

# Dokumentation HSL3 SDK

## Versionshistorie

Version	ab Firmware	Beschreibung / Änderungen	Datum
1.0	4.13.0	Erstveröffentlichung	23.01.2026

## Inhaltsverzeichnis

Versionshistorie .....	1
Dokumentation HSL3 SDK.....	3
Einführung .....	3
Abkündigung HSL2 .....	3
Haftungsausschluss.....	3
Entwicklungsumgebung.....	4
Einrichtung:.....	4
Nutzung:.....	4
Erste Schritte.....	5
Schritt 1: Quellcode-Datei anlegen.....	5
Schritt 2: Code-Struktur.....	5
Schritt 3: Baustein generieren .....	6
Schritt 4: Übertragen .....	7
Klassen .....	8
LogicModule .....	8
Hsl3Framework.....	9
Methoden .....	9
Hsl3DebugSection.....	12
Methoden .....	12
Hsl3Slots.....	14
Zugriff.....	14
Methoden .....	14
Hsl3Slot .....	16
Attribute.....	16
Bibliotheken.....	16
Externe Bibliotheken .....	16

HSL3 Generator.....	17
Übersicht.....	17
Voraussetzungen .....	17
Aufruf des Generators .....	17
Beispiel.....	17
Beschreibung der Befehlszeilenparameter .....	18
Konfigurationsdatei .....	18
Attribute der Konfigurationsdatei .....	18
HSL 3 - Kommunikation zwischen Logikbaustein-Instanzen.....	21
Methoden im Framework.....	21
Beispiel.....	22
Ablauf der Kommunikation.....	22
Hinweise zur Migration von HSL2 zu HSL3 .....	23
Einführung .....	23
Unterschiede zwischen HSL 2 und HSL 3 .....	23
Tipps für die Migration von Logikbausteinen von HSL 2 zu HSL 3.....	23
Python 2 in Python 3 Quellcode konvertieren mit 2to3.....	23
Strings und Bytes .....	24
Verwendung von remanenten Speichervariablen.....	25

## Dokumentation HSL3 SDK

### Einführung

Seit der Firmware 4.13 (Python Version 3.9) besteht die Möglichkeit, mit Hilfe des HSL3 SDK eigene Logikmodule für den Gira HomeServer bzw. Gira FacilityServer zu entwickeln.

Dieses Dokument beschreibt die Möglichkeit, Logikbausteine mit dem HSL3 SDK zu entwickeln. Die Grundlagen der Logikbaustein-Entwicklung sind im HSL1 SDK beschrieben.

In zukünftigen Firmware Versionen des Gira HomeServer wird auch die aktuell verwendete Python Version 3.9 angehoben werden. Daher sind HSL3 Logikbausteine grundsätzlich auf Kompatibilität zu neueren Firmware Versionen zu prüfen. Es empfiehlt sich daher, bereits bei der Erstellung von Logikbausteinen darauf zu achten, dass keine abgekündigten Python Funktionen verwendet werden.

### Abkündigung HSL2

Das HSL2 SDK wird in zukünftigen Versionen der Firmware nicht mehr unterstützt. Bausteine auf Basis des HSL2 SDKs müssen auf das HSL3 SDK migriert werden.

### Haftungsausschluss

Verantwortlich für die Erstellung und das Funktionieren von Logikbausteinen ist der Ersteller. Diese Dokumentation stellt eine Hilfestellung dar. Auch wenn die Unterlagen mit großer Sorgfalt erstellt und geprüft sind, können Fehler nicht vollkommen ausgeschlossen werden. Deshalb übernimmt der Inverkehrbringer bezüglich des Inhalts weder eine juristische Verantwortung, noch irgendeine Gewähr.

Technische Änderungen bleiben vorbehalten!

## Entwicklungsumgebung

### Einrichtung:

Um eine zur Version passende Python Entwicklungsumgebung einzurichten, folgende Anleitung:

1. Installieren sie das Tool „Python install manager“ von <https://www.python.org/downloads/windows/>
2. Eingabeaufforderung (cmd.exe) starten und folgende Befehle ausführen:

```
C:\Users\Your Name> py install 3.9
C:\Users\Your Name> cd Projekte
C:\Users\Your Name\Projekte> py -3.9 -m venv hsl3
C:\Users\Your Name\Projekte> cd hsl3
C:\Users\Your Name\Projekte\hsl3> Scripts\activate
(hsl3) C:\Users\Your Name\Projekte\hsl3>
```

3. Den Inhalt der Datei „hsl3\_generator\_und\_beispiele.zip“ in den Ordner (Projekte\hsl3) entpacken.
4. Für jeden Logikbausteinprojekt einen weiteren Ordner anlegen, in dem der Source Code abgelegt wird.

### Nutzung:

1. Eingabeaufforderung (cmd.exe) starten

```
C:\Users\Your Name> cd Projekte\hsl3
C:\Users\Your Name\Projekte\hsl3> Scripts\activate
(hsl3) C:\Users\Your Name\Projekte\hsl3>
```

2. Um aus dem Code einen Logikbaustein zu erzeugen, kann der mitgelieferte Generator genutzt werden.

```
(hsl3) C:\Users\Your Name\Projekte\hsl3> python generator3.cpython-39.pyc
--source examples\binary_trigger\config_binary_trigger.json
--target 10700_binary_trigger.hsl --debug
```

3. Die erzeugte \*.hsl Datei kann dann ab Experte 4.13.0 importiert und genutzt werden.

Weitere Informationen im Kapitel HSL3 Generator.

