初級者向けテキスト

組込みソフトウェア管理者・技術者育成研究会
ーSESSAMEー
(http://www.sessame.jp/)

-*- 目　　　　次 -*-

-*-

組込みソフトウェア技術者・管理者向けセミナー

SESSAMEの紹介およびコースの概要

二上 貴夫／坂本 直史

1．SESSAMEの紹介およびコースの概要
2．開発課題と失敗事例の解説
3．組込み向け構造化分析の例・設計の概要(1)
4．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
5．組込み向け構造化設計(1)
6．組込み向け構造化設計(2) 実習/回答と補足説明

7．プログラミング　組込み用語基礎知識
8．ソフトウェアテストの概要
9．プログラミング実習への説明
10．プログラミング　実習
11．プログラミング　実習の回答と補足説明
12．ソフトウェアテスト　実習
13．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

ご協力をお願いします

- 運営
  - 楽しく学びましょう
  - 質問は、恥ずかしがらずその場でしてください
  - 最後にアンケートがあります。ご協力を！
- お行儀
  - 携帯電話は、マナーモードにしてください
  - 地震・火事での避難経路をご確認ください

日本の組込みソフトウェア開発

- 日本には無数の組込みソフト開発の現場がある
  - 大小あわせて数十万人の組込みソフト技術者
- 現場の大多数は、枯れた方法を使って開発
- 枯れた方法＝枯れた技術＋枯れたプロセス
  - 枯れた技術：仕様書＋コーディング＋実機テスト
  - 枯れたプロセス：不断の努力＋迅速な不具合対応
- いろいろな”咲いている技術”がある
  - これを使えることは重要だと SESSAMEは考える

3

どんな技術が必要か

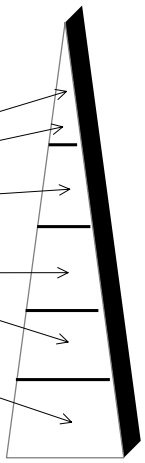
- 大雑把に組込み規模で必要技術は増える
  - 従来のCプログラムの見当で
    - 10^2ステップの世界（1～数千行くらい）
      - 個人管理技術（計画、予測、理解）
      - テスト技術（合理的な検証）
      - 構造化技術（機能分化と階層）

初級コース

    - 10^4ステップの世界（数万～数十万行）
      - 要求分析（ウインドステートなど）
      - オブジェクト指向技術（分散と協調）

中級コース

    - 10^6ステップの世界（数万～数十万行）
      - プロセス管理技術（チームと連携）



4

コース概要

- 講義と演習のペア
  - アンケートで要望の多かった演習を実施
  - チームで作業 / 発表もあり
    - 自分に役立つよう積極的に取り組んで下さい
- 題材
  - 講義　話題沸騰ポット
  - 演習　鹿脅し→ セミナ用に開発した教材

5

初級のカリキュラムと意図

- ソフトウェア工学の基本的な技術
  - コース対象
    - 入社 2～ 3年目程度のこれから活躍が期待される技術者
  - カリキュラムの組み方
    - 開発工程に沿ってどこの何を学べば良いかを示した
    - 例題は身近なものから選んだ
  - 有用な知識を選択
    - 分析と設計 :構造化から組込みに有効な範囲を摘出
    - プログラミング :組込み特有の基礎知識を解説
    - テスト:一般論＋組込みに役立つ知見を追加
- 自己診断 :学ぶべき項目をチェック

