CRO TP4

Programmation embarquée : *The Tissi-Emesspi Challenge*
(4h séance sur machines du département), 7 décembre 2023

1 Présentation du *Tissi-Emesspi challenge*

L'objectif de ce TP est d'approcher la programmation embarquée et de pratiquer la programmation bas niveau en C avec les clé *ez430*. Ce TP est organisé comme un challenge ou un hackaton : un message chiffré est envoyé par l'ez430 de l'enseignant (le serveur), vous devez le déchiffrer avec votre clé ez430 (le client). Ce déchiffrement a lieu en plusieurs étapes : le message déchiffré à l'étape i donnant des informations pour déchiffrer le message à l'étape $i+1$. Les groupes ayant terminé le déchiffrement doivent envoyer le message "`hello`" chiffré avec le code de la dernière étape (3eme clé trouvée). Ils sont alors identifiés par le serveur et enregistrés par l'enseignant au fur et à mesure.
Le programme qui vous est fourni contient les appels aux drivers nécessaires pour recevoir les messages envoyés par l'enseignant, mais vous devez faire le déchiffrement vous-même. Le principe est le suivant : les paquets font une taille fixe :`PKTLEN`=32 octets. Sur ces 32 octets, le premier octet (non chiffré) indique l'identifiant de l'émetteur (qu'il faudra changer pour pouvoir vous distinguer du voisin), le second octet contient un entier qui indique le système de chiffrement utilisé pour les 28 octets restants (le CC250 ajoute deux octets pour indiquer le RSSI et le LQI au message envoyé, il nous reste donc 28=30-2 octets). Le schéma d'un paquet est représenté sur la figure 1.

```
/*********************************************************/
/* Packet structure:                                     */
/* ----------------------------------------------------- */
/* | ID | coding_state | message(<=28bytes) | RSSI| LQI| */
/* ----------------------------------------------------- */
/*   0         1          2                          31  */
/*********************************************************/
```

FIGURE 1 – Structure des paquets utilisés par le protocole du TP : le premier octet contient votre identifiant (entre 0x11 et 0x30), le deuxième octet contient un entier qui indique avec quelle clé est codé votre message qui suit sur les 28 octets suivants.

Le chiffrement utilisé est simplement *un ou-exclusif bit à bit avec une valeur* (soit sur un octet, soit sur 4 octets), le déchiffrement est donc un ou-exclusif avec la même valeur. La valeur de ce masque pour l'étape $i+1$ est indiquée dans le message que vous avez décodé à l'étape i. Lors de la dernière étape, vous devez vous-même envoyer un paquet chiffré d'une manière spécifique commençant par "`hello`" (Attention : SANS majuscule au début). Ce protocole est représenté sur la figure 2.

2 Découverte de la carte ez430 (Rappels, déjà vu en ARC)

Pour chaque binôme, allez prendre le matériel nécessaire au TP : dans chaque boîte, vous trouverez un genre de clé USB qui ressemble au schéma de la figure 3. Ne le branchez pas tout de suite.

Comme tout objet technologique, notre plate-forme de TP s'accompagne d'une documentation technique abondante. Pour ne pas vous noyer sous la doc, nous vous en avons copié les extraits essentiels directement dans le sujet, sous forme d'encadrés essentiellement en fin de sujet. Pour les plus curieux, nous vous avons aussi mis à disposition les documents sur Moodle :

ez430.pdf décrit notre carte d'expérimentation et les différents composants présents sur la carte.

MSP430.pdf est le manuel générique de la famille MSP430. Le processeur est documenté au chapitre 3 de ce document.

