組み込みマイクロコントローラのアプリケーションをOTAでアップデート――その機能の設計ノウハウとトレードオフについて学ぶ

著者: Benjamin Bucklin Brown

概要

多くの組み込みシステムは、人間が作業するのは困難な場所や、現実的には作業が不可能な場所に配備されます。IoT（Internet of Things）の分野では、この傾向が特に顕著になります。その種のアプリケーションでは、数多くのデバイスが配備されます。また、各デバイスのバッテリ寿命には限りがあることが多いはずです。人間や機械の状態を監視する組み込みシステムにも、そうした例が存在します。ソフトウェアのライフサイクルが短いことも相まって、多くのシステムでは、OTA（Over-the-Air）によるアップデートをサポートすることが求められています。つまり、組み込みシステムで使われているマイクロコントローラ／マイクロプロセッサ上のソフトウェアを、無線通信を利用して新しいものに置き換えるということです。このOTAアップデートは、スマートフォンに代表される携帯端末でも使われている手法なので、既に多くの人にとってなじみ深いものになっています。ただ、リソースが限られたシステムをOTAアップデートに対応させるには、設計と実装における多くの課題に対処しなければなりません。本稿では、OTAアップデートに対応する複数のソフトウェア設計の例を示し、対処すべきトレードオフについて説明します。また、超低消費電力のマイクロコントローラ製品を2つ取り上げ、OTAアップデートに活用可能なハードウェア機能を紹介します。

OTAアップデートの構成要素

サーバーとクライアント

上述したように、OTAアップデートでは、デバイス上のソフトウェアを、ワイヤレスでダウンロードした新しいソフトウェア（新しいバージョン）に置き換えます。通常、組み込みシステムにおいてソフトウェアを実行するのはマイクロコントローラです。マイクロコントローラは小さなコンピュータ・デバイスであり、メモリ容量や速度、消費電力に制限があります。一般に、マイクロコントローラはマイクロプロセッサ（コア）と、特定処理向けのデジタル・ハードウェア・ブロック（ペリフェラル）で構成されます。OTAアップデートを導入したいアプリケーションには、アクティブ・モードの消費電流が30µA/MHz～40µA/MHzという超低消費電力のマイクロコントローラが理想的です。特定のペリフェラルを使用し、マイクロコントローラを低消費電力モードで動作させられるようにすることが、OTAアップデートに対応するソフトウェアの設計における重要な部分です。図1に示したのは、OTAアップデートが必要になり得る組み込みシステムの例です。この図において、マイクロコントローラは無線トランシーバーICとセンサーICに接続されています。この構成は、センサーによって環境に関するデータを収集し、無線を使用して得られた情報を定期的に送信するIoTアプリケーションを想定したものです。IoTに対応するシステムにおいて、この部分はエッジ・ノードまたはクライアントと呼ばれます。そして、この部分がOTAアップデートの対象になります。システムのその他の部分は、クラウドまたはサーバーと呼ばれており、新しいバージョンのソフトウェアの供給元となります。サーバーとクライアントは、トランシーバーを使用したワイヤレス通信によって情報をやり取りします。

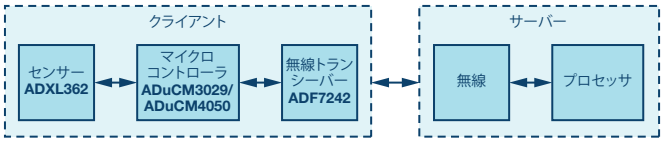


図1. クライアント-サーバーのアーキテクチャを適用した組み込みシステム

ソフトウェア・アプリケーションの仕組み

OTAアップデートのプロセスのうち、ほとんどの部分は、ソフトウェアをサーバーからクライアントに転送する処理で占められます。ソフトウェアは、ソース形式からバイナリ形式に変換された後、バイト・シーケンスとして転送されます。変換処理では、ソース・コードのファイル（拡張子はc、cppなど）のコンパイルとリンクを順次実施した上で実行可能ファイル（拡張子はexe、elfなど）が生成されます。更に、そのファイルが移植性のあるバイナリ・ファイル（拡張子はbin、hexなど）に変換されます。これらの形式のファイルに含まれるバイト・シーケンスは、マイクロコントローラが内蔵するメモリの特定のアドレスに対応します。システムの状態を変更するためのコマンドや、システムが備えるセンサーによって収集したデータなど、ワイヤレスで送信される情報は、いずれも概念的にはデータとして捉えられます。OTAアップデートにおいて、送信されるデータはバイナリ形式の新しいソフトウェアです。多くの場合、バイナリ・ファイルはサイズが大きく、サーバーからクライアントに対して一度に送信することはできません。そこで、バイナリ・ファイルを複数のパケットに分割する処理（パケット化）が必要になります。図2は、こうした一連の処理の概念を表したものです。異なるバージョンのソース・コードを基に異なるバイナリ・ファイルが生成され、OTAアップデートの際に新たなパケットとして送信されるということです。この簡単な例では、各パケットが8バイトのデータで構成されています。前半の4バイトで表されるクライアントのメモリのアドレスに、後半4バイトのデータが格納されます。

