あとがき

東京コスモス電機さんの TWE-Lite は、機能の充実したファームウェアが同梱されていて、基本、あまりモジュール自体は触らずに無線化アダプタとして簡単に使えるようなモノになっています。

が、これを Arduino に繋げて使うと、「何でそこに 32bit のマイコンが大した仕事しないで遊んでるのに 8bit のマイコンに金出さにゃならんのだ！」という微妙な状況が発生します。TWE-Lite の開発環境は、当然ながら組み込み開発の文脈を背負ったもので、これでモノ作るのは Arduino のぬる湯に浸かった後ではかなりダルイ感じになります。であればと、という流れで本稿の「Arduino で TWE-Lite を使えるようにする」という、「ラーメンを作るために小麦を植える」「楽するための面倒は厭わない」正しいプログラマ思想に基づいた開発を行う次第となりました。モノとしては、ちょうど ESP8266 が安くて Arduino からそのまま使えるモジュールとして安定供給されだしたので、かなり間の悪い感じはしますが、あちらにはない省電力っぷりとか、まだ使えるところはあるかと思います。もともと 802.15.4 はインターネット語である TCP/IP と別の思想で動いているので、この辺が楽に動かせると IoT の T の側の世界観を遊べるんじゃないかと。Arduino なんでそんな難しいこともできないんで、F**k'n IoT なモノを作るネタに出来ると良いかと思っております。

Best wish for your happy hacking!

目次

第 1 章

はじめに

1.1 本書について

本書では、JN5164(TWE-Lite) で動作するプログラム開発に関するノウハウの紹介と、開発の実践として JN516x の Arduino 対応のために Arduino の Platform Package の作成方法のガイドを提供する。

1.2 JN516x について

NXP JN516x は NXP Semiconductors N.V[*1] (以下 NXP) から販売されている無線機能付きの OpenRISC[*2] アーキテクチャを採用した 32bit マイコンである。無線機能は IEEE802.15.4 に対応し、IoT やセンサーネットワークなどの用途に適用が可能である。省電力で動作可能でコイン電池での駆動も可能である。

現在、製品シリーズとしては JN5161, JN5164, JN5168, JN5169 がラインナップされている。[*3]

1.3 TWE-Lite について

東京コスモス電機株式会社[*4] (以下 TOCOS) から発売されている JN5164 の OEM 品 (?) が TWE-Lite[*5]である。国内の技適を取得しており、国内で **合法的** に問題なく使うことができる。無線モジュール形態での提供に加えて、DIP28 サイズのブレイクアウトボードに実装したもの[*6]が、秋葉原などの電子部品店を通して供給されており、ホビーユースとしての入手性、扱いやすさに優れたパッケージとなっている。ソフト観点では JN5164 そのものなので、技術情報は

[*1] http://www.nxp.com/
[*2] http://opencores.org/or1k/Main_Page
[*3] http://www.nxp.com/techzones/wireless-connectivity/jn51xx/jn516x.html
[*4] http://tocos-wireless.com
[*5] http://tocos-wireless.com/jp/products/TWE-001Lite.html
[*6] http://tocos-wireless.com/jp/products/TWE-Lite-DIP/index.html

第 4 章

おわりに

4.1 JN516x に関する補遺

4.1.1 ctors/dtors section

C++ のコード生成時に作られる ctors(コンストラクタ) セクションは SDK のリンカ設定ではリンクしていないため、グローバル変数などスタティック領域に確保されるクラスインスタンスの初期化が行われない。具体的には、vftable の初期化などが行われず、仮想関数の解決が正常に行われない。このため、C++ の機能を利用するためにはリンカスクリプトを修正して ctors/dtors セクションをバイナリにリンクするようにし、エントリーポイントの先頭付近で、ctors セクションに含まれる初期化処理を実行する必要がある。

4.2 TWE-Lite に関する補遺

4.2.1 TWE-Lite-R について

TWE-Lite-R は JN-DR-1198 のリファレンスデザインから書き込みに不要な外部接続のクロック、Flash メモリ、LED を抜いたものである。従って、JN-DR-1198 のデザインに準拠したツールは、TWE-Lite-R においても動作可能である。

TWE-Lite-R は FT232 の USB-Serial 変換チップを使っている。FT232 はシリアル通信に使う端子以外にも GPIO として制御可能なピンを持っており、それを用いて JN516x のプログラミングモードに切り替えている。Windows, Linux においては、ドライバの切り替えなしにシリアル通信と GPIO 制御を扱えるが、MacOSX では、シリアル通信と GPIO 制御でそれぞれ別のドライバが必要となる。MacOSX で扱うときは、基板上の PRG ボタンを使って手動でプログラミングモードに遷移させる必要がある。

```
  "architecture": "jn516x",
  "category": "Contributed",
  "version": "0.1.20150429",
  "url": "https://github.com/soburi/JN516x-arduino-package/archive/release-0.1.20150429.tar.gz",
  "archiveFileName": "release-0.1.20150429.tar.gz",
  "checksum": "SHA-256:6c8fe3cdd8684a5604485718f41e252e91401cbc40812d7f3a851e6e8ef6479f",
  "size": "35846",
  "boards": [
    {
       "name": "TOCOS TWE Lite"
    }
  ],
  "toolsDependencies": [
    {
      "packager": "jn516x",
      "name": "ba-elf-ba2-gcc",
      "version": "4.1.2-r8352-build0"
    },
    {
      "packager": "jn516x",
      "name": "jenprog",
      "version": "1.1.0-20150712"
    }
  ]
},
```

name, version, url, archiveFileName, checksum, size については tools の要素と同じ意味である。architecture はプラットフォームのアーキテクチャ名 (CPU アーキテクチャ) を指定する。category は BoardsManager に表示される分類となる。boards も BoardsManager の表示に使われる。サポートしている具体的なボード名 (製品名) を入れる。toolDependencies はこの platform のバージョンが依存するツール類を定義する。これは先ほど定義した tools の内容を参照し、同時にインストールするツールを指定することになる。packager は architecture と同じものを指定する。(これについては明示的な仕様の記載がない) name と version の組でインストールするツールを指定する。

3.6.2 パッケージのインストール

Arduino 以外のパッケージを追加するには、[**環境設定**] から [Additional Boards Manager URLs] のテキストボックスにパッケージ定義の URL を指定する。複数指定する場合は URL をカンマで区切って並べる。設定を変更したのち、Arduino の UI のメニューから [**ツール**]->[**ボード**]->[Boards Manager...] を選択し、Boards Manager を立ち上げると、指定の URL で定義された 3rd party 製のパッケージが表示され、インストールができる。

本稿で作成している JN516x 向けの Platform Package は以下の URL を指定することでインストールできる。http://soburi.github.io/JN516x-arduino-package/package_soburi_jn516x_index.json

