●特集 1「組込みソフト設計入門（仮）」

■■■■■■■■■■■■■■■■■■■■■■

■第 1 章　組込みソフトとは

■■■■■■■■■■■■■■■■■■■■■■

1. はじめに

本特集では組込みシステムの開発の初心者の方を対象としています．初心者とはいっても，他の分野のシステム開発経験は豊富で，単に，組込みの経験がないだけ，という方もおられることと思います．

本特集は，全くの初心者の方には業界の様子が分かるような情報提供になるでしょうし，他の分野の経験がある方に対しては，様子が違うところが分かっていただけるような情報の提供になると思います．

おそらく，組込みシステムの開発に関しては電気の知識や回路図の読解能力が必要なイメージがあるのではないかと思います．でも，心配する必要はありません．本特集では回路図や電気，電子の知識が必要な部分は「ほとんど」ありません．

本来的には「ソフトウェア設計」にそのような知識が必要な部分は非常に限られています．技術領域を抽象化して分類をしてしまえば，十分に組込みソフトの設計に携わることができるようになるものと思います．

具体的には，一般的には方法論があいまいな単なる「ブロック図」を道具として使っていきます．
約束事があいまいな道具を使うことにより，厳密な決まりごとを守らないと開発に着手できないのではないかという先入観から解放できますし，文系のエンジニアの方をはじめとする，組込み初心者の方も比較的入りやすいのではないかと思います．

電気の知識が必要ではない仕事を行っていた方に対して，電気の知識についていくら解説したとしても，共通認識が得られることは非常に難しいことだと思います．特にハードウェアの設計，開発を行う，いわゆる，「ハード屋さん」との間の文化の溝は埋めることは想像以上に難しいでしょう．

その文化の溝を埋める為に勉強に費やす時間があるのであれば，どんどん組込みシステムの開発現場で経験を積んだ方がずっとましな時間の過ごし方だと思います．

また，通常は正面からは採り上げることがない，組込みソフトに関わる業界の構造を紹介，考察し，単なる開発知識だけでは解決できない問題についても整理し，どのようにその問題を回避するのかという，解決策のヒントも提示したいと思います．

これらを行うことにより，組込みソフト設計の勘所を伝えることができれば幸いです．なお，本特集での「組込みソフト」とは，後述する「狭義の組込みソフト」を指しています．

2. そもそも組込みソフトとは

本特集では組込みシステムの開発や製品開発という言葉ではなく，「組込みソフト」という表現を使っています．これは，あくまでもソフトウェア開発という立場で組込みシステムの設計を捉えようとしているからです．

2.1.広義の組込みソフト

「広義の組込みソフト」の代表的なところでは，携帯電話やデジタルカメラなどの開発が考えられます．
このような，広い意味での組込みソフトの定義は以下のようになります．

・見た目が小さい機器に組込まれるソフト
・　内部構成や大きい小さいに関わりなく，機器に組込まれているというソフト

この領域の「組込みソフト」の開発では後述する「狭義の組込み」の知識や経験がなくても設計，実装ができるということが大きな特徴です．

使用される OS も WindowsCE や Linux などの場

合があり，ハードウェアを隠蔽したドライバとAPIが適切に存在していれば多くのアプリケーションの開発に困ることはありません．この場合，リアルタイム制御設計を余り厳密に行わなくても影響が出ないほどの圧倒的に性能の高いプロセッサの使用が許可され，メモリも比較的贅沢に使える，という暗黙の条件も追加されるでしょう．

開発を担当するエンジニアも組込み開発の経験がなくてもほとんどの場合は問題なく開発を行うことが出来ます．

さて，このような開発の場合でも，その中のほんの一部分の「狭義の組込み」が足かせとなって，開発全体に影響を及ぼす可能性が考えられます．それも，影響があることが気付かれないまま問題が潜んでしまう可能性があります．

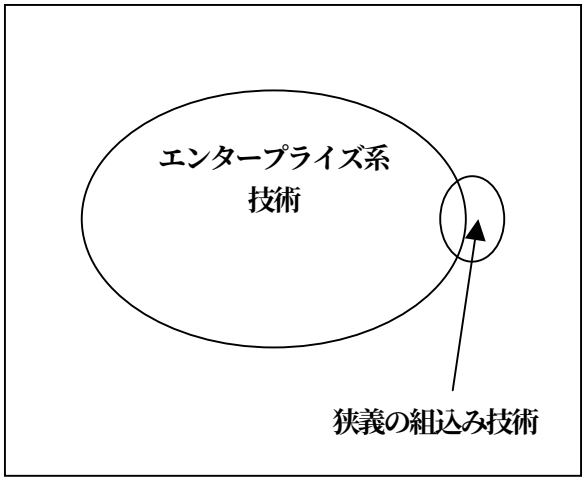
「狭義の組込み」技術の必要性を感じないままプロジェクトを推進してしまい，その領域の開発を独立して担当できるエンジニアが存在しないと，開発の最終段階まで問題は表に出てくることはないと思います．

しかし，「狭義の組込み」技術の必要性を感じていながら，仕方なく，そのようなエンジニアが不在なままプロジェクトの推進を行っている場合は，問題意識を持ちつづけることにより，開発の最終段階にまで問題はの発覚が持ち越されることはないと思います．

要は，問題意識があるかどうかが，プロジェクトの成否を分けるといっても過言ではありません．

但し，「狭義の組込み」領域を専門とするエンジニアを確保できたとしても，全く問題がないわけではありません．それは「狭義の組込み」は分かっても，WindowsCEやLinux上でのドライバの実装方法が分からないエンジニアもまた，多い，という問題です．この場合の一番の問題は「多勢に無勢」で，「狭義の組込みエンジニア」の意見が通らなくなる可能性が高いということです．

図 1-1　広義の組込みに必要な技術領域



是非，開発の視点が「広義の組込み」と「狭義の組込み」では異なる部分があるということを認識していただき，このような事態には発展しないようにしていただきたいと思います．

さて，広義の組込みソフトの開発の領域と狭義の組込みソフト開発の領域のイメージは図1-1のようになります．

ここでは狭義の組込み以外の開発技術を総称して「エンタープライズ系技術」と呼びます．

製品開発という意味ではほとんどがエンタープライズ系のソフトウェアエンジニアが開発することになりますが，この領域のソフトウェアは本特集では「組込みソフト」には分類しません．特に断りのない限り，「組込みソフト」とは「狭義の組込みソフト」のことを指します．

「組込み開発の大規模化」というような表現の場合は，広義の組込みでの全体量のことを言っている場合が多いと思います．エンタープライズ系での開発領域での開発量と狭義の組込み領域での開発量を区別せずに積算してしまうと，コード量が膨大になるのは非常に自然なことだと思います．但し，前述した「狭義の組込みの認識を行わないまま広義の組

図 1-2　広義の組込みソフトの階層

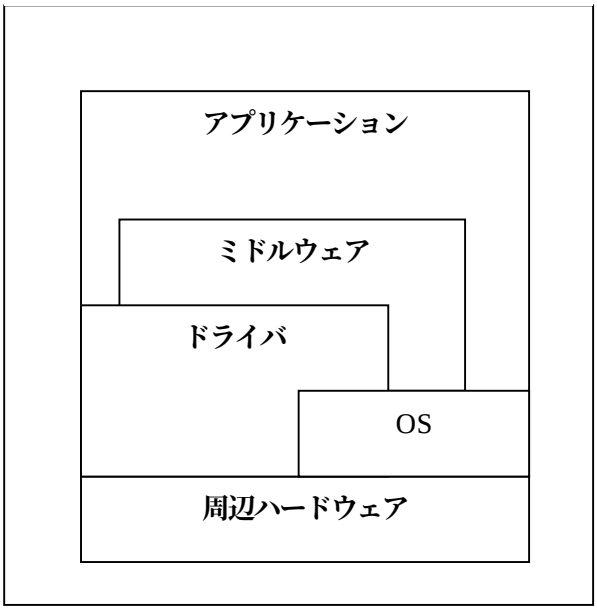
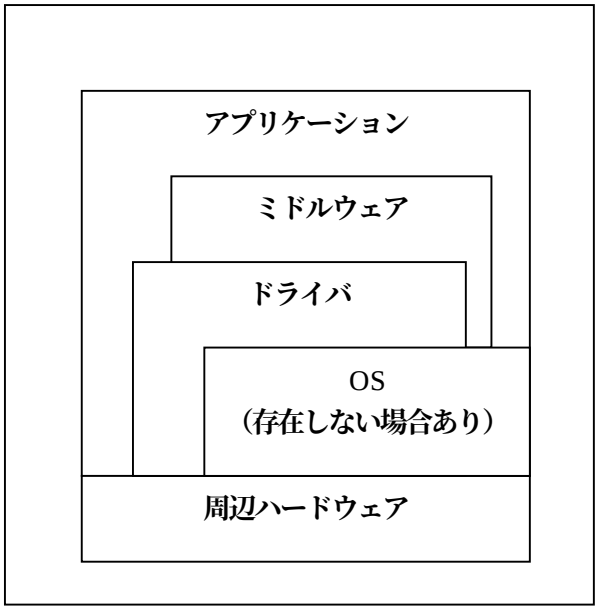


図 1-3　狭義の組込みソフトの階層



込みの開発プロジェクトを推進した場合の問題」のことを指摘しているのであれば，確かに，その通りだと思います．

図 1-2 に広義の組込みソフトの階層を示します．
アプリケーションはミドルウェア，ドライバ，OS だけを相手にすれば良いことを示しています．ミドルウェアが相手にするのはドライバと OS でアプリケーションに機能を提供します．ドライバは周辺ハードウェアを OS と協調して直接制御し，ミドルウェアやアプリケーションへ機能を提供します．OS は必ず存在し，ドライバ，ミドルウェア，アプリケーションに機能を提供します．この階層構造の中で図 1-1 で示したような狭義の組込み技術に対応したソフトの部分は以下になります．

・ドライバ
・OS（ブートローダ）
・OS（割り込み処理周辺）
・OS（I/O 周辺）

2.2.狭義の組込みソフト

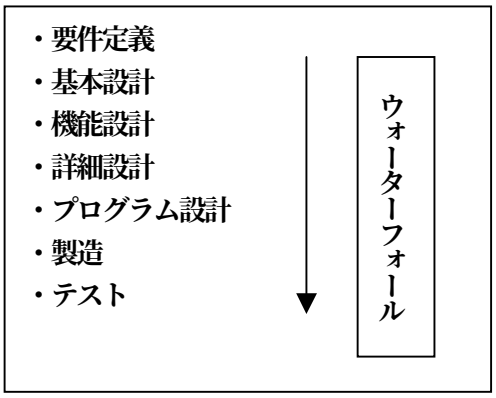
狭い意味での組込みソフトの定義は以下のようになります．

・特定のデバイスの制御を行うソフト
・特定の外部接続されたハードウェアを制御する
・OS を使わないリアルタイム制御のソフト
・OS そのもの
・ドライバ

図 1-3 に狭義の組込みソフトの階層を示します．図 1-2 との大きな違いは見られません．アプリケーションの役割分担の範囲が変わっているだけに見えます．
そう，アプリケーションはミドルウェアやドライバを介して処理する場合だけではなく，直接，周辺ハードウェアを制御する，という部分が異なるだけです．OS がない場合は特にミドルウェアやドライバとの境界が曖昧になります．明確なミドルウェアやドライバの概念が存在しない場合もあります．要するにハードウェアを直接制御するソフトウェアは全て狭義の組込みソフトに分類します．

3. 組込みソフト開発とは

本特集での組込みソフトは前述したように「狭義の組込みソフト」を意味します．その開発とはエンタープライズ系と同様に，大まかには以下の工程に分類されます．工程の呼び方と内容の対応付けは組織により変わってくると思いますが，とりあえず，以下の呼び方をしてみます．



順番に上位の設計から下位の設計，実装を行う典型的なウォーターフォールモデル①です．組込み開発に限っては試行錯誤しながら顧客の要求を確かめて，プロトタイピングしながら開発を行うようなスパイラルモデルは馴染みにくいようです．「製品企画」「マネージメント」「品質管理」「製造」など複数の組織（部署）が絡みあって製品の開発を行うような場合，全ての組織を巻き込む結果となるスパイラルモデルは現実的には実現が難しい②モデルだからです．

これらの開発工程を見ると，昔ながらのエンタープライズ系ソフトウェア開発工程が並んでいるだけのように見えますが，それぞれの工程で検討すべき項目がエンタープライズ系とは異なってきます．
また，広義の組込みソフト開発では UML などでのモデル化も可能ですが，狭義の組込みソフト開発では非常に適用が困難です．後述する組織の問題を知ることでそのことがだんだんと理解できてくるものと思います．

3.1. 要件定義

さて，開発工程の最初の段階，要件定義は製品の機能，ハードウェアの構成，CPU やメモリの条件，周辺ハードウェアの条件などを明確にします．開発するコード量，データ領域，RAM の使用量，スタックの使用量，ROM の使用量などの見積もりも行います．

また，量産，製品検査，出荷後の不具合改修などのための仕組みについても検討しておく必要があります．

例えば，実際に市場に出た後に不具合が発生した場合，プログラムを格納する ROM③の形式がマスク ROM の場合はその部分の ROM の書き換えは出来ません．その上で顧客から回収した不具合のある製品を正常にするためには何らかの手段を考えておく必要があります．

不具合のある製品を交換すれば良い？

いえ，同じ交換するにしても新しく交換のために用意した機器のプログラム部分だけを改修するためには，結局は何らかの仕組みが必要です．もし，新しいマスク ROM を用意するしか方法がない場合，以下のようなことになってしまいます．

・ソフトウェアの修正を行う
・プログラムを ROM に書き込めるイメージの形式にしてメモリメーカに発注
・それなりの数のマスク ROM を製造

---

① 滝のイメージで，水は高いところから低いところに落ちるが，低いところから高いところに戻ることは出来ない．
② 研究開発としての試作は別です．そのような開発では試行錯誤はあたりまえです．関連する部署も研究開発部署だけになります．

③ Read Only Memory の略．C 言語的には const 領域のこと．マスク ROM はメモリチップメーカが製造して内容を後から書き換えることはできない．

・マスク ROM が納品される
・量産ラインで組込み機器を製造
・製品の製造後の品質検査
・動作確認

上記の手順を踏むためには多くの人員と時間が必要です．こんなに時間をかけていたらそのメーカの信用はなくなって，売り上げにも響くことになるでしょう．

そもそもマスク ROM は，少なくとも何万台とか何十万台，何百万台というような単位で，それなりの量で発注, 製造してこそコストが安いのであって，それをいちいち交換する都度，製造することをやっていたらコストメリットが出ません．

そのため, ROM コレクションなどという方法で，プログラムの一部を書き換える④ための仕組みが必要になります．この機能の実現のためには改修プログラムを格納するための EEPROM ⑤ や FlashROM⑥などの書き換え可能な ROM がマスク ROM とは別に必要で，RAM の量やアプリケーションでの対応，CPU での対応機能が必要です．

ハードウェア的な要件，つまり，部品コストなどの要件として，ある程度の条件は製品企画部署やハードウェア設計部署から指定されているとは思いますが, それが本当に可能かどうかの検討も必要です．

これは回路設計，基板設計の前に確定する必要があるため，慎重に検討する必要があります．

開発の最終段階で，「実は, プログラムが多くなって ROM に収まりませんでした」，とか，「データ処理量が多くて RAM が足りなくなりました」, と泣きついてもダメです．これはエンタープライズ系では全く起きない問題です．もし，このような状況になったときに本当に対応すれば，膨大な費用の損失を誰かが負わなければなりません．

したがって，ここではオブジェクト指向モデリングなどを行う余地はほとんどありません．抽象化して部品の独立性を確保してソフトウェアの再利用性に留意して実装しても ROM や RAM が足りなくなるのではこのような開発では本末転倒だからです．

何より，この工程の仕様書の成果物を検証（レビュー）するのはソフト屋さんではなく，ハード屋さんです．また，ソフト屋さんがハードの構成，チップの特性，製品としての機能を理解しているかどうかもレビューされます．

念のために繰り返しますが，ここで話題にしている「組込みソフト」は「狭義の組込みソフト」⑦です．

設計手法やモデリングなど，レビューされる人の問題ではなく，レビューする人の問題でその手法は限定されます．ソフト屋さんはモデリングを議論する必要性を感じているかもしれませんが，ハード屋さんには多くの場合は理解されません．一般のほとんどのソフト屋さんが基本的には回路図が分からないのと同じことだと思います．

なお，経験の少ない組込みソフトウェアエンジニアを有効活用するためには，この工程の設計を行うのはハード屋さんであっても良いと思います．

つまり，回路図やチップのデータシートなどはソフト屋さんには見せずに，この段階でソフト屋さんに歩み寄った仕様書を提示するようにする，という

---

④ 「パッチを当てる」と言われます．
⑤ 電気的に消去，書き換えができる ROM．1 バイト単位に書き換えができる．書き換えできる回数に制限がある．ハード屋さんはいーすけあぴーろむと発音する場合が多い．個人的にはいーいーぴーろむの方が好きです．
⑥ Flash メモリとも言う．EEPROM の一種だが複数バイトのブロック単位にしか書き換えが出来ない．
⑦ 広義の組込みソフトの場合，仮想記憶システムを前提とした OS を使っても構わないのですから，最初からマスク ROM にシステムおよびアプリケーションプログラムを格納することなどありえません．

ことです．そして，この場合はレビューという形ではなく，ソフト屋さんに対しての指示を兼ねた「読み合わせ」のための資料として位置付けます．仕様書のレビューはハード屋さん同士で行い，ソフト屋さんはせいぜい，オブザーバとしてレビューに参加することになるでしょう．

ハードウェアブロック図は回路図を簡略化した形でハード屋さん，ソフト屋さんの双方のコミュニケーションを助ける仕様書の一つで，この工程で描くか，基本設計工程で描くかのどちらかでしょう．

3.2. 基本設計

基本設計ではハードウェア構成に対応したソフトウェア構成，タスク構成[8]などを検討します．
明確な機能を持ち，単独のチップとして CPU に接続される周辺ハードウェアがある場合は，そのチップに応じたタスク，ドライバなどを独立させるべきでしょう．

また，特定の通信インターフェースに依存する部分もドライバとして独立させます．それぞれの通信インターフェースに依存するような割り込み処理の検討もこの工程である程度行います．

但し，既に開発済みの製品群のバージョンアップ製品，コストダウン製品，豪華バージョンなどの派生製品の開発の場合，新規のソフトウェア構成を考えることは難しくなります．

この場合，「前モデル」を開発のベースモデルとして「改造」もしくは「派生」を行うことが開発作業の全てとなります．
したがって，基本設計とは言っても新規の設計は行わず，ベースモデルのソースプログラムのどこがそのまま使えてどこを変えるのか，という視点での検討作業がほとんどの設計作業を占める事になります．

