SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Pauli Latva-Kokko

Koodikattavuuden mittauksen automatisointi turvakriittisen laiteohjelmiston testauksessa

Opinnäytetyö
Kevät 2023
Insinööri (AMK), Tietotekniikka

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Tutkinto-ohjelma: Insinööri (AMK), Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Pauli Latva-Kokko

Työn nimi: Koodikattavuuden mittauksen automatisointi turvakriittisen laiteohjelmiston testauksessa.

Ohjaaja: Juha Yli-Hemminki

Vuosi: 2023 Sivumäärä: 65 Liitteiden lukumäärä: 0

Opinnäytetyön tavoitteena oli automatisoida koodikattavuuden mittausta Epecin laiteohjelmiston integraatiotestauksessa. Tavoite koodikattavuusmittauksen automatisoinnista perustuu ISO 26262 -standardin suosituksiin. Standardi suosittelee mittaamaan laiteohjelmiston integraatiotestauksessa funktio- ja kutsukattavuutta. Arkkitehtuurisen koodikattavuuden ollessa liian matala on uusia testitapauksia lisättävä tai osoitettava riittävä testikattavuus muilla keinoilla.

Koodikattavuusmittaukset suoritettiin Lauterbachin laitteistodebuggerilla, jota voidaan komentaa ohjelmallisesti Python- tai LabVIEW-rajapinnan kautta. Lauterbachin laitteet tukevat ISO 26262 -standardin vaatimia koodikattavuusmittareita, joten niillä voidaan mitata sertifiointiin vaaditut koodikattavuusmittaukset.

Kun kohdelaitteelle ajettiin testejä, siirrettiin koodikattavuusdata isäntäkoneelle tietovirtana ja testitapauksen loputtua se tallennettiin pysyvään tiedostoon. Kun kaikki testitapaukset oli suoritettu, ladattiin koodikattavuusdata pysyvästi tallennetuista tiedostoista, muodostettiin koodikattavuustietokannat ja summattiin ne yhteen.

Koodikattavuustietokantojen pohjalta luotiin HTML-raportti. HTML-raportista nähtiin testauksen koodikattavuus laiteohjelmiston tasolla ja tarkemmin moduuli-, funktio- tai rivitasolla. Opinnäytetyön menetelmiä käyttämällä pystytään Epecin laiteohjelmiston integraatiotestauksessa täyttämään ISO 26262 -standardin vaatimus koodikattavuuden mittauksesta.

[1] Asiasanat: koodikattavuus, laiteohjelmisto, laitteistodebuggeri, ISO 26262

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Degree programme: Bachelor of Engineering, Information Technology

Specialisation: Software engineering

Author: Pauli Latva-Kokko

Title of thesis: Automating code coverage measurement in safety-related firmware testing

Supervisor: Juha Yli-Hemminki

Year: 2023 Number of pages: 65 Number of appendices: 0

The aim of the thesis was to automate the measurement of code coverage in the integration testing of Epec's firmware. The objective to automate code coverage measurement is based on the recommendations of the ISO 26262 standard. The standard recommends measuring function coverage and call coverage during firmware integration testing. If the architectural code coverage is too low, new test cases must be added or sufficient test coverage must be demonstrated by other means.

Code coverage measurements were performed with a Lauterbach In-Circuit Debugger, which can be programmatically commanded via Python or LabVIEW. Lauterbach devices support the code coverage metrics required by the ISO 26262 standard and can therefore be used to measure the code coverage measurements required for certification.

When tests were run on the target device, code coverage data was streamed to the host machine and after the test case was over it was saved permanently to a file. Once all the test cases had been run the code coverage data was processed from permanently stored files and code coverage databases were created and then summed up.

An HTML report was generated based on the code coverage databases. The HTML report showed the code coverage of the testing at firmware level and in more detail at module, function, or line level. Using the methods of the thesis, Epec's firmware integration testing can meet the requirement of the ISO 26262 standard regarding code coverage measurement.

[1] Keywords: code coverage, firmware, In-Circuit Debugger, ISO 26262

SISÄLTÖ

Kuva-, kuvio- ja taulukkoluettelo

Käytetyt termit ja lyhenteet

API Ohjelmointirajapinta (*engl. Application Programming Interface*) on kahden tai useamman ohjelmiston välinen rajapinta, jonka kautta ohjelmistot voivat kommunikoida toistensa kanssa.

ASIL ASIL eli Automotive Safety Integrity Level on standardin ISO 26262 määrittelemä riskiluokka turvattuotteille. ASIL-tasot ovat ASIL A, ASIL B, ASIL C JA ASIL D, joista ASIL D asettaa tuotteelle kaikista tiukimmat kokonaisvaatimukset ja ASIL A kaikista matalimmat.

Debuggaus Debuggaus eli virheenjäljitys on testauksessa löytyneen virheellisen toiminnan syyn etsimistä koodista ja virheen löytymisen jälkeen virheen analysointia sekä virheen korjaamista koodista.

ICD ICD eli In-Circuit Debugger on erillinen laite, jota käytetään yleensä laiteohjelmiston virheenjäljitykseen. Tässä työssä termi on suomennettu laitteistodebuggeriksi.

ICE ICE eli In-Circuit Emulator on erillinen laite, jota käytetään yleensä laiteohjelmiston virheenjäljitykseen. ICE emuloi eli jäljittelee kohdelaitteen prosessoria omalla prosessorillaan, kun taas ICD hallinnoi kohdelaitteen prosessoria.

IEC 61508 IEC 61508 -standardi on nimeltään ”Sähköisten/elektronisten/ohjelmoitavien elektronisten turvallisuuteen liittyvien järjestelmien toiminnallinen turvallisuus”. Sen on määritellyt sähköalan standardoimisjärjestö IEC.

Instanssi Instanssilla tarkoitetaan tässä työssä jonkin ajettavan tietokoneohjelman ilmentymää. Tietokoneohjelman ilmentymällä on oma sisäinen tilansa, joka voi erota muista saman tietokoneohjelman instanssien tiloista.

ISO 26262 ISO 26262 -standardi määrittelee sähkö- ja/tai elektronisten laitteiden toiminnallisen turvallisuuden standardin sellaisille laitteille, jotka on asennettu tiekäyttöön tarkoitettuihin kulkuneuvoihin. Sen on määritellyt kansainvälinen standardoimisjärjestö ISO.

JTAG JTAG eli Joint Test Action Group on piirilevyjen testaamiseen ja varmistamiseen käytettävä teollisuusstandardi. Se määrittelee mm. JTAG-portin, jota käytetään usein laitteistodebuggerin liittämiseen.

Koodikattavuus Koodikattavuus (*engl. code coverage*) lasketaan jakamalla testauksessa suoritetun koodin määrä kaiken koodin määrällä. Koodia mitataan yleensä lähdekoodina tai objektikoodina.

Lause Ohjelmoinnissa lauseet ovat ohjelmointikielten syntaksisia perusyksiköitä, jotka suorittavat jonkin toiminnon. Muun muassa muuttujaan sijoitus, funktiokutsu, paluukutsu, silmukkarakenne ja ehtolause ovat esimerkkejä ohjelmointikielen lauseista.

Lockstep Lockstep-järjestelmät ovat vikasietoisia järjestelmiä, joissa samat operaatiot ajetaan kahdella tai useammalla järjestelmällä samanaikaisesti. Operaatioiden tuloksia verrataan keskenään, jolloin virheet voidaan havaita tai korjata.

Objektikoodi Objektikoodi (*engl. object code*) on kääntäjän tuottamaa konekieltä, joka kertoo laitteelle, miten ohjelmointikielellä kirjoitettu ohjelma pitää suorittaa.

SIL SIL eli Safety Integrity Level tarkoittaa turvatoimilla aikaansaadun suhteellisen riskitason pienenemistä. SIL-turvatuotteet luokitellaan yhdestä neljään. SIL4-luokan turvatuotteessa riskitaso on kaikista pienin.

TCP TCP eli Transmission Control Protocol on luotettava tietoliikenneprotokolla, jota voidaan käyttää laitteiden tai ohjelmien väliseen keskinäiseen luotettavaan tiedonsiirtoon.

Tracing Tracing eli jäljitys tarkoittaa tärkeiden ohjelmiston tapahtumien tallentamista lokiin tai muistiin virheenjäljityksen helpottamiseksi. Jäljitys on pääasiassa ohjelmiston kehittäjien käyttämä työkalu, ja tallennettavat ohjelmistotapahtumat ovat usein matalan tason tapahtumia.

UDP UDP eli User Datagram Protocol on yhteydetön tietoliikenneprotokolla, jota voidaan käyttää laitteiden tai ohjelmien väliseen keskinäiseen tiedonsiirtoon. UDP ei sisällä samanlaista kättelyä ja tiedon oikeellisuuden varmistusta kuin TCP.

1 JOHDANTO

Tämän opinnäytetyön päätavoitteena on automatisoida koodikattavuuden mittausta Epecin laiteohjelmiston integraatiotestauksessa. Turvaluokitelluille tuotteille ISO 26262 -standardi suosittelee koodikattavuuden mittausta integraatiotestauksessa (International Organization for Standardization (ISO), 2018d, luku 10.4.5). Standardin mukaan todisteeksi siitä, että laiteohjelmiston integraatiotestaus on kattava ja integraatiotestauksen tavoitteet on saavutettu, on integraatiotestauksessa mitattava kutsu- tai funktiokattavuutta. Mikäli arkkitehtuurinen koodikattavuus on liian matala, on uusia testitapauksia lisättävä tai osoitettava riittävä testikattavuus muilla keinoilla. ISO 26262 -standardi antaa selkeän motiivin koodikattavuuden mittauksen automatisoinnille. Työssä koodikattavuuden mittaukseen käytettävät Lauterbachin laitteet tukevat ISO 26262 -standardin vaatimia koodikattavuusmittareita, joten niillä voidaan mitata sertifiointiin vaaditut koodikattavuusmittaukset (Lauterbach 2022b, s. 8).

Toissijaisena tavoitteena tavoitellaan koodikattavuusmittauksen yleisiä hyötyjä. Koodikattavuusmittauksella laiteohjelmistosta voidaan löytää kuollutta tai epäaktiivista koodia, mikä voidaan poistaa tai dokumentoida tarkemmin. Lisäksi voi löytyä testaamatonta, mutta tarpeellista koodia, jolloin voidaan lisätä testitapauksia kattamaan tämä koodi.

Työn teoriakehyksessä esitellään ensin ISO 26262 -standardi, sen yhteys muihin funktionaalisen turvallisuuden standardeihin ja turvatuotteen ohjelmistokehityksen referenssimalli. Seuraavaksi käydään läpi testauksen tasot ja laiteohjelmiston testauksen erityispiirteitä. Tämän jälkeen käsitellään koodikattavuusmittaus, sen eri mittarit tai metriikat ja ISO 26262 -standardin asettamat vaatimukset koodikattavuusmittauksen suhteen. Lisäksi käydään läpi laiteohjelmiston virheenjäljityksessä käytettyjä tekniikoita ja laitteita.

Työn käytännön osuus jakautuu kahteen ylimmän tason lukuun. Ensin esitellään työn lähtötilanne ja siinä käytettävät käytännön menetelmät. Seuraavaksi käydään läpi koodikattavuusmittauksen eteneminen vaihe vaiheelta Robot Frameworkillä sekä TestStand-testinhallintaohjelmistolla. Työn tuloksena saadaan koodikattavuusraportti, josta nähdään testatun yksikön funktio- tai kutsukattavuus. Lopuksi pohditaan työn onnistumista ja työssä ratkaisematta jääneitä asioita.

1.1 Epec

Epec on Ponsse-konserniin kuuluva teknologiayritys, joka on erikoistunut sähköisiin ohjausjärjestelmiin, kustomoituihin tuotteisiin, täys- ja puolisähköisiin ajoneuvojärjestelmiin sekä avustaviin ja autonomisiin järjestelmiin (Epec, 2022a).

Epec tuottaa tuotteitaan ja palveluitaan erityisesti työkoneisiin sekä kuorma-autoihin. Kohdeyrityksiä ovat esimerkiksi metsä-, maatalous-, kaivos-, kuljetus- ja rakennusalan työkoneita valmistavat tai käyttävät yritykset (Epec, 2022b). Epecin tuotteita ovat esimerkiksi ohjausyksiköt, näytöt, telematiikkayksiköt ja pilvipalvelut.

1.2 Työn tausta

Tämän työn tekijälle annettiin työharjoittelussa Epecillä tehtäväksi tutkia, miten laiteohjelmiston koodikattavuuden mittaus voitaisiin automatisoida turvatuotteiden integraatiotestauksessa. Ilman jonkinlaista keinoa automatisoida koodikattavuuden mittaus testauksessa, kuluisi integraatiotestaukseen todella paljon ihmistyötunteja.

Tiedossa oli, että koodikattavuutta voidaan mitata jo käytössä olleilla Lauterbachin laitteilla. Automatisointiin oli vaihtoehtoina käyttää Python- tai LabVIEW-rajapintaa. Työharjoittelun aikana myös kokeiltiin käytännössä, että automatisointi toimii Epecin SC52-ohjausyksiköllä, käytettäessä LabVIEW-rajapintaa.

2 ISO 26262

ISO 26262 -standardi, nimeltään ”Road vehicles – Functional safety”, on kansainvälinen standardi, joka määrittelee sähkö- ja/tai elektronisten järjestelmien funktionaalisen turvallisuuden standardin sellaisille laitteille, jotka on asennettu tiekäyttöön tarkoitettuihin kulkuneuvoihin (International Organization for Standardization (ISO), 2018a, s. vi). ISO 26262 -standardi on adaptaatio IEC 61508 -sarjan standardeista, ja siinä on otettu huomioon tieliikennekäyttöisten kulkuneuvojen erityispiirteet. Funktionaalinen turvallisuus eli toiminnallinen turvallisuus tarkoittaa järjestelmän virheellisestä toiminnasta aiheutuvien kohtuuttomien ihmisiin kohdistuvien turvariskien estämistä (mts. 14).

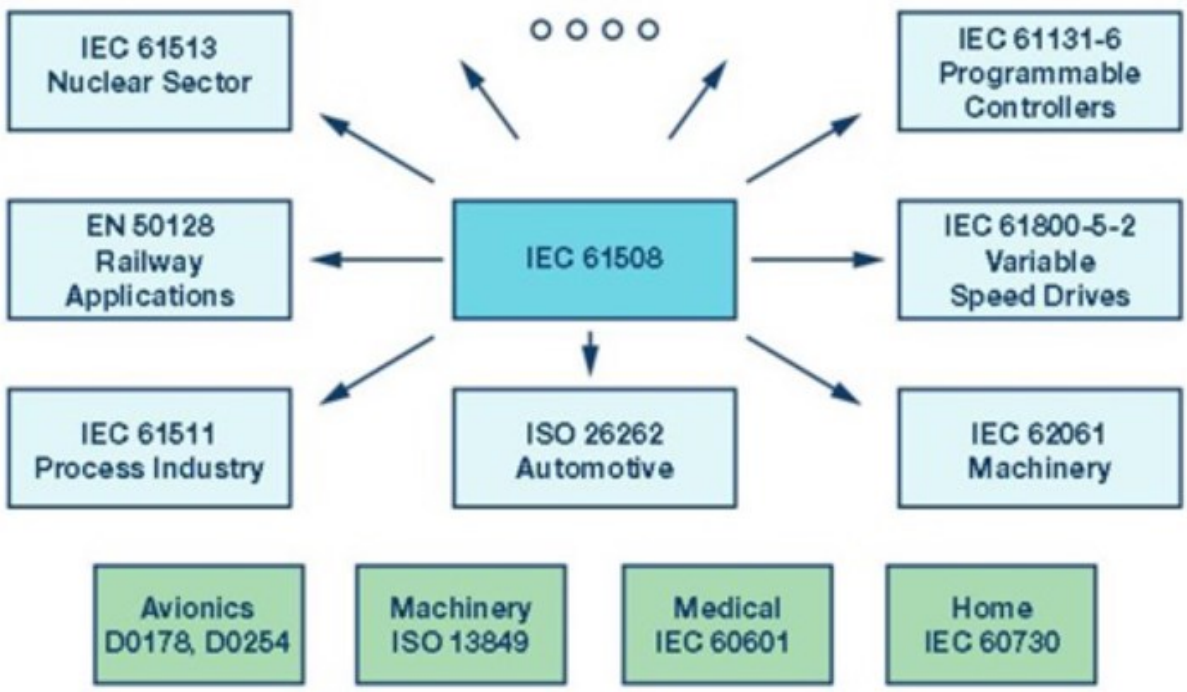
Autoala ja tieliikenteen kulkuneuvot ovat jatkuvasti monimutkaistuneet, ja niissä olevien ohjelmistojen sekä mekatroniikan määrä on lisääntynyt (International Organization for Standardization (ISO), 2018a, s. vi). Tämä on johtanut järjestelmällisten virheiden ja satunnaisten laitteistovikojen riskin kasvuun. ISO 26262 -standardi antaa ohjeistusta näiden riskien pienentämiseen sopivilla vaatimuksilla ja prosesseilla.

Standardissa määritellään järjestelmille turvatasot eli ASIL tasot (International Organization for Standardization (ISO), 2018a, s. 2). ASIL tulee sanoista ”Automotive Safety Integrity Level”. ASIL-turvatasoja merkitään kirjaimilla A, B, C ja D, joista ASIL D-tasoon kuuluvat järjestelmät ovat läpäisseet kaikista tiukimmat ISO 26262 -vaatimukset ja omaavat kaikista tiukimmat turvamekanismit kohtuuttoman riskin välttämiseksi. Virheellinen toiminta voisi johtua esimerkiksi sähköisestä tai ohjelmallisesta viasta.

2.1 Yhteys IEC 61508 -standardiin

ISO 26262 -standardi on alakohtainen adaptaatio standardista IEC 61508. IEC 61508 -standardi määrittelee toiminnallisen turvallisuuden vaatimukset sähköisille, elektronisille ja ohjelmoitaville elektronisille turvallisuuteen liittyville järjestelmille (Suomen Standardoimisliitto (SFS), 2011, s. 1). Standardi on nimeltään” Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”. Standardin ohjeet on tarkoitettu yleiseen käyttöön ja teollisuus voi käyttää niitä sellaisenaan (mts. 12). Alla on

havainnollistava kuvio (Kuvio 1) IEC 61508 -standardiin liittyvistä alakohtaisista standardeista, ja siitä nähdään myös yhteys ISO 26262- standardiin.



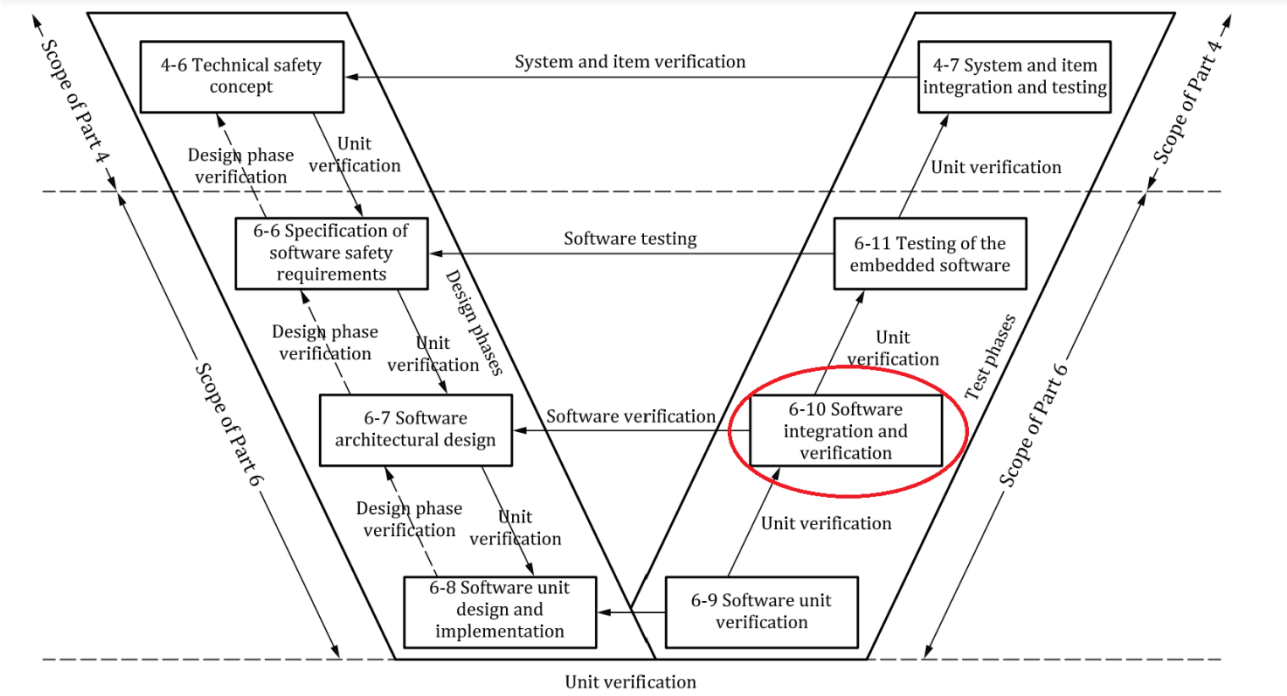
Kuvio 1. Funktionaalisen turvallisuuden standardit eri aloille (Meany, i.a.).