図2．ソフトウェア・アプリケーションのバイナリ変換とパケット化

主要な課題

上記のようなOTAアップデートのプロセスには、解決すべき3つの主要な課題が存在します。1つは、メモリに関する課題です。このプロセスでは、新しいソフトウェアをクライアント・デバイスの揮発性／不揮発性メモリ上に適切に配備し、アップデートが完了したら実行可能な状態にする必要があります。旧バージョンのソフトウェアは、新しいバージョンに問題があった場合に備えて、フォールバック用に残しておかなければなりません。また、リセットをかけている間や電源に関する処理が行われている間には、その時点で使用中のソフトウェアのバージョンやメモリ内の位置といったクライアント・デバイスの状態を保持しておく必要もあります。2つ目の課題は、通信に関するものです。上述したように、OTAアップデートでは、新しいバージョンのソフトウェアをサーバーからクライアントに送信する必要があります。その際には、各クライアントが備えるメモリ上の特定のアドレスをターゲットとする個々のパケットを生成しなければなりません。つまり、パケット化の方法、パケットの構造、データの転送に使われるプロトコルといったあらゆることを考慮して、ソフトウェアを設計する必要があるということです。3つ目の課題はセキュリティです。新しいソフトウェアをサーバーからクライアントにワイヤレスで送信する場合、そのサーバーは送信元として信頼できるものでなければなりません。このセキュリティ上の課題は、認証作業によって解決する必要があります。また、新しいソフトウェアには機密情報も含まれているはずなので、第三者に読み取られることがないようにしなければなりません。このセキュリティ上の課題を、機密性と呼びます。もう1つ、セキュリティの面で重要な要素があります。それは完全性です。つまり、新しいソフトウェアをOTAで送信している最中にデータが破損しないことを保証しなければなりません。

セカンドステージ・ブート・ローダ（SSBL）

ブート・シーケンスの概要

プライマリ・ブート・ローダ（PBL）は、マイクロコントローラの読み取り専用メモリに永続的に格納されているソフトウェア・アプリケーションです。PBLが存在するメモリ領域は、インフォ・スペース（Info Space）と呼ばれます。この領域には、ユーザがアクセスすることはできない場合があります。PBLは、リセットがかけられるたびに実行され、一般的に必須となるハードウェアの初期化を行います。また、ユーザが定義したアプリケーション・ソフトウェアをメモリに読み込む場合もあります。マイクロコントローラがフラッシュ・メモリなどの不揮発性メモリを内蔵している場合、PBLが読み込みを行う必要はなく、フラッシュ・メモリ内のプログラムに制御を引き渡すだけで済みます。PBLがOTAアップデートを全くサポートしない場合には、セカンドステージ・ブート・ローダ（SSBL）が必要になります。PBLと同様に、SSBLはリセットがかかるたびに実行されます。またSSBLにはOTAアップデートのプロセスの一部が実装されます。図3に、その場合のブート・シーケンスを示しました。以下では、SSBLが必要になる理由と、このアプリケーションの役割を定めることが設計時の重要なトレードオフ項目になる理由を説明します。

経験に基づく教訓：SSBLは必須

概念上は、SSBLを省いて、すべてのOTAアップデートの機能をユーザ・アプリケーション内に実装する方が簡単に思えるかもしれません。そうすれば、OTAアップデートのプロセスにおいて、既存のソフトウェア・フレームワーク、OS、デバイス・ドライバをシームレスに活用できるからです。図4に、このアプローチを選択した場合のシステムのメモリ・マップとブート・シーケンスを示しました。

