

Implementierung von CITE-Tests für die SIM2000ST-E

Praxisbericht (T2\_1000)

für die Prüfung zum

Bachelor of Sience

des Studiengangs Informatik  
Studienrichtung Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Jonas Angene

28.09.2020

Matrikelnummer 7931777

Kurs TINF19B3

Ausbildungsfirma SICK AG Waldkirch

Betreuer B. Eng Tobias Weckerle

Gutachter der Studienakademie Prof. Dr. Jürgen Vollmer

|  |
| --- |
|  |

**Erklärung**

Ich versichere hiermit, dass ich meinen Projektbericht (T2\_1000) über meine erste Praxisphase, mit dem Thema: Implementierung von CITE-Test für die SIM2000ST-E, bei der Firma SICK selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung der Ausbildungsstätte vorliegt.

Inhaltsverzeichnis

[Ehrenwörtliche Erklärung](#_Toc414952425)

[Sperrvermerk II](#_Toc414952428)

[Inhaltsverzeichnis III](#_Toc414952430)

[Abkürzungsverzeichnis IIV](#_Toc414952431)

[Abbildungsverzeichnis V](#_Toc414952432)

[1 Einleitung](#_Toc414952434)

[1.1 Vorwort](#_Toc414952435)

[1.2 SICK AG](#_Toc414952436)

[1.3 Global Business Center 05 2](#_Toc414952437)

[1.4 Organisation der GBC 05 & Mein Team 4](#_Toc414952438)

1.5 Aufgabenstellung 5

[2 Grundlagen 6](#_Toc414952439)

[2.1 Lua 6](#_Toc414952445)

[2.2 Sick AppSpace 10](#_Toc414952440)

[2.3 Continuous Integration Test Enviorment 12](#_Toc414952441)

[2.4 Sensor Integration Machine 13](#_Toc414952444)

[3 Grundlagen für die CITE-Tests 14](#_Toc414952446)

[3.1 Testaufbau 14](#_Toc414952447)

[3.2 Benötigte Applikationen 14](#_Toc414952448)

[3.3 Setup und Teardown für alle Test Cases 16](#_Toc414952447)

[3.4 Upload von Remoteapps 18](#_Toc414952448)

[4 Die CITE Test Cases 19](#_Toc414952449)

[4.1 Temperatur-Monitoring 19](#_Toc414952450)

[4.2 Serielle Kommunikation 20](#_Toc414952451)

[4.3 Digitale Ausgänge 25](#_Toc414952451)

[4.4 Encoder 2](#_Toc414952451)

[5 Resümee und Fazit 31](#_Toc414952452)

[Quellenverzeichnis VII](#_Toc414952455)

Abkürzungsverzeichnis

Global Business Center……………………………………………GBC

Continous Integration Test Enviorment…………………………..CITE

Sensor Integration Machine……………………………………….SIM

Device Under Test………………………………………………….DUT

Abbildungsverzeichnis

[Abbildung 1: SICK AG Hauptsitz in Waldkirch 2](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 2: Darstellung aller GBCs mit Bezeichnungen 3](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 3: Darstellung der Teams und Gruppen der GBC 05[4] 5](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 4: Typ Deklaration in Lua 6](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 5: Beispiel für Funktionen in Lua 7](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 6: funktionsweise von „*Tabeln*“ 7](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 7: if und else Beispiel 8](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 8: Schleifen in Lua 8](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 9: Geöffnete AppStudio Instanz [7] 11](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 10: Consolenfeed eines CITE-Tests (Beispiel TC\_EncoderTest) 12](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 11 & 12: rechts die SIM2000ST und links die Eco Variante [8] 13](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 13: Aufbau des Tests 14](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 14: Device Webseite 15](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 15: Setup-Funktion in der main.lua 17](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 16: TearDown-Funktion in der main.lua 17](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 17: Funktion die als Tempreature-Monitor-Test ausgeführt wird 19](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 18: Partity Dokumentation (trans & reciev = Parity-Einstellung) 21](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 19: Beispiel für Tabel in der TC\_SerialTest 22](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

Abbildung 20:[Beispiel für die Testmethode eines Seriellen Tests 22](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 21: ***setSettingOnHost***-Funktion 23](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 22: ***sendMessage***-Methode 23](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 23: Methode die beim Event ‚OnReceive‘ für den Host 25](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 24: setDigitalOut-Funktion in der TC\_DigitalOut.lua 26](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 25: Control-Flow des Decoders 28](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 26: ***controlDirection*** Funktion 29](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

[Abbildung 27: Beispiel für Race-Conditions des Encoders 30](file:///C:\Users\gerspachr\Documents\Word-Vorlage-Master\PA_BA_Arbeiten_Mastervorlage_V1.0.dotm#_Toc387246645)

# **Einleitung**

## **Vorwort**

Dieser Bericht thematisiert die Tätigkeiten meiner Praxisphase vom 03.08.2020 – 25.08.2020 und die Umsetzung meiner Aufgabenstellung in der „GBC 05 – Industrial Integration Space“. Dabei erhielt ich einen tiefen Einblick in die Abteilung so wie das Sick **C**ontinous **I**ntegration **T**est Enviorment kurz CITE. Der Bericht befasst sich mit der Umsetzung von CITE-Tests im Sick AppStudio unter der Verwendung der Skriptsprache Lua. Die CITE-Tests werden dabei benötig, um die Funktionalität des neuen erscheinenden Gerätes, die SIM2000ST-E, zu testen.

## **SICK AG**

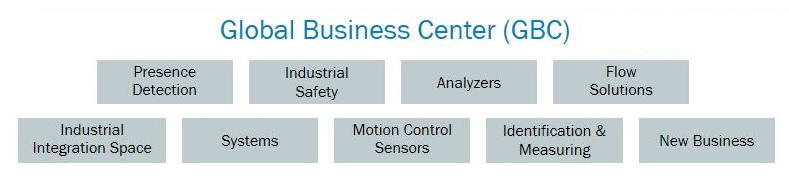
Bei der Sick AG handelt es sich um einen die weltweitführenden Hersteller von Sensoren und Sensorlösungen für ein weites Portfolio von Anwendungsgebieten. Dabei trägt die Firma mit ihren Produkten maßgeblich zur industriellen Automatisierungen, in den unterschiedlichsten Branchen, teil. Die Geschäftsfelder lassen sich dabei in die drei Teilsegmente Fabrik-, Logistik- und Prozessautomation. 1946 gründete, das mittlerweile in Waldkirch ansässige, Unternehmen Dr. E.h Erwin Sick in Varstetten bei München. Den wirtschaftlichen Durchbruch erlangte Sick dabei 1952 mit der Veröffentlichung des ersten Unfallschutz-Lichtvorhangs, der Start der Erfolgsgeschichte des Unternehmens. Heute Beschäftig das Unternehmen mehr als 10.000 Personen in über 80 Ländern Welt weit. Dabei erwirtschaftete das Unternehmen beispielsweise 1.8 Milliarden Euro (2019) was nur ein Indikator für den zunehmenden Erfolg des Unternehmens ist. [1][2]



Abbildung 1: SICK AG Hauptsitz in Waldkirch

## **Global Business Center 05**

Das Unternehmen lässt sich Tief schichtig gliedern und sehr weithingehend unterteilen. Dies resultiert vor allem an den unterschiedlichen Branchen die die Firma bedient. Das Unternehmen lässt sich dabei in sechs Bereiche unterteilen. Für das Verständnis des Praxisberichtes ist nur der Teilbereich vier von Relevanz. Dabei handelt es sich um die Global Business Center (GBCs). Die GBCs stehen dabei für die Produktion der Firma auch sie unterteilen sich in neun Aspekte und behandeln für sich die unterschiedlichen Sensorherstellungsbereiche die die Firma unterhält. [3]

Abbildung 2: Darstellung aller GBCs mit Bezeichnungen

(Oben von links GBC 1-4, Unten von Links GBC 5 – 9)

Die Tätigkeit meiner Praxisphase fand jedoch ausschließlich in der GBC05 – Industrial Integration Space statt. Die Aufgaben der GBC05 lassen sich grundsätzlich in 3 Bereiche gliedern:

1. Digital transformation

Das Hauptkonzept ist dabei die digitale Transformation. Bei der digitalen Transformation handelt es sich im weitesten Sinne um den sich stetig verändernden Prozess in einem Unternehmen der auf digitale Technologien zurückgeführt werden kann. Digital Technologien führen dabei zur Entwicklung neuerer digitaler Technologien die es dann zu integrieren gilt.

1. Vertical integration

Das zweite Bereich ist die vertikale Integration. Darunter fällt die vertikale Datenintegration vom Sensor in die Cloud, basierend auf dem bereits gesammelten Anwendungs-Know-how von Sick. Dabei werden bereits angewandte und standardisierte Lösungen.

1. Digital services & solutions advice

Der letzte Teilbereich nennt sich Digital Service & Lösungsberatung. Es handelt sich im vollem Maße um die Dienstleistungs- und Beratungsbereich der GBC 05. Es geht darum dem Kunden zu helfen die Herausforderung der Industrie-4.0, also eine sehr umfassende Digitalisierung der Produktion, zu meistern. Dazu werden flexible Datenstrukturen verwendet, so wie als auch dem Kunden in Beratungsgesprächen aktiv mit Wissen und Information geholfen. [3]

## **Organisation der GBC 05 & Mein Team**

Die Organisation der GBC05 ist recht weitläufig weswegen ich drauf kurz eingehen werde um zu es verständlich zu machen, wie die GBC arbeitet. Grundsätzlich ist die GBC in zwei Gruppen aufgeteilt den Product und Framework Buildern.