## Erste Schritte

Jeder Logik-Baustein basiert auf einer Klasse mit dem Namen LogicModule. Bei der Instanziierung wird dem Konstruktor eine Instanz der Klasse Hsl3Framework übergeben.

### Schritt 1: Quellcode-Datei anlegen

Jede Datei muss mit dem Präfix hsl3\_ und der Baustein-ID beginnen und mit dem Suffix .py enden. Nur Dateien mit diesem Aufbau können vom HSL3 Generator umgewandelt werden.

Als Baustein-ID z. B. die 10001 verwenden. Aus der Baustein-ID und dem freien Zusatz "my\_module" ergibt sich dann für die Quellcode-Datei der Dateiname hsl3\_10001\_my\_module.py.

### Schritt 2: Code-Struktur

Nachdem die leere Datei angelegt wurde, kann das Skript mit folgendem Grundgerüst gefüllt werden:

```
class LogicModule:
    def __init__(self, hsl3):
        self.fw = hsl3

    def on_init(self, inputs, store):
        pass

    def on_calc(self, inputs):
        pass

    def on_timer(self, timer):
        pass
```

Jeder Logikbaustein besteht immer aus der Klasse LogicModule.

## Schritt 3: Baustein generieren

Mit Hilfe des im SDK befindlichen HSL3 Generator kann der Baustein in eine HSL-Datei umgewandelt werden.

### Schritt 3.1: Konfigurationsdatei erzeugen

In der Konfigurationsdatei wird die äußere Hülle des Bausteins erzeugt. Dafür kann unter dem Dateinamen config.json eine Konfigurationsdatei mit folgendem Inhalt erzeugt werden:

```
{
  "module": {
    "id": "10001",
    "context": "hsl3.examples",
    "category": "Examples",
    "name": "My first module",
    "hsl_filename": "10001_my_module.hsl",
    "inputs": [
      {
        "type": "number",
        "identifier": "In1",
        "init_value": 0,
        "label": "Input 1"
      }
    ],
    "outputs": [
      {
        "type": "string",
        "identifier": "Out1",
        "init_value": "-",
        "label": "Output 1"
      }
    ],
    "store": [
      {
        "type": "string",
        "identifier": "Sto1",
        "init_value": "-",
      }
    ],
    "timer": [
      {
        "identifier": "Tim1",
      }
    ]
  }
}
```

```
],  
  "scripts": [  
    {  
      "filename": "hsl3_10001_my_module.py"  
    }  
  ]  
}
```

Diese einfache Konfiguration erzeugt einen Baustein mit einem Eingang und einem Ausgang.

### Schritt 3.2: Generator ausführen

Den Generator im Ordner der Konfigurationsdatei aufrufen:

```
python generator3x.pyc -source=config.json
```

Der Aufruf erzeugt die gewünschte HSL-Datei.

### Schritt 4: Übertragen

Die generierte HSL-Datei kann direkt mit dem HS/FS Experte genutzt und direkt auf das Gerät oder zum schnellen Testen auf einer HS/FS VM aufgespielt werden.

## Klassen

### LogicModule

Die folgenden beschriebenen Methoden stellen das Grundgerüst eines Logik-Bausteins dar. Besteht ein Projekt aus mehreren Dateien, darf die Klasse LogicModule nur im Hauptskript vorkommen.

### Konstruktor

```
__init__(self, hsl3fw)
```

Muss angegeben werden und wird aufgerufen, wenn der Baustein instanziiert wird.

Der Code wird im allgemeinen Kontext der HSL3-Umgebung aufgerufen. Hier dürfen keine blockierenden Aufrufe getätigt werden. Jede Verzögerung hat Einfluss auf alle HSL3-Bausteine.

### Parameter:

- hsl3 - Übergibt eine Instanz der Klasse *Hsl3Framework*.

### Methoden

```
on_init(self, inputs, store)
```

Wird bei der Initialisierung der Logik aufgerufen. Der Code wird im Kontext des Bausteins aufgeführt.

### Parameter:

- inputs - Enthält alle Eingangswerte. Instanz der Klasse Hsl3Slots.
- store - Enthält die Werte aller Speichervariablen. Instanz der Klasse Hsl3Slots.

```
on_calc(self, inputs)
```

Wird aufgerufen, wenn ein Eingang beschrieben wurde und der Baustein neu berechnet wird.

### Parameter:

- inputs - Enthält alle Eingangswerte. Instanz der Klasse Hsl3Slots.

```
on_timer(self, timer)
```

Wird aufgerufen, wenn ein Timer ausgelöst wurde. Werden keine Timer verwendet, kann auf diese Methode verzichtet werden.

### Parameter:

- timer - Enthält alle definierten Timer und deren Zustand. Instanz der Klasse Hsl3Slots.



