ソフトウェア工学　139－5
(2002. 10. 10)

組込みシステム向けオブジェクト指向フレームワーク

玉木 裕二† 宮田 尚史‡ 深谷 哲司†

†株式会社 東芝 研究開発センター システム技術ラボラトリ
‡株式会社 東芝 セミコンダクター社 システムLSI事業部 システム・ソフトウェア技術統括部

近年、組込みシステムはますます大規模、複雑化が進み、かつ、短期間での開発が求められるようになってきている。このような状況を解決するために、組込みシステム開発へのオブジェクト指向技術適用に注目が集まっている。しかし、組込み故の多くの制約事項がネックとなり、大きな成果が得られたという報告はまだ少ない。本稿では、組込みシステムの持つ特徴、およびその課題を整理し、オブジェクト指向技術を導入するときに障害となり得る事項を抽出する。実装面の課題に焦点を当て、デバイスの依存性の解決手段として開発したDevice Driver Framework(DDF)、マルチタスク関連処理の解決手段として開発したMulti Task Framework(MTF)について紹介し、あわせてこれらの適用事例を紹介する。

Object-Oriented Framework for Embedded Systems

Yuji TAMAKI† Takashi MIYATA‡ Tetsuji FUKAYA†

†System Engineering Laboratory, Research & Development Center, Toshiba Corporation
‡System & Software Dept., System Lsi Div., Semiconductor Company, Toshiba Corporation

Recently embedded systems are getting larger in size and more complex, while their development cycles are getting shorter. Under such circumstances, embedded system development projects have been seeking the way to apply the object-oriented technology. There are, however, only a few successful experiences have been reported because of a number of restrictions of the embedded system. In this paper, we identify potential obstacles to the object-oriented technology introduction to the embedded system development by organizing the characteristics and the problems of embedded systems. We introduce our Device Driver Framework (DDF) and Multi Task Framework (MTF). DDF has been developed to solve the device interdependency from the implementational point of view. MTF has been proposed to deal with multi-task-related processings. We also provide application examples of those frameworks.

1 はじめに

近年、組込みシステムはますます大規模、複雑化が進んでいる。その一方で、組込みシステムの製品寿命は短くなり、結果的に、短期間での開発が求められるようになってきている。組込みシステムに関するこのような状況を解決するために、従来にも増して効率的な開発手法の整備が急務となってきている。この解決策の一つとして近年、組込みシステム開発へのオブジェクト指向技術の適用に注目が集まっている。オブジェクト指向技術は、ソフトウェアの柔軟性や再利用性などの向上という点で、既にビジネスアプリケーション領域ではかなり浸透している技術であり、これを組込みシステム開発にも適用しようとする試みも増えつつある。

しかし、組込みシステム分野では、組込み故の特性である、サイズ、性能などの制約、周辺デバイスによる制約など、多くの制約事項がネックとなり、オブジェクト指向技術の持つ良さが十分に発揮され、大きな成果が得られたという報告は、まだ少ない。

我々は、株式会社 東芝で開発している様々な組込みシステムにオブジェクト指向技術を適用し、その開発を効率的に進められるようにすることを目指している。この研究/開発の中で我々は、特に、オブジェクト指向技術導入のポイントとして、

- デバイス依存性などを如何に解決していくか
- 組込みシステムで多く見られるマルチタスク関連

処理を如何にスマートに実現するか
に着目し、これらの設計から実装につなげる技術を追及している。

本稿では、この研究に関して、まず組込みシステムの持つ特徴およびその課題を整理し、オブジェクト指向技術を導入するときに障害となり得る事項を抽出した。次いで、デバイス依存性の解決手段として開発した Device Driver Framework(DDF)、マルチタスク関連処理の解決手段として開発した Multi Task Framework(MTF) について紹介し、あわせてこれらの適用事例を紹介する。

2 組込みソフトウェアの特徴と課題

組込みシステムにおいて、そのソフトウェアはデバイスを制御するためのソフトウェアという側面が強い。デバイスとは、プロセッサに内蔵された、あるいは外部に接続されたハードウェアのことである。具体的には、内蔵の SIO(シリアル I/O)、タイマコントローラ、A/D コンバータや、プロセッサポートに接続されたモータドライブ回路、センサ、あるいは、システム独自の ASIC などである。組込みソフトウェアは、これら個々のデバイスを適切なタイミングで動作させることで、機器全体の制御を行うことになる。

