Vol 55, No 3—August 2021

ANALOG DEVICES
想像を超える可能性を
AHEAD OF WHAT'S POSSIBLE™
AnalogDialogue

no-OSドライバとプラットフォーム・ドライバについて学ぶ、活用する

著者：Mahesh Phalke、シニア・ソフトウェア・エンジニア

技術を急速に進化させるには、それに関連する設計の過程を簡素化する必要があります。その際に鍵になるのが、ソフトウェアのサポートです。例えば、ファームウェアやドライバのサンプル・コードをユーザに提供するといった具合です。アナログ・デバイセズは、速度、消費電力、サイズ、分解能の面で高いレベルの性能を備えるA/Dコンバータ（ADC）やD/Aコンバータ（DAC）を提供しています。本稿では、それらと共に使用するアプリケーション用ファームウェアを構築する際に、no-OSドライバやプラットフォーム・ドライバを活用する方法について説明します。

アナログ・デバイセズは、高精度のADC/DACをサポートするものとして、no-OSドライバをベースとする組み込みファームウェアのサンプルを提供しています。ここで言うno-OSドライバとは、デバイスの構成、ADC/DACからのデータのキャプチャ、キャリブレーションの実行などを担うソフトウェアのことです。このno-OSドライバをベースとするファームウェアのサンプルは、表示、保存、より高度な処理を行うために、ホストとなるPCにデータを転送する処理を円滑化する役割を果たします。

no-OSドライバとプラットフォーム・ドライバ

no-OSドライバは、OSを使わないシステムや、（特定のOSではなく）汎用のOSを採用したシステムと共に使用できるように設計されています（図1）。また、OSのサポートを一切受けることなく、ベアメタルのシステム上で使用することが可能です。no-OSドライバは、デジタル・インターフェースを介して特定のADC/DACにアクセスするためのハイレベルのAPI（Application Programming Interface）を提供するように設計されています。no-OSドライバ自身、それらのAPIを使用してデバイスとやり取りすることにより、レジスタのアドレス（メモリ・マップ）やその内容を把握することなく、データにアクセスして構成や読み書きを行えるようになっています。

no-OSドライバは、プラットフォーム・ドライバ層を利用することにより、複数種のハードウェア／ソフトウェア・プラットフォームで再利用できます。それにより、ファームウェアのポータビリティが高まります。プラットフォーム・ドライバ層は、SPI（Serial Peripheral Interface）、I^2C、GPIOといったプラットフォーム固有のインターフェースからno-OSドライバを分離し、低レベルの詳細について把握する必要性を排除します。そのため、no-OSドライバは、変更を加えることなく複数のプラットフォームで再利用できます。

