RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'électronique

Mémoire de Master en électronique

Etude des approches de développement des Blocs Simulink. Application au Microcontrôleur ARM LPC1768.

Abdelghafour SID

Sous la direction de
Mr. R. SADOUN

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Président	M .Taghi	MAA	Ecole Nationale Polytechnique
Promoteur	R .SADOUN	MCA	Ecole Nationale Polytechnique
Examinateur	M .ADNANE	PhD	Ecole Nationale Polytechnique

ENP 2017

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'électronique

Mémoire de Master en électronique

Etude des approches de développement des Blocs Simulink. Application au Microcontrôleur ARM LPC1768.

Abdelghafour SID

Sous la direction de
Mr. R. SADOUN

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Président	M .Taghi	MAA	Ecole Nationale Polytechnique
Promoteur	R .SADOUN	MCA	Ecole Nationale Polytechnique
Examinateur	M .ADNANE	PhD	Ecole Nationale Polytechnique

ENP 2017

ملخص

الهدف من هذا العمل هو دراسة منهجيات تطوير قوالب و بتحديد قوالب مخصصة للمحيطات الخاصة بهذا المتحكم لقيام بالاتصالات مثل (ADC, CAN, SPI...).

وبعد الدراسة نختار طريقة ونقوم بتطبيقه على متحكم ARM LPC1768

كلمات الدالة :

ARM, LPC1768 ADC, CAN, SPI

Abstract:

The aim of this work is to study the development methodologies of Simulink blocks, and especially blocks for peripherals for hardware targets, after that we applying a method to develop peripherals for communication (ADC, CAN, SPI ..) for ARM LPC1768.

Key Words: Block Simulink, ARM, LPC1768 ADC, CAN, SPI

Résumé

Le but de ce travail d'étudier les méthodologies de développement des blocs Simulink, et particulièrement des blocs pour des périphériques pour des cible hardware, après on applique une méthode pour développer des périphériques pour la communication (ADC, CAN, SPI..) pour ARM LPC1768.

Mots clés : Bloc Simulink, ARM, LPC1768 ADC, CAN, SPI

Remerciement

En premier lieu, je tiens à remercier Dieu, de ma avoir aidé et de m'avoir donné la patience et la foi pour finaliser ce mémoire.

Au terme de ce travail, je tiens à exprimer ma profonde gratitude et mes sincères remerciements à mon encadreur Mr Rabah SADOUN pour tout le temps qu'il m'a consacré, ses directives précieuses, et pour la qualité de son suivi tout au long du projet.

Mes plus vifs remerciements s'adressent aussi à tout le cadre Professeurs et administratifs de L'école Nationale Polytechnique Alger

Mes remerciements vont en à toute personne qui a contribué de près ou de loin à l'élaboration de ce travail.

Abdelghafour SID

Table des matières

LISTE DES FIGURES

LISTE DES FIGURES

Introduction Générale

Aujourd'hui, il existe plus de systèmes intelligents qu'il n'y a d'humains sur terre. Comme les fonctionnalités et les capacités de ces systèmes augmentent, ils deviennent plus complexes. Le logiciel embarqué d'une voiture moderne par exemple comporte environ deux cents millions de lignes de code [4]. Concevoir, développer et tester un programme de cette taille qui interagit avec les humains et le monde physique très utiliser le Model Based Design.

Model-Based Design offre de nombreux avantages et rend le processus de développement plus rapide, plus fiable, avec une qualité supérieure, un coût inférieur et une plus grande souplesse.L'outil principal utilisé pour appliquer ce processus de développement est Simulink.

Simulink est une plate-forme de simulation multi-domaine et de modélisation de systèmes dynamiques. Il fournit un environnement graphique et un ensemble de bibliothèques contenant des blocs de modélisation qui permettent le design précis, la simulation, et le contrôle de systèmes de communications et de traitement du signal.

Il prend en charge la conception et la simulation au niveau système, la génération automatique de code, ainsi que le test et la vérification en continu des systèmes embarqués.

L'outil Embedded coder sur Simulink qui nous permet de générer les codes à partir d'un modèle Simulink ou MATLAB, avec la possibilité de manipuler les périphériques de notre cibles et générer des codes prêt à être compiler et exécuter.

Le problème c'est qu'on n'a pas des blocs périphériques pour toutes les cibles, il y a plusieurs Cibles qui sont très utilisé mais ils n'ont pas des périphériques. Notre but est de présenter les méthodes de développer des blocks Simulink pour des périphériques de n'importe quelle Cible Hardware. Eton applique la méthode la plus simple sur un ARM LPC1768.

.

Chapitre 1 Génération de code et Mode d'exécution sur Simulink

Introduction

La génération automatique de code est l'une des tâches les plus difficiles dans le domaine de l'ingénierie logicielle. Les algorithmes utilisés pour générer le code, la validation et la vérification du code peuvent varier en fonction du type de modèles, du champ d'application, du matériel cible et de la complexité du modèle.

2.1 Génération de Code :

On génère de code automatique nous on veut dire générer automatiquement le code(C / C ++) ou HDL, des programmes Matlab ou des modèles Simulink en utilisant les outils fournis par MathWorks. Ce processus peut également impliquer la génération de fichiers Make pour la compilation et la liaison et aussi la construction des exécutables de production à partir de Matlab ou de tout autre IDE.

L'intérêt de la génération de code automatique

1- La première raison est d'accélérer l'exécution du modèle. Les programmes Matlab et les modèles Simulink sont toujours interprétés et exécutés à partir de l'interprète Matlab et du moteur de simulation Simulink. La nature interprétative de l'exécution est la plupart du temps "moins efficace" que l'approche compilée. C'est pourquoi, dans certains cas, nous préférons compiler et exécuter out ou partie de notre modèle sur une autre plateforme pour accélérer l'exécution. [1]
2- la génération de code est de pouvoir créer des exécutables autonomes pour les systèmes embarqués et les robots afin que l'algorithme développé à l'aide des produits MathWorks puisse être implémenté sur le matériel réel et exécuté sans aucune dépendance à la plate-forme de développement et de modélisation. [1]
3- La nécessité de communiquer avec le matériel, par exemple dans le test Hardware-In-the-Loop (HIL), ce qui est possible en utilisant un code de pilote de périphérique matériel qui est généralement écrit en C / C ++.[1]

L'outil de génération de code

Embedded Coder génère un code C et C++ lisible, compact et rapide pouvant être utilisé sur les processeurs embarqués, les cartes de prototypage rapide sur cible et les microprocesseurs utilisés dans la production en grande série. Embedded Coder permet des options de configuration MATLAB Coder et Simulink Coder supplémentaires, ainsi que des optimisations perfectionnées pour contrôler finement les fonctions, fichiers et données du code généré. Ces optimisations améliorent l'efficacité du code et facilitent l'intégration avec le code existant, les types de données et les paramètres d'étalonnage employés dans la production. Il est possible d'incorporer un environnement de développement tiers au processus de compilation

pour produire un exécutable en vue d'un déploiement clé en main sur le système embarqué [2].