このような派生開発の場合，仮にソフトウェア構造上，エレガントで抽象化されて再利用性が高い方法で設計して，その方が処理速度やメモリ使用量の点でもメリットが高かったとしても，ベースモデルとの変更点が多ければその設計は却下される可能性が高くなります．また，ベースモデルのバグの改修も制限されます．

バグの改修すら制限されるのは，このような製品開発の場合，「変わる」ということが最上位の「否承認事項」であり，前モデルの「結果的に正常に動いていた」という実績の方が優先されるからです．

本特集では，新規であっても派生開発であってもどちらでも共通する設計成果物について検討します．この場合もレビューする人は開発担当のソフト屋さんというよりもメーカのプロジェクトマネージャということになります．ソフトウェアの設計手法云々よりも，そのマネージャが納得する仕様書の提示が求められることになるでしょう．

したがって，ここでも UML などという表記方法ではなく，フローチャートやデータフローダイヤグラム，ブロック図というような古典的な図を用いることになります．タイミングチャートや状態遷移図，状態遷移表などが使われることもあります．設計書の記述ツールは Word，Excel，パワーポイントなどを利用します．それらを使って，図を描くことが多いと思います．描くのが楽だからといって UML 用のツールでステートチャート図（UML2.0 からはステートマシン図）などを書いたら書き直しを指示されることになるかもしれません．

フローチャートは，下流の機能設計やプログラム設計でも使われます．それぞれ，詳細度が異なるだけです．

また，チップなどのデータシートに関連して，要件定義で概要が記述されているだけの場合は，基本

[8] リアルタイム OS などを使わない場合でも独立した機能を持つ処理群をタスクとして分類することは自然です．

設計でソフトウェアとしての実装について明確にします．特に，アプリケーションと独立したドライバとしての実装を行う場合は，割り込み処理時間，および，タイミングチャートから読み取った制約時間などのリアルタイム制御としての条件を洗い出して，整理します．

一方，ハード屋さんが回路図を見れば分かるような情報，プルアップ，プルダウンなどによるポートの状態はソフト屋さんに分かるようにポートの一覧表という形でまとめ直した方が良いと思います．

3.3. 機能設計

基本設計で定義したタスク構成のタスク毎の機能，タスク間のインターフェース，データ処理について記述します．但し，派生開発の場合，かつ，前モデルで仕様書がある程度用意されている場合はこの部分での仕様書の手直しの必要性はほとんどないでしょう．

この段階での仕様書をレビューするのはソフトウェアエンジニア同士です．
別の表現では，基本設計までを「外部設計」，機能設計以降を「内部設計」と呼んでも構いません．

繰り返しますが，仕様書というのは誰がレビューを行って，誰がそれを承認するかによって，表現や表記方法が変わります．いくら UML などが優れていたとしてもその点を勘違いするべきではありません．もちろん，自分で好きに設計して，自分で承認できるような状況に置かれているのであれば何でもありですが，そのような状況は特殊ケースとして取り上げません．

逆に UML などの表記法が組織的に求められるのであればそれに従えば良いだけの話です．

本特集では設計としては上位に位置し，内容的に抽象度が高い，「外部設計」について解説していきます．

但し，古典的な図表を使うからといってオブジェクト指向設計が出来ないわけではありません．タスク間のインターフェースの抽象度や結合度などの検討などはこの工程で行うことになります．

以降，詳細設計，プログラム設計，製造，テストと開発工程は下流に下りていきますが，内容が詳細になればなるほど，環境依存度が高くなって行きますので，抽象的に論じることが難しくなるため，本特集では割愛します．

4. 立場によって異なる設計

組込みソフトの分類および設計工程について整理をしてきましたが，組込みソフト開発を行うエンジニアが活躍する業界について，もう少し考察してみましょう．

図 1-4 に製品開発の大まかな関連組織を示しました.

ソフトが組込まれた製品を開発する場合，その大元は何らかのメーカが想定されます．そのような全体的な枠組みの中で，ソフトウェアを設計する担当部署を整理すると以下のようになります.

(1) メーカのハードウェア設計・開発部署
(2) メーカのソフトウェア設計・開発部署
(3) メーカのプロジェクトマネージメント部署
(4) OS ベンダー
(5) ミドルウェアベンダー
(6) 半導体商社・チップメーカ
(7) 受託ソフトウェア会社，派遣会社

以下，それぞれについて述べます.

4.1.メーカのハードウェア設計・開発部署

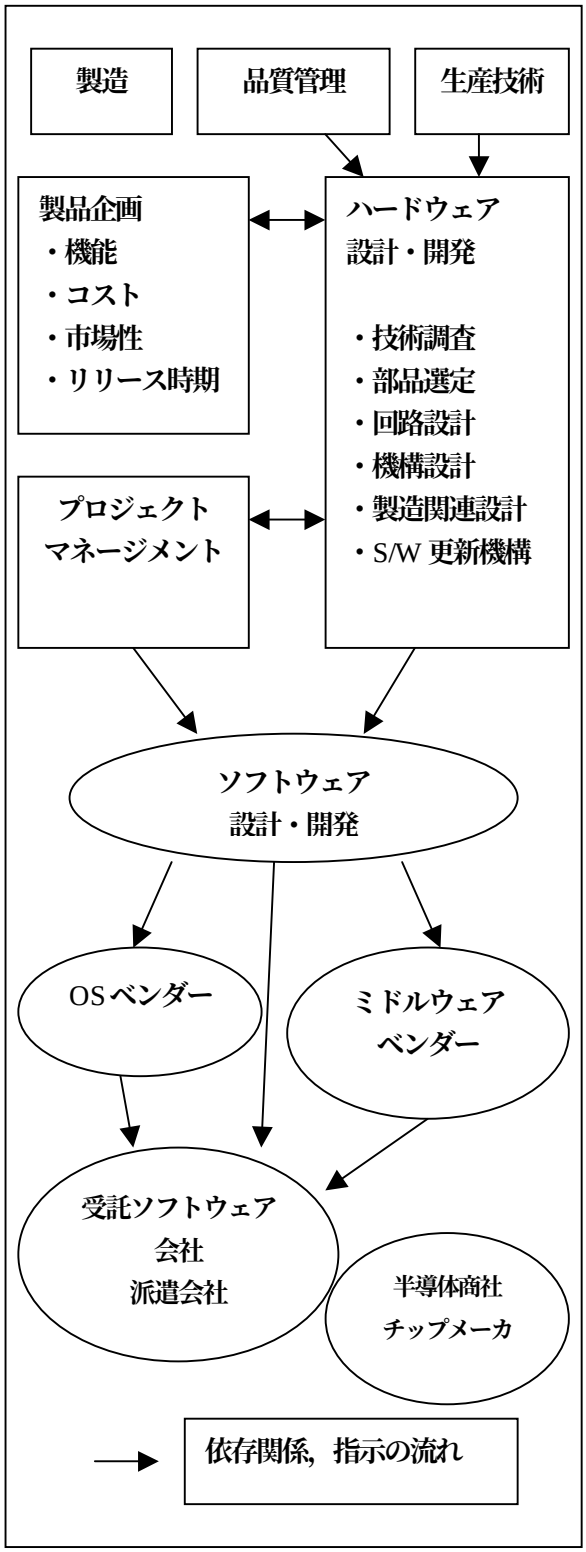
メーカの中には特にソフトウェア専門の開発部署が存在しない場合があります．小規模なメーカの場合は特にその可能性が高くなります．そのような場合も含めると，ソフトウェアの実装をハードウェア設計・開発部署で行うことも少なくありません.

このような組織では開発が自社内で閉じている場合は良いのですが，手が足りなくなって外注に仕事を発注するようなケースに発展すると，問題が顕著化します．それは，「仕様書が全くない」という問題です.

ハード屋さんとしては，ソフトウェアの仕様書を作成する時間がなくてそうなっているというよりも，書く必要を感じていない，または，書き方が分からないということが大きな理由になると思います.
書く必要性を感じない大きな理由はハードウェアの局所的なコントロールだけを行って全体のパフォーマンスの設計，つまり，リアルタイム制御システムとしての設計を行っておらず，問題点を認識していないことがあるからだと思います.

また，ハード屋さんは回路図やチップのデータシートがあれば，ある程度の単純なソフトウェアであ

図 1-4　製品開発に関連する組織



れば「作成」することができます．そしてそのスタイルで同じようにソフト屋さんも開発ができると思い込んでいます．しかし，ここに大きな問題が潜んでいると思います．　つまり，ソフト屋さんは基本的には「文系」出身者が多いと思った方がいいからです．

生産技術部署や品質管理部署があるのにソフトウェアの仕様書が存在しないというような状況があるのか，と思われるでしょうが，それは彼らにとっては何の問題にもなりません．回路図やデータシートなどの「立派な設計書」は揃っているからです．また，「品質」という言葉もソフトウェアに対するものではなく，出来上がった製品に対するものなので，ソフトウェアの開発体制まで影響は及びません．

このように，文化が異なりますので，コミュニケーションにはそれ相応の困難が伴います．

文化の違いが明らかな場合はいろいろと考えて対応を行うことが必要です．

その一つの方法は徹底的に「責任分界点」を明確にした設計書を先手必勝で突きつけることです．そしてハードウェアについては何も分からないということを明言する．その代わり，ソフトウェアエンジニアに対する仕様提示の仕方はレクチャーする．というやり方です．文化の違いがあることを前提に話を進める場合とそれに気付かないで自分の論理で話を進めるのとでは，結果は大きく異なって来るはずです．

なお，同じメーカでも年間 100 万台作るメーカとオーダーメイドで年間 100 台作るメーカでは開発体制は全く異なることでしょう．コストを安くするという意識の持ち方も恐らく大きく違いますし，品質管理の仕方も異なることでしょう．これにより，自ずと，開発のどこが重視されるのかも変わってきます．

4.2.メーカのソフトウェア設計・開発部署

この組織の場合は，少々分類の幅が広くなります．ハードウェア設計・開発部署と大差ないことも多々ありますし，徹底的に先進的な組織の場合もあります．組織として有効に働いている場合はハードウェア設計・開発部門との仲介役になってくれます．

風土によっては適切な設計手法を取り入れようとしても，従来のやり方で問題が出ていないと判断されると，その提案は却下される場合もあります．逆に新しいやり方を積極的に取り入れることができるような風土であれば，適切な設計手法，実装方法を採用することができるでしょう．

ソフトウェアの実装に関しては，製品モデルごとのソフトウェア継承がコピーして派生させるのか，コンポーネント毎に常に組み立てる形式なのかも組織によって性格が異なります．

いずれにしても，新しい技術などを採用できる可能性のある唯一の組織といえます．

4.3.メーカのプロジェクトマネージメント部署

ハードウェア開発，ソフトウェア開発の実務経験がないような管理専門部署の場合，細かい技術的な問題点が黙殺される可能性があります．
ハードウェア開発出身のマネージャはソフトウェアのことが分からずに全体の調整を行うようなことになり，最終的なソフトウェアの性能が破綻することがあります．

但し，ハードウェア知識が豊富な組込みソフトエンジニアがマネージャの場合は非常に適切な管理ができる可能性もあります．回路設計などの詳細な専門分野はハード屋さんに任せてソフトウェア開発部署に対する指示も的確に出来ます．ハードウェア開発出身のマネージャは狭義の組込み技術は分かっても広義の組込み技術の習得は現実的には困難だと思われるからです．組込み開発の大規模化問題はこのような構図で発生するのではないでしょうか．

ハードウェア設計・開発部署がマネージメントする場合と同じように見えるかもしれませんが，直接ハードウェア設計･開発部署が担当する場合は，「マネージメントを行っている意識」がありませんから，結果は全く違います．ソフトウェア開発者との意思の疎通は時間がかかったとしても自然にできるようになると思います．

それに対して，プロジェクトマネージメント部署が独立し，かつ，マネージャがハードウェア設計出身の場合，ソフトウェア設計開発手法が過去の資産に引きずられて，新しい手法の取り込みが遅れてしまいがちです．ソフトウェアの部品化というよりも過去の同等製品のソフトウェアのソースコード一式をコピーしてきてそれを流用して新しい製品の開発を行うようなことが行われます．

過去の製品のソースコードに仮に問題があったとしても，結果的に問題なく動作しているのであれば，そのソースコードを手直しすることは基本的に許されません．このような組織ではレビューを行うなどの体制的な整備は徹底していますので，ソースコードの手直しを行うためにはマネージャの承認が必要だからです．そして，ほとんど承認されることはないようです．なぜなら，「適切な設計」よりも「前と同じ」ということが優先される場合すらあるからです．このような組織の場合，バグもタイミングを取る一つの要素だと信じられている場合があります．また，手直ししたソースコードに関しては全てのテストを行う必要があるため，納期的，開発メンバー的な組織的な問題に発展する場合もありますので，なかなか，手直しは行われないという側面もあります．

4.4. OS ベンダー

OS ベンダーとしてメーカの製品開発に関わる場合は，全体のソフトウェアの実装を担当するというよりも，メーカの開発したハードウェアへの自社 OS の移植という仕事が多いでしょう．ハードウェア設計の段階から，コンサルティングを行うこともあるでしょう．また，単純に標準的なパッケージを用意してメーカに対しては提供するだけという立場もあります．

まれに，OS を採用する前提で製品全体の設計の依頼が来ることもあるとは思いますが，このような場合はそのメーカにはソフトウェア設計・開発部署が存在しない可能性が高いので，直接的にハードウェア設計･開発部署との調整を行うことになります．

なお，当然のことながら，OS ベンダーの自社製品や同等種類の OS が不利になる説明は行われないため，技術領域が偏る可能性があります．自社の OS を第三者的視点で俯瞰した上で，更にシステム全体の開発を行う経験も比較的に少ないため，設計が適切ではない場合もあります．

4.5. ミドルウェアベンダー

パートナーとなる OS ベンダーがいる場合が多いので，商流としての依存関係があります．OS に対する知識，技術力があるため，製品全体の設計，マネージメントをメーカから依頼されることもあります．しかし，基本的には製品をまとめ上げるということは得意分野ではないと思います．

そのベンダーが熟知した OS のドライバの開発，OS のポーティングなどを汎用的に行うことは得意なのですが，特殊な環境に依存しているエンドユーザのシステムの内部での振る舞いなどを詳細に把握して，そのメーカの特注品としての API やドライバなどを開発することへの対応は契約上からも難しくなるようです．

これはベンダーに問題があるのではなく，発注側であるメーカにそれを責任を持って管理する必要があるにも関わらず，多くの場合，丸投げにしてしまって，管理者が不在になってしまうことがある，ということに問題があるのだと思います．

4.6. 半導体商社やチップメーカ

CPU や周辺コントローラなどを販売する半導体商社，または，CPU や周辺コントローラのチップ

メーカが組込みソフトの設計・開発を行う業務を担当する場合もあります.
この場合，その商社やチップメーカが扱う半導体の種類によって担当する分野が多岐に渡ります.
開発製品に商社が扱う CPU が採用されているのであれば周辺のコントローラが別の商社が扱う製品であったとしても全体の設計を行う可能性があります．周辺のコントローラだけを扱う商社の場合はそのコントローラのドライバやミドルウェアだけの担当を行うことになります.

4.7. 人材派遣

ソフトウェア受託会社や人材派遣会社などからメーカに常駐して設計・開発を行うという開発者人口が実は一番大きいのではないでしょうか. この場合，結果的にはメーカのプロジェクトマネージメント部署と外部のエンジニアで開発を推進する，ということになります.

さて, 人材派遣されてメーカに常駐して開発の仕事を行う場合，最初のうちは「テスター」という職域の仕事をすることが多いようです.

「開発」という仕事のイメージからは程遠い位置付けの仕事ではあるのですが，一番広範囲に製品の機能に関わる仕事でもあります.「開発」の場合は，役割分担により，自分の開発範囲はある程度限定されるわけですが，テストを行う対象は全てを網羅する必要があり，ある程度無作為に項目だけで作業分担される可能性は高いでしょう．しかし，このような立場の場合，逆に，機能的には広範囲に製品を知ることになり，業務知識を得るという立場では一番良いポジションであるとも言えます.
この工程の経験は後に設計を行うことになる場合でも必ず役に立ちます．単にテスト項目をつぶしていくという作業に追われているだけでなく，なるべく，その項目の背景や意味を理解していくことにより自然と設計の力がついてくることになると思います．テスト仕様書に設計上の概念説明がないのであれば，自ら調査してその項目を追加してテスト仕様書としての完成度を高めるということを試みるのは悪くないことです.

5. 本特集で最終的に作成される設計書のイメージ

本特集では，最終的に，以下の設計書を扱います.「見せたい人」というのはレビューの承認者である場合もありますし，実装担当者である場合もあります．要するに誰に見てもらいたくて書く（描く）のか，ということでの分類です．本特集では成果物そのものとしての仕様書を作成するということではなく，それらを構成するための情報を整理します.
表 1-1 に設計に関する成果物と概要を示します.

表 1-1 本特集で扱う設計書類

成果物名	概要	見せたい人
ハードウェアブロック図	回路図を簡略化したもの．特に周辺に接続されるハードウェアと CPU の関係を明確にする.	ハード屋さん
ポート定義表 端子一覧表	回路図を読まなくても分かるように I/O などのポートの状態を表にしたもの．プルアップ，プルダウンなどの用語は使わず，初期値，デフォルト値などの表現でまとめる.	ハード屋さんと実装担当者
ソフトウェアブロック図	タスクやドライバの関連を図で表したもの	プロジェクトマネージャと実装担当者
状態遷移図	全体のシステムの状態の遷移，個別のタスクの状態の遷移を表したもの	プロジェクトマネージャと実装担当者
コントローラインターフェース仕様書	ハードウェアを制御するコントローラのデータシートをソフトウェア開発の視点で翻訳したもの．時間制約を把握するためのタイミングチャートも含まれる.	ハード屋さんと実装担当者

ハードウェアブロック図の作成はハード屋さんとの意識合わせに作成し，全体の概要をソフト屋さんも認識することに寄与します．但し，極端な文系の

エンジニアはブロック図程度でも回路図と同様に拒否反応を示す場合がありますので，余り細かい記述を行うことは逆効果になります．次の段階ではハードウェア構成に対応する形のソフトウェアブロック図の作成を行います．

　また，ハードウェアのデータシートに記述してあることをソフトウェアの仕様書としてどのように反映させるかを記述した，タイミングチャートについても設計書としてまとめます．
　すなわち，工程としては，要件定義と基本設計部分に注力します．この段階が終われば実装上はほとんどエンタープライズ系と同じに話を進めることが出来ます．いえ，むしろ，エンタープライズ系の開発よりもシンプルに開発を行うことができると思います．

　モデリングの表記方法としての UML などはハード屋さんが望まない限り使えません．

6. おわりに

　本章では，組込みソフトの分類と業界について整理しました．次章以降では，身近な家電品である，電子レンジを例に取り，実際に設計を行ってみることにしましょう．

　ちなみに筆者は電子レンジの開発は行ったことがありませんので，細かい部分は想像で設計を行います．
　具体的に誰でもが知っている機器を設計するということを通して，必要最低限の組込みソフトの設計ができるようになることを目指します．電気の知識も不要です．ですが，パルス，電源，GND 程度の言葉と略号は出てきます．
　なお，これまでの説明でソフト屋さんが書いた設計書の読み手がエンタープライズ系とは異なり，ソフト屋さん以外であることが多いということが分かってきたと思います．その意味でソフトウェアエンジニアリング的に新しい手法などは取り入れにくいということも理解できているものとして，話を進めていきたいと思います．

■■■■■■■■■■■■■■■■■■■■■■
■第 2 章　ブロック図を描いてみよう
■■■■■■■■■■■■■■■■■■■■■■

1. はじめに～ブロック図とは

みなさんは，打ち合わせなどを行うときに，ホワイトボードを使うことがあると思います.