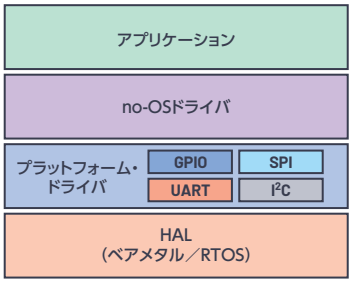

図1. 高精度のADC/DAC用ファームウェアのソフトウェア・スタック

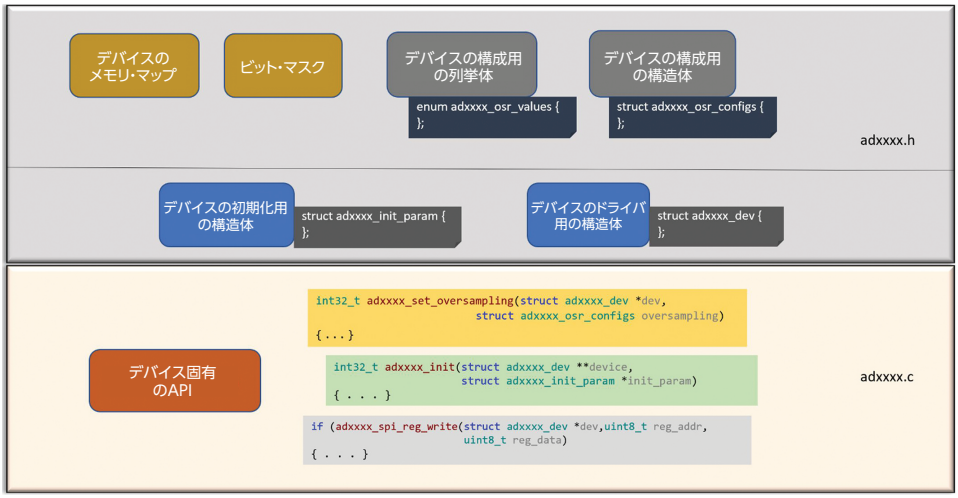

図2. no-OSドライバのコードの構造

no-OSドライバの使い方

図2に示したのは、no-OSドライバの一般的なコードの構造です。

通常、高精度のADC/DACに対応するno-OSドライバのコードは、C言語で記述された2つのソース・ファイルに含められます。adxxxx.cとadxxxx.hの2つです。adxxxxの部分は、デバイスの品番（「AD7606」や「AD7124」など）を表します。各デバイス用のヘッダ・ファイル（adxxxx.h）には、そのデバイス固有の構造体、列挙体、レジスタのアドレス、ビット・マスクに対するプログラミング用のパブリックなインターフェースが含まれています。必要なソース・ファイルにヘッダ・ファイルを含めることにより、それらの構造に対するパブリックなアクセスが可能になります。各デバイスのソース・ファイル（adxxxx.c）には、デバイスの初期化／リムーブ、デバイスのレジスタに対する読み書き、デバイスからのデータの読み出し、デバイスに固有のパラメータの取得／設定などに使用するインターフェースが実装されています。

no-OSドライバは、以下に示す共通の機能セットを中心として構成されています。

- デバイスに固有のレジスタのアドレス、ビット・マスクのマクロ、デバイスの構成用の列挙体、デバイス固有のパラメータ（オーバーサンプリング、ゲイン、リファレンスなど）の値を読み書きするための構造体の宣言
- no-OSドライバが備えるデバイスの初期化／リムーブ用の関数と、デバイス固有の初期化処理、ドライバ用の構造体、記述子によるデバイスの初期化と初期化の取り消し
- adxxxx_read_register() や adxxxx_write_register() など、デバイスのレジスタに対する読み書きを行うための関数によるデバイスのメモリ・マップやレジスタの情報へのアクセス

no-OSドライバのコードの使い方

続いて、no-OSドライバのコードの使い方について説明します。

デバイス固有のアドレス、ビット・マスク、パラメータの設定用の列挙体／構造体を使用する

上述したとおり、ヘッダ・ファイル（adxxxx.h）には、デバイス固有のすべての列挙体と構造体の宣言が含まれています。それら列挙体と構造体は、デバイスのパラメータの設定やアクセスを行うために、デバイス固有の関数やAPIに引き渡されます。図3に示したのはそれらに関連するコードです。