→ 足元固めて一層の飛躍を

6

開発課題と失敗事例の解説

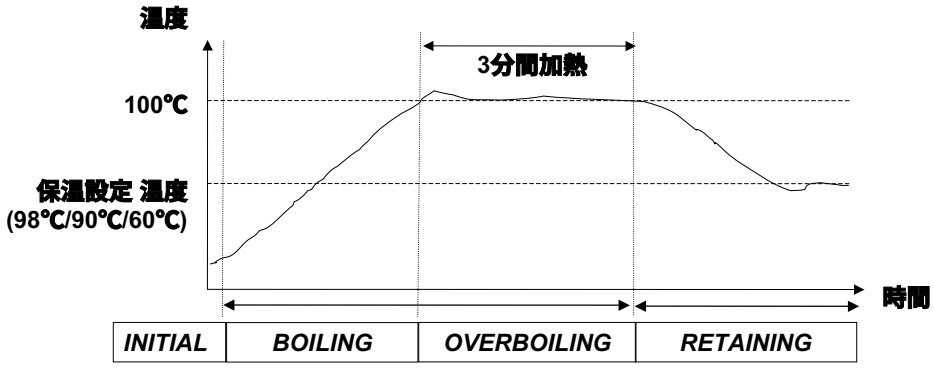
須田 泉

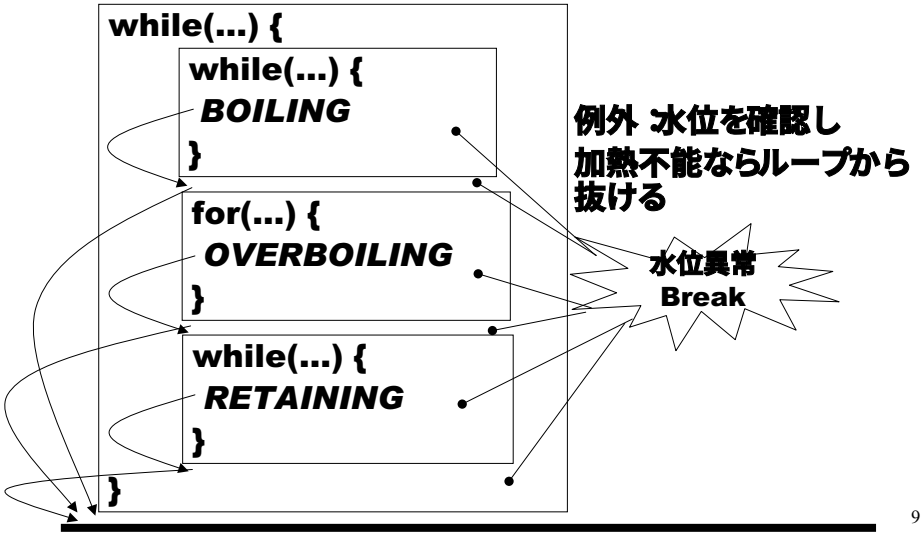
１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