一方で、最近では、携帯電話やカーナビゲーションシステムに代表されるように、複雑なマンマシンインタフェースを持つ製品も増えて来ている。こうした製品では、LCD にメニューを表示したり、ボタンを使って文字を入力させたりと、パソコンや EWS 上のソフトウェアと同等の機能を提供しなければならなくなった。このため、従来のデバイスを制御するためのソフトウェアという側面に加えて、こうしたアプケーションソフトウェアとしての側面も色濃くなってきたと言える。

本章では、このような組込みソフトウェアの特徴を検討した上で、オブジェクト指向技術導入の必要性と、導入に伴なう技術的な課題について議論する。

2.1 組込みソフトウェアの特徴とオブジェクト指向技術

組込みソフトウェアを考える上では、組込みという制約下での、プロダクトとしての特徴と、これらを開発していく上でのプロセス面での特徴の、2 面を考慮する必要があると考える。具体的には、

プロダクト面の特徴

PD1:処理時間やメモリ使用量などの性能面で、厳しい制約がある
PD2:対応すべき OS やデバイスが多様であり、その変更も頻繁である
PD3:単純な制御処理以外に、アプリケーションとして様々な情報処理機能が付加される傾向が加速し、PC 上のアプリケーションソフトと同等の規模と複雑さを持つ

プロセス面の特徴

PR1:開発にはクロスプラットフォーム開発特有の実装技術やノウハウが必要となる
PR2:開発量に比べ、極めて短期間での開発が求められる場合が少なくない
PR3:開発プロジェクトが肥大化し、多人数での分散・分業による開発が主流になっている

これらの特徴を踏まえて、組込みソフトウェア開発の現状を一言で表現すると、ただでさえ通常の PC アプリケーションと比較して実装が難しいにも関わらず、その機能が PC アプリケーション並みに大規模、複雑化してきている、と言える。こうした特徴に適切に対処していくためには、

- 多人数による効率的な開発を可能にするような、分析、設計のモデルを構築すること、
- 開発環境、OS、デバイスが変わっても、アプリケーションコードへの影響範囲を最小限に留めるような設計をすること、
- 組込みソフトウェア固有の、実装の難しさに関する問題を解決すること、

の三つを実現することが不可欠であると考える。一つ目の課題については、現在、上流における設計手法の検討や、良い設計を行うための “考え方の教育” の開発も進めている。

本稿では、二つ目と三つ目の課題に焦点を絞り、それぞれアプローチを提案する。二つ目の課題は、「デバイスの多様性・変更蓋然性」を如何に解決するかといった課題に集約される。また、三つ目の課題は、組込みソフトウェアの実装を難しくしている要因の一つが、タスクの並行性にあり、これをオブジェクト指向にどのような形で反映させていくかといった「タスクとオブジェクトの関係」を解決することがポイントとなる。

2.2 デバイスの多様性/変更蓋然性

組込みソフトウェアは、プロセッサに内蔵のデバイス、あるいは外付けのデバイスを制御するという場面が多く、扱うデバイスの多様性/変更蓋然性の高さが開発を難しくする要因の一つとなっている。この問題に対しては、デバイスを直接操作する部分をモジュール化し、上位のアプリケーション等へ API を提供するデバイスドライバ層を設けるアプローチが一般的である。UNIX などの汎用 OS では、デバイスドライバを OS 内部に組み込み、デバイスをファイルとして抽象化する方式が採られている。一方、多くの組込みシステムで採用されている μITRON 仕様 OS[3] などのリアルタイム OS(以下、RTOS) は、UNIX のようなデバイスを扱うための枠組みを提供していないため、デバイスドライバは上位アプリケー

ションに直接組み込むライブラリとして実装することになる。

近年の組込みシステムでは、デバイスの制御という側面よりも、マンマシンインタフェースを持ち、高機能化/複雑化したアプリケーションソフトウェアとしての側面が強くなっている。さらに、これまで以上に**扱うデバイスの多種多様化が進み、開発後半でのデバイス変更や同製品の派生種毎にデバイスの変更等**が頻発する。

汎用OSのアプローチは、**"ファイル"**というインタフェースを提供することでデバイスの見え方を標準化している。しかし、画一的なインタフェースは組込みシステムにおけるデバイスの多様性/変更蓋然性という観点からは柔軟性に欠け、デバイスの変更に対して、デバイスドライバだけでなく上位アプリケーションも大幅な修正を余儀なくされる場合が多い。