IEC 61508 -standardi käyttää turvallisuuden eheyden luokitteluun SIL-tasoja (Suomen Standardoimisliitto (SFS), 2011, s. 16–17). SIL tulee sanoista ”Safety Integrity Level”. Suomeksi puhutaan turvallisuuden eheyden tasoista. SIL-luokitus kuvaa riskin pienentämisen tasoa, jolla riski voidaan vähentää hyväksyttävälle tasolle. Standardissa järjestelmät luokitellaan neljään eri tasoon: SIL1, SIL2, SIL3 ja SIL4. Näistä tasoista SIL4 asettaa järjestelmällä kaikista tiukimmat vaatimukset riskin pienentämisen suhteen, jotta voitaisiin saavuttaa vaarallisen vikaantumisen pienempi todennäköisyys.

Tason 4 luokitusta ei yleensä haeta työkoneisiin tai tehdaskoneisiin, vaan se on varattu sovelluksiin, joissa virheellinen toiminta voisi johtaa kymmenien tai satojen ihmisten kuolemis- tai loukkaantumisvaaraan (Meany, i.a.). Tällaisia käyttökohteita ovat esimerkiksi junat tai ydinvoima. Työkoneissa tai tehdaslaitteissa käytettävät järjestelmät ovat yleensä SIL2- tai SIL3 -luokituksen saaneita järjestelmiä.

2.2 Turvatuotteen ohjelmistokehitys

ISO 26262 -standardissa on esitelty referenssimalli ohjelmistokehitykseen funktionaalisen turvallisuuden mukaisille laitteille. Kyseessä on V-malli, josta nähdään ohjelmistokehityksen vaiheet koko järjestelmän turvakonseptista kohti järjestelmän testausta ja integraatiota.



Kuvio 2. ISO 26262 -standardin tuotteen ohjelmistokehityksen referenssimalli (International Organization for Standardization (ISO), 2018d, s. 4).

Kuviosta 2 nähdään, että vasemmasta yläkulmasta lähdetään V-mallissa tuotteen toteutuksessa liikkeelle teknisen turvakonseptin luomisesta edeten yksityiskohtaisempiin asioihin kohti ohjelmistoyksikköjen suunnittelua ja toteutusta. Tämän jälkeen edetään testauksessa yksikkötestaustasolta kohti suurempia ja monimutkaisempia kokonaisuuksia, ja lopulta päädytään koko järjestelmän testaukseen ja integraatioon. Kuvan vasemmalla puolella näkyviä ohjelmistotasoja vastaavat oikealla puolella kyseiseen tasoon kohdistuvat testaus ja varmistus.

Tässä opinnäytetyössä keskitytään kuvassa oikealla puolella olevaan toiseksi alimpaan laatikkoon, eli integraatiotestaukseen, ja vielä tarkemmin laiteohjelmiston integraatiotestauksen koodikattavuuden mittaukseen.

2.3 Turvamekanismit

Turvatuotteelle tehdään kehitysvaiheessa vaara- ja riskianalyysi (International Organization for Standardization (ISO), 2018b, luku 7.2). Tämän analyysin ja tavoiteltavan ASIL-luokan pohjalta johdetaan turvatavoitteita. Turvatavoitteista johdetaan edelleen yksittäisiä funktionaalisen turvallisuuden vaatimuksia. Näiden vaatimusten takia turvatuotteeseen lisätään erilaisia turvamekanismeja.

Turvamekanismit ovat teknisiä ratkaisuja, joilla voidaan havaita, vähentää, sietää, hallita tai välttää vikoja (International Organization for Standardization (ISO), 2018a, luku 3.142). Turvamekanismien tavoitteena on säilyttää laitteen normaali tarkoituksenmukainen toiminnallisuus tai asettaa laite turvalliseen tilaan. Turvallinen tila tai turvatila on operaatiotila, jossa riski ihmisille, esimerkiksi laitteen kuljettajalle tai muille tienkäyttäjille, ei ole kohtuuton. Turvatilaan päädytään laitteen havaitseman vian tai virheen seurauksena.

2.4 Lockstep-turvamekanismi

Tuplaydin lockstep -turvamekanismin ideana on ajaa samat käskyt kahdella eri prosessorin ytimellä ja verrata näiden käskyjen tuloksia keskenään (International Organization for Standardization (ISO), 2018c, s. 55). Jos prosessorin ytimet päätyvät eri tuloksiin, laite siirtyy turvatilaan. Toinen ydin voi ajaa samat operaatiot hieman ajallisesti jäljessä toisesta ytimestä, jotta ulkoiset häiriöt eivät vaikuttaisi molempien ytimien tuloksiin samanaikaisesti vääristäen molempia.

Tuplaydin lockstep -turvamekanismissa sama koodi ajetaan molemmissa prosessorin ytimissä (International Organization for Standardization (ISO), 2018c, s. 55). Turvamekanismin seurauksena kahdella prosessorin ytimellä saavutetaan vain yhden ytimen laskentateho. Hyvin tehty lockstep-arkkitehtuuri ottaa huomioon myös yhteisistä juurisyistä johtuvat vikatilanteet, joissa molemmat ytimet tekevät samat operaatiot virheellisesti ja päätyvät samaan virheelliseen lopputulokseen.

3 TESTAUS

Testauksella tarkoitetaan ohjelman suorittamista siten, että tarkoituksena on löytää siitä virheitä (Chopra, 2018, s. 1). Testauksessa ohjelmalle annetaan järkeviä ja epäjärkeviä syötteitä ja verrataan saatuja tuloksia odotettuihin tuloksiin. Tässä opinnäytetyössä keskitytään dynaamiseen testaukseen. Dynaamisessa testauksessa testataan sellaista koodia, joka voidaan jo kääntää ja suorittaa. Dynaamisessa testauksessa voidaan testata yhtä ohjelmamoduulia tai kokonaista ohjelmaa.

Bernard Homèsin (2012, s. 9) mukaan ohjelmiston testauksella on kaksi erilaista tavoitetta:

- Löytää virheitä ja ongelmia ohjelmistosta, jotta ne voidaan korjata ennen markkinoille vientiä ja näin asiakkaat saavat laadukkaamman tuotteen.
- Auttaa arvioimaan riskiä, joka liittyy tuotteen markkinoille vientiin, sekä auttaa arvioimaan, kuinka tehokkaita organisaation prosessit ovat eliminoimaan ohjelmiston virheiden ja ongelmien syntymistä.

Testauksen tavoitteet vaihtelevat ohjelmiston elinkaaren eri vaiheissa (Homès, 2012, s. 9). Esimerkiksi yksikkö- ja integraatiotestauksessa tavoitteena on löytää mahdollisimman paljon vikoja mahdollisimman lyhyessä ajassa. Hyväksymistestauksessa taas testauksen tavoitteena on näyttää asiakkaalle, että ohjelma toimii suunnitellusti, ja saada näin asiakkaan hyväksyntä ohjelmalle.

Testaus koostuu neljästä päätasosta: yksikkötestauksesta, integraatiotestauksesta, järjestelmätestauksesta ja hyväksymistestauksesta (Homès, 2012, s. 59). Testauksen tasot tapahtuvat edellä mainitussa järjestyksessä alkaen yksikkötestauksesta. Seuraavissa luvuissa käsitellään nämä neljä testauksen päätasoa.

3.1 Yksikkötestaus

Yksikkötestaus tarkoittaa yhden ohjelmayksikön tai moduulin testaamista eristettynä muusta ohjelmasta (Chopra, 2018, s. 228). Yksikkötestauksessa testien tuloksia verrataan ohjelmayksikön vaatimusmäärittelystä ja suunnitelmasta johdettuihin tuloksiin. Yksikkötestien kirjoittaminen on helpompaa kuin muiden testaustasojen, koska testataan vain yhtä

ohjelmayksikköä kerrallaan. Yksikkötestauksessa saavutetaan yleensä parempi koodikattavuus kuin muilla testaustasoilla ja ulkoisten rajapintojen aiheuttamat ongelmat eivät vaikuta tuloksiin.

ISO 26262 -standardin mukaan yksikkötestauksen tarkoituksena on varmistaa, että ohjelmistoyksikköjen suunnittelu ja toteutus täyttää sille asetetut turvallisuusvaatimukset ja myös turvallisuuteen liittymättömät vaatimukset (International Organization for Standardization (ISO), 2018d, luku 9.2).

3.2 Integraatiotestaus

Integraatiotestauksessa testataan sisäisten ohjelmamoduulien toimintaa yhdessä toistensa kanssa sekä sisäisten moduulien toimintaa yhdessä ulkoisten järjestelmien kanssa (Chopra, 2018, s. 229). Integraatiotestaus tuo mukanaan huomattavasti enemmän monimutkaisuutta kuin pelkkä yksikkötestaus, koska ohjelmamoduulien keskinäinen toiminta voi olla hyvin monimutkaista ja sen ymmärtäminen vaatii testaajalta syvempää osaamista.

ISO 26262 -standardin mukaan integraatiotestauksen tarkoituksena on varmistaa, että ohjelmiston rajapinnat eri ohjelmistoelementtien välillä on varmistettu arkkitehtuurisen mallin vaatimalla tavalla (International Organization for Standardization (ISO), 2018d, luku 10.2). Laiteohjelmisto voi koostua turvallisuuteen liittyvistä ja turvallisuuteen liittymättömistä ohjelmistoelementeistä.

3.3 Järjestelmätestaus

Järjestelmätestaus on testausta, joka toteutetaan kokonaiselle järjestelmälle. Järjestelmä voi olla ohjelmisto tai laiteohjelmisto (Chopra, 2018, s. 238). Siinä testataan, toimiiko ohjelmisto vaatimusmäärittelyn mukaisesti. Järjestelmätestaus tehdään vasta yksikkö- ja integraatiotestausvaiheiden jälkeen. Järjestelmätestaus testaa tuotetta oikeaa käyttöympäristöä vastaavassa ympäristössä, ja sen tekevät yleensä eri henkilöt kuin yksikkö- ja integraatiotestauksen tekivät. Se tuo testaukseen mukaan asiakkaiden näkökulman ja valmistelee tuotteen hyväksymistestausta varten. Se auttaa yritystä arvioimaan riskejä, jotka liittyvät tuotteen markkinoille viemiseen.

ISO 26262 -standardin mukaan järjestelmätestauksen tarkoituksena on varmistaa, että laiteohjelmisto täyttää sille asetetut turvallisuusvaatimukset, kun se ajetaan kohdeympäristössä (International Organization for Standardization (ISO), 2018d, luku 11.1). Laiteohjelmisto ei myöskään sisällä ei-toivottuja funktionaalisuuksia tai ominaisuuksia, jotka vaikuttaisivat funktionaaliseen turvallisuuteen.

3.4 Hyväksymistestaus

Hyväksymistestaus on testauksen viimeinen vaihe, minkä tekevät asiakkaat itse (Chopra, 2018, s. 260). Asiakkaat määrittelevät testitapaukset, jotka tuotteen pitää läpäistä, jotta tuote voidaan hyväksyä. Hyväksymistestauksen tarkoituksena ei ole enää löytää vikoja, vaan varmistaa että tuote toimii ennalta määritellyllä tavalla. Jos hyväksymistestauksessa ilmenee isoja ongelmia, tuotteen julkaisua markkinoille saatetaan joutua lykkäämään ja sitä joudutaan jatkokehittämään. Tämä tarkoittaa tuottajayritykselle usein suuriakin taloudellisia tappioita, minkä takia kattava testaus aiemmissa testausvaiheissa onkin erittäin tärkeää.

Turvatuotteille hyväksymistestauksen voi tehdä myös jokin valtuutettu viranomainen tai muu ulkopuolinen valtuutettu taho (Homès, 2012, s. 65). Ulkopuolinen taho varmistaa, että sääntöjä ja standardeja on noudatettu ohjelmiston toteutuksen ja testauksen aikana. Tällaisia hyväksymistestauksia tehdään yleensä vain rajatuilla säännellyillä aloilla, kuten lentoliikenteessä, lääketieteessä ja junaliikenteessä. Tämänkaltainen testaus keskittyy pääasiassa varmistamaan ohjelman toteutuksen ja testitiimien tuottaman todistusaineiston perusteella, että sääntöjä ja standardeja on noudatettu.

3.5 Laiteohjelmiston testaus

Gansslen (2008, s. 57) mukaan laiteohjelmiston testaus eroaa sovellusohjelmien testauksesta mm. laiteohjelmiston reaaliaika- ja kriittisyysvaatimusten suhteen. Gansslen mukaan laiteohjelmiston pitää pystyä toimimaan kaatumatta pitkiä aikoja ja sen käyttö voi olla hyvin epätasaista. Koska reaalimaailman tapahtumat ovat vaikeasti ennustettavia ja satunnaisia, on laiteohjelmiston simulaatiotestaaminen vaikeampaa ja epäluotettavampaa kuin sovellusohjelmien simulaatiotestaaminen.

Laiteohjelmisto on usein myös rahallisten ja turvallisuuteen liittyvien asioiden kannalta kriittisessä asemassa (Ganssle, 2008, s. 57). Jos esimerkiksi metsäkoneen ohjausyksikön laiteohjelmisto ei toimisi oikein eikä sitä saataisi tehdasasetusten palautuksella korjattua, pitää yleensä laiteohjelmiston tekijöiden kirjoittaa korjauksia ohjelmistoon ja päivittää ne etäyhteydellä. Pahimmassa tapauksessa pitää koko ohjausyksikkö irrottaa ja lähettää takaisin valmistajalle tarkempiin tutkimuksiin. Tämä kuluttaa sekä ohjausyksikön valmistajan että metsäkoneen omistajan aikaa ja aiheuttaa rahallisia menetyksiä molemmille.

Vielä korkeammat testauskriteerit pitää asettaa turvakriittisten järjestelmien laiteohjelmistolle. Tällaisten järjestelmien laiteohjelmiston virheellinen toiminta voi johtaa vakaviin henkilövahinkoihin. Esimerkiksi sairaaloissa, työkoneissa, tieliikenteessä ja lentoliikenteessä on käytössä paljon turvakriittisiä järjestelmiä. Turvakriittisiin järjestelmiin liittyvät omat standardinsa, kuten esimerkiksi ISO 26262 ja IEC 61508.

Kattava reaaliaikaisen toiminnan testaus laiteohjelmistossa vaatii yleensä laitteen kiinnittämistä fyysisesti testipenkkiin tai virtuaalisen simulaattorin rakentamista (Ganssle, 2008, s. 60). Fyysisessä testipenkissä voidaan esimerkiksi simuloida käyttäjän nappien painamista, mitata sisään tulevia ja ulos lähteviä virta- ja jännitearvoja sekä katkaista testattavasta laitteesta virrat pois tai laittaa ne takaisin päälle. Tarkoitukseen sopivien fyysisten testipenkkien tai virtuaalisten simulaattorien rakentaminen ja kehittäminen on usein kallista, ja siksi niitä ei aina käytetä.

4 KOODIKATTAVUUS

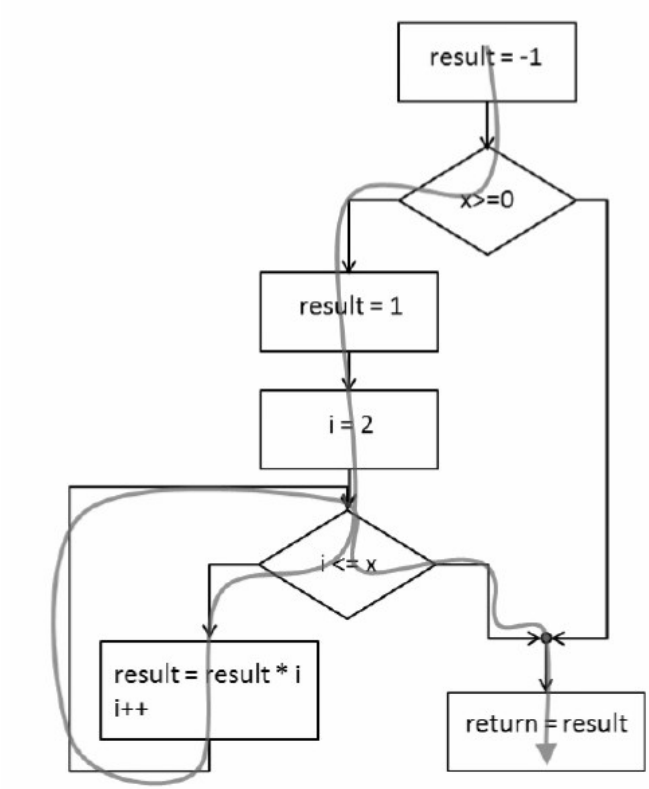
Koodikattavuus tarkoittaa testauksessa läpikäydyn koodin määrää suhteessa kaiken koodin määrään. Koodikattavuusmittauksen tarkoituksena on varmistaa ohjelmiston tai sen osien testauksen kattavuus. Koodia mitataan yleensä lähdekoodina tai objektikoodina. Yleensä koodikattavuutta mitataan työkaluilla, jotka huolehtivat automaattisesti koodin instrumentoinnista (Homès, 2012, s. 176). Instrumentoinnilla tarkoitetaan lisäinformaation tai jälkien lisäämistä koodiin, jotta ajon aikana tai sen jälkeen voidaan nähdä, mitkä osat koodista on käyty läpi. Instrumentoidun koodin ajo on hitaampaa, koska se sisältää koodikattavuuteen liittyvää lisäinformaatiota. Jos alkuperäistä lähdekoodia muokataan, pitää instrumentointi tehdä uudestaan, jotta koodikattavuus voidaan mitata uudelleen.

Yleensä koodikattavuuden mittauksen taustalla on laadullisia ja taloudellisia seikkoja: on halvempaa korjata ongelmat ennen tuotantoon vientiä, ja asiakkaalle jää parempi mielikuva yrityksestä, kun ohjelmisto toimii hyvin. Koodikattavuusmittauksen taustalla voi olla myös ohjelmiston vaatimus jonkin turvallisuusluokan täyttämisestä, esimerkiksi ISO 26262 -standardi vaatii koodikattavuuden mittausta yksikkö- ja integraatiotestaustasolla.

4.1 Yksikkötestaustason koodikattavuusmittarit

Koodikattavuutta voidaan mitata usealla eri mittauskriteerillä, ja saatu tulos riippuukin käytettävästä mittaustavasta. Seuraavissa kappaleissa käydään läpi yleisimpiä yksikkötestaustasolla käytettäviä koodikattavuusmittareita.

Lausekattavuus. Lausekattavuus (*engl. statement coverage*) kertoo, kuinka monta prosenttia testattavan yksikön kaikista suoritettavista lauseista on suoritettu testauksessa (Homès, 2012, Glossary). Homès esittää lausekattavuuden mittauksesta yhdessä funktiossa yksinkertaisen esimerkkikuvion (Kuvio 3), joka näkyy seuraavalla sivulla.



Kuvio 3. Lausekattavuus (Homès, 2012, s. 175).

Kuviosta 3 nähdään, että harmaalla viivalla piirretty reitti, jota pitkin testauksessa on kuljettu, on käynyt läpi kaikki funktion lauseet. Kuvasta huomataan kuitenkin myös, että ensimmäinen päätöskohta on saanut vain toisen arvon, eikä ”x<0”-tilannetta ole testattu, vaikka lausekattavuus on 100 %.

Päätöskattavuus. Päätöskattavuus (*engl. decision coverage* tai *branch coverage*) tarkoittaa prosentuaalista osuutta kaikista päätöskohdista testattavassa ohjelmassa, jotka ovat saaneet testauksessa kaikki mahdolliset arvot (Homès, 2012, s. 181). Kun esimerkiksi funktiossa oleva ehtorakenne on saanut arvon tosi ja epätosi, niin kyseinen päätöskohta on saanut kaikki mahdolliset arvot.

Ehtokattavuus. Ehtokattavuus (*engl. condition coverage*) tarkoittaa prosentuaalista osuutta testauksessa suoritettavan ohjelman kaikista ehdoista, jotka ovat saaneet testauksessa arvon tosi ja epätosi (Homès, 2012, s .185). Jos päätöksessä on vain yksi ehto, niin silloin kattavuus on identtinen päätöskattavuuden kanssa. Yleensä kuitenkin ehtokattavuus on kattavampi koodikattavuusmittari kuin päätöskattavuus.

Muokattu ehto- ja päätöskattavuus. Muokattu ehto- ja päätöskattavuus (*engl. Modified condition and decision coverage*) tarkoittaa prosentuaalista osuutta kaikista päätöskohdista ja ehtokohdista testattavassa ohjelmassa, jotka ovat saaneet testauksessa kaikki mahdolliset arvot, mutta lisäksi otetaan huomioon se, että vaikuttavatko kaikki ehdot itsenäisesti päätöksiin (Homès, 2012, s. 185). Jos esimerkiksi jokin ehto on aina tosi, se ei vaikuta päätökseen vaan laskee MC/DC-koodikattavuutta.