JN516x の情報が利用可能である。

1.4 Arduino について

Arduino[*7] は最近の電子工作や Make 界隈でよく使われる AVR マイコン[*8] を搭載したマイコンボードとその開発環境である。C 言語を簡易化したような言語で、プログラミング初心者でも簡単にデジタルなモノづくりができる。上級者でも組み込み開発特有の面倒な手続きがいらないので、プロトタイピングツールとしてもよく使われる。最近では、オリジナルの AVR マイコン以外にも ARM[*9]アーキテクチャのものも発売されて、マイコンボードそのものというよりも、IDE を中心としたエコシステムという色彩を強めている。最新版の IDE では、サードパーティ製のボードを使うための BoardsManager の機能が整備され、ESP8266[*10] などの新しいアーキテクチャを Arduino に対応させる動きも活発である。

1.5 本稿で使用する環境、機材

本稿では、開発ターゲットのマイコンとして JN5164(TWE-Lite DIP) を使用し、そのソフトウェア開発環境には NXP から提供されている JN516x の開発環境を用いる。Arduino 向けのパッケージを作成することから、Arduino IDE 1.6.4 版を使用する。開発環境は Windows 上に構築する。

1.6 リソースなど

本書で扱ったリソース類は、jn5164-twelite-guided-tour-resources[*11] に格納している。
また、JN516x 向けの Arduino Platform Package は JN516x-arduino-package[*12] にある。

[*7] https://www.arduino.cc/
[*8] http://www.atmel.com/products/microcontrollers/avr/
[*9] http://www.arm.com/products/processors/instruction-set-architectures/index.php
[*10] http://espressif.com/en/products/esp8266/
[*11] https://github.com/soburi/jn5164-twelite-guided-tour-resources
[*12] https://github.com/soburi/JN516x-arduino-package

第 2 章

JN516x ガイドツアー

2.1 ベンダー提供情報

JN516x の開発環境やマニュアル類は NXP 社のサイト[*1] から参照することができる。NXP 社に買収される前の開発元の Jennic 社のサポートページ[*2] から古い版数の情報をたどることができる。

2.1.1 開発環境

JN516x の開発環境はコンパイラなどのツールチェーンと、無線プロトコルに対応した SDK の部分から構成されている。

後に Arduino のマルチプラットフォーム対応を行うため、本稿では最新版ではなくコンパイラのソースが入手できる版を使用する。以下の 2 つのツールをインストールする。

- JN-SW-4041: SDK-Toolchain[*3]コンパイラなどのツールチェーン、焼き込みツールやデバッグ用ツールなどのパッケージ
- JN-SW-4065: JN516x-JenNet-IP-SDK[*4]JenNetIP(6LoWPAN) のプロトコルスタック。実質的に IEEE802.15.4 のプロトコルを包含する。

2.1.2 リファレンス文書の構成

特に参照頻度の高い文書として、ハードウェア機能のリファレンスである

- JN-UG-3087: JN516x Integrated Peripherals API User Guide[*5]

通信機能の API リファレンスとなる

[*1] http://www.jp.nxp.com/techzones/wireless-connectivity/jn51xx.html
[*2] http://www.jennic.com/support/index.php
[*3] http://www.jennic.com/download_file.php?supportFile=JN-SW-4041-SDK-Toolchain-v1.1.exe
[*4] http://www.nxp.com/documents/other/JN-SW-4065.zip
[*5] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf

パッケージ自体の情報として、名称、メンテナ、URL, e-mail があって、構成要素は platforms, tools の配列に格納される。構成は platforms が tools に依存する形になるので、先に tools を定義する。tools の各要素は以下のような構造を持つ。

```
{
  "name": "ba-elf-ba2-gcc",
  "version": "4.1.2-r8352-build0",
  "systems":
   [
     {
       "host": "i686-mingw32",
       "archiveFileName": "ba-elf-ba2-gcc_i686-mingw32_4.1.2-r8352-build0.tar.bz2",
       "url": "https://github.com/soburi/JN516x-arduino-package/blob/repository-0.0.20150723/b
       "checksum": "SHA-256:15e6c8be6d2d53c5f6d13195bd19b34fe234717a4e014167f7c40b99b104a996",
       "size": "30317284"
     },
    {
      ...
    }
   ]
}
```

name 要素はツールの格納先のフォルダ名に使われる。boards.txt からは、{runtime.tools.[TOOLNAME].path}とすることで、フォルダ名を取得できる。version は同じツールが複数版あるときの識別子として使われる。

systems の各要素はホスト環境ごとの具体的なファイル情報になる。host は Arduino の動作環境を指定する。現行で配布されている環境は以下の通り。

- Linux 64-bit (x86_64-pc-linux-gnu),
- Linux 32-bit (i686-pc-linux-gnu),
- Windows (i686-mingw32),
- Mac 64-bit(x86_64-apple-darwin)

これらの環境向けのそれぞれの配布ファイルごとにファイル情報を指定する。

url はツールの取得元 URL になる。archiveFileName はダウンロードするファイル名を指定する。(通常は http でダウンロードするときに付与されるものだが、WebServer によっては付与されない場合があるため) size は文字通りファイルサイズ。checksum はファイルの正当性確認のためのハッシュ値を付与する。[ALGORITHM]:[CHECKSUM] のフォーマットで、指定する。

platform の要素の定義は、

```
{
  "name": "JN516x Boards",
```

Power ライブラリの設計について

スリープ機能はシステムの起動に深く根差した機能なので、ライブラリとして外出しできる部分だけでは完結せず、復帰理由の取得や、WakeTimer からの時間の取得・復元など cores の init() の実装に沁み出す部分がある。このパッケージの設計指針としては、ライブラリを使わない限りはオリジナルの Arduino とは API 互換とするようにしており、このライブラリを使わない通常の Arduino と挙動は変わらないように実装している。

3.6 パッケージング、配布

3.6.1 パッケージ定義ファイル

作成したパッケージを配布するには、パッケージを構成するツールを記載した定義ファイルを作成し、それを web 上で公開する。パッケージの定義ファイルの仕様については、Arduino IDE 1.6.x package_index.json format specification[*9] にまとめられている。

Arduino からは公開したパッケージ定義ファイルの URL を指定し、パッケージをインストールすることができる。

パッケージ定義ファイルは json の形式で書かれており、ファイル名は `package_[YOURNAME]_[PACKAGENAME]_index.json` とする。

定義ファイルの構造は以下の例のようになる。

```
{
  "packages":
  [
    {
      "name": "JN516x Boards",
       "maintainer": "soburi",
       "websiteURL": "https://github.com/soburi/JN516x-arduino-package",
       "email": "soburi@gmail.com",
       "platforms": [ ... ] ,
       "tools": [ ... ]
    }
  ]
}
```

3rd-party の定義ファイルは packages の配列に複数の要素を入れないことが推奨されている。

[*9] https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.6.x---package_index.json-format-specification

- JN-UG-3024: IEEE 802.15.4 Stack User Guide[*6]
- JN-UG-3080: JenNet-IP WPAN Stack User Guide[*7]

起動処理の詳細、ISP モードの通信プロトコル、バイナリフォーマットを記載した

- JN-AN-1003: JN51xx Boot Loader Operation[*8]