2.2Mode d'exécution sur Simulink :

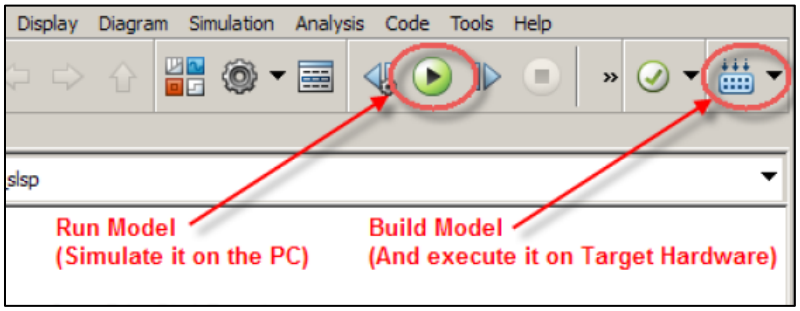


Figure 1 : Deux méthodes d'exécuter un modèle

Première méthode

Lorsqu'on clique sur le bouton vert « Run » tel qu'on a choisi le mode normal, si le modèle est en Simulation cela signifie que l'exécution est sur l'ordinateur.

Pour exécuter le bloc S-FunctionBuilder, Simulink appelle le fichier MEX (MATLAB Exécutable) file. Ce fichier est généré à partir du code C écrit dans le bloc lorsque le bouton "build" de la boîte de dialogue S-Function est pressé. En générale les blocs de périphérique n'effectue aucune opération en simulation de plus, les entrées et les sorties sont toujours de valeur 0.

Deuxième méthode

Le modèle peut être exécuté en générant des codes a partir de modèles et exécuter ces codes sur la cible, si Embedded coder est installé sur Simulink on peut cliquer sur le bouton « Build Model » pour générer les codes et les exécuter sur la cible, mais il est nécessaire d'avoir des blocs de périphérique pour communiquer avec la cible.

On utilise Embedded Coder pour générer les codes de notre modèle Simulink, pour que le code soit compatible avec notre cible on doit installer le package nécessaire, pour ce faire, on utilise la fonction de MATLAB « ADD-ON » qui nous permet d'installer manuellement après avoir télécharger le package à partir de web site de Mathworks.

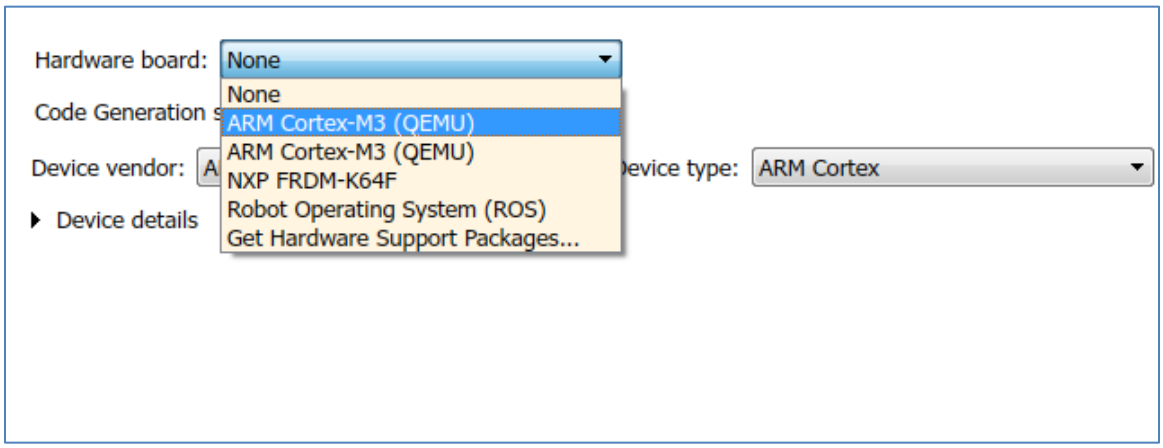


Figure 2 : Package installer sur Simulink

Il existe plusieurs outils sur Simulink qui nous permettent d'optimiser les codes générés et assurer que les codes sont compilable et exécutable sans erreur, parmi les outils d'optimisation sur Simulink le **Code Generation Advisor**, il nous donne des propositions de changement du paramétrage de modèle et des propositions pour éliminer des blocs qui ne servent à rien dans le modèle, il nous donne la main pour vérifier l'optimalité de l'occupation RAM,ROM par rapport au temps d'exécution et permet de détecter des erreurs d'exécution par exemple (division par zéro...)

Mode externe

Dans ce mode d'exécution, pendant l'exécution de l'algorithme sur la cible, Simulink étant actif sur l'ordinateur, il communique avec la cible tel un outil de visualisation ou générateur de données, on peut par exemple envoyer vers l'algorithme qui s'exécute sur la cible des signaux à partir de Simulink et voir les sortie sur un Scop ou Display.

Conclusion

La plupart des packages ne permette pas de manipuler les périphériques de notre cible, l'objectif du chapitre suivant explique comment développer des périphériques pour la cible.

Chapitre 2 Approches de développent des blocks Simulink pour périphériques

3.1 Introduction

Il y a plusieurs approches pour créer des blocks pour notre cible. La première méthode pour développer des blocs de périphériques est basée sur le bloc S-FunctionBuilder cette approche est la plus simple, la deuxième approche est d'utiliser Legacy Code Tool (LCT), et la dernière méthode est basée sur the System Object block.