1. Framework Builder

Die Framework Builder schaffen dabei die Grundlage der zu vor erwähnten Vertical integration. Es gibt dabei verschiedene Team zuständig für die Cloud, Middleware, Software, Hardware sowie auch Design. Ein Produkt der Framework Builder ist beispielweise das Sick IntegrationSpace. Der Cloud betriebene und Web basierende digitale Service der Firma oder auch das Sick AppStudio, die IDE mit den Applikationen für alle programmierbaren Sensoren geschrieben werden können. AppStudio bindet eine zentrale Rolle in meinem Projekt und wird daher später detaillierter vorgestellt.

1. Product Builder

Die Product Builder auf der anderen Seite setzen sich zum einem aus den DCCs mit ihren lokalen sitzen in Asien Europa und den USA zusammen und auf der anderen Seite aus den Entwicklern. Die DCCs arbeiten eng mit dem Kunden zusammen und arbeiten mit diesen individuellen Lösungen heraus. Die Entwickler hingegen bedienen sich aus den Tools der Framework Builder und erstellen vom Kunden benötige Apps oder Geräte. Aufgeteilt sind die Entwickler daher in Application Builder und Device Builder. So ist die schon erwähnte SIM2000ST-E eins dieser Geräte bei dessen Entwicklung ich ein bisschen mithelfen durfte. Eingesetzt wurde ich in das Team der Device Builder, ein Team bestehend aus 5 Personen. [3]

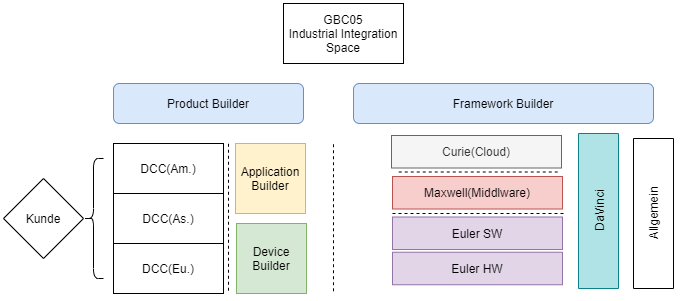
****

Abbildung 3: Darstellung der Teams und Gruppen der GBC 05[4]

## **Aufgabenstellung**

Aufgabe meiner Tätigkeit war es also wie schon erwähnt bzw. wie der Titel des Berichtes schon vermuten lässt das Schreiben von CITE-Tests für die SIM2000ST-E. Wie ich im Verlauf meines Aufenthalts erfahren habe wurde von der Abteilung bzw. Device Builder Team ein Azubi angefragt. Da es sich wie schon erwähnt bei dem Team um ein kleines fünf Mann Team handelt, die außerhalb der SIM2000ST-E auch andere Projekte haben als auch, dass sie verantwortlich sind für die Aktualität der Firmware, wie zum Beispiel die AppEngine, von anderen veröffentlichten Geräten. Das Team ist also momentan recht ausgelastet weswegen es Zeitliche Engpässe gib für die Implementierung von CITE-Tests. Das schon erwähnte und später noch explizit erklärte Test Szenario CITE ist jedoch unabdingbar für die Veröffentlichung eines Gerätes, da dort sämtliche Funktionen, so wie der Kunde sie benutzen will, getestet und verifiziert werden. Als Datum für die erste Veröffentlichung der SIM2000ST-E war Oktober bzw. Anfang November 2020 angesetzt. Das Gerät befindet sich momentan jedoch eher in einem unfertigen Teststadium. Es lässt sich eher von einem Prototyp sprechen. So existieren Bugs die es zu fixen gilt. Meine Aufgabe war es also nicht nur simple Testfälle zu schreiben, sondern Funktionalitäten weitestgehend auszureizen und eventuell auftretende Bugs zu dokumentieren bzw. erklären wie man diese reproduzieren kann.

# **Grundlagen**

## Lua

Lua ist eine „freie Software“ und wurde 1993 an der päpstlichen Universität von Rio de Janeiro entwickelt. Bei Lua handelt es sich um eine imperative Skriptsprache, diese Unterscheidet sich jedoch zu mir herkömmlich bekannten Sprachen wie Java, C++, C und C# [5]. So gibt es keine Typ Definitionen von variablen. Das Programm selbst legt den Type während der Laufzeit fest. Auch können Variablen mit werten eines anderen Typs fehlerlos überschrieben werden. Es lassen sich auch mehre Variablen in einer Zeile deklarieren.

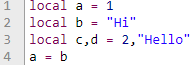


Abbildung 4: Typ Deklaration in Lua

Semikolons können gemacht werden um Befehl abzuschließen jedoch werden sie nicht benötigt und dienen nur dazu den Code übersichtlich zu halten für den Programmierer. Das Schlagwort ***local*** deklariert den „Lebenszyklus“ einer Variable. ***local*** legt fest das die Variable nur in dem Code Block existiert wo sie mit local deklariert wird. Wenn der Block endet wird die Variable gelöscht. Wie später in dem Code meiner Aufgabe zeigen werde macht es durch aus Sinn nicht alle Variablen als local zu deklarieren und diese auch über einen Codeblock weiterhin nutzen zu können. Bei Lua handelt sich direkt nicht um eine objektorientierte Programmiersprache. Es ist zwar möglich durch später erklärte tabels Instanzen zu erstellen die Klassen ähneln jedoch ist gibt es kein Schlagwort ***class*** oder des Weiteren.

### Funktionen

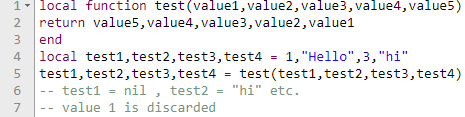
Funktionen erstellen sich durch das Schlagwort ***function*** drauffolgt in einer Klammer die erwarteten Parameter die beim Funktionsaufruf übergebenwerden. Besonders ist dabei, dass bei Lua bei einem Funktionsaufruf nicht alle Parameter gesetzt werden müssen. Es wird dadurch kein Fehler erzeugt. Der Parameter entspricht dann jedoch in der Funktion ***nil*** was nichts gleich kommt. Eine Funktion gibt alle Werte, die nach dem Schlagwort ***return*** stehen, zurück. Auch hier ist es egal ob alle Werte beim Funktionsaufruf zugeteilt werden. Gibt eine Funktion weniger Werte zurück als erwartet werden die Variablen ohne Wert ***nil*** zu geteilt. Auch Funktionen kann man durch ***local*** einen Existenzraum setzen.

Abbildung 5: Beispiel für Funktionen in Lua

### Tabel

Als Arrays benutzt Lua so genannte tabels diese sind im Endeffekt dynamische Arrays jedoch benötigen diese nicht nur zahlen als Index, sondern können auch Strings oder andere Typen benutzen. Tabels besitzen auch keinen explizit festgelegten Typ, sondern können die unterschiedlichsten Typen beherbergen auch Funktionen. Besitzt man nun einen tabel und speichert einen Wert an der Stelle „test“ so kann auf diese entweder so *tabel[test]* oder so *tabel.test* zu gegriffen werden. Lua Arrays beginnen standartmäßig mit eins und nicht mit null.

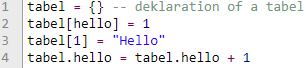


Abbildung 6: funktionsweise von „*Tabeln*“

### If & else

Von der Syntax ähnelt es der Programmiersprache Pascal. So bestehen If-Bedingungen aus dem Schlagwort ***if*** gefolgt von der Bedingung die der übersichtshalber geklammert werden kann. Die Bedingung endet mit dem Schlagwort ***then****,* wobei dann der Auszuführende Code folgt, wenn die Bedingung stimmt. Dieser endet mit dem Schlagwort ***end.*** [3] Sollte jedoch ein else-Fall verlangt werden so folgt nach dem Code erst das else mit seinem Code und dann das ***end.*** Auch existiert ein ifelse was einem else if in anderen Sprachen entspricht.

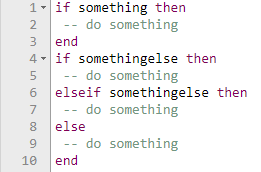


Abbildung 7: if und else Beispiel

### Schleifen

While- und For-schleifen verhalten sich dabei ähnlich. So folgt auch nach dem Schlagwort ***for*** oder ***while*** die Bedingung, diese endet hier jedoch mit ***do.*** Drauf folgt dann wie schon bei der if-Bedingung der auszuführende Code der mit ***end*** endet. Für For-Schleifen gibt es natürlich auch die klassiche Funktionsweise mit Inkrementieren. Dabei folgt nach dem ***for*** die Variable um die es sich handelt. Dieser kann auch direkt ein Wert zu geteilt werden. Drauf folgt getrennt durch ein Komma die Bedingung die, die Schleife stoppt. Hier kann auch eine normale Zahl eingesetzt werden, dies entspricht dann einem == Wert. Danach folgt wieder ein Komma und die zahl um die inkrementiert werden soll pro Schleifendurchlauf. Interessant wird es, wenn man einen Tabel hat und durch dieses Iterieren will, jedoch Strings als Indexe Genomen hat. Hier bietet Lua di Möglichkeit im zweiten for-Statement „index in ipairs(tabel)“ das in jedem Schleifendurchlauf „index“ einem Index Wert zugeteilt wird. Die Schleife endet, wenn alle Indexe einmal deklariert wurden.

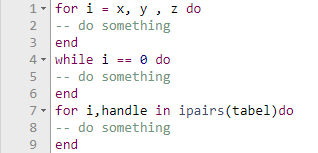


Abbildung 8: Schleifen in Lua

### ENUMs und Handle

Bei ENUMs handelt es sich in meinem Fall meistens um String Werte, die angeben werden um zu sehen welche String Wert übergeben werden müssen bei einem Funktionsaufruf. Ein ENUM kein beispielsweise der Port eines Gerätes sein („SER1“ für den seriellen Port1).