が挙げられる。

2.1.3 サンプルコードなど

サンプルコードの中では Wireless UART の実装サンプル

- JN-AN-1178: IEEE802.15.4 Wireless UART for JN516x [*9]

がプログラムの基本構成、IEEE802.15.4 や UART の使い方について比較的見通しが良いサンプルとなっている。

プロトコルスタック別に雛形のコードも用意されているが、

- IEEE802.15.4 Application Template for JN516x[*10]
- JenNet-IP Application Template[*11]

如何せんソース規模が大きかったり、実動作する部分がなかったりで若干扱いづらい感じがある。

2.2 基本的な処理の流れ

JN516x のプログラムの基本的な流れとして、エントリーポイント、初期化処理、割り込みハンドラの使い方を理解する。

2.2.1 エントリーポイント

JN516x にはプログラムのエントリーポイントが 2 種類存在する。スリープ機能を使用したときの復帰で使い分けが行われる。

■AppColdStart() 電源投入時は `AppColdStart()` の呼び出しが行われる。RAM の保持を行わないスリープ状態からの復帰においても `AppColdStart()` が呼ばれる。いずれの状態からの起動かは API から判別することができる。

[*6] http://www.nxp.com/documents/user_manual/JN-UG-3024.pdf
[*7] http://www.nxp.com/documents/user_manual/JN-UG-3080.pdf
[*8] http://www.nxp.com/documents/application_note/JN-AN-1003.pdf
[*9] http://www.nxp.com/documents/application_note/JN-AN-1178.zip
[*10] http://www.nxp.com/documents/other/JN-AN-1174.zip
[*11] http://www.nxp.com/documents/application_note/JN-AN-1190.zip

■AppWarmStart() スリープ後、RAM 保持状態からの復帰では AppWarmStart() が呼ばれる。

省電力機能を使用しない場合、AppWarmStart() は AppWarmStart() の単純なラッパーとして実装し、実処理は AppColdStart() に実装すればよい。

references

■JN-AN-1003 Boot Loader Operation

■JN-UG-3024 IEEE802.15.4 Stack User Guide

2.2.2 初期化処理

Peripheral 類の使用を開始する前に vAHI_Init() を呼び出して、初期化を行う必要がある。この処理は起動毎、復帰時にそれぞれ必要である。ドキュメントの記載としては、上記のとおり、起動時、復帰時の API 呼び出し前に実行する記載となっているが、復帰時の Wake Timer の経過時間取得など、この規定に従えない処理も存在する。

```
void AppColdStart() {
    /* Peripheral API を初期化 */
    uint32_t api_version = vAHI_Init();
    ...
```

references

■JN-UG-3087 JN516x Integrated Peripherals API User Guide 18. General Functions[*12]

2.2.3 割り込みハンドラ

JN516x の割り込み処理は void (*callback)(uint32_t device_id, uint32_t item_bitmap) の型を持つのコールバック関数を割り込みハンドラとして登録することで行う。device_id は登録する割り込み種別毎に渡される値で、同一の関数で複数の割り込み種別を処理していなければ無視して構わない。item_bitmap には割り込みの発生要因が設定されて通知される。

コールバックの登録については、vAHI_[機能名]RegisterCallback の関数名で統一されている。これらの関数で動作させるそれぞれの機能について登録処理を行う。機能によって、割り込み通知の有効化の指定が必要となる。多くのものは、機能有効化の ...Enable() のような関

[*12] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G23.1010636


```
#define MSEC2WTCOUNT(ms)  ( (ms)  * 320000.f )

void PowerManager::sleep(uint32_t sleepmode, uint32_t ms)
{
  if(ms != 0)
  {
    // タイマ満了予想時刻の計算
    uint64_t estimated = clockCyclesPerMicrosecond( millis() + ms) * 1000 );
    uint32_t e_count = estimated / 0x0FFFFFFF;
    uint32_t e_tick = estimated % 0x0FFFFFFF;

    // NVM に値保持
    vAHI_WriteNVData(2, e_count);
    vAHI_WriteNVData(3, e_tick);

    // WakeTimer を起動
    vAHI_WakeTimerEnable(E_AHI_WAKE_TIMER_0, true);
    vAHI_WakeTimerStartLarge(E_AHI_WAKE_TIMER_0, MSEC2WTCOUNT(ms) );
  }
  // 割り込みを発生する機能の停止
  vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_DISABLE);

  // Sleep に遷移
    vAHI_Sleep((teAHI_SleepMode)sleepmode);

  // Sleep の遷移まで wait する
    while(true) ;
}
```

復帰時の初期化処理でタイマ値の復元を行う。WakeTimer の満了によらない復帰 (Digital I/O の割り込みなど) だった場合は、予想時刻よりも早く復帰しているので、差を反映してタイマ値を設定する。

```
#define WTCOUNT2TTCOUNT(cnt) ( (cnt) / 320000.f  * 16000000.f)
void init()
{
  uint32_t e_count = vAHI_ReadNVData(2);
  uint32_t e_tick = vAHI_ReadNVData(3) );
  if( (u8AHI_WakeTimerFiredStatus() & E_AHI_WAKE_TIMER_MASK_0) == E_AHI_WAKE_TIMER_MASK_0)
  {
    //タイマが満了していなかったら、その時間を満了予想時間から繰り上げる
    uint64_t wt_remains = u64AHI_WakeTimerReadLarge(E_AHI_WAKE_TIMER_0);
    e_count -= (WTCOUNT2TTCOUNT(wt_remains) / 0x0FFFFFFF);
    e_tick -= (WTCOUNT2TTCOUNT(wt_remains) % 0x0FFFFFFF);
  }
  ticktimer_overflow_count = e_count;
  vAHI_TickTimerWrite( e_tick );
}
```

libraries/Power/library.properties の定義ファイルを置く。中身を一通り埋めると以下の例のようになる。

```
name=JN516x Power Management
version=0.1
author=soburi
maintainer=soburi <soburi@gmail.com>
sentence=Power Management functions for JN516x.
paragraph=The library provide power management function, such as sleep and doze.
url=http://github.com/soburi/JN51xx-arduino-package
architectures=jn516x
```

ライブラリのヘッダ、libraries/Power/Power.h を作成する。今回は単純にスリープモードの指定と、復帰時間を指定するようにする。Arduino のライブラリの一般的な実装として、グローバル変数で一つのインスタンスを作成し、このインスタンスを API として使う。ここでは、この指針に従って、アクセス用のインスタンス Power を宣言する。(この指針は、組み込み開発でメモリのフラグメントを避けるために動的なメモリ確保を行わないためのもの)

```
class PowerManager
{
public:
    void sleep(uint32_t sleepmode, uint32_t ms=0);
};

extern PowerManager Power;
```