このような問題に対応するためには、複数のデバイスをまとめ、より抽象度の高いインタフェースを提供するなどの考え方が必要である。すなわち、上位アプリケーションの記述性を高め、かつデバイスの構成から独立したデバイスドライバが望ましい。これらを実現するためには、2つの壁がある。

(1) システムの分析/ 設計段階で、デバイスの変更までを考慮したモデル化、すなわち、いかに適切なインタフェースを定義できるかが、最終的な結果に大きく影響を及ぼす。こうした上流工程はもともと難しい問題であることに加えて、下流工程に割ける時間が少なくなるのではないかという懸念、上流工程の意義の未浸透などから、軽視されがちである。

(2) デバイスドライバの実装では、レジスタ構成や書き込むビットパタン、割込み処理など、個々のデバイスに応じて記述しなければならず、こうした部分の実装には高度なノウハウが必要である。デバイスに精通したエンジニアでないと、実装へ展開する工数が多大になりがちである。

2.3 タスクとオブジェクトの直交性と相違性

近年組込みシステム開発において、RTOSを導入した開発が主流となってきている。RTOSが提供するマルチタスク機能を利用したシステム開発では「タスク」をモジュールの分割単位として考えるのが一般的である。それに加えて、オブジェクト指向技術を導入する場合にはモジュール分割単位として「オブジェクト/クラス」が必要となる。これらの混用が、システム開発の各工程においてオブジェクト指向技術導入の壁となることが多い。それらを整理してみると、問題点として次の3つが挙げられる。

(1) **分析/設計時におけるモジュール分割単位の違い**
「タスク」はシステムを動的側面から見た場合のモジュール分割単位である。それに対し、「クラス」はシステムに対して、主に静的側面から見た場合の分割単位である[2]。従って、タスクとオブジェクト/クラスは直交した概念と言える。

(2) **実装時におけるプログラミングパラダイムの違い**
「タスク」など並行メカニズムを提供するRTOSはC言語による実装が大半であるため、手続き型プログラミングが基盤となる。しかし、C++言語などのオブジェクト指向言語を用いた場合は当然オブジェクト指向プログラミングが基盤となるため、2つのプログラミングスタイルを開発者に強いることとなる。そのため、プロジェクト内で統一が取れていないと実装や保守において混乱を招く。

(3) **設計/実装における「タスク」という概念の違い**
分析/設計工程において、「タスク」は論理的に並行動作する機能的なまとまりを意味する(これはオブジェクト指向分析/設計における同期/非同期の議論に近い)。ところが、実装工程においてはRTOSが提供する並行処理単位となり、その構成はシステムへのパフォーマンス要求により変更される可能性が高い。

3 アプローチ

前述の課題分析を通して、組込みシステム開発にオブジェクト指向技術を効果的に導入するためには、

- デバイスの多様性・変更蓋然性を吸収するようなメカニズムを提供することが必要
- 組込みシステムにおける論理上の並行性(クラス設計)と実装上の並行性(タスク設計)を、ともに満たすようなメカニズムを提供していくことが必要

といった結論を導出できる。

我々はこの結論に対して、これらのメカニズムを提供することを目標に、**Device Driver Framework(DDF) / Multi Task Framework(MTF)**の研究・開発を進めてきた。これら二つのフレームワークは、それぞれ「デバイスの多様性・変更蓋然性」や「並行性」の問題について、オブジェクト指向分析/設計モデルから、C++言語による実装を容易にすることを狙っている。即ち、MTFはオブジェクト指向設計モデルをRTOS上へ容易に展開することを目的としており、DDFはデバイスドライバの柔軟なインタフェース構築および、デバイスへ変更への迅速な対応を可能にすることを目的にしている。

4 DDF

4.1 DDFの概要

4.1.1 DDFの役割と狙い

組込みシステムにおけるデバイスの多様性や変更蓋然性を解決する方法として、我々は**DDF**(Device Driver Framework)の開発を進めている。組込みシ

ステムで周辺デバイスを操作する場合には、デバイスドライバを介して制御する。**DDF** は、制御対象となるデバイスの多様性を吸収し、デバイスドライバを C++ 言語によるクラスライブラリとして提供する枠組みである。言い換えると、**DDF** は上位アプリケーションへ提供する仮想的なデバイス毎に C++ 言語でクラス (デバイスクラス) を実現する役割を担うものである。