Objektikoodikattavuus. Objektikoodikattavuus (*engl. objectcode coverage*) on prosentuaalinen osuus kaikista objektikoodin käskyistä, jotka on testauksessa suoritettu vähintään kerran, ja ehdoista, jotka ovat testauksessa saaneet sekä arvon tosi että epätosi (Lauterbach, 2022b, Object Code Coverage).

Seuraavassa NASA:n taulukossa (Taulukko 1) on vertailtu yksikkötestaustasolla käytettyjä koodikattavuusmittareita toisiinsa.

Taulukko 1. Koodikattavuusmittarien vertailu (Hayhurst & Veerhusen & Chilenski & Rierson, 2001, s. 7).

Koodikattavuus kriteeri	**Lausekattavuus**	**Päätöskattavuus**	**Ehtokattavuus**	**MC/DC**
Kaikki ohjelman aloitus- ja lopetuspisteet on käyty läpi vähintään kerran		X	X	X
Jokainen lause on käyty läpi vähintään kerran	X			
Jokaisen päätöskohdan kaikki vaihtoehdot on käyty läpi vähintään kerran		X		X
Jokainen ehto päätöksessä on saanut kaikki mahdolliset arvot vähintään kerran			X	X
Jokaisen ehdon päätöksessä on näytetty itsenäisesti vaikuttavan päätöksen lopputulokseen				X

Taulukosta 1 voidaan havaita, että korkean MC/DC-koodikattavuuden saavuttaminen vaatii kaikista kattavimman testauksen tekemistä. Objektikoodikattavuus, lausekattavuus, päätöskattavuus, ehtokattavuus ja MC/DC ovat yleensä yksikkötestaustasolla käytettäviä mittareita. Integraatiotestauksessa ja järjestelmätestauksessa ohjelmat ovat jo niin kompleksisia, ettei näitä matalan tason koodikattavuusmittareita yleensä käytetä.

Standardin ISO 26262 mukaisessa ohjelmistokehityksessä suositellaan mittamaan lausekattavuutta, päätöskattavuutta tai MC/DC-kattavuutta yksikkötestaustasolla (International Organization for Standardization (ISO), 2018d, s. 23). Se mitä koodikattavuusmittaria käytetään, riippuu muun muassa tuotteen tavoittelemasta ASIL-luokituksesta.

4.2 Arkkitehtuurisen tason koodikattavuusmittarit

Arkkitehtuurisella tasolla eli integraatiotestauksessa ei enää haluta mitata kattavuutta niin yksityiskohtaisesti, koska nämä mittaukset on jo tehty yksikkötestauksessa. Funktio- ja kutsukattavuutta mittaamalla voidaan todistaa, että ohjelmayksikköjen väliset rajapinnat on testattu kattavasti, sekä näyttää, että rajapinnat eivät sisällä käyttämättömiä funktiota. ISO 26262 -standardin mukaisessa ohjelmistokehityksessä suositellaan mittamaan funktiokattavuutta ja kutsukattavuutta integraatiotestauksessa (International Organization for Standardization (ISO), 2018d, s. 27).

Funktiokattavuus. Funktiokattavuus (*engl. function coverage*) on prosentuaalinen osuus kaikista funktioista, joita on testauksessa kutsuttu vähintään kerran (Lauterbach, 2022b, Function Coverage). Täysi funktiokattavuus ei takaa täyttä kutsukattavuutta, sillä jotain funktiota voidaan ohjelmakoodissa kutsua monessa paikassa, mutta voi olla, että testauksessa vain osa kutsuista käydään läpi (Shwetha, 2022).

Kutsukattavuus. Kutsukattavuus (*engl. call coverage*) on prosentuaalinen osuus kaikista ohjelman funktiokutsuista, jotka on testauksessa suoritettu vähintään kerran (Lauterbach, 2022b, Call Coverage). Täysi kutsukattavuus ei takaa täyttä funktiokattavuutta, koska voi olla, että jotain funktiota ei ole kutsuttu ohjelmakoodissa kertaakaan (Shwetha, 2022).

4.3 Koodikattavuus ISO 26262 -standardissa

ISO 26262 -standardi antaa alla olevan taulukon (Taulukko 2) mukaisen suosituksen eri koodikattavuusmittarien mittaukseen turvatuotteiden yksikkötestauksessa.

Taulukko 2. Rakenteellisen koodikattavuuden mittaus yksikkötestaustasolla (International Organization for Standardization (ISO), 2018d, s. 23).

Koodikattavuusmittari	**ASIL-taso**			
	A	**B**	**C**	**D**
Lausekattavuus	++	++	+	+
Päätöskattavuus	+	++	++	++
MC/DC	+	+	+	++

Taulukossa 2 ”++” tarkoittaa, että metodi on erittäin suositeltavaa kyseiselle ASIL-tasolle, ja ”+” tarkoittaa, että metodi on suositeltavaa kyseiselle ASIL-tasolle (International Organization for Standardization (ISO), 2018d, s. 3).

Taulukosta 2 voidaan nähdä, että kaikista turvakriittisimmän tason ASIL D tuotteelle on erittäin suositeltavaa MC/DC-koodikattavuuden mittaus yksikkötestauksessa, sillä se tarjoaa kaikista vahvimman takuun testauksen kattavuudesta. ASIL A- ja B-tasoa tavoitteleville tuotteille on erittäin suositeltavaa lausekattavuuden mittaus. ASIL A- ja B-tason tuotteille MC/DC-kattavuutta ei suositella yhtä vahvasti kuin lausekattavuutta, sillä täyden MC/DC-kattavuuden saavuttaminen on aikaa vievää ja se saattaisikin viedä turhaan resursseja pois tärkeämmistä asioista.

Integraatiotestauksessa ISO 26262 -standardi suosittelee turvatuotteille mitattavaksi funktio- ja kutsukattavuutta alla olevan taulukon mukaisesti.

Taulukko 3. Rakenteellisen koodikattavuuden mittaus arkkitehtuurisella tasolla (International Organization for Standardization (ISO), 2018d, s. 27).

Koodikattavuusmittari	**ASIL-taso**			
	A	**B**	**C**	**D**
Funktiokattavuus	+	+	++	++
Kutsukattavuus	+	+	++	++

Taulukosta 3 voidaan nähdä, että kaikkien eri ASIL-tasojen integraatiotestauksessa suositellaan mitattavaksi funktio- ja kutsukattavuutta, mutta erityisesti se on suositeltavaa ASIL C ja ASIL D -luokitusta tavoitteleville tuotteille.

4.4 Kuollut tai epäaktiivinen koodi

Kaikkea ohjelmiston lopullisen vaatimustestauksen ulkopuolelle jäävää koodia kutsutaan kuolleeksi tai epäaktiiviseksi koodiksi. Tämä koodi voidaan löytää koodikattavuusmittauksilla lopullisessa vaatimusten testauksessa (Ganssle, 2004, s. 135). Testauksen ulkopuolelle jäävä koodi voidaan jakaa neljään kategoriaan, jotka on esitelty alla olevassa taulukossa (Mackay, QA Systems, i.a.).

Taulukko 4. Testauksen ulkopuolelle jääneen koodin jaottelu (Mackay, QA Systems, i.a.).

Testauksen ulkopuolelle jääneen koodin kategoria	**Kuollut koodi**	**Epäaktiivinen koodi**	**Testaamaton ja turha koodi**	**Testaamaton, mutta tarpeellinen koodi**
Esimerkki	Funktio, jota ei ikinä kutsuta	Koodia ei käytetä tietyllä konfiguraatiolla	Vanhaa koodia projektissa, jota ei enää tarvita mihinkään	Koodi ei suoraan toteuta vaatimusta, mutta on muuten tarpeellinen
Ratkaisu	Poistetaan koodi tai kommentoidaan yli	Dokumentoitu selitys, miksi koodi on joissain tilanteissa epäaktiivista	Poistetaan koodi tai kommentoidaan se yli	Lisätään testitapauksia kattamaan koodi tai täydennetään vaatimusmäärittelyä

Kaikki taulukossa 4 nähtävä testauksen ulkopuolelle jäävä koodi on hyvä löytää mahdollisimman varhain testauksessa, ettei se vahingossa päädy lopputuotteeseen asti ja aiheuta myöhemmin ongelmia.

4.5 Hyödyt ja haasteet

Koodikattavuuden mittaaminen auttaa löytämään testaamattomia osia ohjelmistosta ja on yksi keino ohjelmiston laadun varmistamiseksi. Koodikattavuuden mittaus kohdistaa usein huomion matalan tason testaukseen ja auttaa yrityksiä ymmärtämään, kuinka hyvin ne on toteutettu (Craig & Jaskiel, 2002, Code Coverage Strengths).

Korkea koodikattavuus ei kuitenkaan takaa sitä, että ohjelmisto toimii oikein tai että se toimii asiakkaan haluamalla tavalla (Chopra, 2018, s. 202). Jos esimerkiksi päätöskattavuus on 90 %, mutta asiakkaiden usein käyttämää ominaisuutta ei ole testattu, voi ohjelmisto silti olla epäluotettava suurimman osan ajasta.

Koodikattavuusmittausta tulisi lähestyä riskipohjaisesti (Craig & Jaskiel, 2002, Code Coverage Weaknesses). Jos koodikattavuus on esimerkiksi 90 %, on tärkeää, että juuri ohjelman toiminnan kannalta kriittisimmät 90 % on testattu. Vaikka tavoitteena olisi 100 %:n koodikattavuus, on tärkeää testata ja korjata viat ensin kaikista kriittisimmistä moduuleista.

5 LAITEOHJELMISTON VIRHEENJÄLJITYS

Laiteohjelmiston virheenjäljitys vaatii yleensä vähintään kahden järjestelmän yhteistyötä: testattavan kohdelaitteen ja testaajan oman järjestelmän, jolla laiteohjelmistoa ohjataan, esimerkiksi testaajan kannettavan tietokoneen yhteistyötä (Dice, 2018, s. 73). Tällaista järjestelyä kutsutaan isäntä/kohde-virheenjäljitykseksi (*engl. host/target debugging*).

Visuaalinen tai auditiivinen indikaatio. Yksinkertaisin tapa jäljittää virheitä laiteohjelmistosta on käyttää hyväksi siinä kiinni olevaa näyttöä, lediä tai kaiutinta. Näiden avulla voidaan indikoida, että tiettyyn pisteeseen koodissa on ainakin päästy. Esimerkiksi lähes kaikissa Epecin ohjausyksiköissä on kiinteä ledivalo, jonka väriä voidaan tarvittaessa vaihtaa ohjelmallisesti.

Jäljityspuskuri. Jäljityspuskuri (*engl. trace buffer*) on tietty varattu muistialue, johon kirjoitetaan koodeja, kun laiteohjelmistossa tapahtuu merkittäviä tapahtumia (Ball, 1998, s. 31). Esimerkiksi jokin puskuriin kirjoitettava koodi voisi tarkoittaa sitä, että laiteohjelmisto on käynnistynyt uudelleen virtakatkon jälkeen. Jäljityspuskuria voidaan käyttää hyödyksi laiteohjelmiston virhetilanteessa, kun halutaan jälkeenpäin nähdä, mitä on tapahtunut laiteohjelmistossa ennen virhettä. Puskuria voidaan käydä läpi loppumerkistä taaksepäin, ja päästä vihille siitä, mikä virheen voisi aiheuttaa.

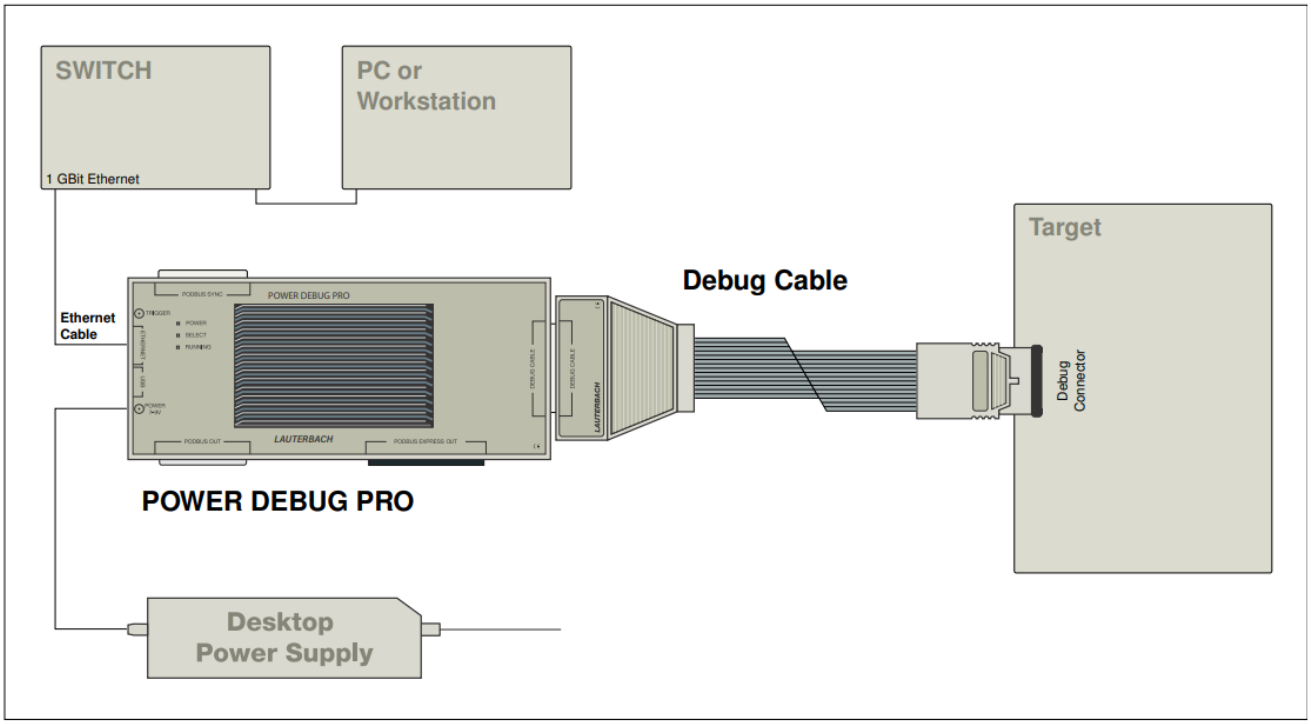
Sarjaväylä. Sarjaväylällä voidaan tuoda helposti tekstimuotoista dataa kehittäjän tietokoneelle, mikä antaa huomattavasti enemmän mahdollisuuksia laiteohjelmiston virheenjäljitykseen (Dice, 2018, s. 75). Tämä vaatii yleensä jonkinlaisen kaapeliyhteyden tai langattoman yhteyden kohdelaitteen ja isäntäkoneen välillä.

5.1 In-Circuit Debugger

In-Circuit Debugger eli lyhyemmin ICD on laite, joka kiinnitetään sekä kohdelaitteeseen että isäntäkoneeseen (Ibrahim, 2015, s. 80). Sen avulla isäntäkone voi hallita laiteohjelmiston ajoa kohteen käynnistymisestä sen sammumiseen. ICD kiinnitetään yleensä kohdelaitteeseen JTAG-liitännällä. Tässä työssä termi ICD on suomennettu laitteistodebuggeriksi.

Laitteistodebuggerilla voidaan esimerkiksi asettaa laiteohjelmiston lähdekoodiin tietylle riville pysäytyspiste (*engl. breakpoint*), johon suoritus pysähtyy, kun ohjelman ajo saapuu kyseiselle riville (Ibrahim, 2015, s. 80). Tästä eteenpäin lähdekoodia voidaan käydä hallitusti läpi rivi tai funktio kerrallaan tai pysähtyä seuraavaan pysähdyspisteeseen. Laitteistodebuggerilla voidaan myös lukea muuttujien ja rekisterien arvoja. Laitteistodebuggeri tarjoaa kaikista parhaat keinot bugien löytämiseen laiteohjelmistosta, ja sen käyttö on hyvin samankaltaista kuin ohjelmallisen virheenjäljittimen.

Tässä työssä laitteistodebuggeria käytetään koodikattavuuden mittaukseen integraatiotestauksessa. Alla on esimerkkikuvio (Kuvio 4) tässä työssä käytettävästä Lauterbachin laitteistodebuggerista.



Kuvio 4. Lauterbachin PowerDebug PRO -laitteistodebuggeri (Lauterbach, 2022j, s. 11).

Kuvion 4 laitteistodebuggeri on yhteydessä isäntätietokoneeseen joko USB-kaapelilla tai Ethernet-kaapelilla ja kohdelaitteeseen esimerkiksi JTAG-liitännällä. Laitteistodebuggerilla on oma virtalähteensä, ja lisäksi siihen saa kiinnitettyä lisälaitteita PODBUS-liitännällä.

5.2 In-Circuit Emulator

In-Circuit Emulator eli lyhyemmin ICE on hieman eri laite kuin laitteistodebuggeri, ja eron hyvin selittävää lähdettä oli melko hankala löytää. Usein nämä kaksi termiä saattavatkin mennä hieman sekaisin. Liioittelevan tai valheellisen markkinoinnin vuoksi joidenkin tuotteiden nimissä voikin esiintyä lyhenne ICE, vaikka oikeasti nimessä pitäisi olla lyhenne ICD.

ICE emuloi eli jäljittelee kohdelaitteen prosessoria, eikä siksi kohdelaitteessa tarvitse välttämättä vielä olla toimivaa prosessoria (Ibrahim, 2015, s. 80–81). Emulaattori tarjoaa samat mahdollisuudet ohjelmiston läpikäyntiin virheenjäljityksessä kuin ICD, mutta ajettava koodi suoritetaan emulaattorin sisällä eikä se siten vaikuta suoraan kohdelaitteen toimintaan, toisin kuin laitteistodebuggeri, joka käyttää esimerkiksi jonkin verran kohdelaitteen muistia. ICE sopii parhaiten uusien tai viallisten laitteiden virheenjäljitykseen.

5.3 Ohjelmallinen virheenjäljitys

Yksinkertaisimmillaan ohjelmallinen virheenjäljitys voi tarkoittaa vain printf-tulostuksia virheen jäljittämiseksi (Dice, 2018, s. 76). Konsolitulosteilla voidaan nähdä muuttujien arvot joka rivillä ja päästä käsiksi siihen, miksi ohjelma esimerkiksi kaatuu. Konsolitulosteita voi tehostaa abstrahoinnilla. Dice antaa esimerkiksi yksinkertaisen kirjoitusoperaation muistiin: sen sijaan, että kirjoitetaan operaatiot joka kerta uudestaan, luodaan muistikirjoitusta varten funktio, joka automaattisesti tulostaa myös muuttujien arvot konsoliin.

Dicen (2018, s. 76) mukaan monesti unohdettu keino virheenjäljityksessä on ottaa kääntäjän optimoinnit pois päältä. Kun kääntäjä optimoi koodia, se saattaa siirtää sitä tai hypätä joidenkin rivien yli. Virheenjäljityksessä tämä ei ole hyvä asia, koska optimoinnin seurauksena muuttujan arvo voi olla eri kuin mitä koodia lukemalla voisi ajatella.

6 LÄHTÖTILANNE JA MENETELMÄT

Epecin laiteohjelmistojen kehityksessä on tähän asti käytetty laitteistodebuggereita lähinnä virheenjäljitykseen hankalien ongelmien tai virheiden kanssa. Käytössä olevissa laitteistodebuggereissa on kuitenkin mahdollisuutena myös jäljittää ja tallentaa koodikattavuutta testiajon aikana isäntäkoneelle tiedostoihin, joista voidaan testiajon jälkeen luoda koodikattavuusraportit. Tällä tavalla koodikattavuutta haluttaisiin mitata erityisesti turvaluokiteltujen tuotteiden integraatiotestauksessa ISO 26262 -standardin vaatimalla tavalla.

6.1 Työn toteutukseen käytettävä ohjausyksikkö

Epecin SC52-ohjausyksikköä käytetään tämän työn käytännön osuuden koodikattavuusmittaukseen. SC52-ohjausyksikkö on Epecin turvatuotteisiin kuuluva tuote, joka on saanut SIL2-luokittelun mukaisen sertifikaatin (Epec, 2022c).

Kuva 1. Epecin SC52-turvatuote (Epec, 2022c).

