

Ajout de commandes AT

Sommaire

1	Objectif	5
2	Prérequis.....	5
3	Implémentation d'une commande AT	5
3.1.	Définition des commandes AT.....	5
3.2.	Syntaxe d'une commande AT	5
3.3.	Structure AT	6
3.4.	Exemple d'intégration d'une commande AT	8
3.5.	Définir une macro dans le fichier at.h	8
3.6.	Fichier at.c.....	10

Liste des Figures

Figure 1 : Structure définissant une commande AT	7
Figure 2 : Tableau de toutes les commandes AT supporté.....	7
Figure 3 : Implémentation d'une macro dans le fichier at.h	9
Figure 4 : déclaration des en-têtes de fonction pour les commandes get et set	10
Figure 5 : Exemple de corps de fonction pour les commandes get, set	11

Glossaire

CR : « En C, un retour chariot (Carriage Return) est représenté par la séquence d'échappement '\r'. C'est un caractère spécial qui, lorsqu'il est rencontré par le compilateur, fait passer le curseur au début de la ligne courante sur l'écran, écrasant tout ce qui se trouve à droite de cette position. Il est souvent utilisé conjointement avec '\n' pour créer des retours chariots et des sauts de ligne. »

Fichier .c : « Un fichier .c est un fichier de code source écrit dans le langage de programmation C. Il contient des fonctions, des variables et d'autres constructions utilisées pour créer un programme qui peut être compilé en code machine pour l'exécution sur un ordinateur. Le fichier .c est généralement édité à l'aide d'un éditeur de texte ou d'un environnement de développement intégré (IDE) puis compilé à l'aide d'un compilateur C, tel que GCC ou Clang, pour générer un fichier exécutable. »

Fichier .h : « Un fichier .h est un fichier d'en-tête en C. Il contient généralement des déclarations de variables, de fonctions et de structures de données utilisées par d'autres fichiers source dans un projet. Les fichiers d'en-tête sont généralement inclus dans les fichiers source C à l'aide de l'instruction #include. Les fichiers d'en-tête permettent de regrouper les déclarations communes dans un seul emplacement pour une utilisation répétée dans plusieurs fichiers source, ce qui facilite la maintenance et la modification du code.»

LF : « En C, une nouvelle ligne (Line Feed) est représentée par la séquence d'échappement '\n'. C'est un caractère spécial qui, lorsqu'il est rencontré par le compilateur, fait passer le curseur à la ligne suivante à l'écran.

Pointeur : « En C, un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. Il permet de manipuler directement la mémoire et de stocker des adresses de variables. On déclare un pointeur en utilisant l'opérateur de résolution de l'adresse (*) suivi du type de la variable pointée.

1 Objectif

L'objectif de ce document est d'enseigner à l'utilisateur comment intégrer des commandes AT personnalisées

2 Prérequis

Ce document s'appuie fortement sur l'utilisation du langage C. Ainsi, une connaissance de base de C est un prérequis supposé.

3 Implémentation d'une commande AT

3.1. Définition des commandes AT

Les commandes AT sont un ensemble d'instructions utilisées pour contrôler le Dragino, où AT est l'abréviation de "attention". Les commandes AT tirent leur nom du fait que la syntaxe de chaque commande émise commence par "AT".

3.2. Syntaxe d'une commande AT

```
AT+<COMMANDE><SUFFIXE><DONNÉES>
```

Les commandes AT peuvent être classées en 4 catégories avec différents suffixes.

Type	Utilisation	Suffixe
Tester/Aider	Fournit une brève description de la commande utilisée	?
Lire (<i>get</i>)	Utilisé pour obtenir la valeur d'une commande donnée	=?
Mise en place (<i>set</i>)	Utilisé pour fournir une valeur à une commande	=
Exécuter (<i>run</i>)	Utilisé pour exécuter une commande	Rien

Tableau 1 : Commandes AT suffixes disponibles

N.B :

L'argument <DONNÉES> ne doit être saisi que dans le cas d'un suffixe “=” . L'argument <DONNÉES> doit être omis sinon.