4.1.2 DDF の特徴

DDF では、デバイスの変更が論理モデルへ影響することを最小限に抑えるために、インタフェースと実装を完全に分離させる形でデバイスクラスを実装する方針を採用した。

- 上位アプリケーションに対し、デバイスの物理的な構成、接続形態や、個々のデバイスの操作プロトコルなどに依存しない、抽象的なインタフェースを提供する
- デバイスクラス内部では、インタフェースを具体的な操作に変換し、実際の制御を行う

さらに、デバイスの多様性/変更蓋然性に迅速に対応するためには、デバイスクラス内部での制御も容易に実装できる必要がある。このため、**DDF** には、個々のデバイスを低レベルに扱う部分を、ライブラリとして提供する枠組みを導入した。

これらの特徴を実現することで、**DDF** を利用した場合、従来のデバイスドライバ実現方法と比べ、次のような効果が期待できる。

(1) デバイスの変更に迅速に対応できる
(2) 様々な抽象度のインタフェースに対応できる
(3) インタフェースは、システムをどのように分析したかによって決まってくるものであるため、フレームワーク利用者が、自由にそれを定義して利用するといた、自由度が増加する

4.2 DDF の実現方法

4.2.1 デバイス変更への迅速な対応

デバイスの変更を容易にするために、デバイス操作に必要となる種類をあらかじめ用意して、フレームワーク内で提供する。様々なデバイスを分析し、デバイスドライバ用に必要となるクラスを抽出した。クラスの責務を分類すると、基本機能と例外機能の 2 種類のサービスを提供する必要があることが分かった。

基本機能は、レジスタ操作などの低レベルな処理を行う操作 *Primitive Method* と、これらの組合せ/拡張を行うための操作 *Composite Method* に分類できる。例外機能はクラス内で不変制約を保証する役割を持つ。

Primitive Methodポートアクセス、レジスタアクセス、タイマを利用したタイミング判定、割込みハンドラ処理

Composite Methodポートのポーリング、コールバック処理、チャタリング除去

Primitive Method を有するクラス群は、実際のデバイスに依存する部分であり、*Composite Method* を有するクラス群は、デバイス非依存にできる部分である。再利用の観点からも、これらを分離した。

4.3 DDF の構成

前述したフレームワーク要件を如何に **DDF** で実現したかについて述べる。デバイスクラスを含めた、**DDF** の構成を図 1 に示す。

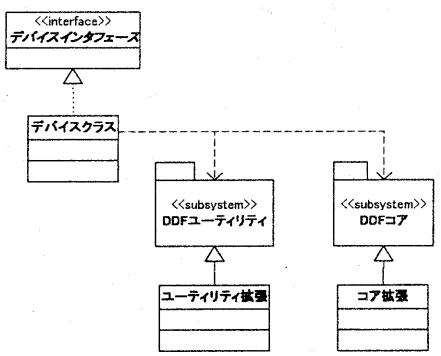

図 1: デバイスクラスの構成

(1) デバイスインタフェース
デバイスを扱うためのインタフェースを提供するクラス。純粋仮想関数の宣言だけで構成される。これはデバイスクラス作成者が記述する。

(2) **DDF** コア
レジスタ操作や割込み処理、I/O など、個々のデバイスを直接操作する *Primitive Method* を提供するクラス群 (図 2)。扱う対象となるデバイス毎 (A/D コンバータ、DMA、タイマ、ASIC、等) にパッケージ化し、更に各パッケージ内を *Primitive Method* 毎にクラスとして実装している。
今回は、東芝製 RISC プロセッサ TX19[5] の内蔵ペリフェラルを対象に **DDF** コアの実装を行った。実装例として、TX19 に内蔵されている A/D コンバータを挙げる。この A/D コンバータは、以下の 4 つの変換モードを持っている。

- 指定した一つのチャネルを 1 回変換する
- 指定した一つのチャネルを繰り返し変換する
- 指定した複数チャネルを各 1 回ずつ変換する
- 指定した複数チャネルを順に繰り返し変換する

この各モードを一つのクラスとして実装し、変換の開始を指示する `start()` という共通のインタフェースを設けた。なぜなら、A/D コンバータクラスは何れのモードであっても **"データを変換する"** という共通の責務を持つからである。
例外機能として、デバイス資源の不正な使用を検知機能を組み込んだ。一般にデバイスドライバの

