インテリジェントローダー

～京都マイクロコンピュータが提供する、次世代のローダー～

【第一回】こんなローダーが欲しかった！

SOLID 開発プラットフォームの中の新兵器、それがインテリジェントローダーです。その名のとおり、「かしこく」それでいて縁の下の力持ち的なローダーが、組込みソフトウエアの開発スタイルを大きく変えようとしています。本書では 3 回にわたって、インテリジェントローダーを詳しく解説していきます。

第一回目は、「こんなローダーが欲しかった！」インテリジェントローダーとは何か、をご紹介します。

なお、第二回目は、SOLID の API を使って具体的にロードする方法を、第三回目ではインテリジェントローダーの応用的な使い方や、動作の仕組みについてご紹介予定です。

インテリジェントローダーが生まれたきっかけ

お客様の「困った」を何とかしたい

ソフトウエアの規模が大きくなると、複数メンバーによるチーム開発だけでなく、協力会社に委託をして共同開発するというスタイルが当たり前になっています。しかし、分散開発されたプログラムを実行・デバッグするには、一旦全てのソースコードを集結してビルド作業を行い、実行モジュールを生成する必要があります。

このような場面で、実際にソフトウエアを開発しているお客様から次のようなお困り事を聞くことが多くあります。

・ ビルド時間、ロード時間が長く、開発効率が悪い。何より、開発エンジニアの思考が中断されるのが困る。

・ 協力会社さんとの共同開発では、権利関係によってソースコードを開示できないケースがある。そのため、ビルド作業大変複雑になっている。

・ Linux のようなマルチスレッド機能を使いたいが、メモリ容量は節約したい。大容量メモリである DDR は電源回路コストが高いので使いたくない。

京都マイクロコンピュータでは、このようなお客様の不便を解消するため、「Linux における"ファイルからアプリケーションを実行する" という操作をリアルタイム OS 向けアプリケーションで使うにはどうすればいいか」を、お客様と共に考えてきました。

そのような背景の元で生まれたのが SOLID インテリジェントローダーです。

インテリジェントローダーの仕様

組込みソフトウエアにおける一般的なローダーは、開発環境でビルドされたプログラムをターゲット実機で実行するため、実メモリ上に実行モジュールを転送します。

SOLID インテリジェントローダーの大きな特徴としては、ソフトウエアを分割開発した場合に、分割されたアプリケーション単位で個別にビルドされた実行モジュールを、個別にロードする機能を持つ点です。

1) 分割開発されたアプリケーション単位で個別にビルドできる
2) 分割開発されたアプリケーションを個別にロードし、他のアプリケーションと結合して実行できる
3) 他のアプリケーションのソースコードが無くても、分割開発した当該アプリケーションのソースコードがあれば、プログラム全体のデバッグ操作ができる
4) 指定したアプリケーションのアンロードが可能

上記の仕様にすることにより、分散拠点で独立性の高い開発を進められるという大きなメリットがあるだけでなく、開発中のソフトウエアを部分的に更新する場合のビルドおよびロード時間を大幅に短縮する事もできます。

インテリジェントローダーを使ってできること

モジュール単体ビルド･ロードの便利な使い方

ソフトウエア規模が大きく、複数のエンジニアや拠点で分割開発する例として、図 1-1 の様に開発会社（とりまとめ）が 2 拠点の開発協力会社に機能単位でプログラム開発を分担する場合を考えてみます。

開発会社はソフトウエア全体の制御を行う部分（ハードウエアの初期化、MMU を利用したメモリ空間全体構成の設定や RTOS カーネルなど）を担当します。ここではこの部分を「本体アプリ」と呼ぶことにします。

開発会社は、協力会社 1 に、機器ハードウエアの自己診断プログラムを開発委託することにします（アプリ１）。また協力会社 2 には、同社の持つノウハウを組み込んだデータ処理プログラム（アプリ２）を開発委託することにします。アプリ 1、アプリ 2 ともに比較的本体アプリと切り離して開発が進められる例です。

（本体および各アプリ間で共有するグローバル変数や関数名を指定する方法は第二回目で説明します）

図 1-1 開発協力会社がアプリケーション単位でプログラムを分担して開発

ソースコードを開示する必要がない

このような体制で開発する場合、本体アプリはROM 等に書き込まれた実行モジュールだけがあればよく、ソースコードを協力会社に提供しなくても問題ありません(図 1-2)。SOLID 環境で開発されたプログラムであれば、本体アプリのソースコード無しでも、開発を分担する協力会社 1、協力会社 2 は各々の担当アプリケーションを単体で開発し、ビルド・ロードを行い、メインモジュールと一緒にデバッグできます。