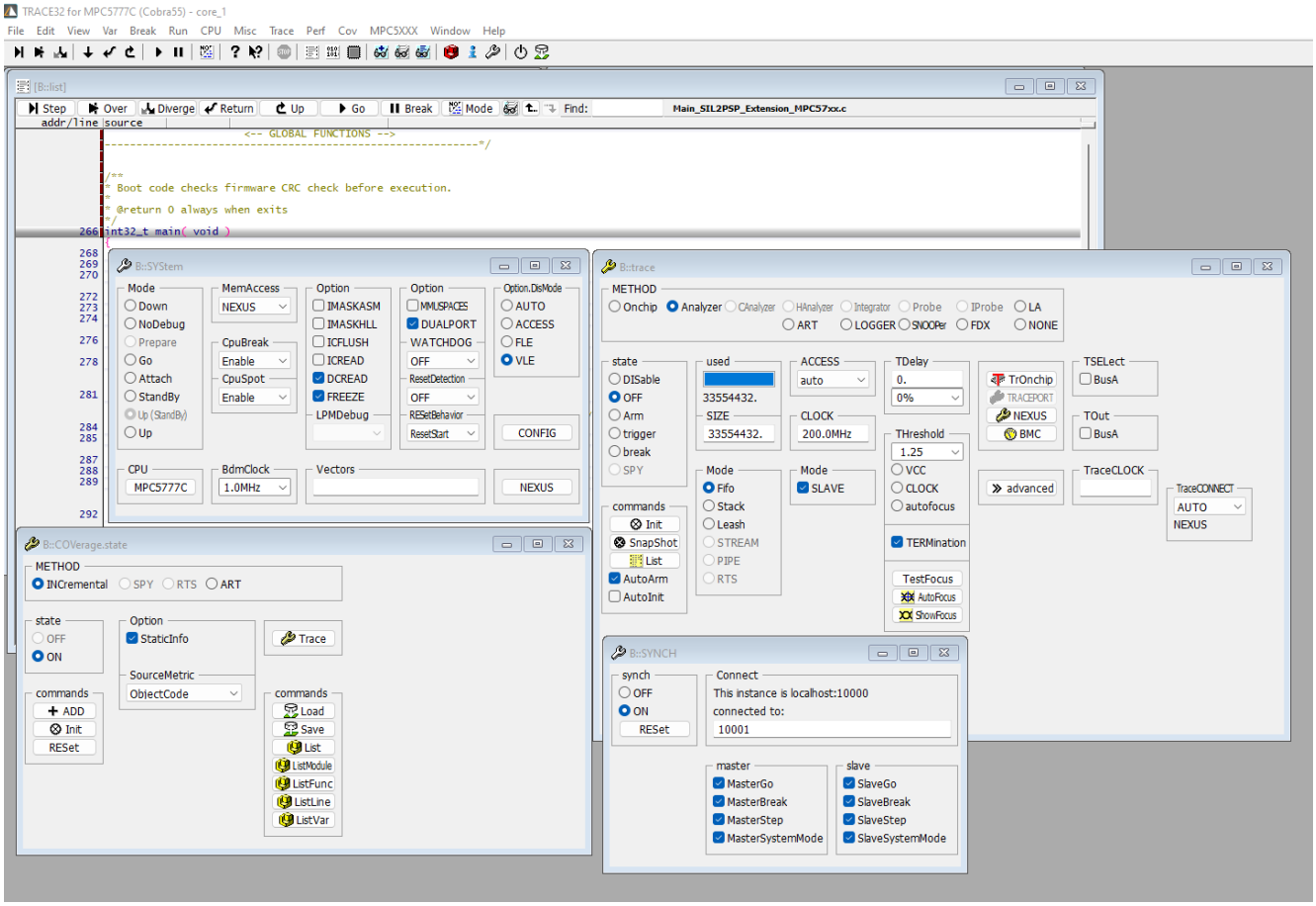
Epecin turvatuotteessa on päällä keltainen tarra. Ohjausyksikössä on 32-bittinen moniydinprosessori, jossa on lockstep- ja muistisuojaus (Epec, 2022c). Ohjausyksikössä IO-pinnien lukumäärä on 22 (14 tuloa ja 8 lähtöä).

Epecin SC52-turvatuotteessa on niin sanottu ”turvaydin” (sisäinen tietolähde, 14.3.2023). Sisäisesti se käyttää kahta e200z7-ydintä ja vertailee niiden suorituksen tuloksia keskenään. Laiteohjelmisto ei pysty erikseen muuttamaan, tai käyttämään vain toista ydintä näistä kahdesta. Molemmat e200z7-ytimet ovat automaattisesti päällä ja suorittavat aina saman koodin.

Epecin tapauksessa tämän työn tekijää ja toimeksiantajaa kiinnostaa koodikattavuuden mittauksessa vain se, mitä tapahtuu turvaytimessä, koska kaikki ohjelmakoodi ajetaan siellä. Lisäksi SC52:ssa on myös yksi tavallinen ydin, jota ei käytetä, eikä siksi sen koodikattavuuttakaan kannata mitata.

6.2 Trace32

Trace32 on Lauterbachin tuoteperhe, joka sisältää erilaisia kehitystyökaluja mikroprosessorien kehittämiseen ja virheenjäljitykseen. Tuoteperheeseen kuuluu mm. laitteistodebuggereita ja työkaluja reaaliaikaiseen jäljitykseen (Lauterbach, 2022d). Tässä työssä käytettävä laitteistodebuggeri ja PowerTrace-lisäosa kuuluvat Trace32-tuoteperheeseen. Tuoteperheen laitteiden ohjaukseen käytettävä ohjelmisto sekä graafinen käyttöliittymä on nimeltään PowerView. Seuraavalla sivulla on esimerkkikuvio käyttöliittymästä (Kuvio 5). Kuvassa on asetettu pysäytyspiste laiteohjelmiston pääfunktioon, jonne suoritus pysähtyy automaattisesti kohdelaitteen käynnistyksessä virtakatkon jälkeen.

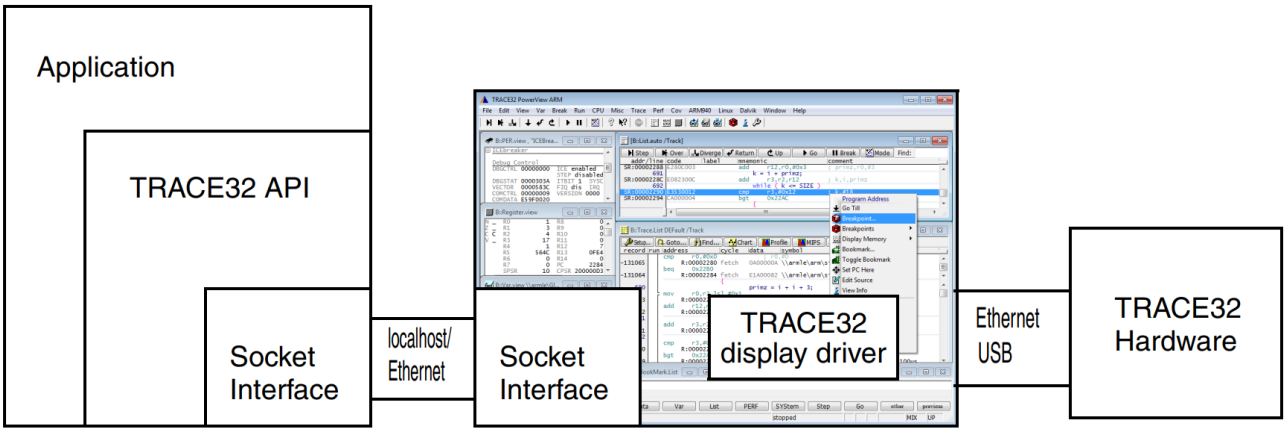


Kuvio 5. Trace32-käyttöliittymä.

Tässä työssä tavoitteena on automatisoida Trace32-ohjelmiston käyttöä koodikattavuuden mittauksessa skriptejä ja ohjelmiston ulospäin tarkoitettuja rajapintoja hyödyntäen. Automatisoinnissa on tarkoitus hyödyntää Pythonilla ja LabVIEW:llä tehtyjä skriptejä, joilla Trace32-ohjelmaa voidaan komentaa rajapinnan kautta. Rajapinnan kautta päästään käsiksi samoihin komentoihin, joita käyttäjä voi valita käyttöliittymän kautta.

6.3 Trace32-etärajapinta

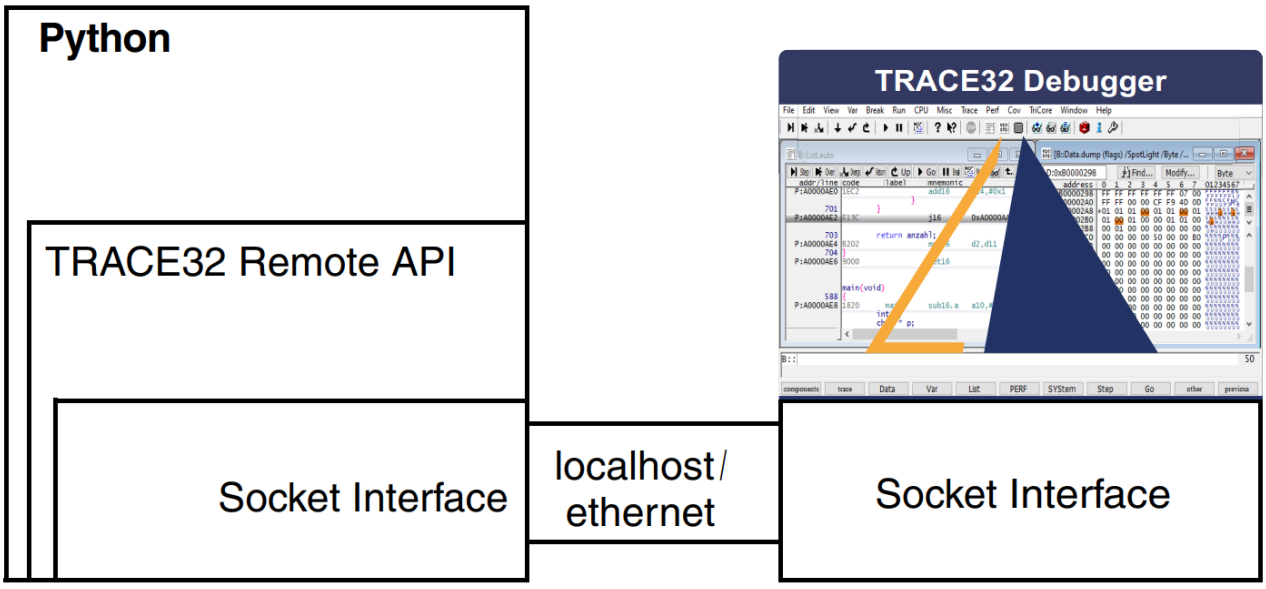
Trace32-ohjelmaa voidaan komentaa ulkopuolelta etärajapinnan kautta (Lauterbach, 2022a, System Configuration Overview). Tämä Lauterbachin kirjoittama etärajapinta on kirjoitettu C-ohjelmointikielellä, ja se muodostaa yhteyden Trace32-ohjelman kanssa pistokerajapintaa käyttäen. Seuraavalla sivulla on Lauterbachin havainnollistava kuvio etärajapinnan toiminnasta (Kuvio 6).



Kuvio 6. Trace32-etärajapinnan toiminta (Lauterbach, 2022a, s. 9).

Kuviosta 6 nähdään, että ylimmällä tasolla on C-ohjelma, josta kutsutaan etärajapinnan funktioita. Etärajapinta kommunikoi pistokerajapinnan kautta komentoja Trace32-ohjelmalle, ja Trace32-ohjelma taas välittää käskyt USB- tai Ethernet-yhteyden kautta edelleen laitteistodebuggerille. Laitteistodebuggeri pystyy esimerkiksi JTAG-yhteyden kautta hallitsemaan prosessorin toimintaa kohdelaitteessa. Pistokerajapinta käyttää kommunikointiin joko TCP- tai UDP-protokollaa.

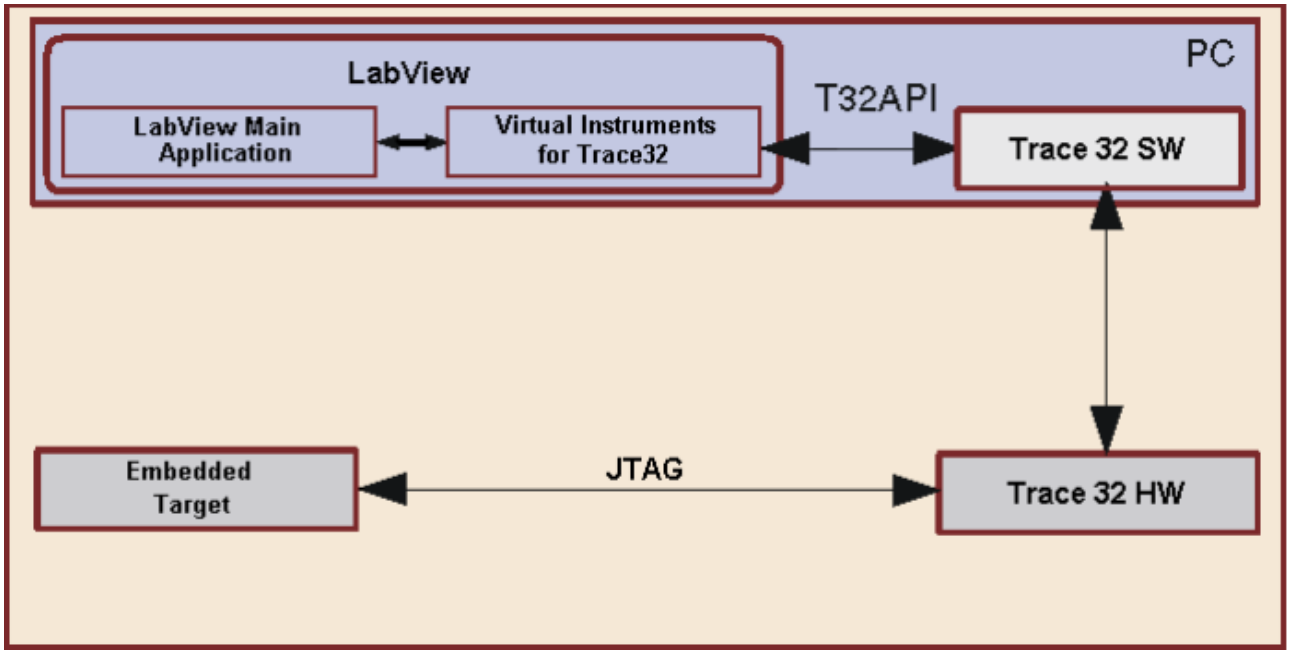
Trace32-etärajapintaa voidaan käyttää myös Python-ohjelman kautta. Alla on havainnollistava kuvio (Kuvio 7).



Kuvio 7. Trace32, Python-etärajapinta (Lauterbach, 2022c, Introduction).

Kuviosta 7 nähdään, että käytännössä Python-ohjelma korvaa kuviossa 6 esiintyneen C-ohjelman.

Laiteohjelmiston testaukseen käytettävää testipenkkiä ohjataan ensisijaisesti National Instrumentsin TestStand-testinhallintaohjelmistolla. Trace32 tarjoaa rapapinnan LabVIEW-ohjelmointiympäristöön, mitä TestStand käyttää hyödykseen (Lauterbach 2022e, Overview). Rajapinta toimii hyvin samankaltaisesti kuin Python-rajapinta. Alla on havainnollistava kuvio rajapinnan toiminnasta (Kuvio 8).



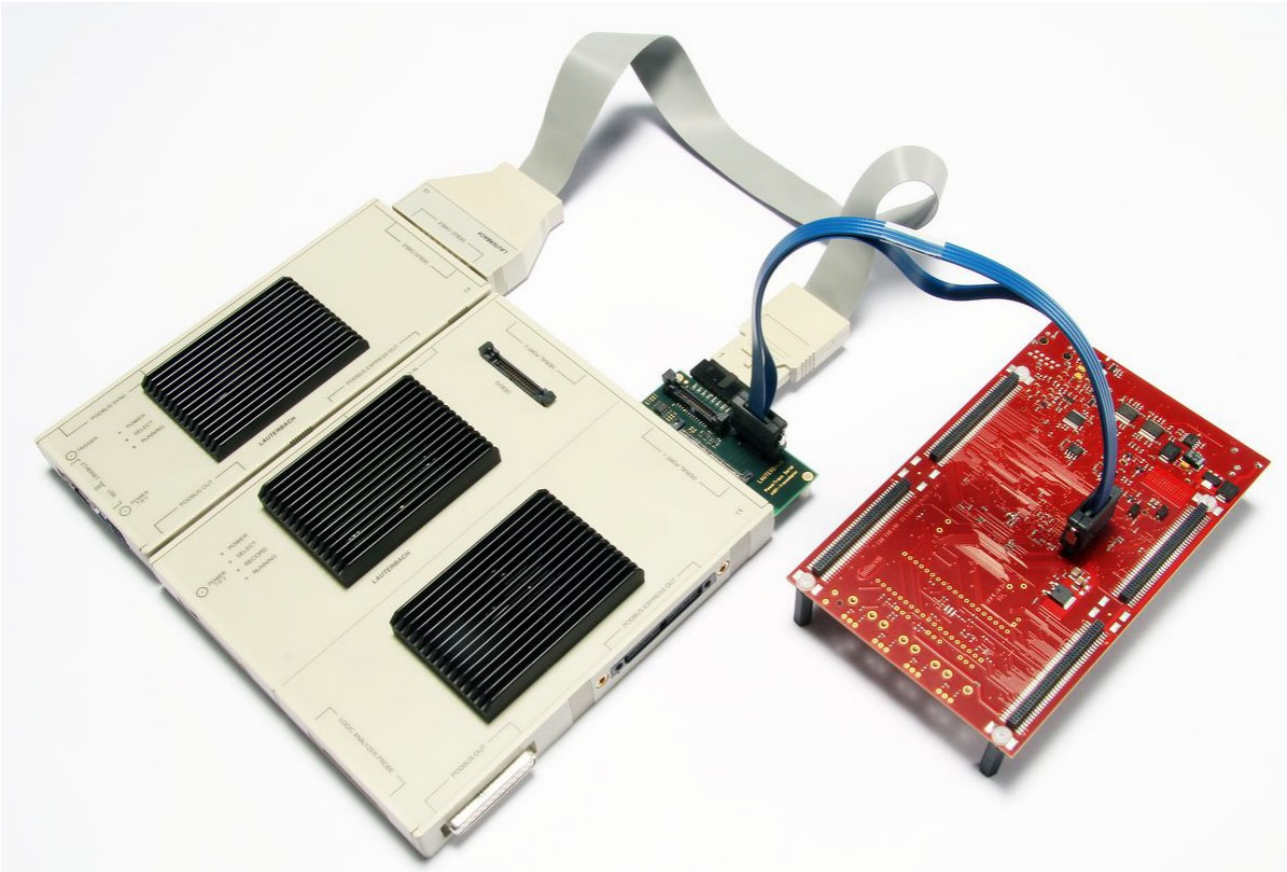
Kuvio 8. Trace32, LabVIEW-rajapinta (Lauterbach 2022e, Overview).

Kuviosta 8 nähdään, että LabVIEW-ohjelma kommunikoi Trace32-ohjelman kanssa etärajapinnan kautta samaan tapaan kuin Python-ohjelma kuviossa 7 (Lauterbach 2022e, Overview). LabVIEW-ohjelmoinnissa käytetään graafisia blokkeja ohjelmointiin. Yhtä kokonaista blokkiohjelmaa tai aliohjelmaa kutsutaan ”Virtual Instrument” -nimellä. Trace32 tarjoaa erilaisia Virtual Instrument -aliohjelmia, joilla Trace32-ohjelmaa voidaan komentaa samaan tapaan kuin Pythonilla. Ainoa merkittävä ero on se, että LabVIEW-rajapinta voi muodostaa yhteyden Trace32-ohjelmaan ainoastaan UDP-protokollalla, kun taas Python-ohjelma voi muodostaa yhteyden myös TCP-protokollalla.

6.4 PowerDebug PRO ja PowerTrace

Koodikattavuuden mittaukseen käytettävä laitteistodebuggeri on Lauterbachin PowerDebug PRO. Lisäksi nimenomaan koodikattavuuden mittaukseen tarvitaan PowerTrace II tai PowerTrace III -moduuleja, joiden avulla pystytään jäljittämään ajonaikaista tietoa ja lähettämään sitä isäntäkoneelle myöhempää analyysiä varten (Lauterbach, 2022i, What is Trace?).

Trace-moduuli siis tallentaa ajon aikana ”jälkiä” siitä, missä kaikkialla koodissa on kuljettu, ja lähettää tätä tietoa ajon aikana isäntäkoneelle väliaikaistiedostoon. Voidaan esimerkiksi tallentaa tieto siitä, missä kaikissa funktioissa on käyty. Riippuen käytettävästä datan keräystavasta, voidaan tästä väliaikaistiedosta muodostaa koodikattavuusraportti joko testauksen jälkeen tai sen aikana (Lauterbach 2022b, s. 11). Alla on esimerkkikuva (Kuva 2) PowerDebug PRO -laitteistodebuggerista ja PowerTrace-moduulista.



Kuva 2. Lauterbachin PowerDebug Pro ja PowerTrace-moduuli (Lauterbach, 2022f, s. 9).