開発時は、デバイスのレジスタ操作の誤りや、不正なレジスタアクセスを行う場合がある。そのような不正な使用を検知し、ユーザに通知する仕組みを実装した。

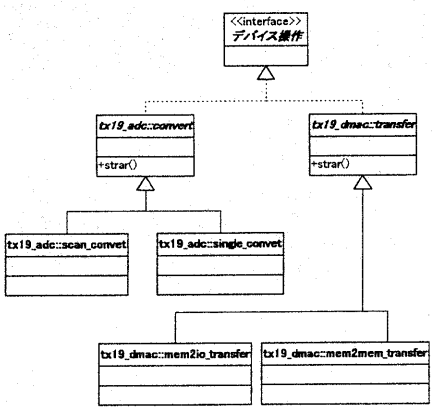

図 2: DDF コアの構成 (抜粋)

(5) **DDF** ユーティリティ
DDF コアが提供する機能を組み合わせて、より上位の機能を提供するクラス群。具体的には、
- 任意のメモリを I/O ポートとして透過に扱うためのクラス
- 上記 I/O ポートに対して、ポーリング、チャタリング除去、コールバックを行うためのクラス

などがある。

(3) デバイスクラス
デバイスインタフェースを具体的に実装するクラス群。**DDF** ユーティリティ、**DDF** コアを使用してデバイスを制御する。

(4) ユーティリティ拡張
フレームワークが提供する **DDF** ユーティリティを派生させて、その振る舞いをカスタマイズするなど、**DDF** ユーティリティを拡張する部分。

(5) コア拡張
フレームワークが提供する **DDF** コアを派生させて、その振る舞いをカスタマイズしたり、システム独自の ASIC などを扱うために、**DDF** コアを拡張する部分。

また、**DDF** が利用するデバイスの割り当てを設定するコンフィグレータを実装した。これを利用することで、システム起動時に機能レジスタ、制御レジスタなど、必要なデバイスの初期設定を確実に行うことができる。

5 MTF

5.1 MTF の概要

5.1.1 MTF の役割と狙い

MTF は組込みシステムに多用されるタスクの並行処理などを効率的に処理することを狙ったライブラリである。特に、これらの並行処理メカニズムの移植性、再利用性を向上したり、並行性に起因するシステムの不安定な振る舞いを除去し、安全な並行プログラムを容易に記述可能にすることを狙っている。また組込みシステムで利用される RTOS に関しても、RTOS が提供する機能を適切に利用し、また、C++言語機能を利用した RTOS 機能の拡張を容易にするといったこともあわせて狙っている。これにより **MTF** を用いて記述したモジュールと、RTOS を用いて記述したモジュールが同じアプリケーション内で共存できるようになる。

5.1.2 MTF の特徴

上記のような事項を実現するため、我々は組み込みシステムの資源制約を考慮し最適化したライブラリとして **MTF** を開発している。

我々が開発している **MTF** では、RTOS 資源に対するラッパーライブラリを用意し、オブジェクト指向プログラミングに統一する。また並行性に対する抽象度の違いを吸収するために、RTOS ラッパーライブラリとそれを用いたより高位のライブラリ (例えばオブジェクト指向で扱う並行性を表現したもの) を用意して対処している。これにより、RTOS 機能対応部分と、その機能を組み合わせて実現できる機能の分離が可能となり、論理的な並行性に関する部分は特定の RTOS に依存せずに再利用できる。

このような特徴をもつ **MTF** を組み込みシステム開発に適用していくことで、我々は現状の開発からオブジェクト指向開発へ段階的に移行していくことを目指している。

5.2 MTF の構成

MTF は、RTOS が提供する資源を容易に拡張可能な枠組みを提供する C++言語で実装されたクラスライブラリである。**MTF** は、次のパッケージで構成される (図 3)。

5.2.1 MTF OS ラッパー

RTOS が提供する機能に対応する資源オブジェクトを規定するクラス群。これにより、特定の OS 実装に依存することなく、並行プログラムを記述できる。今回は μITRON 仕様 OS[3] に対する実装を行った。このパッケージは RTOS が提供する機能毎にサブパッケージに分割されている。

5.2.2 MTF ユーティリティ

並行プログラムの構造化を図るためのヘルパークラス群。ラッパークラスにより抽象化した OS 資源

を用いて実装している。具体的には次のものがある。

- 論理上の並行性を実現するクラス (Actor)[2]
- RTOS を利用した STL 互換メモリ管理クラス
- 構造化ロッキング (モニタ) を実現するクラス

