Interface Ottomator100

#include <string>

#include <vector>

#include "robot.h"

#include "ottoutils.h"

class Ottomator

{

public:

Ottomator();

~Ottomator();

// Utilities

std::string getAllStatusesOfDSS1ForBit(int index, bool verbose = false);

void updateM\_statusMatrix();

void resetM\_statusMatrix();

std::vector<std::string> manageStatusGate();

int manageNextFor(int sequence, int actionSuccess, int error, int specialIndex = 0, int specialManagement = 0);

// Main Sequences

std::vector<std::string> birthSequence();

std::vector<std::string> frameworkSequenceTo(bool initiate);

std::vector<std::string> workSequenceTo(bool fetch, int sampleN);

// Demontrators

std::vector<std::string> cycler();

void displayM\_statusMatrix();

// Case getters and setters

int\*\* getM\_statusMatrix();

int getBirthSequenceCase();

void setBirthSequenceCase(int n);

int getFrameworkSequenceCase();

void setFrameworkSequenceCase(int n);

int getWorkSequenceCase();

void setWorkSequenceCase(int n);

std::vector<std::string> getM\_sampleArray();

void setM\_sampleArray(std::vector<std::string> sampleArray);

int getNCurrentSample();

void setNCurrentSample(int n);

// Integration

void writeLog(std::string text);

// Components

Robot m\_robot;

private:

// State attributs

int m\_statusMatrix[16][6];

// Cycler attibuts

std::vector<std::string> m\_sampleArray;

};

Comment utiliser Ottomator :

Méthode 1 :

#include "ottomator.h"

using namespace std;

int main()

{

// Create objects

Ottomator theSequencer;

// Connect to serial port

string port;

theSequencer.m\_robot.m\_busManager.modbusConnectRTU(port); // Eg: "COM9"

// Populate the sample array

vector<string> sampleArray;

theSequencer.setM\_sampleArray(sampleArray);

theSequencer.resetM\_statusMatrix();

// Run cycler

messages = theSequencer.cycler(); // cycler s’occupe de tout jusqu’à ce que le 1er échantillon de la liste sampleArray est mis dans le château. A ce moment, le message retourné est "Fetch completed".

// Comptage

// ...

messages = theSequencer.cycler(); // en relançant le cycler sans réinitialiser la m\_statusMatrix, le cycler s’occupe de tout jusqu’à ce que le 1er échantillon de la liste sampleArray est remis à sa place. A ce moment, le message retourné est "Putback completed". De plus le cycler retirera le 1er élément de la liste sampleArray.

// Run cycler

messages = theSequencer.cycler(); // cycler s’occupe de tout jusqu’à ce que le 1er échantillon de la (nouvelle) liste sampleArray est mis dans le château. A ce moment, le message retourné est "Fetch completed".

// Comptage

// ...

// Putback

messages = theSequencer.cycler();

// ... etc etc etc

// ... Et quand le dernier échantillon est reposé :

messages = theSequencer.cycler(); // en relançant le cycler sans réinitialiser la m\_statusMatrix, le cycler détecte que la liste sampleArray est vide. Il s’occupe alors de tout remettre en place pour terminer le batch. Le message retourné est alors "Finish completed"

}

Méthode 2 :

Tu peux aussi organiser le cycle comme tu le sens. Pour cela voici les fonctions utiles :

* Toutes les informations sont dans la m\_statusMatrix. Les 3 fonctions suivantes permettent de la mettre à jour, la réinitialiser et l’obtenir :

void updateM\_statusMatrix();

void resetM\_statusMatrix();

int\*\* getM\_statusMatrix();

* Les séquences de base sont les suivantes :

std::vector<std::string> birthSequence();

std::vector<std::string> frameworkSequenceTo(bool initiate);

std::vector<std::string> workSequenceTo(bool fetch, int sampleN);

* + Pour initialiser la machine :
    - **birthSequence()**
  + Pour préparer le terrain avant les mouvements :
    - **frameworkSequenceTo(1) ;**
  + Pour chercher un échantillon sampleN :
    - **workSequenceTo(1, sampleN) ;**
  + Pour reposer un échantillon sampleN :
    - **workSequenceTo(0, sampleN) ;**
  + Pour remettre tout en place à la fin d’un batch :
    - **frameworkSequenceTo(0) ;**

Chacune de ces fonctions retourne un vecteur de messages. Le premier permet de savoir si l’opération est s’est passé avec succès. Si erreurs, les erreurs apparaissent.

