第1章　時間スケール，ハードウェアとソフトウェアの切り分け，協調モデル
組み込みソフトウェアの特徴

組み込みソフトウェアを開発するには，ハードウェア開発者や関連する周辺分野の技術者と上手に連携する必要がある．そのためには，ソフトウェア開発者はハードウェアを中心とした周辺分野についてある程度の知識をもたなければならない．そのうえで，RTOSやオブジェクト指向，形式的手法を使いこなしてソフトウェアを設計・実装する必要がある．

本書では，組み込みソフトウェアを開発するうえで注意しなければならない周辺知識を交えながら，RTOSなしの場合や，RTOSを利用する場合のソフトウェア開発手法をまとめた．まず本章では，組み込みソフトウェアの特徴についてレビューする．

1　時間スケールの多様さ

● 秒単位からナノ秒単位までをインターフェースする

普段，我々が使用している時間の最小単位は秒である．それに対して，組み込みソフトウェアが動作する環境の時間の単位は，だいたいミリ秒(ms)からマイクロ秒(μs)となる．たとえば，「割り込み遅延時間は10μsである」とか「最長割り込みマスク時間は15μsである」などという言い方がされる．さらに，組み込みハードウェアの時間の単位は，ミリ秒からナノ秒(ns)のレベルだ．

組み込みソフトウェアは，制御対象の時間スケールと組み込みハードウェアの時間スケールとのインターフェースを取らなければならない．すなわち，扱わなければならない時間の範囲が非常に広い．普通の文章では，表現しきれないほど広いのだ．

● 瞬間か？ 継続か？ 日本語表現から見る動作の解釈

動詞には，瞬間動詞，継続動詞，状態動詞の3種類がある．たとえば，「起こる」，「倒れる」，「見かける」などは瞬間動詞に分類されている．したがって，単純に考えれば瞬間動詞の動作時間は「一瞬」と認識される時間になる．標準語では，瞬間動詞に「〜ている」を付けると動作継続を表す場合と，結果継続を表す場合の2種類がある(**図1**)．

どちらの意味になるかは，状況と動詞の種類に依存する．たとえば，パソコンを「起動している」と言うと，今ではほとんど動作継続と解釈される．もし，パソコンが本当に一瞬で起動できれば，「起動している」といえば結果継続の意味になり，いつでも使えることを意味する．実際は，「起動しているところなので少し待ってください」の意味になることが多いのだが….

さらに，「押す」のように結果を意識しない瞬間動詞の場合は，反復，習慣，経歴，経験などの意味になる．たとえば，エレベータのボタンを「押している」と言う場合には，何度も押していることを表したりもする．

日常生活と組み込みソフトウェアと組み込みハード

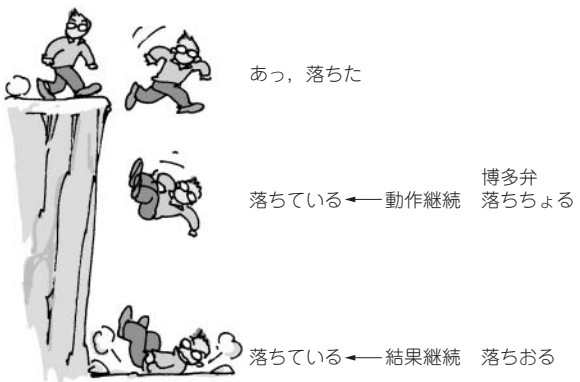


図1　動作継続と結果継続のちがい

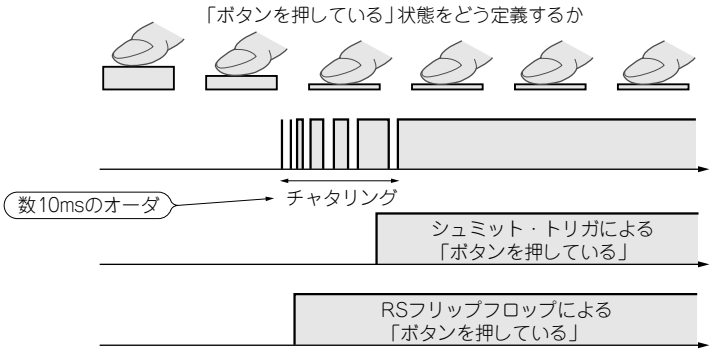


図2
ボタンを押したときの
チャタリング

ウェアでは，瞬間を意識する時間スケールが異なるため，動作継続か結果継続か繰り返しなのかがあいまいになる．基本的には仕様書は日常生活レベルの用語で書かれるので，イベントの順番などがあいまいになりやすく，これを明確にするためには，タイミング・チャートなどで補足しなければならない．