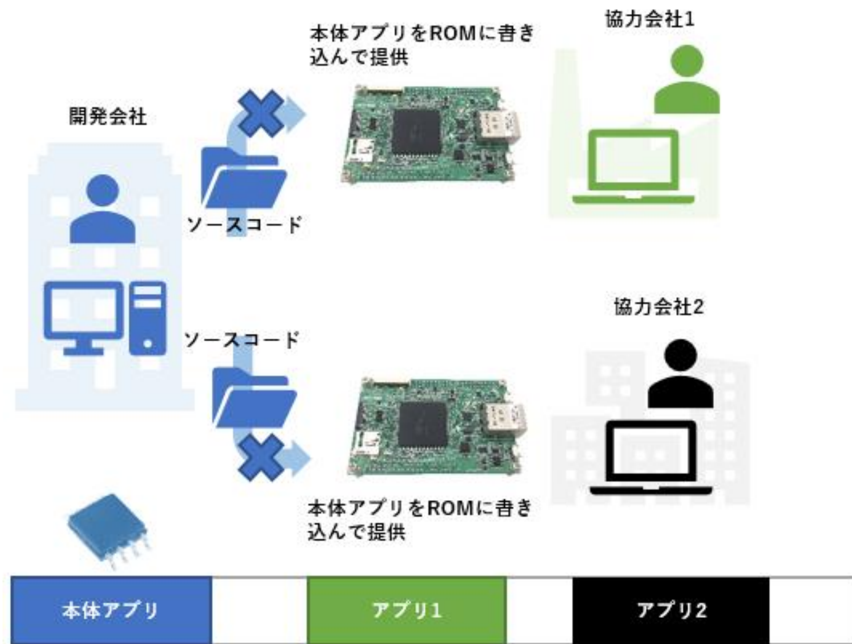

図 1-2. ソースコード無しで開発を分担

またこれとは逆に、協力会社がソースコードを開示することなく、開発会社（とりまとめ）に実行モジュールのみを提供するだけで、本体アプリを単体で更新してビルド・ロードい、開発会社側でプログラム全体を動作させることが出来ます(図 1-3)。

実機デバッグにおいては、ソースコードが存在する部分はソースコードデバッグが、ソースコードが無い部分はアセンブラレベルでのデバッグが出来ます。もちろんソースコードが無い部分であっても、ブレークポイントを設定してプログラムの動作確認ができます。

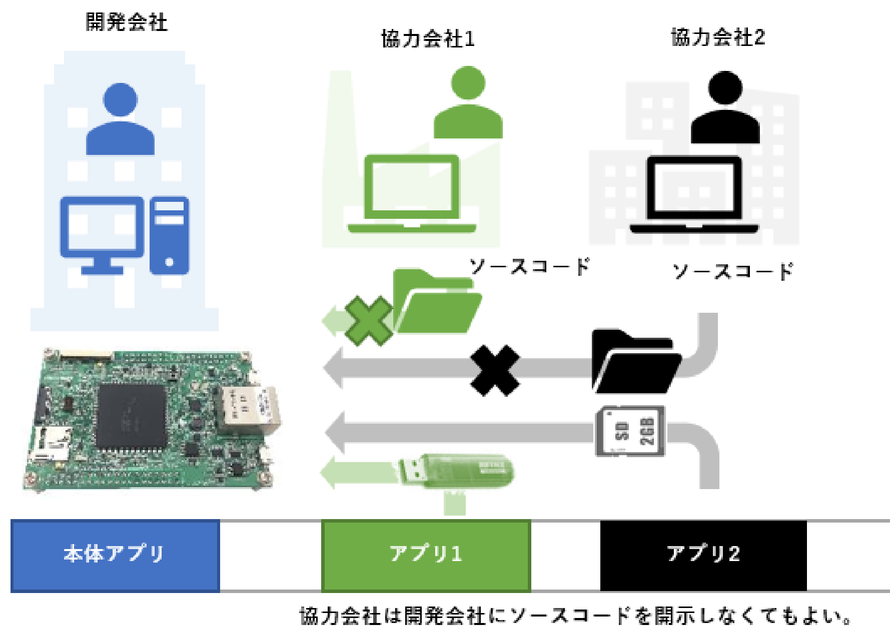

図 1-3. 協力会社がソースコードを開示しなくても各社がビルド・ロード・デバッグ可能

SOLID の開発対象は RTOS システムのようなリアルタイム制御システムなので、全てのアプリは単一の論理空間に配置されます。物理アドレス上も固定配置されるため、一度ロードしたプログラムはユーザーの意図しないタイミングでアンロードしたり、アドレスが移動する事はありません。これも Linux のマルチスレッド動作などと SOLID が大きく異なる点であり、特長でもあります。

（第二回目につづく）

インテリジェントローダー

～京都マイクロコンピュータが提供する、次世代のローダー～

【第二回】ローダーの使い方

第二回目では、SOLID の API を使って具体的に分割ロードする方法をご紹介します。

ローダーを使ったプログラムの作り方

本体アプリとサブアプリに機能分割する

まず通常の分割開発と同様に、機能や目的に応じて開発対象の分割方法を決めます。このときプログラムの基本的な制御を行う部分を「本体アプリ」と定義し、SOLID で開発します。

本体アプリは、プログラム全体に必ず 1 つだけ存在するようにし、RTOS カーネルもこの本体アプリ内に配置することをお勧めします。

次に SOLID 環境で分割されたアプリケーションである「サブアプリ」を開発します。サブアプリには SOLID コアサービスや RTOS を含む必要はありません。またサブアプリの数は 0～N 個と、柔軟に分割可能です。