スリープ機能のメインは vAHI_Sleep() で、この関数を呼び出すことでスリープ状態に移行する。スリープからの復帰の契機は、Digital I/O の外部からの割り込みか WakeTimer の満了で復帰する。WakeTimer で復帰する場合は vAHI_Sleep() を実行する前に WakeTimer を仕掛けておく。スリープ割り込み処理の入れ違いも避けたいので、先に割り込みを無効化するのが望ましい。vAHI_Sleep() の実行から実際にスリープ状態に遷移するには多少のラグがあるので、実行したら、無限ループでプログラムを停止させる。また、割り込みとの入れ違いも避けたいので、vAHI_Sleep() の実行前に割り込みの停止を行う。

必要に応じて、Non-Volatile Memory(NVM) にカウンターを保持することで、タイマを継続してカウントする実装を行うことができる。ただし WakeTimer は 32kHz のクロックで動作しているので、16MHz のクロックで動いている TickTimer と比較すると精度が悪くなる。ここでは、タイマ満了予定時点での tick 数とカウンターを NVM に設定して、復帰時にタイマ満了していなかったら、タイマの残存時間を戻す方法で実装する。

数のパラメータとして割り込み通知の有効化を指定するが、vAHI_TickTimerIntEnable() のように独立した関数で指定するものもあり、統一されていない。それぞれの API 仕様を参照のこと。

```
    /* システム制御機能のコールバック通知を受け取る */
    vAHI_SysCtrlRegisterCallback(sysctrl_callback);
```

references

■JN-UG-3087 JN516x Integrated Peripherals API User Guide A[*13]

2.2.4 イベントループと Queue API

必要に応じて、イベントループを作成する。JN516x ではこの目的で Queue API が用意されているので、これを利用してもよい。この場合は Queue API の内部で割り込みハンドラの登録が行われるので、割り込みハンドラの登録処理を行わない。

```
void AppColdStart() {
    uint32_t api_version = vAHI_Init();
    /* Queue の初期化 */
    uint32 success = u32AppQApiInit(NULL, NULL, NULL);

    /* Event loop */
    while(true) {
        AppQApiHwInd_s *psAHI_Ind = psAppQApiReadHwInd();
        if(psAHI_Ind) {
            /* イベント処理 */
        }
    }
}
```

references

■JN-AN-3024 IEEE 802.15.4 Stack User Guide A.Application Queue API[*14]

[*13] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G40.1006067
[*14] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G15.1006178

2.2.5 DBG Functions

デバッグ用に便利な printf 形式の UART 出力機能が dbg.h に定義されている。AppColdStart() ないし、AppWarmStart() で以下のように初期化することで UART にメッセージを出力できる。

```
#ifdef DBG_ENABLE
    DBG_vUartInit(E_AHI_UART_0, E_AHI_UART_RATE_9600);
#endif
    for(int i=0; i<1000; i++) ; /* 安定化のための待ち... */
    DBG_vPrintf(TRUE, "DBG Printf enabled");
```

references

■JN-UG-3075 JenOS User Guide 6. Debug (DBG) Module[*15]

2.3 Peripherals API

Peripherals API は JN516x の比較的多彩なハード機能を使うための API である。いくつかの主要な機能について、例を交えて説明する。

2.3.1 Digital I/O

最も基本的なハード制御機能で、ピンを入出力端子として使用する。ピンから出力される電圧の HIGH/LOW を制御する。また、ピンに入力されている電圧が HIGH か LOW かを検知する。

JN516x の DIO 設定関数は、2 つの引数をそれぞれ、On にするピン、Off にするピンのビット列として受け付ける。共通ルールとして、いずれのビット列も 0 であるピンについては、**現状維持**の動作を行う。共に 1 が立っているピンについては、それぞれの関数の仕様にある優先動作に従う。vAHI_DioSetDirection() で入出力の設定、vAHI_DioSetPullup() で、内蔵プルアップ抵抗の On/Off を設定する。

```
/* 0〜3 番ピンを出力に、4〜7 番ピンを入力にする。 */
vAHI_DioSetDirection(0xF0, 0xF);

/* 0,1 番ピンをプルアップ、2,3 番ピンのプルアップを解除 */
```

[*15] http://www.nxp.com/documents/user_manual/JN-UG-3075.pdf#G10.1004281

3.5 機種固有ライブラリの作成

3.5.1 フォルダ構成

BoardsManager パッケージの中に含める機種固有ライブラリは libraries 配下にフォルダを作成する。

3.5.2 library.properties

まず、ライブラリの定義ファイルを作成する。

定義ファイルは library.properties というファイル名で、ライブラリの最上位のフォルダに置く。

このファイルの仕様は Arduino IDE 1.6.x package_index.json format specification[*8] で定義されている。 これに従って設定を作成する。ほとんどが説明用の記述なので、プログラム的な意味があるのは、

- nameArduino のメニューに表示される表示名
- version バージョン。BoardsManager でインストールするときのアップデート要否に使われる。
- architectures アーキテクチャを指定する。アーキテクチャは boards.txt で定義される値。Arduino の API のみで構成されていてアーキテクチャ依存のないライブラリは"*"を指定する。

の 3 つ。ここ以外は大きな影響はないが適切な情報を設定する。

3.5.3 ヘッダファイル

Arduino のメニューからライブラリをインポートすると、**そのライブラリに含まれるすべての.h ファイルをアルファベット順にインクルード** する。わかりやすさからは、ライブラリ名と同名のヘッダファイルが一つ読み込まれるのが良いかと思う。メインのヘッダ以外は拡張子を.h 以外 (.inc など) にするなどして、インポート時に読み込まれるヘッダファイルを限定するようにするとよい。

3.5.4 作例: Power ライブラリ

JN516x のスリープ機能を使って省電力動作を行う Power ライブラリを作成する。

まず、ライブラリのフォルダとして libraries/Power を作成する。その中に、

[*8] https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification#libraryproperties-file-format

```
void UART0_Handler(uint32_t u32DeviceId, uint32_t u32ItemBitmap)
{
  Serial.IrqHandler(u32ItemBitmap);
}
```

ハンドラの処理本体は UARTClass::IrqHandler() で実装している。

```
void UARTClass::IrqHandler(const uint32_t status)
{
  // Did we receive data?
  if ((status & E_AHI_UART_INT_RXDATA) == E_AHI_UART_INT_RXDATA)
    _rx_buffer->store_char(u8AHI_UartReadData(_dwId));

  ...
}
```

3.4.3 cores のいろいろ

cores にある、ハードウェア依存の実装が必要なものを以下に示す。

UARTClass
: UART(Serial) の機能実装

WInterrupts
: digital I/O の割り込み処理 attachInterrupt() を実装

wiring
: 時計機能の delay(), millis(), micros(). delayMicroseconds() を実装

wiring_analog
: アナログ入出力機能 (ADC, PWM) analogRead(), analogWrite() を実装

wiring_digital
: デジタル入出力 digitalRead(), digitalWrite(), pinMode() を実装

wiring_pulse
: パルス検出 pulseIn() を実装


```
vAHI_DioSetPullup(0x3, 0xB);
```

vAHI_DioSetOutput() と u32AHI_DioReadInput はシンプルな書き込み、読み出しを行う。

```
/* 0,1 番ピンを HIGH、2 番ピンを LOW にする。3 番ピンは [現在のまま] */
vAHI_DioSetOutput(0x3, 0x4);

/* 4 番ピンを読み取る */
bool pin4isHigh = ( ( u32AHI_DioReadInput() & 0x10 ) >> 4) ? true : false;
```