● チャタリングを人間から見た場合とソフトウェアから見た場合の違い

電源をON/OFFする場合を考えてみよう．組み込みソフトウェアの時間スケールで見ると，ボタンを押した直後は，接点でON/OFFを繰り返す現象が起こっている．ディジタル的に見れば，この過渡期的な状態はON/OFFを繰り返しているように見える．しかし，アナログ的に見れば，しだいに電圧が変化しているようにみえる．この現象は，チャタリング(chattering)とか，接点はずみなどと呼ばれている(**図2**)．

チャタリングが起きると，ボタンを押された回数を数えている場合には，数えすぎてしまうことになる．ボタンでCPUに割り込みをかけているような場合には，動作が不安定になる．日常生活レベルでは，ボタンを押すのは一瞬だが，組み込みソフトウェアでは時間スケールが拡大されるため，本当の目的が伝わらないかもしれない．動作継続と結果継続の両方の可能性を検討しなければならないのだ．

チャタリングを取り除くには，ソフトウェアで処理する場合と，ハードウェアで処理する場合の2種類がある．さらにハードウェアで行うには，フリップフロップを使う場合と，シュミット・トリガ回路を使う場合がある．フリップフロップは一定以上の幅のパルスがあればボタンを押されたと認識するが，シュミット・トリガ回路の場合は電圧がしきい値以上になるとボタンを押されたと認識する．ハードウェアの時間スケールは，組み込みソフトウェアの時間スケールより拡大されて，一様だと思っていた時間間隔の中に，さらにイベント順などの順序構造が入ってくる．つまり，ハードウェアについて考える場合には，ソフトウェアより短いタイム・スケールでその違いを検討しなければならない．

たとえば，**図2**では電圧の変化を垂直線で表現しているが，アナログの世界では**図3**の下のような曲線になるし，ハードウェアどうしの動きを記述するバス・

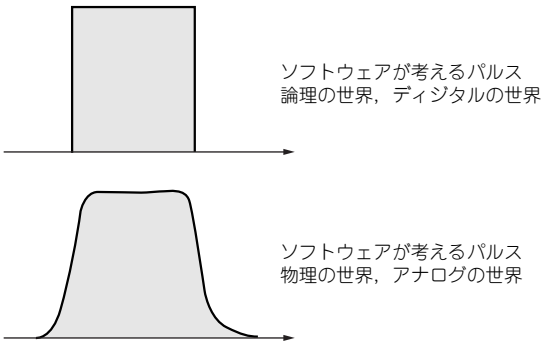


図3　ディジタル世界のパルスとアナログ世界のパルス

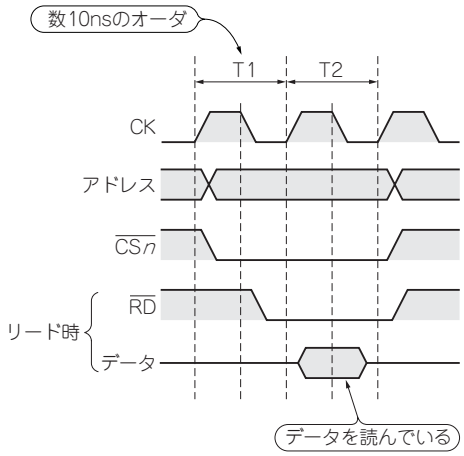
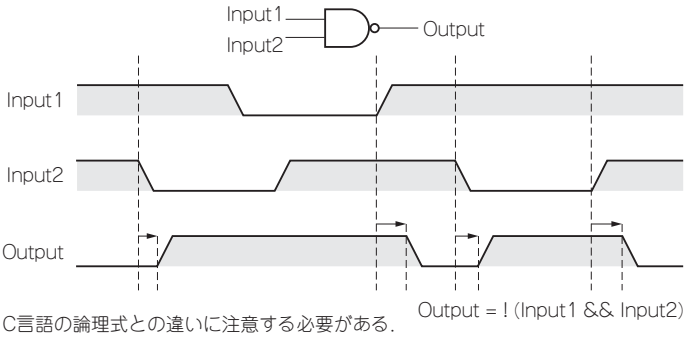
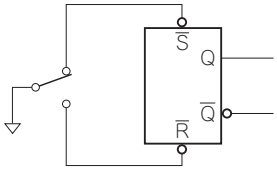