Chaque commande renvoie une chaîne d'état, qui est précédée et suivie de <CR><LF> dans un format “<CR><LF><Statut>”. Les statuts possibles sont :

- OK : la commande s'exécute correctement sans erreur.
- AT_ERROR : erreur générique
- AT_PARAM_ERROR : un paramètre de la commande est erroné

- `AT_BUSY_ERROR` : le réseau LoRa® est occupé, la commande n'a donc pas pu se terminer
- `AT_TEST_PARAM_OVERFLOW` : le paramètre est trop long
- `AT_NO_NETWORK_JOINED` : le réseau LoRa® n'a pas encore été rejoint
- `AT_RX_ERROR` : détection d'erreur lors de la réception de la commande

Les fichiers concernés par l'intégrations de commandes AT sont :

- `command.c` : contient le pilote de commande principal (driver) dédié aux commandes AT
- `at.c` : contient les API des commandes AT
- `at.h` : En-tête (header) du module pilote `at.c`

3.3. Structure AT

Chaque commande AT est une structure (struct) de type `ATCommand_s`. La struct AT est composée de 2 pointeurs, 3 pointeurs de fonction, et une variable.

1. Un pointeur de caractère en lecture (`string`) qui contient la commande après le "AT".
2. Un entier (`size_string`) qui contient la taille de la commande string, sans compter le caractère nul.
3. Trois pointeurs de fonction (`get`, `set`, `run`). Chacun de ces pointeurs de fonction prend un pointeur de caractère en lecture et renvoie un type `ATError_t` :
 - a. `get` : utilisé pour l'implémentation de lecture qui utilise le suffixe « `=?` ».
 - b. `set` : utilisé pour l'implémentation d'écriture qui utilise le suffixe « `=` ».
 - c. `run` : utilisé pour l'implémentation d'exécution qui n'utilise aucun suffixe.

Le type `ATError_t` renvoie un statut parmi la liste des statuts possibles mentionnée ci-dessus.

4. Un pointeur de caractère en lecture (`help_string`) utilisé pour imprimer une brève description de la commande AT. Ceci est utilisé dans l'implémentation du test/help auquel on accède par le suffixe « `?` »

La figure 1 montre l'implémentation de la struct `ATCommand_s` dans le fichier `command.c` :

```

command.c
58
59 /* Private typedef ----- */
60 /**
61  * @brief Structure defining an AT Command
62  */
63 struct ATCommand_s {
64     const char *string;           /*< command string, after the "AT" */
65     const int size_string;        /*< size of the command string, not including the final \0 */
66     ATError_t (*get)(const char *param); /*< =? after the string to get the current value*/
67     ATError_t (*set)(const char *param); /*< = (but not =? \0) after the string to set a value */
68     ATError_t (*run)(const char *param); /*< \0 after the string - run the command */
69 #if !defined(NO_HELP)
70     const char *help_string;      /*< to be printed when ? after the string */
71 #endif
72 };
73

```

Figure 1 : Structure définissant une commande AT

De plus, il existe un tableau de type `ATCommand_s` qui va affecter l'implémentation des variables et des pointeurs définis dans chaque struct `ATCommand_s`. La figure 2 montre la mise en œuvre de cette implémentation :

```

command.c
96
97 /**
98  * @brief Array of all supported AT Commands
99  */
100 static const struct ATCommand_s ATCommand[] =
101 {
102     {
103         .string = AT_SEN_CNT,
104         .size_string = sizeof(AT_SEN_CNT) - 1,
105 #ifndef NO_HELP
106         .help_string = "AT" AT_SEN_CNT ": Gets or set the number of sensors\r\n",
107 #endif
108         .get = at_sencnt_get,
109         .set = at_sencnt_set,
110         .run = at_return_error,
111     },
112
113     {
114         .string = AT_DEBUG,
115         .size_string = sizeof(AT_DEBUG) - 1,
116 #ifndef NO_HELP
117         .help_string = "AT" AT_DEBUG ":Set more info input\r\n",
118 #endif
119         .get = at_return_error,
120         .set = at_return_error,
121         .run = at_DEBUG_run,
122     },
123

```

Figure 2 : Tableau de toutes les commandes AT supporté

La ligne 100 représente la déclaration du tableau de type `ATCommand_s`. La partie encadrée en vert a une implémentation existante d'une commande AT, et la partie encadrée rouge montre un exemple de l'intégration de chaque membre d'une struct `ATCommand_s` dans ce tableau.