入力ピンとして使用する場合は、信号の立ち上がり**もしくは**立ち下がりで割り込み処理を行うこともできる。両方検知するためには、2 つのピンを使って立ち上がりと立下りを検知する必要がある。

```
/* 4,5 番ピンの割り込みを有効にする */
vAHI_DioInterruptEnable(0x10, 0x0);

/* 4 番ピンの立ち上がり時、5 番ピンの立下り時に割り込みを発生させる */
vAHI_DioInterruptEdge(0x10, 0x20);
```

ピンへの入力を契機にスリープ状態からの復帰を行える。使い方は割り込み検知の時と関数名以外は同じ。

```
/* 6 番ピンの変化でスリープから復帰する */
vAHI_DioIWakeEnable(0x70, 0x0);

/* 6 番ピンの立ち上がり時にスリープから復帰する */
vAHI_DioWakeEdge(0x70, 0x00);
```

references

■**JN-UG-3087: JN516x Integrated Peripherals API User 5. Digital Inputs/Outputs (DIOs)**[*16]

[*16] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G9.1004281

■JN-UG-3087: JN516x Integrated Peripherals API User 21. DIO and DO Functions[*17]

2.3.2 Tick Timer

Tick Timer はクロックを元にした高精度のハードウェアタイマを提供する。typical な使用方法として、システム起動からの経過時間の取得に利用できる。タイマ満了時のコールバックでタイマ満了回数をカウントして、タイマ周期を乗算することでシステム起動時からの経過時間を知ることができる。また、スピンループによるウェイト処理もこれを利用して実現できる。

最大周期を設定してタイマーを開始する。

```
/* 明示的に停止してカウンタを 0 に設定 */
vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_DISABLE);
vAHI_TickTimerWrite(0);
/* タイマ周期の最大値（約 16 秒）をセット */
vAHI_TickTimerInterval(0x0fffffff);
/* コールバック通知を設定 */
vAHI_TickTimerIntEnable(true);
vAHI_TickTimerRegisterCallback(ticktimer_callback);
/* タイマの繰り返し実行を開始 */
vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_RESTART);
```

現在のタイマー値とタイマ満了回数から経過時間が計算できるので、タイマ満了時のコールバック `ticktimer_callback()` でタイマ満了回数をカウントする。`micros()` はシステムの動作クロックを係数として起動からの経過時間を算出する。

```
#define clock2usec(a) ( (a) / clockCyclesPerMicrosecond() )
#define MICROSECONDS_PER_TICKTIMER_OVERFLOW (clock2usec(0x0FFFFFFF))
uint32_t ticktimer_overflow_count = 0;

void ticktimer_callback(uint32 u32Device, uint32 u32ItemBitmap)
{
    ticktimer_overflow_count++;
}

uint32_t micros() {
    unsigned long m;

    m = MICROSECONDS_PER_TICKTIMER_OVERFLOW * ticktimer_overflow_count +
        clock2usec(u32AHI_TickTimerRead());
    return m;
}
```

[*17] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G26.1020675

初期化処理

`UARTClass::init()` で初期化処理を実装する。UART のパラメータ設定はそこそこ数があって、以下の関数を呼んで初期化を行う。

bAHI_UartEnable()
: UART を有効化する。送受信のバッファを指定する。

vAHI_UartSetRTSCTS()
: RTS,CTS の有効無効を設定する。Arduino としては RTS,CTS は使っていないので、無効にする。

vAHI_UartSetBaudRate()
: 標準的な BaudRate 値を設定する。

vAHI_UartSetControl()
: シリアルポートの設定、パリティ、データ長、ストップビットを設定する。

vAHI_UartSetInterrupt()
: 割り込み発生の契機を設定する。データ受信時と送信 FIFO 空の時の割り込みを設定する。

vAHI_Uart0RegisterCallback()
: UART0 の割り込みハンドラを設定

Arduino の API として初期化パラメータが渡されるのは、BaudRate、パリティ、データ長、ストップビットの 4 項目の設定についてである。 `vAHI_UartSetBaudRate()`, `vAHI_UartSetControl()` にパラメータをマッピングして初期化を行う。

送信処理

`Serial.write()` で送信処理を行っている。割り込み処理中でなければ、1 バイトの送信を行い、割り込み処理中はバッファリングする。

1 バイトの送信は `vAHI_UartWriteData()` で行う。送信処理中の判定は `u8AHI_UartReadInterruptStatus()` の戻り値の `E_AHI_UART_INT_TX` ビットで判別できる。

JN516x の機能としては、`u16AHI_UartBlockWriteData()` で DMA を使った効率的な転送ができるが、母体からの変更が大きくなるので、割愛する。

受信処理

`UARTClass` の実装では、データ受信の割り込み発生時に受信バッファにデータを格納する。実際のデータの読み出しは `Serial.read()` の要求で行う。データ受信の割り込みはのパラメータの `E_AHI_UART_INT_RXDATA` ビットを設定して通知される。

割り込みハンドラは `variant/arduino_due_x/variant.cpp` で実装されているので、`variant/twelite/variant.cpp` で実装する。

```
}
```

のような感じで。

3.4.2 Serial

Arduino の頻出機能として、ログ出力に使う Serial がある。Arduino の機能としては比較的規模の大きいものであるが、避けて通れないものの一つなので、読み解いていく。

クラス階層

Serial は C++ のクラスの継承を使って実装されているので、まず、この構成を確認する。

Serial のインスタンスは UARTClass 型のグローバル変数として variants/arduino_due_x/variant.cpp で宣言されている。

この UARTClass は

Print < Stream < HardwareSerial < UARTClass

という継承関係を持っている。 それぞれの階層の役割として、

Print

型ごとに適当な表示を行うための一連の print(), println() 関数群、print() が利用する最終段の出力関数の write() が定義されている。この write() は 仮想関数となっており、このクラスを継承したクラスで実装される。(言い換えれば、このクラスを継承したクラスで print() を動作させるには、 write() のみ実装すればよい)

Stream

read(), available(), peek(), flush() の input に関する機能を定義している。これらはいずれも仮想関数で、ハードの実装にあわせて継承先で実装する。

HardwareSerial

Serial としての begin(), end() を定義している。演算子 bool() の定義もやっているが、実質的に使っていない。

UARTClass

実装クラス。継承元で定義されている仮想関数の実装と割り込みの処理を行っている

という役割になっている。それぞれの階層で定義している仮想関数を UART のハード実装にあわせて UARTClass で実装する。

UARTClass の改造

SAM 向けの UARTClass は当然 SAM のハード構成に依存したものになっているが、バッファリングの処理が多いので、実際にハードを触る箇所は多くない。必要なところを JN516x 向けに置き換えていく。

references

■JN-UG-3087: JN516x Integrated Peripherals API User 5. Digital Inputs/Outputs (DIOs)[*18]