Handle sind Integer Werte die für ein Objekt stehen. Wie ich später zeigen werde gibt es Funktionen mit denen man Objekte erstellen kann, die sogenannten CROWNs. Ein Handle ist dabei meistens der Rückgabewert einer ***.create()*** Funktion. Beispielsweise für Encoder: Encoder.create(„ENC1“) gibt den handle für einen Encoder zurück. „ENC1“ ist dabei ein ENUM-Wert. Ein Handle repräsentiert die Instanz eines Objekts und wird benötigt um andere Funktionen des Objekts zu benutzen. (bspw. Encoder.getTicksPerSecond(EncoderHandle)).

### Events

Events beschreiben passierbare Vorkommnisse, die vordefiniert sind. Dabei werden diese mit Funktionen gelinkt die beim Auftreten eines bestimmten Events ausgeführt werden. So ist zum Beispiel der Aufruf einer Main–Funktion das Resultat des Events „Engine.onStarted“, was beim Skriptstart ausgelöst wird.

Events müssen dafür im Skript registrieren. Dies geschieht mit

***Script.register(„ExistirendesEvent“,AuszuführendeFunktion)***.

### Warum Lua?

Lua ist die Sprache die AppStudio verwendet und mit der im Enddefekt alle Sensoren programmierbar sind. Es wurde Lua gewählt vor allem wegen dem kleinen Footprint bezogen auf die Ressourcen. So sind Lua Scripts recht klein, schnell und leicht Erweiterbar perfekt für embedded Systems. Auch kann der der Interpreter über eine C-Bibliothek angesprochen werden. So können Programme sowohl Lua als auch C-funktionen wie auch C++-funktionen nutzen. Heutzutage ist es jedoch denkbar das man eher Python benutzen würde da Speicherplatz als recourse nicht das Problem ist [8].

## **SICK AppSpace**

Das Eco-System AppSpace ist ein Software bundel, dass es einem ermöglicht einem SensorApps für programmierbare Sick Geräte zu Entwickeln. Es besteht aus zwei Softwarekomponenten für den Kunden. Einmal das AppStudio und der AppManager. Zum anderen besteht es aus der AppEngine die das Gerät programmierbar macht und in das Eco-System einbindet.

### **AppEngine**

Die Sick AppEngine bildet die Basis für AppStudio. Es ist im Endeffekt eine große Bibliothek an allen Implementierten Funktionen und Algorithmen die jemals für programmierbare Sensoren geschrieben wurden. Die AppEngine ist dabei Geräte spezifisch und wird auf das entsprechende Gerät zugeschnitten. Sie fügt tabel hinzu die Common Reusable Objects Wired by Name (CROWN) in ihnen sind Parameter und Funktionen gespeichert um Sensorspezifisch zu agieren. Eine CROWN die ich später Benutzen werde ist die SensorNetwork-Cown in ihr sind beispielsweise sämtliche Funktionen wie auch ENUMs enthalten. Funktionen sind dabei meistens keine Lua-Funktionen. Da es sich bei Lua um eine Skriptsprache handelt, die der Interpreter in Echtzeit bearbeiten müsste, wird für die meisten Algorithmen und auch Funktionen C++ benutzt um die Performance zu erhöhen [6].

Da es auch CROWNs gibt mit denen man tiefgreifend in ein Gerät eingreifen könnte gibt es sogenannte „Hidden-CROWNs“ diese werden in der Entwicklungsumgebung, nicht wie die anderen, blau angezeigt und erscheinen dem Benutzer nicht, wenn er die automatische Vervollständigung benutzen will. Auch gibt es sogenannte „Experimental-CROWNs“ diese erscheinen dem Benutzer als Fehler, wobei das Skript sich trotzdem ausführen lässt. Dies sind CROWNs die noch im Testphase befinden. Erwirbt der Kunde ein Gerät mit irgendeiner AppEngine-Version so befinden sich im Regelfall keine „Experimental-CROWNs“ in ihr [6].

### **AppStudio**

AppStudio ist eine eigens von Sick entwickelte Integriere Entwicklungsumgebung (IDE) um eigene Software in Form von Applikationen für programmierbare Sensoren zu entwickeln. Verwaltet und entwickelt wird sie auch von der GBC 05. Wie schon vorgestellt Benutz AppStudio als Programmiersprache Lua mit denen durch die AppEngine hinzugefügten CROWNs. Dabei entfallen auch ein paar Standartbibliotheken von Lua wie zum Beispiel der Konsolen Input, da dieser für Laufende Applikationen keine Relevanz hat. Um AppStudio erfolgreich zu starten benötigt man eine AppStudio-Lizenz. Es wird jedoch kein Gerät benötigt um zu programmieren, da AppStudio eine große Auswahl an verschiedensten Emulatoren bietet um Test-Applikationen zu schreiben.

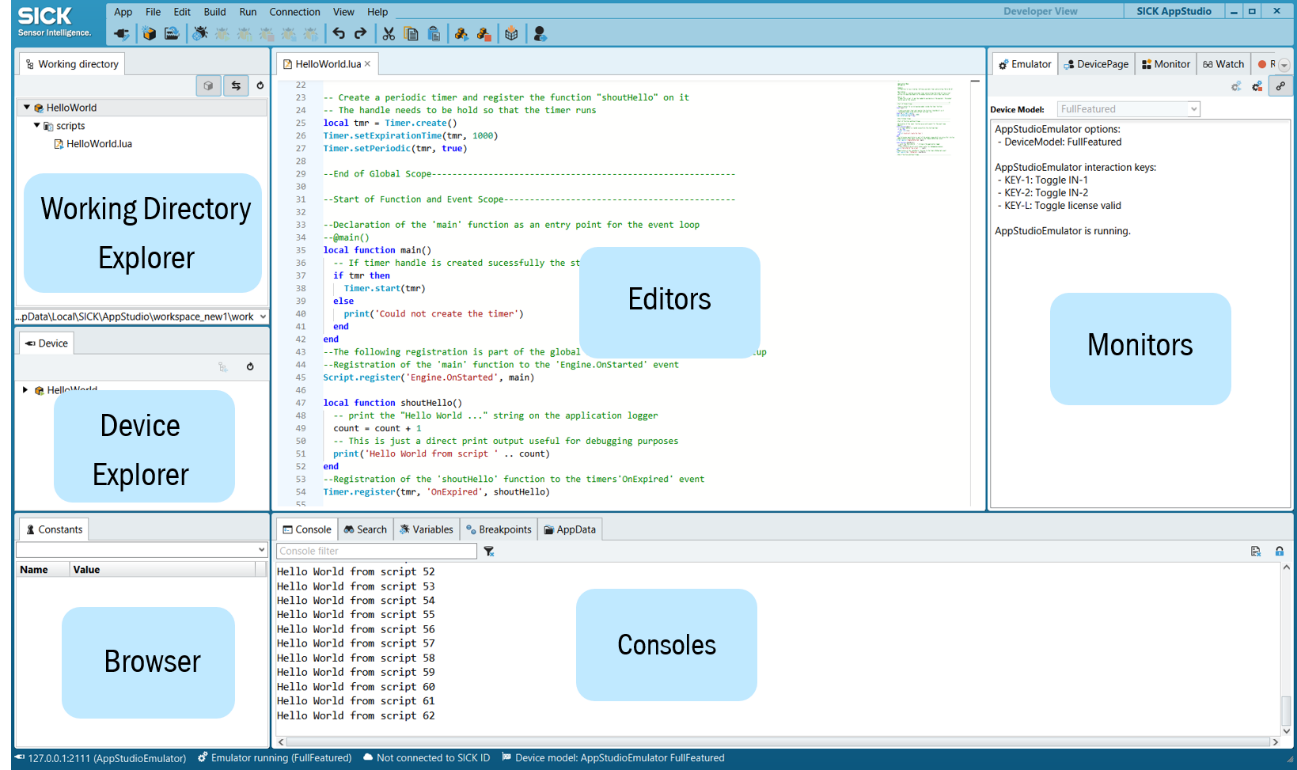


Abbildung 9: Geöffnete AppStudio Instanz [7]

### AppManager

Der AppManager ist ein Verwaltungs-Tool für all seine programmierbaren Sick-Geräte. Es ermöglicht es, bereits geschriebene Applikationen auszutauschen und auf seinem Gerät zu installieren. Es existiert bereits ein breiter AppPool an SensorApps um Standartlösungen abzudecken. Auch können hier Geräte aktualisiert und Back-ups gespeichert werden. Mit dem AppManager lassen sich jedoch nicht alle Installierten Applikationen des Gerätes anzeigen. So werden Systemrelevante Apps versteckt.