失敗事例1　基本動作でコーディング





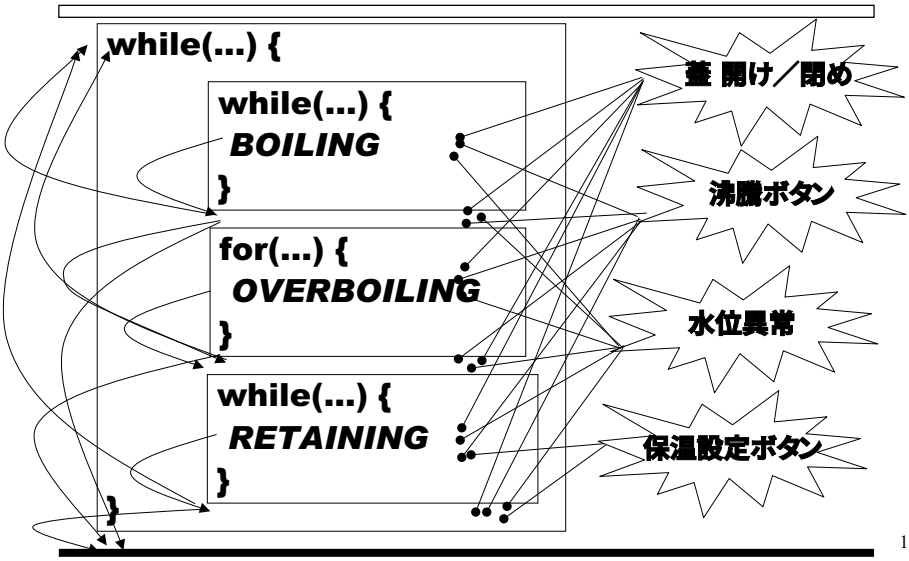
ユーザ操作によるイベントを追加すると

1、蓋の開け閉め→再沸騰

2、沸騰中に沸騰ボタン（沸騰中断）→保温

3、保温中に沸騰ボタン→再沸騰

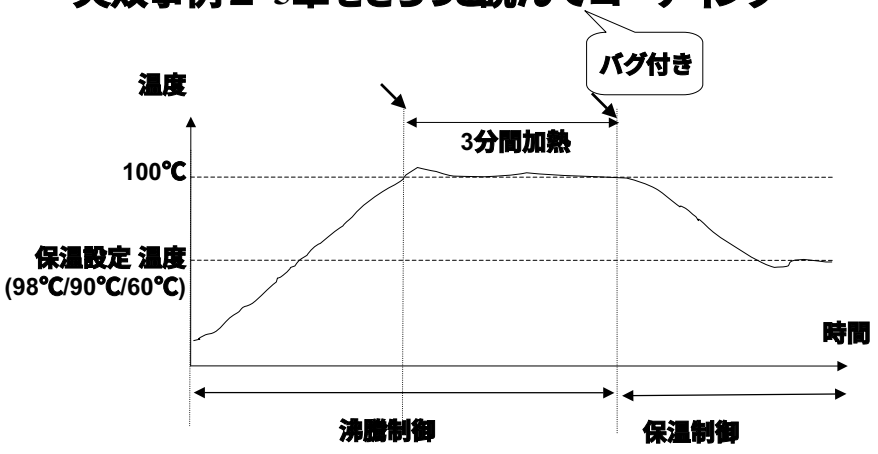
4、保温中に保温設定変更→目標温度変更



要求仕様の曖昧さに気付かない？

1、ミルクモードで沸騰完了すると100度で保温？
2、沸騰中に沸騰中断したら保温？
過渡期に表示パネルに矛盾が生じる
3、蓋が開いた時、沸騰ボタン、沸騰・保温ランプ以外の表示パネルはどうする？
4、蓋が閉まった直後の保温設定は前回を憶えておく？
5、タイマーは何分まで設定できる？
6、タイマーキャンセル方法は？

失敗事例２ 5章をさらっと読んでコーディング




```
while(...) {
    if (Portが変化) {
        Eventの解析
    }
    if (沸騰制御) {
        if (3分加熱終了した) {
            保温制御に遷移
        } else if (沸騰100℃に達した) {
            3分タイマースタート
        }
    }
    ヒーター制御（PID制御）
}
```

- 蓋（Open→Close）が変化
- 沸騰ボタンが押された
- 保温設定ボタンが押された

- OnOff制御？

ポットの状態

- **bool retain = False　/* 状態を保温制御/ 沸騰制御*/**
  →状態が追加できない(過渡期、エラー発生）

仕様の誤解

- **沸騰ボタンがトグル制御になっていない**
- **温度制御方式に漏れがある 沸騰時はOnOff制御）**
- **蓋が開く、水が無くなる、ポットが倒れる**
  →ハード的にスイッチが切れると想定している