■JN-UG-3087: JN516x Integrated Peripherals API User 21. DIO and DO Functions[*19]

2.3.3 Sleep

vAHI_Sleep() の呼び出しで、省電力モードに移行する。メモリの On/Off, 32kHz オシレータの On/Off の組み合わせ、もしくは DeepSleep が設定できる。スリープ状態からは I/O の駆動か、Wake Timer によってのみ復帰可能となるたが、Wake Timer は 32kHz のクロックで動作しているため、Wake Timer で自律的に復帰する場合には 32KHz オシレータは On にする必要がある。32kHz オシレータを停止した状態では、Digital I/O の割り込みを使って復帰する必要がある。

メモリ On 状態でのスリープ状態からの復帰は AppWarmStart(), DeepSleep を含むメモリ Off 状態からの復帰では AppColdStart() が呼ばれる。

復帰状態の詳細は u16AHI_PowerStatus() で判別することができる。

```
void AppColdStart()
{
    /* Wake Timer の満了をチェック */
    uint8_t which_timer_expired = u8AHI_WakeTimerFiredStatus();

    /* Peripheral API を初期化 */
    uint32_t api_version = vAHI_Init();

    /* 6 番ピンの変化で Sleep から復帰する */
    vAHI_DioIWakeEnable(0x70, 0x0);
    /* 6 番ピンの立ち上がり時に Sleep から復帰する */
    vAHI_DioWakeEdge(0x70, 0x00);

    /* do somethings ... */

    /* オシレータ OFF, RAM 保持で sleep (あまりやらない設定?) */
    vAHI_Sleep(E_AHI_SLEEP_OSCOFF_RAMON);
}

void AppWarmStart()
{
    /* どのピンの入力で WakeUp したのかを判定 */
    uint32_t whats_happen = u32AHI_DioWakeStatus();
```

[*18] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G9.1004281
[*19] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G26.1020675

```
    /* Peripheral API を初期化 */
    uint32_t api_version = vAHI_Init();

    /* WakeTimer を有効化 */
    vAHI_WakeTimerEnable(E_AHI_WAKE_TIMER_0, true);

    /* do somethings ... */

    /* Wake Timer をセット．一定時間後に復帰 (60sec * clock_per_sec) */
    vAHI_WakeTimerStartLarge(E_AHI_WAKE_TIMER_0, ( 60 * 32000.f);
    /* オシレータ ON, RAM 保持なしで sleep */
    vAHI_Sleep(E_AHI_SLEEP_OSCON_RAMOFF);
}
```

references

■JN-UG-3087: JN516x Integrated Peripherals API User 19. System Controller Functions vAHI_Sleep)[*20]

■JN-UG-3087: JN516x Integrated Peripherals API User 19. System Controller Functions u16AHI_PowerStatus)[*21]

2.4 Protocol Stack

2.4.1 JN516x で使えるプロトコル

JN516x では IEEE802.15.4, 6LoWPAN/JenNetIP, ZigBee の 3 種類のプロトコル実装が SDK として提供されている。

IEEE802.15.4

IEEE802.15.4 は JN516x で利用できるプロトコルスタックで最もプリミティブなプロトコルスタックである。6LoWPAN や ZigBee からも下位のデータリンク層として利用されているが、単純な通信であれば、直接 MAC 層を利用してデータ送受信を実装することも可能である。

6LoWPAN/JenNetIP

6LoWPAN は IEEE802.15.4 の上で IPv6 の通信を行うための規格である。IPv6 の (割と大きな) パケットの情報を削って、低電力デバイス向けのネットワークにデータを流せるようにする。一般の IPv6 ネットワークとはゲートウェイを介して接続する。JenNetIP は NXP が開発した 6LoWPAN を拡張したプロトコルである。JN516x の SDK としては、JenNetIP の一部として 6LoWPAN が含まれる形となる。本稿では扱わない。

[*20] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G24.1011260
[*21] http://www.nxp.com/documents/user_manual/JN-UG-3087.pdf#G24.1029146


```
void pinMode(uint32_t pin, uint32_t mode)
{
    if (mode == INPUT) {
        vAHI_DioSetDirection((1UL<<pin), 0);
        vAHI_DioSetPullup(0, (1UL<<pin));
    } else if (mode == INPUT_PULLUP) {
        vAHI_DioSetDirection(0, (1UL<<pin));
        vAHI_DioSetPullup((1UL<<pin), 0 );

    } else {
        vAHI_DioSetDirection(0, (1UL<<pin) );
    }
}
```

digitalWrite() はもっと単純で、vAHI_DioSetOuptut() にマッピングされる。

```
void digitalWrite(uint32_t pin, uint32_t val)
{
    if (val == LOW) {
        vAHI_DioSetOutput(0, (1UL<<pin));
    } else {
        vAHI_DioSetOutput((1UL<<pin), 0);
    }
}
```

delay(), millis(), micros()

点滅の時間待ち合わせ処理の delay() を実装する。Arduino の delay() はスピンループでの待ち合わせを行っている。TickTimer で現在の Tick 数がわかるので、これで起動からの経過時間取得の millis(), micros() を実装して、その時間で待ち合わせのスピンループを回せばよい。micros() は前章記載の実装を使い、delay() は

```
void delay(unsigned long ms)
{
    uint32_t start = (uint32_t)micros();

    while (ms > 0) {
        yield();
        if (((uint32_t)micros() - start) >= 1000) {
            ms--;
            start += 1000;
        }
    }
```

```
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

呼び出さなくてはならない関数は setup() と loop() の 2 つ、呼び出されてちゃんと動かなければならない関数は pinMode(), digitalWrite(), delay() の 3 つである。これらを実装していく。

起動まわり

Arduino のエントリーポイントは setup() と loop() の 2 か所がある。この 2 つの関数は cores/arduino/main.cpp で実装されている main() (!) の中で呼ばれている。

基本形としては、新規に cores/arduino/start.cpp を起こして、ストレートに

```
extern "C" void AppColdStart()
{
    main();
}
```

で良い。システムとしての初期化処理は、後述の init() で処理されるが、言語レベルの初期化やデバッグ向けの初期化などは、ここに置く。

初期化処理

main() で一番初めに呼び出される関数 init() は variant/arduino_due_x/variant.cpp で実装されている。variant 配下の構成はボード固有であるが、ここは、元の構成からあまり変更ないよう、variant/twelite/variant.cpp で実装する。

init() では、主にペリフェラル回りの初期化処理を行う。前章で触れたように、vAHI_Init() の呼び出し、割り込みハンドラ等の初期化、TickTimer の起動を行う。

pinMode(), digitalWrite()

pinMode(), digitalWrite() は Digital I/O の機能に対応する。これらの関数は cores/arduino/wiring_digital.c に実装されている。

ピンの入力・出力の方向を決める関数なので、ほぼ vAHI_DioSetDirection() と同等の機能だが、パラメータ種別にに INPUT_PULLUP があってプルアップ抵抗の有効無効も一緒に指定されるので、vAHI_DioSetPullup() でプルアップ抵抗の指定も一緒に行う。

ZigBee

ZigBee は ZigBee Alliance によって制定された IEEE802.15.4 の上の通信プロトコル。Zigbee については商用ライセンス等の制約が多いため、本稿では扱わない。

2.4.2 IEEE802.15.4 Protocol Stack の要点

IEEE802.15.4 のプロトコルスタックを使って、単純なデータ送信を行う方法について概略を示す。サンプルコードの JN-AN-1178: IEEE802.15.4 Wireless UART for JN516x がわかりやすいので、これを繙いていく。

初期化

まず、 u32AppQApiInit() で、イベントキューを初期化する。