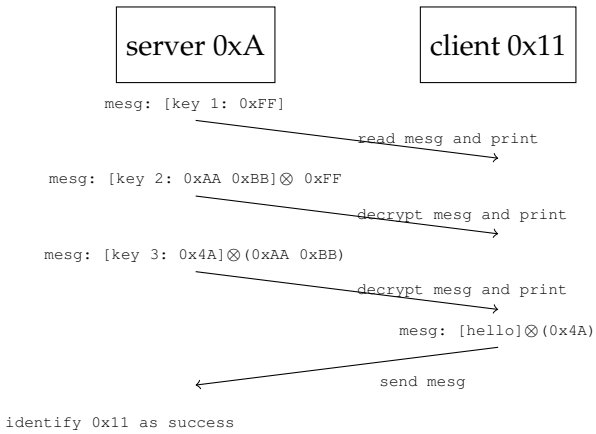


FIGURE 2 – Les différents échanges entre le serveur (prof, identifiant Ox0A) et le client (étudiant, identifiant 0x11 ici). Les trois messages sont envoyés en boucle par le serveur (les clés sur la figure ne sont pas les bonnes bien sûr). Une fois que la troisième clé est identifiée, le client envoie un message "hello" chiffré avec la 3eme clé.

Extrait de la documentation : ez430.pdf page 5

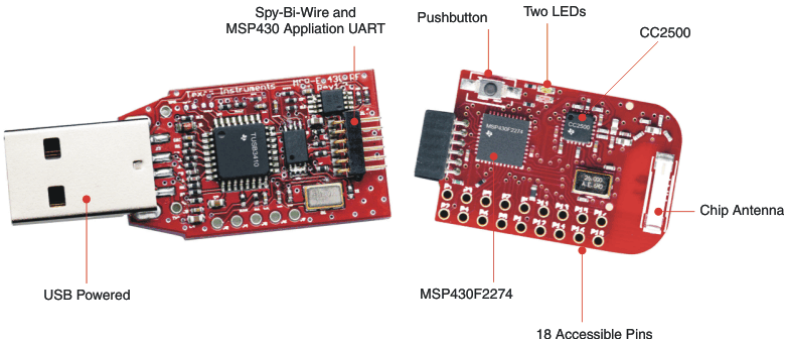


FIGURE 3 – La clé ez430 avec le micro-contrôleur MSP430

msp430F22x4.pdf donne les détails techniques de notre modèle précis de MSP430.

3 Installation des outils : chaîne de compilation (msp430-elf-gcc) et msp-debug

Pour vérifier que vous avez installé correctement la chaîne de compilation `mspgcc`, vous tapez la commande :

```
msp430-elf-gcc -v
```

Attention : nouvelle chaîne de compilation `gcc` v8

Depuis 2023 nous utilisons une nouvelle chaîne de compilation (`gcc v8`) fournie par Texas Instrument. La manipulation du `printf` est un peu délicate, donc à chaque fois que vous utilisez un `printf` bien mettre `\r\n` (dans cet ordre) à la fin de la chaîne que vous affichez, sinon il restera dans le buffer de sortie jusqu'au prochain `printf`.

Si la commande n'est pas connue, activez la *toolchain* comme indiqué ci-dessous. A priori le TP est à exécuter sur les machines du département ou la chaîne de compilation est installée. En annexe A, vous trouverez les instruction pour installer la chaîne sur votre machine Ubuntu si vous le souhaitez. Une fois la *toolchain* activée, il faut encore télécharger les sources des programmes que l'on va utiliser (section 3.3).

3.1 Activation de la chaîne de compilation sur les machines du département

La programmation du eZ430-R2500 se fait en C, nous utiliserons le compilateur `msp430-elf-gcc` qui est installé sur les machines du département. la DSI a déployé la chaîne de compilation msp430 dans le répertoire `/opt/msp430-2023`. Pour pouvoir l'exécuter il faut copier le contenu du fichier `/opt/msp430-2023/env.sh` dans votre `.bashrc`, cela va mettre à jour les variables `$PATH`, `$MSPINCLUDES`, etc. Voici le contenu du fichier `env.sh` que vous trouverez aussi sur Moodle, copiez le dans votre `.bashrc`, vous comprenez ce code ?
Enfin, lancer une nouvelle fenêtre shell et tapez la commande :
`go_mspgcc`
Cela modifie votre invite (en rajoutant `[MSPGCC8]` devant) et vous avez maintenant accès à la commande `msp430-elf-gcc`

```
# -*-sh-*-

function go_mspgcc()
{
    # use your own mspgcc
    export MSPHOME=${HOME}/tools/msp430/
    export MSP430DIR=${MSPHOME}/msp430-gcc-8.3.1.25_linux64
    export MSPINCLUDES=${MSPHOME}/msp430-gcc-support-files/include
    export MSPLDLIBS=${MSPINCLUDES}
    export MSP430_GCC_INCLUDE_DIR=${MSPINCLUDES}

    export MSPGCCDIR=${MSP430DIR}
    export MSP430PREFIX=msp430-elf
    export CC=${MSP430PREFIX}-gcc
    export LD=${MSP430PREFIX}-ld
    export MSPFLAGS=-mhwmult=none

    export PATH=${PATH}:${MSPGCCDIR}/bin
    export PS1='[MSPGCC8]'${PS1}
}
```