ブロック図は簡単に言えば，ホワイトボードで描ける程度の四角や丸で囲んだものを並べてそれらの間を線や矢印で接続したものです.

設計に必要なのは実はこの程度の道具だけです.

ホワイトボードでは UML の表記方法をはじめとする，標準化された表記ツールを使って図を描くことはほとんどないと思います．手書きだと標準化された表記方法を守ることは非常に面倒だからです．描き手はそれぞれ思い思いの表現で図を描きながら，言葉で表現を補足することになると思います．それを誰もとがめたりはしないと思います.

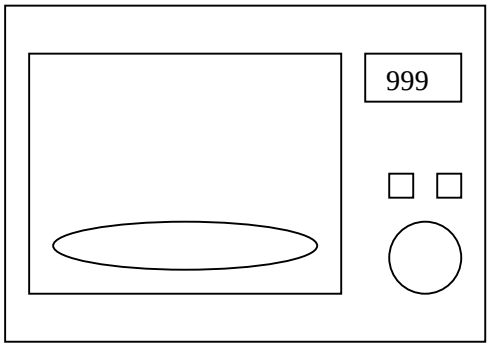
ホワイトボードは面積が限られているため，それほど多くの内容を網羅することはできませんが，それでもそれぞれのテーマの中で，一目で全体像を描くことを自然に行うことになると思います．このようなまとめ方をすることが設計の第一歩です.

仕様書としてのブロック図も同じようなことで，必ず，１枚でそれぞれのテーマでの全体像を描くべきものです．表記上の約束事も特に標準化されている必要はなく，見せたい人が分かる表現，組織毎の約束事を守った表現を使う, ということで十分です.

したがって，これから紹介する「ブロック図」も筆者の勝手な約束事で書いているために，標準的なものだというつもりで見る必要はありません.

さて，ここでは，電子レンジのハードウェアブロック図を描いてみようと思います．電子レンジを選択したのは，現在の日本の家庭にはほとんどあると思われるからです.

それでは，ブロック図を描くのに先立って，簡単にどんな電子レンジかを定義しましょう.



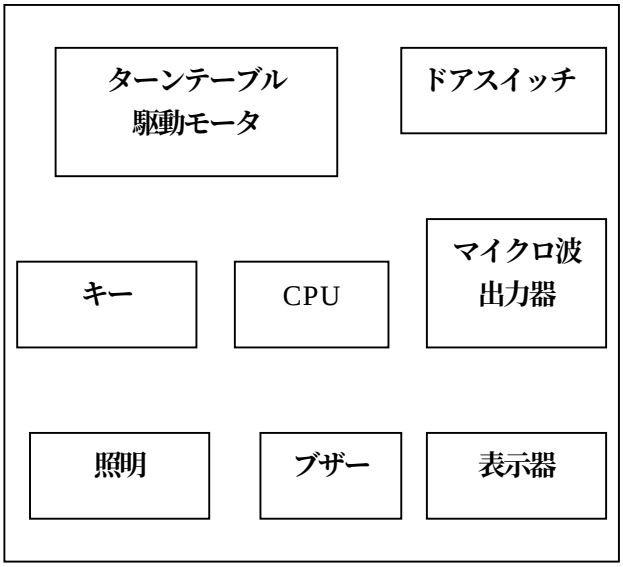
(1) あたためキーと分秒設定キーだけの入力操作
(2) キーを押すたびにキータッチ音が鳴る
(3) 表示器で分秒設定とあたための際の残り時間表示を行う
(4) あたためるものは庫内のターンテーブルに乗せる
(5) ターンテーブルはあたため操作の度に前回と逆方向に回転する
(6) 扉を開けると庫内に明かりが点る
(7) あたため中も庫内に明かりが点る
(8) あたため終わるとブザーで終了を知らせる
(9) あたため中に扉を開けると警告音を発してマイクロ波出力を直ちに停止させる
(10) 扉が開いたままだとあたためることはできない

この時点では，まだ，どんなハードウェア構成かなどは不明です．ハードウェアは抽象的でわからなくても，とりあえず描いてみましょう.

2. 機能を並べただけのブロック図

箇条書きの要件で登場した機能（キーワード）を並べただけのブロック図は図 2-1 のようになります.

図 2-1 機能を並べただけのブロック図



このブロック図では CPU とのインターフェースが全く不明です．CPU とそれ以外のブロックがどのように関わるのかも全く不明です．単純に何があるのかがわかるだけです．したがって，ソフトウェア実装に関する検討も開始できる状況ではありませんが，「何があるか」はわかるブロック図になります．

なお，このブロック図では機能とブロックの名前の変換が以下のように行われています．

機能表現	機能ブロック名
あたためる	マイクロ波出力器
ターンテーブルの回転	ターンテーブル駆動モータ
あたためキー，分秒設定キー	キー
キータッチ音，警告音，終了通知	ブザー
扉を開けると，閉めると	ドアスイッチ
明かり	照明
分秒設定表示と残り時間表示	表示器
～する	CPU

このように，ただ，機能を並べるだけとは言っても，適当なデバイスなどを想定して名前を付け替えることは行わなければなりません．

3. データの流れを追加したブロック図

イベントやデータの流れのイメージをつかむために図 2-1 のブロック図に CPU との間の矢印を追加して図 2-2 を描いて見ましょう．本当に，単純に矢印を間に描いただけです．少しだけ入力と出力を分けるためにレイアウトは変更してみました．

図 2-2 データの流れを追加したブロック図

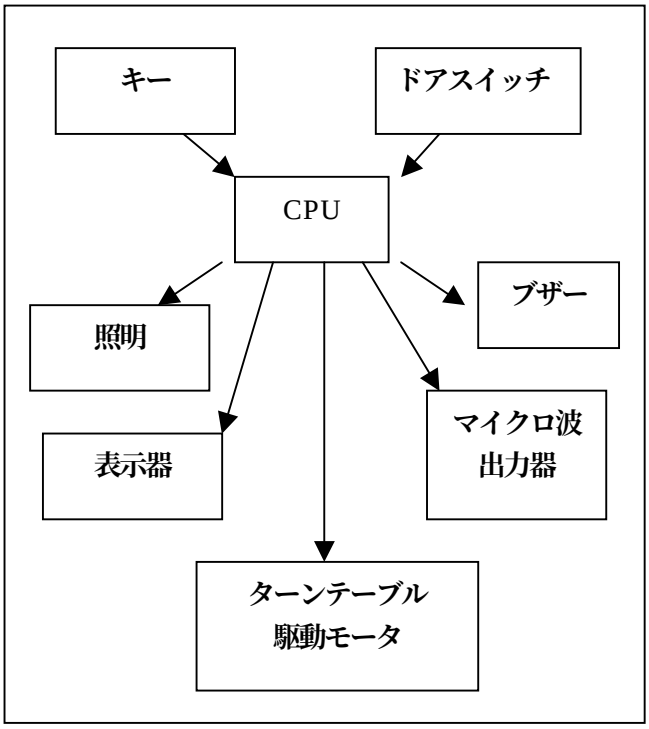


図 2-2 のブロック図でもまだ，どのようにソフトウェアを実装してよいかはわかりません．何が足りないのでしょうか？

相変わらず，矢印の実装上の意味が解りませんし，インターフェースがどのようなものか不明だからです．

また，あたためる時間設定と表示に関する情報など，

欠落している情報もまだまだあります.

さて，ここで，ハード屋さんから使用する CPU の条件と周辺コントローラの条件が開示され，以下の要件が追加されます.

(1) ワンチップマイコン(GIHYO80)を使用する. GIHYO80 は CPU，FlashROM，RAM，クロックタイマカウンタ，6 本の I/O ポートから構成されている.
(2) I/O ポートは入出力のどちらの機能かをプログラムにより個々に設定可能
(3) GIHYO80 に特別な通信コントローラは内蔵されていない
(4) 割り込み入力は 2 本用意されている
(5) 表示コントローラ，キーボードコントローラ，拡張 I/O コントローラが１チップになった複合コントローラ KP7C001 を GIHYO80 の外に接続する
(6) KP7C001 と GIHYO80 との接続は２線式の半二重の同期シリアルで通信を行う.

CPU に出入りする矢印は図 2-2 では７本ありますが, 明確になったワンチップマイコン, GIHYO80 は非常に貧弱なチップで I/O ポートが 6 本しかありません. 単純に矢印の数にあわせることすらできません.

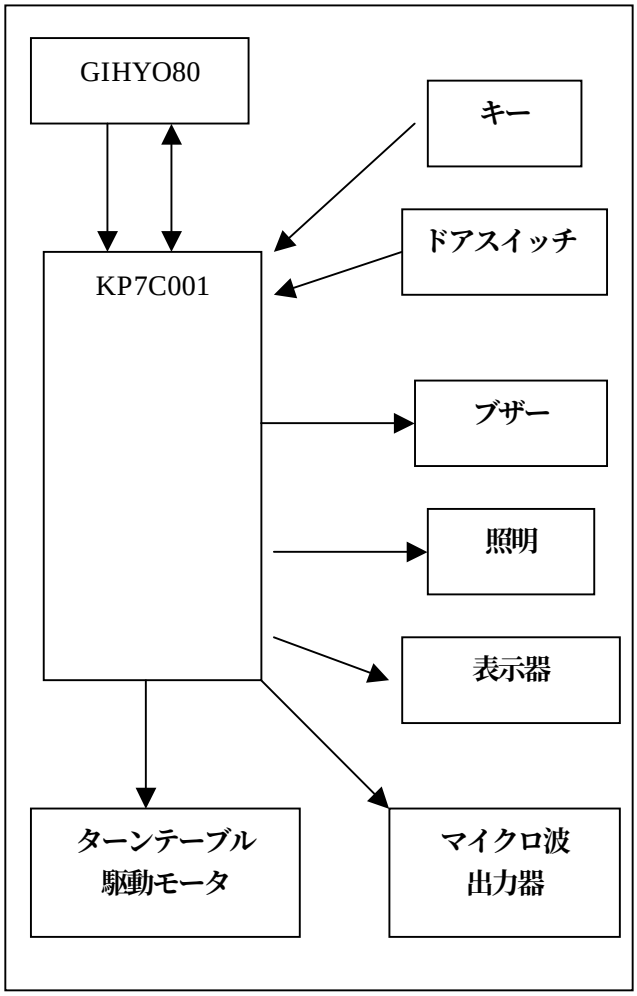
ですが，何やら，ワンチップマイコンとは別に複数の機能を持った複合コントローラチップが外付けで用意されているようです.

4. 周辺コントローラを追加したブロック図

再び，ブロック図を描いて見ましょう．図 2-3 のようになります.

この図でわかる様に複合コントローラチップが使われることにより，ワンチップマイコンが相手にするのはひとつだけになり，それ以外は直接面倒を見なくて良くなったようです.

図 2-3 コントローラと CPU の関連ブロック図



さて，そろそろ，プログラミングできるでしょうか？

残念ながら全くできませんね. この段階で不足している情報をまとめると，概ね以下のようなことになります.

(1) GIHYO80 の仕様がわからない
(2) GIHYO80 と KP7C001 のインターフェースが良くわからない
(3) KP7C001 の仕様がわからない

これらの情報はソフトウェアエンジニアが用意するものでも，作り出せるものでもありません．それぞれのチップメーカが用意するもので，その仕様や約束に従ってチップ間の接続やプログラミングを行う必要があるわけです．

つまり，この場合の設計に必要なのは，まず，チップの仕様書の読解能力です．チップ間のインターフェースが汎用的なものなのであればその知識も必要になりますし，その仕様書に記述されたハードウェア上の制限などを読み取ってソフトウェアに反映させる力も必要になります．
それらのチップの仕様書は一般的には「データシート」と呼ばれ，仕様書の補助資料として扱われることになります．それらの内容を参照しないことには開発を進めることはできません．

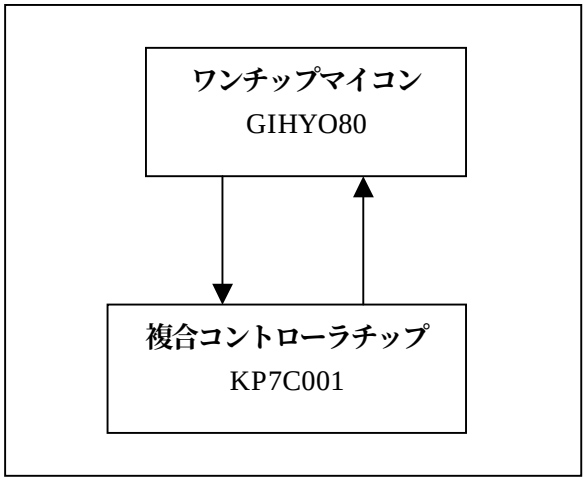
以上のようにワンチップマイコンおよび複合コントローラチップの詳細なデータシートはまだ明らかになっていませんが，ブロック図を描くということについては，ひとまず，ここで区切りましょう．この図で重要なのはワンチップマイコンが相手にするのはひとつの複合コントローラチップだということです．このことがわかるかわからないのでは今後の設計を進める上で大きく異なります．

念のため，コントローラ同士の視点のみのブロック図を描くと図 2-4 のようになります．

5. ハードウェアの情報を読み取る

ブロック図はハードウェアの構成やソフトウェアの構成を抽象的に俯瞰するための道具です．
ただし，抽象化，簡略化したとは言っても，簡略化した部分の共通認識が必要な場合もありますので，少し，横道にそれてハードウェアの情報の読み取り方と不要な情報を見ないコツについて述べてみようと思います．

図 2-4 コントローラ同士の視点のブロック図



5.1. 回路図

回路図はブロック図を詳細にして，製造に必要な部品を全て網羅して接続しただけの図と考えることにしましょう．余り深追いはしないことにします．
電気，電子の知識がないと回路図は全く意味不明の線画の集まりに見えると思います．しかしながら，回路図の詳細をソフト屋さんは見る必要はありません．見なければいけないのはそこに現われているブロック構造です．

実は，ハードウェアブロック図であっても，ハード屋さんがブロック図を描くということはほとんどないと思います．それはなぜかというと，ハード屋さんはブロック図を描く習慣がないと思われるからです．というよりも，恐らく，ブロック図程度は頭の中でイメージしていてわざわざ描く必要性を感じていないのだと思います．

ハード屋さんは，使用部品，コントローラチップ，ワンチップマイコンなどの機能もある程度把握していますので，いきなり回路図を書き始めてしまうのでしょう．もちろん，最初の段階の回路図はブロック図のように単純な構造だったのかもしれませんが，

その簡略化されたブロック図の段階で仕様書として完結させる，ということはしないのだと思います．

例えるならば，ソースコードが全てだと思っているソフト屋さんと同じだということです．ソースコードに書いていることをわざわざ仕様書には書かない，という主義の方はソフトウェアエンジニアでも少なからず存在しているとは思います．ですが，100%ではないと思います．しかし，ハード屋さんは間違いなく限りなく 100%に近い割合で，そのようなスタイルになっていると思います．

一番良いのは，ハード屋さんの情報を鵜呑みにせずに，自分で機能要件からブロック図を展開してみるということです．その上であとから回路図を眺めて，自分が展開したブロック図と比較してみると，それ程ブロック構造に違いがないことに気づくことでしょう．もし，ブロック構造に大きな違いがあればその点だけを重点的に検討すれば問題点や認識のずれの修正を行うことが出来ます．

ハード屋さん，および，組込みソフトの開発の経験を積んだソフト屋さんは，恐らく無意識にブロック図の展開を頭の中で行っています．ですから，いきなり回路図を理解しているわけではないのです．

そのようなことを行わない，ブロック図の展開も行わない段階の初心者の方が，いきなり回路図をみても理解できるはずなどはないのです．そのときに回路図の見方の説明で抵抗，コンデンサ，コイル，ロジック IC，CPU 周辺の知識，などなど，を習得しようとするとその敷居の高さで挫折してしまうのだと思います．

したがって，ソフト屋さんとして回路図を見るのに電気や電子の知識は不要[9]だと開き直ることをお勧めします．実際問題としても，機能ブロックの認識ができれば何の問題もありません．多くの場合，機能ブロックにはそれぞれの機能を担うチップなどが配置されています．それらを接続する間の複雑な回路は見る必要はありません．

但し，一応，最低限の用語だけは覚えておきましょう．用語の認識の必要性は，その用語に出会ったときに，単純に戸惑わない，という目的のもので，理解する必要はありません．用語の置き換えができればソフトウェアの開発上は何の問題もありません．

