILworX anlegen und Eigenschaften festlegen, siehe Kapitel 4.1 und Kapitel 4.2.
2. C++-Funktionsbaustein im Editor konfigurieren. Hier werden die Eingangs- und Ausgangsvariablen des C++-Funktionsbausteins und der benötigte Stack-Bedarf festgelegt, Siehe Kapitel 4.3.
3. C++-Quelldatei-Funktionsrumpf aus C++-Funktionsbaustein exportieren, siehe Kapitel 4.3.4.
4. C++-Quelldatei-Funktionsrumpf mit eigenen Funktionen ergänzen und bearbeiten, siehe Kapitel 5.1.
5. Ergänzte C++-Quelldatei-Funktionsrümpfe und eventuell zusätzliche C++-Quelldateien in den C++-Funktionsbaustein importieren, siehe Kapitel 4.3.5.

4 Konfiguration

In diesem Kapitel werden die Konfiguration der C++-Funktionsbausteine und die Funktionen des C++-Funktionsbaustein-Editors beschrieben.

4.1 C++-Funktionsbausteine in SILworX anlegen

Ein C++-Funktionsbaustein kann folgendermaßen in eine Funktionsbaustein-Bibliothek eingefügt werden.

Ist bereits eine Bibliothek vorhanden, kann auch diese für die C++-Funktionsbausteine verwendet werden.

Eine Bibliothek für die C++-Funktionsbausteine anlegen

1. Die Konfiguration selektieren und aus Kontextmenü oder Aktionsleiste **Neu** auswählen.
   ☑ Das Dialogfenster *Neues Objekt* öffnet sich.
2. Im Dialogfenster **Bibliothek** anklicken.
3. Ins Feld *Name* den Namen der neuen Bibliothek eintragen.
4. **OK** klicken.

Eine neue Bibliothek für die C++-Funktionsbausteine ist unter der Konfiguration angelegt.

Einen C++-Funktionsbaustein anlegen

1. Die neue Bibliothek selektieren und aus Kontextmenü oder Aktionsleiste **Neu** auswählen.
   ☑ Das Dialogfenster *Neues Objekt* öffnet sich.
2. Im Dialogfenster **C++-Funktionsbaustein-Typ** anklicken.
3. Ins Feld *Name* den Namen des neuen C++-Funktionsbaustein-Typs eintragen.
4. **OK** klicken.

Ein neuer C++-Funktionsbaustein-Typ ist unter der Bibliothek angelegt.

4.2 Dialog Eigenschaften des C++-Funktionsbausteins

Der Eigenschaften-Dialog des C++-Funktionsbausteins zeigt die Eigenschaften, die allen POE-Typen gemeinsam sind. Die spezifischen Eigenschaften des C++-Funktionsbausteins befinden sich im C++-Funktionsbaustein-Editor.

Parameter	Beschreibung	Veränderbar
Name	Name eines Funktionsbaustein-Typs, wie er im Strukturbaum, im Zeichenfeld des Programm-Editors und im Register Bausteine der Objektauswahl angezeigt wird. Pro Funktionsbaustein-Typ kann nur ein Name vergeben werden. Der Name erscheint innerhalb des grafischen Objekts. In der Schnittstellenanzeige kann der Name eines Funktionsbaustein-Typs auch geändert werden. Doppelklick auf den Namen im grafischen Objekt und einen neuen Namen eingeben. Beenden der Eingabe mit der ENTER-Taste.	Ja
Ausziehbar	Diesen Parameter aktivieren, wenn der Funktionsbaustein ausziehbar sein soll.	Ja
Minimale Ausziehbarkeit [Pins]	Dieser Parameter definiert die Anzahl von Eingängen, die im Zeichenfeld mindestens angezeigt werden. Dies entspricht der Standardgröße.	Ja, wenn Ausziehbar aktiviert ist.