3.4. Exemple d'intégration d'une commande AT

Prenons la partie encadrée en rouge. Cette partie représente une commande AT qui était ajoutée. Cette commande n'est pas disponible dans le code source de base. Cette partie représente l'intégration une commande AT qui peut lire ou attribuer le nombre de sondes HYT939. Le but est d'apprendre à implémenter cette commande AT. Dans cet exemple, les implémentations de cette commande sont :

- AT+SENCNT=?
- AT+SENCNT=X

Selon la structure du code source de base du Dragino, la partie qui vient après les caractères "AT" est affectée au membre `.string`. Dans ce cas, le membre `.string` doit contenir comme contenu `<" +SENCNT">`. Cependant, pour rendre le code plus structuré et plus facile à manipuler, une macro `AT_SENCNT` sera utilisée ultérieurement pour représenter le contenu `<" +SENCNT">`.

La valeur de `sizeof(AT_SENCNT)` moins 1 est affectée au membre `.size_string`.

Une directive `#ifndef` avec la macro `NO_HELP` devait être ajoutée autour du membre `.help_string` pour lui attribuer la phrase de description de l'aide. Le `.help_string` doit être attribué en utilisant la syntaxe suivante

```
.help_string = "AT"<MACRO de la commande AT> <brève description de la commande AT><CR><LF>
```

Chacun des pointeurs de fonction `get`, `set`, `run` doit respectivement pointer vers la fonction qui implémente sa fonctionnalité. Ces fonctions seront définies ultérieurement dans le fichier source `at.c`. Dans cet exemple, le membre de pointeur de fonction `.get` pointe vers la fonction `at_sencnt_get` et le membre de pointeur de fonction `.set` pointe vers la fonction `at_sencnt_set`.

Cependant, étant donné que cette commande AT n'a pas d'exécution (`run`) use , le membre du pointeur de fonction `.run` doit renvoyer un état `AT_ERROR`. Il existe une fonction dans le code source appelée `at_return_error` qui fait cela.

3.5. Définir une macro dans le fichier at.h

Pour continuer avec l'exemple mentionné, la syntaxe suivante doit être utilisée pour définir une macro :

```
#define NOM_MACRO valeur
```