## Hsl3Framework

Eine Instanz dieser Klasse wird jedem Baustein bei der Instanziierung übergeben.

### Methoden

#### `set_output(index_or_key, value)`

Setzt den angegebenen Ausgang auf den übergebenen Wert. Darf nur im Kontext des Bausteins aufgerufen werden. Wird die Methode in einem anderen Kontext/Thread aufgerufen, wird eine Exception ausgelöst.

#### Parameter:

- `index_or_key` - Index oder Schlüssel des Eingangs. Existiert der Index oder Schlüssel nicht, wird eine Exception ausgelöst.
- `value` - Wert, der auf den Eingang geschrieben werden soll. Ein mehrfaches Aufrufen hintereinander überschreibt den vorherigen Wert. Es wird nur ein Wert auf den Ausgang gesendet.

Wichtig: Der Parameter muss vom Typ `float`, `int` oder `bytes` sein. Die Firmware des HS/FS unterstützt innerhalb der Abarbeitung der Logik nur Strings vom Typ `"bytes"`. Ein De-/Encoding muss ggf. durch den Baustein selbst vorgenommen werden.

#### Exceptions:

- Wird die Methode nicht aus dem Thread des Kontexts aufgerufen, wird eine Exception vom Typ `Hsl3ContextError` ausgelöst.
- Wird beim Parameter `value` `None` oder ein Wert vom Typ `"string"` übergeben, wird eine Exception vom Typ `ValueError` ausgelöst.

#### `set_store(index_or_key, value)`

Setzt die angegebene Speichervariable auf den übergebenen Wert. Darf nur im Kontext des Bausteins aufgerufen werden.

#### Parameter:

- `index_or_key` - Index oder Schlüssel der Speichervariable. Existiert der Index oder Schlüssel nicht, wird eine Exception ausgelöst.
- `value` - Wert, der auf die Speichervariable geschrieben werden soll. Muss vom Typ `float`, `int` oder `bytes` sein, ansonsten wird eine Exception ausgelöst.

Wichtig: Der Parameter muss vom Typ `float`, `int` oder `bytes` sein. Die Firmware des HS/FS unterstützt innerhalb der Abarbeitung der Logik nur Strings vom Typ `"bytes"`. Ein De-/Encoding muss ggf. durch den Baustein selbst vorgenommen werden.

#### Exceptions:

- Wird die Methode nicht aus dem Thread des Kontexts aufgerufen, wird eine Exception vom Typ `Hsl3ContextError` ausgelöst.
- Wird beim Parameter `value` `None` oder ein Wert vom Typ `string` übergeben, wird eine Exception vom Typ `ValueError` ausgelöst.

`set_timer(index_or_key, seconds)`

Setzt den angegebenen Timer auf den übergebenen Wert. Darf nur im Kontext des Bausteins aufgerufen werden.

Parameter:

- `index_or_key` - Index oder Schlüssel des Eingangs. Existiert der Index oder Schlüssel nicht, wird eine Exception ausgelöst.
- `seconds` - Wert in Sekunden, nach denen der Timer ausgelöst wird. Wird der Wert `None` oder `0` übergeben, wird der Timer angehalten. Wird ein ungültiger Wert übergeben, wird eine Exception ausgelöst.

Exceptions:

- Wird die Methode nicht aus dem Thread des Kontexts aufgerufen, wird eine Exception vom Typ `Hsl3ContextError` ausgelöst.

Beispiel:

```
# Der erste Timer wird in 120 Sekunden aufgerufen
self.fw.set_timer(1, 120)
# Der zweite Timer wird in 240 Sekunden aufgerufen
self.fw.set_timer(2, 240)
# Der zweite Timer wird angehalten
self.fw.set_timer(2, 0)
```

`run_in_context(callback, params)`

Ruft eine Methode im Kontext des Bausteins auf. Diese Methode ist immer dann notwendig, wenn Daten aus einem fremden Thread im Kontext des Bausteins aufgerufen und verarbeitet werden sollen.

Parameter:

- `callback` - Methode, die im Kontext des Bausteins aufgerufen wird.
- `params` - Tuple, welches der aufgerufenen Methode als Parameter übergeben wird.

Exceptions:

- Wird beim Parameter `callback` keine Methode übergeben, wird eine Exception vom Typ `ValueError` ausgelöst.
- Wird beim Parameter `params` kein Tupel übergeben, wird eine Exception vom Typ `ValueError` ausgelöst.

Beispiel:

```
# Ruft die Methode calc_and_send der eigenen Instanz auf.
# Dem Aufruf werden die drei Parameter 1, 2 und 3 übergeben
# def calc_and_send(self, p1, p2, p3):
#     pass
self.fw.run_in_context(self.calc_and_send, (1, 2, 3))
```

`get_logger(host, port, console, level)`

Erzeugt einen Logger, mit dem Log-Meldungen über Syslog abgesetzt werden können. Der HS-eigene Syslog-Server ist unter der Adresse 127.0.0.1:65002 zu erreichen.