Maximale Ausziehbarkeit [Pins]	Dieser Parameter definiert die Anzahl von Eingängen, die im Zeichenfeld höchstens angezeigt werden. Die Maximale Ausziehbarkeit ist abhängig von der Anzahl der mit VAR_INPUT definierten Eingänge. Mit der Funktion *Ausziehen* kann eine POE im Zeichenfeld bis zur maximalen Ausziehbarkeit vergrößert werden.	Ja, wenn Ausziehbar aktiviert ist.
Standard-Instanzname	Instanzname eines Funktionsbaustein-Typs, wie er im Zeichenfeld des Programm-Editors angezeigt wird. Für jede Instanz eines Funktionsbaustein-Typs kann ein eigener Instanzname vergeben werden. Anderenfalls wird der Standard-Instanzname, gefolgt von einer laufenden Nummer «_n» angezeigt. Der Instanzname erscheint außerhalb des grafischen Objekts, am oberen Rand. Der Standard-Instanzname kann in der Schnittstellenanzeige nicht geändert werden.	Ja
Standardbreite	Die Standardbreite definiert die Breite des Funktionsbaustein-Typs in Gittereinheiten des Zeichenfelds. Der Standardwert ist 10. Dies bedeutet, dass das Objekt im Zeichenfeld zehn Gittereinheiten breit ist. Es ist zu beachten, dass die Höhe des Objekts von SILworX vorgegeben wird. Standardwert: 10	Ja
Variante	**Normal**: Die benutzerdefinierte POE ist offen und Dritte haben uneingeschränkten Zugriff auf die C++-Quelldateien. Dies ist die Standardeinstellung. **Read-Only**: Mit der Einstellung Read-Only gelten die folgenden Einschränkungen gegenüber einem offenen C++-Funktionsbaustein: ▪ Der Editor eines Read-Only C++-Funktionsbausteins kann geöffnet, jedoch nicht verändert werden. Im Editor sind alle Editier- und Änderungsmöglichkeiten gesperrt. ▪ Der Eigenschaften-Dialog eines Read-Only C++-Funktionsbausteins kann geöffnet, jedoch nicht verändert werden. Read-Only ist nicht reversibel und kann auch nicht durch Kopieren/Einfügen und Archivieren/Wiederherstellen umgangen werden. Bevor SILworX einen C++-Funktionsbaustein verschließt, muss die Sicherheitsabfrage mit *Ja* beantwortet werden. i In jedem Fall darauf achten, dass ein nicht verschlossenes Backup des C++-Funktionsbausteins verfügbar ist! Der Codegenerator erzeugt für einen Read-Only-C++-Funktionsbaustein exakt den gleichen Code, wie für einen offenen C++-Funktionsbaustein (identischer CRC). So kann während der Testphase mit offenen C++-Funktionsbausteinen gearbeitet werden. Diese können dann nach Prüfung und Fertigstellung verschlossen werden, ohne dass sich dabei die CRCs ändern. Zur besseren Unterscheidung werden Read-Only-C++-Funktionsbausteine im Programmeditor farblich hervorgehoben.	Ja

Variante	Know-How-Schutz	Mit der Einstellung Know-How-Schutz gelten folgende Einschränkungen gegenüber einem offenen C++-Funktionsbaustein: ▪ Der Editor eines C++-Funktionsbausteins mit Know-How-Schutz kann nicht geöffnet werden. ▪ Im Eigenschaften-Dialog eines C++-Funktionsbausteins mit Know-How-Schutz wird nur der Name angezeigt. Dieses Feld ist nicht editierbar. ▪ Für einen C++-Funktionsbaustein mit Know-How-Schutz wird in der Dokumentation nur das Interface und in den Eigenschaften nur der Name ausgedruckt. Alle anderen Daten sind nicht zugänglich. Know-How-Schutz ist nicht reversibel und kann auch nicht durch Kopieren/Einfügen und Archivieren/Wiederherstellen umgangen werden. Bevor SILworX einen C++-Funktionsbaustein verschließt, muss die Sicherheitsabfrage mit *Ja* beantwortet werden. i In jedem Fall darauf achten, dass ein nicht verschlossenes Backup des C++-Funktionsbausteins verfügbar ist! Der Codegenerator erzeugt für eine POE mit Know-How-Schutz exakt den gleichen Code, wie für einen offenen C++-Funktionsbaustein (identischer CRC). So kann während der Testphase mit offenen C++-Funktionsbausteinen gearbeitet werden. Diese können dann nach Prüfung und Fertigstellung verschlossen werden, ohne dass sich dabei die CRCs ändern. Zur besseren Unterscheidung werden C++-Funktionsbausteine mit Know-How-Schutz im Programmeditor farblich hervorgehoben.	Ja

Tabelle 2: Eigenschaften C++-Funktionsbaustein

4.3 Editor der C++-Funktionsbausteine

Der Editor des C++-Funktionsbausteins beinhaltet die Komponenten zur Erstellung der C++-Funktionsbausteine einschließlich der Import- und Export-Funktionen für die C++-Quellcode-Dateien.

4.3.1 Stack-Bedarf

Für HIMax und HIMatrix Steuerungen sieht SILworX eine Stack-Reserve von 4 kByte vor.
Für C++-Funktionsbausteine mit einem höheren Stack-Bedarf kann dieser in SILworX für jeden C++-Funktionsbaustein einzeln eingestellt werden.

