アーキテクチャ分析やモデリングの基礎から
リアルタイム・システムの考え方まで

Interface 増刊

見本

組み込みシステム開発に役立つ理論と手法

藤倉俊幸 ◉ 著

CQ出版社

第 1 章

なぜ組み込みシステムにアーキテクチャが必要か

最初に，組み込みシステムにおけるソフトウェアのアーキテクチャについて解説します．建物のアーキテクチャならわかりやすいと思いますが，ソフトウェアのアーキテクチャと聞くと理解しにくい概念ではないかと思います．しかし，ソフトウェアも建物や自動車などと同じように複雑に組み合わせて作る創作物と考えると，アーキテクチャは非常に重要な概念になります．

ソフトウェア・アーキテクチャの起源は，構造化プログラミングで有名なエドガー・ダイクストラなどの研究によりますが，現在は抽象化と分割を行うことによって複雑性を減らすことが大きな目的になっています．

1.1 ハードウェア・アーキテクチャとソフトウェア・アーキテクチャ

アーキテクチャというと，プロセッサのアーキテクチャの方に興味のある人が多いかもしれませんが，本章では組み込みシステム全体のアーキテクチャについて考えます．

設計の基本は分割です．分割すれば，内側と外側ができます．まず，ここに気付く必要があります．プロセッサのアーキテクチャの場合，外側は x86 や ARM などのレジスタの種類やサイズ，メモリ管理や割り込みモデルのようなユーザから見える構造を指します．もちろん，内側にも構造があり，内側の構造を指す場合は構成（organization）などと呼ばれます．

組み込みシステムを作る場合，様々に異なるアーキテクチャを持つ CPU の中から様々な要因で CPU を選択します．選ぶ基準はいろいろありますが，CPU が決まるとその CPU アーキテクチャはソフトウェアの構造にも影響を与えます．例えば，ソフトウェアはメモリ空間や割り込み処理モデルなどの影響を大きく受けます．

現在は，開発期間を短縮するためにソフトウェアとハードウェアを同時に作ることが常識になっ

イラスト 1.1　ソフトウェアも構造が大切

ています．したがって，CPU が決まってからソフトウェアの構造を考えているようでは遅いのです．同時に考えるのであれば，CPU のアーキテクチャを選ぶ基準をソフトウェアに適用すればよくなります．

そこで，こう考えます．まず，システム全体のアーキテクチャがあり，その一部としてハードウェア・アーキテクチャやソフトウェア・アーキテクチャがあるということです．統一されたアーキテクチャがあれば，ソフトウェアとハードウェアを並行に開発しても矛盾は生じないはずです．

ハードウェアも含めた要求分析を行って，何を作るかが決まって，どう作るかを考えはじめる最初のフェーズで，システム全体のアーキテクチャを考えるのが理想的です．そして，その中でソフトウェア・アーキテクチャも考えます．つまり，設計工程でアーキテクチャを考えることになります．

それでは，いわゆる設計とアーキテクチャはどこが違うのでしょうか．

1.2 設計とアーキテクチャ

カーネギーメロン大学の Web サイト（http://www.sei.cmu.edu/architecture/start/community.cfm）を参照すると，いろいろなアーキテクチャが説明されています．それらのアーキテクチャに共通しているのは，システムを部分に分割して構造を見せることと，部分間の相互作用と各部分の外的特性を明らかにすることです．

ここで重要なことは，構造には種類があることです．例えば，論理的な構造と時間的な構造，静的な構造と動的な構造などです．ソフトウェア・アーキテクチャを表現するには，1 種類の構造だけでは不十分です．**表 1.1** に示した構成物を構造ごとに用意する必要があります．組み込みソフトウェアの場合は，時間的な構造と動的な構造がとくに重要になります．

全体を分割して部分（コンポーネント）を作り，コンポーネントの外側だけを定義することで，つまりコンポーネントの内部を定義しないことで，ソフトウェアを抽象化して表現したことになります．