シンボルを共有する

本体アプリとサブアプリ間では、通常グローバル変数や関数を共有する必要がありますので、SOLID においても同様の処理が必要です。

SOLID においてシンボルを共有する方法は 2 つあります。1 つはビルド時に共有シンボル定義ファイル（テキストファイル）を参照する方法で、もうひとつはプログラム内に API を使って定義する方法です。

まず、グローバル変数や共有する関数名などのシンボル情報を定義します。SOLID では、これらのシンボを他のアプリで利用するため、

Export シンボル：
　他のアプリにシンボルを利用させる

Import シンボル：
　他のアプリのシンボルを利用する

という 2 つの API を用意しています。

図 2-1．では、本体アプリが API で Export 指定したシンボルをサブアプリ(アプリ 1、アプリ 2)がテキストファイルとしてビルドにより共有する方法を示しています。

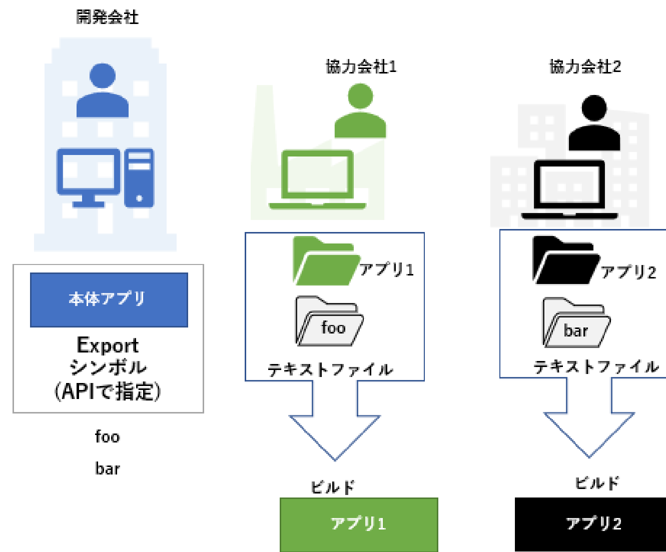

図 2-1 本体アプリのシンボルをサブアプリで共有する（テキストファイルを利用）

サブアプリ(アプリ 1、アプリ 2)が本体アプリのシンボルを Import するには、図 2-2.のようにサブアプリ内で Import シンボル API を使って直接シンボルを指定することも可能です。