Parameter:

- `host` - Optional, die Zieladresse des Syslog-Servers (Default: 127.0.0.1)
- `port` - Optional, der IP-Port des Syslog-Servers (Default: 65002)
- `console` - Optional, wenn True, dann werden die Ausgaben auch auf dem Bildschirm ausgegeben. Sollte nur zu Testzwecken verwendet werden (Default: False).
- `level` - Optional, legt das Log-Level fest (Default: `logging.INFO`). Hier werden die Konstanten auf dem Modul `logging` verwendet.

Beispiel:

```
# Ohne Parameter
```

```
self.logger = self.fw.get_logger()
```

```
# Externer Syslog-Server
```

```
self.logger = self.fw.get_logger(host="192.168.0.12", port=514)
```

`create_debug_section()`

Erzeugt eine Instanz der Klasse `Hsl3DebugSection`.

Beispiel:

```
self.debug = self.fw.create_debug_section()
```

```
self.debug.set("Field 1", "?")
```

```
self.debug.set("Field 2", "?")
```

`get_instance(instance_id)`

Liefert die Instanz eines HSL3-Bausteins zurück.

Parameter:

- `instance_id` - Die ID eines HSL3-Bausteins. Die Instanz muss zum gleichen Kontext gehören.

`get_context_id()`

Liefert die ID des aktuellen Kontexts zurück (String).

`get_instance_id()`

Liefert die ID der aktuellen Instanz zurück (Int).

`get_module_id()`

Liefert die ID des aktuellen Moduls zurück (Int).

## Hsl3DebugSection

Eine Instanz dieser Klasse wird beim Erzeugen einer Debug-Sektion zurückgegeben.

Die Reihenfolge innerhalb der Sektion wird durch die Reihenfolge der Definition bestimmt.

### Methoden

`set(name, value)`

Definiert ein Feld in der Sektion und setzt dessen Wert.

Parameter:

- name - Name des Feldes.
- value - Wert des Feldes.

`inc(name, value=1)`

Der Wert des Feldes wird erhöht.

Parameter:

- name - Name des Felds. Ist das Feld noch nicht vorhanden, wird der Wert mit value initialisiert.
- value - Wert, um den der Inhalt des Felds (Default: 1) erhöht wird. Ist der Wert kein numerischer Wert, wird eine Exception ausgelöst.

`avg(name, value=None)`

Berechnet den Durchschnitt eines Wertes.

Parameter:

- name - Name des Felds. Ist das Feld noch nicht vorhanden, wird der Wert mit value initialisiert.
- value - Ist der Wert ein numerischer Wert, wird der Durchschnitt berechnet. Wird None übergeben, wird das Feld zurückgesetzt.

`timestamp(name, value=None)`

Parameter

- name - Name des Felds. Ist das Feld noch nicht vorhanden, wird der Wert mit value initialisiert.
- value - Wenn gesetzt, wird dieser Wert übernommen, ansonsten der aktuelle Zeitstempel.

`log(msg)`

Parameter

- msg - Trägt den Text in das Protokoll ein.

exception(msg)

## Parameter

- msg - Trägt die aktuelle Exception in das Protokoll ein und fügt den Text als zusätzliche Information hinzu.

## Hsl3Slots

Instanzen dieser Klasse werden beim Aufruf der folgenden Methoden übergeben:

- `on_init(inputs: Hsl3Slots, store: Hsl3Slots)`
  - `inputs` - Enthält alle Eingänge
  - `store` - Enthält alle Speichervariablen
- `on_calc(inputs: Hsl3Slots)`
  - `inputs` - Enthält alle Eingänge
- `on_timer(timer: Hsl3Slots)`
  - `timer` - Enthält alle definierten Timer

## Zugriff

Auf eine Instanz der Klasse `Hsl3Slots` kann direkt über den Index oder Schlüssel des einzelnen Slots zugegriffen werden.

Beispiel:

```
def on_init(self, inputs, store):
    # Liefert Instanz von Hsl3Slot
    slot_e1 = inputs[1]
    # Liefert den Wert von Eingang 1
    slot_e1_wert = inputs[1].value

def on_calc(self, inputs):
    # Ruft den Wert von Eingang 1 ab
    if inputs[1].changed:
        slot_e1_wert = inputs[1].value
```

## Methoden

`get(index_or_key)`

Parameter

- `index_or_key`

Rückgabe:

Liefert ein Objekt vom Typ `Hsl3Slot` zurück oder löst eine `Exception` aus, wenn der Index oder der Schlüssel nicht existiert.

`keys()`

Rückgabe:

Liefert eine Liste aller vorhandenen Schlüssel zurück.

`changed(index_or_key)`

Parameter

- `index_or_key`

## Rückgabe:

Liefert *True* zurück, wenn ein Wert eingetroffen ist. Wurde an diesem Eingang kein Wert empfangen, oder es wurde kein Wert zu dem übergebenen Schlüssel gefunden, wird *False* zurückgegeben

`value(index_or_key)`

## Parameter

- `index_or_key`

## Rückgabe:

Liefert den Wert zurück. Wurde der entsprechende Index oder Schlüssel nicht gefunden, wird *None* zurückgegeben.