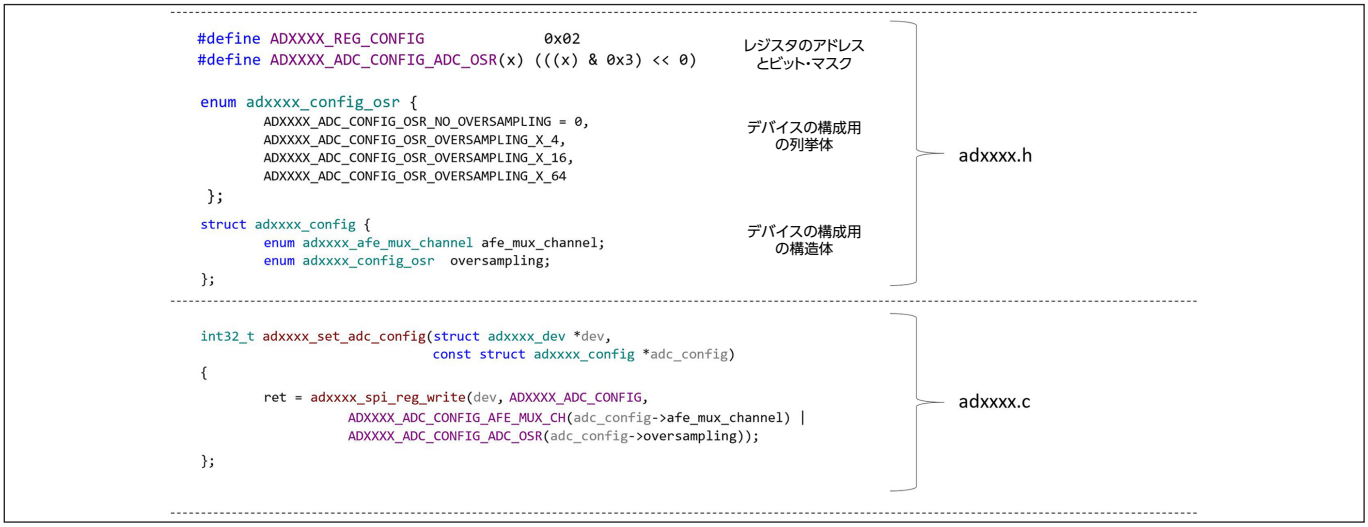

図3. デバイスの構成に使用する
列挙体、構造体、API

図3を見ると、adxxxx_configという構造体があります。この部分のコードは、マルチプレクサによってチャンネルを選択し、そのチャンネルのオーバーサンプリング・レートを設定するために使用されます。この構造体のメンバ（afe_mux_channelとoversampling）は、このヘッダ・ファイルの中で列挙型で宣言されています。両方のフィールドについて、ユーザが選択可能なすべての値を定数として含む列挙体が定義されています。

adxxxx.cでは、adxxxx_set_adc_config()という関数を定義しています。この関数は、構成用の構造体を介してユーザから引き渡された構成情報／パラメータを取得します。その上で、adxxxx_spi_reg_write()関数を呼び出すことにより、デジタル・インターフェース（この例ではSPI）を介してそれらのデータをデバイスのADXXXX_REG_CONFIGというレジスタに書き込みます。

no-OSドライバのデバイス用の構造体と初期化用の関数を使用して、デバイスを初期化する

no-OSドライバには、上述したデバイスの構成用の列挙体と構造体に加えて、以下の2つの構造体が含まれています。

- デバイスの初期化用の構造体
- デバイスのドライバ用の構造体

それぞれのコードは図4に示したとおりです。

```
struct adxxxx_init_param {
  /** SPI initialization parameters */
  spi_init_param spi_init;
  /** GPIO initialization parameters */
  struct gpio_init_param *gpio1;
  enum adxxxx_device_id device_id;
  /** Configuration register settings */
  struct adxxxx_config config;
}
```

デバイスの初期化用の構造体

```
struct adxxxx_dev {
  /** SPI descriptor */
  spi_desc *spi_desc;
  /** GPIO initialization parameters */
  struct gpio_desc *gpio1;
  enum adxxxx_device_id device_id;
  /** Configuration register settings */
  struct adxxxx_config config;
  uint8_t data_buffer[8];
}
```

デバイスのドライバ用の構造体

図4. デバイスの初期化用の構造体と
ドライバ用の構造体

デバイスの初期化用の構造体を利用することで、ユーザ・アプリケーションのコード内で、デバイス固有のパラメータと構成についての定義を行うことができます。初期化用の構造体には、デバイス固有の他のパラメータの構造体と列挙体のメンバが含まれています。図5に、デバイスの初期化用の構造体に対応する定義の例を示しました。

デバイスの初期化用の関数であるadxxxx_init()により、ドライバ用の構造体にデバイスの初期化用のパラメータが読み込まれます。ドライバ用の構造体は、ヒープ領域から実行時（動的）メモリに割り当てられます。デバイスのドライバ用の構造体と初期化用の構造体の中で宣言されているパラメータは、ほぼ同一です。前者は後者の実行用のバージョンだと表現できます。