## **Continuous Integration Test Environment (CITE)**

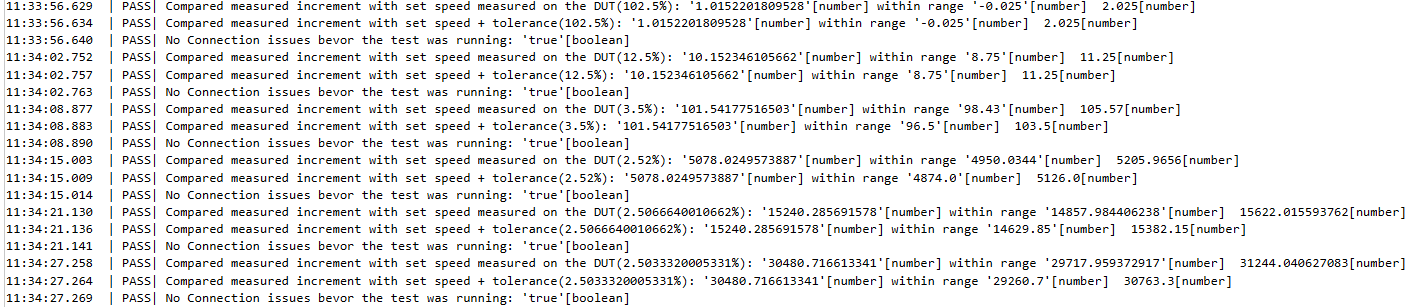
Bei CITE handelt es um Integrationstest für Ecosysteme der AppEngine. Dabei ist es nicht wie einfache Unittests dafür gedacht die Funktionsweisen von CROWNs zu Bewerkstelligen. CITE ist dafür gedacht Gerätespezifisch simple Funktionstest zu machen, die einer Kunden App ähneln um zu verifizieren, dass das Gerät alles leisten kann was es verspricht. CITE-Tests sind zwar Sick intern, werden jedoch nicht nur von der GBC05 benutzt. Auch die GBC06 testet beispielweise Lösungsaufbauten für Kunden mit CITE. CITE ist zwar eine auch eine AppEngine CROWN jedoch ist das eher die API um mit der eigentlichen CITE des Geräts zu kommunizieren. Bei CITE handelt es sich eigentlich um eine dieser versteckten Apps die auf dem Gerät vom Werk aus installiert sind. Bei einem CITE-Test gibt es immer einen Host, also ein Gerät auf dem CITE installiert (Beispielsweise die SIMs). Das zu testende Gerät wird so an den Host angeschlossen, so dass dieser mit seinen TestCase-Skripts gezielt Tests starten kann. In einem Script selbst finden mit der Funktion ***CITE.check()*** überprüfungen statt. Als Parameter kann entweder ein Boolean und eine Test Nachricht übergeben werden oder auch ein Wert, ein Vergleichsoperator als String und ein Weiterer Wert. Stimmen die Werte überein gelingt der Check ansonsten schlägt er fehl. Erfolgreiche Checks und nicht erfolgreiche Checks werden zusammengezählt und am Ende ausgegeben. Dabei ist wichtig das man dem Check eine aussagekräftige Nachricht bzw. Beschreibung in Form eines Textes mitgibt. So kann die Fehlerursache besser festgestellt werden.

Abbildung 10: Consolenfeed eines CITE-Tests (Beispiel TC\_EncoderTest)

## **Sensor Integration Machine (SIM)**

Bei der Sim handelt es um eine Produktfamilie die von der GBC05 produziert wird, sie dient zu Applikationslösung. So können mehrere Sensoren an eine SIM angeschlossen werden, die diese dann verwaltet oder über Apps gesteuert bestimmte Werte bearbeitet oder Algorithmen mit empfangenen Daten ausführt und diese auswertet. Sie sorg im weitesten Sinne das Sensoren mit einander kommunizieren können, im Kontext der Industrie 4.0. Das Portfolio der verschiedenen SIM bildet sich beispielweise aus der Rechenleistung oder der Anzahl an Anschlüssen. In meinem Einsatz hatte ich vor allem mit der 2000er Reihe zu tun. So besteht mein Testaufbau aus einer SIM2000ST als Host und einer SIM2000ST-E. Die SIM2000ST ist ein bereits veröffentlichtes Produkt der GBC05 und wird auch regelmäßig verkauft. SIM2000ST-E auch genannt SIM2000-Eco ist eine noch unveröffentlichte Prototype in der Entwicklung. Da die SIM2000ST schon ein bisschen in die Jahre gekommen ist bietet die ECO ein leistungsstärkeres Update (Faktor 2) [?], mit den neusten Komponenten. Es entfallen dabei jedoch auch Funktionen der SIM2000ST. Dafür ist ECO mit halben Preis geplant (Eine SIM2000ST Gen1 kostet ungefähr 3200€). [8][9]

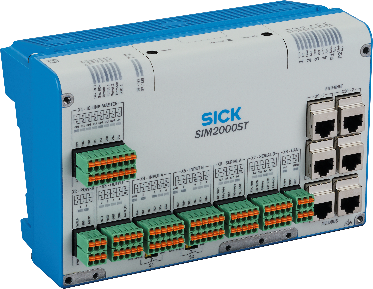
 

Abbildung 11 & 12: rechts die SIM2000ST und links die Eco Variante [8]

# **Grundlagen für die CITE-Tests**

In diesem Kapitel werden Grundlagen erläutert die der Implementierung Vorausgesetzt werden muss. Diese beziehen sich explizit auf den Test für die SIM2000-Econice

## **Testaufbau**

Im finalen Testaufbau sind die Anschlüsse wie folgt verbunden:

* Serial: Port1 des Host mit Port1 des DUT usw.
* Digital: Out1 des DUT mir In1 des Hosts usw.
* Encoder: IO-link 1 des Hosts als Decoder mit dem Encoder 1 des DUT

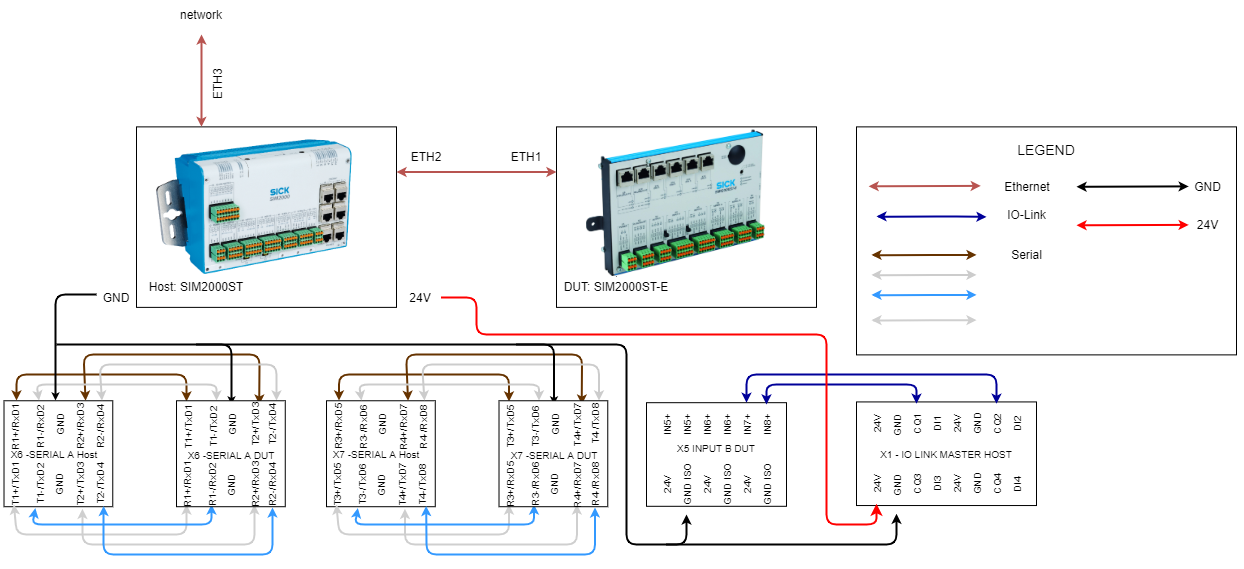


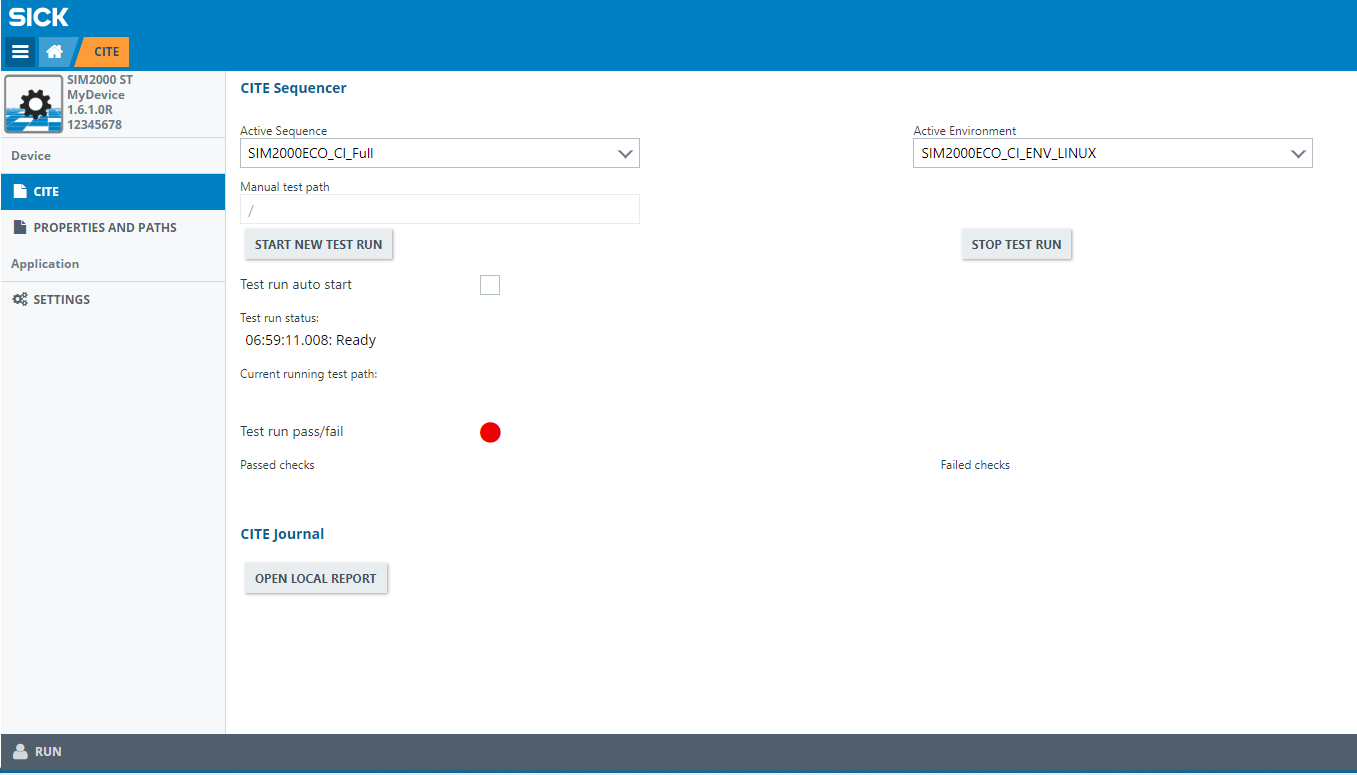
Abbildung 13: Aufbau des Tests

## **Benötigte Applikationen**

Um die Test Cases schreiben zu können bekam ich vor implementierte Applikationen: ENV\_Device, DUT\_Libary, SEQ\_Device sowie die TC\_Device. Diese entsprachen dem groben Gerüst meines zu implementierenden CITE-Tests und enthielten vor allem die Setup Funktionen und Bedingungen für mein **D**evice **U**nder **T**est (DUT).