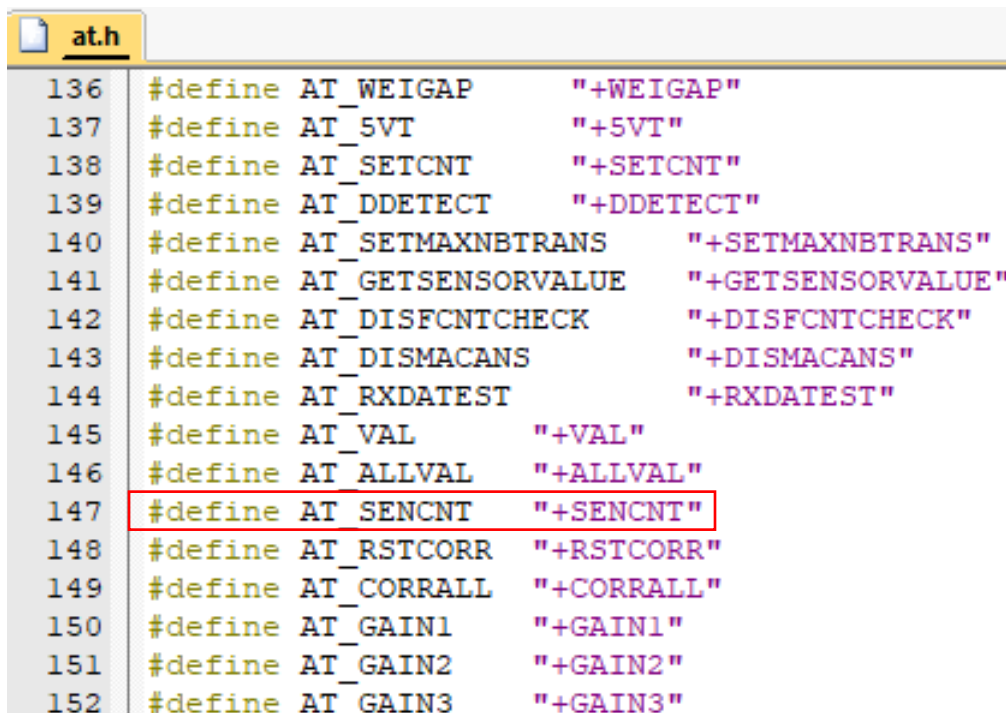

NOM_MACRO

Elle représente le nom de la macro (AT_SENCNT dans cet exemple).

valeur

Elle représente une valeur constante. Dans ce cas, la valeur de la constante représentée est la chaîne "+SENCNT".

Sur la figure 3, la partie encadrée sur la ligne 147 montre la mise en œuvre de la macro AT_SENCNT.



```
136 #define AT_WEIGAP      "+WEIGAP"
137 #define AT_5VT        "+5VT"
138 #define AT_SETCNT      "+SETCNT"
139 #define AT_DDETECT     "+DDETECT"
140 #define AT_SETMAXNBTRANS "+SETMAXNBTRANS"
141 #define AT_GETSENSORVALUE "+GETSENSORVALUE"
142 #define AT_DISFCNTCHECK "+DISFCNTCHECK"
143 #define AT_DISMACANS    "+DISMACANS"
144 #define AT_RXDATEST    "+RXDATEST"
145 #define AT_VAL         "+VAL"
146 #define AT_ALLVAL      "+ALLVAL"
147 #define AT_SENCNT      "+SENCNT"
148 #define AT_RSTCORR     "+RSTCORR"
149 #define AT_CORRALL     "+CORRALL"
150 #define AT_GAIN1       "+GAIN1"
151 #define AT_GAIN2       "+GAIN2"
152 #define AT_GAIN3       "+GAIN3"
```

Figure 3 : Implémentation d'une macro dans le fichier at.h

De plus, les en-têtes des fonctions `at_sencnt_get` et `at_sencnt_set` illustrées dans la figure 2 doivent être définis dans `at.h`, comme illustré dans la figure 4.

```

at.h
674
675 /**
676  * @brief Print sensor count
677  * @param Param string of the AT command - unused
678  * @retval AT_OK
679  */
680 ATErrror_t at_sencnt_get(const char *param);
681
682 /**
683  * @brief Set sensor count
684  * @param Param string of the AT command
685  * @retval AT_OK if OK, or an appropriate AT_xxx error code
686  */
687 ATErrror_t at_sencnt_set(const char *param);
688

```

Figure 4 : déclaration des en-têtes de fonction pour les commandes get et set

La syntaxe utilisée dans les lignes 675-679 et 682-686 est à des fins de documentation uniquement et n'a aucun impact sur le code compilé.

`@brief`

Définit une brève description du travail de la fonction.

`@param`

Définit le/s paramètre/s d'entrée de la fonction.

`@retval`

Utilisé pour spécifier l'argument de retour de la fonction.

Les fonctions ajoutées doivent avoir un type de retour `ATErrror_t` qui forcera un statut de retour pour informer l'utilisateur en cas d'échec ou de succès, selon la convention utilisée par le code source.