図4　バス・タイミング・チャートの例

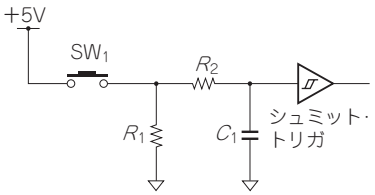


C言語の論理式との違いに注意する必要がある．
実世界にはタイミングがある

図5　NANDゲートのタイミング・チャート



（**a**）RSフリップフロップ――ノイズに弱い



（**b**）*CR*回路とシュミット・トリガ――ノイズに強い

図6　チャタリング除去回路の例
――どちらを採用すべきか？

タイミング・チャートなどでは**図4**や**図5**のように傾きのある線で表現する．

組み込みソフトウェアの仕様を記述するには，イベント順などの時間的な表現が重要になる．これを正確に表すときには，自然言語はあまり頼りにならず，図や数式を使用して表現する必要がある．数式を毛嫌いする人が多いのだが，自然言語よりはずっと正確で簡潔だ．また，文章で書くよりも短時間で済むというメリットもある．文章は，あいまいなくせに，時間がかかる厄介なしろものなのだ．

2　ハードウェアとソフトウェアの切り分け

● ソフトでやるべきか？　ハードでやるべきか？

ボタンのチャタリングを取り除くことは，ソフトウェアでもハードウェアでも行うことができる．どちらで行うかを決めるのは，システム設計者の仕事になる．現在は，ハードウェア担当者が行うことがほとんどだが，本来ならば，ソフトウェア開発者もシステム設計に関与すべきである．

システム開発コストにおいてソフトウェア開発費はかなりの比重を占めている．新しいボードを開発するとき，鉛フリー化にともなってボードを再設計したところが多かったと思うが，このようなとき，ハードウェア技術者がソフトウェア技術者に希望を聞いても，ソフトウェア技術者からはあまり意見が出てこない．しかし，RTOSやC++を使えるようにメモリを増やして欲しいとか，デバッグ用にシリアル・ポートを追加して欲しいとか，余ったI/Oポートを引き出しておいて欲しいとか，そこは割り込みにして欲しいとか，ソフトウェア技術者が言い出さないとだれも検討してくれない．こういった要求をしておけば，たとえば，RTOSやC++を使ってソフトウェアの再利用性が向上させられ，並行開発もやりやすくなる．デバッグ/テスト工程をどのように短く，効率的にするか検討することは，開発期間短縮と信頼性向上には欠かせない．イベント・ドリブンなアプリケーションは，ポーリングだけでは実現できない．これらの意見は，システム開発コストの削減のためにも考慮されるべきものだ．

図6はチャタリング除去回路の例だ．このように同じ目的を実現するためには複数の方法がある．どちらの方法を用いるか，ハードウェア技術者に任せっきりではなく，ソフトウェア技術者も積極的に意見を述べるべきであろう．

さらに，すでにハードウェア自身をVerilog-HDL（Verilog Hardware Description Language，**図7**）やVHDL（VHSIC Hardware Description Language，**図8**）などのソース・コードとして記述する方向にある．つまり，ハードウェアをコーディングする時代になってきた．ハードウェアとソフトウェアの道具立ては似てきたが，ソフトウェア技術者がハードウェアをコーディングできるわけではない．ハードウェアに物申すソフトウェア技術者を増やさないと，システム設計は効率的に進まないだろう．

システム設計にソフトウェア開発者が参加することで，ソフトウェアも含めた最適化が可能になる．また，最終的なシステム・テストでもソフトウェアの手法を応用できるなどのメリットが得られることになる．

● 時代の流れによる切り分けの変化

より広い意味でのシステム設計は，電気回路とソフ

```
構造記述
module RSFF ( R, S, Q, Q_B );
    input R, S;
    output Q, Q_B;
    nor( Q , R, Q_B );
    nor( Q_B, S, Q );
endmodule


動作記述
module RSFF( R, S, Q, Q_B );
input R, S;
output Q, Q_B;
reg Q, Q_B;
always @( R or S )
    case({ R , S })
        1:begin Q <= 1; Q_B <= 0; end
        2:begin Q <= 0; Q_B <= 1; end
        3:begin Q <= 0; Q_B <= 0; end
    endcase
endmodule
```

