Vinkkejä robottiharjoitustyöhön

ASE-6030 Automaation reaaliaikajärjestelmät

Versiohistoria:

3.0 (2016)	P. Kannisto	Uusi vinkkidokumentin versio Windows 10 IoT Core -alustalle.

1. Toteuttaminen

1.1. Tarvittavat ohjelmistot ja laitteet

Ohjelmointiin tarvittavat ohjelmistot:

- Windows 10
  - Jos tämä on ongelma, ota yhteys kurssihenkilökuntaan!
- Visual Studio 2015
  - Opiskelijat saavat Visual Studion Microsoftin DreamSparkista.

1.2. Projektipohja

Lataa projektipohja kurssin sivuilta ja avaa se Visual Studiossa. Täydennä tiedostoa "AppLogic.cs" ja lisää oman toteutuksesi vaatimat luokat projektiin.

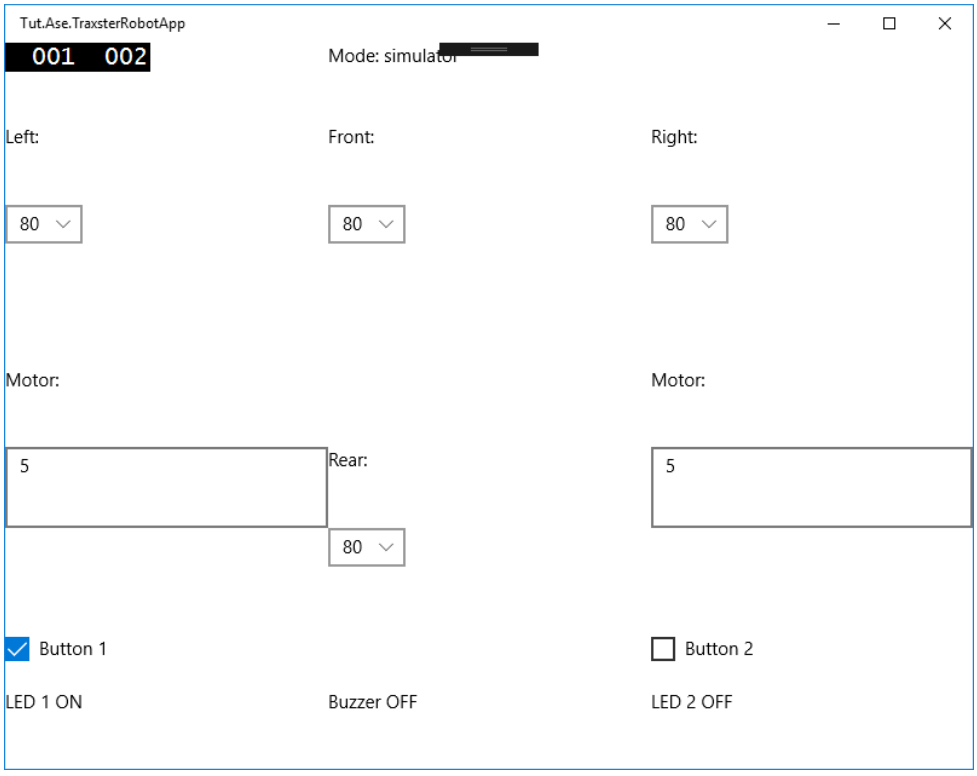
1.3. Laitteiden tunnisteet robotilla

Laitteiden pin-tunnisteet (tvast.) löytyvät tiedostosta *DeviceConstants.cs*.

2. Simulaattori

Simulaattori on testialusta sovellukselle. Se on toteutettu robottisovelluspohjan käyttöliittymänä. Sovelluspohja osaa tunnistaa, milloin sovellusta ajetaan työpöytäkoneessa ja milloin laitteella. Kun ollaan laitteella, simulaattorikäyttöliittymä ei ole aktiivinen, vaan käytetään fyysisiä laitteita.