原価削減→ソフトに仕様変更（よくある話）

失敗事例のようなプログラムを作ってしまうと

仕様変更で破綻（作り直し）
保守性が低い
再利用できない

トライ＆エラー
アプローチ

要求仕様曖昧
設計者の理解も曖昧

⇒ **品質の低下**

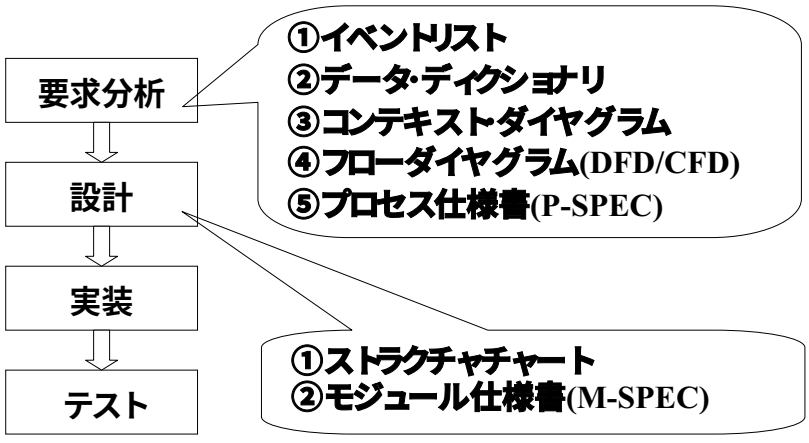
分析していますか？

余白）

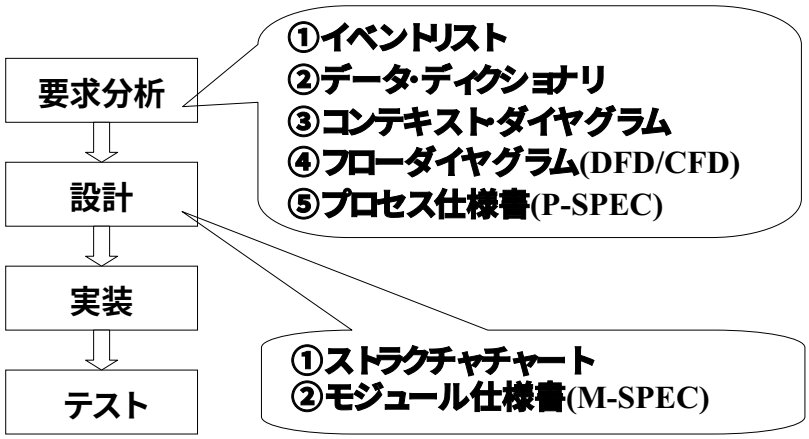
開発課題と失敗事例の解説
～組込み向け構造化分析の例～

鈴木 圭一

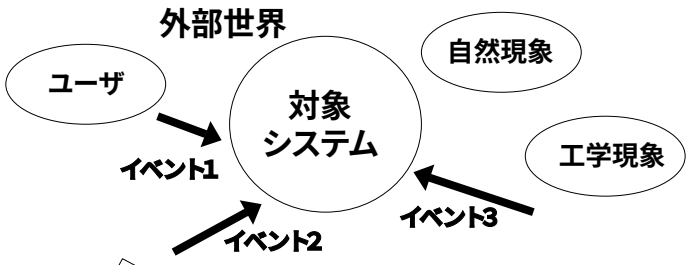
はじめに －構造化分析/設計の成果物－



アジェンダ



①イベントリスト (1)



イベントリスト

イベント	スティミュラス	アクション	レスポンス
イベント1			
イベント2			
イベント3			
⋮			

①イベントリスト (2)

- ***イベント***
  対象システムの外で生じ、その発生をシステムが制御できない事象。
- ***スティミュラス***
  イベントが発生したことを伝える情報入力。
- ***アクション***
  イベントが発生した時の対象システムの果たすべき機能。
- ***レスポンス***
  イベントが発生した時の対象システムの外部への反応。

＜イベントリスト初版＞

イベント	スティミュラス	アクション	レスポンス
タイマボタン押下	・現在の状態(タイマが押されているかいないか)	・タイマ起動 ・タイマ１分追加 ・ブザーを３回鳴らす	・タイマ残り時間表示
保温モードボタン	・現在の保温設定状態と押された回数	・保温モード(高温、節約)	・保温モード表示
ロックボタン	・現在のロック状態(ロック / 解除) ・蓋センサーの状態(開 / 閉)	・給湯	・ロックランプ点灯 / 消灯
給湯ボタン押下	・水位検出		・給湯口より水排出 ・水位メータで水位を表示
給湯ボタン離上		・給湯停止	
沸騰ボタン押下	・現在の状態(沸騰中 / それ以外) ・沸騰終	・沸騰中止→保温	・沸騰ランプ消灯 ・沸騰ランプ点滅 ・ブザーを３回鳴らす
蓋が開けられる			・沸騰ボタン操作不可 ・沸騰ランプ消灯
蓋が閉じられる		・温度制御可能な水位ならば沸騰状態に移行 ・沸騰終了後、保温状態に移行	・沸騰ランプ点滅 ・沸騰ランプ消灯