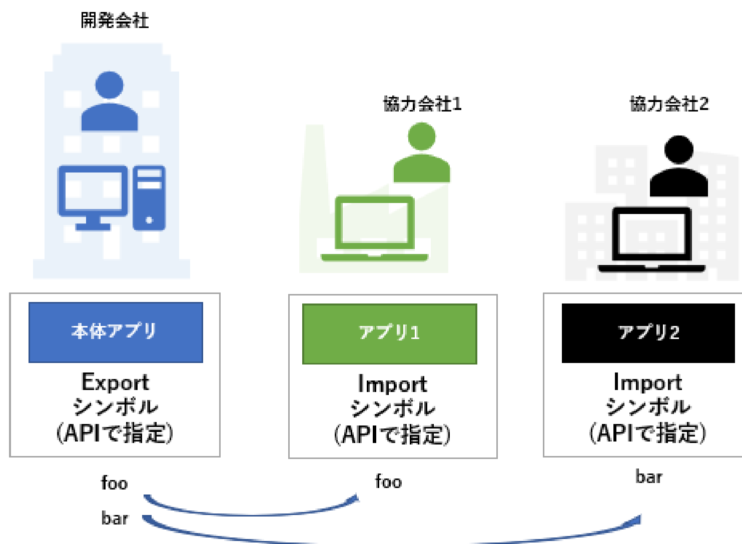

図 2-2. 本体アプリのシンボルをサブアプリで共有する（API を利用）

これとは逆に、サブアプリ内にあるシンボルを本体アプリや他のアプリが利用する場合は、図 2-3 や図 2-4 のように定義します。

シンボルの共有はサブアプリ間においても同様の手順で定義できます。

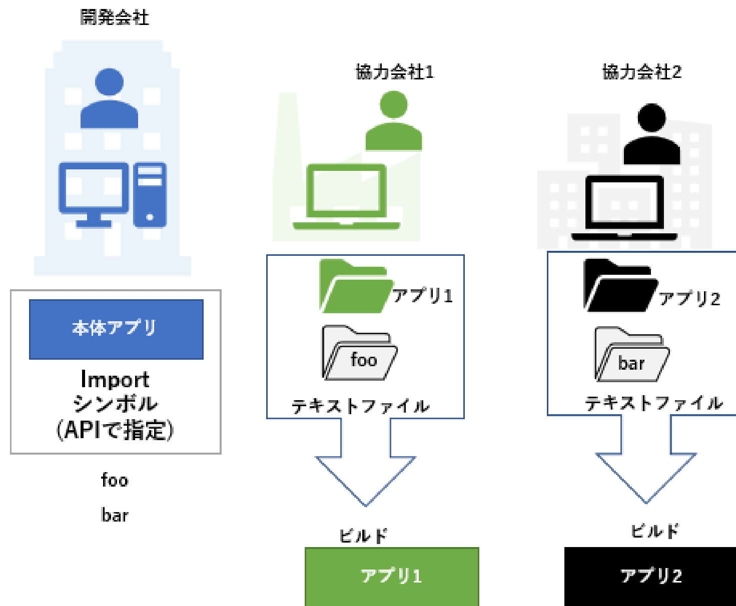

図 2-3　サブアプリのシンボルを本体アプリで共有する（テキストファイルを利用）

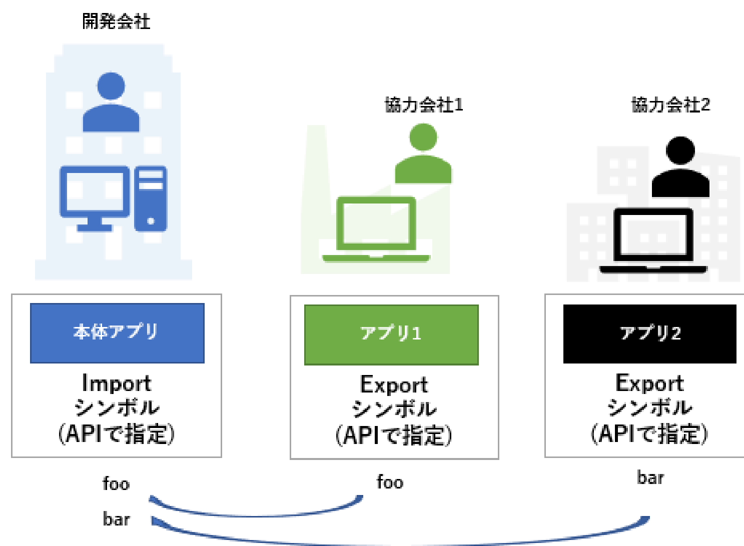

図 2-4　サブアプリのシンボルを本体アプリで共有する（API を利用）

以上のように、SOLID では API を使ってシンボル共有を定義します。API で定義されることで、SOLID ではアプリ間のシンボル解決をするための処理をランタイムソフトウエアとして生成し、その実体を本体アプリ内に常駐させます。SOLID では、このようなアプリ内に常駐する管理用のランタ

イムプログラムをコアサービスと名付けています（コアサービスは京都マイクロコンピュータの造語で、ロード以外に MMU の管理などの機能も持っています。詳しくは第三回目でご紹介します）。

コアサービス内にあるインテリジェントローダーは、本体プログラムを実行した際に、あたかもリンカが行っているようなアドレス解決を行いながら、メモリ上に実行モジュールを配置していく機能を持っています

インテリジェントローダーを使って分割開発されたアプリを合体して実行する

シンボル共有定義ができた「本体アプリ」「サブアプリ(アプリ 1、アプリ 2)が各開発拠点でビルドできたところで、それらのアプリケーションを結合して実行するまでの流れを紹介します。

ここで SOLID が用意したのが、本体アプリがサブアプリをロードするための API です。

ローディング API
　[SOLID_LDR_LoadFile()]：
　実行モジュールが存在するファイルパスを指定し、メモリ上にプログラムをロードします。この時点ではアプリ間の共有変数など相互参照しているアドレスは未解決の状態なので、まだアプリの実行は出来ません。

実行可能チェック API
　[SOLID_LDR_CanExec()]：
　ローディング API でロードすると決めたアプリが実行可能なのか（アドレス解決が矛盾なくできているか）をチェックし、問題無く実行できるならば True を返します。