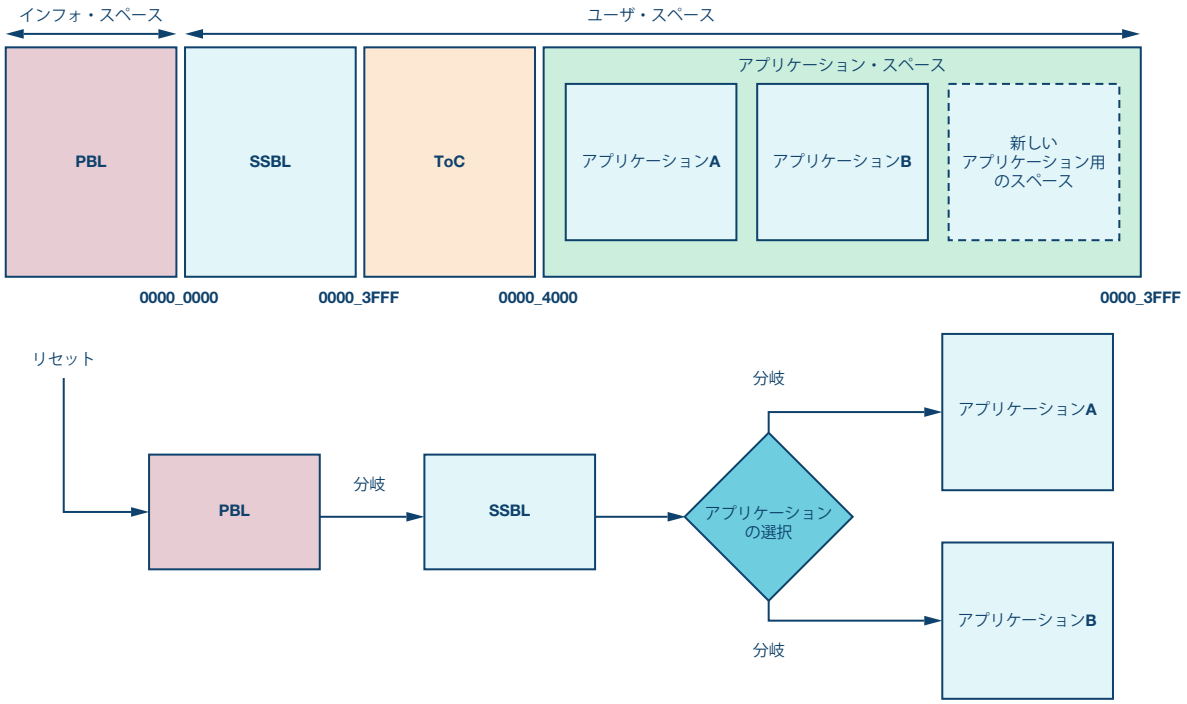


図3．SSBLを使用する場合のメモリ・マップとブートのフローの例

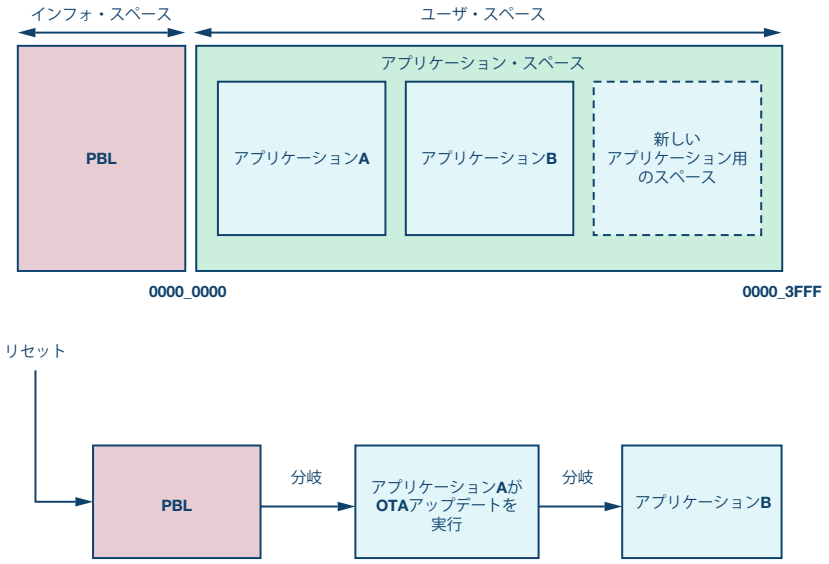


図4. SSBLを使用しない場合のメモリ・マップとブートのフローの例

アプリケーションAは、現場に配備されたマイクロコントローラにインストールされているアプリケーションです。このアプリケーションには、OTAアップデートに関連するソフトウェアが含まれています。サーバーから要求があった場合には、それを使用してアプリケーションBのダウンロードが行われます。アプリケーションBのダウンロードと検証が完了したら、アプリケーションAは、アプリケーションBのリセット・ハンドラへの分岐命令を実行することにより、アプリケーションBに制御を引き渡します。リセット・ハンドラは、ソフトウェア・アプリケーションのエントリ・ポイントとなる小さなコードであり、リセットがかかったときに実行されます。ここでは、関数の呼び出しと等価な分岐が実行されることによってリセットがかかります。SSBLを使わない場合、このようなアプローチになるのですが、この方法には以下に示す2つの大きな問題があります。

- 多くの組み込みソフトウェア・アプリケーションは、リアルタイムOS（RTOS）の環境で稼働します。RTOSの環境では、ソフトウェアが並行のタスクに分割され、システム内でそれぞれが独立した処理として実行されます。例えば、図1に示したアプリケーションでは、センサーで取得したデータの読み取り、そのデータに対するアルゴリズムの適用、無線通信といったタスクがRTOS環境で実行されると考えられます。RTOSそのものは常に稼働しており、非同期のイベントや時間ベースの特定の遅延に基づいて、タスク間の切り替えを実施します。ここで、RTOSのタスクから新しいプログラムに処理を分岐するのは、安全な方法だとは言えません。他のタスクがバックグラウンドで実行されたままになるからです。RTOS環境でプログラムを安全に終了させるには、リセットをかけるしかありません。
- 図4において、PBLからアプリケーションAではなくアプリケーションBに分岐すれば、上述した問題は解決できるように思えます。しかし、一部のマイクロコントローラでは、PBLは常に割り込みベクタ・テーブル（IVT）を備えるプログラムを実行します。IVTは、アプリケーションにおける重要な部分です。割り込み処理の関数が記述され、アドレス0に配置されます。アプリケーションBに分岐してリセットするには、IVTの再配置が必要になります。このIVTの再配置の間に電源に関する処理が生じると、システムは永久に故障した状態のままになるおそれがあります。

上記の問題は、図3に示すようにSSBLをアドレス0に固定することによって解決できます。SSBLはRTOS対応のプログラムではないので、新しいアプリケーション（ソフトウェア）に安全に分岐できます。SSBLのIVTはアドレス0にあり、再配置されることはありません。そのため、電源に関する処理によって、システムが破壊的な状態に陥る心配もありません。

設計上のトレードオフ：SSBLの役割

ここまでに、SSBLとアプリケーション・ソフトウェアの関係について詳しく解説しました。続いては、SSBLの機能について説明します。このプログラムで実現しなければならない最低限の機能は、現在実行されているアプリケーション（とその開始位置）を特定し、そのアドレスに分岐することです。一般に、マイクロコントローラのメモリ内にある様々なアプリケーションの位置情報は、図3に示したToC（Table of Contents）に格納されています。ToCは、メモリ上の永続的な共有領域にあり、SSBLとアプリケーション・ソフトウェアの両方が、互いに通信を実施するために使用します。OTAアップデートのプロセスが完了すると、ToCに格納されたデータは新しいアプリケーションの情報で更新されます。OTAアップデートの機能の一部をSSBLに含めることも可能です。どの部分を含めるかを決定することが、OTAアップデート用のソフトウェアを開発する際の重要な検討事項になります。上記のような最低限の機能を実現するSSBLは、極めて単純なものです。したがって、簡単に検証できます。また、アプリケーションのライフサイクルの過程で変更が必要になることはほぼありません。但し、それは、新たなアプリケーションのダウンロードと検証を、各アプリケーションが個々に行わなければならないということを意味します。その場合、無線スタック、デバイスのファームウェア、OTAアップデート用のソフトウェアにおいて、コードの重複が生じる可能性があります。一方、OTAアップデートのプロセス全体をSSBLに含めることも可能です。その場合、アプリケーションがやるべきことは、ToCにフラグを設定してアップデートを要求し、リセットを実行することだけです。ダウンロードと検証はSSBLが行います。これにより、コードの重複は最小限に抑えられ、アプリケーションに固有のソフトウェアが簡素化されます。しかし、この場合はSSBLのアップデート（アップデート用のコードのアップデート）を行わなければならないかもしれないという新たな課題が発生します。