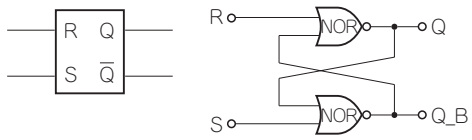


図7　Verilog-HDLで記述した例（NORの場合）

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RS_FF is
    port ( R, S     : in  std_logic
           Q, Qnot  : out std_logic );
end RS_FF;

architecture STRUCTURE of RS_FF is

signal S1, S2 : std_logic;

begin
    S1      <= R nor S2;
    S2      <= S nor S1;
    Q       <= S1;
    Qnot    <= S1;
end STRUCTURE;
```

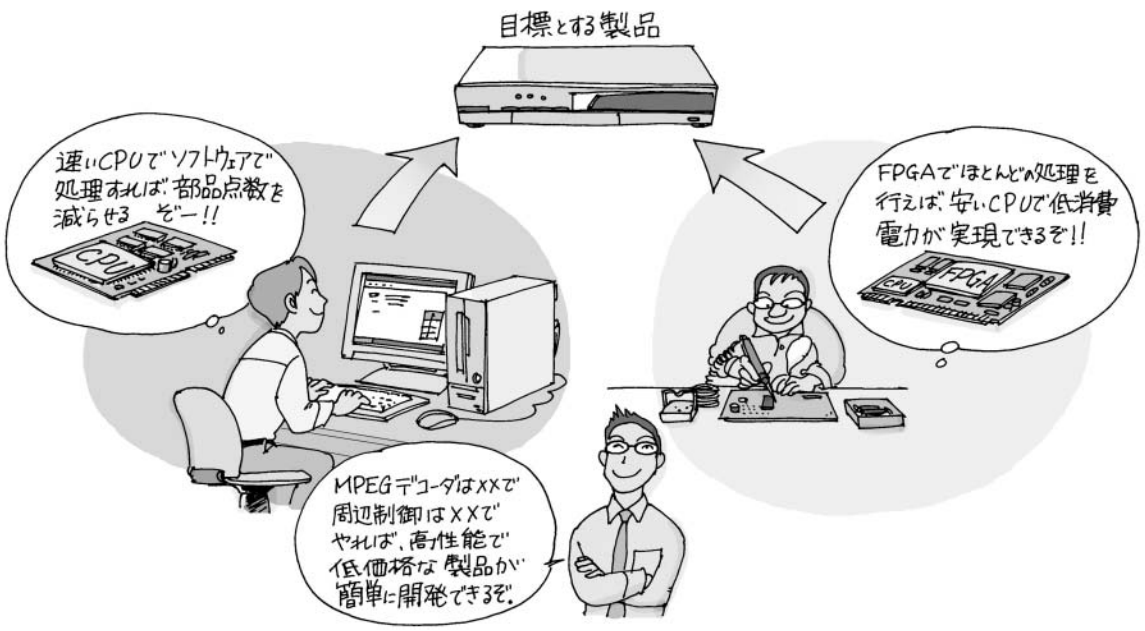
図8　VHDLで記述した例

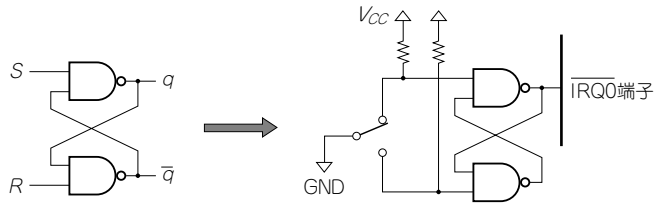
トウェアの切り分けだけではなく，いわゆるメカとエレキとソフトウェアの切り分けまでを指す．今のところ，開発コストと製品のパフォーマンスが切り分けのキー・ポイントになっている．開発コストは，製品のライフタイム全体の中でだれが何をするのかによって判断しなければならない．ソフトウェアで実現すれば，ハードウェアの部品代が浮く，という時代ではない．パフォーマンスについては，設計の段階でどのように見積もるかが重要になってくる．

3　想定外を想定する

● 想定外の状況が発生しても大丈夫なシステムとは

世の中には，何が起きても「想定内」にしてしまうタフな人がいる．良くできた組み込みソフトウェアも，当然タフでなければならない．想定外の状況になっても，想定外になることを想定しておけばよい，した





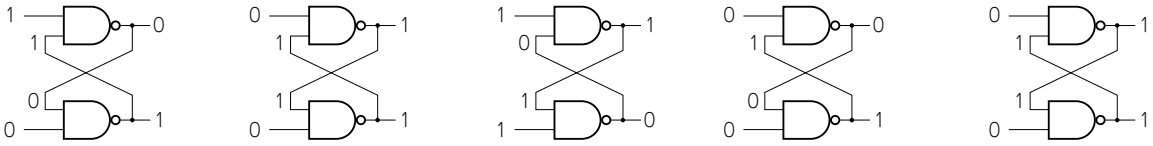


図9　RSフリップフロップの入力禁止

がって，想定内なのである．

たとえば，**図9**のNANDゲートによるフリップフロップで，

- 入力がS＝‘1’，R＝‘0’の場合，最初はqが‘1’でも‘0’でも，最終的な安定状態ではq＝‘0’となる
- その後，R＝‘1’に変化しても，そのままq＝‘0’を保つ
- 次に，R＝‘1’のままでS＝‘0’とすると，q＝‘1’となり，安定化する
- そして，S＝‘1’にしてもq＝‘1’が保存される
- つまり，S＝‘1’，R＝‘0’でq＝‘0’が確定し，S＝‘1’，R＝‘1’になってもq＝‘0’は保存される