また，設計を基本設計と詳細設計に分ける場合があります．このとき，アーキテクチャと基本設計の成果物は同一のものという印象を受けます．しかし，アーキテクチャの場合，コンポーネントの内部については言及しないところが異なります．

設計工程の分類は開発現場ごとに異なる場合が多く，呼び方もまちまちですが，一般的と思われるものを**図 1.1** に示しました．アーキテクチャの位置付けの例を無理やり示したつもりですが，**図 1.1** では複数種類の構造が必要になるというところが伝わりません．それから，アーキテクチャと設計では抽象度が違います．アーキテクチャは抽象度が高く，再利用性が高いものです．

ところで，非常に簡単なソフトウェアのようにコンポーネントを分ける必要がない場合には，アーキテクチャは必要ないのでしょうか．実のところ，「アーキテクチャ」と「アーキテクチャの記述」は分けて考える必要があります．簡単なソフトウェアでアーキテクチャが自明であっても，外から見た動作や特性に関するアーキテクチャを記述することは必要です．

たとえ簡単なソフトウェアであっても，プロ

表 1.1　アーキテクチャの構成物

構成物	内　容
システムのコンポーネント	モジュール，レイヤ，プロセス，タスク
コンポーネントの外的特性	提供するサービス，パフォーマンス特性 エラー処理，共有リソース
コンポーネント間のインターフェース	–

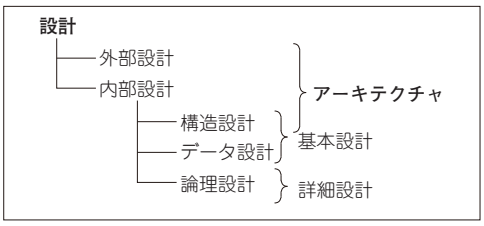


図 1.1　設計とアーキテクチャ

第 7 章

マルチタスク・プログラミングを実装する方法

本章では，マルチタスク・プログラミングを理解するための第一歩として，タスクと関数の違いから説明します．まず，タスクとは何かという基本的なところを抑えることが重要だと思います．

7.1 ロードと起動の違い

コンピュータの世界で「ロード」と「起動」の意味は，環境によって少しずつ異なります．例えば，「プログラムをロードする」という言い方は，ROMにプログラムを配置する場合には使いません．

ロードによってメモリに展開されるものは，モジュールとかコンポーネントと呼ばれます．これらは，ソフトウェアを（静的に）構成する要素になります．起動によって，コンテキストが（動的に）生成されてCPU時間が割り当てられます．起動されたものは，プロセスとかスレッドと呼ばれます．

ホスト系コンピュータのシングルタスク・プログラミングでは，「ロード」と「起動」は一対一に対応するので，意識する必要はないかもしれません．しかし，マルチタスク・プログラミングでは，一対多の対応になるので意識する必要があります．

図 7.1 はWindowsのタスク・マネージャですが，一つのロードされた実行イメージに対して複数のスレッドが起動されています．同じイメージが複数ロードされている場合もありますが，これはWindowsのようなプロセス型OSの特徴です．

組み込みプログラミングでは，プログラムをROMに焼いて，すでにロードされた状態にしておくことが多いので，「ロード」の概念がなく「起動」だけになる場合もあります．

Windowsなどでは，同一のプログラムを複数走らせることをしばしば行います．例えば，COM1とCOM2に対してハイパーターミナルを同時に利用したりします．このようなことが簡単にできるのは，Windowsがプロセス型OSだからです．リアルタイムOSの開発環境でもシェル[注1]が複数走っているように見せることは可能ですが，起動されているシェル・ウィンドウは完全に独立ではありません．

リアルタイムOSを選択する際，メモリ・プロテクションの有無やパフォーマンスなどでプロセス型かスレッド型かを決めることが多いと思いますが，モジュールの考え方や並行性と多重度の観点も重要です．モジュール，プロセス，スレッドの関係を**図 7.2** にまとめました．