Kuvassa 2 Trace- ja virheenjäljityssignaalit on yhdistetty yhteen JTAG-kaapeliin adapterin avulla. JTAG-kaapeli taas on kiinni kohdelaitteessa.

6.5 PRACTICE-skriptit

PRACTICE-skriptien avulla voidaan automatisoida Trace32-ohjelmiston käyttöä ja automatisoida esimerkiksi laitteistodebuggerin asetuksien säätäminen tietylle kohdelaitteelle (Lauterbach, 2022h, s. 3). PRACTICE on riviorientoitunut testauskieli, jota voidaan käyttää yleisimpien digitaalimittaustekniikan ongelmien ratkaisuun. Alla on esimerkki yksinkertaisesta PRACTICE-skriptistä (Kuvio 9).

```
; Script containing two script calls

PRINT "Start"
DO modul1                       ; Execute script module 1
DO modul2                       ; Execute script module 2
                                ; the file extension (*.cmm) can be
ENDDO                           ; omitted
```

Kuvio 9. Trace32, PRACTICE-skriptit (Lauterbach, 2022h, Script Nesting).

Kuvion 9 skripti tulostaa ensin Trace32-ikkunaan ”Start” ja sitten aloittaa ”modul1”-nimisen aliskriptin suorituksen. Tämän jälkeen suoritetaan aliskripti ”modul2”, ja sitten skriptin suoritus loppuu. PRACTICE-skriptin tiedostopääte on ”cmm”.

6.6 Lähtötilanteen skriptit

Lähtötilanteessa Epecillä oli valmiita skriptejä, jotka mm. käynnistävät Trace32-instanssit, asettavat prosessorin ytimet virheenjäljitystilaan sekä lataavat laiteohjelmiston ja virheenjäljityssymbolit. Skriptit on muokattu Epecin tarpeisiin Lauterbachin esimerkkiskripteistä. Alla on skripti, joka käynnistää Trace32-instanssin yhdelle ytimelle (Kuvio 10).

start_debug-usb.bat | start_cores.cmm | config_mc.t32 | mpc5777C_debug.cmm

```
1  SET t32path=C:\T32\bin\windows64
2  IF NOT EXIST %t32path%\t32mppc.exe SET t32path=c:\t32\bin\windows
3  IF NOT EXIST %t32path%\t32mppc.exe SET t32path=c:\t32\
4  start %t32path%\t32mppc -c c:\T32\config_mc.t32 10000 TRACE32_A USB CORE=1 -s mpc5777c_debug.cmm
5
6
7
```

Kuvio 10. Trace32-instanssin käynnistysskripti.

Skripti etsii ensin Trace32-sovelluksen polun ja sitten käynnistää Trace32-sovelluksen (Lauterbach, 2022j, s. 61). C-lippu tarkoittaa, että Trace32 käyttää konfiguraatiotiedostona *config_mc.t32*-tiedostoa ja konfiguraatiotiedostolle annetaan neljä parametria. S-lipulla annetaan käynnistysskripti *mpc5777c_debug.cmm*, jonka Trace32 ajaa heti käynnistymisensä jälkeen. Konfiguraatiotiedostossa asetetaan asetuksia Trace32-sovellukselle, ja siitä on kuva alla (Kuvio 11).

config_mc.t32 | start_debug-usb.bat | mpc5777C_debug.cm

```
1
2   IC=NETASSIST
3   PORT=${1}
4
5   ; Environment variables
6   OS=
7   ID=T32${1}
8   TMP=C:\t32\temp
9   SYS=C:\t32
10  HELP=c:\t32\pdf
11
12  ; USB information
13  PBI=
14  USB
15
16  ; Printer settings
17  PRINTER=WINDOWS
18
19  ; Screen fonts
20  SCREEN=
21  FONT=SMALL
22  HEADER=Trace32 ${2}
23
24  RCL=NETTCP
25  PORT=20000
26
```

Kuvio 11. Trace32, moniydinkonfiguraatioasetukset.

Konfiguraatiotiedostossa asetetaan ensin ”InterCom”-portti kahden Trace32-instanssin väliseen sisäiseen kommunikaatioon. Sisäisen kommunikaation portti saa kuviossa 10 annetun ensimmäisen parametrin arvon eli 10 000. Seuraavaksi asetetaan ympäristömuuttujien arvoja. PBI-kentässä ilmoitetaan, miten laitteistodebuggeri on yhdistetty isäntäkoneeseen, tässä tapauksessa USB-yhteydellä (Lauterbach, 2022j, Section PBI). RCL-kentän alle annetaan portti, jonka kautta Python-ohjelma kommunikoi Trace32-sovelluksen kanssa, kuten nähtiin jo aiemmin kuviossa 7. Python-ohjelma kommunikoi Trace32-sovelluksen kanssa TCP-yhteyden kautta.

Kuviossa 10 nähtiin, että Trace32-instanssille annettiin käynnistysskriptiksi *mpc5777c_debug.cmm*-tiedosto. Tämä käynnistysskripti kutsuu ensimmäisenä aliskriptiä *start_cores.cmm*, joka käynnistää toisen Trace32-instanssin toiselle ytimelle.

start_cores.cmm | start_debug-usb.bat | config_mc.t32 | mpc5777C_debug.cmm

```
&nodename=NODENAME()
&intercomport_core0=FORMAT.DECIMAL(1.,intercom.port())
&intercomport_core1=FORMAT.DECIMAL(1.,intercom.port()+1)

&configparams="NET NODE=&nodename PACKLEN=1024"
if "&nodename"=="" ;no IP address to debug interface -> USB connection
(
  &configparams="USB"
)

;use executable path the same es for current Trace32 instance
&patht32mppc=OS.PED()+"/t32mppc"
&patht32mtpu=OS.PED()+"/t32mtpu"

if !intercom.ping(localhost:&intercomport_core1)
(
  OS &patht32mppc -c config_mc.t32 &intercomport_core1 TRACE32_B &configparams CORE=2
)

INTERCOM.WAIT localhost:&intercomport_core1


ENDDO
```

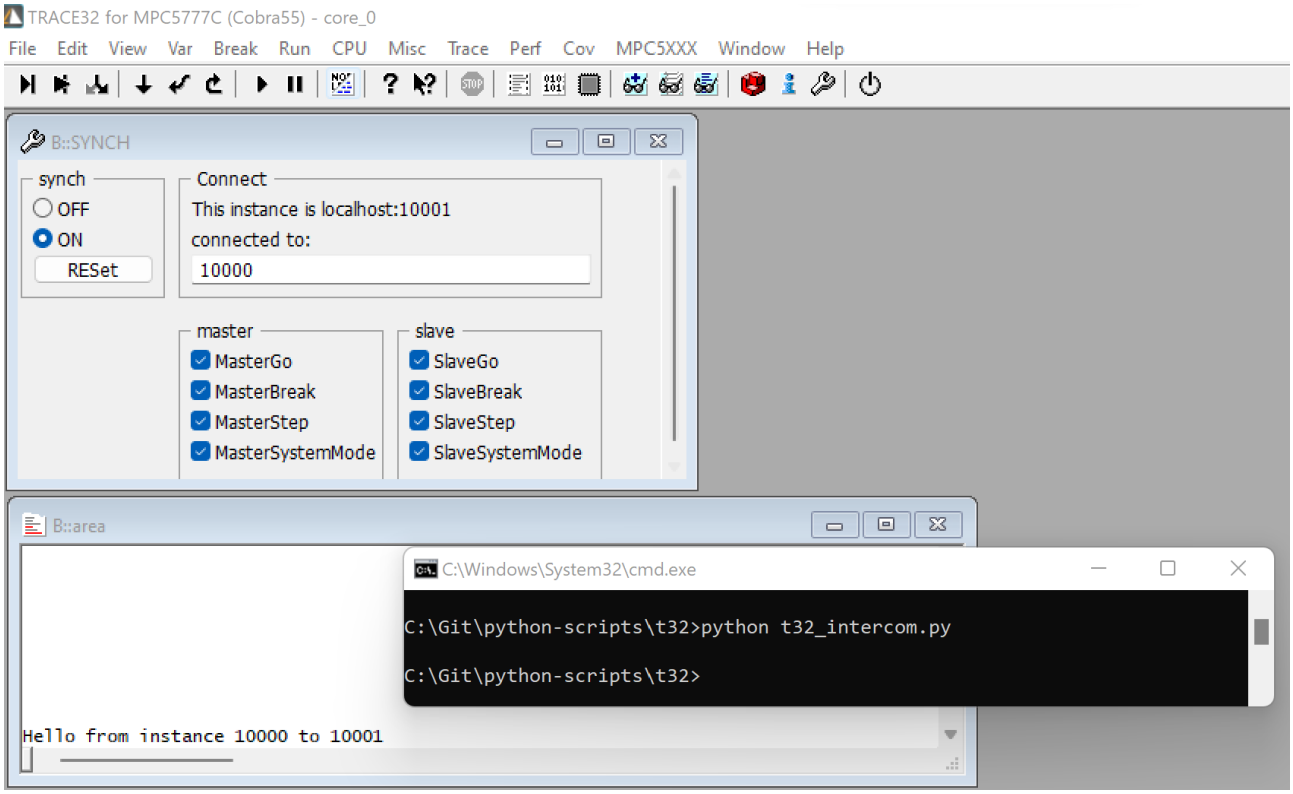
Kuvio 12. Toisen Trace32-instanssin käynnistäminen.

Kuviossa 12 asetetaan InterCom-portit Trace32-instanssien väliseen kommunikointiin, jotka saavat arvot 10 000 ja 10 001. Sitten käynnistetään toinen instanssi samaan tapaan kuin ensimmäinen. Käynnistyksen jälkeen odotetaan, että toinen Trace32-instanssi on valmiustilassa ja palataan takaisin mpc5777c_debug-skriptiin, missä asetetaan loput asetukset molemmille ytimille (Lauterbach, 2022g, InterCom.WAIT). Muita tehtäviä asetuksia ovat esimerkiksi virheenjäljityssymbolien lataus, Trace32-ikkunoiden asetukset ja Trace32-instanssien välisen kommunikaation asetukset.

6.7 Python-ohjelman kokeilu

Ensimmäinen käytännön vaihe oli testata yhteyden ottamista Trace32-instanssiin Python-ohjelmalla. Trace32-instanssit käynnistetään ensin edellisessä luvussa nähdyillä käynnistysskripteillä. Trace32-ohjelman Python-paketti löytyy sovelluksen asennuskansiosta ja se voidaan asentaa pip-paketinhallintatyökalulla.

Ensimmäinen isompi ongelma oli, että Pythonin kautta haluttiin komentaa SC52-ohjausyksikön turvaydintä, mutta käynnistysskriptissä turvaytimen Trace32-instanssi käynnistettiin vasta tavallisen ytimen jälkeen. Tämä johti tilanteeseen, jossa Pythonin kautta annetut komennot eivät menneet suoraan turvaytimelle vaan tavalliselle ytimelle. Ensimmäinen ratkaisuidea oli käyttää Trace32-instanssien välistä sisäistä kommunikointia eli InterComia (Lauterbach, 2022g, InterCom.execute). Tällöin Python-ohjelma käskyttää toisen ytimen käskyttämään turvaydintä InterCom-komennolla. Alla on esimerkkikuva tilanteesta (Kuvio 13).



Kuvio 13. Trace32, InterCom-komennon toiminta.

Kuviossa 13 lähetetään Python-ohjelman kautta viesti Trace32-instanssilta 10 000 instanssille 10 001. Alla on vielä kuva Python-koodista (Kuvio 14).

Kuvio 14. InterCom-komennon ajo Python-skriptistä.

Kuvion 14 skriptissä otetaan ensin käyttöön Trace32-asennuspaketti. Sitten muodostetaan TCP-yhteys ja lähetetään InterCom-komento Trace32-instanssille 10 000, joka lähettää komennon edelleen instanssille 10 001.

Yllä esitellyssä tavassa komentaa turvaydintä toisen Trace32-instanssin kautta on kuitenkin se huono puoli, ettei se pysäytä Python-ohjelman suoritusta vaan Python-ohjelma jatkaa suoritustaan eikä odota mitään paluuviestiä turvaytimen Trace32-instanssista. Tämän takia muokattiin Trace32-instanssien käynnistysskriptejä niin, että turvaydin käynnistetään ensin ja toinen ydin vasta sen jälkeen. Tällöin Pythonin kautta annetut komennot pysäyttävät Python-ohjelman suorituksen ja Python-ohjelma jää odottamaan komennon suoritusta loppuun ennen kuin se etenee.

6.8 Robot Framework

Robot Framework on Pythonilla kirjoitettu, avainsanoihin perustuva avoimen lähdekoodin automaatiokehikko (Robot Framework Foundation, 2023, 1.1 Introduction). Robot Frameworkiä käytetään koodikattavuusmittauksen havainnollistamiseen tässä työssä, koska sillä voidaan luoda helposti ymmärrettäviä testitapauksia, ja se tarjoaa avuksi paljon valmiita testikirjastoja. Robot Framework myös luo testauksesta automaattisesti HTML-raportin ja HTML-lokitiedoston, joista testauksen etenemistä on helppo tarkastella jälkeenpäin.

6.8.1 Omien testikirjastojen luominen

Python-kielellä voidaan luoda omia testikirjastoja, joita Robot Framework voi käyttää testien ajamisessa tai automatisoinnissa (Robot Framework Foundation, 2023, 4.1.2 Creating test library class or module). Yleensä Python-kirjasto on käytännössä Python-luokka. Kaikkia Python-luokan metodeja, joiden ensimmäinen kirjain ei ole alaviiva, voi kutsua Robot Frameworkin kautta vastaavalla avainsanalla. Alla on esimerkki (Kuvio 15) Python-luokasta, jossa on kaksi metodia.

```
class MyLibrary:

    def my_keyword(self, arg):
        return self._helper_method(arg)

    def _helper_method(self, arg):
        return arg.upper()
```

Kuvio 15. Python-luokan metodeita vastaavat avainsanat (Robot Framework Foundation, 2023, 4.1.3 Creating keywords).

Kuviossa 15 nähdään kaksi Python-luokan metodia. Metodia ”my_keyword” kutsuttaisiin Robot Frameworkin kautta avainsanalla ”My Keyword”. Metodia ”_helper_method” ei voi kutsua Robot Frameworkin kautta, sillä sen nimi alkaa alaviivalla.

6.8.2 Omien avainsanojen luominen

Tarvittaessa voidaan luoda omia avainsanoja, joita testitapaukset käyttävät avukseen (Robot Framework Foundation, 2023, 2.7 Creating user keywords). Avainsanat kirjoitetaan usein omaan resurssitiedostoon eli eri tiedostoon kuin testitapaukset. Avainsanat voivat käyttää apunaan Robot Frameworkin mukana tulevia avainsanoja, aiemmin luotuja omia avainsanoja tai erilaisten testikirjastojen avainsanoja. Seuraavalla sivulla on esimerkkikuvio (Kuvio 16) omien avainsanojen luomisesta.

```
*** Keywords ***
Open Login Page
    Open Browser    http://host/login.html
    Title Should Be    Login Page

Title Should Start With
    [Arguments]    ${expected}
    ${title} =    Get Title
    Should Start With    ${title}    ${expected}
```

Kuvio 16. Omien avainsanojen luominen Robot Frameworkillä (Robot Framework Foundation, 2023, 2.7.1 User keyword syntax).

Kuviossa 16 luodaan kaksi uutta avainsanaa: ”Open Login Page” ja ”Title Should Start With”. Käytännössä ensimmäinen avainsana avaa nettisivun ja tarkistaa, että otsikko on ”Login Page” ja toinen avainsana tarkastaa, että nettisivun otsikko alkaa tietyllä merkkijonolla.

6.8.3 Testitapausten luominen

Kun on luotu testauksen vaatimat testikirjastot ja avainsanat, voidaan siirtyä tekemään testitapauksia. Testitapaukset käyttävät testikirjastojen ja resurssitiedostojen avainsanoja avukseen (Robot Framework Foundation, 2023, 2.2.1 Test case syntax). Testitapausten syntaksi on hyvin samanlainen kuin avainsanojen. Alla on esimerkki testitapauksesta (Kuvio 17).

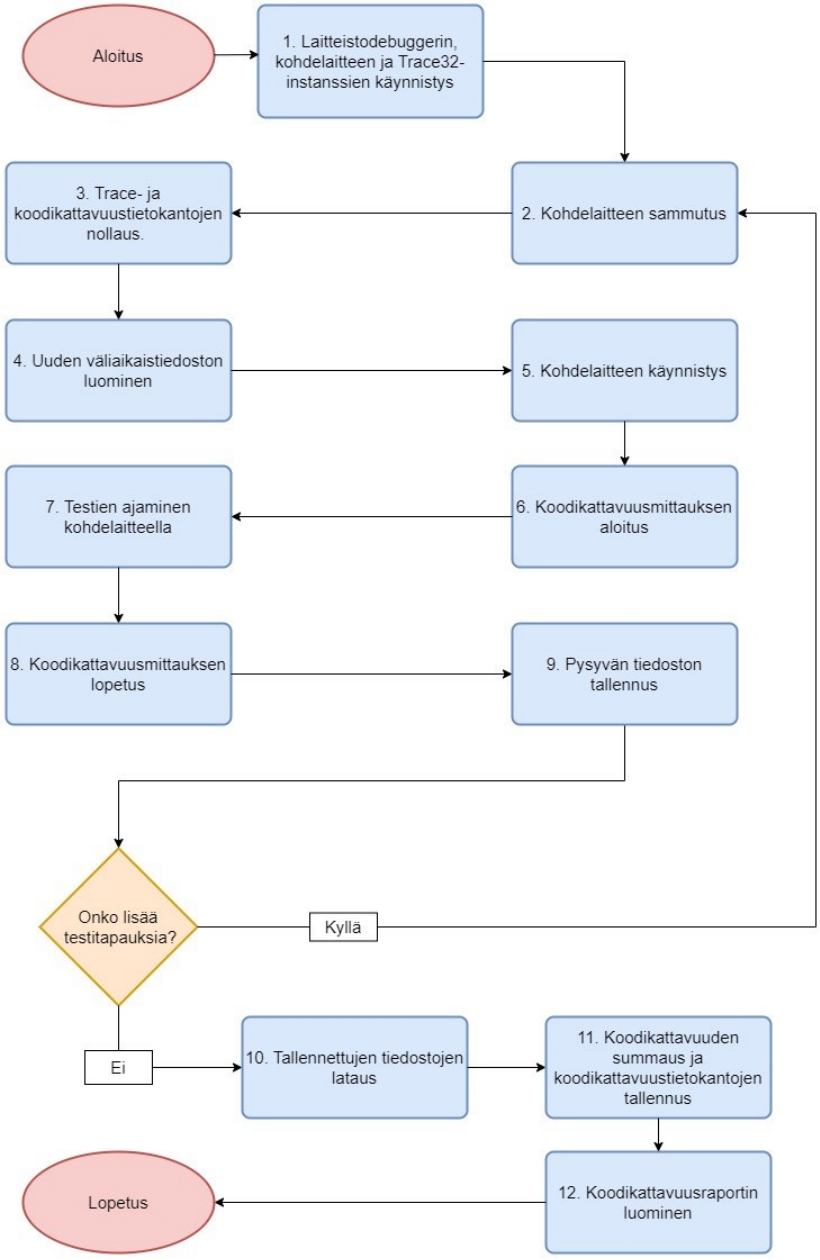
```
*** Test Cases ***
Valid Login
    Open Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
```

Kuvio 17. Testitapausten luominen Robot Frameworkillä (Robot Framework Foundation, 2023, 2.2.1 Test case syntax).

Kuviossa 17 on luotu testitapaus nimeltä “Valid Login”. Testitapauksessa tarkastetaan, että nettisivulle päästään kirjautumaan sisään käyttäjätunnuksella ”demo” ja salasanalla ”mode”.

7 KOODIKATTAVUUDEN MITTAUS

Ennen laiteohjelmiston testauksen aloittamista on tärkeää suunnitella sitä, missä järjestyksessä asioita tehdään ja mitä komentoja laitteistodebuggerille annetaan etärajapinnan kautta testauksen aikana. Seuraavassa kuviossa (Kuvio 18) havainnollistetaan koodikattavuuden mittausta vuokaavion avulla, josta näkyvät tarvittavat prosessit ja niiden väliset yhteydet. Kuviosta nähdään selkeästi, miten koodikattavuuden mittaus kannattaa automatisoida.



Kuvio 18. Koodikattavuusmittauksen vuokaavio.

Kuviosta 18 nähdään, että ensin on käynnistettävä laitteistodebuggeri. Se voidaan tehdä ohjelmallisesti esimerkiksi ohjelmoitavalla jatkojohdolla, tai sen voi tehdä manuaalisesti testi-insinööri. Seuraava vaihe on kohdelaitteen käynnistys eli virran päälle kytkeminen. Epecin tapauksessa testipenkkiä ohjaava TestStand-ohjelmisto huolehtii kohdelaitteen käynnistyksestä ja sammutuksesta. Trace32-instanssit käynnistetään jo aiemmin luvussa 6.6 esitellyillä käynnistysskripteillä. Seuraavia vaiheita käydään vähän tarkemmin läpi käyttämällä havainnollistamisessa apuna Robot Frameworkiä.