4.3.1.1 Zusätzlichen Stack-Bedarf ermitteln und einstellen

Besteht ein höherer Stack-Bedarf, muss der Anwender abschätzen, wie viele Bytes durch die in dem C++-Funktionsbaustein aufgerufenen C++-Funktionen maximal auf dem Stack belegt werden.

HIMA empfiehlt, zu dem geschätzten maximalen Stack-Bedarf immer eine ausreichend große Reserve einzubeziehen, um ein Höchstmaß an Verfügbarkeit zu gewährleisten.

Ist der Stack-Bedarf nicht ausreichend, kann dies beim Aufruf des C++-Funktionsbausteins zu einem Stack-Überlauf und damit zum Fehlerstopp der Steuerung führen, d. h. zum Übergang der Steuerung in den sicheren Zustand.

Bei der Abschätzung des Stack-Bedarfs muss der Anwender folgende Merkmale des C++-Quellcodes in Betracht ziehen, die den Stack stark vergrößern können:

- große Arrays oder Strukturen als lokale Variablen
- große Strukturen als Übergabevariablen
- verschachtelte C++-Funktionen

HIMA empfiehlt zur Abschätzung des Stack-Bedarfs die folgende Vorgehensweise:

1. Im C++-Quellcode die C++-Funktion mit den umfangreichsten lokalen Variablen und Übergabevariablen ermitteln.
2. Im C++-Quellcode den „tiefsten“ Aufrufpfad aus verschachtelten C++-Funktionen ermitteln, welcher die zuvor ermittelte C++-Funktion beinhaltet.
3. Gesamtgröße aller lokalen Variablen und Übergabevariablen dieses Aufrufpfades ermitteln. Diesen Wert als Stack-Bedarf in den C++-Funktionsbaustein eintragen, evtl. zuzüglich einer weiteren Reserve.
4. Prüfung des C++-Funktionsbausteins auf ausreichenden Stack durch einen Funktionstest auf der HIMax/HIMatrix Steuerung. Eine Prüfung des Stacks in der Offline-Simulation ist nicht möglich.

In SILworX lässt sich der zusätzlich benötigte Stack-Bedarf für jeden C++-Funktionsbaustein einstellen. Die Obergrenze des Wertebereichs entspricht dem maximal zulässigen Gesamt-Speicherbedarf des Programms. Der für den Stack nutzbare Speicherbereich ist daher in der Praxis kleiner.

Parameter	Beschreibung
Stack-Bedarf für HIMatrix Layout 2	Wertebereich: 0…1 048 064 Byte Standard: 0
Stack-Bedarf für HIMatrix Layout 3	Wertebereich: 0…5 177 344 Byte Standard: 0
Stack-Bedarf für HIMax	Wertebereich: 0…10 481 664 Byte Standard: 0

Tabelle 3: Stack-Bedarf für HIMax/HIMatrix Steuerungen

4.3.2 Lokale Variablen im POE-Editor

Das Register **Lokale Variablen** enthält alle lokalen Variablen, die in der aktuellen Logik verwendet werden können. Dies beinhaltet auch die Variablen, welche als Eingangsvariablen und Ausgangsvariablen von C++-Funktionsbausteinen definiert sind.

Im POE-Editor stehen für die Programmierung der C++-Funktionsbausteine die Variablentypen VAR, VAR_INPUT und VAR_OUTPUT zur Verfügung.

4.3.3 Quelldateien

Ein C++-Funktionsbaustein enthält die folgenden Standard Quelldateien:

- Header Datei "SRCUSR<Typ-Dateiname>.h"
- Cpp Datei "SRCUSR<Typ-Dateiname>_Content.cpp"

Optional können zusätzliche Quelldateien importiert werden, siehe Kapitel 4.3.5.

4.3.4 Export von Quelldateien aus einem C++-Funktionsbaustein

Die Funktion **Quelldateien exportieren** exportiert alle in dem C++-Funktionsbaustein vorhandenen Dateien in das Zielverzeichnis (z. B. auf dem PADT). Die so exportierten C++-Funktionsrümpfe können dann in einem Text-Editor bearbeitet und mit eigenen Funktionen erweitert werden.

Quelldateien exportieren

1. Im Strukturbaum **Konfiguration, Bibliothek** öffnen.
2. Rechtsklick auf **C++-Funktionsbaustein** und im Kontextmenü **Edit** wählen.
3. Ins Feld **Quelldateien exportieren** klicken und das Zielverzeichnis auswählen.
4. Mit **OK** bestätigen.
   ☑ Sind im Zielverzeichnis bereits Dateien mit dem gleichen Namen vorhanden, werden in einem Dialogfenster die folgenden Optionen angeboten:
   - *Kopieren und alte Dateien ersetzen*
   - *Nicht kopieren*
   - *Beide Dateien erhalten und alte Datei umbenennen*

Pro erfolgreich exportierter Datei wird eine Meldung in das Logbuch eingetragen.