注1：シェルは，オペレーティング・システムの一部であるが，中心部分という意味を持つ．主として，アプリケーションなどからの指示を受けて解釈し，起動や制御などを行うプログラムを指す．

図 7.1　Windows の実行イメージと起動されたスレッド

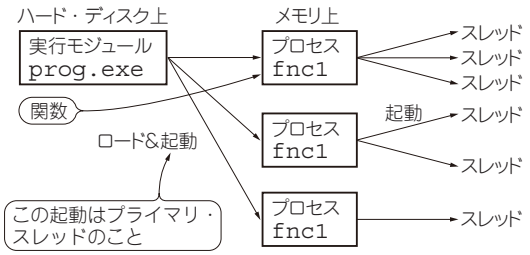


図 7.2　モジュールとプロセスとスレッドの関係

マルチタスク・プログラミングは，静的なソフトウェア構成要素である関数と動的なスレッドとの違いを明確に認識するところからスタートします．

7.2　タスクの構造

● 待ち状態とは

スレッドをリアルタイム OS が生成し，さらにスレッドの実行状態をリアルタイム OS が管理することで，スレッド実行の一時中断 / 再開などが可能になります．この中断 / 再開によって，並行性が実現されます．あるスレッドが実行を中断していることを「待たされる」とよくいいますが，このときは別のスレッドが動いているか，割り込みハンドラのようなシステム部分が動いているかのどちらかです．

プロセッサ自身には待ち状態というものはなく，リアルタイム OS などのソフトウェアが待ち状態を作り出すと考えてよいでしょう．したがって，リアルタイム OS を使わないシステムの場合は，ここでいう意味の待ち状態は存在しません．外部の事象をマルチプロセッサ・システムにおけるスピンロック[注2]とかビジーウェイト[注3]のような待ち方で待つことは可能ですが，シングルプロセッサ・システムでは，そもそも「一人しかいない環境」なので待つものがありません．

第9章

リアルタイム処理とは時間的制約を守ること

プログラムの正確さには，一般に論理的な正確さと時間的な正確さがあります．論理的な正確さとは，計算結果が正しいことや判断が正しいことです．また，時間的な正確さとは，いつまでに計算や制御の結果を出せばよいか，いつ入力データを読めばよいかといった要求に対する正確な振る舞いのことです．

論理的な正確さばかりでなく時間的な正確さも要求されるのが，リアルタイム・アプリケーションの特徴です．そこで本章では，リアルタイム・アプリケーションを理解するために，時間的な正確さについて考えてみます．

9.1 時間領域の仕様

● 論理的な正確さと時間的な正確さ

論理的な正確さと時間的な正確さの違いを，シリアル通信の場合で考えてみます．シリアル通信では，**図 9.1**（**a**）のようにデータ・パケットの始まりと終わりの定義をコントロール・コードSTXとETXによって行います．これはすなわち，データ値を利用した定義ですが，これを実現するためには，データ値が正しいという論理的な正確さが必要になります．

一方，**図 9.1**（**b**）のようにパケットの区切りを一定時間データの到着がないこと，すなわちギャップがあることで定義する場合は，時間を利用した定義になります．

図 9.1（**a**）のようなデータ領域の要求仕様を実現するためには，パケットの検出に関して，通信データの正確な扱いが求められます．また，**図 9.1**（**b**）の時間領域の要求仕様に対しては，通信データの中身を監視する必要はありませんが，「パケッ