3.2 The Legacy Code ToolApproach

The Legacy Code Tool (LCT) est un outil de Simulink qui permet aux utilisateurs d'intégrer des fonctions C (ou C ++) existantes dans un modèle de simulation et de génération des code, ce qui peut être facilement utilisé pour développer des blocs de périphériques.

Cette approche est basée sur la création d'au moins deux fichiers distincts qui doivent être intégré ensemble, Le premier est un fichier C (ou C ++) qui gère les interactions de niveau inférieur avec la cible, le deuxième est un fichier MATLAB nécessaire pour définir une structure de données (contenant des informations telles que les fonctions à appeler et leur emplacement, les entrées et les sorties...).

On commence par crée le fichier C (ou C ++) pour l'intégrer dans le modèle Simulink

On prend cet exemple d'un fichier C++ de ARDUINO

```
#include <Arduino.h>
#include "aout_arduino.h"

extern "C" void aout_init(uint8_T pin) {
    pinMode(pin, OUTPUT);
}

extern "C" void aout_output(uint8_T pin, uint8_T val) {
    analogWrite(pin, val);
}
```

Figure 3 : Exemple d'un fichier C++ de ARDUINO

Pour intégrer ce code on doit créer un script MATLAB :

Partie1:

```
% 1) Initialization
def = legacy_code('initialize');
def.SFunctionName = 'aout_sfun';
def.SampleTime = 0.05;
```

Cette partie initialise la structure de données de « Legacy Code Tool » et définit le nom du block et la fréquence d'échantillonnage

Partie 2:

```
% 2) Simulation
def.OutputFcnSpec = 'void NO_OP(uint8 p1, uint8 u1)';
def.StartFcnSpec = 'void NO_OP(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def, '-DNO_OP=//')
```

Cette deuxième partie spécifie les détails de la fonction qui doivent être intégrées, par exemple : Types de données ou nombre d'arguments, tout en spécifiant que le comportement du code est en mode simulation.

L'option du compilateur: -DNO_OP = //,Remplace chaque occurrence de "NO_OP" par "//" Donc effectivement commentant les fonctions d'initialisation et de sortie. En conséquence, aucun code ne sera appelé dans la simulation.

Partie3:

```
% 3) Executing on Hardware
def.SourceFiles = {fullfile(pwd,'..','src','aout_arduino.cpp')};
def.HeaderFiles = {'aout_arduino.h'};
def.IncPaths = {fullfile(pwd,'..','src')};
def.OutputFcnSpec = 'void aout_output(uint8 p1, uint8 u1)';
def.StartFcnSpec = 'void aout_init(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
legacy_code('rtwmakecfg_generate',def);
```

L'étape suivante consiste à redéfinir OutputFcnSpec et StartFcnSpec(les propriétés de la structure de données), afin d'appeler les fonctions pour interagir avec la cible. Nous spécifions également l'emplacement des fichiers source et d'en-tête qui doivent être inclus.

Les dernières lignes de cette section sont utilisées pour générer le fichier de compilateur de (TLC) et l'exécutable qui s'exécute sur la cible.

Partie 4:

% 4) Create Simulink Block
legacy_code('slblock_generate',def);

Enfin, dans Cette section de code on génère un bloc S-Function. Le bloc S-Function est également masqué automatiquement, donc on ne peut manipuler que les paramètres de bloc.

3.3. The MATLAB System Block Approach

Cette approche est une interface avec les objets systèmes, ces objets systèmes sont des classes héritées à partir de *matlab.System* et contiennent des API qui nous permettent de développer des blocs pour des périphériques.

API : est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciel.

L'objet système a des méthodes appelées setup, step et release, qui peuvent être utilisées pour définir l'initialisation, la sortie et le code de terminaison pour un bloc de périphériques.

Voici l'Objet système pour tous les blocs des périphériques

Partie 1

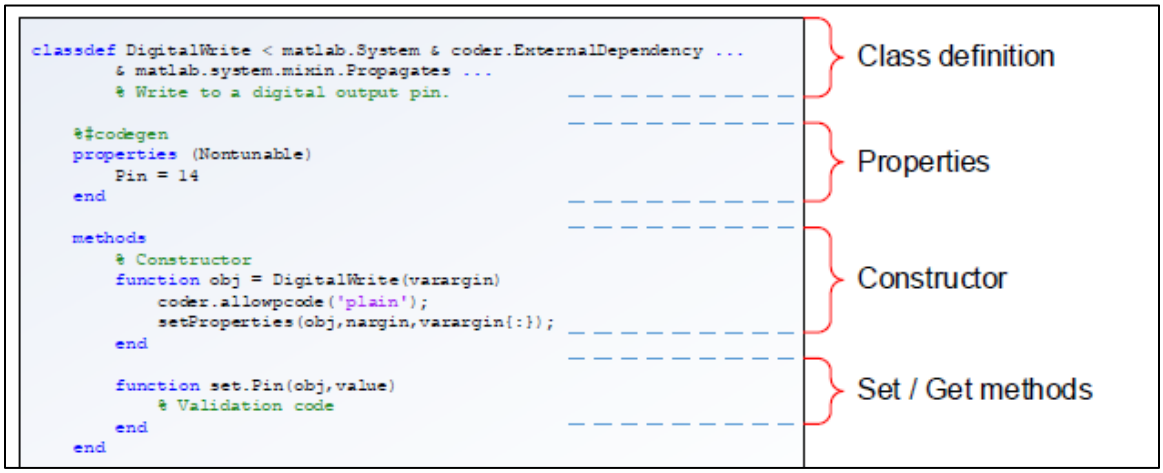


Figure 4 : Partie 1 de code Objet système

Class definition:

On définit le nom de l'objet et les classes dont il hérite, Tous les objets système doivent Hériter de matlab.System.

Properties :

On définit les propriétés de bloc par exemple un bloc représente une sortie analogique et le paramètre est un numéro correspondant à un pin

Constructor

On définit des classes pour l'initialisation de la mémoire

Set / Getmethods

Cette partie nous permettra de vérifier et de valider le code par exemple on peut vérifier le code dans la partie propriété avant de lui attribuer une propriété spécifique.