結局、どの機能をSSBLに含めるかは、クライアント・デバイスにおけるメモリの制約、ダウンロードされるアプリケーション間の類似性、OTAアップデート用のソフトウェアの移植性に依存することになります。

設計上のトレードオフ：キャッシュと圧縮

OTAアップデート用のソフトウェアを設計するにあたっては、もう1つの重要な検討を行わなければなりません。それは、OTAアップデートのプロセスにおいて、新しいアプリケーションをメモリ内にどのように配置すればよいのかということです。一般に、マイクロコントローラは、不揮発性メモリ（フラッシュ・メモリなど）と揮発性メモリ（SRAMなど）という2種類のメモリを搭載しています。フラッシュ・メモリは、アプリケーション・ソフトウェアのコードや読み取り専用データ、ToC、イベント・ログといったシステム・レベルのデータを格納するために使用されます。一方、SRAMには、固定値ではないグローバル変数やスタックなど、アプリケーション・ソフトウェアにおいて変更が加わる可能性のあるデータが格納されます。図2に示したアプリケーションのバイナリ・データは、すべて不揮発性メモリに格納されます。揮発性メモリに格納される部分は、アプリケーションの起動ルーチンで初期化されます。

OTAアップデートのプロセスにおいて、クライアント・デバイスは、バイナリの一部を含むパケットをサーバーから受信するたびに、それらをSRAMに格納します。このパケットは圧縮されている場合とされていない場合があります。バイナリを圧縮すればサイズが小さくなり、送信されるパケットの数を少なく抑えられます。また、ダウンロードしたデータを格納するためのSRAMの容量も少なくて済みます。一方で、圧縮と展開の処理に時間がかかり、アップデートのプロセスが長くなるというデメリットがあります。また、圧縮に関連するコードをOTAアップデート用のソフトウェアに含めておかなければなりません。

新しいアプリケーションは、最終的にはフラッシュ・メモリに配置されますが、アップデートのプロセスではまずSRAMに格納されます。言い換えると、OTAアップデート用のソフトウェアは、アップデートのプロセスにおけるいずれかの時点でSRAMに格納されたデータをフラッシュ・メモリに書き込む処理を実行する必要があります。新しいアプリケーションを一時的にSRAMに格納する処理のことをキャッシュと呼びます。OTAアップデート用のソフトウェアによるキャッシュに関する手法としては、次の3つが考えられます。

- キャッシュを行わない：新しいアプリケーションの一部を含むパケットが到着するたびに、フラッシュ・メモリの対象となるアドレスにそれらのデータを書き込みます。この方法は非常にシンプルであり、OTAアップデート用のソフトウェアにおけるロジックの量は最小限で済みます。但し、フラッシュ・メモリにおいて、新しいアプリケーション用の領域のデータを完全に消去する必要があります。このことは、フラッシュ・メモリの劣化につながると共に、オーバーヘッドも増加します。
- パーシャル・キャッシュ：キャッシュ用のSRAM領域を用意し、新しいパケットが到着したらその領域に格納します。その領域がいっぱいになったら、格納済みのデータをフラッシュ・メモリに移管して、その領域を空にします。この方法は、パケットが順番どおりに到着せず、新しいアプリケーションのバイナリ・データがとびとびに存在する状態になると、複雑なものになる可能性があります。SRAMのアドレスとフラッシュ・メモリのアドレスをマッピングする手段が必要になるからです。そのための1つの方法は、フラッシュ・メモリの一部分のミラーとしてキャッシュを使うことです。フラッシュ・メモリは、消去の最小単位となるページという小さな領域に分割されます。この自然な分割方法に基づき、フラッシュ・メモリの1ページ分のデータをSRAMにキャッシュします。そして、キャッシュがいっぱいになったとき、または次のパケットが異なるページのものであるときに、そのページのデータをフラッシュ・メモリに書き込みます。その上で、キャッシュのデータを消去します。これは優れた方法だと言えるでしょう。
- フル・キャッシュ：OTAアップデートのプロセスにおいて、新しいアプリケーションの全体をSRAMに格納し、サーバーからのダウンロードが完了した時点で一気にフラッシュ・メモリに書き込みます。この方法では、フラッシュ・メモリへの書き込み回数が最小限に抑えられます。また、OTAアップデート用のソフトウェアのキャッシュ向けロジックが複雑になるのを防ぐことができます。つまり、上記2つの方法の欠点を克服できるということです。但し、通常、システム上で使用できるSRAMの容量はフラッシュ・メモリの使用可能領域の容量よりもはるかに小さいはずです。そのため、ダウンロードできる新しいアプリケーションのサイズが制限されることになります。

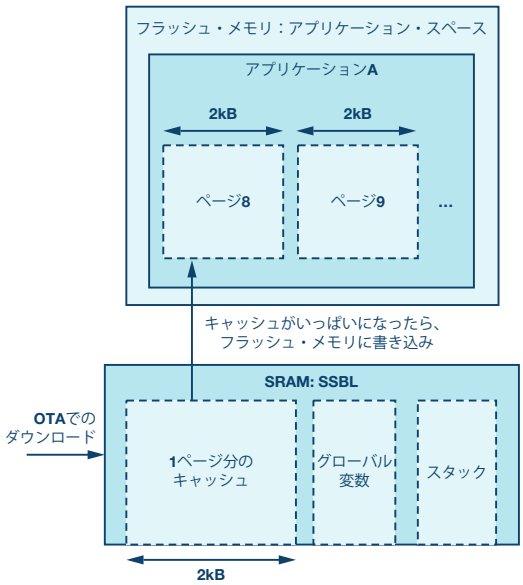


図5. SRAMを使用してフラッシュ・メモリの1ページ分をキャッシュする方法

OTAアップデートを行う際に適用可能なパーシャル・キャッシュの方法はもう1つあります。その概要を図5に示しました。図3と図4におけるアプリケーションA用のフラッシュ・メモリの一部を拡大し、SSBL用のSRAMの機能的なメモリ・マップを示しています。この例では、フラッシュ・メモリのページのサイズを2kBとしています。この部分の設計は、新しいアプリケーションがどれくらいのサイズなのか、またOTAアップデート用のソフトウェアをどこまで複雑にしてもよいのかということに基づいて最終的に決定します。