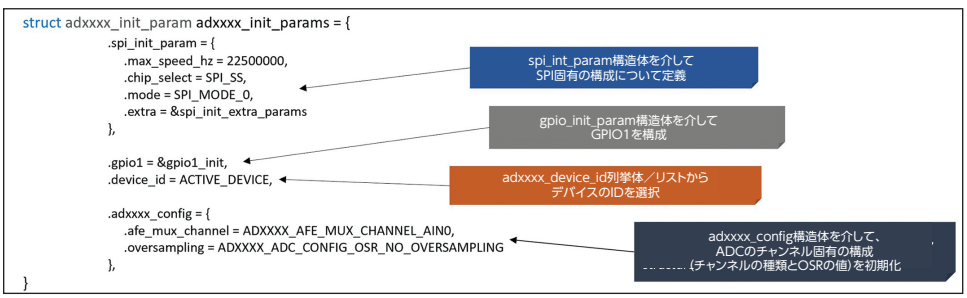

図5. ユーザ・アプリケーションにおける定義の例。
デバイスの初期化用の構造体について記述しています。

一般的な初期化用の関数の概要と初期化の流れは、以下のようになります。

【ステップ1】
デバイスの初期化用の構造体の定義（またはインスタンス）を、アプリケーションのコード内に記述し（例えば、struct adxxxx_init_params）、ユーザに固有のデバイスのパラメータと、プラットフォームに依存するドライバのパラメータを初期化します。これらのパラメータは、コンパイルの際に定義されます。なお、初期化用の構造体の中で定義されるパラメータは、デバイスごとに異なることには注意が必要です。

```
struct adxxxx_init_param adxxxx_init_params = { … }
```

【ステップ2】
デバイスのドライバ用の構造体に対するポインタ・インスタンス（変数）を、アプリケーションのコード内に記述します。

ユーザ・アプリケーションでは、そのコード内に、デバイスのドライバ用の構造体に対するポインタ・インスタンスを1つ記述する必要があります。このインスタンスをno-OSドライバのすべてのAPI／関数に引き渡すことにより、デバイスに固有のパラメータへのアクセスが行われます。このポインタ・インスタンスは、ヒープ領域内に動的に割り当てられたメモリのデータを参照します。この処理は、no-OSドライバ内に定義されたadxxxx_init()など、デバイスの初期化用の関数によって実行されます。

```
static struct adxxx_dev *p_adxxxx_dev = NULL;
```

【ステップ3】
デバイスの初期化用の関数を呼び出すことにより、デバイスやその他のプラットフォーム固有のペリフェラルを初期化します。

```
/* Initialize ADxxxx device and peripheral interface */
init_status = adxxxx_init(&p_adxxxx_dev, &adxxxx_init_str);
if (init_status != SUCCESS) {
       return init_status;
}
```

no-OSドライバに定義されたadxxxx_init()関数は、adxxx_init_param構造体を介して引き渡されたユーザ固有のパラメータによってデバイスを初期化します。この初期化用の関数には、デバイスのドライバ用の構造体に対するポインタ・インスタンスと、デバイスの初期化用の構造体のインスタンスの2つが引数として引き渡されます。ユーザ・アプリケーションのコード内では、adxxxx_init()関数を複数回呼び出すことができます。但し、この関数を再度呼び出す場合には、その前にデバイスのリムーブ用の関数を呼び出す必要があります。

デバイスのレジスタに対する読み書き用の関数を使用して、メモリ・マップ（レジスタの内容）にアクセスする

ユーザは、no-OSドライバのデバイス固有のadxxx_read/write()関数を介して、デバイスのレジスタの内容（製品のID、スクラッチ・パッド・メモリの値、OSRなど）にアクセスすることができます（図6）。

多くの場合、ユーザはレジスタにアクセスするための関数を直接使用することはありません。それらの関数は、adxxxx_ spi_reg_read/write()といったデバイス固有の関数によって呼び出されます。可能であれば、レジスタにアクセスするための関数を直接使用する代わりに、デバイスの構成やステータスの確認用のAPIを使用して、デバイスのメモリ・マップにアクセスするべきです。そうすれば、デバイスのドライバ用の構造体と実際のデバイスの構成の間の同期が確保されるからです。

プラットフォーム・ドライバの概要

プラットフォーム・ドライバは、プラットフォームに固有のAPIをラップするHAL（Hardware Abstraction Layer：ハードウェア抽象化層）の1つです。no-OSドライバやユーザ・アプリケーションのコードからプラットフォーム・ドライバを呼び出すという方法によって、基盤として存在するハードウェア・プラットフォーム／ソフトウェア・プラットフォームに対する独立性が実現されます。プラットフォーム・ドライバは、プラットフォームに固有の低レベルのハードウェア機能をラップします。そうした機能の例としては、SPI/I^2Cの初期化／読み書き、GPIOの初期化／読み書き、UART（Universal Asynchronous Receiver/Transmitter）の初期化と送受信、ユーザ固有の遅延、割込みなどが挙げられます。