- S＝‘0’，R＝‘1’でq＝‘1’が確定し，S＝‘1’，R＝‘1’になってもq＝‘1’は保存される

これは，NANDの真理値表を傍らに置いて紙と鉛筆で確認できることである．

ところが，たとえば，S＝‘1’，R＝‘0’でq＝‘0’が確定した後で，S＝‘0’，R＝‘0’とするとq＝‘1’となり，その後，S＝‘1’，R＝‘1’になってもq＝‘1’もq＝‘0’も取りうる可能性があり不定となってしまう．すなわち，S＝R＝‘0’になることは，1ビットの情報を記憶するということに対しては想定外なのである．

しかし，想定外であることがわかった瞬間，想定外ではなくなり対応可能となる．たとえば，**図9**に示したスイッチによって割り込みを発生させる回路のように，SとRをプルアップして，さらに，同時にGNDにならないようなスイッチを使用する．

想定外を想定するためには，すべてのケースを網羅くふうが必要になる．そのためには，網羅的に状態の組み合わせを作ってくれるモデル・チェッキングのような形式的手法が有効だ．

4　協調しながら同時に動かす

組み込みシステムの特徴の一つは，同時に動く複数のものから構成されるということだ．そして，ただ同時に動くだけではなく，与えられた目的を実現するために協調して動く．ここで，「動くもの」とはCPUやデバイスなどのハードウェア，スレッドやプロセスなどのソフトウェアだ．そして，協調して動くということは互いの動作を監視したり，タイミングを合わせたりするということだ．

たとえば，本を読んでいて，あるページを読み終われば，次のページに進む．そのためにはページをめくることになる．ページをどうやってめくるかについては，ほとんど意識しなくてもだれでもできることだ．つまり，どうやって腕を伸ばすのか，どの指を使うのか，どうやってその指を曲げるのか…我々は，これらの制御を同時に行っている．指を曲げるときには，内側の筋肉を収縮させると同時に外側の筋肉を弛緩させなければならない．どこにどのような筋肉が付いているかはまったく知らないけれども，協調しながら同時に動くように実装されている．「ページをめくりたい」と思っただけで，あるいはそのようなことを考えることもなく，ただ本を読みたいと思うだけで自動運転されている．どうして自動運転できるのだろうと，考えすぎると何もできなくなってしまうが…．

このような組み込みシステムを作りたいものだ．そのためには，協調しながら同時に動くものを扱う方法を学ばなければならない．

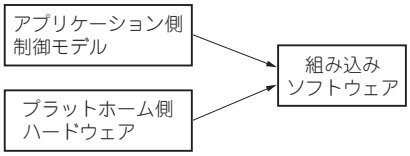


図10　二つの視点――アプリケーション側の制御モデルとプラットホーム側の制御モデル

5 制御モデルの作成を第一に考える