Die Cpp Datei "SRCUSR<Typ-Dateiname>_Content.cpp.as_imported“ wird beim Export erzeugt (zuletzt importierte .cpp-Datei)

4.3.5 Import von Quelldateien in einen C++-Funktionsbaustein

Mit der Funktion **Quelldateien importieren** werden die zuvor exportierten Quelldateien in den C++-Funktionsbaustein zurückgelesen. Dabei werden alle vorhandenen Quelldateien in dem C++-Funktionsbaustein gelöscht.
Die auf diese Weise importierten und zuvor im Text-Editor mit eigenen Funktionen erweiterten C++-Quelldateien sind nun im C++-Funktionsbaustein eingebunden.

Die Dateinamen der Quelldateien müssen die Endungen ".h" oder ".cpp" tragen.

Quelldateien importieren

1. Im Strukturbaum **Konfiguration, Bibliothek** öffnen.
2. Rechtsklick auf **C++-Funktionsbaustein** und im Kontextmenü **Edit** wählen.
3. Ins Feld **Quelldateien importieren** klicken und das Quellverzeichnis auswählen.
4. Mit **OK** bestätigen
   ☑ Ein Dialog wird geöffnet mit dem Hinweis, dass alle vorhandenen Quelldateien in dem C++-Funktionsbaustein gelöscht werden. Danach werden alle Quelldateien neu in den C++-Funktionsbaustein eingelesen.
5. Mit **OK** bestätigen, wenn Quelldateien gelöscht und neu importiert werden sollen.

5 C++-Quellcode erstellen

In diesem Kapitel wird die Einbindung eigener C++-Funktionen in den C++-Quellcode-Funktionsrumpf beschrieben und welche Regeln dabei beachtet werden müssen.

5.1 Einbindung des C++-Codes

Um die gewünschte Funktion im C++-Quellcode zu implementieren, werden die C++-Quelldateien in den benutzerdefinierten Abschnitten mit den entsprechenden Anweisungen erweitert.

Jeder benutzerdefinierte Abschnitt beginnt unmittelbar nach einer Kommentarzeile der Form:

```
// Start of user code section: <title>
```

Der benutzerdefinierte Abschnitt endet unmittelbar vor einer Kommentarzeile der Form:

```
// End of user code section: <title>
```

Der Platzhalter `<title>` steht für eine englischsprachige Bezeichnung des betreffenden Abschnitts.

i Die Kommentarzeilen, welche die benutzerdefinierten Abschnitte markieren, dürfen nicht verändert werden.

5.1.1 Schnittstellen der C++-Funktionsbausteine für Funktionsbausteinsprache

Ein C++-Funktionsbaustein wird in C++ durch eine Klasse dargestellt. Jede Instanz des Bausteins wird in eine Instanz dieser Klasse übersetzt.

Die in SILworX als VAR oder VAR_OUTPUT deklarierten Variablen erscheinen in C++ als Attribute dieser Klasse.

Der Aufruf des C++-Funktionsbaustein durch einen anderen Funktionsbaustein in SILworX entspricht dem Aufruf der Funktion "CallFunctionContent" der Klasse des Bausteins in C++.

Die in SILworX als VAR_INPUT deklarierten Variablen erscheinen in C++ als Parameter dieser Funktion.

5.1.2 C++-Funktionsbaustein in Funktionsbausteinsprache verwenden

Zur Verwendung eines C++-Funktionsbausteins in der Funktionsbausteinsprache, aus dem Register *Bausteine* den C++-Funktionsbaustein in den Programmeditor ziehen und verbinden.

5.1.3 Code generieren

Es ist kein zusätzlicher Generierungsschritt für C++-Funktionsbausteine nötig. Die *.cpp-Quelldateien werden bei der normalen Codegenerierung in SILworX für eine Ressource übersetzt.

5.2 Prüfung und Diagnose

Zur Überprüfung der korrekten Funktion des C++-Codes ist dieser in einer geeigneten C++-Entwicklungsumgebung zu testen.

Zudem muss im SILworX Online-Test der fertige C++-Funktionsbaustein als Blackbox auf dem System getestet werden, siehe Sicherheitshandbuch HI 801 002 D. Durch Ansteuern der Eingänge und Beobachten der Ausgänge wird geprüft, ob sich der C++-Funktionsbaustein gemäß der gewünschten Funktion verhält.

Eine Diagnose des C++-Codes durch SILworX wird nicht durchgeführt.