Code à copier dans le fichier `$HOME/.bashrc`

3.2 Activation de `mspdebug`

La chaîne de compilation configurée ci-dessus permet de générer un binaire au format compatible avec le MSP430. L'outil `mspdebug` sert, lui, à télécharger le code binaire sur la carte elle-même (en passant par le port USB grâce à un protocole JTAG).
Tester la bonne installation de `mspdebug` :

— Branchez sur un port USB la carte ez430.

— Lancez la commande `mspdebug rf2500`

— Si tout se passe bien, vous vous retrouvez avec une invite :

```
(msp-debug)
```

3.3 Récupération et compilation des sources utiles au challenge

Les sources des programmes que nous allons charger sur la clef sont récupérables via Moodle (page du cours CRO). Récupérez sur Moddle l'archive `TCMSP-init.tar` et extrayez ce fichier dans le répertoire de votre choix. l'extraction crée un répertoire nommé `code` qui contient trois sous-répertoires

— `ez430-drivers` qui contient tous les codes des pilotes (*drivers*) pour les périphériques du micro-controleur (UART, LEDs, boutons etc), ainsi que des exemples pour faire fonctionner ces périphériques.

— `protothreads` qui contient un libraire de *protothread* dont nous ne serviront pas.

— `TP4-student-init` qui contient le code que vous allez modifier pour le challenge.

QUESTION 1 ► Faites `make` dans repertoire `code`, normalement cela lance un `make` dans les sous répertoires sans échouer (assurez vous que vous avez bien exécute `go_mspgcc` avant).

3.4 Programmation de la clé ez430 et utilisation de `minicom`

QUESTION 2 ► Allez dans le répertoire `TP4-student-init`, tapez `make` (normalement cela est déjà fait donc il n'y a rien à faire).

QUESTION 3 ► téléchargez ce binaire sur votre clé ez430 (`make download`), et lancez un terminal série pour écouter `/dev/ttyACM0`, par exemple `minicom` :
`minicom -D /dev/ttyACM0`
Au bout de quelques secondes, vous devez voir apparaître une ligne ressemblant à cela :

```
sent:     0A 00 68 65 6C 6C 6F 2C 20 73 63 72 61 6D 62 6C 65 72 20 31 3A 20 4F 78
in ascii: hello from ID: 0x0A
```

Il s'agit du paquet envoyé par votre client, vous êtes maintenant en position de comprendre le fonctionnement du programme `src/main.c`.

4 Comprendre ce qu'il se passe

Maintenant que votre chaîne fonctionne, prenons un peu de temps pour expliquer les différentes composants qui sont mis en œuvre. Ces composants se retrouvent presque toujours en programmation de systèmes embarqué.

4.1 Cross-compilation et chargement du programme sur le MSP430

Si l'on se place dans le répertoire `TP4-student-init`, la commande `make` va simplement compiler le programme qui est dans le fichier `src/main.c` et placer l'exécutable dans le fichier `bin/serial.elf`. Cependant le compilateur utilisé n'est pas `gcc` mais `msp430-elf-gcc` qui est une version de `gcc` qui génère du code *au format elf du MSP430* (et non pas au format `elf` du processeur Intel de la machine sur laquelle vous travaillez). On appelle cela la *cross-compilation* : générer un code au format d'une autre machine que celle ou on travaille.
les commandes (simplifiées) qui sont exécutées quand vous lancez la commande `make` sont les suivantes (le `-lez430` correspond à l'utilisation de la bibliothèque libez430.a compilée dans le répertoire `ez430-drivers`) :

```
msp430-elf-gcc -g -Wall -mmcu=msp430f2274  -c src/main.c -o .obj/main.o
msp430-elf-gcc -mmcu=msp430f2274 .obj/main.o -static  -lez430 -o bin/serial.elf
```