以下，代表的な用語，略語，用語の置き換えの例です．用語の置き換えは理解度によって自分なりのもので構いません．

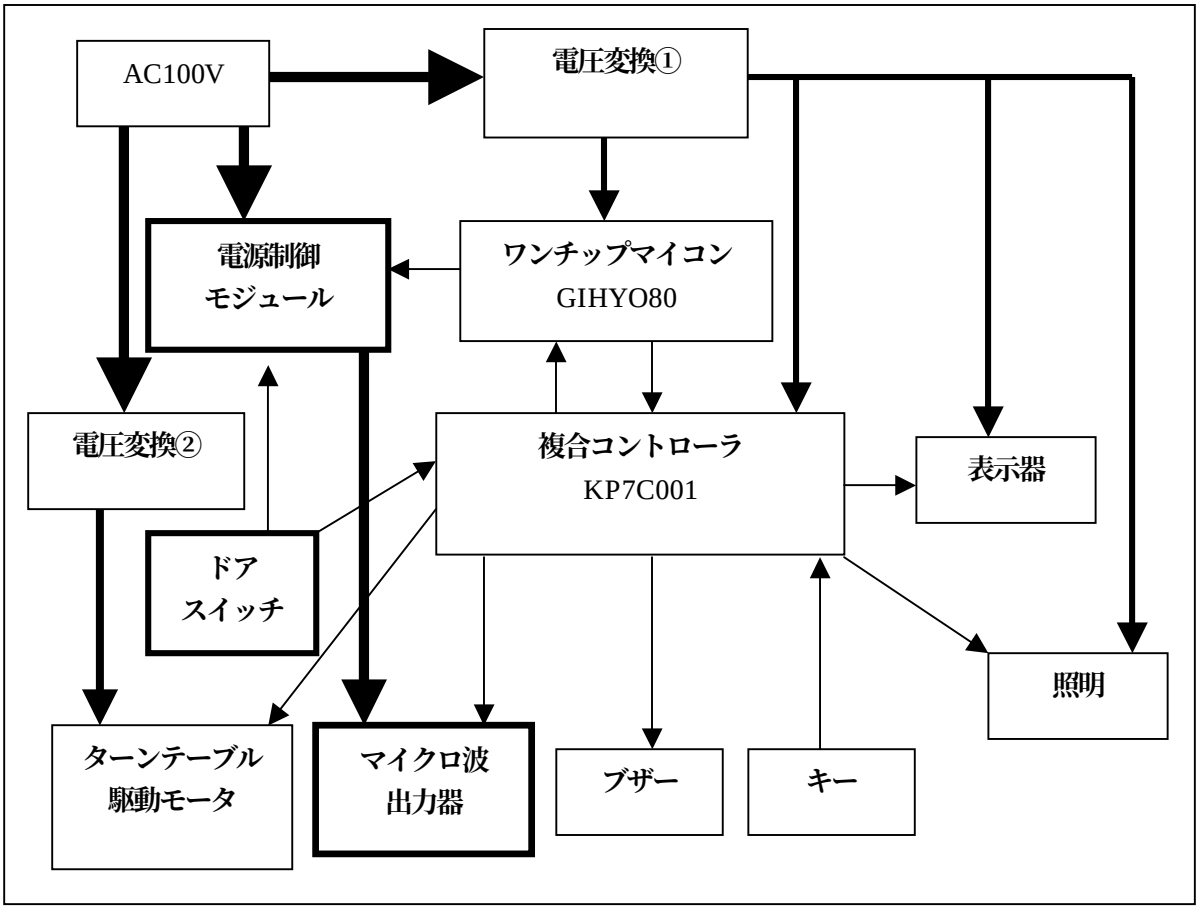
用語	略号	用語の置き換え，説明
電圧	V Vcc Vdd	1 HIGH TRUE 直流しか考えなくてよい
アース グランド	GND DGND AGND	0 LOW FALSE
TTL レベル		HIGH 電圧が 5V
CMOS レベル		HIGH 電圧が 3.3V
電流		ソフト屋さんは考えなくて良い．消費電流がどうこうという話をされたら分からないと宣言する．
インピーダンス	Z	ソフト屋さんは考えなくて良い．直流抵抗の兄弟
入力が未接続		デフォルト値が０か１か分からない．不定．
チップ デバイス		CPU やマイコン，コントローラなどを集積した IC などの総称

5.2. 電源系統図

機能ブロックを図にするときにデータの流れ，情報の流れ，イベントの流れなどを矢印で結んで表現しました．ですが，通常，電源がないと機器は動作しません．ソフトウェアエンジニアの視点では電源は供給されている前提で物事は進みます．エンタープライズ系開発では周辺デバイスや OS に関してもプログラムの起動時点でお膳立てされている，単な

[9] 組込みシステムで扱う，CPU 周辺のデジタル回路に限ります．

図 2-5 電源系統を追加したブロック図



る環境に過ぎません.

ですが，組込みソフト開発では周辺デバイスの電源の制御を CPU で行う場合も少なくありません. そこで, ブロック図に電源に関する情報を追加した, 電源系統に関しての図を描いてみます．それが，図 2-5 です．図 2-4 では複合コントローラ相手の 2 本の線しか出ていませんでしたが，出力が 1 本追加されてしまいました.

かなり，複雑になってきましたが，ここで必要なのは機器やチップには電源が必要だという認識をする必要がある，ということです．条件が追加されたとは言ってもただそれだけです．電気の知識は必要ありませんね．但し，電源といっても電圧や直流, 交流の違いがあるので，そのような部分の変換をする部分があり，電源の制御を CPU で行う場合があるということを示しています.

まず，電子レンジですから，家庭用の AC100V が電源になります. 左上の AC100V というブロックがそれを表しています.

一方でワンチップマイコンなどは直流 5V や 3.3V の電源で動作するものが多いのでそれを表現するための「電圧変換①」ブロックがあります．太い矢印は電圧が高いことを示しているだけです．「電圧制御モジュール」は電圧を変換するとともに, CPU の制御で電源の供給を ON にしたり，OFF にする制御をします．「電圧変換②」も電圧を変換しますが,

CPU からの制御は行われずに，単にコンセントを挿したら変換された電圧が矢印の先に供給されることを意味します．

この新しいブロック図で表された機能を以下に整理します．

(1) 複合コントローラの電源供給は CPU では制御しない
(2) 照明の電源供給は CPU では制御しない
(3) 表示器の電源供給は CPU では制御しない
(4) ターンテーブル駆動モータの電源供給は CPU では制御しない
(5) マイクロ波出力器の電源供給は CPU で制御する
(6) マイクロ波出力器の電源供給はドアスイッチから直接切断できる

以上は最初の電子レンジの機能の定義には現われていなかったことです．こういう仕様は実際の回路図を見れば分かりますが，回路図を見なくても分かるように，ソフトウェア設計としての要求仕様として明確にしておくべき項目です．

照明や表示器の電源電圧はワンチップマイコンなどのチップと同じ電圧とは限りませんし，使わないときには無駄に電力を消費しないように部分的に電源供給を行わないようにするということは暗黙の機能として存在する場合があります．

新しいブロック図での「電源制御モジュール」はマイクロ波出力器の出力はドアが開いているときには必ず停止させる必要があることを重視した制御を行うためのものです．マイクロ波出力がドアが開いた状態で行うことは危険なため，避けなければなりません．機能としては，一番の優先度で処理する処理ではありますが，CPU での制御は不要であることも意味します．このような仕様に関して，製品仕様から先にソフトウェアのモデリングを行ってリアルタイム制御設計を行うと，高優先度処理対象としてソフトウェアで処理する範疇で無駄な設計を行う羽目になります．したがって，ハード屋さんの設計を先に理解する必要があります．

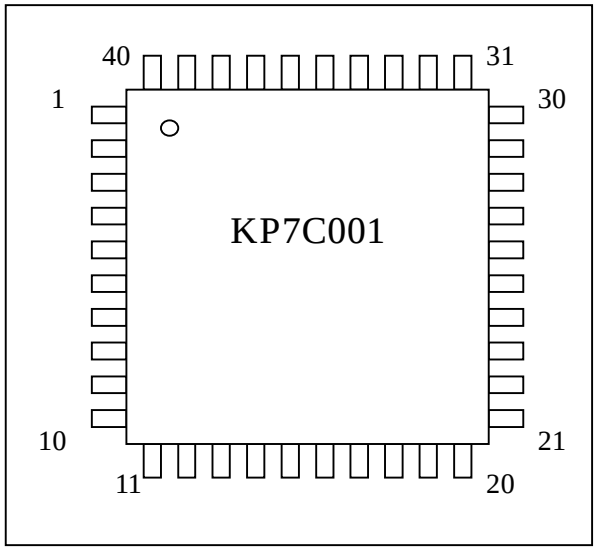
このように，周辺デバイスの電源の ON/OFF の制御が必要な場合は CPU から制御するための出力を必要とします．ハード屋さんから仕様提示がなければ先回りして確認しておく必要があります．

なお，この電源系統を表したブロック図では GND は省略しています．

5.3. デバイスのデータシート

デバイスのデータシートとしてここでは，複合コントローラ KP7C001 のデータシートを採り上げます．完全なオリジナル仕様なので，世の中に実在しませんので，ご注意ください．また，十分な検討は行われていませんので，説明の都合上必要のある部分以外はいい加減な仕様になっています．

図 2-6 複合コントローラ



データシートに書かれている内容は概ね以下のような構成になっています．

(1) 機能概要
(2) パッケージ⑩の説明

⑩ DIP(Dual Inline Package)とか SOP(Small

(3) 電気特性，温度などの動作環境条件
(4) ブロック図
(5) ピンアサイン
(6) 端子機能説明
(7) 通信インターフェース
(8) タイミングチャート

上記で組込みソフトの開発に関係があるのは，(5)~(8)です．

ブロック図はこれまで見てきたのと同様のものですが，チップの内部の構造を簡略化して示すために存在します．要するに，ハード屋さんはブロック図を描かないと述べましたが，デバイスメーカのデータシートでは当たり前のように載っています．.ワンチップマイコンなども同様です．

図 2-6 に複合コントローラのパッケージイメージを示します．パッケージの周辺に左上の○印が 1 番ピンのある方向です．ピン番号は 1 番から 40 番まであります．
また，それぞれのピンの機能は以下の表になります．ピン名の部分で，アンダーバーならぬ，アッパーバーがある部分は有効な論理が負論理であることを表しています．

負論理とは「アクティブ・ロー」という言い方もされます．つまり，電圧が高い HIGH の時に**1**，LOW の時に**0**という論理で，暗黙には「HIGH = 1 = ON」という解釈がされますが，負論理の場合は「LOW=0=ON」という解釈がなされます．プログラミング上は信号の状態と意味がわかってさえいれば論理の違いに余り意味はありませんので見方を覚えておくだけで構いません．

ピン番号	ピン名	機能
1	Vcc	5V 電源
2	CS	チップセレクト入力 HIGH=選択 LOW=非選択 選択時のみコントローラは動作する
3	$\overline{\text{RESET}}$	リセットパルス入力
4	DCLK	データクロック入力
5	DIO	データ入出力
6	DIR	データ入出力方向指示 LOW=入力 HIGH＝出力
7	$\overline{\text{INT}}$	キー入力割り込み出力 キーボード入力時 LOW データ出力で保持しているキーコードを出力すると HIGH になる．キーバッファは 16
9	KSCAN0	キースキャン**0**
10	KSCAN1	キースキャン**1**
11	KSCAN2	キースキャン**2**
12	KSCAN3	キースキャン**3**
13	KIN0	キー入力**0**
14	KIN1	キー入力**1**
15	KIN2	キー入力**2**
16	KIN3	キー入力**3**
17	BZR	ブザー出力 コマンドにより，鳴動周波数，鳴動時間指示可能．キータッチ音自動生成機能付き．
18	GPIO0	汎用データ入出力**0**
19	GPIO1	汎用データ入出力**1**
20	GPIO2	汎用データ入出力**2**
21	GPIO3	汎用データ入出力**3**
22	GPIO4	汎用データ入出力**4**
：		表示器インターフェース
：		
37		
38		表示器インターフェース
39	AGND	アナロググランド
40	GND	デジタルグランド

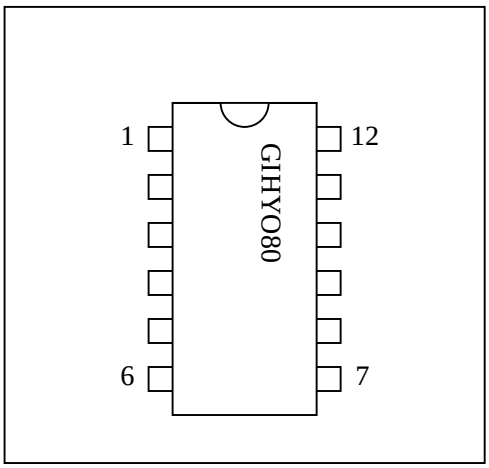
Outline Package)など IC の形状や材質などに依存するもので，同じコントローラでもいくつかのパッケージを型番の枝番で分類している場合があります．

データシートの詳細な検討については章を改めて説明します．

5.4. ワンチップマイコンのデータシート

ワンチップマイコンのデータシートは多くの場合，ハードウェア編とソフトウェア編に分かれています．今回のワンチップマイコンは小規模ですが，動作す

図 2-7 ワンチップマイコン



るためのクロックを発生させる水晶発振器，RAM，FlashROM，入出力ポート，割り込みコントローラなどを内蔵しています．図 2-7 にワンチップマイコンのパッケージイメージを示します．本来は FlashROM の書き換えのための仕組みが必要ですが，説明を簡略化するためにピンの機能説明から省きます．どうやってプログラムを格納するのかは考えないことにします．

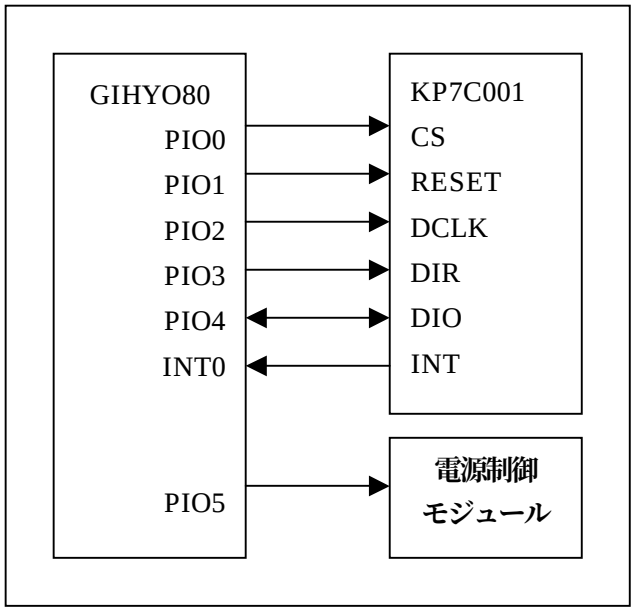
一般的に，ワンチップマイコンのデータシートには以下のような内容が記載されています．

(1) 機能概要
(2) パッケージの説明
(3) 電気特性，温度などの動作環境条件
(4) ブロック図
(5) メモリマップ
(6) レジスタ
(7) 命令
(8) 端子機能説明
(9) タイマカウンタ説明
(10)内蔵 I/O 説明
(11)割り込み制御説明
(12)通信インターフェース(GIHYO80 にはない)
(13)タイミングチャート

以下にワンチップマイコンのピンの機能を示します．

ピン番号	ピン名	機能
1	Vcc	5V 電源
2	$\overline{\text{RESET}}$	リセットパルス入力
3	MODE	内蔵 FlashROM 書き換え選択
4	$\overline{\text{INT0}}$	割り込み入力 0
5	$\overline{\text{INT1}}$	割り込み入力 1
6	PIO0	ポート入出力 0
7	PIO1	ポート入出力 1
8	PIO2	ポート入出力 2
9	PIO3	ポート入出力 3
10	PIO4	ポート入出力 4
11	PIO5	ポート入出力 5
12	GND	グランド

図 2-8 コントローラ間結線図



このワンチップマイコン内の CPU が開発するソフトウェアプログラムを実行するのですから，このデータシートは全体的に確認する必要があります．但し，最初から全てに目を通す必要はなく，その都度不明点をを確認することで少しずつ把握していけば良いでしょう．

さて，ここで，ワンチップマイコンと複合コントローラ間の接続について確定しましょう．
図 2-8 に結線図を示します．

この図が最終形態のソフトウェア開発に関連するブロック図です．ワンチップマイコンと複合コントローラとの間の接続は 6 本になります．ワンチップマイコンからは更に電源制御モジュールへの出力が 1 本あります．INT0 の処理は割り込み処理としての処理になりますので，ポートの状態を直接的には認識する必要はありません．つまり，CPU としては計 6 本のポートと 1 系統の割り込みについて処理することになります．

6. おわりに

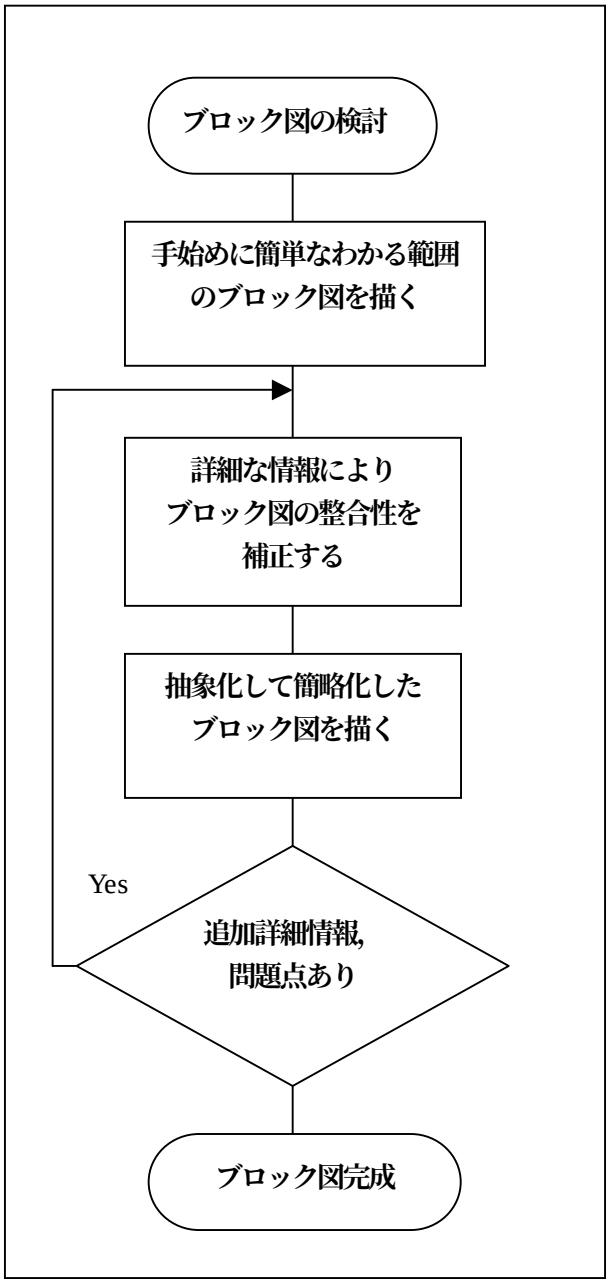
本章では，実際に使用する CPU を含むワンチップマイコンと周辺コントローラとの関わりを示すブロック図を描いてみながら，さらに，コントローラの機能についてのデータシートの概要を見てきました．

狭義の組込みソフト開発の場合にはエンタープライズ系の設計とは異なる，以下の特徴がありました．

(1) 要求定義は機能要件だけでは確定できない
(2) 基本設計は各種データシートの把握がないとできない
(3) 回路図の読解が求められるが，自分で機能展開してブロック図を描くことにより回路図の詳細な理解は不要
(4) 電源制御については機能要件からは把握できない
(5) 必ずしも電気，電子の知識は必要ない
(6) 基本設計段階以前での抽象化モデリングは現実的にはできない

