Macro-opérations de communication

Noyau exécutif SynDEx pour le microcontrôleur PIC 18F2680

Table des matières

1. Objet du document

Les macro-opérations disponibles dans le module calcul pour le microcontrôleur PIC18F2680 ainsi que la chaîne de compilation utilisée sont décrites dans le document *main.pdf*.

Ce document décrit les macro-opérations définies dans le module de communication par bus CAN2.0A (Controller Area Network) spécifiques au microcontrôleur PIC18F2680. Il décrit aussi les procédures de chargement de code applicatif et de chronométrage.

Les définitions et abréviations utilisées dans ce document sont les mêmes que celles utilisées dans *main.pdf.*

2. Les macro-opérations

Cette section présente les opérations de communication définies dans le fichier *CAN.m4x*. L'encadré présente le nom de l'opération générique. Le texte de présentation traite de l'opération appelée par l'opération générique pour générer le code source spécifique au processeur cible. Le nom de l'opération spécifique est précisé entre parenthèses si nécessaire.

2.1. Configuration

```
{processorName}_ID
```

Le fichier *test.m4m* contient des paramètres de configuration pour chaque opérateur. Des macros constituées du nom de l'opérateur suffixé par « _ID » permettent d'attribuer les identifiants SynDEx de chaque opérateur.

Ces identifiants sont utilisés lors des phases de chargement de code et de collecte des chronométrages.

```
CAN_IDMSB
```

La macro CAN_IDMSB définie dans le fichier *CAN.m4x* représente les 8 octets de poids fort de l'identifiant CAN utilisé par le noyau d'exécutif. Les 3 bits restants sont attribués en utilisant la macro *lindex* (voir *syndex.m4x)*.

La valeur de la macro CAN_IDMSB doit impérativement correspondre à l'identifiant utilisé pour le chargement de code.

2.2. La séquence de communication

```
thread_(mediaType,mediaName {, procrName})
```

La macro *thread_* introduit une séquence de communication. Elle appelle successivement les macros :

- *CAN_shared_(mediaName,procrNames)* pour générer le code de l'automate de communication,
- *basicThread_(mediaName)* pour définir le point d'appel de la séquence de communication,
- *CAN_ini_(mediaName,procrNames)* pour initialiser le média de communication ainsi que l'automate de communication.

```
endthread_()
```

La macro *endthread_* termine une séquence de communication en appelant :

- *CAN_end_(mediaName,procrNames)* pour rétablir la configuration initiale du médium,
- *basicEndthread_(mediaName)* pour signaler la fin de la séquence de communication.

```
send_(bufferName, senderProcrType,senderProcrName {,receiverProcrName})
```

La macro *send_* appelle la macro *CAN_send_(1bufferName, 2senderType,3senderName {, receiverNames})* pour préparer une transmission de données vers un ou plusieurs opérateurs.

```
recv_(bufferName, senderProcrType,senderProcrName {,receiverProcrName})
```

La macro *recv_* appelle la macro *CAN_recv_(1bufferName, 2senderType,3senderName {, receiverNames})* pour préparer une réception de données.

```
sync_(dataType,size, senderProcrType,senderProcrName {,receiverProcrName})
```

La macro *sync_* appelle la macro *CAN_sync_(1bufferType, 2bufferSize)* pour synchroniser l'opérateur sur un échange de messages réalisé entre des opérateurs distincts.

2.3. Le « forward-loader »

```
loadFrom_(procrName {,destMedia})
loadDnto_(srceMedia {,procrName})
```

Le « forward-loader » appelle les macros *CAN_loadFrom_(procrName{,destMedia})* et *CAN_loadDnto_(srceMedia{,procrName})* pour terminer la procédure de chargement de code.

2.4. Le « backward-saver »

```
saveFrom_(srceMedia {,procrName})
saveUpto_(procrName {,destMedia})
```

Le « backward-saver » appelle les macros *CAN_saveFrom_(srceMedia {,procrName})* et *CAN_saveUpto_(procrName {,destMedia})* pour transférer les mesures effectuées par chaque opérateur vers l'hôte chargé de les sauvegarder.