Partie 2 :

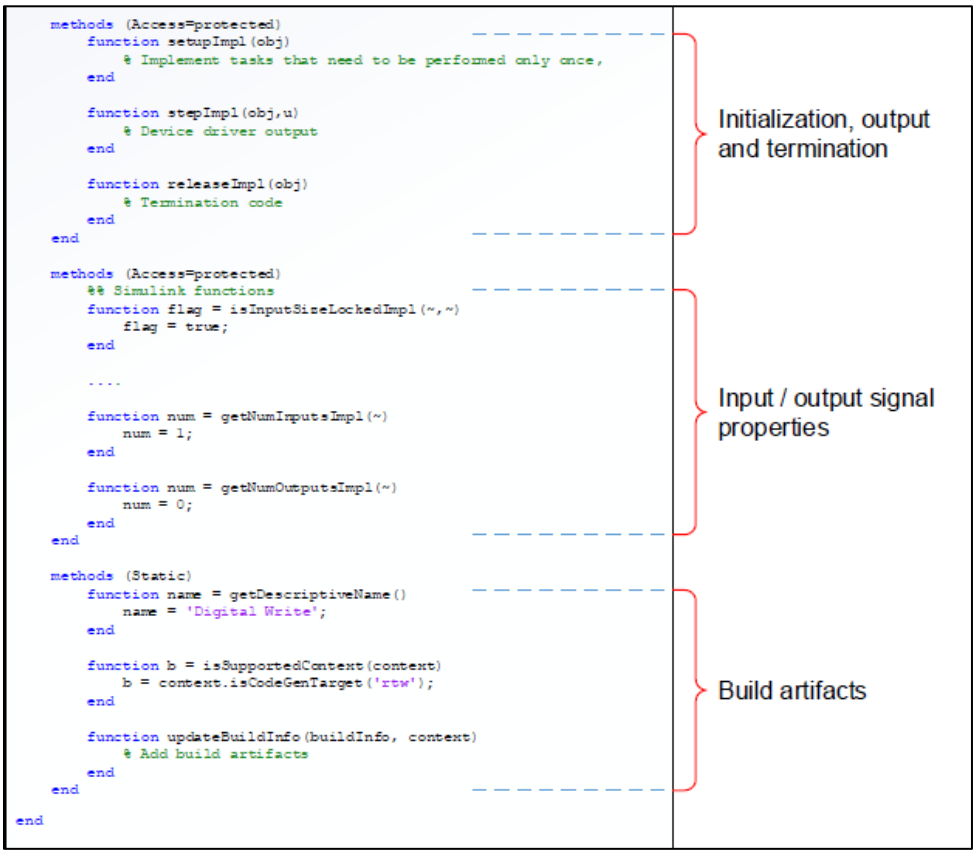


Figure 5 : Partie 2 de code Objet système

Initialization, output and termination

Les méthodes -impl dans cette section nous permettrons de définir ce qui se passelors de l'initialisation, de la sortie et de la fin. Nous utilisons setupImpl pour initialiser le périphérique, stepImpl pour lire ou écrire sur le périphérique et releaseImpl pour libérée les ressources utilisées. Ces trois méthodes sont l'arrière-plan de la définition du comportement d'un bloc de pilotes de périphériques.

Input / output signal properties

Dans cette section de code, nous définissons le nombre d'entrées er de sorties d'un bloc, les types et les tailles de données.

Buildartifacts:

Dans cette section du code, on définit les fichiers sources, les chemins d'accès, les bibliothèques partagées, les chemins de recherche de la bibliothèque et les définitions du préprocesseur requises pour compiler le code du pilote de périphérique.

Remarque

Cette approche est la plus complexe par rapport aux autres méthodes, elle est destiné pour développer des blocs périphériques complexes, pourraient être meilleures pour les développeurs plus familiers avec la programmation en orienté objet.

Cette approche n'est compatible qu'avec les versions MATLAB R2015a ou plus

3.4 The S-FunctionBuilderApproach

La première étape est d'ajouter le bloc S-FunctionBuildera partir de User-DefinedFunctions sur la bibliothèque de Simulink.

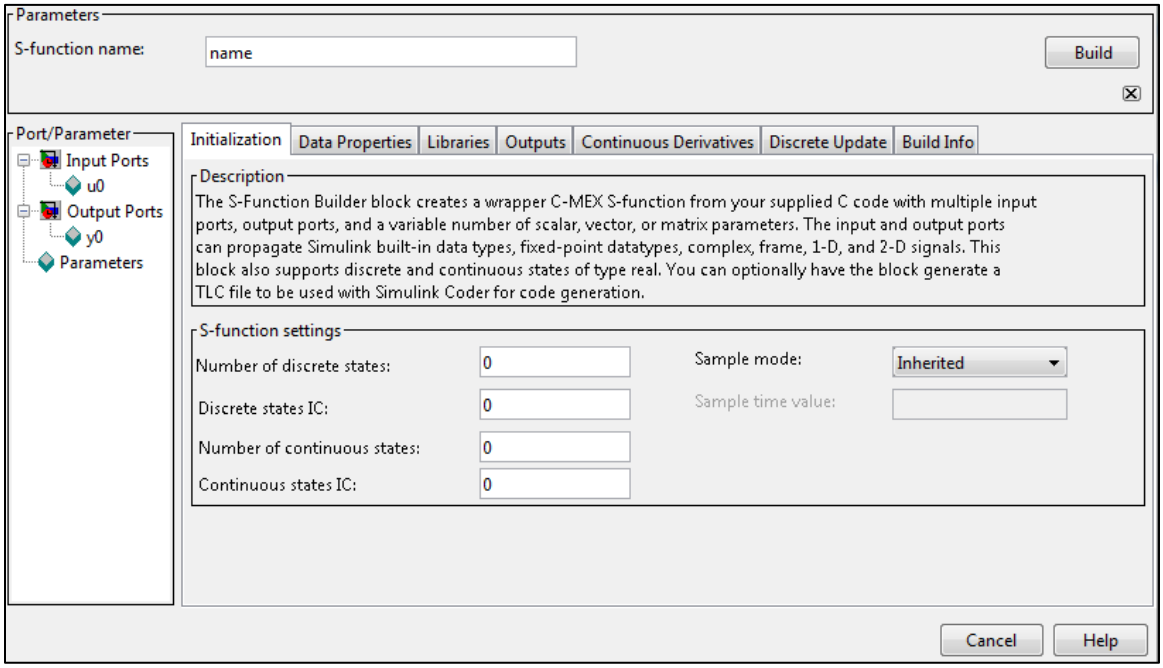


Figure 6 : S-FunctionBuilder

On commence par spécifier le nome de blocs sur **S-functionname,**sur **l'initialisation** on établit le temps d'échantillonnage du bloc et son nombre d'états continus et discrets. Normalement, les blocs périphériques s'exécutent en temps discret et n'ont pas d'états continus. Dans ce cas, nous avons choisi de régler l'heure d'échantillonnage à 50 ms
Cette implémentation d'un bloc de pilote qui nécessite qui doit définir au moins un seul état de temps discret, qui doit être initialisé à 0. On pourrait ajouter d'autres états si nécessaire, mais le premier élément du vecteur d'état discret (c'est-à-dire xD [0]) doit être initialisé à 0 pour que la partie d'initialisation (que nous verrons plus bas) puisse fonctionner.