図 2-5. でその手順を説明します。

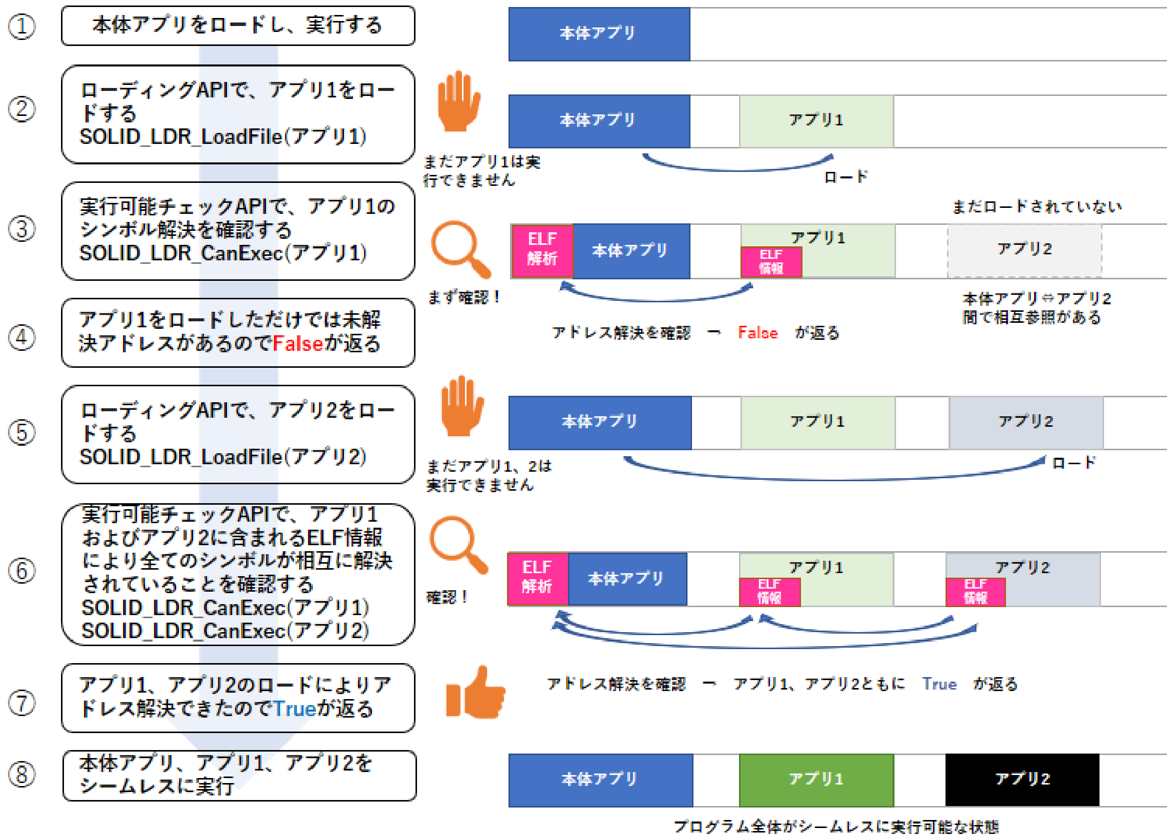

図 2-5　アプリを結合して実行するまでの流れ

① 　まず、本体アプリをロードします。

② 本単アプリから、アプリ 1 をロードします。アプリ 1 は USB メモリやシリアルフラッシュメモリなど、ファイルシステム上に予めコピーしておきます。

③ プログラムが、実行可能な状態かどうかを確認します。

④ しかし、アプリ 2 でも相互参照しているシンボルがあるため、この段階ではシンボル解決できておらず、実行できないという結果(False)が返されます。

⑤ 次にアプリ 2 をロードします。

⑥ 再度、アプリ 1、アプリ 2 が実行可能な状態かどうかを確認します。

⑦ 今度は全てのシンボルの相互参照が確認できたため、シンボル解決が完了したこと(True)が返されます。

⑧ これ以降、本体アプリ、アプリ 1、アプリ 2 は定義された相互参照シンボルを使ってシームレスに実行できるようになります。

（図中 ELF 解析、ELF 情報　という記載がありますが、こちらについては第三回目で説明します

（第三回目につづく）

インテリジェントローダー

～京都マイクロコンピュータが提供する、次世代のローダー～

【第三回】ローダーの応用例と動作の仕組み

最終回となる第三回では、これまでご紹介してきたローダーの応用的な使い方や動作の仕組みについて解説します。また末尾では、インテリジェントローダー開発者からローダーが生まれた背景などもご紹介しています。

ローダーの応用例

バージョン管理もシンプルに効率よく

組込み機器の場合、仕向け先によってプログラムの一部を変更して出荷する事があります。固定のデータ領域にパラメータだけを置き換えるような場合は問題ありませんが、きめ細かい仕向け先対応を行う場合は、プログラムそのものを変更する必要が生じます。

SOLID を使ってプログラム開発をすれば、図 3-1 のように仕向け先別に異なるアプリを各々個別に開発し、その後共通部分と組合せて実行が可能です。

このとき、各アプリのプログラムサイズが異なる場合、配置が異なる場合でも問題なく本体アプリから各アプリをロードでき、通常のアプリケーションのデバッグと同様に実機を使ったソースコードデバッグができます。

もちろん拠点毎のアップデートに際しても、当該拠点のアプリだけを改訂すればよいのでソフトウエアのバージョンアップや検証範囲が限定でき、システム全体の管理が大変シンプルに効率よく行えます。

図 3-1 仕向け先別にアプリを分けて開発する

このようなモジュール分割開発ができないと、図 3-2 のように仕向け先のアプリを全て込んでビルドしているケースがあるのではないでしょうか。

これでは全体のプログラムサイズが大きくなりビルド・ロード時間が長くなってしまうばかりでなく、本来は独立で管理できるはずの拠点単位のアプリケーション(アプリ A、B・・・)の全てのバージョン組み合わせ管理の必要が生じるなど開発効率があまり良いとは言えません。

図 3-2 残念なプログラム配置

インテリジェントローダーの仕組み

コアサービスという縁の下の力持ち

SOLID では「コアサービス」と名付けた一連のランタイムが IDE と密接に連携することで、特長的な機能の多くを実現しています。インテリジェントローダーもコアサービスに実装されている機能のひとつです。

コアサービスには、ローダーの他にターゲットのリソース(メモリ/MMU, 割り込み, タイマ等)の管理と、ユーザーへの補助(デバッグ）といった機能があります。

コアサービスはターゲットシステムの一部に常駐しており、システムセットアップや動的解析機能を使ったデバッグ時など、必要に応じて IDE とも連携して動作するソフトウエアです。 少し古いですが「BIOS のようなもの」と言えばイメージがわきやすいかと思います。ソフトウエアの構造としては、第二回目の「ローダーの使い方」でもご紹介したようにシンプルな API 群を使ってリソースを操作するものとなっています。