* En bonus, si tu veux surveiller le passeur en dehors d’un processus, il y a une fonction de surveillance de base.

std::vector<std::string> manageStatusGate();

* + Si tout va bien, elle retourne un vecteur vide. Sinon, les erreurs apparaissent.

Depuis l’objet Ottomator toutes les fonctions des étages inférieurs sont accessibles. Donc tu peux tout piloter si tu veux.

Les étages sont :

* Ottomator
  + Robot
    - BusManager
      * (libmodbus)

Coordonnées des points :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | D |  |  |  |  |  |
| 2 | 1 | 2 | 3 | 4 | 5 |  |
| 3 | 6 | 7 | 8 | 9 | 10 |  |
| 4 | 11 | 12 | 13 | 14 | 15 |  |
| 5 | 16 | 17 | 18 | 18 | 20 |  |
| 6 |  |  |  | A |  | C |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 6 | 5 | 4 | 3 | 2 | 1 |
| 1 | 16 | 11 | 6 | 1 |  |  |
| 2 | 17 | 12 | 7 | 2 |  |  |
| 3 | 18 | 13 | 8 | 3 |  |  |
| 4 | 19 | 14 | 9 | 4 |  |  |
| 5 | 20 | 15 | 10 | 5 |  |  |
| 6 |  |  |  | A | D | C |

// State attributs

int m\_statusMatrix[16][6];

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | nC | x | y | z | p | v |
| 1 | F | CLBS | CLBS | CLBS | CLBS | LAT |
| 2 | I | CEND | CEND | CEND | CEND | EMGV |
| 3 | B | PEND | PEND | PEND | PEND | P1 |
| 4 | Fw | HEND | HEND | HEND | HEND | P2 |
| 5 | W | STP | STP | STP | STP | MOTO |
| 6 |  |  |  |  |  | OPEN |
| 7 |  | BKRL | BKRL | BKRL | BKRL | CLOS |
| 8 |  | ABER | ABER | ABER | ABER |  |
| 9 |  | ALML | ALML | ALML | ALML |  |
| 10 |  | ALMH | ALMH | ALMH | ALMH |  |
| 11 |  | PSFL | PSFL | PSFL | PSFL |  |
| 12 |  | SV | SV | SV | SV |  |
| 13 |  | PWR | PWR | PWR | PWR |  |
| 14 |  | SFTY | SFTY | SFTY | SFTY |  |
| 15 |  | EMGS | EMGS | EMGS | EMGS |  |