Une fois le fichier `bin/serial.elf` généré, il faut encore le charger sur le MSP430 de la clé ez430. Pour cela on utilise traditionnellement le protocole JTAG qui permet de charger les binaires sur des cibles embarqué. Il existe un port physique spécifique pour le protocole JTAG mais on peut aussi encapsuler le protocole JTAG dans le protocole USB et donc charger le code du MSP430 en utilisant la connexion USB entre le PC et le MSP430, c'est ce qui est fait dans l'outil `mspdebug`. Ces actions sont illustrées sur la figure 4-(a).
Une fois le programme chargé sur le MSP430, il démarre automatiquement. La clé ez430 à la possibilité de communiquer par radio (cf Annexe C) mais aussi directement avec le PC grâce à la liaison série (ou UART) qui passe par le port USB. Le protocole UART, mieux expliqué à la section suivante, permet d'échanger des octets dans les deux sens entre le PC et le MSP430. Cette communication est illustrée en figure 4-(b).

QUESTION 4 ► Analysez la commande exécuté lorsque vous faites `make download` (faites `make download -n`). Demandez à l'enseignant si vous ne comprenez pas

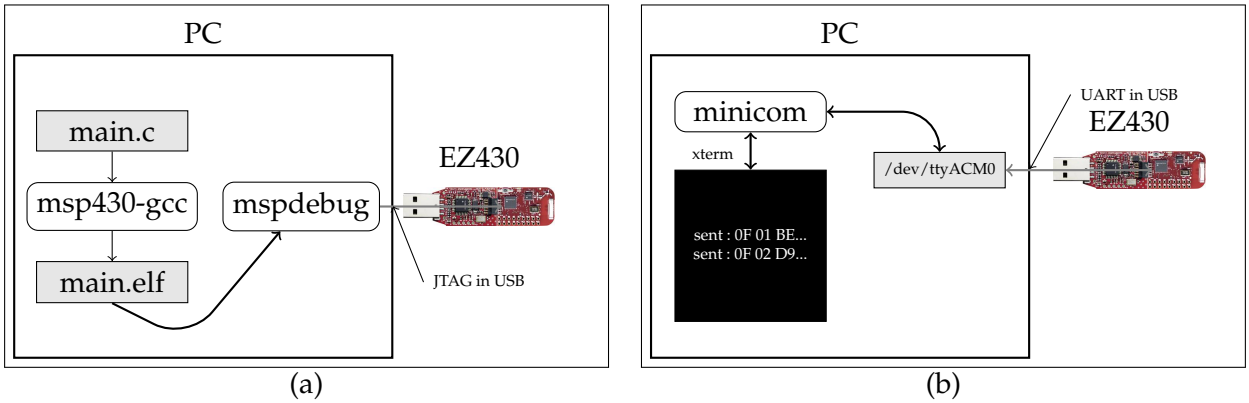


FIGURE 4 – (a) Cross-compilation et chargement du binaire sur le MSP430 via le protocol JTAG dans USB, et (b) communication série (ou UART) entre le PC et le MSP430 à travers le device linux `/dev/ttyACM0`.