● 制御モデルの作成が重要である

実際に動くものを作ろうとすると，ハードウェアやソフトウェアが「わかった」だけでは，何もできない．たとえば，倒立振子を作ろうと，力学モデルを作って，運動方程式を解いて，それに対して制御モデルを作って，それをソフトウェアで実現するとしよう．ソフトウェアで実現するときに，ハードウェアとソフトウェアの知識がはじめて役に立つ．そして実装にあたっては，アプリケーション側の制御モデルと，プラットホーム側の制御モデルの二つの視点が必要になる(**図10**)．

組み込みソフトウェアの要求仕様には，制御モデル(**図11**)からの要求が入ってくる．たとえば，デッドライン要求などは制御モデルから出てくる．また，どのようなフィードバック系を作れば系が速く安定化するかは，制御モデルを見れば解答が書いてあり，これを使えば魔法のように解いてくれる．とは言っても過信してはいけない．メカやエレキの技術者が，ソフトウェア技術者に言えば何でもやってくれると誤解しているのと同じことになってしまう．どのような制御モデルでも，最後の詰めは試行錯誤で勝負することになる．制御技術者もソフトウェア技術者もそこから，学ぶのだ．

業務系で言うなら，ビジネス・ルールにあたる部分が，組み込みソフトウェアでは物理モデルと制御モデルになる．このアプリケーション側からどのように要求を抽出するか，さらに，ソフトウェアでできることを逆に提案して要求自身を開発していくことがこれからは重要になってくる．

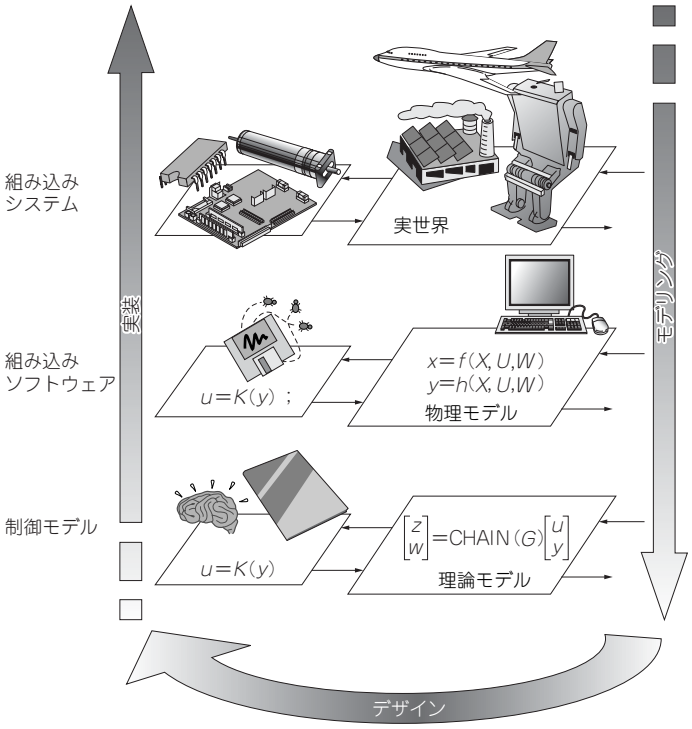


図11　制御モデル――最後の詰めは試行錯誤で勝負

● 制御技術者とも話をしよう

メカ，エレキ，ソフトのほかに，制御技術者といわれる人たちとも話ができなければならない．制御技術者は，MATLABなどを使って普段からシミュレーションを行い，ソフトウェア面には詳しい人達だ．したがって，メカやエレキの技術者よりは，話しやすいだろう．

しかし，扱っている対象が微分方程式などの解析風のものが多く，変数はたいていfloatだったりする．それに実時間ではなくコンピュータの中に作り出した架空の時間スケールでモデルを動かしているなどの違いがある．制御技術者と話をすると，普段，ハードウェア技術者たちがどのような気持ちでソフトウェア技術者と接しているか少しだけわかったりする．

結局，ソフトウェア技術者はいろいろなものの間に入ってインターフェースをとるのが仕事になる．時間のギャップを埋めて，アナログとディジタルの間を取りもって，制御モデルをプラットホームに載せ，などなど八面六臂の日々が続く．ゆえにシステム設計に一番近いところにいるのではないかと思われるのである．