セキュリティと通信

設計上のトレードオフ：ソフトウェアとプロトコル

OTAアップデート向けのソリューションでは、セキュリティと通信について考慮する必要もあります。図1に示したような多くのシステムでは、センサーによるデータの取得といった（OTAアップデートとは関係のない）システムの通常動作のために、ハードウェアとソフトウェアによって通信プロトコルが実装されます。したがって、サーバーとクライアントの間には、（恐らくはセキュアな）ワイヤレスの通信手段が既に確立されています。図1のような組み込みシステムで使われる通信プロトコルとしては、Bluetooth® Low Energy（BLE）や6LoWPANなどが挙げられます。それらのプロトコルによって、セキュリティとデータ交換がサポートされている場合もあります。そうしたケースでは、OTAアップデート用のソフトウェアにおいて、アップデートのプロセスにそのプロトコルを利用できる可能性があります。

OTAアップデート用のソフトウェアにどれだけの通信機能を実装しなければならないかは、最終的には既存の通信プロトコルによって、どの程度の抽象化が提供されているのかによって決まります。既存の通信プロトコルには、サーバーとクライアントの間でファイルを送受信するための手段が用意されており、OTAアップデート用のソフトウェアでは、それをそのままダウンロード処理に利用できます。但し、通信プロトコルがプリミティブなもので、未処理のデータを送信する手段しか提供していないケースもあります。その場合には、OTAアップデート用のソフトウェアに、パケット化の処理と、新しいアプリケーションのバイナリ・データにメタデータを付加する処理を設けなければならない可能性があります。セキュリティ上の課題についても、同じことが言えます。既存の通信プロトコルでは、機密性を維持するためにワイヤレスで送信されるバイト単位のデータを復号する処理がサポートされていないケースもあるでしょう。その場合には、OTAアップデート用のソフトウェアで復号処理を実施しなければならないかもしれません。

結論として、カスタムのパケット構造への対応、サーバーとクライアントの同期、暗号化、鍵の交換といった機能をOTAアップデート用のソフトウェアに実装するかどうかは、システムが採用している通信プロトコルの機能と、セキュリティや堅牢性に関する要件によって決まります。次のセクションでは、上述したすべての課題を解決する完全なセキュリティ・ソリューションを提案します。また、そのソリューションにおいて、マイクロコントローラが備える暗号化用のペリフェラルを活用する方法も示します。

セキュリティに関する課題の解決

セキュリティに関するソリューションでは、以下のような事柄を網羅する必要があります。まず、ワイヤレスで送信される新しいアプリケーションの機密性を維持しなければなりません。また、新しいアプリケーションに破損があれば、それを検出する必要があります。更に、新しいアプリケーションの送信元が悪意ある第三者ではなく、信頼できるサーバーであることを確認しなければなりません。これらの課題は、暗号化処理によって解決することができます。具体的には、暗号化とハッシュという2つの処理をセキュリティ向けのソリューションに盛り込みます。ここで言う暗号化とは、クライアントとサーバーで共有鍵（パスワード）を使用し、ワイヤレスで送受信されるデータを読み取りにくくする処理のことです。マイクロコントローラの中には、ハードウェア・アクセラレータによって、AES-128またはAES-256の暗号化（両者の違いは鍵のサイズです）をサポートしているものがあります。サーバーは、暗号化されたデータと共に、破損がないことを保証するためのダイジェストのデータを送信することができます。ダイジェストのデータは、データ・パケットに対するハッシュ処理によって、生成されます。ハッシュ処理とは、不可逆の算術関数によって、一意的なコードを生成するというものです。ワイヤレス通信中にいずれかのビットのデータが反転するなど、サーバーによって生成された後に、メッセージまたはダイジェストの一部が変化してしまうことがあります。クライアント側では、データ・パケットにサーバー側で使用したのと同じハッシュ関数を適用し、ダイジェストを生成します。クライアントで両ダイジェストを比較することにより、問題が発生していればそれを検出できるということです。マイクロコントローラの中には、暗号化用のハードウェア・アクセラレータによって、SHA-256のハッシュ関数をサポートしているものがあります。図6に、マイクロコントローラが備える暗号化用ペリフェラルのブロック図を示しました。OTAアップデート用のソフトウェアは、「Cortex-M4」のアプリケーション・レイヤに存在します。この図には、ペリフェラルに格納された鍵を保護するための機構も描かれています。それを利用することにより、OTAアップデート用のソフトウェアは、クライアントの鍵を安全に格納します。