Simulaattori on pelkistetty, mutta se avulla voidaan helposti todeta, miten sovellus reagoi etäisyysanturien lukemien muutoksiin sekä napinpainalluksiin. Virheiden etsintä ja korjaaminen itse laitteella on huomattavasti simulaattoria vaivalloisempaa.



3. Dokumentointi

Dokumentoi vain sellaiset osat, joita projektipohjassa ei ole (ts. on tehty itse). Selitä toki, minkälaisiin puitteisiin sovellus tehdään.

Käyttöliittymän osalta dokumentoidaan napit ja ledit. Sovelluksen graafinen käyttöliittymä on simulaattori, jolla ei ole harjoitustyön tekijöiden tekemän sovelluslogiikan kannalta merkitystä.

3. Infrapuna-anturin lukeman laskeminen

ALUSTAVA LASKUKAAVA:

d = (r + 1403) / 32

Huomaa: laskukaava tulee vielä muuttumaan, mutta käyttäkää tätä aluksi. Myös simulaattori käyttää tätä alustavasti.

4. Ohjelman käyttäminen fyysisellä robotilla

Ohje tulee myöhemmin.

5. C#:n käsitteitä ja koodiesimerkkejä

5.1. Poikkeusten käsittely

Poikkeusten käsittely on välttämätöntä, jotta ohjelmasta tulee vikasietoinen. Kannattaa suhtautua virhetilanteisiin siten, että mistä tahansa laitteiston toiminnosta voi lentää poikkeus. Mikäli virheen ei oleteta toistuvan uusintayrityksellä, voidaan lisätä myös silmukka, joka yrittää tarvittaessa pari kertaa uudelleen.

```
// Yritetään maks. kolmesti
for (int a = 0; a < 3; ++a)
{
    try
    {
        // Tee jotain, josta voi lentää poikkeus
        // ...

        break;
    }
    catch (Exception e)
    {
        // Käsittele poikkeus, esim. tulosta virheviesti
        // ...
    }
}
```

5.2. Avainsanat async ja await

Avainsana *async* funktion määrittelyssä tarkoittaa, että funktion suoritus voi olla pitkäkestoinen ja että se voidaan suorittaa rinnan muiden toimintojen kanssa.

Avainsana *await* koodissa tarkoittaa, että odotetaan asynkroniseksi määritetyn funktion suorituksen valmistumista. Jos jossakin funktiossa on tällainen kohta, myös sen funktion on oltava merkitty sanalla *async*.

Ks. http://stackoverflow.com/questions/14455293/how-and-when-to-use-async-and-await

5.3. Rinnakkainen suoritus: Task

Uuden loppumattoman tehtävän voi luoda vaikka kuten alla; "runTask" on tehtävää loppumatta suorittavan funktion nimi. Kun tehtävä on aloitettu, ohjelman suorittaa koodia siitä eteenpäin rinnakkain tehtävän kanssa.

Alla *runTask*-funktion toteutus. Huomioita:

- Poikkeukset. Ainakin .NET 4.5:ssä käsittelemätön poikkeus aiheuttaisi toiminnolle vain "silent failuren", ja muu osa sovelluksesta toimisi edelleen. Hoida siis poikkeusten käsittely!
- Parametrit. Funktiolle voi määrittää mielivaltaiset parametrit, vaikka tässä niitä ei ole lainkaan.

```
System.Threading.Tasks.Task.Run(() => runTask());

...

private async System.Threading.Tasks.Task runTask()
{
    while (true)
    {
        try
        {
            // Tee jotain...
        }
        catch (Exception e)
        {
            // Käsittele poikkeus, esim. lisää siitä tieto lokiin...
        }

        // Odotus 300 ms
        await System.Threading.Tasks.Task.Delay(300);
    }
}
```

5.4. Rinnakkainen suoritus: Timer

Käytä esim. luokkaa *System.Threading.Timer*. Huomaa, että Timerin tapahtumankäsittelijää suoritetaan omassa säikeessään.