5.3 C++-Sprachumfang

Es dürfen nur die in diesem Kapitel aufgezählten Sprachelemente verwendet werden.

5.3.1 Allgemeine C++-Sprachelemente

- Konstanten mit unterstützten Datentypen
- Lokale Variablen mit unterstützten Datentypen
- Globale C++-Variablen mit unterstützten Datentypen
- Literale
- Kommentare
- if .. else
- switch
- while, do..while
- for
- Funktionsdeklarationen und Funktionsaufrufe
- sizeof
- Zuweisungen
- return
- break
- continue
- extern
  nur für C++-Funktionen und Variablen, jedoch keine extern-Deklarationen mit Angabe einer Programmiersprache, z. B. extern "C" typedef void (*CFunction)(void) ist verboten.
- #include
- #ifdef, #if .. #elif .. #else .. #endif
- #define
- #undef (nur für selbst-definierte Elemente)

5.3.2 Namensräume

Für jeden C++-Funktionsbaustein ist ein Namensraum reserviert. Dieser ist dem globalen Namensraum unmittelbar untergeordnet. Der Namensraum-Bezeichner besteht aus dem Präfix N_ gefolgt vom Namen der Klasse des C++-Funktionsbausteins.

Beispiel: namespace N_USRFunctionBlock1 { /* ... */ }

- Ein C++-Funktionsbaustein muss alle nach außen sichtbaren Elemente, die er definiert, dem Namensraum des C++-Funktionsbausteins unterordnen (mittelbar oder unmittelbar).
  Mit "nach außen sichtbar" ist hier gemeint, dass der Name des Elements laut C++-Standard "external linkage" hat. Dazu zählen zwar auch Namensräume, aber der Namensraum des C++-Funktionsbausteins kann sich natürlich nicht selbst untergeordnet sein.
- Ein C++-Funktionsbaustein darf beliebige untergeordnete Namensräume in seinem reservierten Namensraum definieren.
- Ein C++-Funktionsbaustein darf beliebige Alias-Namen für Namensräume definieren, auch im globalen Gültigkeitsbereich ("global scope").
  Beispiel: namespace FB1 = N_USRFunctionBlock1;
- Ein C++-Funktionsbaustein darf Using-Direktiven verwenden, auch im globalen Gültigkeitsbereich.
  Beispiel: using namespace N_USRFunctionBlock1;

5.3.3 Globale C++-Variablen

In einem C++-Funktionsbaustein können lokale und globale C++-Variablen verwendet werden.

⚠ WARNUNG

Die globale C++-Variablen werden beim Reload mit ihren (neuen) Initialwerten überschrieben.

- Der Zugriff (lesen, schreiben) auf globale C++-Variablen darf von allen C++-Funktionsbausteinen eines Programms erfolgen.
- Der Wert von C++-globalen Variablen bleibt zwischen den Anwenderprogrammzyklen erhalten.
- In der Definition jeder von Null verschieden initialisierten globalen C++-Variablen eines C++-Funktionsbausteins muss auf den Bezeichner der globalen C++-Variablen ein Aufruf des Makros X_USERDATA_INIT gefolgt von einem C++-Initialisierungsstring enthalten sein.

  Beispiel:

  namespace N_USRFunctionBlock1 {

  int32 gLastError X_USERDATA_INIT = 1;

  myStruct gStruct X_USERDATA_INIT = { 5, true, 9.3 };

  }
- Bei einer Deklaration der globalen C++-Variablen, die keine Definition ist, darf der C++-Funktionsbaustein das Makro X_USERDATA_INIT hingegen nicht verwenden.
  Beispiel: namespace N_USRFunctionBlock1 { extern int32 gLastError; }
- In der Definition jeder Null-initialisierten globalen C++-Variablen eines C++-Funktionsbausteins muss auf den Bezeichner der globalen C++-Variablen dennoch der Aufruf eines Makros folgen. Für diesen Fall gibt es zusätzlich das Makro X_USERDATA_ZERO, welches ohne weitere Angaben verwendet wird.
  Beispiel:
  namespace N_USRFunctionBlock1 {
  int32 gLastError X_USERDATA_ZERO;
  myStruct gStruct X_USERDATA_ZERO;
  }
- Bei einer Deklaration der globalen C++-Variablen, die keine Definition ist, darf der C++-Funktionsbaustein das Makro X_USERDATA_ZERO hingegen nicht verwenden.
  Beispiel: namespace N_USRFunctionBlock1 { extern int32 gLastError; }

5.3.4 C++-Zeiger

Zeiger sollten gemäß IEC 61508 vermieden werden, wo dies sinnvoll möglich ist.