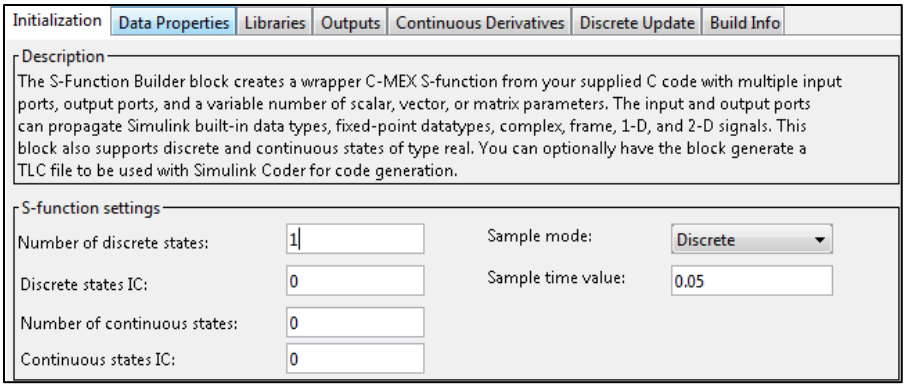
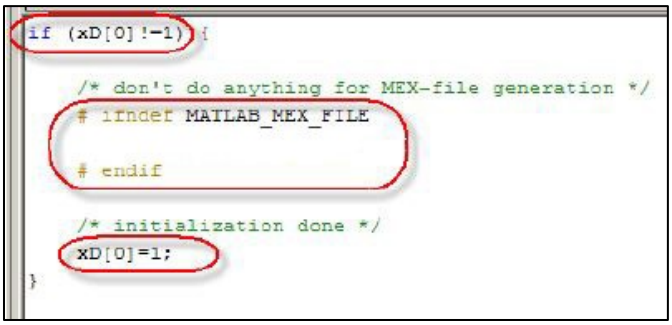


Figure 7 : S-Function settings

Sur **Data Properties** on ajoute des entres et des sortie et on spécifier les types d'entre et de sortie et la taille.

Discrete Update

La condition initiale pour l'état discret est 0 (ceci est configuré par l'onglet d'initialisation), donc la première fois que cette fonction Discret Update est appelée xD [0] est à 0 et le code dans les parenthèses suivant le "if" est exécuté. La dernière ligne entre parenthèses metxD [0] à 1, ce qui empêche tout autre élément dans les parenthèses d'être exécuté à nouveau



Figue 8 : Discrete Update

Outputs pane :

Définit les actions que le bloc effectue (en général sur ses sorties) lorsqu'il est exécuté, La première chose à remarquer est que le code entre parenthèses suit la condition xD [0] == 1. Puisque xD [0] est à 0 au début et est ensuite mis à 1 par le premier appel de **Discrete Update,** cela signifie que le code entre parenthèses est exécuté uniquement après l'exécution du code d'initialisation

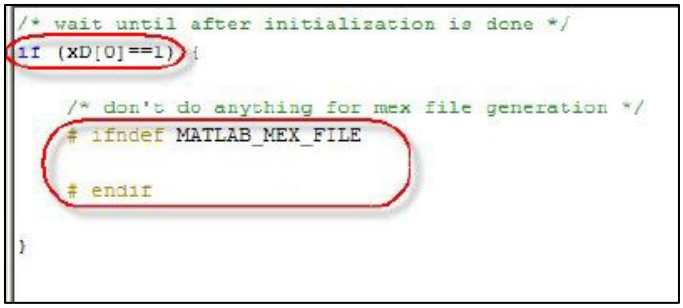


Figure 9 : Outputs pane

Lorsque le fichier MEX est généré à partir du bloc S-Function, l'identifiant "MATLAB_MEX_FILE" est défini au moment de la compilation, l'instruction de compilation conditionnelle # ifndef MATLAB_MEX_FILE empêche tout le code qui suit (jusqu'à # endif) d'être inclus dans la compilation lorsque l'identifiant MATLAB_MEX_FILE est défini.

En conséquence, lors de la génération de l'exécutable pour la simulation les codes écris dans cet espace ne seront pas inclus dans la compilation, et le code résultant ressemblera à ceci

```
if (xD[0]!=1) {
xD[0]=1;
}
```

Ce code définira simplement xD [0] à 1 la première fois qu'il sera exécuté et ne fera plus rien.

D'autre part, lorsqu'un exécutable qui doit être exécuté sur la cible, est généré, l'identifiant "MATLAB_MEX_FILE" ne sera pas défini et, par conséquent, la ligne centrale sera incluse dans la compilation et le code résultant ressemblera à ce qui suis :

```
if (xD[0]!=1) {

#code

xD[0]=1;

}
```

Libraries:

Ces ongletsnous permettront d'ajouter des bibliothèques externes, par exemple « mbed.h, arduino.h » pour ajouter les fonctions des périphériques.