4.2 Exécution et visualisation des information du port série

UART : le port série et `minicom`

L'UART ou *port série* est utilisé pour transmettre des informations entre la clé EZ430 et l'ordinateur hôte. Lorsque l'on branche la clé sur le port USB, un port (`COM4` sous windows, `ttyACM0` sur linux) apparaît sur la machine hôte. Le MSP pourra alors envoyer des informations en utilisant, par exemple, la fonction `printf`.
En général la communication depuis un PC vers par le port série se fait par un logiciel dédié (`minicom` pour linux, expliqué en annexe, ou `hyperterminal` pour Windows). Minicom est un programme de contrôle de modem et d'émulation de terminal pour les OS Unix-like, écrit par Miquel van Smoorenburg d'après le populaire Telix pour MS-DOS. Nous allons l'utiliser pour communiquer avec le port série créé par la connection du cable USB-série sur le device `/dev/ttyACM0`
Nous avons juste à configurer le *device* qui est `/dev/ttyACM0` (le débit du port série peut être configuré aussi, mais il s'adapte en général). On donc soit lancer minicom avec les bon paramêtres :
`minicom -D /dev/ttyACM0`
Pour sortir de minicom, on tape `Ctrl-a q`. Pour rentrer en mode configuration taper `Ctrl-a z` puis `O` pour configurer minicom[a]. Normalement pour ce TP vous n'aurez pas besoin d'arréter minicom, lorsque l'on reprogrammera le MSP, la connection sera retrouvée automatiquement.

a. Selectionner "configuration du port série", choisir la lettre (A et E) pour configurer le device et le débit (9600 8N1 par exemple)

5 Prise en main du programme fourni

Ouvrez le programme `TP4-student-init/src/main.c` dans un éditeur.

QUESTION 5 ▶ Trouvez le code de la fonction `main`. la fonction `main` effectue les action suivantes, identifiez les lignes de code correspondantes :

— Arrêt du watchdog timer.
— Réglage des horloges utilisées sur le MSP430
— Initialisation du TimerA pour envoyer des interruption toute les 100 ms
— Initialisation du port série (ou UART)
— Mise ne place du *handler d'interruption* de l'UART (expliqué plus loin)

— Initialisation du périphérique SPI
— Initialisation du périphérique radio (CC2500)
— Autorisation des interruptions
— lancement de la fonction `periodic_send()`

QUESTION 6 ► Trouvez la définition de la fonction `periodic_send`. Comprenez vous ?

QUESTION 7 ► Regardez la fonction `radio_send_message`, comprenez son fonctionnement (on suppose pour l'instant que la fonction `cc2500_utx` envoie le message donné en argument par radio).

QUESTION 8 ► Regardez la fonction `encode_message`, comprenez son fonctionnement.

QUESTION 9 ► Lisez l'encart ci-dessous sur les *timer* et *callback*.

Timers et callback

Le MSP430 possède plusieurs *timers*. Un timer est un mécanisme hardware que l'on va configurer pour "compter le temps" et envoyer des interruptions au bout d'un certain temps. Ici Le timer est configuré pour envoyer une interruption toute les 10ms :

```
#define TIMER_PERIOD_MS 10
[...]
timerA_start_milliseconds(TIMER_PERIOD_MS);
```

Lorsque le MSP430 reçoit l'interruption du timer, il interrompt l'exécution du programme en cours, exécute le *handler d'interruption* (quelque fois appelé *callback*) et revient ensuite à l'exécution du programme en cours.
Ici on autorise au programmeur à définir lui même le callback, c'est à dire à définir la fonction qui sera exécutée sur une réception de l'interruption du timer :

```
timerA_register_cb(&timer_tick_cb);
```

Le timer callback (fonction `timer_tick_cb()`) va incrémenter, à chaque interruption des compteurs appelés `timer[i], i=0,NUM_TIMER` et permet ainsi d'avoir plusieurs timer *software* à partir d'un seul timer *hardware*. Ici on n'utilise qu'un seul timer software (`NUM_TIMER`=1) qui s'appele `TIMER_RADIO_SEND` :

```
#define TIMER_RADIO_SEND timer[0]
```

Vous pouvez maintenant comprendre globalement le comportement (en emission de paquet) de votre ez430 : il envoie par radio toute les 10 secondes (1000× 10ms = 10s), un paquet non chiffré contenant la chaîne `"hello world"` . Au moment d'envoyer ce message, il l'affiche sur le port série (donc nous on le voit dans Minicom) et il repasse ensuite en mode réception de message.