### **DUT\_Libary**

In dieser Applikation wird die AppSpace-Web Seite erstellt, die man später entweder in AppStudio oder im Browser unter dem Local-Host einsehen kann.

Abbildung 14: Device Webseite

Dort startet man Den CITE-Test und sieht seinen Ablauf mit erfolgreichen und fehlgeschlagenen Tests ein. Die App enthält drei Geräte Setups zum einen das DUT, ein TemplatDevice und ein DebugDevice. Das TemplatDevice ist dabei ein Emulator. Für den CITE-Test ist jedoch nur das DUT von Relevanz, mit der CROWN ***CITE.DUT.Defenition.setProperty(HandleDesDUTs,NameDerProperty,PropertyWert)***

werden hier vor allem Werte für Einstellung gesetzt die dann mit ***CITE.DUT.Definition.getProperty(HandleDesDUTs,NameDerProperty)***

wieder returned werden können. Hier werden vor allem eingestellt unter welcher IP der Host den DUT erreichen kann wie auch Standartpasswörter und Standartpfade für Dateien die während dem Test benötigt werden.

### **ENV\_Device**

Diese App ähnelt der der DUT\_Lirbary stark, hier werden nur nicht die DUT Einstellung mit ***setProperty*** gesetzt, sondern die des Der Environment CROWN (***CITE.Env***). Auch sind hier wieder Debug-Device und der Emulator dabei.

### **SEQ\_Device**

In dieser App wird die Sequenzen implementiert die später auf der AppSpace Seite eingestellt werden können. Dabei haben diese Sequenzen unterschiedlichen Umfang was die Anzahl an Tests angeht. Der Pfad eines Testfalls wird hier mit

***CITE.Sequencer.scheduleTestPath(sequenzHandle,TestPfad)***

der Sequenz hinzugefügt.

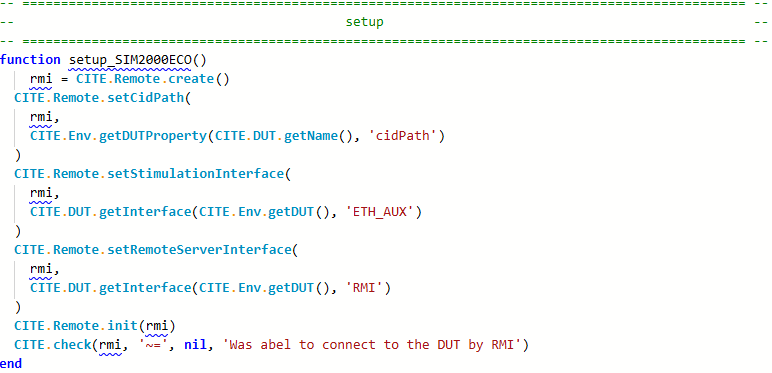
### **TC\_Device**

Hier werden die Test Cases hinzugefügt. Die Test Cases liegen hier mit all ihren Methoden als Lua-Skript vor. Testfälle werden hinzugefügt mit ***CITE.Sequencer.addTest(TestPfad,TestName,TestMethode)****.*

Auch gibt es die Funktionen ***addSetup*** und ***addTeardown*** um jeweils Methoden hinzuzufügen die vor dem Test und nach dem Test ausgeführt werden sollen. Auch im Setup oder Teardown werden CITE-Checks ausgeführt. Jedoch wird ein Test gar nicht erst ausgeführt, wenn das Setup fehlschlägt oder gilt komplett als Fehlgeschlagen, wenn das Teardown fehlschläg.

## **Setup und Teardown für alle Test Cases**

Zunehmend alle Test Cases benötigen ein und dasselbe Handle, das alle gleich in dem jeweiligen Test Setup erstellen müssten. Um redundanten Code zu vermeiden wird diese in dem main.lua Skript der TC\_Device App als Globale Variablen erstellt. Dies ist möglich da das Main-Skript mit ***require(Testcases.TC\_TestCaseName)*** den Inhalt der anderen Test Case Skripte einfach an die Stelle kopiert. Das handelt(***rmi***) ist die Instanz der **R**emote **M**ethod **I**nvocation (RMI) – Connection. Das Handle wird benötigt um Methoden vom Host aus auf dem DUT auszuführen, mit ***CITE.Remote.invokeMethod(rmiHandle,MethodName,EventuParameter).*** Diese Methode wird ausschließlich von allen Test Cases benötigt, da so Messwerte bzw. Einstellung leicht vom Host aus eingelesen bzw. ausgeführt werden können.

Abbildung 15: Setup-Funktion in der main.lua

Die Teardown-Methode zerstört den Handle damit, bei erneuten Teststart der Handle wieder erstellt werden kann. Außerdem werden beim Teardown alle Apps, die während dem Test auf das DUT geladen wurden, wieder gelöscht.

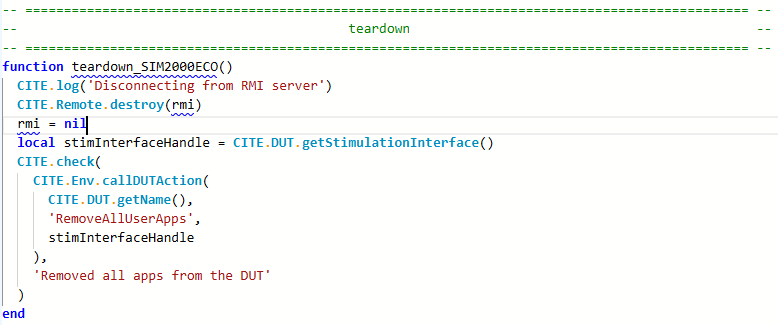


Abbildung 16: TearDown-Funktion in der main.lua

In der Main befindet sich außerdem die ***SetupRMIConnection***-Methode diese wird benötigt da sich die RMI-Connection, beim Uploaden von Apps vom Host auf das DUT, schließt.

## **Upload von Remoteapps**

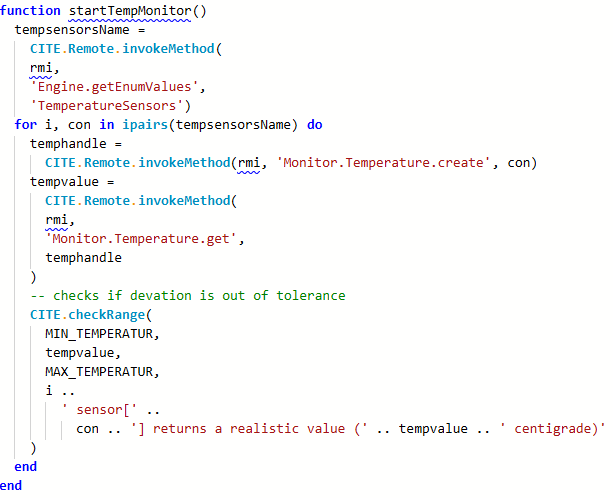
Da es nicht immer ausreicht Methoden auf dem DUT zu invoken, benötigt man die Möglichkeit auch eigene Apps auf das DUT zu laden. Dies ist ein normales Prozedere bei anderen CITE Test, weshalb ich die dazu benötigten Apps in meinen Test integriert haben. Da es sich dabei nicht um meinen eigenen Code handelt werde ich dieses Tool deshalb unerklärt lassen. Soweit sei jedoch gesagt, dass die Remoteapps im resources-Ordner der TC\_Device App abgelegt werden, um sie dann von da aus als „SickArchiver“-Datei auf das DUT zuladen. Dieser Vorgang findet immer im Setup eines Testes statt, der eine Remoteapp benötigt. Die eigenen Methoden müssen „geserved“ werden und können dann ganz normal Invoked werden. Es wird eine „eigene“ CROWN auf dem gerät erstellt.

# **Die CITE Test Cases**

Hier wird nun auf die explizite Umsetzung und Implementierung der Test Cases eingegangen. Die Skripte liegen Testcases-Ordner der TC-Device App und sind wie folgt benannt: TC\_TestName.

## **Temperatur-Monitoring**

Der TC\_TemperatureMonitor Test Case prüft die Werte, der in der Eco verbauten Temperatursensoren. Dafür wird der Tabel ***tempsensorName*** erstellt in dem alle Namen der Temperatursensoren gespeichert werden. Durch diesen Tabel wird mit einer For-Schleife für jeden Namen durch iteriert. In der For-Schleife selbst wird mit dem Namen eines Temperatursensors, der handle erstellt mit dem dann die Temperatur mit ***Monitor.Tempreature.get(handle)*** abgefragt wird. Mit ***CITE.checkRange*** wird überprüft ob der Wert zum einem nicht **nil** und zum anderen nicht unrealistisch ist. Unrealistisch wäre er dabei bei Temperaturen unter 0°C und über 90°C. Ist dies beides nicht der Fall war der Check erfolgreich, ansonsten misslingt er.