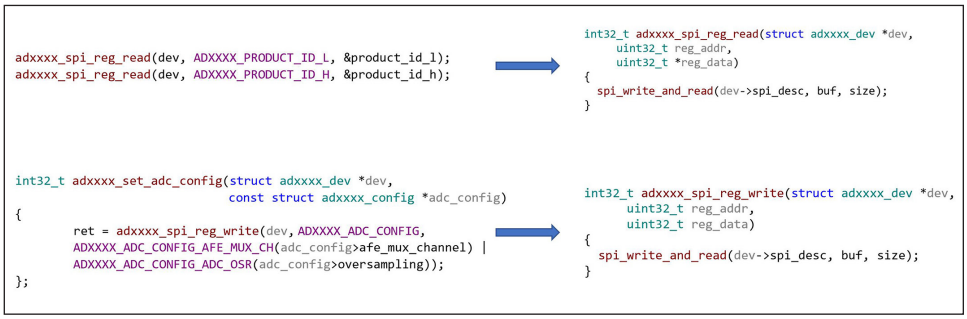

図6. レジスタの内容にアクセスするためのコード

図7に示したファイル構造は、SPIに対応するプラットフォーム・ドライバのモジュールの例です。

プラットフォーム・ドライバの使い方

通常、プラットフォーム・ドライバのコードは、C/C++言語で記述された3つのソース・ファイルとして用意されます。以下、それぞれについて順に説明します。

1）spi.h

1つ目のファイルはspi.hです。これは、プラットフォームに非依存のヘッダ・ファイルであり、SPIの機能に必要なデバイスの構造体と列挙体を含みます。これによって定義されるC言語のプログラミング・インターフェースは、プラットフォームに依存しません。

初期化用の構造体とデバイスの構造体で宣言されるパラメータは、任意のプラットフォーム上のSPIに対して共通のものです。

デバイスの初期化用の構造体に含まれるvoid *extraパラメータにより、使用しているプラットフォームに固有のパラメータなど、追加のパラメータを引き渡すことができます。

SPIのドライバ用の構造体とSPIの初期化用の構造体の中で宣言されているパラメータは、ほぼ同一です（図8）。前者は後者の実行用のバージョンだと表現できます。

2）spi.cpp/.c

このファイルには、spi.hファイルで宣言されている関数が実装されます。それらの関数は、特定のプラットフォームにおけるSPIのペリフェラルの初期化と、データの読み書きに使用されます。プラットフォームという語は、広い意味では、マイクロコントローラ（ターゲット・デバイス）とソフトウェア（RTOSやMbed OSなど）の組み合わせのことを指します。このファイルはプラットフォームに依存するので、別のプラットフォームにポーティングする際には変更を加えなければなりません。

図9は、Mbed OSをベースとするプラットフォーム上のSPIについて詳しく示したものです。

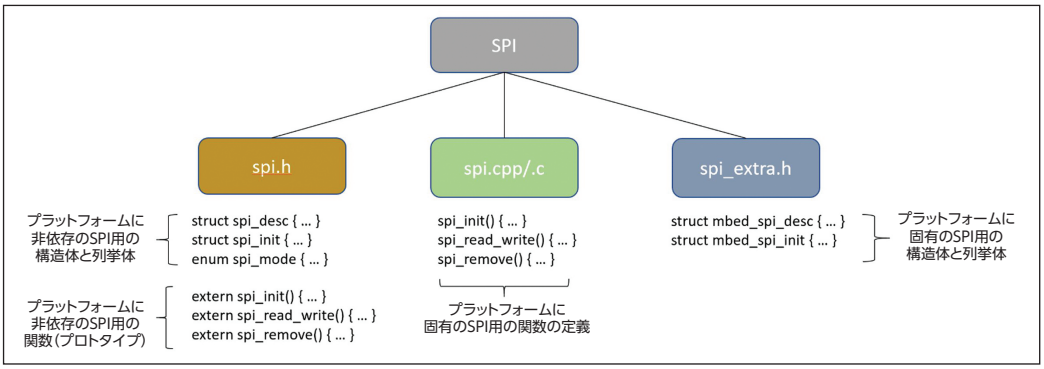

図7. SPIに対応するプラットフォーム・ドライバのファイル構造

```
typedef struct spi_init_param {
    uint32_t max_speed_hz;
    uint8_t chip_select;
    enum spi_mode mode;
    void *extra;
} spi_init_param;
```

SPIの初期化用の構造体

```
typedef struct spi_desc {
    uint32_t max_speed_hz;
    uint8_t chip_select;
    enum spi_mode mode;
    void *extra;
} spi_desc;
```