Ein C++-Funktionsbaustein darf für Zeiger nur folgende Werte verwenden:

- die entsprechenden Nullzeiger-Werte.
- die Adressen von Objekten, die derselbe oder ein anderer C++-Funktionsbaustein definiert.
- die Adressen von Funktionen, die derselbe oder ein anderer C++-Funktionsbaustein definiert.

5.3.5 Includes

Eine C++-Funktionsbaustein darf Include-Anweisungen der Form #include "Dateiname" verwenden, wobei jede dieser Anweisungen folgende Bedingungen erfüllen muss:

- Der Dateiname muss ohne Verzeichnispfad angegeben sein.
- Der Dateiname muss eine zusätzliche Quelldatei desselben oder eines anderen C++-Funktionsbausteins identifizieren.
- Ein C++-Funktionsbaustein (A) kann auf diese Weise eine Quelldatei aus einem anderen C++-Funktionsbaustein (B) einbinden. Dabei muss jedes Anwenderprogramm, das eine Instanz des einen C++-Funktionsbausteins (A) aufruft, auch mindestens eine Instanz des anderen C++-Funktionsbausteins (B) aufrufen.

Ein C++-Funktionsbaustein darf Include-Anweisungen der Form #include <Header> verwenden, wobei jedoch nur die folgenden Header zulässig sind:

float.h

limits.h

math.h

stddef.h

Darüber hinaus darf ein C++-Funktionsbaustein in jeder zusätzlichen Quelldatei die folgende Sequenz von Include-Anweisungen verwenden (die von SILworX auch in der Haupt-Quelldatei "SRCUSR<Typ-Dateiname>_Content.cpp" generiert wird):

```
#include "iec_types.h"

#include "iec_std_types.h"

#include "SRCUSR<Typ-Dateiname>.h"
```

Wenn vorhanden, dann müssen diese Anweisungen am Anfang der Quelldatei stehen (es dürfen nur Kommentare und Leerzeilen vorausgehen). Die gezeigte Reihenfolge muss eingehalten werden, und die Sequenz muss vollständig sein.

5.3.6 Datentypen

Die folgende Tabelle zeigt die zueinander passenden Datentypen, wie diese jeweils in SILworX und C++ verwendet werden. Die jeweiligen Alias-Namen sind ebenfalls im C++-Quellcode zulässig.

i Die jeweilige Zuordnung von Alias-Name und SILworX-Datentyp ist festgeschrieben. Der zugeordnete elementare C++-Datentyp kann sich jedoch theoretisch ändern, insbesondere bei zukünftigen Produkten.
HIMA empfiehlt: Wenn es auf die Größe einer Variablen bzw. auf die Kompatibilität zu den SILworX-Datentypen ankommt, sollte der Anwender die vom PADT definierten Alias-Namen einsetzen, wenn vorhanden.

SILworX-Datentyp	Alias-Namen	C++-Datentyp	Größe in Byte
BOOL	-	bool	1
BYTE und USINT	ubyte, uint8	unsigned char	1
WORD und UINT	uword, uint16	unsigned short	2
DWORD und UDINT	udword, uint32	unsigned long	4
LWORD und ULINT	uldword, uint64	unsigned long long	8
SINT	int8	char	1
INT	int16	short	2
DINT	int32	long	4
LINT und TIME	int64, time	long long	8
REAL	real	float	4
LREAL	lreal	double	8

Tabelle 4: Datentypen

Für Variablen des Datentyps bool ist als gültige interne Darstellung nur das Bitmuster 0x00 für false bzw. 0x01 für true erlaubt. Dies ist von Bedeutung, wenn der Speicher einer bool-Variablen auf anderem Wege geändert wird als durch Zuweisung dieser Variablen.

Zudem können noch void, Strukturen, Arrays, Zeiger, Referenzen sowie selbst definierte Datentypen verwendet werden.

5.3.7 Funktionen und Operatoren

Es dürfen zusammengesetzte Ausdrücke verwendet werden.

5.3.7.1 Logische Operatoren

Die folgende Tabelle zeigt die logischen Operatoren, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	C++-Operator	Verwendbare SILworX-Datentypen
AND	&&	BOOL
OR	||	BOOL
NOT	!	BOOL

Tabelle 5: Logische Operatoren

5.3.7.2 Bitweise Operatoren

Die folgende Tabelle zeigt die bitweisen Operatoren, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	C++-Operator	Verwendbare SILworX-Datentypen
AND	&	alle ANY_Bit-Datentypen außer BOOL
OR	|	alle ANY_Bit-Datentypen außer BOOL
NOT	~	alle ANY_Bit-Datentypen außer BOOL
XOR	^	alle ANY_Bit-Datentypen außer BOOL
SHR	>>	alle ANY_Bit-Datentypen außer BOOL
SHL	<<	alle ANY_Bit-Datentypen außer BOOL