3. Chargement du code applicatif

Après avoir été compilé par la chaîne de compilation décrite dans le document *main.pdf* et avant d'être exécuté, le code applicatif est chargé dans la mémoire programme de l'opérateur cible.

3.1. Le « bootloader »

Le « bootloader » est un programme inscrit dans la mémoire non-volatile de chaque opérateur et exécuté lors de son démarrage.

Il charge le code applicatif transmis par un processeur hôte en utilisant le bus CAN2.0A. Son fonctionnement est décrit dans le document « CAN Boot-Loader Specification » disponible sur le site www.syndex.org.

Le « bootloader » fixe l'identifiant (*CAN_IDMSB*) et la vitesse de transfert du medium CAN (*CAN_BRGCONx*).

Chaque opérateur est identifié de manière unique par un identifiant (*PROCID)*. Cet identifiant est défini en tête du fichier source du « bootloader ».

L'environnement MPLAB (www.microchip.com) ainsi qu'un programmeur sont nécessaires

pour inscrire le « bootloader » dans la mémoire programme de microcontrôleur.

3.2. Formatage du code objet

```
hex2sdx hexfile sdxfile
```

Le programme *hex2sdx* est utilisé pour traduire le code objet généré par l'assembleur de la chaîne de compilation, au format intel hex32, en code binaire compatible avec le chargeur de code décrit dans la spécification mentionnée précédemment.

Un compilateur C (Visual C++ 6.0 a été utilisé pour le développement) est nécessaire pour recompiler le programme *hex2sdx*.

3.3. Le « downloader »

```
dwnbin [CHRONOS]  FILE0 SID0 FILE1 SID1 ... FILEn SIDn
```

Le programme *dwnbin* est utilisé, en liaison avec le « bootloader » pour le chargement de code dans chaque opérateur par le bus CAN2.0A. Son fonctionnement est décrit dans le document « CAN Downloader Specification » disponible sur le site www.syndex.org.

Les paramètres, situés en tête du fichier source *dwnbin.c*, permettent respectivement de fixer les 8bits de poids fort de l'identifiant ainsi que la vitesse de transfert du médium CAN2.0A utilisé. Ces paramètres doivent impérativement correspondre à ceux utilisés pour le « bootloader ».

Un compilateur C (Visual C++ 6.0 a été utilisé pour le développement) est nécessaire pour recompiler le programme *dwnbin*.

Le programme *dwnbin* est compatible avec l'interface USB-CAN proposée par la société PEAK systems (www.peak-system.com/index_f.html).

4. Macros de chronométrage

Le chronométrage consiste à mesurer les performances temps réel de l'exécutif, il n'est pas nécessaire à son bon fonctionnement. Ces mesures permettent :

- de caractériser les macro-opérations de synchronisation et de communication du noyau générique d'exécutif,
- de caractériser chaque nouvelle macro-opération utilisée dans de nouveaux algorithmes,
- de vérifier pour chaque implantation d'un algorithme sur une architecture, que les performances prédites par SynDEx, calculées à partir des caractéristiques mesurées, correspondent à la réalité.

Le chronométrage s'effectue en quatre phases.

4.1. Initialisation

Chrono_ini_

Une première phase d'initialisation du tampon de stockage des mesures est effectuée par la macro *Chrono_ini_* **en fin de phase d'initialisations de la séquence de calcul**.

Chronos_

L'allocation du tampon est faite par la macro *Chronos_* (appelée **en fin de liste des macros d'allocation mémoire**) qui prend **en argument un entier spécifiant le nombre de mesures** que doit pouvoir mémoriser le tampon. Chaque mesure comprend deux entiers, l'un étant une étiquette identifiant le point de mesure (**entre deux étapes de la séquence de calcul**), l'autre étant une date lue sur l'horloge temps réel locale du processeur.

4.2. Mesure des dates

Chrono_lap_(label)

Une seconde phase, de mesure, indépendante sur chaque processeur, est effectuée pendant le déroulement de son programme. Des macros de chronométrage *Chrono_lap_* **sont insérées entre les étapes de la séquence de calcul**. Chaque point de mesure est identifié par une étiquette différente, passée en argument de la macro, qui est enregistrée, avec la date mesurée sur l'horloge temps-réel du processeur, dans un tampon.