次章では設計で必要になるデータシートの読解について解説します．データシートの読解が終わってから，ソフトウェアのモデリングを含む設計を行う

図 2-9 ブロック図検討フロー



必要がありますので，データシートの検討を行う以前にモデリングを行っても現実的な実装にまでは至りません．設計の過程では，抽象化した簡略的なブロック図とそれより詳細に示したブロック図を検討し，更に，そこで現れた問題点を再度簡略化したブロック図にまとめなおす，ということを繰り返していきます．

データシートの読解も単独のコントローラデバイスとして抽象化してブラックボックスに出来れば良いのですが，多くの場合はそのような抽象化にはいたりません．なぜならば，機能単位で抽象化するために部品が開発されるわけではないからです．

■■■■■■■■■■■■■■■■■■■■■■
■第 3 章　データシートを読んでみよう
■■■■■■■■■■■■■■■■■■■■■■

1. はじめに

ブロック図を描きながらシステムの検討を行った結果，あらかじめチップメーカが用意した「データシート」の読解が設計には必要であることが分かってきました．このようなデータシートは開発担当のエンジニアの手が及ばない部分です．本章では，オリジナルのワンチップマイコンおよび複合コントローラのデータシートを例にとり，開発に必要な情報をデータシートから読み取って検討することにより，ソフトウェアの実装を行えるようにします．

2. ワンチップマイコンのデータシート

今回ブロック図を描くのに勝手に創り出した「GIHYO80」というワンチップマイコンですが，このマイコンのように昨今は CPU 単体だけで一つのチップになっていることは，全くありません．

2.1. ワンチップマイコンの内部の構成

図 3-1 にワンチップマイコンのブロック図を示します．ここで，示したようにワンチップマイコンの内部は更にいくつかの機能もしくはコントローラにより構成されていることがわかります．また，このマイコンのメモリマップは図 3-2 のようになります．

図 3-1 ワンチップマイコンブロック図

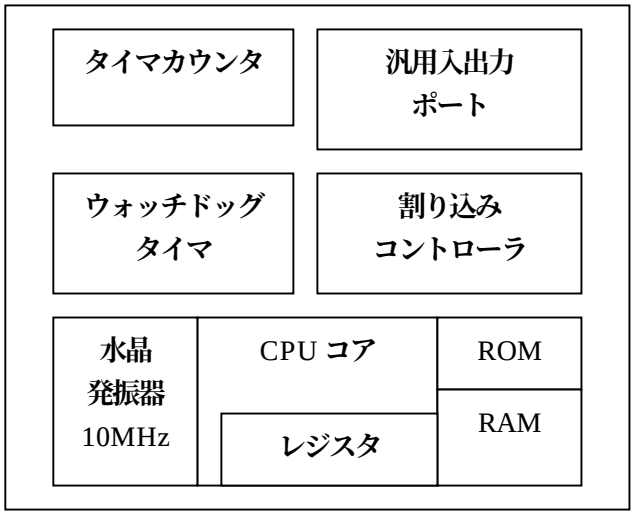
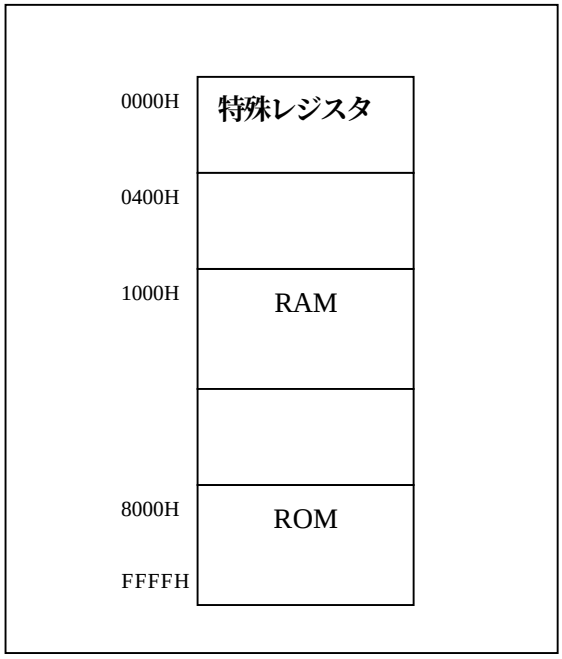


図 3-2 メモリマップ



さらに，特殊レジスタの機能一覧は表 3-1 のようになります．今回のシステムに直接関係のないレジス

タは省略します.

一般的に，分厚い膨大な量のワンチップマイコンのデータシートを隅々まで確認することを行う必要はありません．以下の順序で確認を行うことをお勧めします.

(1) 初期化に関する機能を確認する
(2) 開発対象のブロック図から必要な機能だけを拾い集めて，関連部分だけをみる
(3) 割り込み処理に関する機能を確認する

表 3-1 CPU の特殊レジスタ一覧

ｱﾄﾞﾚｽ	略号	機能
0000H		
0020H	INTCTL	割り込み制御レジスタ bit0：INT0 の割り込みを許可する bit1：INT1 の割り込みを許可する bit2：タイマ割り込みを許可する(カウント開始)
0021H	INTR	割り込み要求レジスタ bit0：INT0 割り込み要求 bit1：INT1 割り込み要求 bit2：タイマ割り込み要求 それぞれ割り込み処理ルーチンを抜けると自動的にクリアされる.
0030H 0031H	TIM	16 ビットタイマカウンタ
0032H	TMODE	タイマカウンタモードレジスタ インターバルタイマまたはワンショットタイマの設定を切り替える
0040H	PIOD	PIO 方向レジスタ ビット毎に PIO0~PIO5 までのポートの入出力方向を設定する. ０：入力／１：出力 リセット後は入力. bit0 が LSB で PIO0 に対応し，bit5 が PIO5 に対応する.
0041H	PIOR	PIO レジスタ 出力時，対応ポートに 0 を書けばポート出力は LOW になり，1 を書けば HIGH になる. レジスタを読んだ値は入力ポートに対応するビットに入力状態が表れ，出力ポートに対応するビットは現在の出力状態を表す.

2.1. 初期化処理

CPU およびコントローラチップなどの初期起動はどのように行われるかがデータシートには書かれています．このとき，2 章で紹介したようなワンチップマイコンのピンと機能の一覧で書かれている「RESET」端子のタイミングの確認が必要です．リセットに限らず，パルス波形については通常は図 3-3 のような表現でデータシートには書かれています．

パルスの表現は CPU から見ると，というよりも，論理的には，図 3-3 のイメージではなく図 3-4 のようなイメージになります．

それにも関わらずに図 3-3 のイメージの表現が使われるのには，「過渡特性」を考慮した図を表現する必要があるからです．

図 3-3 パルス波形タイミングチャートの見方

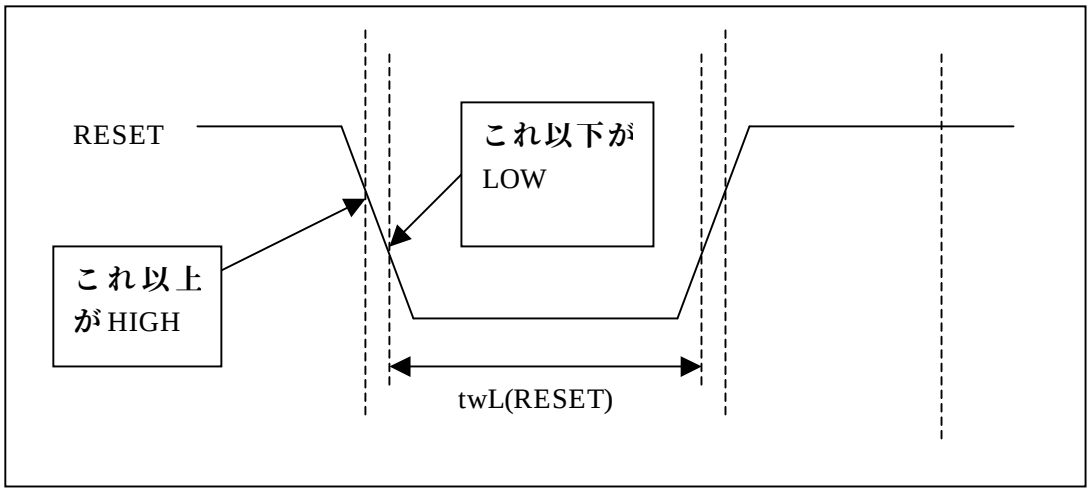
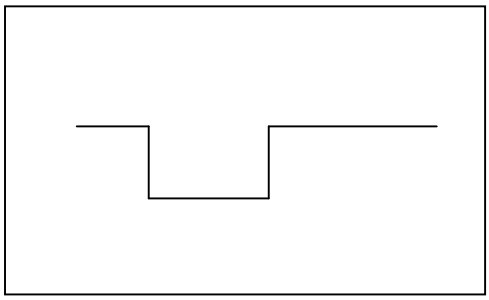


図 3-4 論理上のパルスイメージ



過渡特性とは，実際の電気的な波形は図 3-4 ではなく，図 3-3 のように時間経過によって少しずつ状態が変わることを意味します．つまり，瞬間的に LOW から HIGH に状態が変わったり，HIGH から LOW に変わったりすることはないということです．ですから，その波形の時間条件を表現する場合には図 3-4 ではなく，図 3-3 の表現でパルス波形が表現されることになります．

このようなデータシートでは厳密なタイミング条件を提示して，ハードウェア設計者およびソフトウェア設計者に対して情報提供をすることを目的としています．一般的なエンタープライズ系のソフトウェアエンジニアはこのような表現の波形の意味がわからないため，データシートを敬遠するようなことにもなるのではないでしょうか．

しかし，このような波形の情報にこそ，組込みシステムの設計に必要な時間制約条件が記述されています．

図 3-3 では RESET 端子を LOW にするべき時間の条件が書かれています．この図で書いていませんが，「tWL(RESET)>500us」などのような条件が書かれます．これは，パルス幅は 500 マイクロ秒を超えている必要があるという意味です．

但し, CPU に対するリセットパルスの幅はソフトウェアには影響しません．後述する複合コントローラの RESET は図 2-8 により，CPU で制御することになっておりますので，複合コントローラに対する CPU の制御ではタイミングチャートの時間制約を考慮する必要があります．

2.2. データシートは抜粋して把握

表 3-1 の特殊レジスタ機能一覧を，省略して書いたように，実際の開発でも膨大な情報の中から必要な情報だけを拾い集めて，そのシステムの開発のための仕様書としてまとめることが基本設計のひとつの作業になります．

データシートをそのままソフトウェアエンジニア全員が仕様書の一部として参照するのではなく，開発する上で間違いのない最低限の情報を適切に網羅した物を設計書として整備していく必要があります．

今回の GIHYO80 では必要な機能だけを持つワンチップマイコンに仕立て上げましたが，実際にはもっと多くの機能を持ったマイコンが使われることになります．この場合，全ての人間がそのようなデータシートを参照するのでは，解釈により実装にばらつきが出る可能性が高いため，データシートは必ず抜粋，まとめる，ということを行うべきです．

2.3. 割り込み処理

割り込み処理は CPU とは別の独立した機能を持つプロセッサにある範囲の仕事を丸投げして，そのプロセッサの仕事が終わったら教えてもらうために使う技術です．

CPU では別のプロセッサからの通知をもらうと，あらかじめ登録された割り込み処理（C 言語で言うところの関数があらかじめ関連付けられている）が実行されることにより，通知を受け取ることになります．

割り込み処理は通常 CPU に複数受け付けるための仕組みがあり，それと同時に割り込み処理毎に優先度というものが設けられています．エンタープライズ系の開発では割り込み処理の概念は出てきませんが，イベントが発生したときに処理を行うべき，「コールバック関数」に馴染みがあるのであれば，そのイメージで捉えておけば問題ないでしょう．

さて，ワンチップマイコン，GIHYO80 では割り込みは外部の割り込み処理が 2 系統と内部の割り込み処理 2 系統があります．内部の割り込み処理はタイマ割り込みとウォッチドッグタイマ割り込みです．

今回はウォッチドッグタイマは使わないことにし，割り込み処理としては INT0 割り込みとタイマ割り込みの 2 種類を使うことにします．

割り込み処理ルーチンは ROM 領域の先頭に配置された割り込みベクターテーブルにアドレスが格納されているものとします．

図 2-8 の結線図から分かるとおり，INT0 は複合コントローラの割り込み出力に接続されており，キーボード入力があったときに割り込み通知されます．このイベントをトリガーとして複合コントローラからキーデータを読み込めばどのキーが押されたかが分かります．今回扱うキーは「分」キーと「秒」キー，および，「あたため」キーの 3 種類です．

キーデータとしてどのような値が送られてくるのかは複合コントローラのデータシートおよび回路図を参照しないと不明ですが，たったこれだけのキーデータについてならば，データシートを調べることを行わなくとも，ハード屋さんに確認した方が早いでしょう．

こうして判明したキーコードは単純にシンボルで表しましょう．

したがって，キー割り込み処理の中身は以下のようなサンプルプログラムになります．このようにデータシートが提示されることにより，やっと，プログラミングの目処はついてきます．

```
/* INT0 タイマー割り込み処理 */
void int_key(void)
{
        void req_get_key(void);
        req_get_key();
}
```

割り込み処理中ではキー入力データを取得することをリクエストするだけで実際のキー取り込み処理を割り込み処理内では行いません．

この構造は，リアルタイム OS を使う場合も使わない場合も同じです．割り込み処理内で実際の処理を実装してしまうことも可能ですが，割り込み処理は最低限の処理を行って，細かい，もしくは，重い処理は別のタスクで処理するのが鉄則です．

但し，現実的には今回のようなそれほど複雑でない処理の場合は割り込み処理内で記述されることも多々あります．特に，ハード屋さんがサンプルとして提供するようなプログラムはそのようなものが多くなります．このサンプルをそのまま使ってしまうようなことをしても構いませんが，それは，きちんと検討した上でなら，という条件付です．何も考えずにそのまま流用することは組込みソフトウェアの開発を行ったことにはなりません．

横道にそれますが，これと同じことが，いわゆる，ミドルウェアにも当てはまります．オブジェクト指向設計で実装されたミドルウェアは独立性が高く，再利用が可能かもしれませんが，以下の情報がミドルウェアとして提供されないのであればそのまま使ってはいけません．

・処理内部でのコンテキストスイッチ[11]の有無
・処理内部での処理ループの有無
・リエントラント構造[12]かどうか
・処理時間の目安
・動的なメモリ割り当てのサイズと解放タイミング
・スタック領域のサイズ

私が知る限り，ミドルウェアでこのような情報が開示された例はありませんから，結局はこのようなミドルウェアを利用する場合には以下の検討が必要になります．

・処理時間の実測
・機能要求を満たしているか
・リエントラント構造かどうかの確認
・メモリの要件に問題はないか

これらの確認も設計としては必要になってきますが，実際にはそのような検討が行われることは少ないようです．どちらかと言えば再利用可能な部品を使うことによる開発量の削減に重きが置かれる傾向にあります．

同じ考え方で，組込みソフトではいわゆる標準ライブラリ関数も使用されない傾向にあります．

単純な C のライブラリ関数である，memcpy()や memset()なども利用されないことがあります．これは標準ライブラリ関数をリンクすると他の必要ではない関数のコード部分までリンクされてしまう[13]と

---

[11] リアルタイム OS などでタスクの切り替えが起こるということ
[12] 静的な領域を使っているとリエントラント構造ではない．スタック領域，動的領域などを使っているかどうか．リエントラントな処理は必ずスレッドセーフである．
[13] 実際には適切にリンクすれば無駄なリンクは発

いう弊害も一つの理由ですが，処理の中身が見えない，ということも敬遠される理由です．

さて，キーデータの取得の詳細な処理はこの段階では不明ですが，割り込みが使われるという上位概念だけで，ほとんどの処理の設計は可能です．細かいデータ入出力はドライバが独立して行う，ということを前提とするのです．

3. 複合コントローラのデータシート

次に，ワンチップマイコンと同様に勝手に創り出した KP7C001 という複合コントローラのデータシートを見てみましょう．

図 3-5 にブロック図を示します．

かなり複雑な構成に見えてしまいますが，このブロック図についても実際には全てを理解する必要はありません．

関連する部分のみを参照するようにします．キーデータについてはこのブロック図からはマトリックスキーボードに接続される，と言うことになりますので，実際には回路図を参照しないとキーコードは判明しないことがわかりますが，2 次元配列をイメージすると分かりやすいかもしれません．

	X=0	X=1	X=2	X=3
Y=0				
Y=1		○		
Y=2				
Y=3				

上記の例で○がついているところがキーが押されている部分で丸がついていない他の部分にもキーが配置されているものとします．

Y をキースキャンとし，4 列の X 部分が 1 ビットずつのデータを取り込む部分になります．この配列からキーコードを生成する決まりごとは標準的なものはありませんが，キースキャン番号と X の位置で表現することが比較的多いかもしれません．

例えば，上記の表の場合は，Y=1 で X=1 なので，1 バイトで表現可能で 16 進 C 言語表現で，0x11 と表現することができます．但し，このキーコードの場合は以下のように 2 箇所のキーが押されたときのキーコードは表現できなくなります．

	X=0	X=1	X=2	X=3
Y=0				
Y=1		○		○
Y=2				
Y=3				

2 箇所のキーが同時に押されたことを知るためには X の位置をビットに置き換えれば対応が可能です．

X=1 が一つだけならば，0x12 というキーコードを割り付けて，X=3 一つだけならば，0x18 というコードを割り付けます．両方押されたら 0x1A となります．もしくは，両方同時に押されるということを禁止する場合もあり，これらの制約はハードウェア設計上適切に考慮されている場合がある一方で，後になってソフトウェアで調整する場合も多々あります．