QUESTION 10 ► Comme le timer, la radio (qui est un autre chip matériel qui est un peu expliqué en annexe C) va envoyer une interruption au MSP430 lorsqu'elle reçoit un paquet. Comprenez vous la ligne 334 de `main.c:cc2500_rx_register_cb(radio_cb);` ? Analysez la fonction `radio_cb`.

QUESTION 11 ► Dernière chose : allez voir la fonction `dump_message` et vérifier que vous la comprenez bien, demandez des explications à l'enseignant le cas échéant.
Vous êtes maintenant en position pour commencer le challenge.

6 Début du challenge proprement : travail à faire

QUESTION 12 ► **Changez votre identifiant** (macro `ID`) et mettez le à la valeur : 16+N ou N est le numéro de votre machine en salle TC (exemple : pour la machine `TC405-109-03`, N vaut 3, dont ID vaut 19 soit 0x13). Vérifiez que la valeur est effectivement prise en compte quand vous envoyez un paquet, faites valider par un enseignant.

QUESTION 13 ▶ Votre port série affiche tout ce qu'il reçoit, notamment les message des autres clients qui n'ont pas d'intérêt pour vous. Filtrez l'affichage des paquets reçus, de manière à n'afficher que les paquets envoyé par l'ID 0x0A (i.e. le serveur).
Vous avez maintenant compris le fonctionnement de votre programme initial, il faut décoder les trois phrases, avec trois niveaux de codage (`coding_state`) différents. Puis envoyer le message `hello` avec le bon niveau de codage.

QUESTION 14 ▶ Peuplez les fonctions `decode_message` et `encode_message` pour pouvoir décoder les message reçu et encoder les message envoyé comme vous le voulez.
On rappelle ici le principe de codage expliqué au début du sujet :
Le codage utilisé est simplement un ou-exclusif avec une valeur (soit sur un octet, soit sur 4 octets), le décodage est donc un ou-exclusif avec la même valeur. La valeur de ce masque pour l'étape $i + 1$ est indiqué dans le message que vous avez décodé à l'étape i. Lors de la dernière étape, vous devez vous-même envoyer un paquet chiffré avec la clé 3 par `"hello"` *(Attention : SANS majuscule au début).*
N'oubliez pas, que lorsque vous envoyez un message chiffré avec la clé 3 d'indiquer (dans le deuxième octet) que vous utilisez le *coding_state* 3. Pour cela **il faut changer la variable coding_state dans votre main.c**

7 Pour aller plus loin

Pour ceux qui ont finit de décoder rapidement, identifier (avec son ID) un autre groupe ayant terminé et mettez en place un Chat chiffré en utilisant le port série en entrée et en sortie.

A Annexe : Installation de la chaîne de compilation `msp430-elf-gcc` sur Linux

A.1 A partir des binaire présents sur Moodle

:

- Récuperez l'archive sur Moodle (lien "toolchain mspgcc v8")
- Choisissez un répertoire, dans votre home ou dans `/etc/local` si vous avez les droits admin
- `tar xvf msp430-2023-Insa.tar`
- Déscendez dans le répértoire `msp430-2023` et préparez l'environnement en copiant le contenu du fichier `env.sh` dans votre `.bashrc` en mettant à jour la variable `$MSPHOME` avec la valeur du répertoire ou vous avez extrait la toolchain.
- Lancer une nouvelle fenêtre shell et tapez `go_mspgcc`.

A.2 À partir des paquets Ubuntu

Les paquets Ubuntu ne sont pas mis a jours malheureusement, il faut juste récuperer le paquet `mspdebug`.

```
sudo apt-get install mspdebug
```

B Annexe : configuration du port série sous linux avec minicom

Minicom est un programme de contrôle de modem et d'émulation de terminal pour les OS Unix-like, écrit par Miquel van Smoorenburg d'après le populaire Telix pour MS-DOS. Minicom apporte une émulation totale ANSI et VT100, un langage de script externe, et d'autres choses encore. Nous allons l'utiliser pour communiquer avec le port série créé par la connection du cable USB-série sur le device `/tty/USB0`