ユーザは OS ラッパーのみを用いた従来型に近いマルチタスクプログラミングも可能であり、またユーティリティを利用した並行オブジェクト指向プログラミングも可能である。更に、実装もテンプレート機能、インライン関数などを用いることで、実行時のオーバーヘッドを抑え、RTOS API の直接呼出しに近い性能を出すことが可能となっている。

また、特定の RTOS 実装に依存することなく、RTOS 資源管理に関するコンフィグレーションを記述できるようにするためのツール、**MTF** コンフィグレータを開発した。静的な資源確保が必要な RTOS に対応している。資源確保以外は **MTF** OS ラッパーを介して設定する。

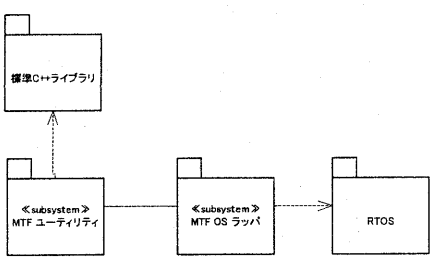

図 3: MTF の構成

6 適用事例

今回開発した **MTF/DDF** を、ブロックキットによるエレベータ制御システムの開発に適用した。エレベータシステムの外観を図 4 に示す。エレベータシステムは、DC モータ、磁気センサ、ボタン、ランプ、ブザーなどから成り、各部品は、東芝製 RISC プロセッサ TX1940 スタータキットボードに接続されている。

6.1 制御ソフトウェアの仕様

開発した制御ソフトウェアには、n 階建て、m 台のかごを群制御可能とし、フロアボタンでかご呼び出し、非常停止など、一般的なエレベータが持つ機能を想定した。

6.2 プロジェクト計画

C++言語やオブジェクト指向技術に精通していないエンジニア 4 人が分析から作業を行い、筆者らは、主にレビューアとして参加した。今回のアプローチの有効性を確認するために、分析/設計作業に比重をおき、デバイスの変更、制御仕様の変更などを十分考慮し、メンバ全員が十分納得できるモデルが構築できるまで、実装へは移行しないこととした。

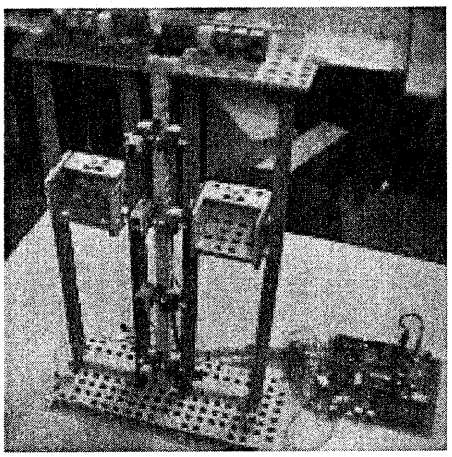

図 4: エレベータシステム

また、実装ははまず PC(Microsoft Windows) 上でのシミュレータ (図 5) を用いて開発を行い、機能的な動作確認が取れてから、ボードを用いた開発へ移行する計画とした。

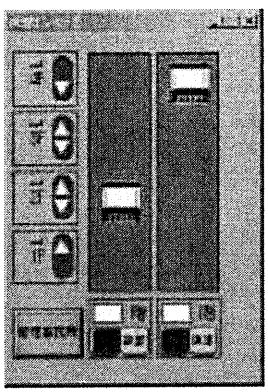

図 5: エレベータシミュレータ

6.3 概念モデル

上流の分析/設計には、約 3ヶ月を費し、最終的に図 6 の概念モデルを完成させた。

各クラスの役割 (抜粋) を、次のように定義した。

かご 自身の運行スケジュールを保持し、そのスケジュールに従ってかごを動作させる。他のかごの運行には、一切関知しない。運行スケジュールの更新は、自分自身だけが行うことができ、これを行うのは、次の三つ場合のみである。

- 自身のかご内で、特定のフロアへの移動要求が発生したとき
- ディスパッチャから、特定のフロアへの立ち寄りを指示されたとき
- 管理事務所から、特別な指示を受けたとき

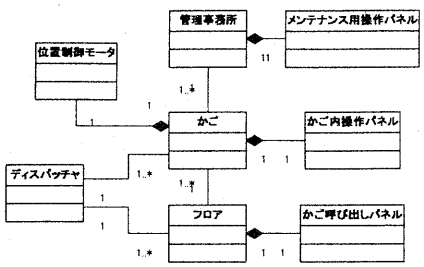

図 6: エレベータシステムの概念モデル (抜粋)