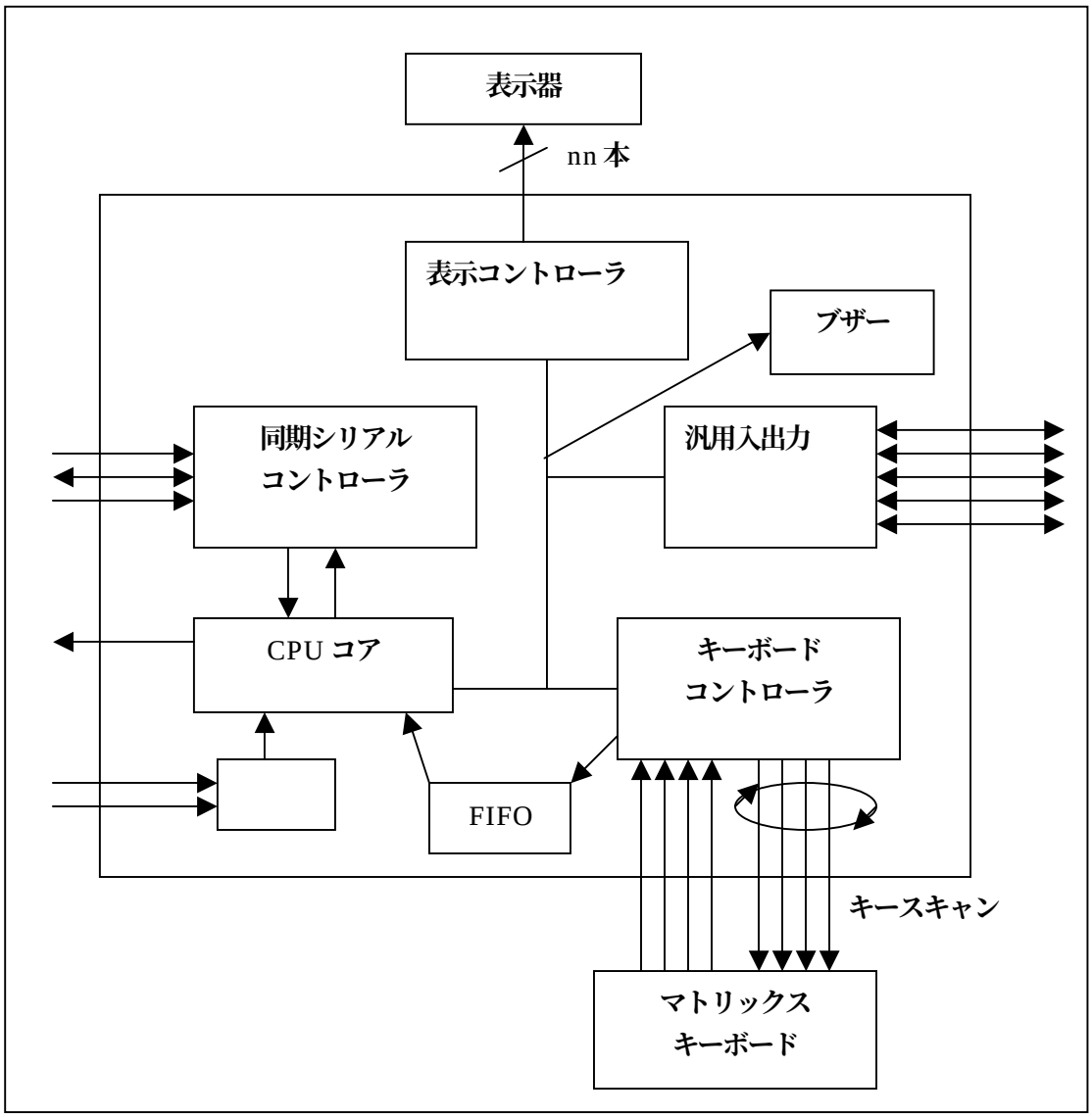
ハード屋さんに確認すれば良いというスタンスを取ったとはいえ，最終的には回路図の確認が必要になるようです．とりあえずは，そのようなことを念頭に置きながら，キーコードについては無視して構いません．回路図としての考え方ではなく，配列としての概念だけ押さえておけば十分です．また，実際には複合コントローラが実際の回路図の結線から，特定の規則でコードを生成するかもしれません．

ここでも，CPU のデータシートと同様にデータシートを見るポイントを整理してみます．

(1) 初期化に関する機能を確認する
(2) 開発対象のブロック図から必要な機能だけを拾い集めて，関連部分だけをみる
(3) CPU とのインターフェースの仕様を確認する
(4) CPU とのインターフェースでやり取りするコ

生しません．慣習として行われないという方が正しい

図 3-5 複合コントローラのブロック図



マンドおよびデータについて確認する

(5) 特殊な機能についての関連がある部分を検討する

3.1.初期化と関連するタイミングチャート

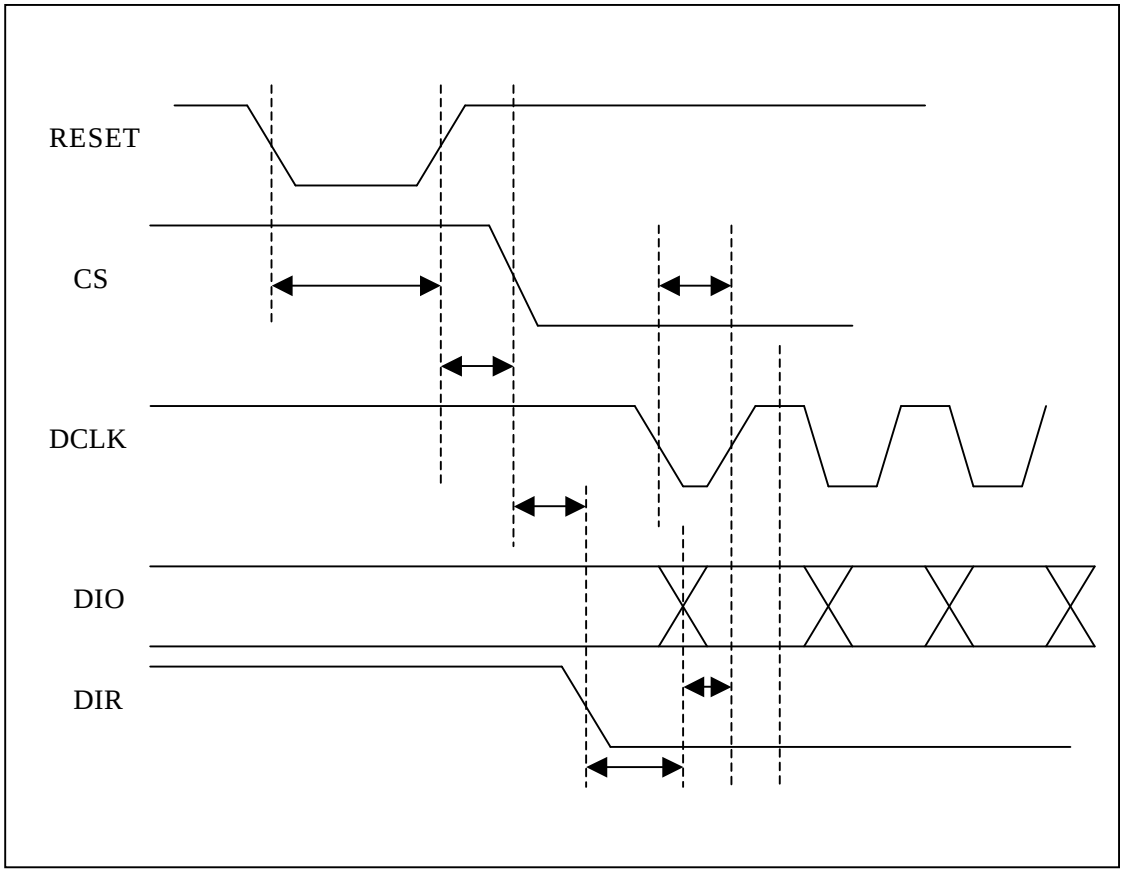
初期化および CPU との通信インターフェースに関するタイミングチャートを図 3-6 に示します．これはあくまでもデータシートに一般的に載っているタイミングチャートのイメージをつかんでいただくためのものですので，現実的な動作をイメージするチャートとして詳細に検討されているわけではありません.

ここで確認しなければいけないのは各パルスの縦の破線とその破線の間を示した矢印です．これはその破線と破線の間の時間間隔についての情報です.

3.2. 同期シリアルインターフェースでの転送

今回のワンチップマイコンと複合コントローラと

図 3-6 タイミングチャート

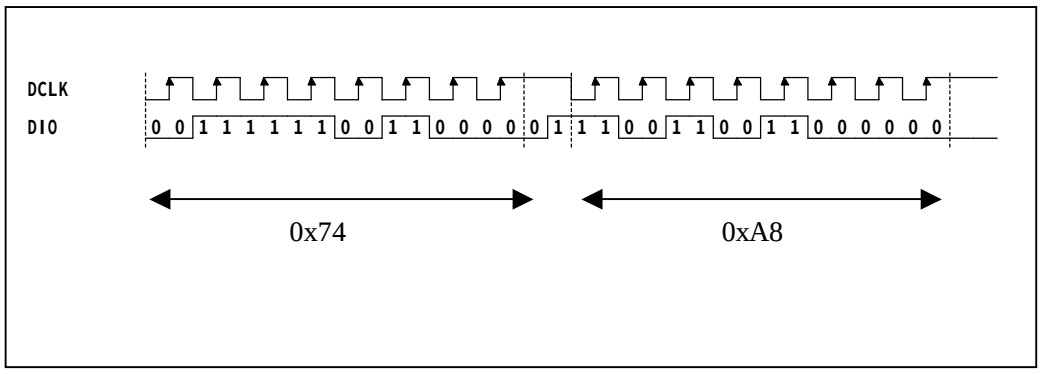


の間の通信は CPU 側にシリアルインターフェース通信用の専用コントローラを搭載していない為，ポート入出力でシリアル通信の機能を実現する必要があります．そのため，タイミングチャートで指定された時間間隔の調整は CPU で行う必要があります．

図 3-7 に同期式シリアル通信のイメージを示します．

データは DIO ラインで出力され，1 バイト単位に

図 3-7 クロック同期シリアル転送イメージ



上位のビットから 1 ビットずつ転送されます．このとき，DCLK ポートへのクロックデータの出力は DIO の状態に同期して，パルス波形を出力する必要があります．パルス波形と DIO の転送の約束事は，DCLK の状態が LOW(0)から HIGH(1)に変化する，立ち上がりエッジで，その時点の DIO ラインのデータが相手に転送される，ということです．

　このような方法によるデータ転送は今回の複合コントローラの特殊な例ではなく，部品点数を減らしたりワンチップマイコンに安価なものを使う目的で比較的よく使われる手法です．

3.3.コマンドデータ

データの転送はシリアル転送されますが，データそのものは複合コントローラで定義されているコマンドによりその意味付けを変えるような仕様とします.

コマンドはどのようなコード体系でも構わないのですが，機能としては以下のようなものが用意されているものとします.

これにより，キー入力，表示器へのデータ表示，ブザーについては大まかにできそうな気がしますが，以下の機能については不明です.

・ターンテーブルモータの駆動
・ターンテーブルモータの回転方向の切り替え
・照明の点灯
・マイクロ波出力器の出力
・ドアの開閉状態の検出

これらの情報はデータシートの検討だけでは，実は解決しません．複合コントローラに接続されるこれらのデバイスとの接続条件，および，個別のデータシートの参照が必要になります.

ここで言う，個別のデータシートとはターンテーブルモータと照明，マイクロ波出力器があります.

コマンド名	機能
KEYREQ	キー用 FIFO[14]バッファの先頭のキーデータを取り出す. キーコードはマトリックスキーの実際の回路に依存するので回路設計者より提示される．また，キーが押されたまま持続する場合は，押されつづけているという意味のキーコードが転送され，離された時点でそれを表すキーコードが転送される. キーデータがない場合は 00H を返す.
KEYSTS	キー用の FIFO バッファに保存されたキーデータ数を返す.
DSPN	数字表示データ転送 1 バイトの上位 4 ビットが表示桁位置を表し下位 4 ビットで 0 から９までの数字を指定する. 表示器は 16 桁まで対応する.
DSPCLR	表示器の表示を全て消去する.
BZR	ブザー出力の周波数と鳴動時間を指定する.
GPIOCTL	汎用入出力ポート 5 本分の入出力方向を指定する. 実際にポートにどのような相手が接続されるのかは回路図に依存するので，回路設計者より提示されるものとする.
GPIOOUT	汎用入出力ポートで出力ポートに指定したポートにデータを出力する.

[14] First In First Out の略でデータを受け渡すときにある複数のデータ単位がメモリに格納できて，メモリに格納した順番で取り出すというデータの受け渡し方．格納する処理と取り出す処理は非同期で動作できる.

4. 仕方なく回路図をみる？

これまで，ワンチップマイコン GIHYO80 と複合コントローラ KP07C001 のデータシートについて検討しましたが，結局は分からない情報がありました．これらの情報は最終的には回路図を見るしかありません．または，回路図の先に接続された別のデバイスのデータシートの参照が必要になります．

この情報の欠落については本特集では省いてしまいます．そのついでに，以下のように決めてしまうことにします．

(1) ターンテーブルモータの回転要求は複合コントローラの 2 本の出力ポートで制御し，両方が LOW の時，モータ停止．片方のポートだけが HIGH の時に，右回転．もう一方だけが HIGH の時に左回転とする．
(2) 照明は複合コントローラの 1 本の出力ポートの状態が HIGH の時に点灯し，LOW の時には消灯する．
(3) マイクロ波出力器の起動は複合コントローラの 1 本の出力ポートを使い，出力器の ON/OFF を制御する．マイクロ波出力器の電源の制御は CPU に接続されたポートにより行う．
(4) ドアの開閉状態は複合コントローラの 1 本の入力ポートを使い LOW の時に閉じていて，HIGH の時に開いている

これで，一応はソフトウェアの開発のための設計を行うことができるようになったと思います．

上記の決め付けの HIGH(1)/LOW(0)の状態は実際の回路設計により「論理が反転」する場合があるということを頭の片隅においてさえいれば，回路図は見る必要はありません．この条件に依存する部分だけをハード屋さんに確認し，仕様の提示を受ければ良いでしょう．余り，深追いする必要はありません．

ちなみに，このように「勝手に決めてしまって話を先に進める」というのも設計の一つのやり方です．

5. おわりに

データシートにはここに挙げた情報だけではなく，非常に詳細な電気特性なども書かれています．これらの資料は組込みシステム開発に慣れていないソフトウェアエンジニアが回路図の次に敬遠するものだと思います．

しかし，見るべきところは非常に少なく，必要なところを抜粋して見ることに慣れてしまえば，自分なりの仕様書に展開することが出来るようになります．また，タイミングチャートとその制限時間を記した表の情報の中から開発に必要な時間制約条件さえ把握してしまえば，後はデータ転送などのコマンドデータについて検討，整理すれば良いだけのものになります．

これらのデータシートはハードウェア設計者を読み手に想定しているために電気の知識が必要なように見えますが，実は，大したものではない，という見方も出来てしまいます．必要な情報をいかに抽出するかが必要なのであり，すべてを理解することは目的ではありません．

直接的に関連するコントローラなどのデータシートの読解により，場合によってはもっと他のデータシートを参照する必要がある場合も見えてきます．また，回路図などの参照が結局は必要になってくるという状況も見えてきました．このような状況が見えさえすれば，「何がわからないかわからない」という状況からは脱することが出来，わからない部分を適切に問い合わせることもできるようになります．

最低限でもこの状態に持っていければデータシートの読解に関しては問題はありません．

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■第４章　ソフトウェアブロック図を描いてみよう
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

1. はじめに

前章までの検討ではまだハードウェアの情報で不明点がありますが，ここでソフトウェアの構成について検討に入りたいと思います．いかにウォーターフォールモデルとは言っても，設計を行う過程では行ったり来たり，横道にそれたり，ということを恐れてはいけません．

別の視点で設計作業を行うことにより，見えなかった問題点が見えてきたりします．

本章では，ハードウェアの現実的な構成を踏まえつつ，ハードウェアに依存する部分と依存しない部分とに分けて，抽象化することを目指して，ソフトウェアブロック図を描いていきます．

2.　ソフトウェアブロック図の分割のしかた

2.1. デバイスの抽象化と分離

現在までにわかっているハードウェアの情報から，以下のようなソフトウェアの構成が考えられます．

・CPU に接続される内蔵コントローラおよび外部の複合コントローラの機能毎にデバイスドライバを設ける
・アプリケーションは抽象化された API を通してデバイスのアクセスを行う
・デバイスの種類，詳細なインターフェースはアプリケーションには無関係に完結させる
・時間管理は OS を使わない場合でも独立したインターフェースを提供できるように分離する

図 4-1 が今回検討している電子レンジを制御するためのソフトウェアのブロック図です．

ハードウェアブロック図とほとんど同じように見える部分もありますが，ハードウェアブロック図では現れなかったブロックを含めて，異なる部分は以下です．

・時間を管理する処理ブロックが追加されている
・全体を管理するという機能ブロックが追加されている
・デバイスドライバが 2 階層存在しその他の機能を表すブロックと分離している

ソフトウェアブロック図を描く場合に重要なのは，ある程度実装が見えた上で描かないと意味がないということです．矢印がブロックの間を結んでいますが，この矢印はシステムコールの発行や関数コールなどになります．このとき，どのような引数が必要かなどはある程度この時点で把握できている必要があります．