7.1 Havainnollistaminen Robot Frameworkillä

Kuvion 18 kaikki vaiheet voidaan automatisoida, mutta ensimmäisen vaiheen automatisointi vaatii ohjelmoitavan jatkojohdon laitteistodebuggerin käynnistyksen takia. Koska käytössä ei ollut ohjelmoitavaa jatkojohtoa, hypätään kuvion 18 ensimmäisen vaiheen yli.

Automatisoinnin havainnollistamiseen käytetään Robot Frameworkiä. Robot Framework on geneerinen avoimen lähdekoodin testiautomaatiokehikko. Ensiksi tehtiin testikirjasto Pythonilla, joka toteuttaa kuvion 18 vuokaavion vaiheet. Robot Framework kutsuu tämän Python-kirjaston metodeita. Alla olevissa kuvioissa (Kuviot 19 ja 20) on esitetty Python-kirjaston koodi.

T32RemoteApi.py M | t32_intercom.py | T32CodeCoverage.robot | T32Resources.resource M

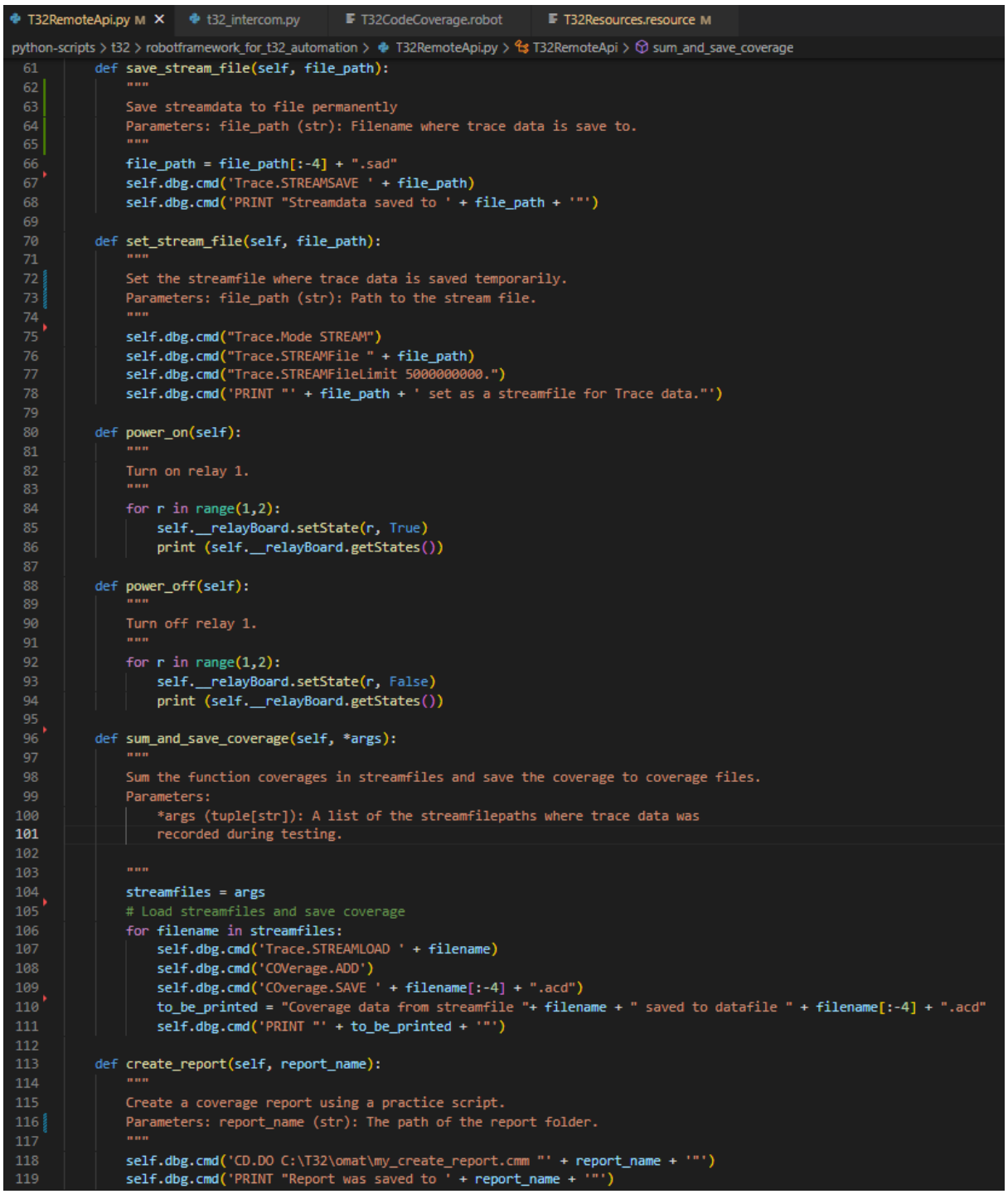
python-scripts > t32 > robotframework_for_t32_automation > T32RemoteApi.py > T32RemoteApi > __init__

```python
import lauterbach.trace32.rcl as t32
import dae_RelayBoard


class T32RemoteApi:
    """
    Class for using t32 python remote API. This class is meant to be used by robotframework testautomation.
    """

    def __init__(self):
        """
        Initialize t32 tcp-connection and dae realayboard usb-connection.
        Dae relayboard is used to power on and power off the target controller.
        """
        self.dbg = t32.connect(node='localhost', port=20000, protocol="TCP", timeout=60.0)

        self.__relayBoard = dae_RelayBoard.DAE_RelayBoard(dae_RelayBoard.DAE_RELAYBOARD_TYPE_8)

        self.__relayBoard.initialise()

        self.__relayBoard.setAllStatesOff()

    def __del__(self):
        """
        Disconnect dae relayboard connection.
        """
        self.__relayBoard.disconnect()

    def reset(self):
        """
        Run a start up script that resets Trace and coverage settings.
        """
        self.dbg.cmd('PRINT "Run start up script C:\T32\omat\start_up.cmm"')
        self.dbg.cmd("CD.DO C:\T32\omat\start_up.cmm")
        self.dbg.cmd('PRINT "Reset complete"')

    def reset_coverage(self):
        """
        Reset coverage and select function as metric.
        """
        self.dbg.cmd('COVerage.RESet')
        self.dbg.cmd('COVerage.METHOD INCremental')
        self.dbg.cmd('COVerage.Option.SourceMetric Function')
        self.dbg.cmd('COVerage.Init')

    def start(self):
        """
        Start recoding trace data.
        """
        self.dbg.cmd('PRINT "GO"')
        self.dbg.cmd("Go")


    def stop(self):
        """
        Stop recording trace data.
        """
        self.dbg.cmd('PRINT "BREAK"')
        self.dbg.cmd("Break")
```

Kuvio 19. Python-kirjasto koodikattavuusmittausta varten.

Kuviosta 19 nähdään, että luokan alustusmetodi luo yhteyden Trace32-ohjelmaan ja USB-ohjattavaan relelautaan, jolla kohdelaite voidaan ohjelmallisesti käynnistää tai sammuttaa. Luokka ottaa käyttöön Trace32-kirjaston ja USB-ohjaukseen tarvittavan kirjaston. Kuvassa näkyy lisäksi kaksi erilaista metodia Trace- ja koodikattavuusdatan nollaamiseen sekä

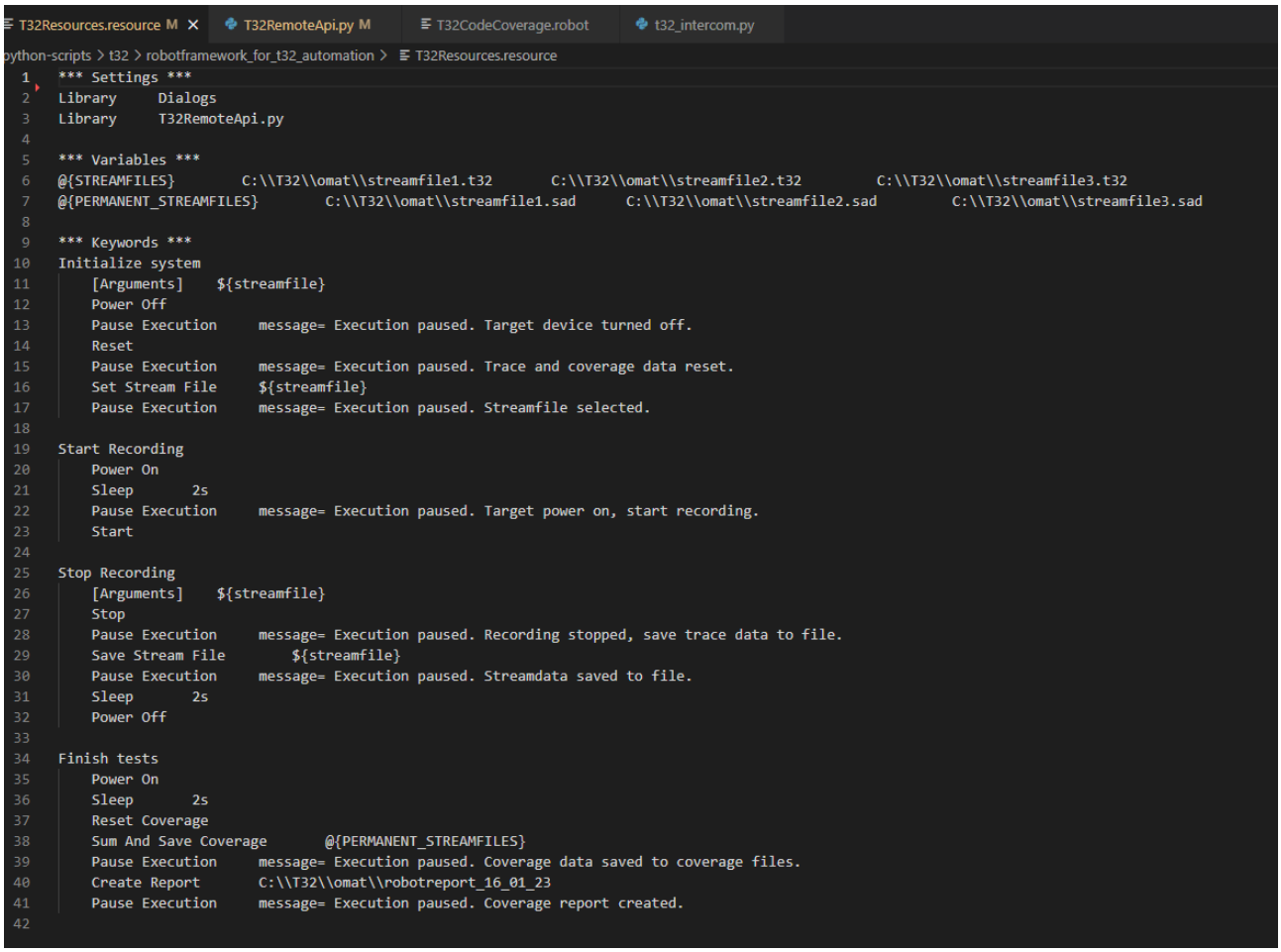
metodit, jotka käynnistävät ja lopettavat koodikattavuusmittauksen. Luokan metodeita kutsutaan sen ulkopuolelta Robot Frameworkin koodista. Koska kaikki luokan koodi ei mahtunut yhteen kuvakaappaukseen, tarvitaan vielä toinen kuva esittämään loput luokan metodeista.



Kuvio 20. Python-kirjasto jatkuu.

Kuviossa 20 näkyvät metodit ”power_on” ja ”power_off” käynnistävät tai sammuttavat kohdelaitteen. Lisäksi luokassa on metodi väliaikaistiedoston luomiseen jäljitysdataa varten ja metodi väliaikaistiedoston pysyvään tallennukseen isäntäkoneelle. Metodi ”sum_and_save_coverage” lataa pysyvästi tallennetut tiedostot, summaa niiden koodikattavuudet yhteen ja tallentaa koodikattavuustiedot pysyvästi omaan tiedostoonsa. Kuviossa näkyy myös metodi ”create_report”, joka kutsuu koodikattavuusraportin luovaa PRACTICE-skriptiä.

Oma Python-kirjasto otetaan käyttöön Robot Frameworkin resurssitiedostossa. Alla on kuvio (Kuvio 21) kyseisestä resurssitiedostosta.



Kuvio 21. Robot Frameworkin resurssitiedosto.

Resurssitiedosto ottaa asetuksissa käyttöön oman Python-kirjaston sekä Dialogi-kirjaston suorituksen pysäyttämistä varten. Resurssitiedosto määrittelee avainsanat, joita voidaan kutsua ylemmältä tasolta, jonne taas on määritelty testitapaukset. Esimerkiksi avainsana

”Start Recording” kutsuu ensiksi kuviossa 20 nähtyä ”power_on”-metodia Python-kirjastosta. Sen jälkeen odotetaan 2 sekuntia, että kohdelaite ehtii käynnistymään. Sitten pysäytetään suoritus ja näytetään viesti käyttäjälle, jotta tilanteesta voidaan ottaa kuvakaappaus. Kun käyttäjä sulkee näytetyn viestin, aloitetaan koodikattavuuden mittaus Python-kirjaston metodilla ”start”.

Itse testitapaukset on määritelty vielä omassa tiedostossaan, josta on alla kuva (Kuvio 22).

T32CodeCoverage.robot | T32Resources.resource M | T32RemoteApi.py M | t32_intercom.py

python-scripts > t32 > robotframework_for_t32_automation > T32CodeCoverage.robot

```
*** Settings ***
Suite Teardown    Finish Tests
Resource          T32Resources.resource


*** Test Cases ***
Test Case 1
      Log To Console      ->
      Log To Console       Reset target device and select streamfile for trace data
      Initialize system       ${STREAMFILES}[0]
      Start Recording
      Sleep       2s
      Stop Recording          ${STREAMFILES}[0]
      Pause Execution     message= Recording of trace data for test case 1 done

Test Case 2
      Log To Console      ->
      Log To Console       Reset target device and select streamfile for trace data
      Initialize system       ${STREAMFILES}[1]
      Start Recording
      Sleep       2s
      Stop Recording          ${STREAMFILES}[1]
      Pause Execution     message= Recording of trace data for test case 2 done

Test Case 3
      Log To Console      ->
      Log To Console       Reset target device and select streamfile for trace data
      Initialize system       ${STREAMFILES}[2]
      Start Recording
      Sleep       2s
      Stop Recording          ${STREAMFILES}[2]
      Pause Execution     message= Recording of trace data for test case 3 done
```

Kuvio 22. Robot Frameworkin testitapaukset.

Kuviosta 22 nähdään, että resurssitiedoston muuttujat ja avainsanat otetaan asetuksissa käyttöön resurssina. Kaikkien testitapausten suorituksen jälkeen ajetaan ”Finish Tests”-avainsana, joka on määritelty resurssitiedostossa. Tämä avainsana suorittaa kuvion 18

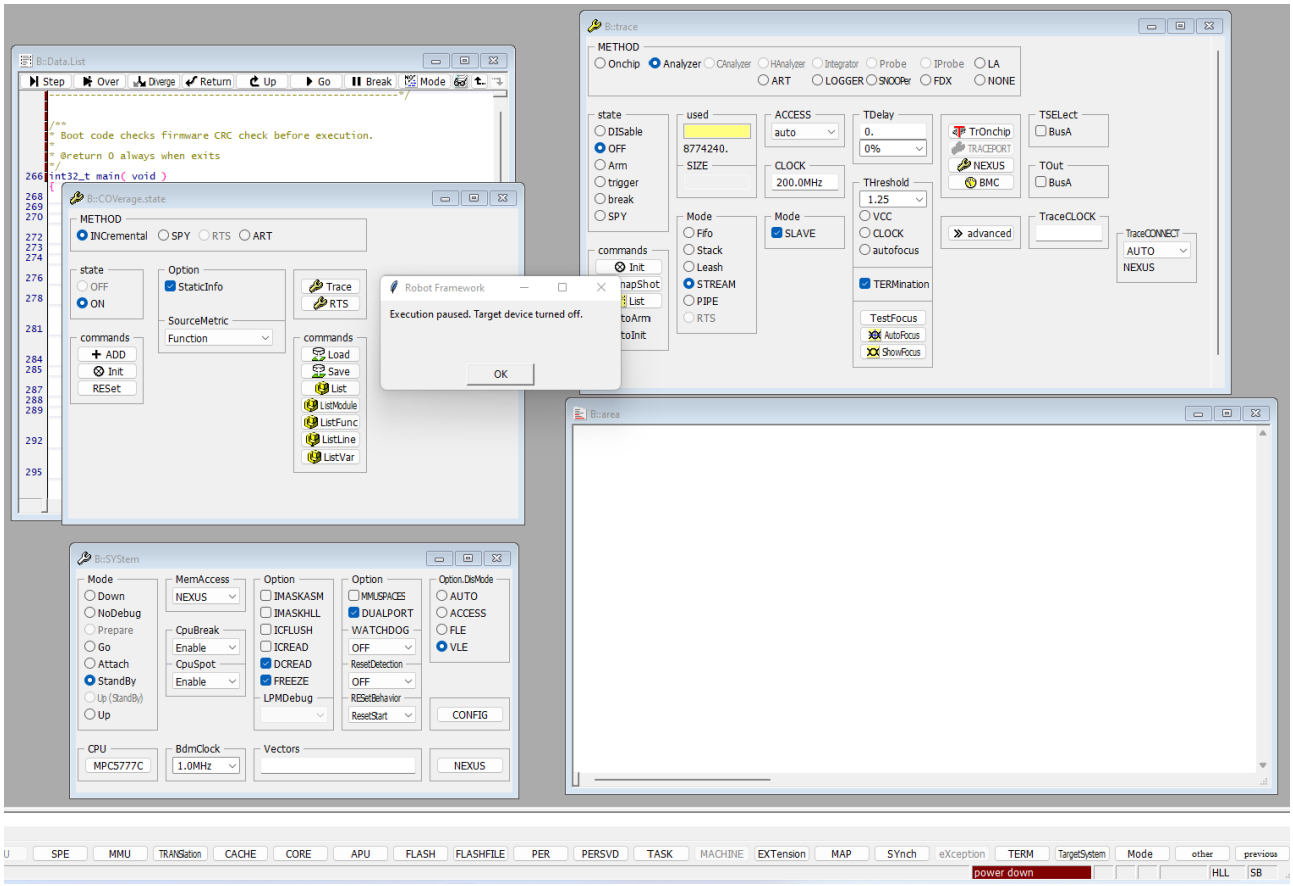
vuokaavion vaiheet tallennettujen tiedostojen latauksesta koodikattavuusraportin luomiseen.

Testitapauksissa on määritelty kolme testitapausta, jotka toistavat kuviossa 18 nähtyä vaihesilmukkaa vaiheesta ”kohdelaitteen sammutus” vaiheeseen ”pysyvän tiedoston tallennus”. Testitapaukset itsessään eivät testaa mitään, vaan niissä vain ajetaan laiteohjelmistoa normaalisti ja mitataan ajon perusteella koodikattavuus. Tämä ei kuitenkaan haittaa, koska tarkoitus on havainnollistaa testiautomaatiota, ei itse testejä.

Avainsana ”Initialize System” suorittaa kuvion 18 vuokaavion vaiheet kohdelaitteen sammutuksesta uuden väliaikaistiedoston luomiseen. Avainsana ”Start Recording” suorittaa vuokaavion vaiheista kohdelaitteen käynnistyksen ja koodikattavuusmittauksen aloittamisen. Itse testit ajettaisiin kohdelaitteella avainsanojen ”Start recording” ja ”Stop Recording” välissä. Nyt vain odotetaan 2 sekuntia ja kerätään dataa väliaikaistiedostoon. ”Stop Recording” -avainsana suorittaa koodikattavuusmittauksen lopetuksen ja väliaikaistiedoston pysyvän tallennuksen isäntäkoneelle.