Abbildung 17: Funktion die als Tempreature-Monitor-Test ausgeführt wird

## **Serielle Kommunikation**

### **Umsetzung**

Bei der seriellen Kommunikation fällt der Test größer aus, da es hier viel mehr Einstellungen gibt, die man einzeln Testen kann. Auch reicht es hier nicht aus, Methoden auf dem DUT zu invoken. Für den Test wird eine Remoteapp benötigt. Vor ab will ich nochmal erwähnen, dass alle seriellen Ports des Hosts mit ihrem äquivalent auf dem DUT verbunden sind (beispielweise: SER1 HOST mit SER1 DUT etc). Die Theorie des Test Cases ist das der Host per serieller Kommunikation zwei Nachrichten verschickt. Diese sehen beispielweise wie folgt aus:

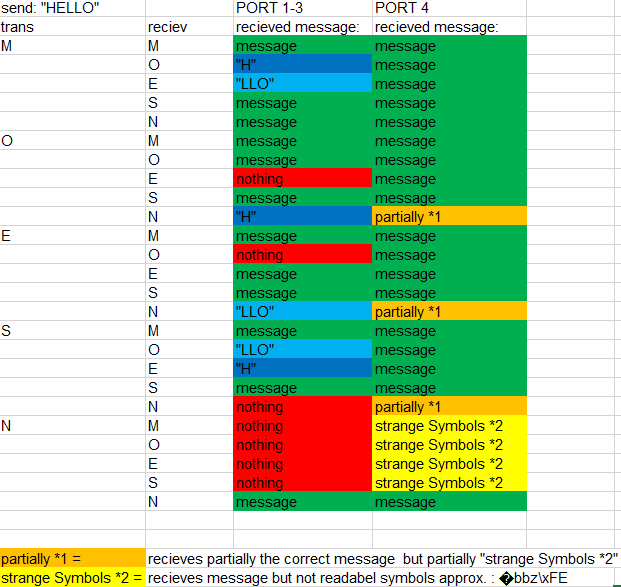
/BAU/9600

/POR/SER1

Die erste Nachricht stellt die Einstellung die momentan getestet wird, mit dem entsprechenden aktuellen Wert da und die zweite den Port, mit dem die Nachricht gesendet wurde. Alle Tests laufen wie folgt ab: Der Host erstellt den PORT mit dem entsprechenden Handle für sich und setzt die Einstellung auf den zu testenden Wert das gleiche invoked er für das DUT. Nun schickt der HOST die beiden Nachrichten hintereinander: zuerst die Einstellung, dann den Port. Das DUT erwartet zuerst die Einstellung. Diese wird in einem lokalen Tabel gespeichert: zum Beispiel ***setting[/BAU/] = 9600.*** Drauf hin wartet es auf den Port. Nach dem der Host den Port schickt zerstört er den Port-Handle des DUT. Erhält das DUT nun den Port erstellt es mit der Port-Information das Handle und setzt die Einstellung, die es erhalten hat, mit dem Wert aus dem Tabel. Danach schickt es eine Nachricht an den Host zurück, die wie bspw. folgt aussieht: SER1/BAU/9600. Der Host erwartet genau die Nachricht und dokumentiert das Erhalten dieser in einem Tabel. Deshalb ist es wichtig das die Nachrichten eindeutig sind da es sonst zu Überlagerungen kommt. So gibt es beispielweise viele Einstellungen mit true und false weshalb der Einstellungsname benötigt wird! Da für jeden Port dieselbe Einstellung nur einmalig getestet wird ist die Alleinstellung bewerkstelligt.

### **Das Auftreten von Bugs**

Während dem Testen fand ich heraus, dass die Ports 1-3 ein anderes Verhalten hatte als Port 4. So kamen zum einen bei einer unterschiedlichen Einstellung der „Parity“ von Host und DUT unterschiedliche Sachen noch an.

Abbildung 18: Partity Dokumentation (trans & reciev = Parity-Einstellung)

Für die Einstellung des „DataBit“ (Integer, entweder 7 oder 8) kam bei der Einstellung „7“, auf beiden Seiten, nur auf Port 4 die verschickte Nachricht an.

Diese Anomalie wurde daher von mir in einem Jira ticket dokumentiert. Dabei kam mir der Gedanke zu testen ob Nachrichten, die nicht ankommen sollten, wirklich nicht ankommen was ich drauf hin umsetzte. Dazu musste ich die Nachrichten die gesendet wird so modifizieren um die Alleinstellung zu garantieren. Dies geschieht dadurch, dass die Nachrichten mit einem „fail“ am Ende gekennzeichnet wird, wenn sie nicht ankommen sollte.

### **Implementierung**

Wie gesagt ist der Test recht redundant, weshalb ich Beispielshaft die „DataBit“ -Test genauer erläutern werde. Die Möglichkeiten verschiedener Einstellungen wird in einem entsprechenden Tabel festgehalten.

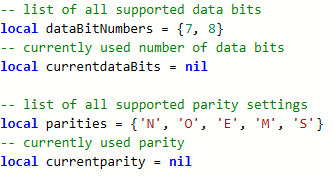


Abbildung 19: Beispiel für Tabel in der TC\_SerialTest

Die Variable ***currenttestsetting*** ist ein String, in dem der Name der aktuellen Einstellung gespeichert wird. Mittels zweier ineinander verschachtelter For-Schleifen werden alle Kombinationsmöglichkeiten durch iteriert.

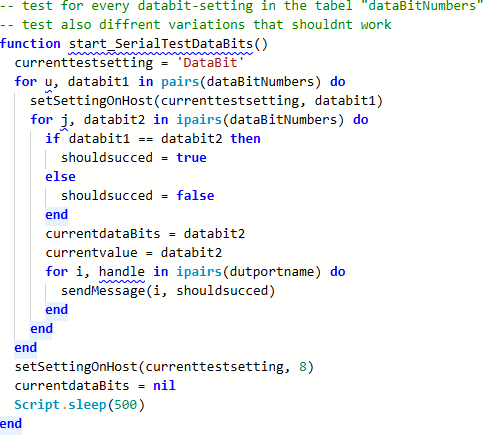


Abbildung 20: Beispiel für die Testmethode eines Seriellen Tests

(hier für das „DataBit“)

Die erste Schleife gilt dabei für die Einstellung des Hosts, die gleich gesetzt wird mit ***setSettingOnHost*** und die zweite für das DUT.

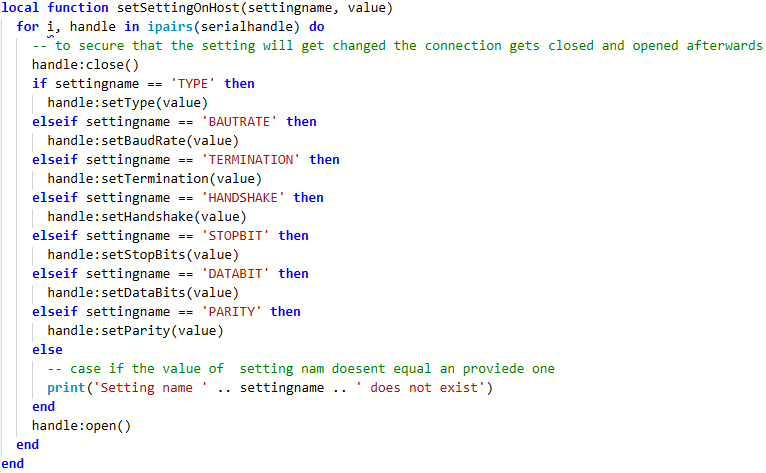
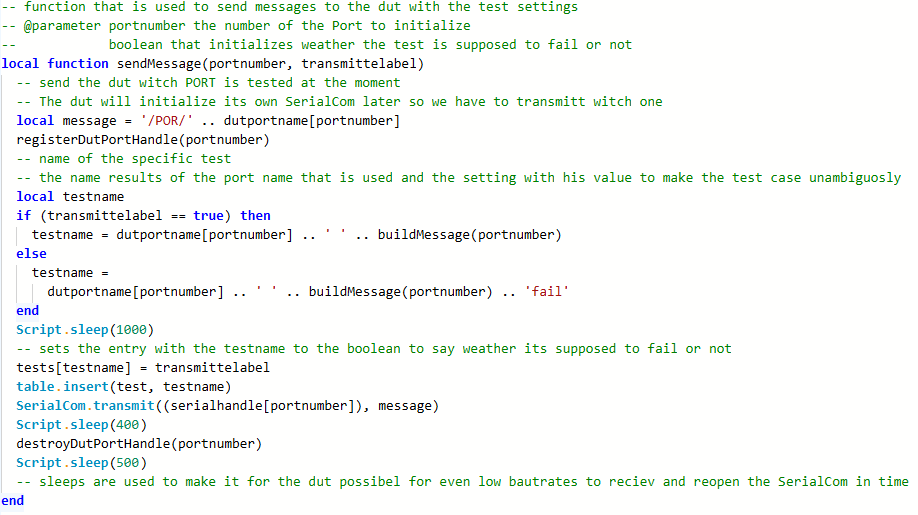


Abbildung 21: ***setSettingOnHost***-Funktion

Nach der zweiten For-schleife wir **shouldsucced** auf true wenn beide Werte gleich sind gesetzt, wenn nicht auf false. Was entweder einem Test entspricht, der gelingen soll oder nicht. Jeder Port wird daraufhin mit dem Test-Szenario getestet, dazu wird die Funktion ***sendMessage*** verwendet.