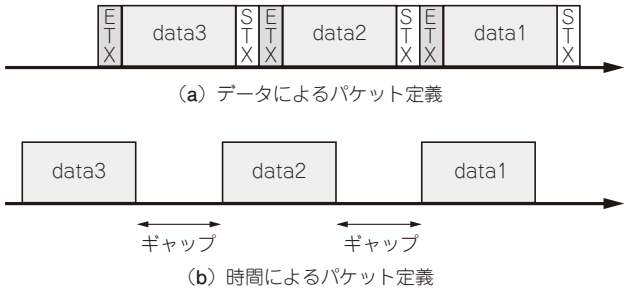


図 9.1
シリアル通信における論理的な正確さと時間的な正確さ

ト間ギャップ」を計測する必要があり，時間的な正確さが求められます．

それぞれの特徴として，データ領域の仕様では連続してパケットを送信することができますが，時間領域の仕様ではギャップを入れなくてはならないので連続通信は遅くなります．しかし，何らかの障害で通信が途絶えた場合，データ領域の仕様ではタイムアウト（一般には1〜3秒）まで待つ必要がありますが，時間領域の仕様ではパケット間ギャップ（一般には10〜30ms）経過を待つだけで受信データのチェックを行えるので，障害検出および対応が早くなります（**図 9.2**）．

● ビジネス系アプリケーションとリアルタイム・アプリケーションの違い

シリアル通信は簡単な例ですが，リアルタイム・アプリケーションの特徴をよく示しています．一般に，データ領域の仕様が多いビジネス系アプリケーションでは平常時のパフォーマンスを求められるのに対し，時間領域の正確さを求められるリアルタイム・アプリケーションでは障害検出も含めた最悪実行時間の短さが求められます．

しかし，この例のように時間領域の仕様が明確な場合はまれでしょう．普通は，時間領域の仕様を抽出することはたいへんな作業になります．例えば，ある化学プラントのタンクがあって，そのタンクの内圧Pがある値A以上になったときにアラームを出すという仕様があったとします．この仕様からただちに，

```
if(P >= A) then  圧力異常アラーム
```

とコードを書くことはできます．一見，これで十分なように見えますが，時間領域の振る舞いがまったく記述されていません．まず，このコードはいつ実行されればよいのかがわかりません．

そのほか，(P >= A)が成立したら，ただちにアラームを出してよいのかどうか，(P >= A)が成立しなくなったら，ただちにアラームを取り消してよいのかどうか，圧力Pの計測タイミング，この条件チェックはいつから始めていつまで続けるのか，なども考慮しなくてはなりません．

ここで示したコードは，ある特定のタイミングで起動された監視タスク内のデータ領域における仕様を記述しているにすぎません．タスクの外のことがわからないのです．一般的に，仕様書で「…のときに…」と表現することで，いつ起動するかという時間領域の仕様と値のモニタリング条件のようなデータ領域の仕様を同時に記述してしまうことがあります．

タスク内のプログラム構造や入力されたデータによって決められる「論理的制御フロー」と，タスク外の条件によって決められるタスクがいつ起動されるかという「時間的制御」は，明確に区別されなければなりません．

リアルタイム・アプリケーションの仕様とするためには，時間的制御に関する記述が必要です．そして，実装においては時間的正確さが求められます．

次に，いくつかの時間的正確さに関する事例を紹介します．

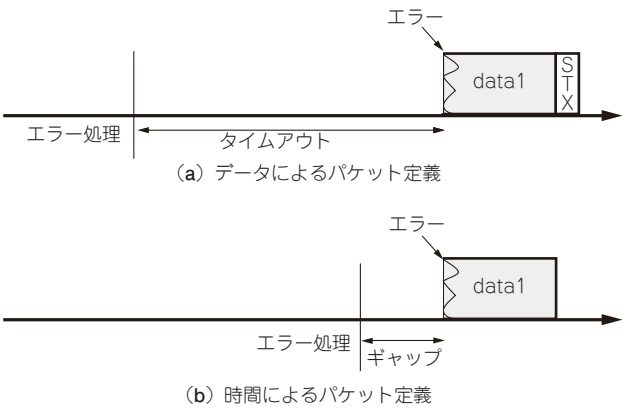


図 9.2
データ領域と時間領域におけるパケット定義の違い

第10章 リアルタイムOSに要求される機能

リアルタイム・アプリケーションは，時間的な正確さを実現しなければならないと前章で述べました．

しかし，現実にそのようなアプリケーションはどのようにしたら実現できるのでしょうか．また，そのためにRTOSがサポートすべき機能とは何になるのでしょうか．本章では，それらについて考えてみます．