SPIのドライバ用の構造体

図8. SPIの初期化用の構造体とドライバ用の構造体

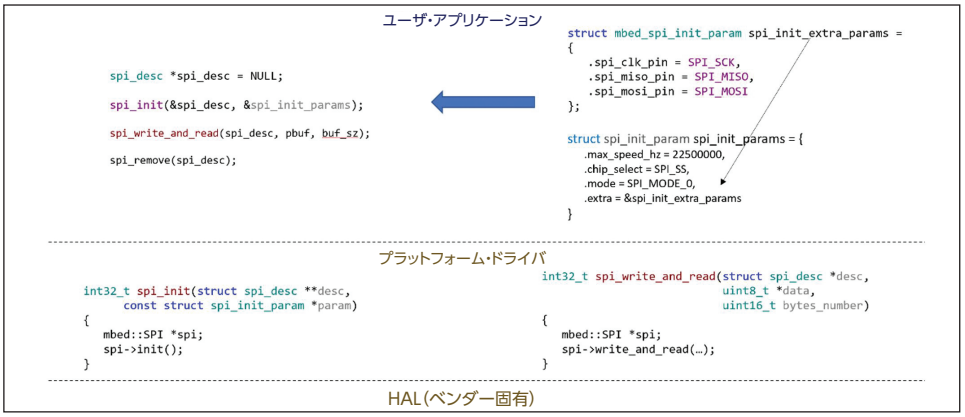

図9. SPIに関連するAPI／関数。spi_init()とspi_write_and_read()に追加されるコードは省略しています。わかりやすさを優先して、細部は省いていることに注意してください。

これらのインターフェース、デバイスの初期化用の構造体、ドライバ用の構造体を使用し、SPIを初期化してデータを読み書きする方法を表しています。

3）spi_extra.h

このファイルには、デバイスの追加の構造体や列挙体が含まれています（図10）。これらは、特定のプラットフォームに固有のものです。このファイルを利用すれば、ユーザ・アプリケーションのコードの中で、汎用のファイルであるspi.hには含まれていない内容で構成を行うことができます。例えば、SPIのピン構成はプラットフォームによって異なりますが、これについてはプラットフォームに固有の追加の構造体に加えることができます。

プラットフォーム・ドライバのポーティング

プラットフォーム・ドライバは、あるプラットフォーム（マイクロコントローラ）から別のプラットフォームへとポーティングすることができます。通常、これはプラットフォームに固有の.cpp/.cファイルと_extra.hファイルを作成することによって行います。プラットフォーム・ドライバは、マイクロコントローラのベンダーから提供されるデバイス固有のHALの1つ上の層に存在します。したがって、ポーティングを実施する際には、プラットフォーム・ドライバにおいて、ベンダーから提供されたHALに含まれる関数やAPIの呼び出しに関連するコードを少し変更する必要があります。

```
typedef struct mbed_spi_init_param
{
  uint8_t spi_miso_pin;
  uint8_t spi_mosi_pin;
  uint8_t spi_clk_pin;
} mbed_spi_init_param;
```

SPIの初期化用の追加の構造体

```
typedef struct mbed_spi_desc {
  void *spi_port;
  void *slave_select;
} mbed_spi_desc;
```