Abbildung 22: ***sendMessage***-Methode

Die Methode ***buildMessage*** sendet die Nachricht mit der Information über die Einstellung und den Wert an das DUT. Dabei wird die Methode ***registerDutPortHandle, die*** Setupfunktion für den DUT Port, ausgeführt. Auch befindet sich hier die Funktion ***setSettingDUTPort***, das selbe macht wie ***setSettingOnHost*** nur das es hier nur für einen einzigen Port ist. Es wird auch das ***OnRecieved*** Event von hier aus auf dem DUT registriert. Wenn das Event passiert wird die Methode ***SerialRecieveEventHandler.OnReciev*** ausgeführt. Dabei handelt es sich um eine Methode aus der RemoteApp. Die Methode baut wie schon gesagt die Nachricht, die zurückgesendet wird und erstellt aus der erhaltenen Nachricht des Events, die Einstellungen, um das Handle zu erstellen und letzten Endes die Nachricht zu übermitteln.

Auffällig sind in der ***sendMessage***-Methode die vielen ***sleeps***, die verwendet werden um die Löschung des Handles und die Neue Erstellung auf der DUT Seite richtig zu Timen. Die Tabel „test“ und „tests“ werden benötig, um zu dokumentieren welche Nachrichten verschickt wurden. Dabei speichert „test“ die erwarteten Nachrichten oder auch nicht erwartete Nachrichten ab, um später normal durch sie durch zu iterieren. „tests“ hingegen speichert unter der Nachricht einen Boolean entsprechend ob die Nachricht ankommen soll oder nicht. Doch warum dieser Aufwand? Führt man das Programm aus, wird man feststellen, dass alle ***OnRecieved*** Events, die zuvor in Setup registriert wurden, auf einmal ausgeführt werden. Dies liegt an der aktuellen AppEngine Version der SIM2000ST, also an der AppEngine des Hosts. Diese beinhaltet noch keine Möglichkeiten zum Threaden bzw. erschwert diese. Der Test Case an sich ist schon ein Event! Wird nun ein weiteres Event ausgelöst so kommt dies in die Eventqueue. Nachdem Ende des Cases wird die Queue nun abgearbeitet. Die ***handleOnRecieve*** Methode des Hosts verarbeitet dann die einzelnen Events hintereinander.

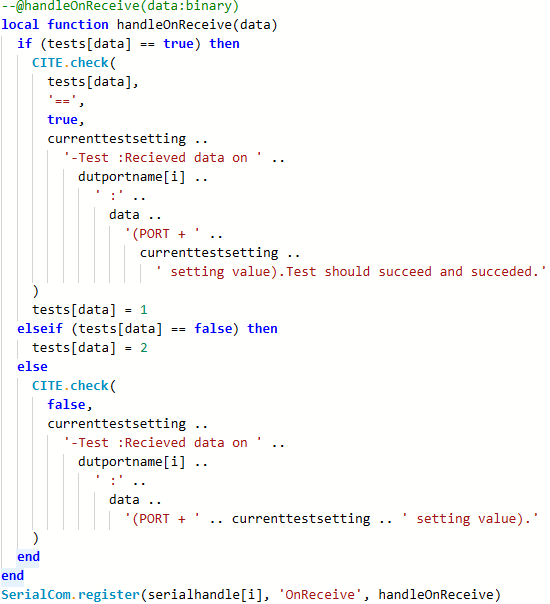


Abbildung 23: Methode die beim Event ‚OnReceive‘ für den Host ausgeführt wird

Das Setzen der „1“ und „2“ hat den Hintergrund das ganz am Ende im Teardown die Methode ***checkForFailedTests*** überprüft was in dem Tabel tests steht. Ist ein Eintrag noch true wurde eine Nachricht nicht empfangen, die hätte ankommen sollen. Ist ein Eintrag „2“ so wurde Nachrichtempfangen die nie hätte ankommen sollen.

Ist der Test soweit durchgelaufen werden die Einstellungen des Hosts auf Standartwerte gesetzt und der nächste Test wird gestartet. Getestete Einstellung sind unteranderem die Baudrate, der Typ (RS232, RS422 und RS485. Wobei RS485 noch nicht für die Eco implementiert wurde), die Parity, die StopBit-Einstellung, die DataBit-Einstellung, die Termination und die Handshake-Einstellung.

## **Digitale Ausgänge**

Die digitalen Outputs bieten im Vergleich zu den Seriellen eher weniger testbare Funktionalität. Dabei werden die Logik-Einstellung so wie die ActivationMode Einstellung einmal mit dem Increment-Modus wie dem Time-Modus getestet.

### **Logik Test**

Die Logik Einstellung kann zwei ENUM Werte habe. Einmal ACTIVE\_HIGH und ACTIVE\_LOW, dabei bedeutet AVTIVE\_HIGH das bei true gesetzten Out high-Pegel anliegt und bei false gesetzten Out low-Pegel. ACTIVE\_LOW würde hingegen das Gegenteil bedeuten. Um die Funktionalität zu testen wird hauptsächlich die Funktion ***setDigitalOut*** verwendet.

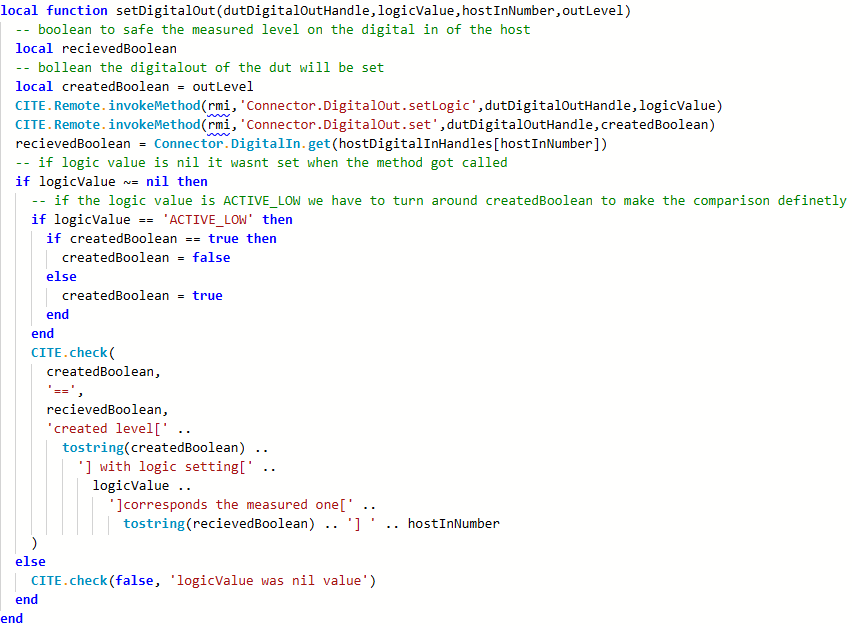


Abbildung 24: setDigitalOut-Funktion in der TC\_DigitalOut.lua

Dabei werden die unterschiedlichen Logik-Einstellung getestet und überprüft ob der Pegel, der am digital In des Hosts ankommt den Eingestellten entspricht.

### **ActivationMode-Einstellung**

Die Einstellung des ActivationModes bedeutet wie lange der der Pegel nach bspw. gesetzten true high bleib. Einmal nach einer bestimmten Zeit und einmal nach einem bestimmten Inkrement stand. Für den Stand des Inkrements würde man normalerweise den Encoder benutzen, jedoch ist das für einen so simplen Test rechtaufwendig. Außerdem wird der Encoder separat getestet. Hätte der Encoder ein Fehler würde dieser Test fehlschlagen, obwohl die Funktionalität vom DigitalOut eventuell gewährleistet wäre. Deshalb ist es sicherer die Conveyer-CROWN zu verwenden. Der Stand des Conveyer lässt sich dabei recht einfach mit einer set-Funktion setzen. Technisch unterscheiden sich die Test Cases jedoch nicht. Beim Inkrement wird der Pegel high gesetzt dann wird überprüft ob er dem entspricht, danach wird der Increment über den Conveyer invoked und der Pegel abermals überprüft. Dabei werden verschiedene Increments überprüft, dies geschieht wie schon bei der seriellen Kommunikation über einen Tabel mit Werten. Der zeitliche ActivationMode ist dabei dasselbe nur mit unterschiedlichen Zeiten.

### **Aufgetretene Bugs**

Auch hier habe ich einen Bug entdecken können. So ist es dem ActivationMode nicht möglich Increments die größer als 32767 sind zu erfassen. Der Pegel wird erst gar nicht gesetzt. Der Conveyer besitzt einen overrun aber erst ab 65536. Dabei ist es ihm nicht möglich negative Werte anzunehmen. Es lassen sich jedoch problemlos negative Increment Stände einstellen. Eventuell ist das ein Softwarefehler der CROWN und es werden unterschiedlich Uint16 und int16 verwendet. Dies würde jedoch keinen Sinn in Bezug auf den Encoder machen. Dieser nimmt, wie wir später sehen werden, zwar auch negative Werte an jedoch auch Increments weit über 100k. So oder so stimmt hier etwas nicht, was natürlich als Jira Issue dokumentiert wurde.

## **Encoder**

Der Encoder der SIM2000ST-E misst Frequenzen bis 30k Herz. Um diese für den Test zu erreichen wird der Decoder der SIM200ST verwendet. Die 2000ST besitzt dafür zwei mögliche Decoder einmal am DigitalOut und einmal am IO-Link Master. Da der Decoder des digitalOut einen Ausgangstransistor mit zu geringer Kapazität besitzt sind nur Frequenzen bis 1000Hz möglich. Der Encoder erkennt zwar noch Frequenzen bis zu 6000Hz, dann brechen die Flanken jedoch zunehmest ein. Deshalb wird der Decoder des IO-Link Master verwendet, der mühelos 30kHz schafft. Der Decoder erstellt sich als sogenannter Controlflow einer graphischen Oberfläche von Appstudio. Der Controlflow erzeugt eine XML-Datei, die dann vom Skript benutzt werden kann.