10.1 最悪実行時間（WCET）

● 最悪実行時間と平均実行時間

リアルタイム・アプリケーションの時間的振る舞いを保証する唯一の方法は，プログラムの実行時間を事前に知ることです．プログラムの実行時間は入力データによって変動しますが，あらゆる条件でプログラムを実行した中でもっとも長い実行時間のことを最悪実行時間（Worst-Case Execution Time，WCET）[注1]といいます．

リアルタイム・アプリケーションでは，このWCETを予測できなければなりません．そして，WCETが0.5秒であるといえば，どのような条件の下でも，たとえエラーが起きた場合でも，0.5秒以内にエラーを通知し，実行を終了するということを意味しなければなりません．

ワープロや表計算プログラムでは，WCETよりも平均実行時間の短さが重要な評価基準になります．例えば，ワープロで文書を作成する場合には，標準的な長さの文書作成で快適な応答があることがまず必要であって，極端に長い文書で応答が悪くなることはあまり問題にされません．

しかし，「リアルタイム・ワープロ」が仮にあったとすれば，どのような長さの文書であっても，あらかじめわかっている時間，つまりWCET以内に応答を返すはずです．

● 最悪実行時間は計算不能

ところが一般のプログラムのWCETは，プログラムの停止性問題[注2]と同様に，計算不能であることが知られています．つまり，任意のプログラム・コードを読み込んで，そのプログラムの

注1：最悪CPU実行時間ともいう．

注2：停止性問題は，プログラムにある入力を入れると，そのプログラムは有限時間内に停止するかどうか，という問題．「プログラムが停止する」とは，そのプログラムがあらゆる入力に対して無限ループなどにはならずに終了するということである．アラン・チューリングが1936年に，停止性問題を解くチューリング機械が存在しないということを証明した． 他にも，あらゆる入力に対して出力があるかどうか，プログラムの特定の部分があらゆる入力に対して実行されるかどうか，二つのプログラムが同じかどうか，などを判定することも計算不能であることが知られている．

WCET を計算するプログラムを書くことはできません．このことは，論理的に証明されています．

そこで，WCET を計算するためには一般のプログラムを対象としないで，再起呼び出しは使わない，動的[注3]なデータ構造は使わない，などの制約条件を付けることが必要になってきます(1)．

動的なデータ構造とは，実行時にサイズが決まる配列とか任意長の文字列などです．動的なデータは，実行時にメモリの確保・開放を行わなければならないため，正確な実行時間を評価できません．

Java や C++ でオブジェクトを動的に生成・消滅させるプログラムの場合には，動的なデータ構造を使うことになるため WCET を計算できない場合があるということです．

ハード・リアルタイム用の言語である「Real-Time Euclid」(2) では，言語要件としてこれらの制限をもっています．さらにこの言語は，プログラマが各ループについて最大ループの回数を指定するようになっています．

`for` 文などではループの回数は自明だと思われるかもしれませんが，例えば，終了回数が変数で指定される場合には，その変数の最大値を指定します．あるいは，`While` 文でファイルから 1 バイトずつ `eof` まで読み出す場合には，最大ファイル長を指定します．これらの値を決めるには，アプリケーション分野の知識が必要になります．

● プログラムに対する制約とアプリケーション分野の情報

WCET を計算可能とするためには，プログラムに対する制約とアプリケーション分野の情報が必要になります（**表 10.1**）．Real-Time Euclid では，この指定に基づいてコンパイル時に実行時間を計算します．