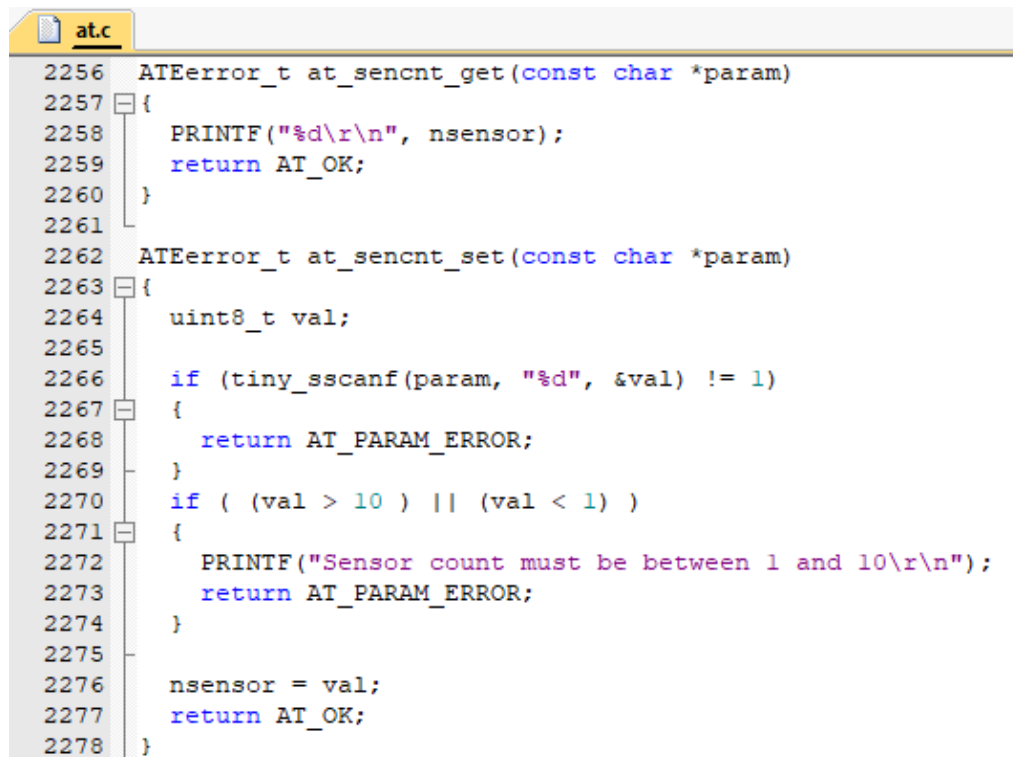
Selon la même convention utilisée, l'entrée doit être de type `const char*`. Le nom de ce pointeur est `param`.

Le même modèle d'en-tête de fonction pourrait être répété pour créer une fonction `at_sencnt_run` personnalisée par exemple.

3.6. Fichier at.c

Le fichier `at.c` contient les opérations qui seront effectuées dans les fonctions `at_sencnt_get` et `at_sencnt_set`. Ces deux fonctions doivent être conformes à leur déclaration respective dans `at.h`.

La figure 5 montre l'implémentation des fonctions `at_sencnt_get` et `at_sencnt_set` dans le fichier `at.c`.



```
2256 ATErrror_t at_sencnt_get(const char *param)
2257 {
2258     PRINTF("%d\r\n", nsensor);
2259     return AT_OK;
2260 }
2261
2262 ATErrror_t at_sencnt_set(const char *param)
2263 {
2264     uint8_t val;
2265
2266     if (tiny_sscanf(param, "%d", &val) != 1)
2267     {
2268         return AT_PARAM_ERROR;
2269     }
2270     if ( (val > 10 ) || (val < 1) )
2271     {
2272         PRINTF("Sensor count must be between 1 and 10\r\n");
2273         return AT_PARAM_ERROR;
2274     }
2275
2276     nsensor = val;
2277     return AT_OK;
2278 }
```

Figure 5 : Exemple de corps de fonction pour les commandes `get`, `set`

Le même modèle de corps de fonction avec une implémentation donnée pourrait être répété pour créer une fonction `at_sencnt_run` personnalisée par exemple.