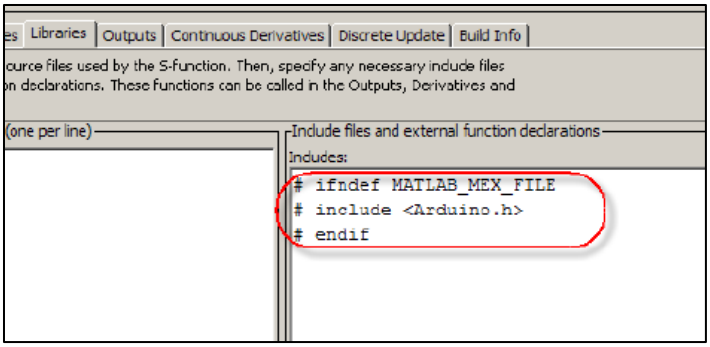


Figure 10 : L'ajoute des bibliothèques externes

Après avoir rempli tous les paramètres on clique sur le bouton « Build » ce qui va nous générer un fichier Name_ wrapper.c qui contient Les codes qu'on a ajoutés suivant :

Les Bibliothèques externes qu'on a ajoutées :

```
/* %%%-SFUNWIZ_wrapper_includes_Changes_
# ifndef MATLAB_MEX_FILE
# include <Arduino.h>
# endif
/* %%%-SFUNWIZ_wrapper_includes_Changes
```

Figure 11 : Bibliothèques externes

La Fonction de sortie, appelée à chaque instant d'échantillonnage, elle contient le code inséré dans " Outputs pane "

```
void LED_test_Outputs_wrapper(const boolean_T *in,
                const real_T *xD)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
/* This sample sets the output equal to the input
      y0[0] = u0[0];
 For complex signals use: y0[0].re = u0[0].re;
      y0[0].im = u0[0].im;
      y1[0].re = u1[0].re;
      y1[0].im = u1[0].im;
*/

# ifndef MATLAB_MEX_FILE
```

Figure 12 : La Fonction de sortie

La Fonction de mise à jour, appelée à chaque instant d'échantillonnage pour calculer la valeur suivante du vecteur d'état interne xD , celle-ci contient le code inséré dans le «Discrète Update »

```
void LED_test_Update_wrapper(const boolean_T *in,
                real_T *xD)
{
  /* %%%-SFUNWIZ_wrapper_Update_Changes_BEGIN --- EDIT HERE TO _END */
/*
 * Code example
 *   xD[0] = u0[0];
```

Figure 13 : La Fonction de mise à jour

Conclusion :

L'approche System Object a tendance à fonctionner mieux pour développer des blocs complexes. D'autre part, les approches S-Function Builder et LCT pourraient être meilleures pour les développeurs plus familiers avec C / C ++ que System Objects et qui veulent développer rapidement des blocs simples qui ont peu de paramètres.

Chapitre 3 Application LPC1768 :

Introduction

On prend l'exemple de LPC1768 pour lequel nous allons utiliser la méthode la plus simple qui est S – Function Builder Approach qui consiste à développer les différents périphériques.

Exemple 1 : LED

On commence par crée un bloc qui pilote les LED de LPC1768 :

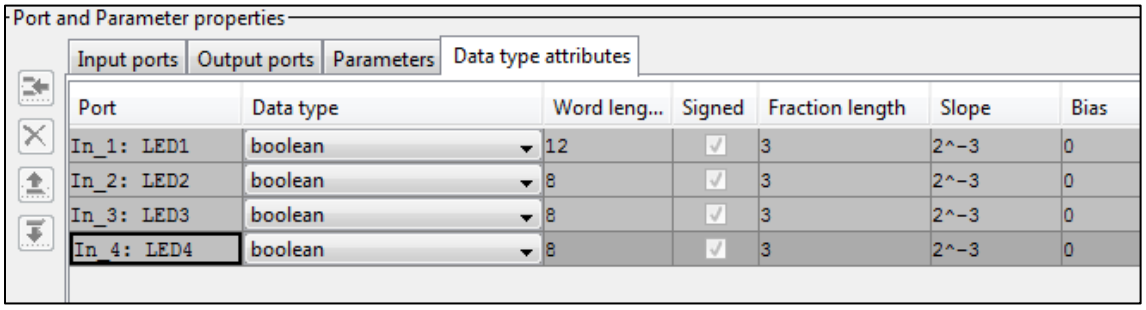


Figure 14 : Définition des entres et les sortie et leur type

On va travailler avec la bibliothèque mbed.h

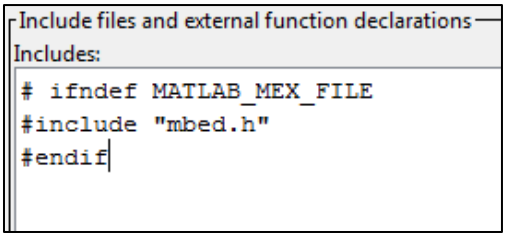


Figure 15 : Bibliothèque externe

On définit la fonction de bloc et les constantes utilisées

```
# ifndef MATLAB_MEX_FILE
 DigitalOut myled1(LED1);
DigitalOut myled2(LED2);
DigitalOut myled3(LED3);
DigitalOut myled4(LED4);
if (led1[0] == 0)
    myled1=0;
else  myled1=1;
if (led2[0]==0)
    myled2=0;
else  myled2=1;
if (led3[0]==0)
    myled3=0;
else  myled3=1;
if (led4[0]==0)
    myled4=0;
else  myled4=1;
# endif
```

Figure 16 : Comportement du bloc à la sortie

Après l'écriture des codes on clique sur build pour générer la fonction Name_ wrapper.c, L'étape suivant est de crée un modèle et de générer les codes puis les tester sur LPC1768 :

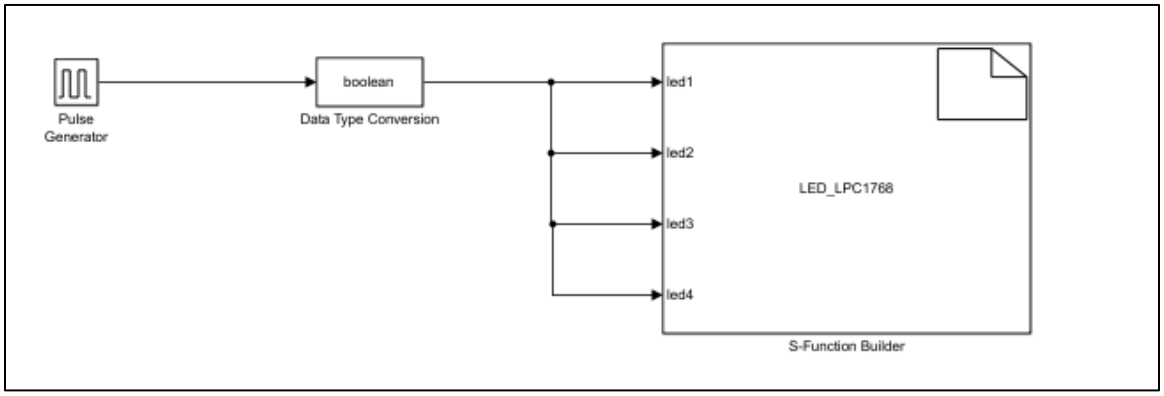


Figure 17 : Modèle pour tester le bloc LED_s

Pour générer les codes il faut s'assurer que le package ARM Cortex-M3 est sélectionné sur configuration – Hardware implémentation