運行スケジュールを更新するときは、最適なスケジュールになるように行う。また、保持している運行スケジュールの中で、現在位置から特定のフロアへ立ち寄る場合のコスト (時間) を見積ることができる。

フロア フロアにいる乗客からの、リフト呼び出し要求を受け付け、要求が発生したことををディスパッチャに通知する。フロアは、どのかごが来るのかについては、一切関知しない。
また、かごの到着等の情報を、フロアにいる乗客に通知する。

ディスパッチャ フロアからのかご呼び出し要求に応じて、どのかごを要求のあったフロアへ行かせるかを決定する。決定には、すべてのかごに、そのフロアへの到着コスト (時間) を問い合わせ、その情報を元に行う。ただし、各かごの詳細な運行スケジュールには、一切関知しない。
決定後、かごにフロアへの立ち寄り要求を指示する。

管理事務所 地震の検知、メンテナンス用操作パネルからの指示など、システム全体の動作を決定するような事象の検出を行う。事象に応じて、かごに特別な指示を出すことができる。

6.4 論理モデルから実装モデルへの展開

上記概念モデルを、さらに詳細に設計し、実装へと展開するときにに、今回我々が提案するフレームワークをどのように利用したかについて述べる。

6.4.1 デバイスの多様性/変更蓋然性

プロジェクト計画で述べたように、まずシミュレーション環境で実装を進め、ある段階からはボード環境へ移行する計画であった。シミュレータでは、かごの移動指示やボタンの状態検知に、共有メモリを仮想的な I/O ポートとして利用する。一方、実機ボードでは、DMA コントローラ、モータドライブ回路などの制御や、実際の I/O ポートの読み書きを行わなければならない。

それぞれの環境に応じたデバイスドライバを作成することは可能だが、より効率的にデバッグを進めるには、ボード環境への移行後も、シミュレーション環境でのデバッグを行えるようにする必要がある。つまり、上位ソフトウェアは、こうした動作環境とは独立にしなければならない。

我々は、概念モデル上で、デバイスとのインタフェースをどこで切るかの検討を繰り返し、かご位置の制御については、次のような結論に達した。

> かごクラスは、自身のかご (つまりモータやセンサなど) の制御の仕方には興味がなく、単に、目的のフロアを指示できれば十分である。したがって、モータやセンサなど、かごを上下させるために必要なデバイス群を「位置制御モータ」という仮想的なデバイスとしてまとめ、目的地と到着時の通知方法をデバイスインタフェースとすれば良い。

ボタンやランプなどについても同様に検討し、それぞれデバイスインタフェースを定めた。

位置制御モータは、図 7 のようにデバイスインタフェースクラスを設け、環境に合わせた実装クラスをそれぞれ用意し、それぞれは、**DDF** コア、**DDF** ユーティリティを用いて実装した。

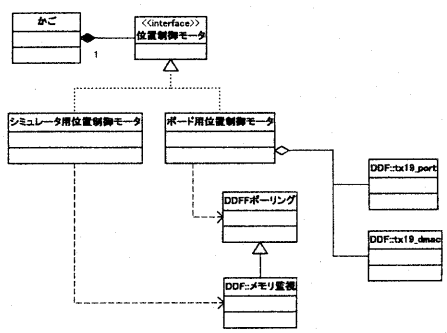

図 7: DDF の利用: 位置制御モータ

6.4.2 タスクとオブジェクトの直交性と相違性

概念モデルで、かご、フロア、ディスパチャ、管理事務所の四つの中心的なクラスを抽出した。これらは、お互いにメッセージを送受信しあうことになる。クラス設計を実施している際に、各クラス間のイベントシーケンスを洗い出し、そこで非同期メッセージとして受取るべきものと、同期メッセージとして受取るべきものを分類した。その結果、いずれのクラスも、非同期にメッセージを送受信できなければならないことが分かった。

非同期メッセージを受け取ってそれを処理する実装方法は様々であるが、一般的には、各オブジェクトにメッセージキューと、それを処理するためのタスクを割り当てる実装となる。しかし、性能チュー

ニングの都合上、一つのタスクで、いくつかのオブジェクトのメッセージ受信処理を行う方が有利かもしれない。

つまり、上記のオブジェクト群は、非同期にメッセージを処理できるアクティブオブジェクトであれば良く、それぞれに個別のタスクを割り当てるか否かという問題とは独立である。