## Hsl3Slot

Instanzen dieser Klasse werden beim Aufruf der folgenden Methoden übergeben:

### Attribute

#### value

Liefert den Wert des jeweiligen Slots zurück:

- Eingang - Der Wert spiegelt den Wert des Eingangs wider.
- Store - Der Wert spiegelt den Wert der Speichervariable wider.
- Timer - Der Wert spiegelt die Dauer des Timers wider.

Der Wert ist immer vom Typ float, int oder bytes.

#### changed

Liefert *True* zurück, wenn sich der Wert geändert hat:

- Eingang - True, wenn sich der Wert geändert hat und eine Berechnung ausgelöst wurde (Aufruf von `on_calc`). Während der Initialisierung immer False (Aufruf von `on_init`).
- Store - Immer False.
- Timer - True, wenn der entsprechende Timer ausgelöst wurde (Aufruf von `on_timer`)

## Bibliotheken

In der Firmware sind alle Standardbibliotheken von Python 3 enthalten.

### Externe Bibliotheken

Neben den Standardbibliotheken stehen noch die folgenden externen Bibliotheken zur Verfügung:

- requests
- websockets
- beautifulsoup4
- pytz
- python-dateutil
- pymodbus



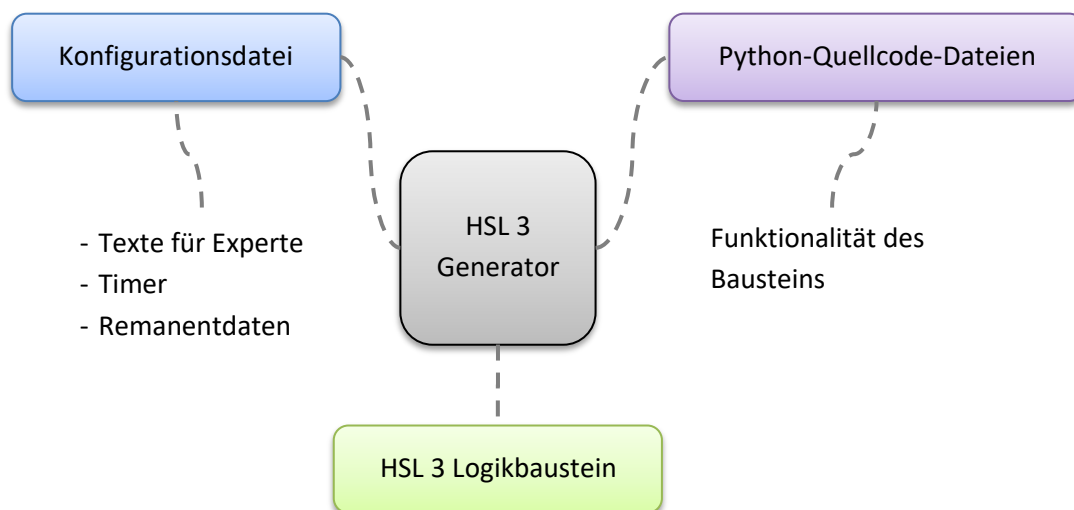
## HSL3 Generator

### Übersicht

Um das Entwickeln von Logikbausteinen zu erleichtern, steht der Generator zur Verfügung. Unter Verwendung einer Konfigurationsdatei und den Quellcode-Dateien des Logikbausteins erstellt der Generator den fertigen HSL 3 Logikbaustein.

In der Konfigurationsdatei werden einige Eigenschaften des Logikbausteins und die Texte für die Anzeige im Experte angegeben.

### Nutzung des Generators



### Voraussetzungen

Zur Verwendung des Generators wird Python 3x benötigt.

### Aufruf des Generators

Der Generator wird über die Kommandozeile mit dem Python-Interpreter aufgerufen. Als Befehlszeilenparameter stehen zur Verfügung: `--source` dient zur Angabe der Konfigurationsdatei (Standardwert: `config.xml`). Mit `--target` wird die Zieldatei festgelegt, wobei diese Angabe Vorgaben der Konfigurationsdatei überschreibt. Ohne diese Angabe bestimmt die Konfigurationsdatei den Namen der Zieldatei.

Optional aktiviert `--debug` Debug-Ausgaben, die standardmäßig deaktiviert sind.

### Beispiel

```
python generator3x.pyc --source my_config.json --target  
10001_my_module.hsl --debug
```

Der Generator wird mit `python generator3x.pyc` aufgerufen. `--source my_config.json` gibt an, dass `my_config.json` als Konfigurationsdatei verwendet

werden soll. `--target 10001_my_module.hsl` legt fest, dass die Ausgabedatei `10001_my_module.hsl` heißen soll, was eine mögliche Vorgabe in der Konfigurationsdatei überschreibt. `--debug` aktiviert zusätzliche Debug-Ausgaben während der Generierung. Als Ergebnis dieses Aufrufs erzeugt der Generator den Logikbaustein `10001_my_module.hsl`.