5.5. Keskinäinen poissulkeminen

C#:ssa keskinäinen poissulkeminen tapahtuu näppärästi avainsanalla *lock*. Kriittiset alueet laitetaan kukin omaan koodilohkoonsa, jota jokin lukkona käytettävä olio kontrolloi. Mikäli jollakin alueella ei käytetä synkronoinnin kohteena olevaa dataa, poissulkemista ei tarvita. Lock-sana on varsin tehokas synkronointitapa: vaikka synkronoidussa lohkossa lentäisi poikkeus, jota ei oteta kiinni, lukko-olio vapautuu aina, kun lohkosta poistutaan.

Vaikka jotakin muuttujaa käsiteltäisiin aina atomisesti, se ei poista keskinäisen poissuljennan tarvetta, koska ympäristö voi aina välimuistittaa arvoja odottamattomasti. Lock-lohko estää odottamattomat rinnakkaisuusongelmat.

Suunnitteluvihjeitä:

- Pyri pitämään poissuljettujen muuttujien lukumäärä mahdollisimman pienenä.
- Hyödynnä luokkajakoa: pyri siihen, että samassa luokassa ei ole sekä poissuljettua että ei-poissuljettua dataa.

```
class SynkronoituLuokka
{
    private int jasenmuuttuja = 0;
    private object lukko = new object();

    public void kerroJaLisaa(int luku1, int luku2)
    {
        int tulos = luku1 * luku2;

        // Kriittinen alue -> tänne ei pääse, jos lukko on jo jonkun käytössä
        lock(lukko)
        {
            jasenmuuttuja += tulos;
        }
    }

    public int annaArvo()
    {
        // Kriittinen alue -> tänne ei pääse, jos lukko on jo jonkun käytössä.
        // C#:ssa primitiivityyppien (value type) muuttujista tulee kopio, kun
        // sellainen palautetaan funktiosta.
        lock(lukko)
        {
            return jasenmuuttuja;
        }
    }

    public System.DateTime mitaKelloOn()
    {
        // Funktio ei käytä mitään jäsenmuuttujaa -> poissulkemista ei tarvita
        return System.DateTime.Now;
    }
}
```

6. Yksikkötestien tekeminen

Hyvä kandidaatti yksikkötesteihin on anturien raakalukemat selkokielisiksi muuntava luokka. Laskukaavan testaaminen on suoraviivaista; toisaalta laskukaava on myös altis virheille, ja siihen päätynyt virhe ei välttämättä ole ilmeinen sovelluksen toiminnassa. Huomaa, että testattavuus on aina myös suunnittelukriteeri; jos ko. ominaisuuden testaaminen on vaikeaa, luokkajakoa kannattaa ehkä miettiä uudelleen.

Yksikkötestien tekoon suositellaan Visual Studion testikehystä.

Testiprojekti kannattaa luoda osaksi samaa solutionia kuin muukin sovellus. Testattava luokka kannattaa lisätä viitteenä testiprojektiin eikä tehdä kopiota. Tällöin luokkaan tehtävät muutokset ovat suoraan myös testiprojektissa.

Testiprojektin luonti:

1. Solution explorerissa: klikkaa solutionin juurta hiiren kakkosnapilla.
2. Valitse valikosta: Add > New project.
3. Valitse projektin tyypiksi Visual C# > Windows > Universal > Unit test app.

Testattavan luokan lisääminen testiprojektiin:

1. Solution explorerissa: klikkaa testiprojektin juurta hiiren kakkosnapilla.
2. Valitse valikosta: Add > Existing item.
3. Etsi testattava luokka.
4. Painaa Add-napissa olevaa ”kynttä”.
5. Valitse alasvetovalikosta Add as link.

Testin toteuttaminen:

1. Täydennä testitapaukset ja luo tarvittavat tynkäluokat.

Testin suorittaminen:

1. Valitse valikosta Test > Run > All tests.