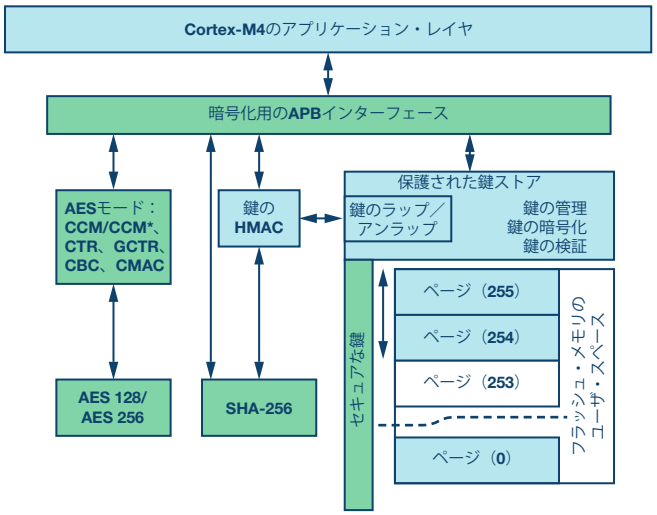


図6. ADuCM4050が備える暗号化用
ハードウェア・アクセラレータのブロック図

解決すべき最後の課題は認証です。そのための一般的な手法は、非対称の暗号化を使用することです。この処理を利用する場合、サーバー側では、公開鍵と秘密鍵のペアを生成することになります。秘密鍵の内容を把握しているのはサーバーだけです。一方の公開鍵は、クライアントに渡されます。サーバーは秘密鍵を使って、特定のデータ・ブロックに対する署名データを生成します。これはOTAで送信されるパケットのダイジェストのようなものです。クライアントは送信されてきた署名データに対し、公開鍵を使って検証を行います。それによって、クライアントは、そのメッセージは悪意ある第三者ではなく、信頼できるサーバーから送信されたものであることを確認できます。図7に、こうした一連の処理の流れを示しました。実線の矢印は各機能に対する入出力を表し、点線の矢印はOTAで送信される情報を表しています。

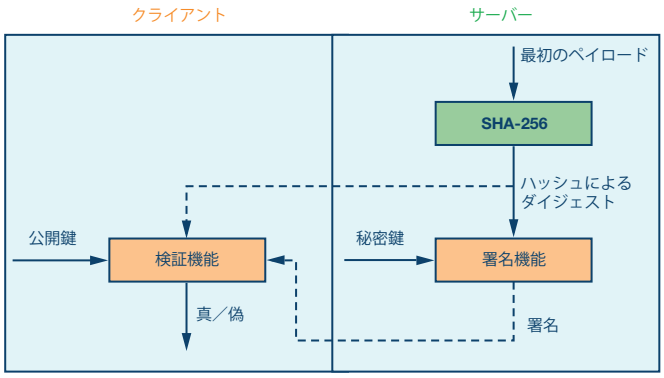
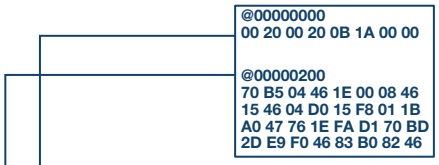


図7. 非対称暗号化による
メッセージの認証

非対称暗号化に向けたハードウェア・アクセラレータを備えるマイクロコントローラは、ほとんど存在しません。ただ、micro-eccなどのソフトウェア・ライブラリを使用することで、その機能を実装することは可能です。micro-eccは、特にリソースに制約のあるデバイスをターゲットにしています。このライブラリには、ユーザが定義した乱数生成用の関数が必要です。これについては、マイクロコントローラが備える真性乱数生成器（ペリフェラル）を使用することで実装可能です。非対称暗号化は、OTAアップデートを実施する際の信頼性の問題を解決します。但し、処理に時間がかかることに加え、データと共に署名も送信しなければならないため、パケットのサイズが大きくなるという問題があります。ダウンロードの完了時に、最終パケットのダイジェストか、新しいソフトウェア・アプリケーション全体のダイジェストを使用して、一度だけチェックを実行するという方法も考えられます。ただ、それでは、第三者が不正なソフトウェアをクライアントにダウンロードするということが行えるので、理想的な方法だとは言えません。理想的な方法は、署名に関するオーバーヘッドを毎回生じさせることなく、受信する個々のパケットが信頼できるサーバーからのものであることを確認できるようにすることです。これは、ハッシュ・チェーンによって実現可能です。

ハッシュ・チェーンは、ここまでに説明してきた暗号化の概念を一連のパケットに組み込み、それらを数学的にまとめるものです。図8に示すように、最初のパケット（番号0）には、次のパケットのダイジェストが含まれています。最初のパケットのペイロードは、実際のソフトウェア・アプリケーションのデータではなく、署名です。2番目のパケット（番号1）のペイロードには、バイナリ・データの一部と3番目のパケット（番号2）のダイジェストが含まれています。クライアントは、最初のパケットの署名を確認し、ダイジェストH0を後で使うためにキャッシュします。2番目のパケットが到着すると、クライアントはペイロードをキャッシュし、H0と比較します。それらが一致すれば、この後続のパケットは信頼できるサーバーから送られてきたものであると確認できます。そのため、署名のチェックを行うためのオーバーヘッドをすべて省くことが可能です。これらのチェーンの生成は、多くの演算量を要するタスクです。これについてはサーバーが担います。クライアントはキャッシュとハッシュを実行するだけで、到着した各パケットが破損していないこと、完全であること、認証されていることを確認できます。



番号	ペイロード	ハッシュ
0	P0 = 署名(H0, 公開鍵)	H0 = H(P1 + H1)
1	P1 = 00 20 00 20 0B 1A 00 00	H1 = H(P2 + H2)
2	P2 = 70 B5 04 46 1E 00 08 46	H2 = H(P3 + H3)
n	Pn = …	Hn = H(Pn+1 + Hn+1)

図8. パケット・シーケンスへの
ハッシュ・チェーンの適用

実験用の設定

以下では、メモリ、通信、セキュリティ関連の設計上の課題を解決するものとして、超低消費電力のマイクロコントローラ「ADuCM3029」と「ADuCM4050」を紹介します。これらの製品は、フラッシュ・メモリ、SRAM、暗号化用のアクセラレータ、真性乱数生成器など、本稿で説明してきたOTAアップデート用のペリフェラルを備えています。また両製品のデバイス・ファミリ・パック（DFP）では、OTAアップデート向けのソリューションを構築するためのソフトウェア・サポートを提供しています。加えて、DFPにはシンプルで柔軟なインターフェースを備えたペリフェラル用のドライバが含まれています。