SPIのドライバ用の追加の構造体

図10. SPI用の追加の構造体

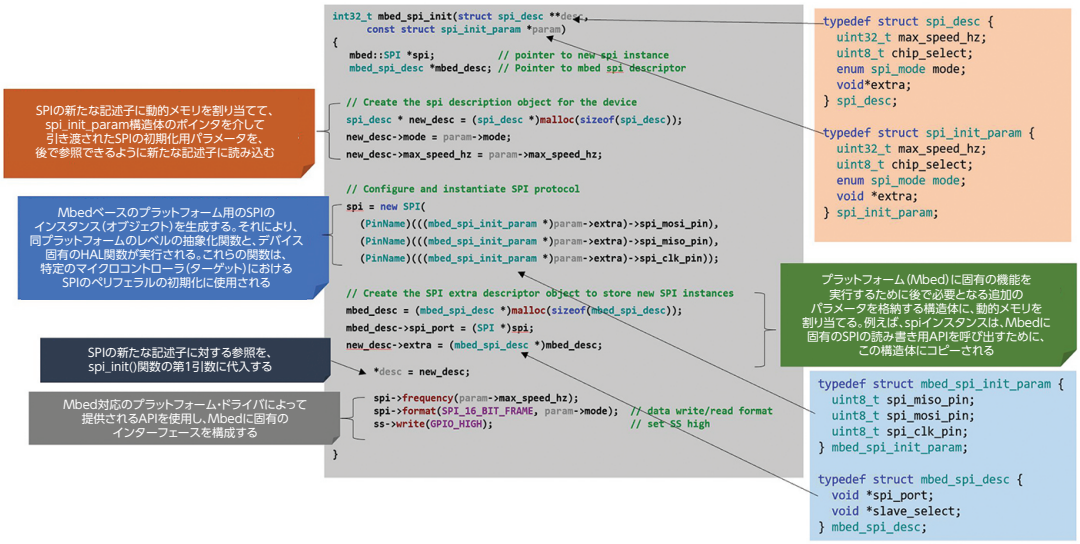

図11. Mbedベースのプラットフォームに固有のSPIの初期化を実行するためのコード

図12は、Mbed OSをベースとするSPIのプラットフォーム・ドライバとアナログ・マイクロコントローラ「ADuCM410」のSPIのプラットフォーム・ドライバの違いを示したものです。

アナログ・デバイセズのWikiとGitHubのページには、no-OSドライバのリポジトリとGitHub上にあるプラットフォーム・ドライバのソース・コードへのリンクが用意されています。

no-OSドライバの開発への寄与

no-OSドライバは、OSを使用しないシステム用にC言語で記述されています。アナログ・デバイセズのno-OSドライバは、GitHub上でオープン・ソースとして提供されています。高精度のADC/DACだけでなく、加速度センサー、トランシーバー、光電デバイスなど、アナログ・デバイスの数多くの製品もサポートしています。ソース・コードの扱いに慣れている方なら、必要な変更に関するコミットを行い、その内容をレビューするためのプル・リクエストを作成することで、ドライバの開発に寄与できます。

数多くのサンプル・プロジェクトが、Linux環境やWindows環境を対象として実施されています。サンプルの多くは、HDL（ハードウェア記述言語）によって記述され、Xilinx®製またはIntel®製のFPGA上で実行されます。

なお、アナログ・デバイセズのWikiでは、Mbed OSとADuCMxxxをプラットフォームとして使用する高精度ADC/DAC用に開発されたサンプルを提供しています。

spi_extra.h

```
typedef struct mbed_spi_init_param {
  uint8_t spi_miso_pin; // SPI MISO pin (PinName)
  uint8_t spi_mosi_pin; // SPI MOSI pin (PinName)
  uint8_t spi_clk_pin;  // SPI CLK pin (PinName)
} mbed_spi_init_param;
```

```
struct aducm410_spi_init_param {
  /** Select the SPI channel */
  enum spi_channel spi_channel;
  /** Select the operation mode */
  enum master_mode master_mode;

  /** SPI pin muxing configured in application */
  bool miso_gpio_muxing;
  bool mosi_gpio_muxing;
  bool cs_gpio_muxing;
};
```

spi.cpp/.c

```
int32_t mbed_spi_write_and_read(struct spi_desc *desc,
                                uint8_t *data,
                                uint16_t bytes_number)
{
   ss->write(GPIO_LOW);
   for (size_t byte = 0; byte < bytes_number; byte++) {
     data[byte] =  spi->write(data[byte]);
   }
   ss->write(GPIO_HIGH);
}
```

```
int32_t aducm410_spi_write_and_read(struct spi_desc *desc,
                                    uint8_t *data,
                                    uint16_t bytes_number)
{
   while (!(SpiSta(spi_port) & BITM_SPI_STAT_XFRDONE)) ;
      spi_port->STAT |= BITM_SPI_STAT_XFRDONE;

   for (size_t byte = 0; byte < FIFO_SIZE; byte++) {
      SpiTx(spi_port, data[byte_indx + byte]);
   }
}
```

MbedのHAL（ベンダー固有）　　HAL（ADuCM410）

図12. プラットフォーム・ドライバの違い

著者について

Mahesh Phalke（mahesh.phalke@analog.com）は、アナログ・デバイセズのシニア・ソフトウェア・エンジニアです。インドのバンガロールを拠点とする高精度コンバータ向けソフトウェア・グループに所属。2011年にプネー大学で電子工学の学士号を取得しています。

AD5508-0-8/21