C 言語のような言語を使う場合には，プログラムに上記の制約を規約として課してコーディングしなければなりません．そして，首尾よく解析可能なソース・コードができたら，次はプログラムの各部分に対してコンパイラがどのようなアセンブラ・コードを生成するかを調べる必要があります．そして最後に，そのアセンブラ・コードを実行するハードウェア環境を調べなければなりません．

例えば，プログラムを RAM で動かすのか ROM で動かすのかによっても，メモリ待ち時間が変わるため実行時間も変わります．デバイスからメモリに，CPU を介さずに直接データを転送する DMA（Direct Memory Access）を使用する場合，デバイスと CPU は並列動作ができます．しかし，過去によく使われたサイクル・スチール方式では，デバイスと CPU でバスのメモリ・サイクルが競合すると CPU は待たされます．待たされる時間は DMA デバイスに依存するので，予測することが困難です．

現在のプロセッサはパイプラインやキャッシュを使用しているので，実行時間が確率的に変動するため，予測がさらに難しくなります．これらの確率的要因があっても WCET を計算する手法(3)が提案されています．しかし，ソート・プログラムのようなループ回数などがデータに強く依存する場合には，実際の実行時間より 2 倍程度大き目に WCET を計算してしまいます．例として，参考文献 (3) に報告されている結果を**表 10.2** に示します．

この表から，パイプラインやキャッシュの効果

表 10.1　WCET に関係する要因

①	ソース・コード	アルゴリズム，制御構造，クリティカル・セクションなど
②	コンパイラ	どのような機械語を生成するか．最適化の有無など
③	ハードウェア	パイプライン処理，キャッシュ，メモリ，DMA など
④	OS	スケジューリング方式，同期方式，メモリ管理法，割り込みハンドリングなど

注 3：「動的」とは，「プログラムの実行時」にという意味である．これに対して，コンパイル時に決まるものは「静的」と呼ばれる．

第12章

さまざまなリアルタイム・スケジューリングの手法

第10章では，時間的制約を満たすために必要なRTOSの機能を実現させる方法について説明しました．そして，タスクについて単なるプライオリティ以外の時間的属性が必要であることを述べました．本章では，そのような時間的属性が与えられているタスクのスケジューリング法について説明します．ここで取り上げるスケジューリング法は，時間的制約を満足させることを主目的とするため，「リアルタイム・スケジューリング法」と呼ばれます．

12.1 NP完全問題

タスク属性のうち，最悪実行時間（WCET）を一般のプログラムについて求めることは不可能（計算不能）[注1]であることをすでに述べましたが，たとえWCETがわかっていたとしても，一般の場合のタスク・スケジューリング問題は計算困難であることが知られています．

一般の場合では，WCETは計算不能で，スケジュール作りは計算困難なのです．計算困難というのは，タスクの数が少ないうちは計算できるものの，タスクの数が増えて，例えば10ぐらいになると，とたんに計算できなくなるということです．

計算できないというのは，厳密にいうと計算アルゴリズムはあるものの計算が終了するまでに，例えば100年かかるなど，時間がかかりすぎて実用的な範囲で計算できないということです．

「計算困難」という呼び方は，計算時間が問題のサイズ，この場合はタスクの数とともに指数関数的に増える場合に使います．スケジュール問題は，「NP完全」（NP-complete）(1)という問題クラス[注2]に属していて，現在のところ，このクラスに属する問題は指数関数的に時間がかかると考えられています．

ただし，NP完全問題が一つでも短い時間（多項式時間）で計算できれば，すべてのNP完全問題も多項式時間で計算できるという予想があります．したがって，将来はわかりませんが，とにかく現状では一般のスケジュール問題は手に負えないことがわかっています．

しかし，このままではリアルタイム・システムを作れないので，WCETの場合と同様に，タスク

注1：計算するためのアルゴリズム自身が存在しないということ．

注2：NPハード（困難）問題は，計算機科学の一分野である計算複雑性理論における問題の集合の一つ．NP（Non-deterministic Polynomial）は，問題がある性質を持つかどうかを判定したいときに，持つことを証明できる問題を集めた集合のこと．NPハード問題は，NPに属するどの問題に対しても同等以上に難しい問題の集合．NPハード問題の中で，NPにも属する問題を「NP完全問題」という．