そこで、MTF を用いて、図 8 に示すような構成で実装した。Runnnable クラスは、アクティブオブジェクトとして振舞うためのエントリ run() を、純粋仮想関数として持つクラスである。Actor クラスは、メッセージキューを持ち、キューから一つずつメッセージを取り出しては、派生クラスのメッセージ処理関数を呼び出しを繰り返すためのクラスである。Context クラスは、RTOS のタスクや割り込みハンドラなどの実行コンテキストを提供するためのクラスである。

かごクラス等は、上記 Actor クラスから派生させアクティブオブジェクトとした。さらに、各アクティブオブジェクトにどのように実行コンテキストを割り付けるかを決定する、Configurator クラスを設けた。実装レベルのタスクの割り付けの調整は、主に、Configurator クラスを修正することで実施する。

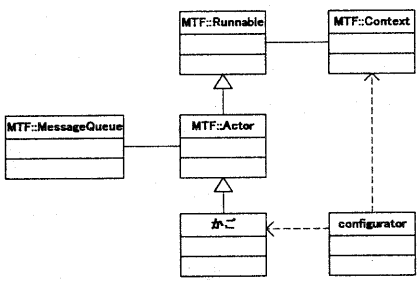

図 8: MTF の利用: かごクラス

6.5 DDF の評価

シミュレーション環境で実装を開始してから、エレベータを動作させるまでに約 1ヶ月を要したのに対して、シミュレーション環境からボード環境への移行はわずか 2 日であった。これは、(1) 上流設計段階で、適切なデバイスインタフェースを定めたこと、(2) 実装段階でもボード環境のためのデバイスドライバの実装を容易に行えたことによる。すなわち、**DDF** が、2.2 節で延べた課題を解決できたことを確認できた。

6.6 MTF の評価

MTF を用い、図 8 の構成を採用することにより、(1) 各クラスを必要なタスク構造へ無理なく割り当てるなど、クラスとタスクを分けて考えることができた。また、(2)**MTF** を用いることで、すべてのプログラムを統一したスタイルで記述できた。さらに、一つのタスクに、いくつかのオブジェクトの処理を任せたり、逆に、一つのオブジェクトに複数のタスクを割り当てるなどのチューニングを行う際、Configurator クラスの修正だけで対応でき、(3) 概念モデルを崩すことなく、実行効率チューニングを行うことができた。すなわち、2.3 節で延べた課題を解決できたことを確認できた。

7 おわりに

本稿では、組込みシステム開発において、オブジェクト指向分析/設計モデルから、実装に展開する際の問題を分析した。さらに、実装に展開する問題を解決するために、**MTF/DDF** を開発し、適用事例でその評価を行った。

組込みシステムにおいても上流工程におけるオブジェクト指向分析/設計の重要性が確認できた。また、分析/設計工程の比重を高くしても、実装への展開を支援する **MTF/DDF** を利用すれば、開発効率を得られることを確認できた。設計レベルで再利用が行えれば、次製品における効率化がさらに見込める。

言うまでもなく、組込みソフトウェア開発にオブジェクト指向技術を適用するためには、上流の分析/設計を如何に上手くやるかという点も、重要なポイントである。今回、この点には触れなかったが、現在我々は、上流設計手法についての研究や、より良い “設計の考え方” を行うための教育の開発も進めている。

今後は、こうした上流の分析/設計手法の明確化と、より広い製品への適用を進める中で、フレームワークとして提供すべき機能の拡充を行う予定である。

参考文献

[1] Bjarne Stroustrup, The C++ Programming Language, 3rd Edition, Addison-Wesley, 1998.

[2] James Coplien, Advanced C++ Programming Styles and Idioms, Reading, MA:Addison-Wesley, 1992.

[3] 坂村 健 監修, μITRON 3.0 標準ガイドブック改定新版, パーソナルメディア, 1997.

[4] uITRON4.0 仕様研究会 デバイスドライバ設計ガイドライン WG, デバイスドライバ設計ガイドライン WG 中間報告書—DIC アーキテクチャの提案—,http://itron.gr.jp/GUIDE/device-j.html, 1999.

[5] (株) 東芝 セミコンダクター社, 32 ビット TX System RISC TX19 ファミリー TX1940 編 データブック, (株) 東芝 , 2001.