ハードウェアの構成

本稿で説明した概念の検証と確認を行うために、ADuCM4050をベースとするOTAアップデート用ソフトウェアのリファレンス・デザインを開発しました。クライアントとしては、トランシーバー用ドーターボードのU字型コネクタを使用し、トランシーバーIC「ADF7242」に評価用ボード「ADuCM4050 EZ-KIT®」を接続しました。図9の左側にあるのがクライアント・デバイスです。サーバーとしては、Windows PC上で動作するアプリケーションを、Pythonを使って開発しました。このPythonアプリケーションは、シリアル・ポートを介して別のADuCM4050 EZ-KITと通信します。そのADuCM4050 EZ-KITにも、クライアントと同じようにADF7242が接続されています。但し、図9の右側のADuCM4050 EZ-KITはOTAアップデートは実行しません。ADF7242から受信したパケットをPythonアプリケーションに中継するだけです。

図9. 実験用のハードウェア構成

ソフトウェア・コンポーネント

ソフトウェアのリファレンス・デザインでは、クライアント・デバイスのフラッシュ・メモリを図3に示したように分割しました。メインのクライアント・アプリケーションは、他の構成や他のハードウェア・プラットフォームでも使用できるように、構成が可能（コンフィギュラブル）で移植性を備えるものとして設計しました。図10に、クライアント・デバイスのソフトウェアのアーキテクチャを示しました。このアプリケーション全体をSSBLと呼ぶこともできますが、ここでは、まさにSSBLに当たる部分（紫色）とOTAアップデート用のソフトウェアの部分（橙色）を論理的に区別することにします。先述したように、後者は必ずしも同じアプリケーション内に実装する必要はないからです。図10に描かれているハードウェア抽象化レイヤによって、OTAに対応するクライアント・ソフトウェアは移植性を得ることができています。また、基盤にあるどのライブラリ（青色）にも依存しません。

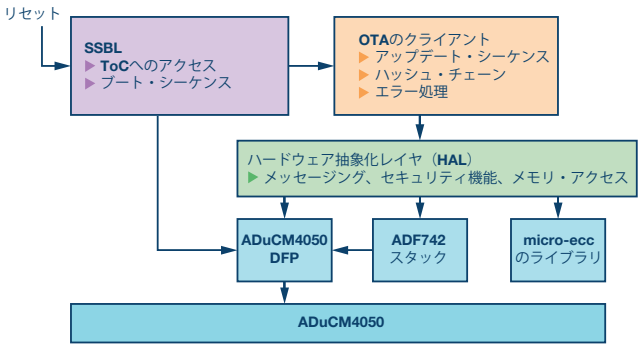


図10. クライアントのソフトウェアのアーキテクチャ

このソフトウェア・アプリケーションには、図3のブート・シーケンス、新しいアプリケーションをサーバーからダウンロードするためのシンプルな通信プロトコル、ハッシュ・チェーンが実装されています。通信プロトコルでは、各パケットが、12バイトのメタデータのヘッダ、64バイトのペイロード、32バイトのダイジェストで構成されると定められています。加えて、以下のような機能を備えます。

- キャッシュ：ユーザが定めた構成に応じ、キャッシュしない方法と、フラッシュ・メモリの1ページをキャッシュする方法のうちどちらかを使用できます。
- ToC：ToCは、2つのアプリケーションだけを保持するように設計されています。フォールバック用のアプリケーションを残しておくために、新しいアプリケーションは必ず古い方の領域にダウンロードされます。これをA/Bアップデート方式と呼びます。
- メッセージング：ユーザが定めた構成に応じ、メッセージング用にADF7242かUARTをサポートします。UARTでメッセージを送受信する場合、図9の左側のADuCM4050 EZ-KITは必要なく、右側のクライアント用キットだけを使用します。この有線のアップデート方式は、システムを初めて起動してデバッグする際に重宝します。

実験の結果

システムとしての機能的な要件を満たし、様々なテストに合格させるのはもちろん大事なことです。それ以外に、ソフトウェアの性能もプロジェクトの成否を左右する重要な要素となります。組み込みソフトウェアの性能を評価するために一般的に使用されるのは、フットプリントとサイクルという2つの基準です。ここで言うフットプリントとは、ソフトウェア・アプリケーションが消費する揮発性メモリ（SRAM）と不揮発性メモリ（フラッシュ・メモリ）の容量のことです。一方のサイクルとは、ソフトウェアが特定のタスクを実行するために使用するマイクロプロセッサのクロック・サイクル数のことを指します。これは、一見するとソフトウェアの実行時間に似ています。ただ、ここではOTAアップデートの実行時に低消費電力モードに入り、マイクロプロセッサが動作を停止してサイクルが消費されない可能性があることも考慮しています。実は、このソフトウェアのリファレンス・デザインは、どちらの評価基準に対しても最適化されていません。それでも、プログラムのベンチマーク測定に使用できますし、設計上のトレードオフについて比較検討する上でも役に立ちます。

図11と図12は、このリファレンス・デザインのフットプリントを示したものです。ADuCM4050を使用し、キャッシュを行わないという条件で実装されています。これらのグラフは、図10の各コンポーネントに対応するフットプリントを表しています。図11に示すように、アプリケーション全体で約15kBのフラッシュ・メモリを使用します。ADuCM4050のフラッシュ・メモリが512kBであることを考えると、かなり少なく抑えられていると言えます。しかも、実際のアプリケーション・ソフトウェア（OTAアップデートに関連するソフトウェア）のフットプリントは、約1.5kBしかありません。残りはDFP、micro-ecc、ADF7242のスタックといったライブラリによって使用されています。この結果は、システムにおけるSSBLの役割を考慮し、設計上のトレードオフについて検討する際に役立ちます。15kBのフットプリントの大部分は、アップデートのプロセスを実現するために使用されます。SSBLそのものは約0.5kBしかなく、フラッシュ・メモリ用のドライバなどのようにデバイスにアクセスするためのものとして、DFPのコードが1kB～2kBほど追加されています。