Il faut configure le débit du port série (9600 bauds) et le device linux correspondant au port série (`/dev/ttyUSB0`). On peut soit lancer minicom avec les bon paramêtres :

`minicom -b 9600 -D /dev/ttyUSB0`

Soit lancer minicom en mode setup pour le configure manuellement (souvent necessaire) : `minicom -s`

Selectionner "configuration du port série", choisir la lettre (A et E) pour configurer le device et le débit (9600 8N1). Une fois en mode de fonctionnement normal, vous voyez dans le terminal ce qui arrive sur le port série et ce que vous tapez au clavier est envoyé sur le port série (caractères ascii). Pour rentrer en mode configuration taper `Ctrl-a z` puis `O` pour configurer minicom. Enfin `Ctrl-a q` pour sortir

C Communication radio grace au CC2500

le CC2500

- Le CC2500 est un transceiver low power destiné à la bande 2.4-GHz (ISM : Industrial , Scientific and Medical), SRD (short range device) : 2400-2483.5 MHz
- Il n'a pas de couche MAC/PHY directement intégrée mais il est très paramétrable :
  - plusieurs modes de communication avec divers débits (jusqu'à 500 kbps),
  - plusieurs modulations,
  - optionnellement des codes correcteurs d'erreurs.
- Il est contrôlé par le MSP grâce à une liaison SPI entre les deux chips (CC2500 et MSP430).
- Le protocole SPI est un autre protocole série utilisé dans l'embarqué, contrairement au protocole UART, il est *synchrone*, c'est à dire qu'un fil transporte l'horloge. C'est un protocole ou il y a un maître est un (ou plusieurs) esclaves, il utilise 3 ou 4 fils (SIMO signifiant *Slave In, Master Out*)
  - P3.0 : STE (RF_STE, inutile si il y a un seul esclave)
  - P3.1 : SIMO (RF_MOSI)
  - P3.2 : SOMI (RF_MISO)
  - P3.3 : CLK (RF_SPI_CLK)
- Il y a aussi deux autres fils connectant le CC2500 aux GPIO du MSP430 : GDO0 et GDO2, qui permettent de transmettre des interuptions.
- La reception d'un paquet déclenchera une interruption qui sera traité par le callback que nous déclareront, ici pour notre TP nous avons :

  ```
  cc2500_rx_register_cb(radio_cb);
  ```

  C'est donc la fonction `radio_cb` qui sera appelée à chaque paquet reçus.