Beschreibung der Befehlszeilenparameter

Parameter	Kurzform	Beschreibung	Beispielwert	Standardwert
<code>--source</code>	<code>-s</code>	Konfigurationsdatei	<code>my_config.json</code>	<code>config.xml</code>
<code>--target</code>	<code>-t</code>	Zielfile (überschreibt Vorgaben der Konfigurationsdatei)	<code>10001_my_module.hsl</code>	Konfigurationsdatei gibt den Namen vor
<code>--debug</code>	<code>-d</code>	Aktiviert Debug-Ausgaben		Debug-Ausgaben deaktiviert

## Konfigurationsdatei

Die Konfigurationsdatei beschreibt die grundlegenden Eigenschaften des Logikbaustein wie z.B. Name, ID, Kategorie, Anzahl, Bezeichnungen und Typen von Ein- und Ausgängen usw.

Die Konfigurationsdatei kann in den Dateiformaten JSON oder XML vorliegen. Beide bieten den gleichen Umfang an Parametern bzw. Attributen zur Erstellung des HSL3 Logikbaustein.

Attribute der Konfigurationsdatei

### module

Definition eines einzelnen Logik-Bausteins.

#### category

Baustein-Kategorie für den GLE (Grafischer Logikeditor) im Experte.

#### context

Kontext, in dem der Baustein definiert ist. Siehe auch: Threading.

#### id

ID des Logikbaustein

#### name

Name im GLE (Grafischer Logik-Editor des Experten)

#### version

Versionsbezeichnung des Bausteins

#### inputs

Eingänge des Logikbaustein

#### outputs

Ausgänge des Logikbaustein

**stores**

Remanente Variablen des Logikbaustein

**timers**

Timer des Logikbaustein

**scripts**

Python-Quellcode-Dateien des Logikbaustein (Pfadangaben relativ zur Konfigurationsdatei)

**translations**

Enthält alle im GLE sichtbaren Texte des Bausteins. Pro Sprache muss eine translation angegeben werden.

Sind die Texte in der eingestellten Sprache nicht vorhanden, verwendet der Experte die bei den module, input und output Tags definierten Texte.

**input**

Definiert einen Eingang des Logikbaustein.

**type**

Eingangstyp: *number*, *string*, *base\_path* oder *destination\_port*

number: Numerischer Wert

string: Alphanumerischer Wert / Zeichenkette

base\_path: Port, an dem der Baustein lauschen muss, wenn er von außen über den Basis-Pfad erreicht werden soll.

destination\_port: Basis-Pfad, über den alle Aufrufe an den Baustein weitergeleitet werden. Wird der HomeServer URL angehängt.

**init\_value**

Wert, mit dem der Eingang initialisiert wird.

**label**

Bezeichnung des Eingangs, wie er im GLE angezeigt wird.

**identifizier**

Bezeichnung des Eingangs, wie er im Python-Quellcode verwendet wird.

**output**

Definiert einen Ausgang des Logikbaustein.

**type**

Ausgangstyp: *number* oder *string*

number: Numerischer Wert

string: Alphanumerischer Wert / Zeichenkette

**init\_value**

Wert, mit dem der Ausgang initialisiert wird.

**label**

Bezeichnung des Ausgangs, wie er im GLE angezeigt wird.

**identifizier**

Bezeichnung des Ausgangs, wie er im Python-Quellcode verwendet wird.

**store**

Definiert eine remanente Variable des Logikbaustein.

**type**

Typ: *number* oder *string*

number: Numerischer Wert

string: Alphanumerischer Wert / Zeichenkette

**init\_value**

Wert, mit dem die remanente Variable initialisiert wird.

**identifizier**

Bezeichnung der remanenten Variable, wie er im Python-Quellcode verwendet wird.

**timer**

Definiert einen Timer, der vom Logikbaustein verwendet wird.

**identifizier**

Bezeichnung des Timer, wie er im Python-Quellcode verwendet wird.

**script**

Angabe der Python-Quellcode-Dateien, die vom Logikbaustein verwendet werden.

**filename**

Python-Quellcode-Datei (Pfadangaben relativ zur Konfigurationsdatei).

Der Dateiname muss das Prefix `hs13_<ID>` haben.

Z.B. `hs13_10001_my_module_src.py`

**folder**

Verzeichnis mit Python-Quellcode-Dateien (Pfadangaben relativ zur Konfigurationsdatei).

Es werden alle Dateien im Verzeichnis berücksichtigt,

deren Dateiname das Prefix `hs13_<ID>` hat.

**translation**

Definiert die Texte eines Bausteins für eine Sprache.

**language**

Sprachkürzel. Gültig sind alle Sprachen, die der Experte unterstützt.

**name**

Name im GLE (Grafischer Logik-Editor des Experten) in der Sprache *language*.

**category**

Baustein-Kategorie für den GLE (Grafischer Logikeditor) im Experte in der Sprache *language*.

**translation\_inputs**

Enthält die Namen für alle Eingänge des Bausteins in der Sprache *language*.

**translation\_outputs**

Enthält die Namen für alle Eingänge des Bausteins in der Sprache *language*.

**translation\_input**

Name für einen Eingang in der Sprache *language*.

**label**

Bezeichnung des Ausgangs in der Sprache *language*, wie er im GLE angezeigt wird.

**translation\_output**

Name für einen Ausgang in der Sprache *language*.