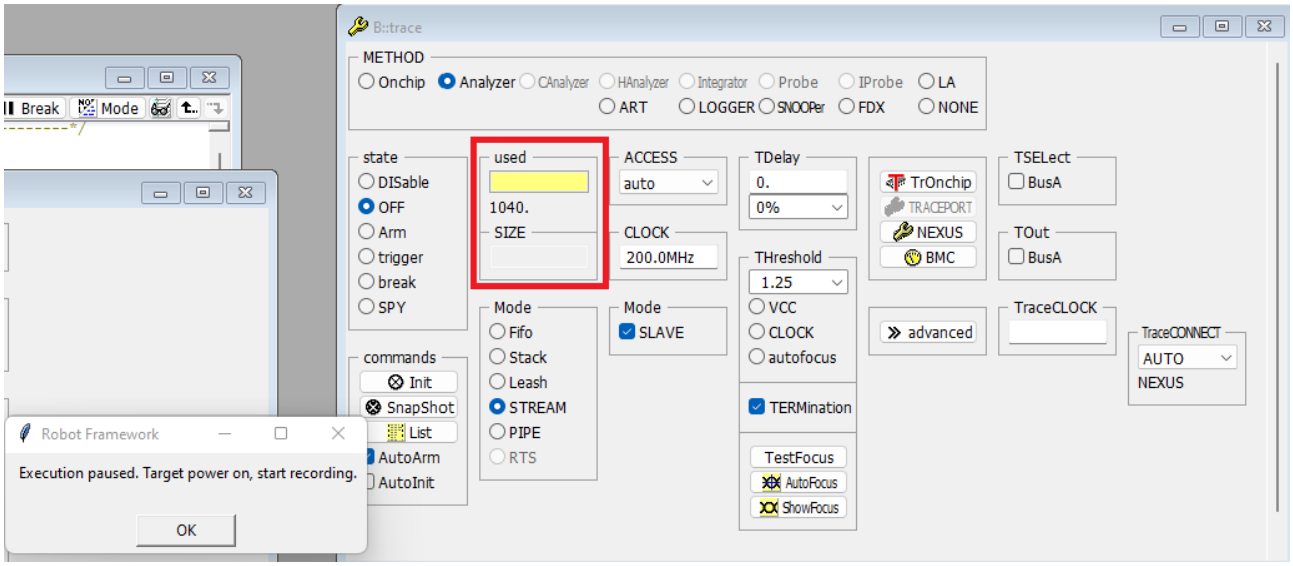
7.2 Mittauksen eteneminen

Havainnollistamisen lähtötilanteessa eli kuvion 18 vuokaavion vaiheessa ”kohdelaitteen sammutus” Trace32-ohjelmassa näyttää tilanne seuraavalla sivulla olevan kuvion kaltaiselta (Kuvio 23).



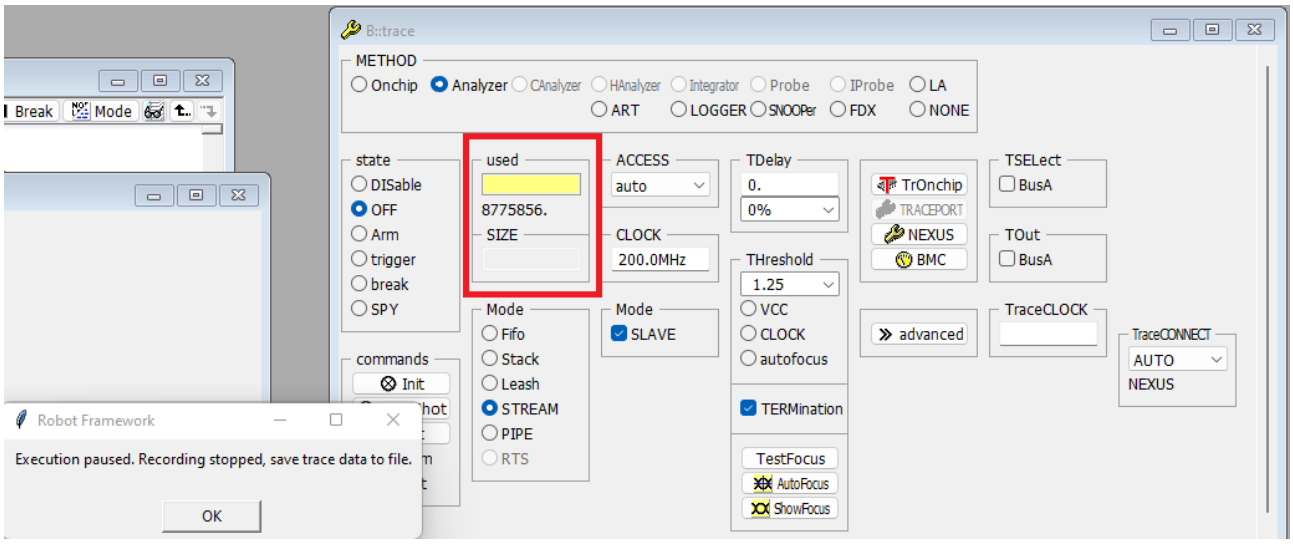
Kuvio 23. Trace32, kohdelaite on sammutettu.

Kuviossa 23 kohdelaite on sammutettu ja Robot Framework pysäyttää suorituksen, kunnes käyttäjä painaa ok-nappia. Tämän jälkeen suoritetaan Trace- ja koodikattavuustietokantojen nollaus sekä luodaan väliaikaistiedosto jäljitysdataa varten. Kun vielä käynnistetään kohdelaite, ollaan vuokaavion vaiheessa ”kohdelaitteen käynnistys”. Trace-ikkunassa tilanne näyttää silloin seuraavalla sivulla olevan kuvion kaltaiselta (Kuvio 24).



Kuvio 24. Trace32, kohdelaite on valmis koodikattavuusmittaukseen.

Kohdelaitteen käynnistyksen jälkeen laiteohjelmiston suoritus on pysähtynyt pysäytyspisteeseen laiteohjelmiston pääfunktiossa. Trace-ikkunassa nähdään, että jäljitysdataa on jo hieman kertynyt ennen pysähdyskohtaa. Tämän jälkeen aloitetaan keräämään jäljitysdataa väliaikaistiedostoon, ja samalla ajetaan testejä kohdelaitteella. Kun testit on saatu ajettua, pysäytetään jäljitysdatan mittaus eli ollaan vuokaavion vaiheessa ”koodikattavuusmittauksen lopetus”. Trace-ikkunassa tilanne näyttää silloin alla olevan kuvion kaltaiselta (Kuvio 25).

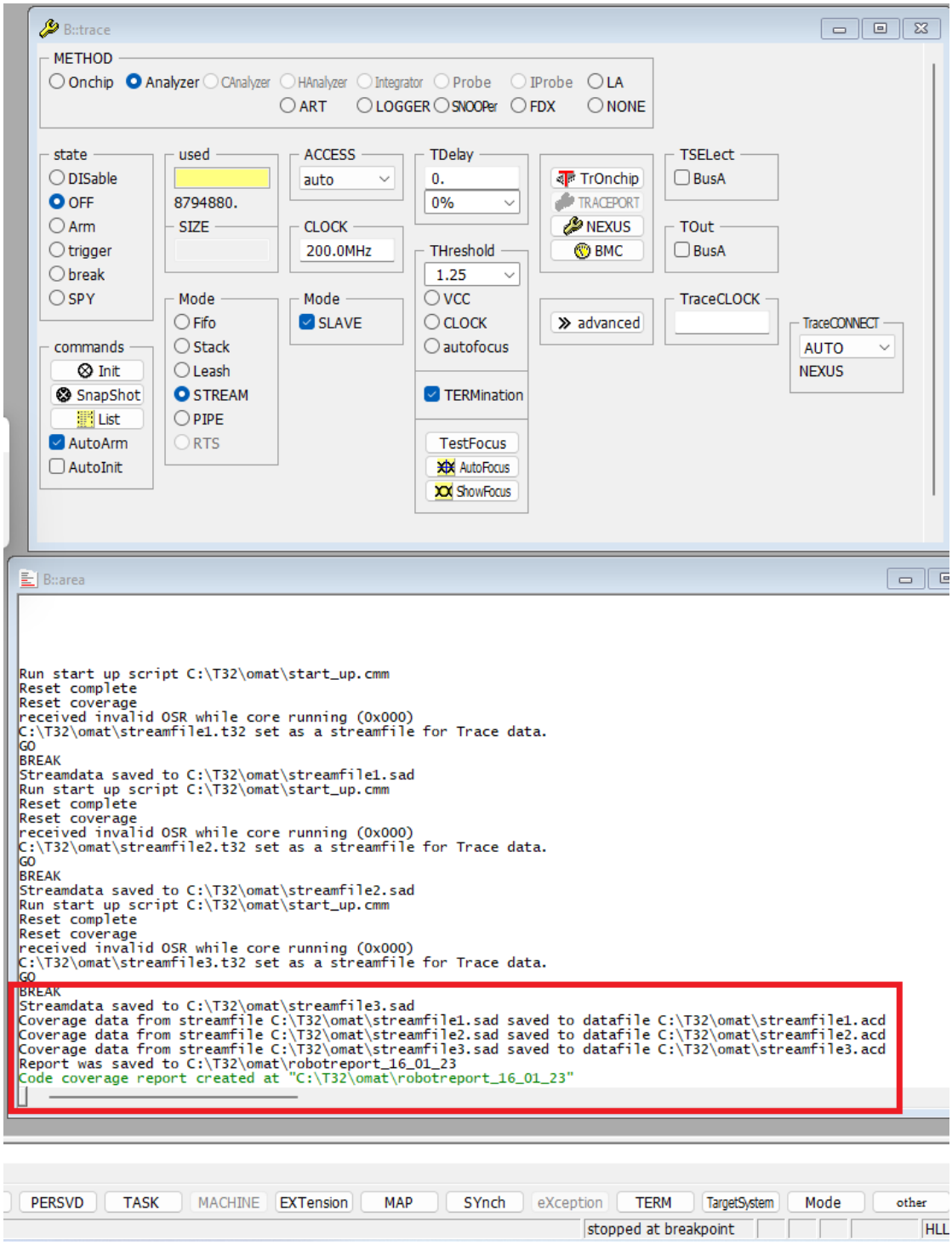


Kuvio 25. Trace32, jäljitysdata on nauhoitettu.

”Trace”-ikkunasta nähdään, että jäljitysdataa on kerääntynyt jo jonkin verran. Tämä data tallennetaan pysyvästi tiedostoon kuviossa 20 nähdyllä ”save_stream_file”-metodilla. Kun

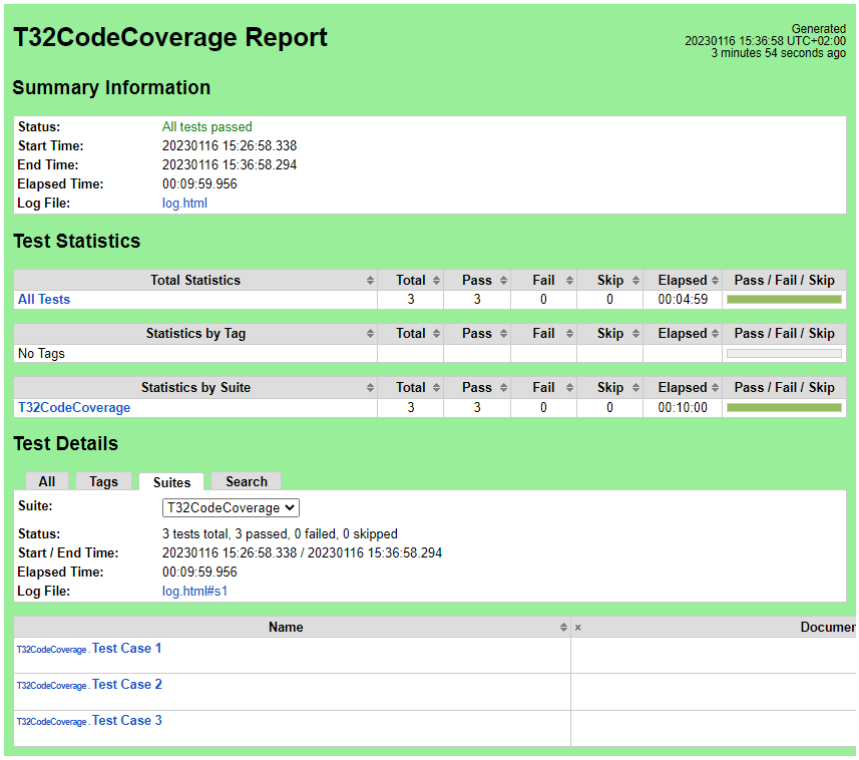
data on tallennettu tiedostoon, siirrytään seuraavaan testitapaukseen eli takaisin kuvion 18 vuokaavion vaiheeseen ”kohdelaitteen sammutus”.

Kun kaikki kolme testitapausta on suoritettu, ladataan tallennetut tiedostot ja prosessoidaan niiden koodikattavuus. Kaikkien tiedostojen koodikattavuus summataan yhteen, ja koodikattavuustietokannat tallennetaan omiin tiedostoihinsa. Viimeinen vaihe on luoda koodikattavuusraportti. Tähän käytetään valmista PRACTICE-skriptiä, jolle annetaan vain parametrit siitä, millainen raportti halutaan. Kun kaikki koodikattavuusmittauksen vuokaavion vaiheet on suoritettu, näyttää tilanne Trace32-ohjelmassa seuraavalla sivulla olevan kuvan kaltaiselta (Kuvio 26).



Kuvio 26. Trace32, koodikattavuusraportti valmis.

Kuvion 26 ”area”-ikkunan tulostuksista nähdään, että kolme eri testitapausta on suoritettu ja kaikkien tapausten koodikattavuus on tallennettu tiedostoihin. Lopuksi koodikattavuusraportti on tallennettu omaan kansioonsa. Lisäksi koodikattavuusmittauksesta Robot Framework on tehnyt automaattisesti oman suoritusraporttinsa, joka näkyy seuraavalla sivulla olevassa kuviossa (Kuvio 27).



T32CodeCoverage Report

Generated
20230116 15:36:58 UTC+02:00
3 minutes 54 seconds ago

Summary Information

Status:	All tests passed
Start Time:	20230116 15:26:58.338
End Time:	20230116 15:36:58.294
Elapsed Time:	00:09:59.956
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	3	0	0	00:04:59	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
T32CodeCoverage	3	3	0	0	00:10:00	

Test Details

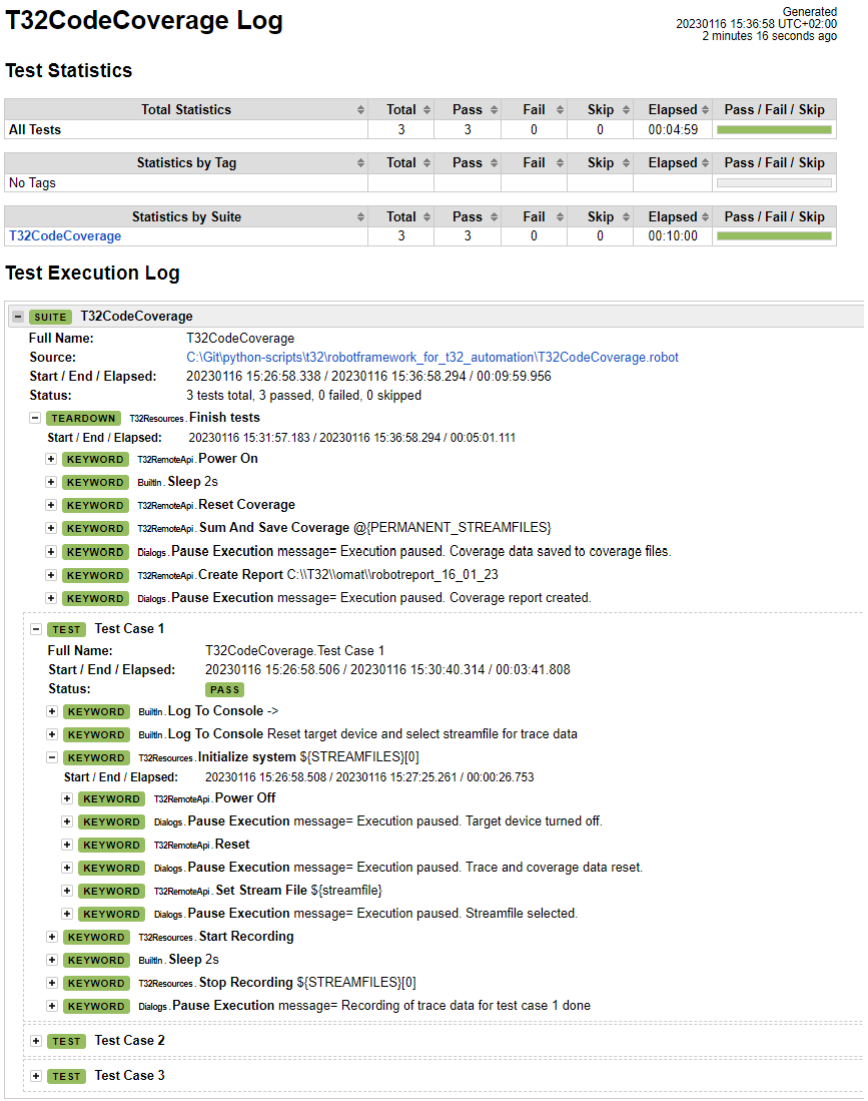
All | Tags | Suites | Search

Suite:	T32CodeCoverage
Status:	3 tests total, 3 passed, 0 failed, 0 skipped
Start / End Time:	20230116 15:26:58.338 / 20230116 15:36:58.294
Elapsed Time:	00:09:59.956
Log File:	log.html#s1

Name	Documer
T32CodeCoverage . Test Case 1	
T32CodeCoverage . Test Case 2	
T32CodeCoverage . Test Case 3	

Kuvio 27. Robot Frameworkin raportti.

Kuviosta 27 nähdään lähinnä, että kaikki testit on suoritettu, sekä niiden aloitus- ja lopetusajat. Tarkemmin testien suoritusta voi katsoa Robot Frameworkin luomasta lokitiedostosta, joka näkyy seuraavalla sivulla (Kuvio 28).



T32CodeCoverage Log

Generated
20230116 15:36:58 UTC+02:00
2 minutes 16 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	3	0	0	00:04:59	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
T32CodeCoverage	3	3	0	0	00:10:00	

Test Execution Log

SUITE T32CodeCoverage
Full Name: T32CodeCoverage
Source: C:\Git\python-scripts\t32\robotframework_for_t32_automation\T32CodeCoverage.robot
Start / End / Elapsed: 20230116 15:26:58.338 / 20230116 15:36:58.294 / 00:09:59.956
Status: 3 tests total, 3 passed, 0 failed, 0 skipped
TEARDOWN T32Resources. Finish tests
Start / End / Elapsed: 20230116 15:31:57.183 / 20230116 15:36:58.294 / 00:05:01.111
KEYWORD T32RemoteApi. Power On
KEYWORD BuiltIn. Sleep 2s
KEYWORD T32RemoteApi. Reset Coverage
KEYWORD T32RemoteApi. Sum And Save Coverage @{PERMANENT_STREAMFILES}
KEYWORD Dialogs. Pause Execution message= Execution paused. Coverage data saved to coverage files.
KEYWORD T32RemoteApi. Create Report C:\\T32\\omat\\robotreport_16_01_23
KEYWORD Dialogs. Pause Execution message= Execution paused. Coverage report created.
TEST Test Case 1
Full Name: T32CodeCoverage.Test Case 1
Start / End / Elapsed: 20230116 15:26:58.506 / 20230116 15:30:40.314 / 00:03:41.808
Status: PASS
KEYWORD BuiltIn. Log To Console ->
KEYWORD BuiltIn. Log To Console Reset target device and select streamfile for trace data
KEYWORD T32Resources. Initialize system ${STREAMFILES}[0]
Start / End / Elapsed: 20230116 15:26:58.508 / 20230116 15:27:25.261 / 00:00:26.753
KEYWORD T32RemoteApi. Power Off
KEYWORD Dialogs. Pause Execution message= Execution paused. Target device turned off.
KEYWORD T32RemoteApi. Reset
KEYWORD Dialogs. Pause Execution message= Execution paused. Trace and coverage data reset.
KEYWORD T32RemoteApi. Set Stream File ${streamfile}
KEYWORD Dialogs. Pause Execution message= Execution paused. Streamfile selected.
KEYWORD T32Resources. Start Recording
KEYWORD BuiltIn. Sleep 2s
KEYWORD T32Resources. Stop Recording ${STREAMFILES}[0]
KEYWORD Dialogs. Pause Execution message= Recording of trace data for test case 1 done
TEST Test Case 2
TEST Test Case 3

Kuvio 28. Robot Frameworkin lokitiedosto.