とは言え，最初からそのようなインターフェースが見えていなくても構いません．ブロックを描いて，線を引くだけでも問題点や機能についての整理ができるようになります．

以降，それぞれのソフトウェアブロック間のインターフェースについて検討してみましょう．

2.2. 時間管理ブロック

この機能ブロックはリアルタイム OS がある環境では OS 部分となり，以下のような機能が想定されます．

・スケジューラ
・メインループ
・アイドルタスク
・タイマハンドラ

また，OS がない環境では以下が想定されます．

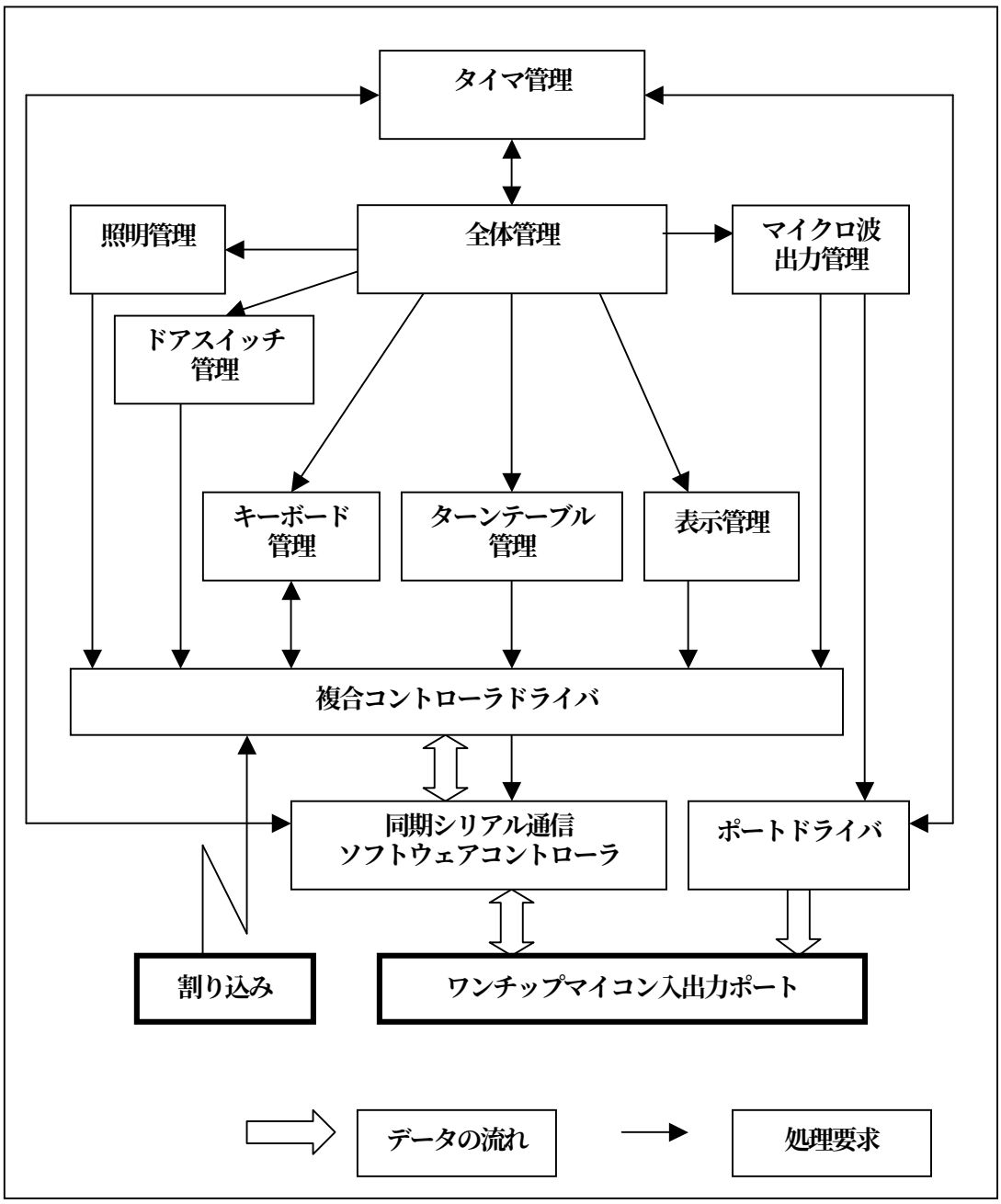
・メインループ処理
・タイマ割り込み処理
・タイマ監視を汎用化した関数群

タイマ管理は時間を計り，その時間によるイベン

トで処理を制御するために使います．ワンチップマイコン，GIHYO80 では汎用のタイマカウンタによる割り込みは１系統しか用意されていませんが，時間を計る必要がある機能は現在までの検討の中ですでに挙がっています．

今回の電子レンジの例で時間計測が必要なのは表向きの機能としては「あたため時間の計測」のみです．しかし，データシートの検討で現われたように各種パルス幅の時間制約でも時間の計測が必要に思えます．

図 4-1 ソフトウェアブロック図



では，それらがすべてタイマ管理処理としての時間計測の対象となり得るかどうかはその時間間隔の細かさの単位に依存します．

あたため時間は分秒単位ですが，データシートの検討で現われたパルス幅などの時間間隔はμs(マイクロ秒)，ns(ナノ秒)単位の時間になることでしょう．

一方，今回のプロセッサの水晶発振器は 10MHz ですので 10MHz の逆数が最低の時間間隔の単位となります．つまり，1/10,000,000 の値，100ns になります．

タイマ割り込みを発生させるタイマカウンタは表 3-1 より，16 ビットのカウンタを持っているということですので時間計測が可能な範囲は 100ns～6.5535ms ということになります．ただし，現実的には CPU のクロックと同じシステムクロックで時間間隔をカウント可能なことは少なく，システムクロックの 1/2 とか 1/4 などを基準に考えることになります．つまり，1/4 の周波数を基準とした場合，妥当な時間間隔は 400ns～26.214ms ということになります．いずれにしても，秒はおろか分の計測を行うには短すぎますし，タイミングチャートのパルス幅を表すような基準に使うのも難しそうです．

仮に，400ns 間隔でタイマ割り込みを発生させるようにした場合，CPU の命令時間が 1 クロックで 100ns だとすると，4 命令実行する度に割り込み処理を行うことになります．これでは，他には何も処理が出来なくなることと同じになります．

したがって，システムの基準として利用するようなタイマ割り込み処理の間隔は，一般的には短くても 1ms 程度の設定で使用することになります．現実的には 10ms 程度でも問題はありません．

この 10ms の根拠ですが，高速な応答性が必要なリアルタイムシステムでもない限り，センサーデータを取り込む間隔としても十分ですし，ユーザーインターフェース系，つまり，キーボードの取り込みなどを行う場合でも十分に細かい時間間隔と言えるからです．

人間がキー操作をしたときのレスポンスや表示の更新間隔で遅いと感じ始めるのは概ね 100ms 程度からです．ですので，十分に反応を良くしても 50ms より短くすることはあまり意味がありません．

今回，キー取り込みは複合コントローラに任せていますから，キーが押されたかどうかを周期的に見るような処理を行う必要はありません．

整理すると時間にかかわる要件は以下になります．

機能	時間単位
あたため時間	1 秒
キーに対する反応	50ms
表示の更新	50ms
あたため中のドアオープン ※ハードウェア的にマイクロ波出力は停止するのでソフトウェアでの応答は遅くて構わない	50ms
あたため開始からターンテーブル回転までの応答時間	100ms
あたため開始キー押下からマイクロ波出力までの応答時間	100ms
通信タイムアウト制御	50ms
出力ポートの定期更新	50ms
キータッチ音の応答時間 ※複合コントローラ担当なので管理不要	不要

このように，組込みソフト開発では時間を計測するという機能が不要であるというケースは全くありません．そして，リアルタイム OS などの OS を使わない場合でも必ず実装する必要があります．

したがって，タイマ管理処理は時間を計りたいタスクなどが教えてほしい時間を引数として要求するとその時間間隔または時間が経過したら教えてくれる，機能がなければなりません．

教えてほしい時間を設定する，というのは関数の引数で渡してもいいでしょうし，複数のタスクから設定，参照可能なメモリ領域に書くことでも指定することができます．ですが，管理機能として分離するためには，共通の領域を使うのではなく，機能要求の関数などの引数で渡すべきでしょう．そして，「教える」という機能は出来れば割り込み処理のようなイメージでの実装が望ましい形態です．つまり，

何らかのイベントが発生したら特定の処理プログラムが実行される，というイメージです．

リアルタイム OS を使う場合には最初からこのような機能が用意されていますので，それを利用するだけですが，OS のない環境ではその機能は作りこまなければなりません．

この点は，細かい実装の話になりますので，上位の設計を行う，本特集では省きます．

なお，データシートのタイミングチャートでの時間制約に関する処理はこのタイマカウンタでは行えませんので，別の方法で時間を計測することになります．多くの場合，パルス幅は規定されている時間「以上」という条件のつけ方はされても，「以下」という制限のされ方は行われません．「以下」が指定されるのは何かの変化に対してのハードウェア的な制御が必要な場合のみと考えて間違いではありません．そのような制約はソフトウェアで守るのは現実的ではないからです．

今回は複合コントローラとの通信はソフトウェアでクロックを発生させる都合上，命令の実行時間が時間の最低単位となり，命令の実行ステップ数でタイミングを取ることになります．

以上のことから，タイマ管理機能に対する要求の仕方（インターフェース）は以下になります．

・10ms 単位から 10 分単位の時間までが指定できる
・計時中のタイマ計測のキャンセルが指定できる
・タイムアップ時に実行する処理を指定できる
・要求タスク毎に独立して時間の指定が出来る

なお，ここでいう「要求」というのは実装の仕方によって，動的である必要はありません．あらかじめ静的に指定できる，ということでも良いでしょう．

「静的に指定できる」とは，例えば，あらかじめ処理毎に 10ms 間隔で周期的に処理したいとか，１秒間隔で処理したいというような形で処理の開始アドレスをテーブルで固定持つ, というような場合です．この場合は，１秒間隔で処理を行いたい，という要求はあらかじめプログラミングで決まっていますので，このような場合は「静的に要求した」ということになります．

2.3. 全体管理ブロック

この機能ブロックはイベントが発生することにより動作状態が変化することによる，その時々で行うべき，異なる処理を管理する機能です．

例えば，電子レンジを購入して来て，まだコンセントに差し込んでいない状態が「未起動状態」，コンセントに差し込まれて電子レンジとしてのキーを受け付けるようになるまでの間の状態が「初期起動中」，というような感じです．初期起動中にキーが押されても反応できません．初期起動が終わって「待機中」になってはじめてキーを受け付けることが出来るようになります．

このような状態管理は「状態遷移図」で検討することになりますので，次の章で詳しく説明します．

2.4.ソフトウェアブロックの集約

図 4-1 のソフトウェアブロック図を第 2 章最後の図 2-9 のフローチャートで示したように，詳細検討と簡略化の繰り返しを行うと，ハードウェアの情報を排除して簡略化したソフトウェアブロック図を描くことができます．図 4-2 にそれを示します．

この簡略化，集約化を行う場合の視点の持ち方はいくつかありますが，そのうちのひとつが「並列にそれぞれが独立したイメージで動作する必要があるか否か」です．ドライバやタイマ処理は割り込み処理も絡んで独立して動作する必要があります．

しかし，それ以外のアプリケーション部分は全て順番に処理すれば良いものです．

以下のように順番に行えばよい処理が箇条書きできるものは並列動作の必要はありません．

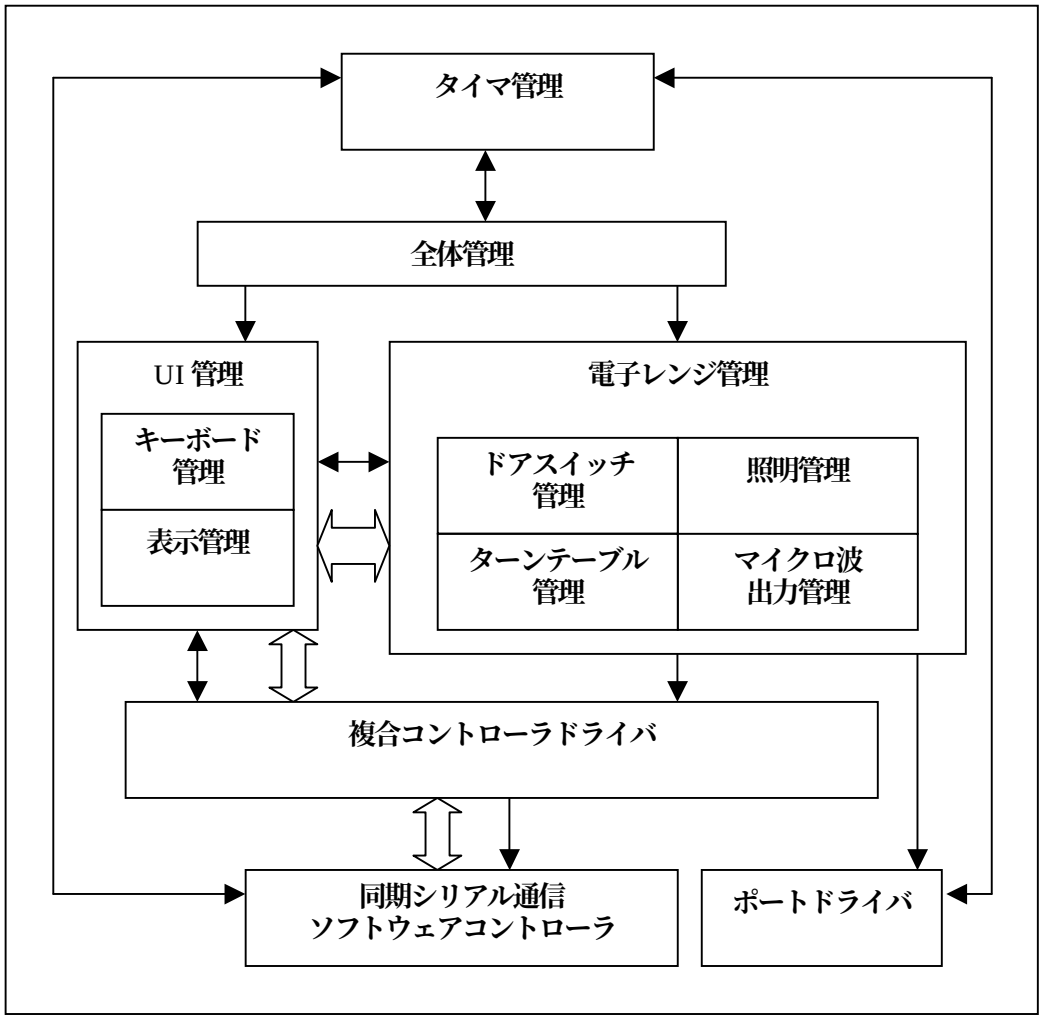
(a) 分キーが押されたら分をカウントアップして表示を更新する
(b) 秒キーが押されたら秒をカウントして表示を更新する
(c) あたためキーが押されたらマイクロ波出力要求を行う
(d) 照明を点ける
(e) ターンテーブルを回す
(f) 設定した時間が経過したらブザーを鳴らしてマイクロ波出力は停止
(g) ターンテーブルも停止
(h) 照明も消灯

また，集約に関しては，別の考え方もあります．それは，「依存度が１対１かどうか」です．この考え方の場合の視点は一度細かく分けたブロックに入る，入り口の矢印と出て行く出口の矢印が共通のブロックかどうかで判断できます．

具体的に見てみましょう．

「照明管理」は「全体管理」から矢印が入り，「複合コントローラドライバ」に出口の矢印が向かっています．「ドアスイッチ管理」も「全体管理」から入り，「複合コントローラドライバ」に向かっています．

図 4-2 ソフトウェアブロック図（ブロックの集約１）



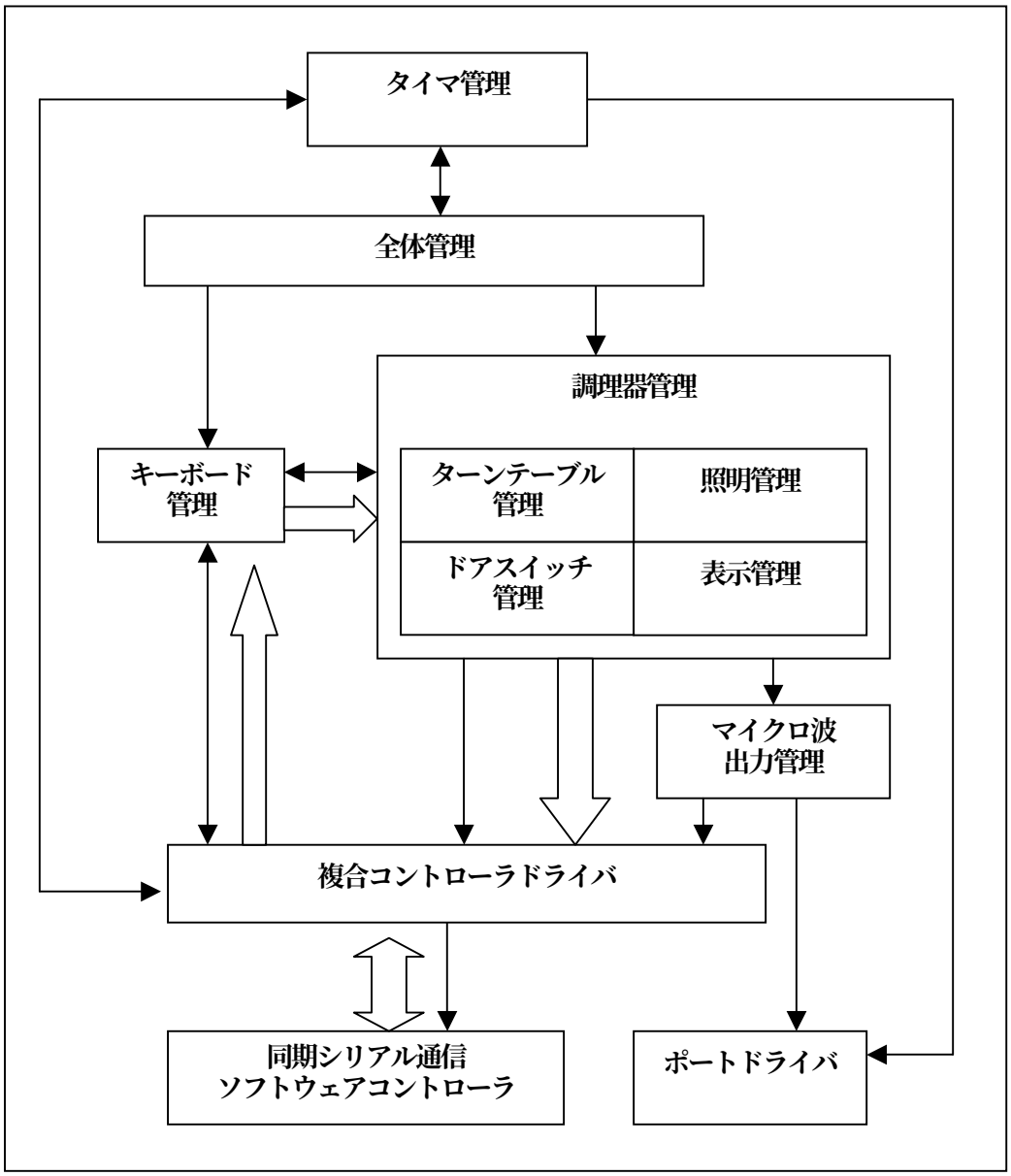
「ターンテーブル管理」と「表示管理」も同じブロックに集約できそうです.

「キーボード管理」は矢印が双方向になっている点が異なり,「マイクロ波出力管理」は出口の行き先が2つありますので,分類としては異なります.このようにまとめ直したものが図 4-3 になります.集約したブロックの名称は無理やり,「調理器管理」としてみました.「マイクロ波出力」という電子レンジの特徴がないのに「電子レンジ管理」という呼び方は不自然です.

このように集約したブロックにはその特徴を持つ名前をつけると,機能や問題点の整理が行いやすくなります.