nC = nCurrentSample

F = isFetching

I = isInitiating

B = birthSequenceCase

Fw = FrameworkSequenceCase

W = WorkSequenceCase

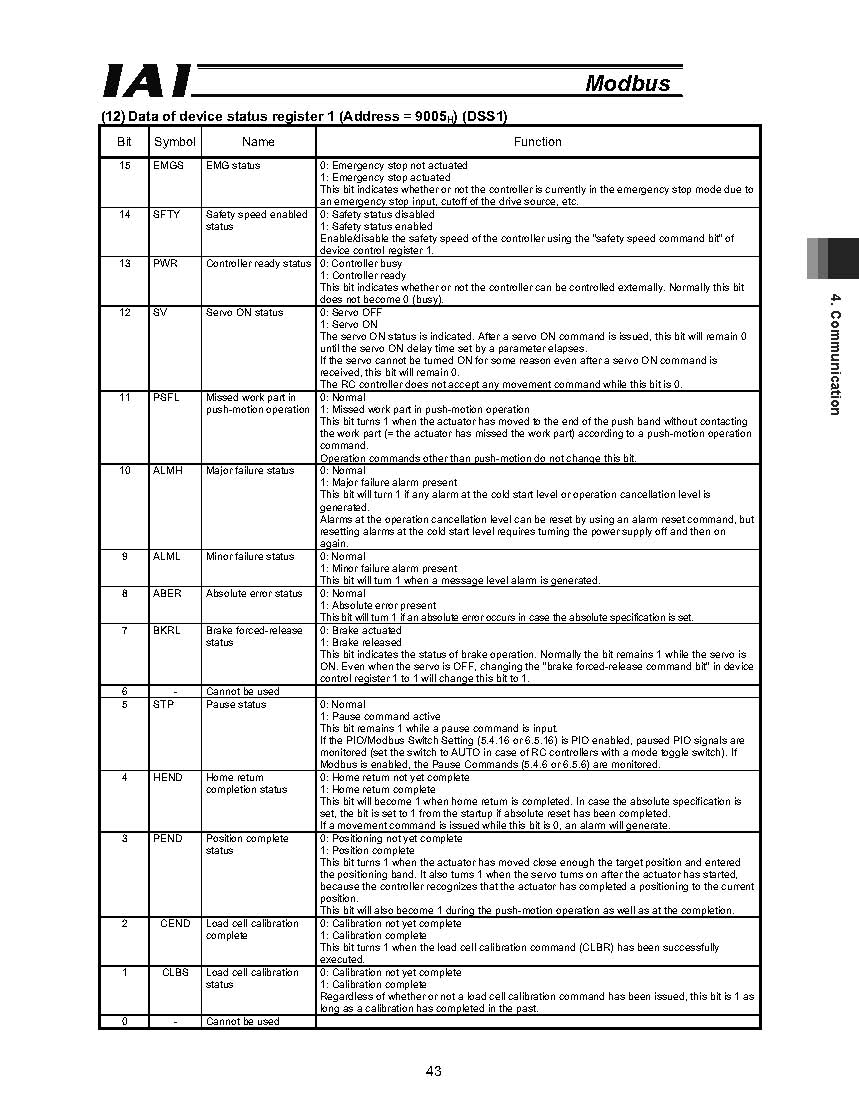
X = position of x axis (0..6, 0 = Home)

Y = position of y axis (0..6, 0 = Home)

Z = position of z axis (0..3, 0 = Home, 1 = High, 2 = Table, 3 = Detector)

P = position of plier (0..2, 0 = Home, 1 = Opened, 2 = Closed)

V = position of castel actuator (1 = Opened, 2 = Closed)



The corresponding bit gets a 1 if not normal value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Messages | Normal value | Bit weight | Fatal error |
| 0 | SV | 1 | 1 |  |
| 1 | STP | 0 | 2 |  |
| 2 | EMGS | 0 | 4 |  |
| 3 | EMGV | 1 | 8 |  |
| 4 | P1 | 0 | 16 |  |
| 5 | P2 | 0 | 32 |  |
| 6 | LAT | 1 | 64 |  |
| 7 | ALML | 0 | 128 |  |
| 8 | ALMH | 0 | 256 | 1 |
| 9 | ABER | 0 | 512 | 1 |
| 10 | MOTO | 0 | 1024 | 1 |
| 11 | XCAS | PSFL at completion | 2048 | 1 |
| 12 |  |  |  |  |
| 14 |  |  |  |  |
| 15 |  |  |  |  |
| 16 |  |  |  |  |

-1 = Stuck in birth sequence

-2 = Stuck in initiation sequence

-3 = Stuck in fetch sequence

-4 = Stuck in putback sequence

-5 = Stuck in finish sequence

-6 = Sequence case out of range

-7 = Empty catch

-8 = Birth completed

-9 = Initiation completed

-10 = Fetch completed

-11 = Putback completed

-12 = Finish completed

-13 = Need input

Logique de « cycler » :

**Reset :**

nC =0

F = 1

I = 1

B = 1

Fw = 1

W = 1

**Begin :**

nC =0

F = 1

I = 1

B = 1

Fw = 1

W = 1

New input

Begin

Begin

**I = 0**

Begin

**Update** sArray

sArray.size > 1

sArray.size = 1

Run or New input

Begin

**"Fetch completed"**

**"Finish completed"**

Begin

Begin

F = 0

F = 1

**Putback**

**"Error …"**

**"Error …"**

I = -1

**"Error …"**

**"Error …"**

**Finish**

**Init**

I = 0

I = 1

B = 11

**Fetch**

B !=11

**"Error …"**

**Birth**

**"Need input"**

sArray.size = 0

sArray.size > 0