について制限を設けてスケジュールできるようにしています．制限としては，周期的タスクのみを扱うとか，プライオリティを固定するなどします．

この制限の仕方によって，いろいろなスケジューリング方法があります．

12.2 タスクの時間的属性

タスクの時間的属性にはどのようなものがあるかをまとめると，**図 12.1** のようになります．ここで，添え字iは，i番目のタスクという意味です．

この他に，周期的（periodic）タスク，非周期的（aperiodic）タスク，散発的（sporadic）タスクの区別があります．

周期タスクというのは，一定間隔で規則的に起動されるタスクです．非周期的タスクは，不規則に外部要求があるたびに起動されるタスクです．散発的タスクは非周期的タスクの一種ですが，最小起動間隔があるような場合をいいます．つまり，一度起動されるとしばらくは起動されないことが保証されるタスクです．また，これらの分類に関連して，周期 T_i，起動間隔，フェーズ F_i などの属性が発生します．

フェーズとは，周期的タスクがいちばん初めに起動された時刻のことです．起動されたタスクの一つ一つの実行を区別したいときには，そのタスクのインスタンスあるいはジョブと呼ぶことがあります．

12.3 スケジューリング方式の分類

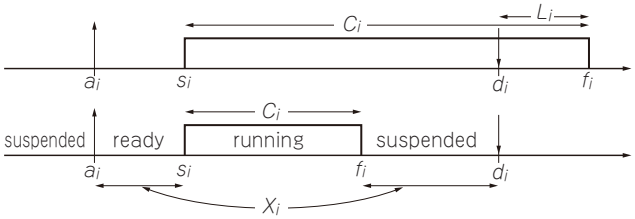
● オンライン / オフライン，動的 / 静的，最適化 / 経験的

スケジュール方法の分類には，いろいろな見方があります．例えば，オンライン方式とオフライン方式があります．

オフライン方式というのは，コンパイル時にタスクの起動時刻などを決めてしまう方式です．この方法では，スーパコンピュータなどを使って最適化されたスケジュールを作ることも可能で，しかも実行時にはオーバヘッドが少ないのですが，その代わり柔軟性がありません．

タスクをプリエンプトするかしないかということも，スケジューリング方式の違いです．プリエンプトしない方がオーバヘッドは少ないものの，スケジュール問題としては格段に難しくなります．

その他に，動的か静的か，最適化か経験的かなどの違いがあります．動的というのは，スケジューラが起動されるたびに各タスクのプライオ



C_i	タスク実行時間	プリエンプトやブロックされない純粋な処理に必要な時間
a_i	起動時刻	タスクに対して実行要求が出された時刻
d_i	デッドライン時刻	実行終了の締め切り時刻．相対時間$D_i = d_i - a_i$を使用することもある．単にデッドラインといった場合，絶対デッドラインd_i の場合と相対デッドラインD_i の場合がある．文脈から判断できない場合は明記する必要がある
s_i	実行開始時刻	タスクが実際にrunning状態になった時刻
f_i	実行終了時刻	タスクが実行を終了した時刻
L_i	遅れ時間	$f_i - d_i$．デッドラインをオーバした時間
X_i	遊び時間	$D_i - C_i$．余裕時間ともいう

図 12.1
タスクの時間的属性

アーキテクチャ分析やモデリングの基礎から
リアルタイム・システムの考え方まで

組み込みシステム開発に役立つ理論と手法

Interface増刊　組み込みシステム開発に役立つ理論と手法
CQ出版社
〒170-8461 東京都豊島区巣鴨1-14-2
TEL.03-5395-2141（販売部）
定価2,310円 本体2,200円

雑誌　01620-5
Ⓛ 2012.6/18

4910016200522
02200