```
    (void)u32AppQApiInit(NULL, NULL, NULL);
```

QueueAPI を使わない場合 u32AppApiInit() で 直接コールバックを処理しても構わない。static な受信バッファのアドレスを返すバッファ取得関数を作成し、MLME と MCPS の 2 種類に対して実処理を行うコールバック関数とバッファ取得関数を指定する。

2.4.3 基本操作

QueueAPI を使う場合は受信イベントはキューイングされるので、メインループの中で dequeue して処理する。サンプルでは AN1178_154_WUART_Coord/Source/AN1178_154_WUART_Coord.c と AN1178_154_WUART_EndD/Source/AN1178_154_WUART_EndD.c にある vProcessEventQueues() で実装している処理がそれである。dequeue は psAppQApiReadMcpsInd(), psAppQApiReadMlmeInd() の関数で行われ、受信したメッセージを取得できる。それぞれで取得できる構造体の u8Type のフィールドでメッセージ種別、メッセージの詳細は uParam フィールドに格納されるので、 振り分けて処理をする。

メインループでの処理は

```
while(true)
{
      MAC_MlmeDcfmInd_s *psMlmeInd = psAppQApiReadMlmeInd();
      switch(ps->MlmeInd->u8Type)
      {
      case MAC_MLME_IND_ASSOCIATE:
```

```
      /* somethings to do */
      }
      vAppQApiReturnMlmeIndBuffer(psMlmeInd);
}
```

のように、Queue から取得したメッセージを dispatch するような処理になる。

送信は `vAppApiMlmeRequest()`, `vAppApiMcpsRequest()` の関数で MLME,MCPS のメッセージの送信を行う。MLME は通信制御に関するもの、MCPS は実際のデータのやり取りに使う。送信処理の一例として、MLME-SCAN で Energy Scan を実行するコードを示す。`MAC_MlmeReqRsp_s` 構造体の `uParam` 共用体が要求種別ごとのパラメータとなる。

```
/* Structures used to hold data for MLME request and response */
MAC_MlmeReqRsp_s   sReqRsp;
MAC_MlmeSyncCfm_s  sMlmeSyncCfm;
/* Start energy detect scan */
sReqRsp.u8Type = MAC_MLME_REQ_SCAN;
sReqRsp.u8ParamLength = sizeof(MAC_MlmeReqStart_s);
sReqRsp.uParam.sReqScan.u8ScanType = MAC_MLME_SCAN_TYPE_ENERGY_DETECT;
sReqRsp.uParam.sReqScan.u32ScanChannels = 0x00000800UL;
sReqRsp.uParam.sReqScan.u8ScanDuration = 3;

vAppApiMlmeRequest(&sReqRsp, &sMlmeSyncCfm);
```

IEEE802.15.4 のメッセージ 1 つの送信は以下のような手順になる。

1. request (送信側が要求を送信を行う)
2. indicate (受信側が要求を受信する)
3. response (受信側が応答を送信する)
4. confirm (送信側が要求に対する応答を受信する)

2.4.4 接続

IEEE802.15.4 のネットワークには少なくとも一台の Coordinator が存在し、これに対して、他の EndDevice のノードが接続して通信が開始される。Coordinator は MLME-START を要求し、ネットワーク (PAN=Private Area Network) を開始する。これに先立って、MLME-SCAN 要求で energy scan を実行して、電波の混雑の少ないチャンネルを確定する。`AN1178_154_WUART_Coord/Source/AN1178_154_WUART_Coord.c` の以下の関数でそれぞれの手順を行っている。

1. `vStartEnergyScan()` (MLME-SCAN の要求)

recipe.S.o.pattern

.S をアセンブルするコマンド

recipe.ar.pattern

.o からスタティックライブラリを生成するコマンド

recipe.c.combine.pattern

elf 形式にリンクするコマンド

recipe.objcopy.bin.pattern

elf をバイナリに変換するコマンド

platform.txt で定義している変数は、ほとんどがこれらのコマンドに収斂する。

コンパイラは Platform Package の定義で構成が変わる。Platform Package としてインストールされたツールのパスは、`{runtime.tools.[TOOLNAME].path}` で参照することができる。

それぞれのコマンドの設定は、チップベンダー提供のサンプルなどを参照して設定する。Arduino 自身は特にこの辺のコマンドに対する制約などは持っていない。それぞれのボードに対して適当なバイナリが生成できればよい。

コンパイルのコマンド以外には、ファイルのアップロードツールなどのコマンドもここで定義している。アップロードに使うツールであれば、`tools.[TOOLNAME].upload.pattern` のパラメータでコマンドラインを指定する。

boards.txt はボードの種別ごとに差し替える必要のあるパラメータを定義する。boards.txt で新規のボードを定義するには `[BOARDNAME].name` のパラメータを定義する。ここで `BOARDNAME` は任意の識別文字列である。

`[BOARDNAME].build.` のパラメータはコンパイル時にコマンドラインから define 値として渡される値となる。これは、platform.txt からは `build.…`と、`BOARDNAME` を抜いた名前で参照できる。必要に応じて任意のパラメータを追加することもできる。`[BOARDNAME].upload.tool` でアップロードツールを指定できる。ここで指定できる値は、Platform Package に含まれるツールの識別子のみとなる。ツールは platform.txt から `tools.…`で参照されるので、こちらの方とも設定を整合する必要がある。

この辺りの仕様については、Arduino のドキュメントに詳細に記載されており、不明瞭な点は少ない。わからない点があれば冒頭に示した仕様を確認すれば特に問題なく設定できるはずだ。

3.4.1 Examples > 01. Basic > blink

Arduino に同梱されているサンプルプログラムの中で一番基本的なプログラムの blink、**いわゆる L チカ**である。まずは、これが動くように最低限の構成を作っていく。ソースの実行部分は以下のようになっている。