CHRONOS_NBITERATIONS

Ce tampon est dimensionné de manière à retenir les mesures des quelques dernières itérations du programme. Le nombre d'itérations à retenir est défini par la macro *CHRONOS_NBITERATIONS.*

4.3. Mesure des décalages entre horloges

Cette phase consiste à mesurer les décalages entre les horloges des processeurs, afin de rendre comparable entre elles les dates mesurées sur des processeurs différents.

Le processeur racine (désigné comme processeur « main » dans le graphe de l'architecture) effectue une communication aller-retour avec chacun de ces descendants. En prenant la précaution durant cette phase d'éviter toute interférence entre calculs et communications, on peut considérer que la date de renvoi du message, mesurée par le descendant, correspond à la moyenne des dates d'émission et de réception du message, mesurées par l'ascendant. Le calcul est décrit dans le rapport *execv4*.

Le calcul du décalage entre horloges et la mise en correspondance des mesures effectuées sur les processeurs de l'architecture sont réalisés par le programme *dwnbin* utilisé pour le téléchargement du code applicatif.

4.4. Collecte des chronométrages

```
Chrono_end_
```

La collecte des chronométrages est effectuée après la terminaison des calculs par la macro *Chrono_end_* générée **en fin de phase de finalisation de chaque séquence de calcul** (après la (les) macro(s) *wait_endthread_*).

```
saveFrom_(srceMedia {,procrName})

saveUpto_(procrName {,destMedia})
```

Pour réaliser la collecte, la macro *Chrono_end_* fait appel aux macros *saveFrom_*, dans le cas du processeur racine, et *saveUpto_*, dans les autres cas. Ces macros sont générées en fin de chaque séquence de communication et prennent en arguments la description de l'arbre de couverture.

```
dwnbin [CHRONOS]  FILE0 SID0 FILE1 SID1 ... FILEn SIDn
```

Le programme *dwnbin* utilisé lors du téléchargement de l'applicatif dispose d'une option *CHRONOS* autorisant l'espionnage des échanges de messages réalisés par les macros *saveFrom_ et saveUpto_*. Cette option est activée (ou désactivée) dans le fichier *makefile* de l'application.

L'option CHRONOS provoque un blocage du programme *dwnbin* une fois la procédure de téléchargement de code applicatif terminée. Cette attente est signalée par le message d'avertissement : « Waiting for chronos upload »; elle se termine lorsque la macro *saveFrom_* signale le début de la procédure de téléchargement des chronométrages.

```
CAN_BSIDLSB
```

Pour signaler le début de la procédure de téléchargement des mesures de chronométrage, la macro *saveFrom_* utilise un identifiant réservé défini par la macro *CAN_BSIDLSB*.

Les mesures de chronométrages sont enregistrées dans le fichier *chronos.log*.

4.5. Exemple de macro-code

```
include(syndex.m4x)dnl
dnl
processor_(pic18f2680,p1,test,
SynDEx-6.8.5 (C) INRIA 2002-2006, 2007-01-22 12:06:08)

semaphores_(
  Semaphore_Thread_x,
  ping_o_p1_x_empty,
  ping_o_p1_x_full)

alloc_(short,ping_o,1)

Chronos_(2)

thread_(CAN,x,p1,p2,p3)
  loadDnto_(,p2,p3)
  loop_
    sync_(short,1,pic18f2680,p2,p3)
    Suc1_(ping_o_p1_x_empty,)
    recv_(ping_o,pic18f2680,p3,p1)
    Pre0_(ping_o_p1_x_full,)
  endloop_
  saveFrom_(,p2,p3)
endthread_

main_
  spawn_thread_(x)
  visu(ping_o)
  Chrono_ini_
  Pre1_(ping_o_p1_x_empty,x)
  loop_
    Suc0_(ping_o_p1_x_full,x)
    Chrono_lap_(100)
    visu(ping_o)
    Chrono_lap_(101)
    Pre1_(ping_o_p1_x_empty,x)
  endloop_
  visu(ping_o)
  wait_endthread_(Semaphore_Thread_x)
  Chrono_end_
endmain_

endprocessor_
```