またコアサービスでは、ブート時の MMU の仮想アドレスへの移行といった、ユーザーが操作するのが面倒な処理も受け持っています。

このような SOLID コアサービスの機能には、従来の開発環境やデバッグツールにおいては、PC 側で処理を行っていたものもあります。「使いやすさ」「リアルタイム」「コンパクト」を追求していく中で、ユーザーが「スマート」に開発できるよう「シンプル」な仕様として出来上がったのがランタイムという形でのコアサービスです。

ちなみに、コアサービスは、京都マイクロコンピュータの造語です。縁の下の力持ちという役割であり、SOLID には無くてはならない重要な存在です。

ロード対象ファイルに含まれる情報は？

SOLID ではツールチェーン(Clang/GCC)でビルド操作を行い実行バイナリのオブジェクトフォーマットである ELF フォーマットでアプリケーションの実行モジュールを生成します。

この ELF フォーマットの実行モジュールには、リロケータブル可能なプログラム本体と、ELF 情報としてロード処理に必要なシンボル情報および再配置情報が含まれています。SOLID で分割ロードをする場合は、図 3-3 のようにコアサービス内にある ELF 解析機構がロード対象の実行モジュール内の ELF 情報を参照しながら、シンボル解決を行います。

なお、この実行モジュールには、ソースコードのあるファイルのルートパス情報や、ソースコード内の行番号情報といったソース情報は含まれていません。

シンボル情報に相当する ELF 情報を含む SOLID 実行モジュールは、それを含まないモジュールに比べて一般的に約 10～20%程度ファイルサイズが大きくなります。

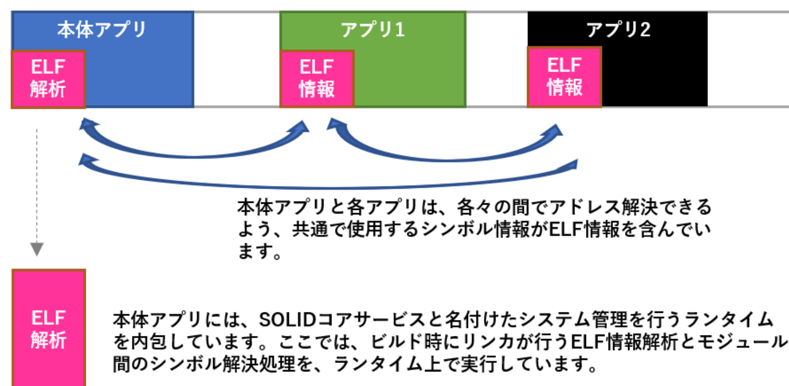

図 3-3 ロード対象ファイルは ELF 情報を含む

ローダーが行っている処理

インテリジェントローダー本体は SOLID で本体アプリをビルドする際に本体アプリ内にコアサービスとして生成されます。

第二回目で紹介した以下の 2 つの API によりインテリジェントローダーが動作します。

ローディング API
[SOLID_LDR_LoadFile()]：

実行可能チェック API
[SOLID_LDR_CanExec()]

これらの API がコールされるとコアサービスに制御が移り、ファイルシステム経由で実行モジュールを読み込み、シンボル情報を解析してアドレス配置を決定し(MMU の設定を含む)、物理メモリにプログラムをロードします。

SOLID では、コアサービスを含む本体アプリは必ず 1 つ必要です。またサブアプリは 0～N 個とその数に制限はありません。

シンボルの相互参照はサブアプリ間であっても有効であり、ローディング API や実行可能チェック API を実行した際に、コアサービスはサブアプリ間のシンボルの相互参照についても解決処理を行います。

なお、インテリジェントローダーでは実行プログラムを RAM 上に配置することを前提としているため、ROM 上やファイルシステム内のプログラムを直接実行することは前提としていません（なぜなら、シンボル解決によるアドレスの再配置ができないからです）。

MMU によるメモリ管理と密接に連携するインテリジェントローダー

上記では、分割ロードをするためにはアプリ相互の共有シンボルや変数などのシンボル解決をランタイム内で処理することをしていることを説明しました。ファイルシステム上にあるデータ（実行モジュール）を解析し、アドレス解決をしながら実メモリ（物理アドレス）にプログラムをロードする際には、MMU を有効活用することでメモリを効率よく利用できるようになります。

SOLID のもう一つの特徴である「MMU の管理を簡単に行う」では、開発環境上のシンプルな GUI で論理・物理アドレスの設定を行うだけで MMU 設定処理を SOLID が生成する、という機能があります。Arm Cortex-A プロセッサでは比較的設定が複雑とされている MMU の処理を SOLID が代わりに行うというものです。

インテリジェントローダーにおいても複雑なシンボル解決とプログラムの配置に関わる一連の処理は SOLID がコアサービスの中で行っており、その過程では MMU によるメモリ空間の管理を同時に処理しています。

インテリジェントローダー行う実行時のアプリ間アドレス解決は広義の論物変換であり、MMU の仕事ともいえるでしょう。

開発者から

ツール屋として、できる事をとことん考えた

Linux のマルチスレッドのような機構をリアルタイム制御システムで実現する一つの手段はオリジナルの OS を開発する方法です。つまり OS のカーネルの機能として分散開発されたアプリケーションを脱着できるようにすることです。しかし現実問題としてお客様に全く新しい OS を提唱することは、かえってお客様の採用のハードルを上げてしまう事になり、お客様にとっても当社にとってもメリットは少ないと思われます。