Tabelle 6: Bitweise Operatoren

5.3.7.3 Vergleichsoperatoren

Die folgende Tabelle zeigt die Vergleichsoperatoren, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	C++-Operator
GE	>=
GT	>
LE	<=
LT	<
EQ	==
NE	!=

Tabelle 7: Vergleichsoperatoren

5.3.7.4 Cast-Operatoren

Die folgende Tabelle zeigt die Cast-Operatoren, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	Alias-Namen	C++-Operator
AtoBOOL	-	(bool)
AtoBYTE	(ubyte), (uint8)	(unsigned char)
AtoWORD bzw. AtoUINT	(uword), (uint16)	(unsigned short)
AtoDWORD bzw. AtoUDINT	(udword), (uint32)	(unsigned long)
AtoLWORD bzw. AtoULINT	(uldword), (uint64)	(unsigned long long)
AtoSINT	(int8)	(char)
AtoINT	(int16)	(short)
AtoDINT	(int32)	(long)
AtoLINT bzw. AtoTIME	(int64), (time)	(long long)
AtoREAL	(real)	(float)
AtoLREAL	(lreal)	(double)

Tabelle 8: Cast-Operatoren

Beim Cast von float oder double zu einem Integer-Datentyp wird in C++ die trunc-Funktion verwendet. Im Gegensatz dazu wird bei den SILworX-Funktionen der Wert gerundet, gemäß IEC 61131-3.

Beim Cast von float oder double zu (unsigned) long long ergibt sich im C++-Code ein Unterschied zu der korrespondierenden SILworX-Funktion, wenn der Wertebereich des Ziel-Datentyps überschritten wird.

5.3.7.5 Arithmetische Operatoren

Die folgende Tabelle zeigt die arithmetischen Operatoren, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	C++-Operator
ADD	+
SUB	-
MUL	*
DIV	/
MOD	%

Tabelle 9: Arithmetische Operatoren

5.3.7.6 Numerische Funktionen

Die folgende Tabelle zeigt die numerischen Funktionen, wie diese jeweils in SILworX und C++ bezeichnet werden.

SILworX-Funktion	C++-Funktion
SQRT	sqrtf(float), sqrt(double)
ABS	abs(char), abs(short), labs(long), llabs(long long), fabsf(float), fabs(double)
SIN	sinf(float), sin(double)
ASIN	asinf(float), asin(double)
COS	cosf(float), cos(double)
ACOS	acosf(float), acos(double)
TAN	tanf(float), tan(double)
ATAN	atanf(float), atan(double)
EXPT	powf(float, float), pow(double, double)
EXP	expf(float), exp(double)
LN	logf(float), log(double)
LOG	log10f(float), log10(double)

Tabelle 10: Numerische Funktionen

5.4 Dateiformat und erlaubte Zeichen

Als C++-Quelldateien für einen C++-Funktionsbaustein sind nur Textdateien mit Zeichen gemäß der Codierung UTF-8 erlaubt.

Der Unicode®-Codepoint eines Zeichens wird hier jeweils als hexadezimale Zahl mit dem Präfix U+ angegeben.

Die Zeichen U+0021 bis U+007E sind im gesamten Quelltext eines C++-Funktionsbausteins erlaubt.

Zur Veranschaulichung die erlaubten Zeichen:

! " # $ % & ' ( ) * + , - . /

0 1 2 3 4 5 6 7 8 9 : ; < = > ?

@ A B C D E F G H I J K L M N O

P Q R S T U V W X Y Z [\ ] ^ _

` a b c d e f g h i j k l m n o

p q r s t u v w x y z { | } ~

Zudem sind das gewöhnliche Leerzeichen (U+0020) und der horizontale Tabulator (U+0009) im gesamten Quelltext eines C++-Funktionsbausteins erlaubt.

Innerhalb von Kommentaren sind zusätzlich die Zeichen U+00A1 bis U+00FF erlaubt, z. B.: « » ± ° ¹ ² ³ ¼ ½ ¾ µ Ä Ö Ü ä ö ü ß

5.5 Regeln für Dateinamen

Für die Dateinamen der zusätzlichen Quelldateien eines C++-Funktionsbausteins gelten die nachfolgend beschriebenen Regeln.