**label**

Bezeichnung des Ausgangs in der Sprache *language*, wie er im GLE angezeigt wird.

## HSL 3 - Kommunikation zwischen Logikbaustein-Instanzen

### Methoden im Framework

Das HSL 3 Framework bietet die Möglichkeit, dass Logikbausteine auf Instanzen anderer Bausteine zugreifen können.

Dies ermöglicht die Kommunikation von Logikbaustein-Instanzen untereinander.

Im Framework ist eine Methoden vorhanden, um eine Instanz eines Logikbausteins zu erhalten:

`get_instance(instance_id)`

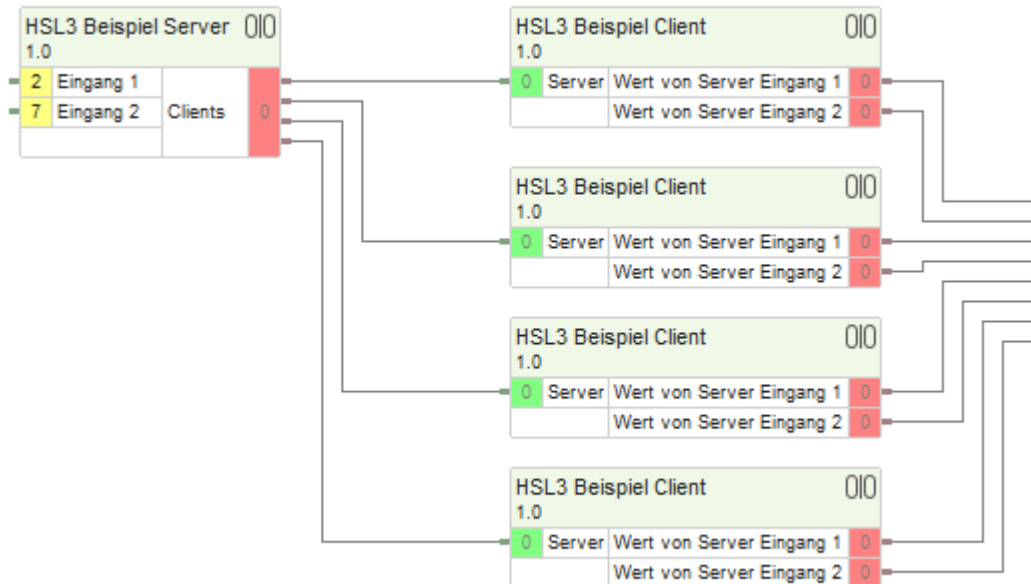
Als Ergebnis gibt die Methoden die zu `instance_id` zugehörige Logikbaustein-Instanz zurück (oder "None", wenn die `instance_id` ungültig ist).

Die `instance_id` ist eine vom Framework zur Laufzeit des HomeServers eindeutig vergebene ID für die jeweilige Instanz eines Bausteins. Sie stimmt nicht mit der fünfstelligen Baustein-ID (z.B. 19042) überein. Die `instance_id` (auch Laufzeit-ID genannt) kann über die Methode `get_instance_id()` ermittelt werden. Folgender Beispielcode würde die eigene Instanz (also `self`) zurückliefern:  
`self.fw.get_instance(self.fw.get_instance_id())`

Die Laufzeit-ID kann einfach über die Verdrahtung von Ein- und Ausgängen zwischen Logikbausteinen ausgetauscht und bekannt gemacht werden.

## Beispiel

Folgendes Beispiel zeigt die Kommunikation zwischen Logikbaustein-Instanzen:



Der Quellcode zu den Logikbausteinen, in diesem Beispiel, liegt dem Paket bei.

## Ablauf der Kommunikation

1. Der Logikbaustein "HSL3 Beispiel Server" fragt bei der Initialisierung mit der Methode `self.fw.get_instance_id` seine `instance_id` ab und schreibt diese auf seinen Ausgang.
2. Die Logikbausteine "HSL3 Beispiel Client" erhalten die `instance_id` auf ihren Eingängen und erhalten mit der Methode `self.fw.get_instance` die Instanz des Logikbausteins "HSL3 Beispiel Server".
3. Durch einen Timer angestoßen, fragen die Logikbausteine "HSL3 Beispiel Client" die Werte der Eingänge des Logikbausteins "HSL3 Beispiel Server" ab, indem sie direkt auf dessen Instanz zugreifen. Anschließend schreiben sie die abgefragten Werte auf ihre Ausgänge.

## Hinweise zur Migration von HSL2 zu HSL3

### Einführung

Die Einführung des HSL 2 Framework SDK erweiterte die Funktionalität des Gira HomeServer bzw. Gira FacilityServer ab Firmware-Version 4.5 um die Möglichkeit, den vollen Python-Sprachumfang in den HS-Logikbausteinen nutzen zu können.

Dazu wurden die Python Versionen 2.6 bzw. 2.7 unterstützt.

Seit der Firmware 4.13 besteht die Möglichkeit, mit Hilfe des HSL3 SDK eigene Logikmodule für den Gira HomeServer bzw. Gira FacilityServer zu entwickeln, die Python 3.9 verwenden.