京都マイクロコンピュータは、長年に渡り開発環境を第一線のお客様と共に手掛けてきた、プロの「ツール屋」ですので、開発ツールの視点からこの問題を解決する方法を考えました。

やりたい事は、複数のプログラムをビルドしロードすることです。開発環境がホスト PC で行っている事をアプリケーションの中で「実行」できれば、同じ結果になる、つまりリンカが行っているアドレス解決処理とローダーの動作をランタイム側で行えばよいのだという点に着目しました。アドレス解決に必要な ELF 情報をランタイムに含め、その ELF 情報を参照してプログラムを物理メモリ上に配置する処理もランタイム側に置けば良いのです。

そのように検討した結果生まれたのが OS に依存しないベアメタルのローダーである SOLID インテリジェントローダーです。ベアメタルのローダーを新規開発することで、既存の RTOS やツールチェーンといった汎用性の高いツールに手を加えることなく利用できることがわかってきました。

なおこのローダーはランタイム側で動作するものなので、プログラムサイズと実行時間に対するオーバーヘッドが極力少なくなるよう、十分配慮して開発しました。

DLL 方式と異なる処理を採用した理由

SOLID インテリジェントローダーは、DLL 方式ではなく当社が独自にその方式を考案しインプリしたものです。なぜローダーを独自に作ったのか、その結論に至った背景を一部ここでご紹介します。

まず第一に、ロード対象である複数のモジュール間で、共有シンボルや関数の相互参照がしたかったからです。DLL 方式の場合ロードする側とされる側の親子関係が固定されてしまい、相互参照が解決できない点に不便さを感じました(図 3-4）。

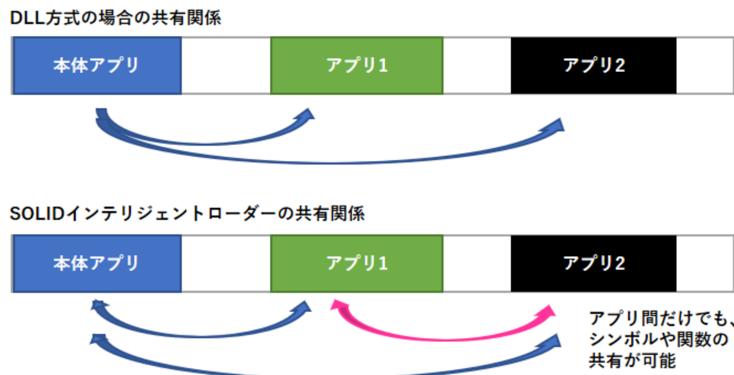

図 3-4 相互参照の関係の相違

次に考慮したのが実行速度の点です。DLL 方式ではスレッド（アプリ）が呼び出された時点ではじめて実行モジュールがメモリ上にダウンロードされます。この場合当然ながら実行時にダウンロード時間がオーバーヘッドとなってしまい、RTOS 利用を対象とするリアルタイム重視の制御システムには不向きです(図 3-5)。そのためシステム起動時の初期化段階で全てのアプリのダウンロードを済ませておき、その後は論理・物理空間の固定したアドレスに全てのプログラムがロードされた状態でシステムを稼働する方法が好ましいと考えました(図 3-6)。