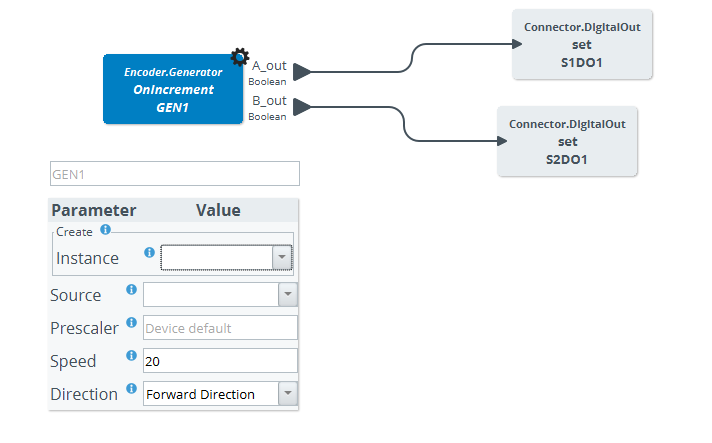


Abbildung 25: Control-Flow des Decoders

### **Direction-Test**

Die Richtung des Decoders kann einmal „FORWARD“ und „BACKWARD“ als ENUM Wert annehmen. Der Encoder besitzt die Funktion ***getDirection*** mit es sich leicht herausfinden lässt ob die Richtung sich geändert hat. Der Test besteht daraus, dass die Richtung eingestellt wird dann geschaut wird ob sie mit der, von der invokten ***getDirection***, returned übereinstimmt. Dies für wird dann jeweils für jede Richtung einmal gemacht. Zuständig ist dafür die Funktion ***controlDirection***. Merkwürdig ist dabei die Zeit, die der Encoder braucht, um eine Richtungsänderung zu detektieren. Erkennen kann man die an dem sechs Sekunden langen ***sleep*** bei kürzeren Zeiten kommt es unregelmäßig dazu das der Test fehlschlägt. Dies erschien mir recht lange, weshalb ich es in einem Jira Issue dokumentiert habe. Dabei bleibt erwähnt das die Frequenz mit der getestet wird nicht sonderlich ausgewöhnlich ist mit 1000Hz.

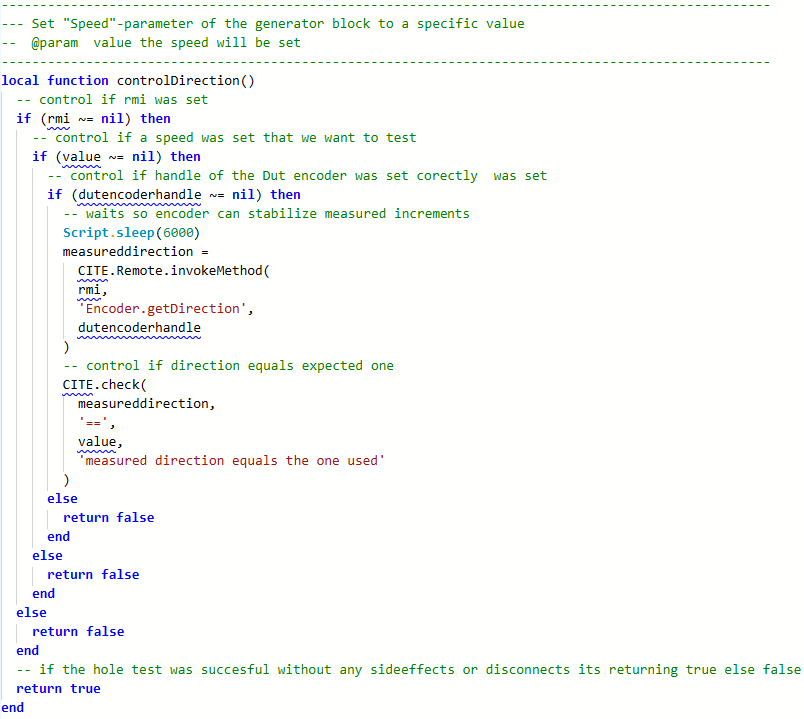


Abbildung 26: ***controlDirection*** Funktion

### **Speed und Increment – Test**

Bei dem Speed handelt es sich um die vom Encoder selbst ermittelte Frequenz an Ticks pro Sekunde. Um diese zu Testen, wird der Inkrement-Stand zwei Mal gemessen, jeweils nach einem ***sleep*** von einer Sekunde. Durch die Subtraktion des zweiten Inkrements-Stand mit dem ersten lässt sich der Speed errechnen. Dieser errechnete Speed wird dann jeweils mit dem Invokten wie auch mit dem eingestellten Verglichen. Da es fasst ausgeschlossen ist das dabei exakte Werte verglichen werden wird mit einer gewissen Toleranz verglichen und geschaut, mit einem ***CITE.checkRange,*** ob der Wert noch in einem Rahmen liegt. Da Frequenzen von einem 1Hz bis 30kHz getestet werden ist eine konstante Toleranz nicht praktikabel. So kommt es bei niedrigen Frequenzen gerne mal zu Toleranzen von fast 100% die trotzdem Akzeptabel sind. Bei Frequenzen von 30kHz ist diese jedoch inakzeptabel. Deshalb wird die Toleranz variabel mit der Methode ***getTolerance***, bei der 1 durch die aktuelle Geschwindigkeit dividiert wird und anschließend mit 100 multipliziert und dann mit 2.5 addiert. Dadurch haben Frequenzen von 1Hz eine Toleranz von ca. 102.5% und Frequenzen von 30kHz eine Toleranz von 2.5%.

Auch sind mir hier ein paar Bugs aufgefallen die ich ausführlich in Jira dokumentiert habe. So hängt sich der Encoder bei gewissen Frequenzen (bspw. 1000Hz) nach einer bestimmten Zeit auf, was sich nur wieder mit einem hard-reset lösen lässt. Dabei ist die Zeit nach dem er sich Aufhängt immer dieselbe. Auch treten unter bestimmten Bedingungen Race-Conditions auf.

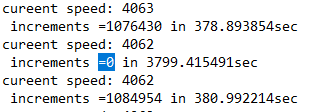


Abbildung 27: Beispiel für Race-Conditions des Encoders

# **Resümee und Fazit**

Diese Kapitel fasst noch mal alles zusammen, zieht ein Fazit und gibt einen Ausblick über den CITE-Test der SIM2000ST-Eco. Das CITE-Testsystem ist ein komplexes Verfahren was die Funktionstüchtigkeit eines Geräts vor der Veröffentlichung wirklich sehr gut verifizieren kann. Auch hilft es bei der Entwicklung herauszufinden Welche Fehler noch bestehen und welche Komponenten vielleicht noch einmal überarbeitet gehören. Die von mir Beschriebenen Testfälle funktionieren soweit einwandfrei und repräsentieren recht gut das Vorgehen bei der Erstellung eines CITE-Tests, sowie den Aufbau. Dabei sei jedoch gesagt das außerhalb dieser Test bereits einen weiteren für die **C**ontroller **A**rea **N**etwork (CAN) – Kommunikation geschrieben habe. Dieser ist in der Implementierung jedoch noch Umfangreicher als der der seriellen Kommunikation. Außerdem ist er zum momentanen Zeitpunkt nicht fertig und kann deswegen nicht Aufgenommen werden. Auch wird es im fertigen Test auch noch einen Test zum IO-Link-Master de Eco geben den ich soweit nicht behandelt konnte. Auch habe ich bereits vorhandene BASE-Tests die relativ oberflächig aber auch umfangreich sämtliche Standard-CROWNs testen hinzugefügt. Diese BASE und CmnAlg Tests sind Bestandteil der Kompletten und ausführlichen Testsequenz und eigentlich bei allen SIM-tests vorhanden.

Quellenverzeichnis

[1] SICK AG, "Wikipedia Sick(Unternehmen," [Online]. Availabel:

https://www.martinfowler.com/articles/continuousIntegration.html [18.09.2020]

[2] SICK Geschäftsbericht 2019 [Online] Availabel: https://www.sick.com/medias/SICK-GB-2019-deutsch-save-hochformat.pdf?context=bWFzdGVyfGNvbnRlbnR8NjM2NzcwN3xhcHBsaWNhdGlvbi9wZGZ8Y29udGVudC9oNTEvaGY1LzEwODI3NDYwNjA4MDMwLnBkZnxkZjljZDc2MTJhZTc0NDdlMzZkMDg0ZjU1NjU0MDAzNWU5NmUxZWMyY2YxMzE2YWU5ZTg5NjlkYzVhYWJiZWQx [18.09.2020]

[3] Sick Intranet [Online] extern:

https://mosaicplus.sickcn.net/display/intranetCC/Company+Presentation [18.09.2020]

[4] Aufbau der GBC05 Teams erstellt von dem Praktikanten Maximillian Humburger und zu Verfügung gestellt

[5] Lua, "lua wiki," [Online] Availabel:

https://de.wikipedia.org/wiki/Lua [18.09.2020]

[6] Information die mir während des Aufenthalts von meinem Betreuer Florian Bartsch, Entwicklungsingenieur der GBC 05, erklärt wurden

[7] AppStudio: Introduction [Online] nur mit Berechtigung einsehbar:

https://supportportal.sick.com/tutorial/appstudio-introduction/ [18.09.2020]

[8] SIM2x00 -SIM2000 Family [Online] Intranet, extern:

https://mosaicplus.sickcn.net/display/wsGBC05/SIM2x00+-+SIM2000+Family

[9] SIM Benchmarks [Online] Intranet, extern & benötigt Berechtigung:

https://mosaicplus.sickcn.net/display/VIP/Benchmarks