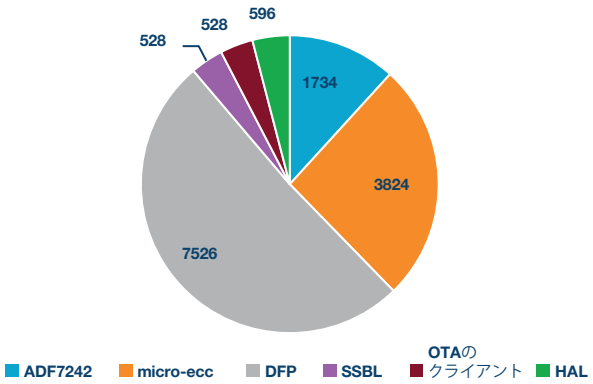


図11. フラッシュ・メモリ上のフットプリント（単位はバイト）

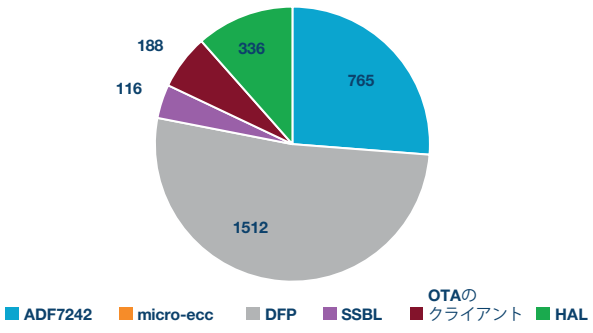


図12. SRAM上のフットプリント（単位はバイト）

ソフトウェアのオーバーヘッドについて評価するために、データ・パケットを受信するたびにサイクルをカウントし、パケット当たりの平均消費サイクル数を調べました。各データ・パケットに対しては、AES-128の復号、SHA-256のハッシュ、フラッシュ・メモリへの書き込み、パケットの一部のメタデータに対する検証が必要になります。パケットのペイロードのサイズが64バイトで、キャッシュを行わない場合、1つのデータ・パケットを処理するために生じるオーバーヘッドは7409サイクルでした。CPUコアを26MHzのクロックで動作させた場合、処理時間は約285マイクロ秒になります。この値はADuCM4050のDFPに含まれるサイクル計数ドライバを使用して計算しました（非調整サイクル）。そして、100kBのバイナリ・データをダウンロード（約1500パケット）している間の平均値を求めました。パケット当たりのオーバーヘッドが最小になるのは、DFPのドライバが、バス・トランザクションの実行時にADuCM4050上のDMA用ペリフェラルを使用し、プロセッサを低消費電力のスリープ状態に移行させたときだと考えられます。DFPで低消費電力のスリープ状態に移行しないようにし、バス・トランザクションでDMAを使用しないように変更すると、データ・パケット当たりのオーバーヘッドは1万7297サイクルに増加します。これは、デバイス・ドライバを効率的に使用することが、組み込みソフトウェア・アプリケーションにどのような効果をもたらすのかということを表しています。パケット当たりのデータのバイト数を小さくすることでも、オーバーヘッドは抑えられます。しかし、パケット当たりのバイト数を2倍の128にしても、同じ実験におけるサイクル数は8362となり、それほど大きく増加することはありませんでした。

サイクルとフットプリントには、フラッシュ・メモリに毎回書き込むか、それともパケット・データをキャッシュするかというトレードオフの影響も現れます。フラッシュ・メモリの1ページをキャッシュする場合、データ・パケット当たりのオーバーヘッドは7409サイクルから5904サイクルに減少します。約20%も減少しているわけですが、これは、キャッシュがいっぱいになったときだけフラッシュ・メモリへの書き込みが行われ、ほとんどのパケットではフラッシュ・メモリへの書き込みが不要になることによるものです。しかし、その代償として、SRAMのフットプリントが増加します。キャッシュを行わない場合、HALに必要なSRAMは、図12に示すようにわずか336バイトです。ただ、キャッシュを使用すると、フラッシュ・メモリのページと同じ大きさの領域を用意しなければならないので、必要なSRAMの容量は2388バイトに増加します。キャッシュを消去するタイミングを判断するための追加のコードが必要になるため、HALに必要なフラッシュ・メモリの容量も少し増加します。

以上の結果は、設計を行う際に下した判断が、ソフトウェアの性能に目に見える影響を与えるということを表しています。どのような状況にも対応できる万能のソリューションなどというものは存在しません。OTAアップデート用のソフトウェアについても、システムごとに異なる要件や制約に応じてチューニングする必要があります。OTAアップデート用のソフトウェア・ソリューションの設計、実装、検証時に遭遇する一般的な問題やトレードオフへの対処が必要になった際には、ぜひ本稿の内容をお役立てください。

参考資料

Nilsson、Dennis Kengo、Ulf E. Larson「Secure Firmware Updates over the Air in Intelligent Vehicles（インテリジェントな車両において、OTAによりセキュアにファームウェアをアップデート）」ICC Workshops、2008 IEEE International Conference on Communications Workshops、2008年5月

著者：

Benjamin Bucklin Brown（benjamin-b.brown@analog.com）は、マギル大学で電気工学の学士号を取得し、2016年にアナログ・デバイセズに入社しました。現在は、民生機器向けのセンシング技術／プロセッシング技術（CSPT）グループで組み込みソフトウェア技術者として業務に携わっています。担当しているのは、ASIC用ファームウェアの開発です。それ以前は、IoTプラットフォーム技術グループに所属し、マイクロコントローラ（ADuCM3029、ADuCM4050）用のデバイス・ドライバやソフトウェア・リファレンス・アプリケーションを開発していました。

Benjamin Bucklin Brown