```
void setup() {
  pinMode(13, OUTPUT);
```

API 実装のソースコードを格納する

firmwares

ISP 用のファームウェア。本稿では扱わない

libraries

core に含まれない標準ライブラリや、ボード専用の拡張ライブラリを格納する

variants

core のアーキテクチャに複数のボードの実装がある場合、ボード依存部分のソースコードが置かれる。

platform.txt

コンパイル時に実行するコマンドなど、platform 依存の動作を定義するためのファイル。

boards.txt

同じプラットフォームを使う複数のボードがある場合に、その差分を定義する。今回の JN516x 向けの実装では bootloaders, firmwares は不要なので作成しない。

既存の SAM コア (Arduino Due) のソースコードをパクって fork して、JN516x 向けに改造して、cores 部分を実装する。その作業の中で、必要に応じてボード依存部分は variants に切り分けていく。

libraries は基本 API に依存するものもあるので、cores の部分が出来上がってからの作業になる。 本稿では Arduino 1.6.4 の SAM コア向けのソースコードをベースに実装していく。

3.4 platform.txt, boards.txt

具体的な API を実装するのに先立って、パッケージの設定ファイルの platform.txt と boards.txt を作成する。この 2 つのファイルでコンパイル時に実行するコマンドを決めている。

これらは Arduino IDE 1.5 3rd party Hardware specification [*7] で仕様が定義されている。platform.txt の主な役割はコンパイルを行うコマンドを定義することだ。

まず、識別名として、 `name` と `version` を設定する。`name` は Arduino の IDE のメニューから見える名前になる。`version` は現状使用されていない。

設定の中核になるのは、`recipe.` で始まるパラメータの設定である。これらがコンパイルやリンクが行われるときのコマンド定義となるからだ。

列挙すると以下のようなコマンドの定義がある。

recipe.c.o.pattern

.c をコンパイルするコマンド

recipe.cpp.o.pattern

.cpp をコンパイルするコマンド

[*7] https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5-3rd-party-Hardware-specification

2. `vHandleEnergyScanResponse()` (MLME-SCAN の結果を処理)
3. `vStartCoordinator()` (MLME-START を要求)

EndDevice 側は、MLME-SCAN 要求で active scan を実行し、接続可能な PAN を探す。PAN が見つかったら、そこに対して MLME-ASSOCIATE を要求する。

実装は `AN1178_154_WUART_EndD/Source/AN1178_154_WUART_EndD.c` のそれぞれの関数で行っている。

1. `vStartActiveScan()`
2. `vHandleActiveScanResponse()`
3. `vStartAssociate()`

MLME-ASSOCIATE 要求は Coordinator が受理し、ASSOCIATE を許可する場合は、その応答で EndDevice にアドレスを割り振って接続が完了する。Coordinator 側は `vHandleNodeAssociation()`、EndDevice 側は `vHandleAssociateResponse()` でそれぞれ、MLME-ASSOCIATE の通知、その通知への応答を処理している。

開始のシーケンスを纏めると以下のようになる。

Coordinator	EndDevice
request SCAN(energy)	
confirm SCAN(energy)	
request START	
confirm START	
	request SCAN(active)
	confirm SCAN(active)
	← request ASSOCIATE
indicate ASSOCIATE ←	
response ASSOCIATE →	
	→ confirm ASSOCIATE

2.4.5 データ送信

データ送信は MCPS-DATA 要求で行う。これは単純に、`vAppApiMcpsRequest()` でデータを送信すればよい。802.15.4 ではヘッダ含めて最大で 127 バイトである。

2.4.6 802.15.4 についての参考資料

802.15.4 のメッセージの内容についてはラピスセミコンダクタ株式会社[*22]の MK72660-01 のデータシート[*23]がわかりやすい。802.15.4 の仕様に基づいたデータの定義なので、意味自体は他社の実装においても差異は無い。

[*22] http://www.nxp.com/documents/application_note/JN-AN-1178.zip
[*23] http://www.lapis-semi.com/jp/data/datasheet-file_db/telecom/FJDK72660_01-01.pdf

第 3 章

Arduino Platform Package の作成

3.1 Arduino Boards Manager

Arduino の version1.6.2 から BoardManager の機能が追加された。これは、Arduino や、その派生ボード、互換ボードの開発環境を Platform Package としてネットワーク経由でインストールできる仕組みである。1.6.4 からは、サードパーティー製の開発環境をインストールするための設定も実装されており、「野良 Arduino」もパッケージの提供さえあれば、arduino の開発環境から使うことができる。Arduino 自身も Arduino Due [*1]や Arduino Zero [*2]などの AVR 以外のアーキテクチャについては Boards Manager で配布を行っている。

3.2 Platform Package の構成要素

Platform Package は Arduino 上でターゲットのボード用のプログラムをビルドするのに必要な要素をすべて含む。すなわち、

- コンパイラ
- アップロードツール
- Arduino core のソースコード

が含まれる。これらはパッケージ配布の定義ファイルでそれぞれダウンロードするアーカイブファイルの URL を指定する。

コンパイラ

JN516x 向けのコンパイラとして、JN-SW-4041 に含まれる gcc 相当のソースコードが TOCOS の web サイト [*3]から取得できる。これをビルドして格納する。(多少古い版の gcc だけど我慢。)

アップロードツール

[*1] https://www.arduino.cc/en/Main/arduinoBoardDue
[*2] https://www.arduino.cc/en/Main/ArduinoBoardZero
[*3] https://www.arduino.cc/en/Main/arduinoBoardDue

TOCOS から、アップロードツールとして TOCOS 版 jenprog [*4]が配布されているが、リセット処理の作りが雑で Arduino の動作とマッチしない。本稿ではオリジナルの pscholl 版 jenprog [*5]を改造したもの [*6]使用する。FT232 を使用した、リセット端子がプログラムに閉じて処理できていないので、Arduino 環境に組み込んだ時にもともとの動作仕様を維持できないため。

Arduino core

Arduino の API 実装となる Arduino core は本稿でのメインテーマである。次のセクションで具体的な実装方法を取り上げる。

3.2.1 その他

Arduino core が SDK に依存するが、ライセンスの問題でこれを含めることができない。ユーザーが別途インストールしたものを参照する形式とする。

3.3 Arduino core の実装

Arduino core は Arduino の基本 API 実装である。

3.3.1 開発用のフォルダ

Arduino のインストール先ディレクトリの `hardware/[提供元]/[プラットフォーム名]` のフォルダを作成することで、そのディレクトリがプラットフォームの検索対象となる。本稿ではこのディレクトリを使う設定で作業する。後述の BoardsManager からインストールする場合は、ユーザーディレクトリの配下に格納される。Platform Package としてパッケージングする場合も、このディレクトリの構成を tar.gz として固めて配布すればよい。Platform Package とは異なる設定となる箇所はディレクトリに別途 `platform.local.txt` を作成し、設定をオーバーライドする。

3.3.2 構成要素

基本となる AVR の core は arduino のディレクトリの `hardware/arduino/avr` にある。
別のアーキテクチャ向けに core を実装する場合は、この構成を踏襲して実装することになる。

bootloaders

文字通り bootloaer のコード。本稿では扱わない

cores

[*4] https://www.arduino.cc/en/Main/ArduinoBoardZero
[*5] http://tocos-wireless.com/jp/tech/Linux/ba-jn5148-toolchain-src.zip
[*6] http://tocos-wireless.com/jp/tech/misc/jenprog/