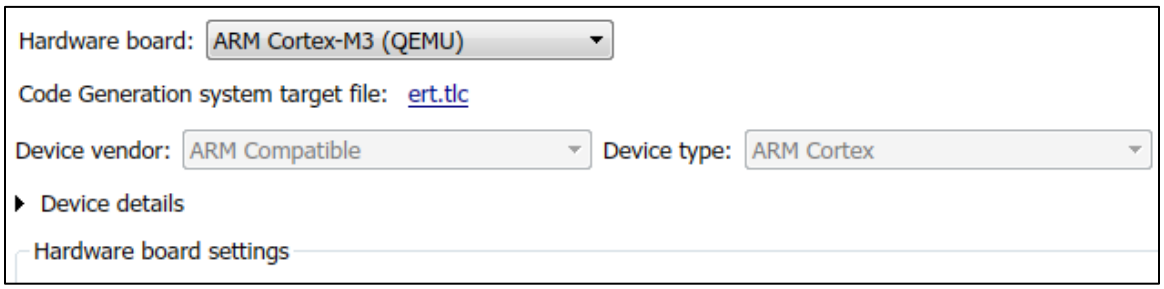


Figure 18 : Sélection du Package ARM Cortex-M3

On presse build pour générer les codes de modèle et on transfert tous les codes C en C++ pour les compiler avec la bibliothèque mbed.h ,La fonction de modèle est appelé : rt_OneStep();

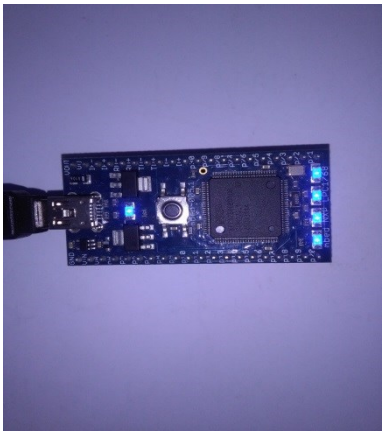


Figure 19 : Démonstration pour le bloc LED_s

Remarque :

Pour exécuter le modèle, on tient compte de la fréquence d'échantillonnage de bloc pour notre cas elle est de 20hz alors c'est à dire que le modèle est exécuté chaque 0.05 s

Exemple 2 ADC :

On développe un bloc ADC et un bloc printf pour écrire sur le port série

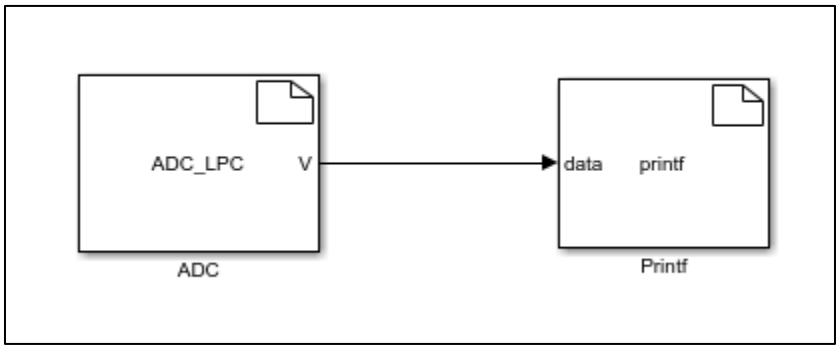


Figure 20 : Modèle ADC + printf

Résultat

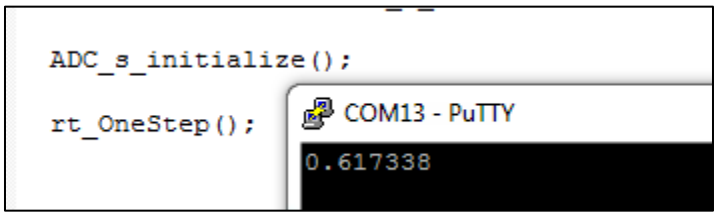


Figure 21 : Démonstration pour bloc ADC

Exemple 3 CAN transmit

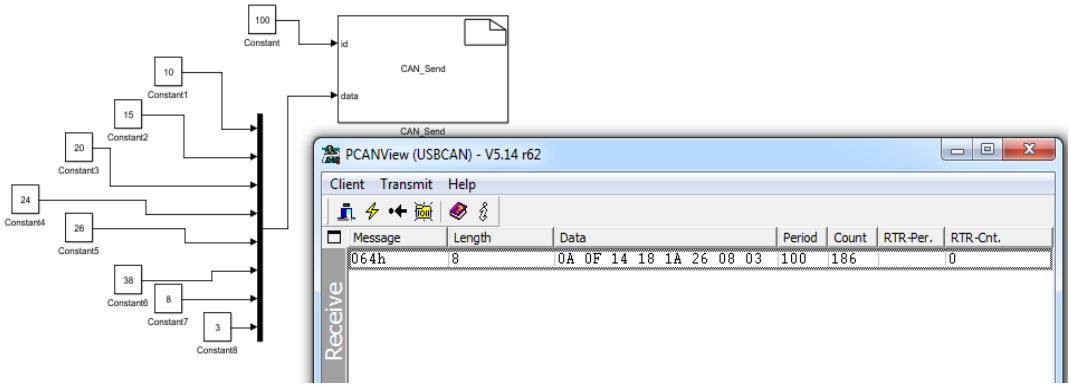


Figure 22 : CAN transmit

Création d'une bibliothèque :

Après la création des blocs on ajoute une bibliothèque qui contient tous les blocs pour LPC1768, alors pour commencer à programmer notre cible on a besoin simplement d'ouvrir la bibliothèque et ajouter les blocs dont on a besoin.