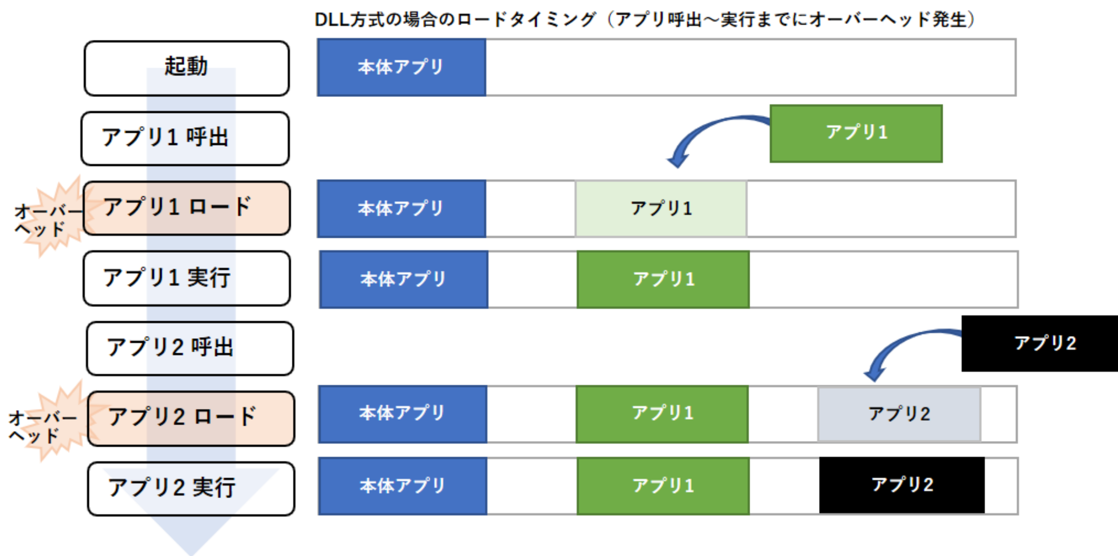

図 3-5 DLL 方式のロードでは、アプリ起動時にオーバーヘッドが発生

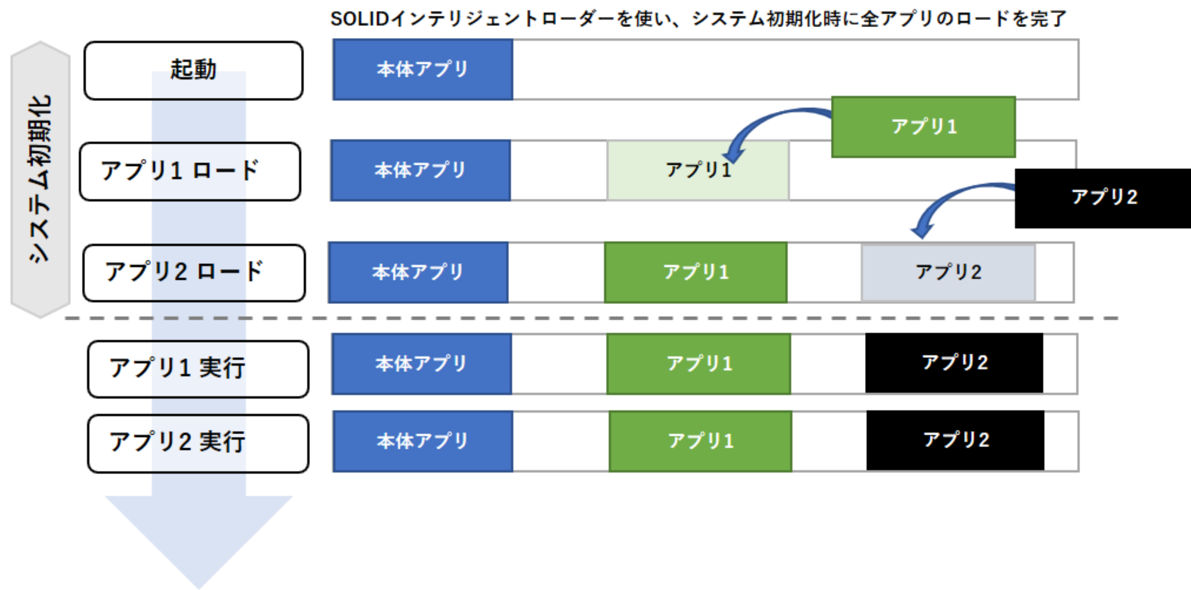

図 3-6 SOLID インテリジェントローダーを使い、システム初期化時にロード完了

京都マイクロコンピュータは、長年にわたりツールチェーンを含む開発環境をご提供してきたツールベンダーであり、リンカの動作が分かっていたからこそ出来たローダーといえます。また SOLID がツール単体ではなく IDE、ツールチェーン、RTOS、デバッガと一体化した構成であったからこそ、ユーザーがシンプルな手順で利用できるような形で、インテリジェントローダー提供できるようになりました。

MMU と連携しているので、メモリを効率よく利用できる

インテリジェントローダーでは、実行モジュールをロードする際に、自動的に MMU を使って論理→物理空間のメモリ配置を行っています。ユーザーは開発途中で再配置／再調整をしなくてもいいよう、論理空間上に余裕を持って実行モジュールを配置すればよく、自由度の高いメモリ配置が可能です。

インテリジェントローダーは、使用していない空間には物理メモリを割り当てないよう無駄なくメモリを配置し、その配置情報がコアサービス内の MMU 管理機能と連携しているので効率よくメモリが利用できます(図 3-7)。是非この点にも着目して、うまくローダーを利用していただきたいと思います。

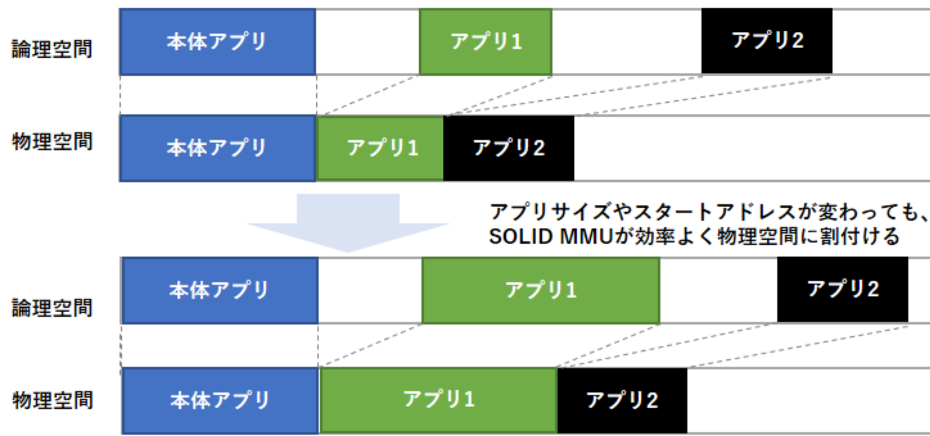

図 3-7 ローダーが物理空間に効率よく実行モジュールを配置

(終わり)