- Der Dateiname darf nur folgende Zeichen enthalten:
  0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z
- Der Dateiname darf insbesondere kein Leerzeichen (U+0020) enthalten und muss die Endung ".h" oder ".cpp" tragen.
- Der Dateiname darf nicht mit einer der Zeichenfolgen "SRC", "HIMA" oder "IEC" beginnen. Dies gilt auch für Varianten der Groß-/Kleinschreibung.
- Der Dateiname darf nicht mit dem Namen einer mit SILworX installierten Header-Datei übereinstimmen. (Das sind z. B. folgende Dateien mit der Endung ".h" aus der Verzeichnisstruktur unter "target_codegen" in der PADT-Installation: amdecl.h, ansi.h, asm.h, base.h, binders.h, cast.h, complex.h, config.h, cpl.h, ctype.h, errno.h, glu.h, hidpi.h, limits.h, lock.h, m68k.h, malloc.h, map.h, math.h, rs6000.h, stat.h, time.h, varargs.h, wchar.h)

5.6 Nicht erlaubte C++-Sprachelemente

Ein C++-Funktionsbaustein darf die erlaubten Sprachelemente nicht dazu verwenden, die Semantik der von SILworX generierten Abschnitte des C++-Quellcodes zu verändern oder zu umgehen.

Ein C++-Funktionsbaustein kann oder darf folgende Features nicht verwenden:

- Dynamische Speicherverwaltung
- goto
- Inline Assembler
- Hinzufügen von Bibliotheken in Binärform
- Rekursion
- Interrupts
- #pragma
- Runtime Type Information (RTTI)
  - dynamic_cast
  - typeid
  - Exception-Handling

5.6.1 Makros

Ein C++-Funktionsbaustein darf keine Makro-Definition hinzufügen oder entfernen, deren Makro-Bezeichner mit einem der folgenden Präfixe beginnt:

_, AM_, AMDEF_, AMDECL_, DLL_, HIMA_, IEC_, LS_, SECTION_, SRCUSR_, TARGET_, X_

5.6.2 Include

Ein C++-Funktionsbaustein darf eine Include-Anweisung nicht mittels eines Makros formen.

Beispiel (folgendes ist verboten):

#define INCFILE "incfile.h"

#include INCFILE

5.6.3 Datentypen

Ein C++-Funktionsbaustein darf keine Zeiger verwenden, die der C++-Standard als "pointers to non-static class members" beschreibt.

Ein C++-Funktionsbaustein darf einen Nullzeiger ("null pointer") nicht dereferenzieren.

Ein C++-Funktionsbaustein darf den Datentyp long double nicht verwenden.

Ein C++-Funktionsbaustein darf Klassen/Unions nicht innerhalb einer Funktion definieren.

5.6.4 Funktionen

Ein C++-Funktionsbaustein darf keine globale Funktion namens main, WinMain oder DllMain definieren. Auch nicht innerhalb eines Namensraums!

5.6.5 Variablen

Ein C++-Funktionsbaustein darf keine statischen Variablen innerhalb einer Funktion definieren.

Ein C++-Funktionsbaustein darf keine statischen Datenelemente in Klassen/Unions definieren.

5.6.6 Literale

Ein C++-Funktionsbaustein darf keine String-Literale verwenden.

5.6.7 Operatoren

Ein C++-Funktionsbaustein darf keine Daten verändern, die durch SILworX als const deklariert wurden.

5.6.8 Sonstige Syntax

Der Quelltext eines C++-Funktionsbausteins darf keine extern-Deklarationen mit Angabe einer Programmiersprache enthalten.

Beispiele (folgendes ist verboten):

```
extern "C" typedef void (*CFunction)(void);

extern "C" { void Func1(void); }
```

5.6.9 Verwendung von globalen SILworX-Variablen

Ein C++-Funktionsbaustein darf nicht auf globale Variablen von SILworX zugreifen, weder lesend noch schreibend.

Anhang

Tabellenverzeichnis

Tabelle 1: Erforderliche Prozessorbetriebssysteme 8
Tabelle 2: Eigenschaften C++-Funktionsbaustein 12
Tabelle 3: Stack-Bedarf für HIMax/HIMatrix Steuerungen 13
Tabelle 4: Datentypen 19
Tabelle 5: Logische Operatoren 19
Tabelle 6: Bitweise Operatoren 20
Tabelle 7: Vergleichsoperatoren 20
Tabelle 8: Cast-Operatoren 20
Tabelle 9: Arithmetische Operatoren 21
Tabelle 10: Numerische Funktionen 21

HI 801 318 D

Albert-Bassermann-Str. 28
68782 Brühl, Deutschland
Tel.: +49 6202 709-0
Fax +49 6202 709-107
security@hima.com
www.hima.com

SAFETY
NONSTOP

Eine detaillierte Liste aller Niederlassungen und Vertretungen
finden Sie unter: www.hima.de/kontakt