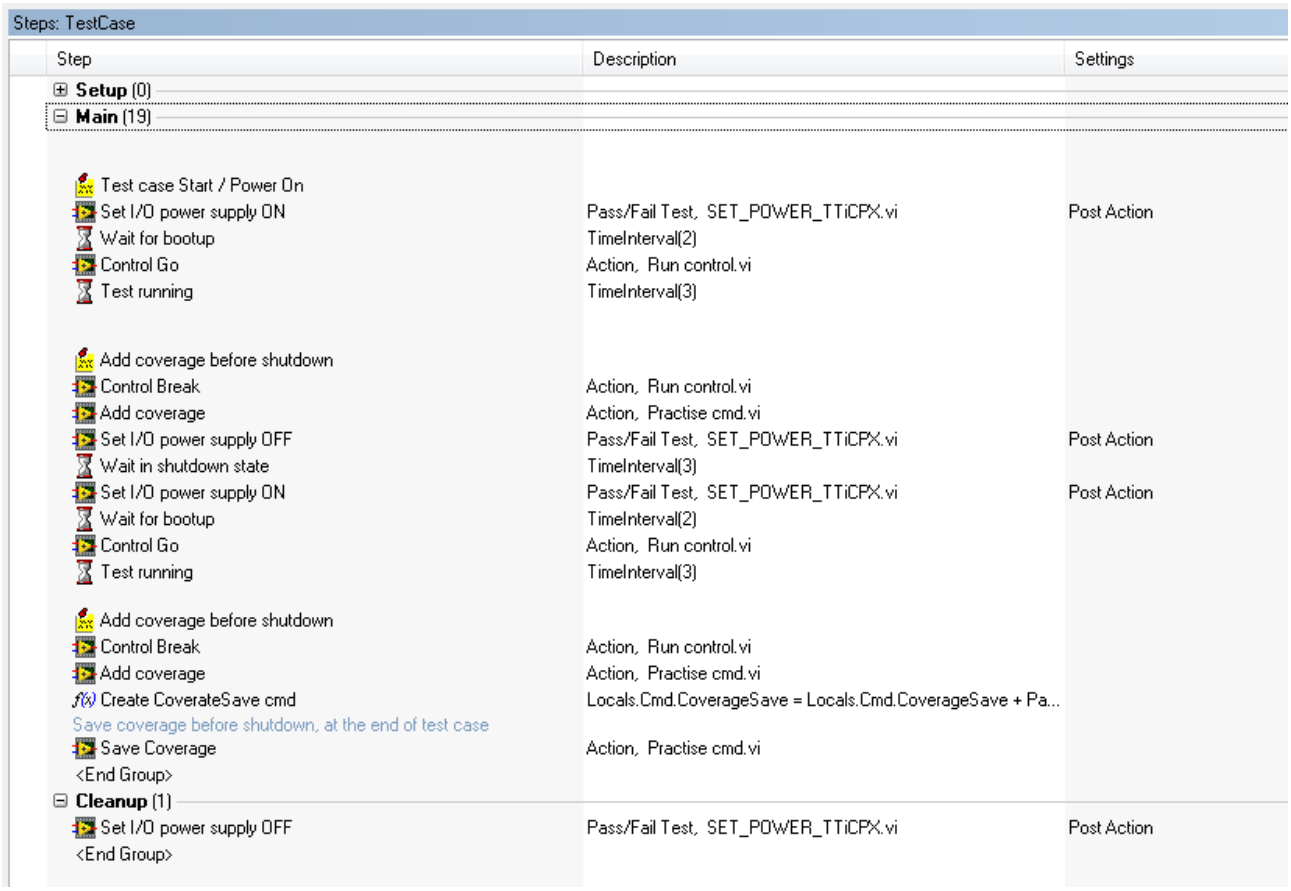
Kuvion 28 lokitiedostosta nähdään avainsanojen sisälle ja myös muuttujien arvoja suoritushetkellä sekä tarkempia aikaleimoja. Tärkein tulos on kuitenkin koodikattavuusraportti, josta puhutaan tarkemmin työn tuloksista kertovassa luvussa.

7.3 Havainnollistaminen TestStand-testinhallintaohjelmistolla

Koska laiteohjelmistojen testausta halutaan tehdä myös TestStand-ohjelmistolla, oli tärkeä havainnollistaa koodikattavuuden mittausta myös tässä ympäristössä. LabVIEW-ympäristö alustetaan purkamalla Trace32-kirjastot ohjelman asennuskansion alta LabVIEW-ohjelman kirjastokansioon (Lauterbach, 2022e, s. 5). Näin saadaan Trace32-komennot käyttöön

omina LabVIEW-moduuleinaan. LabVIEW:llä voidaan komentaa Trace32-instansseja samaan tapaan kuin Pythonilla.

Koodikattavuusmittaus eteni pääpiirteittäin kuvion 18 vuokaavion mukaisesti, vaikkakin jotkin vaiheet saattoivat olla yhdistetty yhteen isompaan PRACTICE-skriptiin. Kohdelaitteen virrat voidaan kytkeä päälle ja pois TestStand-ohjelmiston kautta. Alla olevasta kuviosta (Kuvio 29) nähdään koodikattavuusmittauksen eteneminen askel askeleelta.



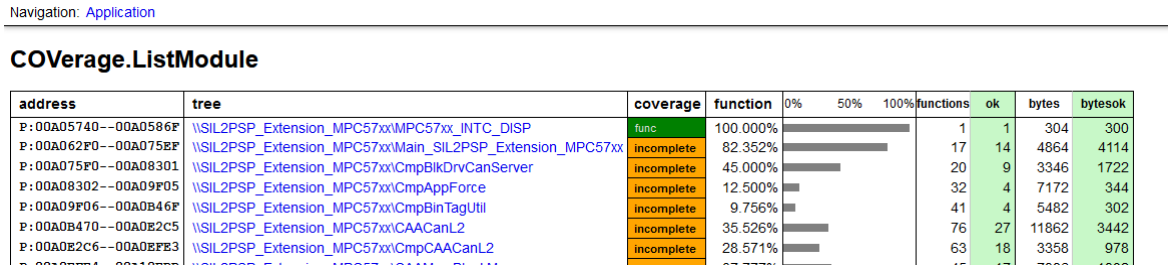
Kuvio 29. Koodikattavuusmittaus TestStand-ohjelmistolla.

Kuviosta 29 nähdään, että ensin asetetaan Trace32-asetukset kohdilleen mittausta varten ja sitten laitetaan kohdelaitteen virrat päälle. Kohdelaitteen käynnistymistä odotetaan pari sekuntia, ja sitten aloitetaan koodikattavuuden mittaus. Tämän jälkeen ajettaisiin itse testit kohdelaitteella. Kun testitapaus on ajettu, lopetetaan koodikattavuuden mittaus ja summataan tulokset koodikattavuustietokantaan. Lopuksi sammutetaan kohdelaite ennen seuraavia testitapauksia. Kun kaikki testitapaukset on suoritettu, suoritetaan viimeiset vaiheet kuvion 18 vuokaaviosta eli tallennetaan koodikattavuustiedostot. Kuviossa 29

koodikattavuusraportin luontia ei ole vielä automatisoitu, vaan se on luotu manuaalisesti Trace32-käyttöliittymän kautta.

8 TULOKSET

Tuloksena koodikattavuusmittauksesta saadaan HTML-raportti, josta nähdään kokonaiskoodikattavuus ja tarkemmin moduuli, funktio ja rivikohtaiset koodikattavuudet. Alla on kuvio moduulitason koodikattavuudesta (Kuvio 30).



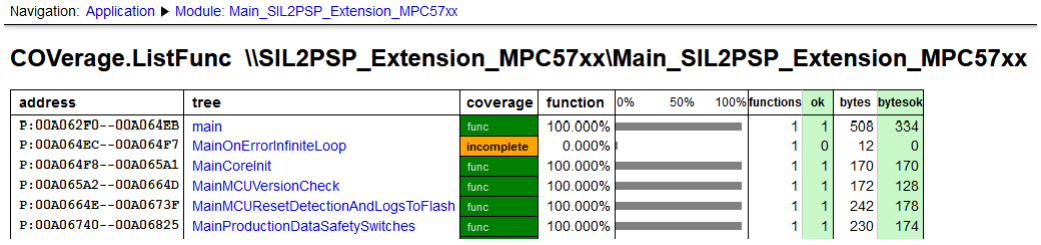
Navigation: Application

COVerage.ListModule

address	tree	coverage	function	0% 50% 100%	functions	ok	bytes	bytesok
P:00A05740--00A0586F	\\SIL2PSP_Extension_MPC57xx\MPC57xx_INTC_DISP	func	100.000%		1	1	304	300
P:00A062F0--00A075EF	\\SIL2PSP_Extension_MPC57xx\Main_SIL2PSP_Extension_MPC57xx	incomplete	82.352%		17	14	4864	4114
P:00A075F0--00A08301	\\SIL2PSP_Extension_MPC57xx\CmpBlkDrvCanServer	incomplete	45.000%		20	9	3346	1722
P:00A08302--00A09F05	\\SIL2PSP_Extension_MPC57xx\CmpAppForce	incomplete	12.500%		32	4	7172	344
P:00A09F06--00A0B46F	\\SIL2PSP_Extension_MPC57xx\CmpBinTagUtil	incomplete	9.756%		41	4	5482	302
P:00A0B470--00A0E2C5	\\SIL2PSP_Extension_MPC57xx\CAACanL2	incomplete	35.526%		76	27	11862	3442
P:00A0E2C6--00A0EFE3	\\SIL2PSP_Extension_MPC57xx\CmpCAACanL2	incomplete	28.571%		63	18	3358	978

Kuvio 30. Koodikattavuusraportti, moduulitaso.

Kuviosta 30 voidaan nähdä moduulien nimet/polut ja osoitealueet. Lisäksi nähdään käytettävä koodikattavuusmittari eli funktiokattavuus sekä funktiokattavuuden arvo prosentteina. Lisätietoina nähdään myös funktioiden kokonaislukumäärä ja testauksessa kutsuttujen funktioiden lukumäärä sekä vastaavat arvot tavuina. Moduulin nimeä klikkaamalla päästään katsomaan moduuliin kuuluvien funktioiden kattavuutta tarkemmin. Tästä on kuvio alla (Kuvio 31).



Navigation: Application ▶ Module: Main_SIL2PSP_Extension_MPC57xx

COVerage.ListFunc \\SIL2PSP_Extension_MPC57xx\Main_SIL2PSP_Extension_MPC57xx

address	tree	coverage	function	0% 50% 100%	functions	ok	bytes	bytesok
P:00A062F0--00A064EB	main	func	100.000%		1	1	508	334
P:00A064EC--00A064F7	MainOnErrorInfiniteLoop	incomplete	0.000%		1	0	12	0
P:00A064F8--00A065A1	MainCoreInit	func	100.000%		1	1	170	170
P:00A065A2--00A0664D	MainMCUVersionCheck	func	100.000%		1	1	172	128
P:00A0664E--00A0673F	MainMCUResetDetectionAndLogsToFlash	func	100.000%		1	1	242	178
P:00A06740--00A06825	MainProductionDataSafetySwitches	func	100.000%		1	1	230	174

Kuvio 31. Koodikattavuusraportti, funktiotaso.

Kuviossa 31 moduuliksi on valittu ”Main_SIL2PSP_Extension_MPC57xx”. Kuvasta nähdään, että jos funktiota on kutsuttu testauksessa vähintään kerran, on funktiokattavuus tietenkin 100 %, ja jos funktiota ei ole kutsuttu testauksessa, jää kattavuus nollaan. Tästä kuvasta esimerkiksi ilmenee, että ”MainOnErrorInfiniteLoop”-funktiossa ei ole käyty testauksen aikana. Koodikattavuusraportin rivitason näkymästä ei otettu kuvakaappauksia, koska siinä näkyy Epecin laiteohjelmiston lähdekoodia, jota ei haluta julkisesti jakaa.

9 POHDINTA

Työn päätavoitteessa eli koodikattavuusmittauksen automatisoinnissa laitteistodebuggerin avulla onnistuttiin hyvin. Automatisoinnin eri vaiheet on hahmoteltu vuokaavioon, automatisointiin tarvittavat Trace32-komennot on selvitetty ja koodikattavuusmittauksia on havainnollistettu onnistuneesti sekä Python- että LabVIEW-etärajapintojen avulla. Täten koodikattavuutta pystytään mittaamaan omalla pöydällä sekä automaattisella testausjärjestelmällä.

Lopputuloksena saatiin koodikattavuusraportti, josta nähdään mm. testauksen kokonaiskoodikattavuus kutsu- tai funktiokattavuutta mitattaessa. Täten täytetään ISO 26262 -standardin suositus koodikattavuuden mittauksesta integraatiotestauksessa. Kun tämän työn menetelmiä ja koodikattavuuden mittausta tulevaisuudessa käytetään testauksessa, voidaan myös kuollutta tai epäaktiivista koodia löytää.

Kaikkia erilaisia testauksen poikkeustilanteita, kuten virtakatkoja tai muita virhetilanteita, ei ole kuitenkaan tässä työssä selvitetty, ja niiden testaus ja tarkempi selvittäminen jääkin tulevaisuuden työksi. Seuraavissa luvuissa käydään läpi merkittävimpiä toteutuksen aikana esiin nousseita huomioitavia seikkoja.

9.1 Trace32-komentojen blokkaavuus

Työn aikana koodikattavuusmittauksia tehdessä kävi ilmi, että on tärkeää ottaa huomioon, missä tilanteissa Python- tai LabVIEW-ohjelman suoritus pysähtyy odottamaan paluuarvoa Trace32-ohjelmalta ja missä tilanteissa ohjelman suoritus jatkaa saman tien eteenpäin, eli onko komento ”blokkaava” vai ei. Lauterbachin dokumentaation ja omien testailujen mukaan Trace32-komennot ovat blokkaavia, jos ne annetaan suoraan Python- tai LabVIEW-ohjelman sisältä.

Jos taas Python- tai LabVIEW-ohjelma käskee Trace32-ohjelmaa suorittamaan PRACTICE-skriptin DO-komennolla, Trace32 palauttaa paluuarvon heti, kun PRACTICE-skripti saadaan onnistuneesti käynnistettyä (Lauterbach, 2022c, s. 15). Jos siis PRACTICE-skriptin suoritus kestää kauan tai sen kestoa ei voida ennalta tietää, olisi parempi antaa

komennot suoraan Python- tai LabVIEW-ohjelman sisältä. Esimerkiksi alla on kuvio (Kuvio 32) kahdesta funktiosta, joista toinen on blokkaava ja toinen ei.

```
96    def sum_and_save_coverage(self, *args):
97        """
98        Sum the function coverages in streamfiles and save the coverage to coverage files.
99        Parameters:
100           *args (tuple[str]): A list of the streamfilepaths where trace data was
101           recorded during testing.
102
103       """
104       streamfiles = args
105       # Load streamfiles and save coverage
106       for filename in streamfiles:
107           self.dbg.cmd('Trace.STREAMLOAD ' + filename)
108           self.dbg.cmd('COVerage.ADD')
109           self.dbg.cmd('COverage.SAVE ' + filename[:-4] + ".acd")
110           to_be_printed = "Coverage data from streamfile "+ filename + " saved to datafile " + filename[:-4] + ".acd"
111           self.dbg.cmd('PRINT "' + to_be_printed + '"')
112
113   def create_report(self, report_name):
114       """
115       Create a coverage report using a practice script.
116       Parameters: report_name (str): The path of the report folder.
117       """
118       self.dbg.cmd('CD.DO C:\T32\omat\my_create_report.cmm "' + report_name + '"')
119       self.dbg.cmd('PRINT "Report was saved to ' + report_name + '"')
```

Kuvio 32. Blokkaavat ja ei-blokkaavat Trace32-komennot.

Kuvion 32 funktio ”sum_and_save_coverage” on blokkaava, koska siinä Trace32-komentoja kutsutaan suoraan Pythonin kautta. Funktio ”create_report” taas ei ole blokkaava, koska Trace32-komentoja kutsutaan PRACTICE-skriptin sisältä, ja tämä funktio vain käynnistää kyseisen skriptin.

9.2 LabView vai Python?

Koska molemmilla työkaluilla voidaan komentaa laitteistodebuggeria hyvin samankaltaisesti, ei ole juuri väliä, kumpaa käytetään. Ainoa merkittävä ero on se, että LabVIEW käyttää ainoastaan UDP-protokollaa, kun Pythonilla on mahdollisuus myös TCP-protokollan käyttämiseen. Valinta kannattaakin mieluummin tehdä sen perusteella, mitä työkalua testauksessa käytetään.

Jos testejä ajetaan TestStand-ohjelmiston kautta, on LabVIEW luonteva valinta laitteistodebuggerin ohjaamiseen. Jos taas testejä ajetaan Robot Framework -kehyksellä, olisi Python luonteva valinta laitteistodebuggerin komentamiseen.

9.3 Poikkeukselliset testitilanteet

Tässä työssä ei juuri otettu kantaa siihen, mitä tapahtuu, jos esimerkiksi testitapauksen sisällä kohdelaitteen virrat katkaistaan ja laitetaan uudestaan päälle. Kun kohdelaitteen virrat katkeavat kesken koodikattavuusmittauksen, yhteys laitteistodebuggeriin katkeaa ja Trace-dataan päätyy käytännössä epämääräistä kohinaa. Kun kohdelaitteeseen kytketään taas virrat, laitteistodebuggeri ottaa taas automaattisesti kiinni.

Tutkittavaksi jääkin, haittaako virheellinen Trace-data koodikattavuusraporttien luomista. Yksi vaihtoehto on myös lopettaa koodikattavuuden mittaus ennen virtakatkoa. Tämä ei kuitenkaan ratkaise tilannetta, jossa edes testaaja ei etukäteen tiedä, voiko testi aiheuttaa laitteessa esimerkiksi sisäisen resetoinnin.

LÄHTEET

Ball, S. R. (1998). *Debugging Embedded Microprocessor Systems*. Newnes.

Chopra, R. (2018). *Software Quality Assurance: A Self-Teaching Introduction*. Mercury Learning & Information.

Craig, R. D. & Jaskiel, S. P. (2002). *Systematic Software Testing*. Artech House, Inc.

Dice, P. (2018). *Quick Boot: A Guide for Embedded Firmware Developers* (2. p.). De|G Press.

Epec. (2022a). *Company*. https://epec.fi/company/

Epec. (2022c). EPEC SC52 SAFETY CONTROL UNIT. https://epec.fi/products/safety-products-sc52/

Epec. (2022b). *Industries we serve*. https://epec.fi/industries-we-serve/

Ganssle, J. (2008). *Embedded Systems: World Class Designs*. Newnes.

Ganssle, J. (2004). *The Firmware Handbook*. Newnes.

Hayhurst, K.J. & Veerhusen D. S. & Chilenski J. J. & Rierson L. K. (2001). *Practical Tutorial on Modified Condition/Decision Coverage*. National Aeronautics and Space Administration (NASA). https://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf

Homès, B. (2012). *Fundamentals of Software Testing*. John Wiley & Sons, Incorporated.

Ibrahim, D. (2015). *PIC32 Microcontrollers and the Digilent Chipkit: Introductory to Advanced Projects*. Elsevier Science & Technology.

International Organization for Standardization (ISO). (2018a). *Road Vehicles – Functional safety, Part 1: Vocabulary* (ISO Standard No. 26262-1:2018).

International Organization for Standardization (ISO). (2018b). *Road Vehicles – Functional safety, Part 3: Concept phase* (ISO Standard No. 26262-3:2018).

International Organization for Standardization (ISO). (2018c). *Road Vehicles – Functional safety, Part 5: Product development at the hardware level* (ISO Standard No. 26262-5:2018).

International Organization for Standardization (ISO). (2018d). *Road Vehicles – Functional safety, Part 6: Product development at the software level* (ISO Standard No. 26262-6:2018).

Lauterbach. (2022a). *API for Remote Control and JTAG Access in C*. https://www2.lauterbach.com/pdf/api_remote_c.pdf

Lauterbach. (2022b). *Application Note for Trace-Based Code Coverage*. https://www2.lauterbach.com/pdf/app_code_coverage.pdf

Lauterbach. (2022c). *Controlling TRACE32 via Python 3*. https://www2.lauterbach.com/pdf/app_python.pdf

Lauterbach. (2022d). *Home*. https://www.lauterbach.com/frames.html?home.html

Lauterbach. (2022e). *Integration with LabView*. https://www2.lauterbach.com/pdf/int_labview.pdf

Lauterbach. (2022f). *PowerTrace Serial User´s Guide*. https://www2.lauterbach.com/pdf/serialtrace_user.pdf

Lauterbach. (2022g). *PowerView Command Reference*. https://www2.lauterbach.com/pdf/ide_ref.pdf

Lauterbach. (2022h). *PRACTICE Script Language User's Guide*. https://www2.lauterbach.com/pdf/practice_user.pdf

Lauterbach. (2022i). *Trace Tutorial*. https://www2.lauterbach.com/pdf/trace_tutorial.pdf

Lauterbach. (2022j). *TRACE32 Installation Guide*. https://www2.lauterbach.com/pdf/installation.pdf

Mackay, A. (i.a.). *What is meant by Structural Code Coverage?*. QA Systems. https://www.qa-systems.com/blog/what-is-meant-by-structural-code-coverage/

Meany, T. (i.a.). *Functional Safety and Industry 4.0*. Analog Devices. https://www.analog.com/en/technical-articles/functional-safety-and-industry-4.0.html

Robot Framework Foundation. (2023). *Robot Framework User Guide*. https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Shwetha, M. S. (2022). *The ISO 26262 dilemma: Function coverage and call coverage*. Liverpool Data Research Associates (LDRA). https://www.ldra.com/ldra-blog/the-iso-26262-dilemma-function-coverage-and-call-coverage%EF%BF%BC/

Suomen Standardoimisliitto (SFS). (2011).*Sähköisten/elektronisten/ohjelmoitavien elektronisten turvallisuuteen liittyvien järjestelmientoiminnallinen turvallisuus. Osa 0: Toiminnallinen turvallisuus ja IEC 61508* (IEC/TR 61508-0:fi).