さて,どちらのブロック図の構成でもソフトウェアの実装は可能でしょうが,どちらが適切かは,その製品のプロダクトの構成にも依存します.電子レ

図 4-3 ソフトウェアブロック図（ブロックの集約2）



ンジを何機種も作っているようなメーカの場合であれば，ソフトウェア部品の流用性を考えると「電子レンジ」の特徴を重視するでしょうし，電子レンジは 1 機種だけだから，流用は全く考えなくても良い，という場合もあるでしょう．

純粋にソフトウェアの構成だけを考えると最適解はひとつしかないと思いますが，メーカの体制なども考えると一概に言い切ることは出来なくなります．

3.おわりに

ソフトウェアの視点でブロック図を検討してみましたが，良くある間違いが細かい機能毎に単純に分割したブロックをそのままタスクとして扱ってしまうことです．本来は集約して単なるモジュールとして扱うことができるものがタスクとして分離することにより，リアルタイム OS を使う場合に CPU を非効率に使うことになります．その結果としてリアルタイム性能を落としてしまうという皮肉な結果になります．

これとは逆に機能的には小さくて何かのタスクの一部のモジュールとして扱いたいようなものでも非同期で動作し，いくつかのタスクから共通で使いたいにもかかわらず，リエントラント性を考慮しないようなモジュールとして実装してしまうこともよく見られます．

一方でリアルタイム OS を使わない場合のブロックの分割については単に表記上の問題だけとなり，間違ったタスク分割をしたとしてもリアルタイム性能には影響が出ない場合もあります．それは，OS のない環境では，並列動作するタスクとして分割しているのか機能モジュールとして分割しているのかが明確ではないからです．

リアルタイム OS などの OS を使う場合の開発ではこのブロック図がそのままタスク関連図やプロセス関連図に置き換わるようなものとして検討が進められる場合がありますが，視点をいくつか変えてソフトウェアブロック図としてのバリエーションを増やして複数の図を描いてみると，それまで見えていなかった問題点が見えてくる場合もあります．

■■■■■■■■■■■■■■■■■■■■■
■第 5 章　状態遷移図を描いてみよう
■■■■■■■■■■■■■■■■■■■■■

1.　はじめに

状態遷移図はステートチャートとも呼ばれますが，システム全体やプロセス，タスクなどの処理の単位のプログラムの状態の変化とその状態変化を引き起こすイベントについてまとめた図です．

今回描く状態遷移図の約束は以下です．

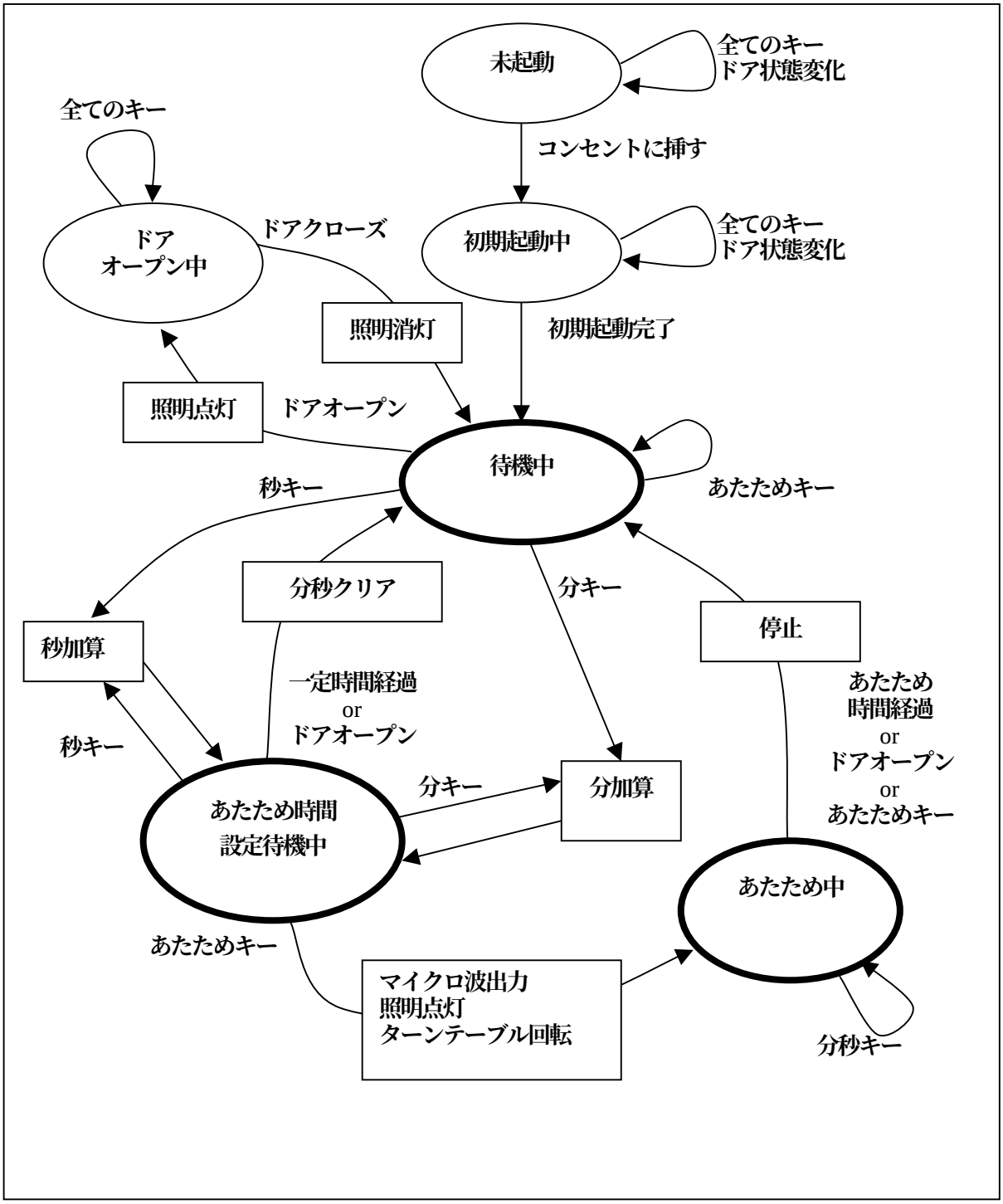
・状態は丸または楕円で表現する
・状態から状態への遷移は矢印で表現する
・矢印の脇には状態遷移の原因となったイベントをことばで書く
・四角で囲んだ部分は状態遷移の過程で実行される処理を示す
・矢印がその元にあった状態に戻る場合にはイベントを無視して状態遷移はしないことを意味する
・キーを押すイベントについては「押す」は省略

イベントは全ての状態で発生する可能性があるので，それぞれの状態から発生するイベントに対しての挙動は全て網羅して検討する必要があります．

2. システム全体の状態遷移図の概要

今回の「電子レンジ」の開発でのイベントは表向きには「ドアを開く」，「ドアを閉じる」，「分キーを押す」，「秒キーを押す」，「あたためキーを押す」のたったの 5 つです．ですが，ハードウェアブロック図やソフトウェアブロック図でのブロック間の矢印の意味，データの流れなどを眺めているとそれだけでは足りないことが見えてきます．足りない点の大きなところではソフトウェアブロック図で挙げた「時間管理」に関して，時間的なイベントがあることが想像できますが，その状態遷移についてのイベントが欠けていることです．

図 5-1 システムの状態遷移図



また, 状態についても機能的には, 「待機中」, 「キー操作中」, 「あたため中」の 3 つだけですが, ドアを開ける, 閉めるというイベントに対応する状態が

欠けていますので「ドアが開いている」という状態を追加します.

この状態を洗い出す手順ですが，イベントとして挙げたものが網羅できるかどうかを考えて拾い出して行きます．あらかじめ洗い出している材料では網羅されていなかった，足りないイベントや状態があれば，実際に状態遷移図を書いていくことにより見えてくることもあります.

3. 機能要件で現われないイベントと状態

以上のような検討を元に描いた状態遷移図が図 5-1 になります．これまで，検討した項目を基本的には網羅したシステム全体としての状態遷移図ですが，事前の機能要件では挙げていなかったものがあります．それは「あたため中」から待機中に遷移する条件に「あたためキー」というイベントがあることです.

状態遷移を描きながら検討することにより，通常の電子レンジでは「取り消しキー」があることを思い出したからです．いくらなんでもあたための最中に取り出したくなったときのキャンセルの機能が「ドアを開ける」というイベントしかないのだとしたら，非常に使いにくいものになることが分かったからです．それで，あたためのキャンセルの機能を追加するために「あたため中」の「あたためキー」の入力は「キャンセル操作」を行うことに「勝手に機能を追加」しました.

現実の開発の現場でも設計を行っていく過程で仕様提示の内容にどうしても納得できないものが出てくる場合があると思います．これはある意味仕方がないことです.

上位設計や製品企画の段階では細かい設計までは落とし込んでいないのですから，細かい問題点を見落としてしまいます．そんな状況の中では逆に開発現場からフィードバックして有益な製品開発にフィードバックできる場合もあります．設計検討を進めていってはじめて見えて来る問題点もありますし，細かい実装をつめていく段階，デバッグの段階，結合テストの段階などで，はじめて気づいてくる問題点もあります.

4. 経由する状態遷移

さて，改めて，図 5-1 の状態遷移図を眺めてみましょう．何かおかしな点はないでしょうか？

それは「あたため時間設定待機中」と「あたため中」に発生する「ドアオープン」イベントについてです．どちらも「ドアオープン中」に遷移すべきなのにそうはなっていません.

これは図の描き方としてそういう形にした，という言い方も出来ますが，実際の開発に則した表現であるとも言えます.

その説明の前に，この図の展開では現実的な問題があるかどうかを状態遷移図そのものを読むことにより判断してみましょう.

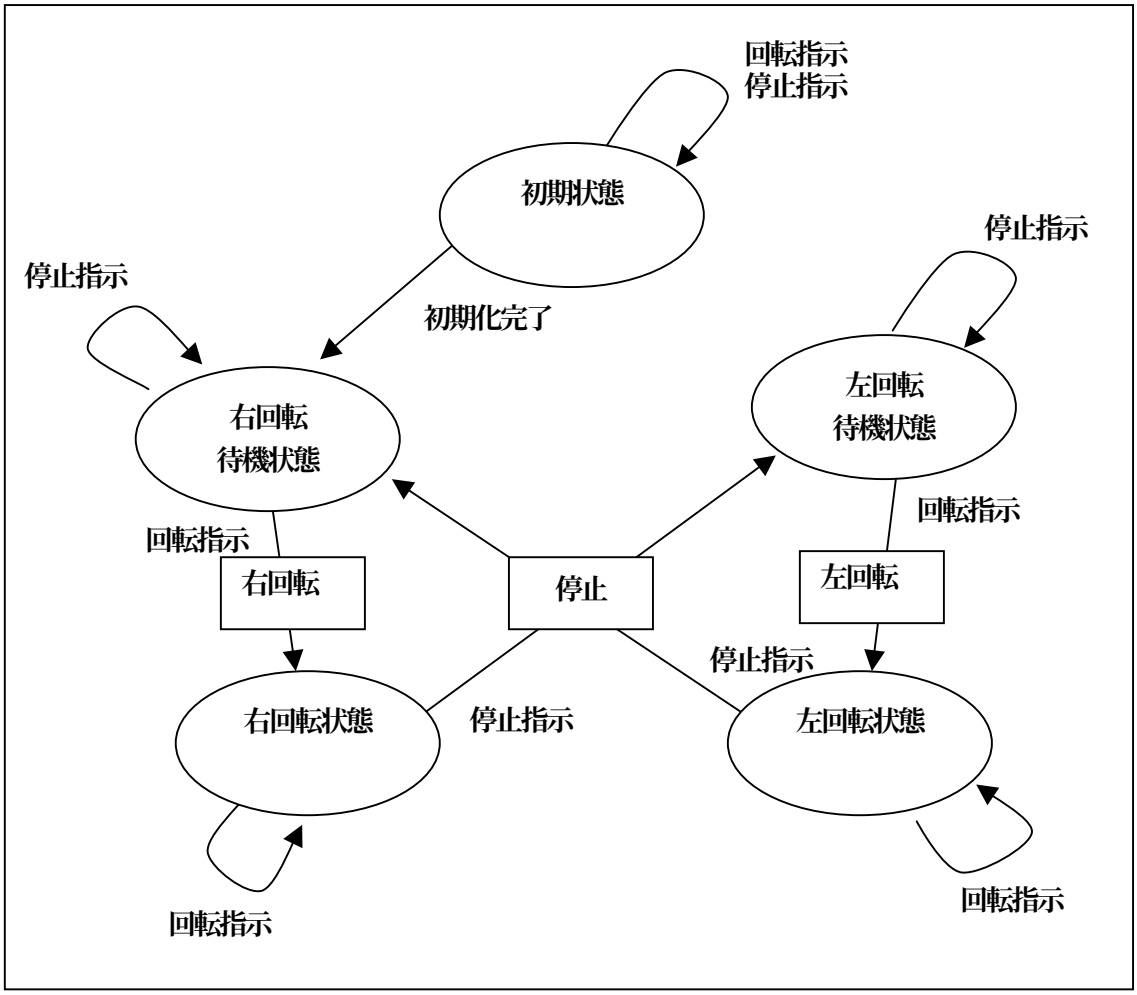
「あたため中」の「ドアオープン」イベントではあたためを停止して, 待機中に遷移します. そして，「待機中」状態でも再び「ドアオープン」イベントを検出してしまいますので，最終的には「ドアオープン中」に遷移します．過程にまどろっこしいものはありますが，最終的には問題なさそうです.

次に，「あたため時間設定待機中」の「ドアオープン」イベントの扱いですが，こちらも一度「待機中」に遷移した後は「あたため中」のときと同じ結果になります.

このように，いくつかの状態を経由して目的の状態に遷移するような表現で図を描くことは問題ありません．それどころか，「実際の開発に即した表現」と合致することも多いのです.

今回の場合，「あたため中」から最終的な「ドアオープン中」に遷移するときと「あたため時間設定待機中」から「ドアオープン中」に直接状態遷移することを考えたとき，それぞれ，一度，「待機中」を経由する場合と経由しない場合とを比べたときに処理

図 5-2 ターンテーブル管理の状態遷移図



は全く同じものになるでしょうか.

「あたため時間設定待機中」で見てみましょう.

「ドアオープン中」に直接遷移する場合は,「分秒クリア」処理と「照明点灯」処理の両方を行う必要があります.「一定時間経過」で「待機中」に遷移する場合には「分秒クリア」しか必要ありません. 直接遷移せずに「待機状態」を経由すれば一律に待機状態へ遷移するときの「分秒クリア」だけを処理すれば良いことになり, 処理を共通, 簡略化することが出来ます.

5. タスクやプロセス単位の状態遷移

今回の電子レンジの開発ではシステムの状態遷移をそれ以上詳細に記述する必要はないようですが, 規模が少しでも大きくなるとタスクやプロセス, 処理モジュール毎に状態遷移の検討が必要になります.

基本的な手順としてはシステムと同様です. 検討内容としては詳細に検討はしますが, 状態やイベントは外部仕様的な, あくまでも, 外部要因としてのイベントと状態を検討します.

タスクやプロセス, モジュールとしての外部要因というのは他のタスクやプロセス, モジュールからの要求イベントをトリガー[15]にする, という意味で

[15] 拳銃で言えば引き金のこと. 何かのきっかけのことだが, イベント発生により何かを起こす場合に良

す.

そのタスクの内部処理の勝手な状態遷移のことを整理しなければならないのではありません.

したがって，以下の点には十分な検討の必要があります.

(1) 全体の遷移が詳細検討時には見えなくなる（忘れてしまう）
(2) 時間的な制約が前提条件になっていることに気付かなくなる
(3) 非 OS 環境のタスクをリアルタイム OS 環境の実装に移植する際，周期間隔の依存度が他のタスクとの間で密結合している可能性がある

ここで，時間的な制約が前提とか，周期間隔の依存度が問題なのは，いずれも，非 OS 環境で起こる問題で，再利用性を妨げ，気付きにくいバグを生む要因にもなっています.

本来はタスクや機能分割したモジュールというものは他のタスクやモジュールとの間にはデータの受け渡し方法に関してのインターフェースとしての約束事があるだけで，時間的制約などは存在すべきではありません.

勝手な内部状態と外部状態の違いについて見てみましょう.

今回の電子レンジではターンテーブルの回転が，あたための度に前回と逆方向に回転するという機能要求がありました．これは外部要件での状態遷移が必要であることを示します．状態遷移図は図 5-2 のようになります．あくまでも外部からの要求に対しての状態遷移だということがわかるでしょうか.

これとは違って，例えば，単純にデータ処理を要求されるだけではなく，その頻度が実は周期的であり，時間を数えるのに使える，ということを前提にしたモジュールがあったとしましょう.

具体的には，そのモジュールから見れば，自分が呼び出されるのは 10ms 周期で起動されるタスクからだけで，初期状態で自分で勝手に作った内部保持用のタイマカウンタをクリアして，呼び出されるたびに，10ms 経過，20ms 経過というような処理を行い，タイマ監視に役立てていました.

このような処理を行っていることはそのモジュールのインターフェース仕様には現われません．内部で勝手にやっていることですから，インターフェースではないからです.

そのモジュールはあるデータ処理を行うための便利なモジュールですから，最初は 10ms 間隔で動作しているタスク A からの要求だけだったはずが，後から[16]，そんな事情を知らないタスク B が同じモジュールを呼び出したとしましょう．そのタスク B はそんな事情を知りませんから便利に使いました．タスク B がそのモジュールを呼び出す頻度はある特定の割り込みが発生したときだとしましょう.

タスク B からの要求は非常にまれであるため，条件が重ならないと発生しません.

ほとんどそのような事実が気付かれないまま，テスト項目からももれてしまう結果となります．このような不具合が，開発中のランダムなテストでも発覚しないで出荷後に発覚する不具合になってしまうのです.

6. おわりに

システムの状態遷移とそれより小さい単位の処理の状態遷移を図に表してみました.

モジュールの内部状態を勝手に制御する場合の危険性の例では，分かりやすいように時間を数える，ということで説明しましたが，外部からの要求回数など決まっているつもりで内部状態の遷移を行わせ

く使う用語.

[16] この場合の，「後から」というのは別のプロジェクトでそのモジュールを利用する，というような派生開発を指します.

て，制御を行うようなことは実際に行われることがあります．

しかし，モジュールとしての独立性を確保するという観点から言えば，このような実装は厳しく制限する必要があります．

一つのモジュールの内部的な状態遷移管理が全体のシステムに影響を及ぼすかどうかについては設計段階でも予想できません．それどころか，実装している担当者自身にも予想出来ていないことが多いと思います．

確かに，その特定の製品開発プロジェクトでは問題は発覚しないと思います．そのようなことからも問題を，見落としがちです．しかもそれが「稼動実績」として評価されてしまうわけです．動いているモジュールだから，安心だという実績です．その結果として，以降の製品開発でも流用されてしまいます．

そして，そのような実装ではこのような処理のことも「状態遷移」という表現を行っている場合があります．ですが，それはここで説明している「状態遷移」とは意味が大きく異なります．

「状態遷移」の検討は，必ず，インターフェースとしての外部イベントをトリガーとし，インターフェース仕様として表面に出ている状態を遷移させるようにしてください．

さて，本章が本特集の最後になります．電子レンジは動いたでしょうか？

細かい部分では仕様についてかなり省略させていただきましたが，想像力でまかなえるのではないかという期待を込めて省略させていただきました．展開に無理な部分はあったとも思いますが，ひととおりの開発に必要な情報をそろえられたとは思っております．