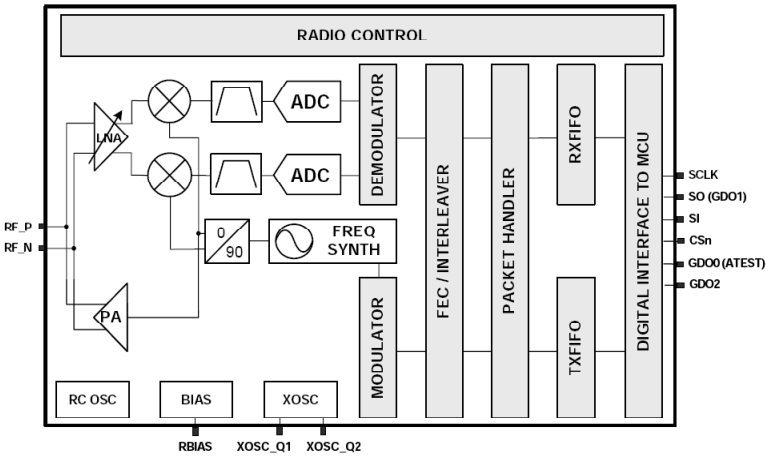


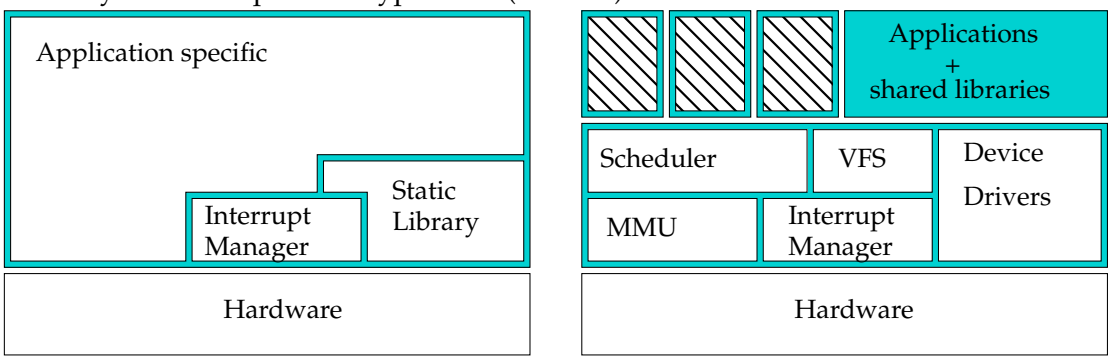
Schéma du CC2500

La configuration du CC2500 est relativement complexe, nous ne la modifieront pas dans ce TP, elle est efféctuée dans le driver (`src/ez430-drivers/src/cc2500.c`), ou elle est stockée dans la variable `rfSettings_default_config`. La compréhension de ce code nécessite de lire documentation du CC2500 disponible sur Moodle ou Internet.

D Multi-tâche embarqué : notion de tâche et protothread

OS embarqués

L'objet correspondant à la notion de "système d'exploitation" est vaste, elle peut aller d'une d'une bibliothèque spécifique pour une application à un système générique type Unix.
En embarqué, les applications sans système d'exploitation (*bare métal* : un simple `main()`, telles les applications que nous avons vu jusqu'à présent sur l'ez430) représentent une part importante des systèmes déployés aujourd'hui.
Ci dessous nous avons représenté l'empreinte mémoire d'un système sans OS (à gauche) et celle d'un système complexe de type linux (à droite).

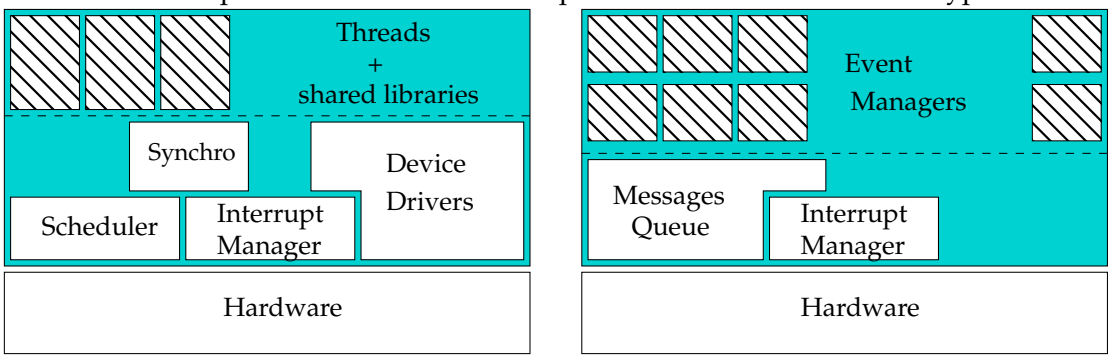


No OS (bare metal) Linux-type OS

Il existe tout de même deux grandes catégories d'OS embarqués :

- Ceux basés sur le modèle "Event driven"
  - Les événements matériels démarrent des fonctions qui s'exécutent sans interruption (*run to completion*).
  - Les changements de contexte, la gestion de pile, l'ordonnancement et la gestion de priorité sont simplifiés.
  - Exemples : TinyOS 1 & 2
- Ceux basés sur le modèle "Thread"
  - Proche du modèle de programmation classique.
  - Mémoire partagée, piles séparées.
  - Changement de contexte.
  - Exemples : FreeRTOS, Contiki

Nous avons représenté ci-dessous les empreinte mémoire de ces deux types d'OS



thread based OS event driven OS

Les protothread ont été proposés par Adam Dundels (auteur de l'OS embarqué Contiki, de la pile μIP notamment) comme un modèle de tâche avec une pile partagée (modèle de *co-routine*).