Das HSL3 SDK ersetzt damit langfristig das HSL 2 SDK und es ist daher notwendig, mit HSL 2 umgesetzte Logikbausteine auf HSL 3 zu migrieren.

Was dabei zu beachten ist, wird hier erläutert.

### Unterschiede zwischen HSL 2 und HSL 3

Neben der schon angesprochenen Verwendung von Python 3 statt Python 2 gibt es einige weitere Unterschiede zwischen HSL 2 und HSL 3:

HSL2	HSL3
Verwendet Python 2.	Verwendet Python 3.
HSL 2 Framework ist Teil des Logikbausteins.	HSL 3 Framework ist Teil der Firmware.
Das Framework enthält u.a. Unterstützung für die Verschlüsselung und Netzwerkkommunikation.	Der Umfang des Frameworks ist auf das Wesentliche reduziert.
Vorgegebene Projektstruktur.	Projektstruktur kann individuell gestaltet werden.
Konfigurationsdatei für den Generator in XML.	Konfigurationsdatei für den Generator in JSON oder XML.
Quellcode des Logikbausteins muss aus einer Datei bestehen.	Quellcode des Logikbausteins kann aus mehreren Dateien bestehen.
Einbindung von eigenen Python Modulen über Import eingeschränkt per Generator möglich.	Einbindung von Python Modulen aktuell nicht vorgesehen.

### Tipps für die Migration von Logikbausteinen von HSL 2 zu HSL 3

Python 2 in Python 3 Quellcode konvertieren mit 2to3

Die Umstellung von Quellcode von Python 2 zu Python 3 ist nicht trivial, da sich Syntax, Standardbibliothek und Verhalten in einigen Kernbereichen verändert haben.



2to3 ist ein offizielles Tool aus dem Python Standardpaket (enthalten bis Python Version 3.12.10), welches den Quellcode von Python 2 nach Python 3 übersetzen kann.

Es arbeitet auf Basis vordefinierter Regeln, die typische Unterschiede zwischen beiden Sprachversionen erkennen und anpassen.

Das Tool eignet sich für die Konvertierung zu Python 3, da es systematisch und reproduzierbar vorgeht. Dennoch deckt es nicht alle Fälle ab: Besonders beim Umgang mit Strings und Bytefolge bleiben oft manuelle Nacharbeiten erforderlich. Für einen erfolgreichen Einsatz empfiehlt es sich, den konvertierten Code umfassend zu testen und die automatischen Anpassungen kritisch zu prüfen.

## Strings und Bytes

In Python 2 ist der Typ *str* eine Folge von Bytes. Er enthält keine Information über die Zeichenkodierung, sondern repräsentiert lediglich eine Bytefolge. Wenn man wirklich Text im Sinne von Unicode Zeichen darstellen möchte, muss man den Typ *unicode* verwenden. Solche Objekte lassen sich mit dem Präfix *u* anlegen. Umwandlungen zwischen *str* und *unicode* erfolgen immer explizit über *encode()* und *decode()*.

Das bedeutet: in Python 2 steht *str* für Bytes, während *unicode* für echten Text steht.

Mit Python 3 wurde dieses Modell konsequenter gestaltet. Dort ist *str* immer eine Folge von Unicode-Zeichen, also Text. Wenn man eine Bytefolge benötigt, gibt es dafür den Typ *bytes*, der mit einem Präfix *b* geschrieben wird. Eine direkte Vermischung von *str* und *bytes* ist nicht erlaubt. Wer Text in Bytes verwandeln möchte, muss explizit *encode()* aufrufen, und umgekehrt wird mit *decode()* aus Bytes wieder ein String.

Dies muss bei der Nutzung von Eingängen, Ausgängen und Speichervariablen von Logikbausteinen in HSL 3 beachtet werden. Die Strings, die von der Logik an den Logikbaustein weitergereicht werden oder die der Logikbaustein an die Logik ausgibt, müssen vom Typ *bytes* sein.

Beispiel für das Schreiben eines Textes vom Type *str* auf einen Ausgang:

```
self.fw.set_output("Ausgang Text",  
mein_text_str.encode(encoding="ascii", errors="ignore"))
```

## Verwendung von remanenten Speichervariablen

Soll dem Logikbaustein nach der Migration der Inhalt einer remanenten Speichervariablen zur Verfügung stehen, muss darauf geachtet werden, in HSL 3 zwei Dummy Variablen anzulegen.

Beispiel HSL 2:

```
<remanent_variables>
  <remanent_variable
const_name="slot_1">slot_1</remanent_variable>
</remanent_variables>
```

Beispiel HSL 3:

```
<stores>
  <store type="string" identifier="dummy1" init_value="" />
  <store type="string" identifier="dummy2" init_value="" />
  <store type="string" identifier="slot_1" init_value="" />
</stores>
```

Die Speichervariablen *dummy1* und *dummy2* sind nur Platzhalter ohne weitere Bedeutung für den Logikbaustein.

Wird ein neuer HSL3 Baustein erzeugt, dann ist das Anlegen der beiden *Dummys* nicht erforderlich. Dies dient nur der Kompatibilität zu den Remanentdaten der HSL2 Vorgänger Bausteine.