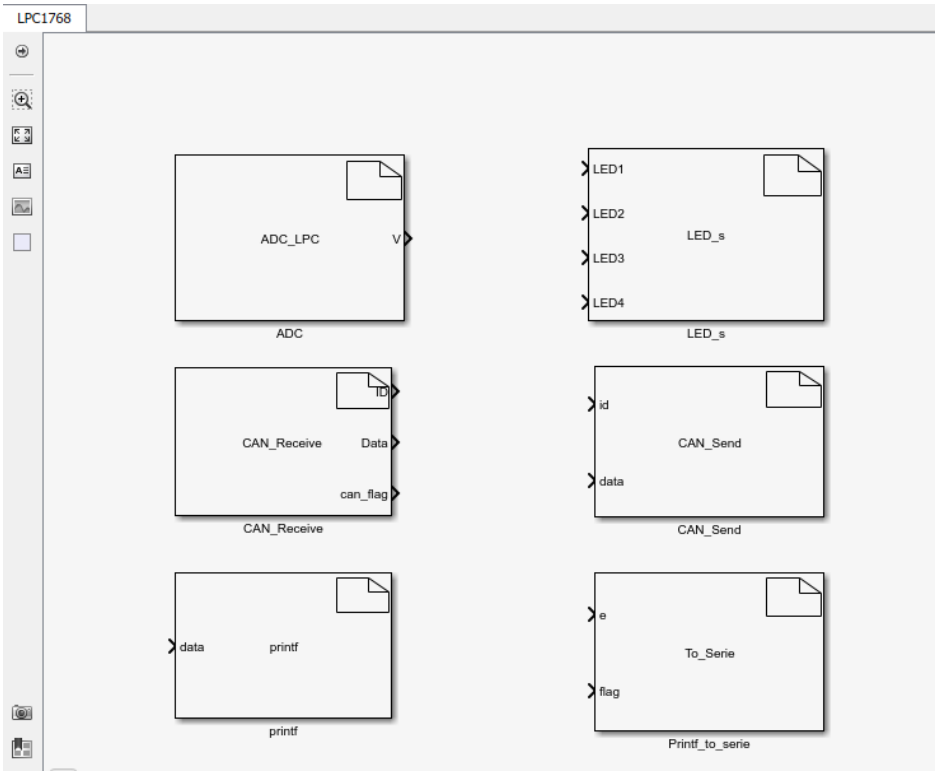


Figure 23 : Bibliothèque LPC1768 sur Simulink

Remarque :

Il est possible de masquer un bloc S-Function Builder comme pour tout autre bloc Simulink. Pour ce faire, il faut sélectionner le bloc et cliquer avec le bouton droit de la souris et sélectionnez "Masquer" puis "Modifier masque" dans le menu, ou utiliser simplement le raccourci clavier "Ctrl + M".

Conclusion :

On a présenté les méthodologies de développement des blocs Simulink pour des périphériques, les deux premières approches (**The Legacy Code Tool, The MATLAB System Block**) sont des méthodes plus difficile à mettre en place, car elles ont besoin d'expérience en programmation C++ et programmation en orienté objet, après on a détaillé l'approche **S-FunctionBuilder**et on a donnée des exemples et tester ces exemples sur notre cible LPC1768.

Bibliographie

[1] RamtinRajiKermani. Model-based Design, Simulation and Automatic Code Generation For Embedded Systems and Robotic Applications,ARIZONA STATE UNIVERSITY December2013.

[2] Lukas Armborst. Generating Simulink Models from AADL system descriptions,RWTH Aachen University Chair for Software Modeling and Verification.

[3] Writing a Simulink Device Driver block, http://www.mathworks.com

[4] Embedded Coder Getting Started Guide, http://www.mathworks.com

[5] Embedded Coder User's Guide, http://www.mathworks.com

[6] Embedded Coder Reference, http://www.mathworks.com

Annexe

LED_LPC1768_wrapper.cpp

```
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
# ifndef MATLAB_MEX_FILE
#include "mbed.h"
#endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
void LED_LPC1768_Outputs_wrapper(const boolean_T *led1,
                         const boolean_T *led2,
                         const boolean_T *led3,
                         const boolean_T *led4,
                         const real_T *xD)
{
# ifndef MATLAB_MEX_FILE
 DigitalOut myled1(LED1);
DigitalOut myled2(LED2);
DigitalOut myled3(LED3);
DigitalOut myled4(LED4);

if (led1[0] == 0) myled1=0;
else   myled1=1;
if (led2[0]==0) myled2=0;
else   myled2=1;
if (led3[0]==0)   myled3=0;
else   myled3=1;
if (led4[0]==0)    myled4=0;
else   myled4=1;
# endif
}
void LED_LPC1768_Update_wrapper(const boolean_T *led1,
                         const boolean_T *led2,
                         const boolean_T *led3,
                         const boolean_T *led4,
                         real_T *xD)
{
DigitalOut myled1(LED1);
DigitalOut myled2(LED2);
DigitalOut myled3(LED3);
DigitalOut myled4(LED4);
}
```

ADC_LPC_wrapper.cpp

```
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
# ifndef MATLAB_MEX_FILE
#include "mbed.h"
# endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define y_width 1

void ADC_LPC_Outputs_wrapper(real_T *V,
                             const real_T *xD)
{
# ifndef MATLAB_MEX_FILE
AnalogIn analog(p20);
float f=0;
f=3.3*analog.read();
V[0]=f;
# endif

}
void ADC_LPC_Update_wrapper(real_T *V,
                            real_T *xD)
{
}
```

CAN_Send_wrapper.cpp

```
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>

# ifndef MATLAB_MEX_FILE
#include "mbed.h"
#include "CAN.h"
#endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
void CAN_Send_Outputs_wrapper(const real_T *id,
                              const real_T *data,
                              const real_T *xD)
{
# ifndef MATLAB_MEX_FILE
DigitalOut led1(LED1);
DigitalOut led2(LED2);
CAN can2(p30, p29);
        char c[8];
        for(int i=0;i<8;i++)
        {
        *(c+i)=data[i];
        }
can2.frequency(1000000);
can2.write(CANMessage(id[0], c, 8));
      led1 = !led1;
# endif
void CAN_Send_Update_wrapper(const real_T *id,
                              const real_T *data,
                              real_T *xD)
{
}
```