実装に関わる具体的な名前(how)は使わない

ユーザにわかる記述内容で表現する

イベントの発生を伝えるデータを記述する

＜イベントリスト改訂版＞

イベント	スティミュラス	アクション	レスポンス
タイマ開始	タイマ　時間	(1)タイマを起動する (2)指定時間を経過したらユーザーに知らせる	(1)・タイマ残り時間を表示する ・操作受付を通知する (2)・タイムアップを警告する ・タイマ残り時間を表示する
タイマ時間追加	タイマ　時間	・現在の残り時間に指定された時間を追加する	(1)・タイマ残り時間を表示する ・操作受付を通知する (2)・タイムアップを警告する ・タイマ残り時間を表示する
保温モード指示	保温モード	・保温モードを設定する ・温度制御を変更する	・操作受付を通知する ・温度／モードを表示する
給湯口のロック	（なし）	・給湯口をロックする	・操作受付を通知する ・ロック中を　表示 する
給湯口のロック解除	（なし）	・給湯口のロックを解除する	・操作受付を通知する ・ロック解除を　表示 する
給湯開始	（なし）	・ポット内の水を給湯する	・操作受付を通知する ・給湯口から水を排出する
給湯終了	（なし）	・ポット内の水の給湯を停止する	・給湯口からの水の排出を停止する
沸騰開始	（なし）	(1)水を沸騰させる (2)沸騰が終了したらユーザーに知らせる	(1)・沸騰中を　表示 する ・保温解除を　表示 する (2)・沸騰終了を警告する ・沸騰解除を　表示 する ・保温中を　表示 する
沸騰中断	（なし）	・水を保温状態にする	・沸騰解除を　表示 する ・保温中を　表示 する
水位の変化	水位	・水温を保温温度にする	・水位表示を更新する
満水	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を　表示 する
水なし	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を表示　する
蓋を閉じる	（なし）	・温度制御可能な水位ならば沸騰を開始する	・沸騰　中を 表示する ・保温　解除を　表示 する
蓋を開ける	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を表示　する
温度異常	（なし）	・温度制御を停止する ・ブザーを鳴らし警告する	・異常を通知する

25

②データディクショナリ

◆構造化分析で用いる各種図表の中で使用するデータの名前と定義を一定の順序で列記した一覧表。 随時作成）

記号	意味	説明
=	～から成り立っている ～に等しい	左辺の構成要素を右辺に示す
+	～に加えて ～と～	構成要素を並列に並べる（その順序に意味はない）
{ }	～の繰り返し ～の反復	{}で囲んだ構成要素を任意の回数で繰り返す。繰り返しの回数（上限や下限）を{ }外に書くこともある。
[|]	～の中から１つを選択する	[ ]内の構成要素から１つを選択
()	オプション	( )で囲んだ構成要素からの任意選択
" "	文字列	引用符で囲んだ文字列が、そのまま構成要素の内容となる

26

＜データ・ディクショナリ改訂版＞

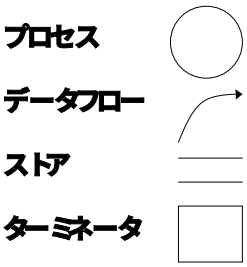
（イベントリスト改訂版時点）

- **タイマ残り時間**＝整数 下限：0 上限：9 単位：分
- **保温モード**＝［高温モード｜節約モード｜ミルクモード］
- **温度異常**＝［高温異常｜温度上がらず異常］
- **水位**＝整数 下限：0 上限：4 単位：なし
- **水温**＝単位：℃
- **高温モード**＝沸騰とほとんど同じように使うための保温モード
- **節約モード**＝消費電力を節約するための保温モード
- **ミルクモード**＝乳児の粉ミルク調乳用として使うためのモード
- **高温異常**＝水温が一定以上に上がった場合
- **温度上がらず異常**＝水温が上がらなくなった場合

27

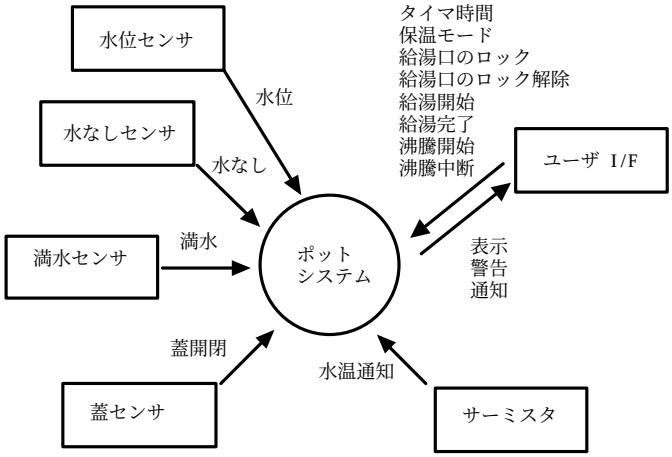
③コンテキスト・ダイヤグラム

- システムの機能要求をネットワーク図として表したものをデータ・フロー・ダイアグラム(DFD)と呼ぶ。そのDFDの中で、対象システムのシステムと外部環境との間にあるデータの境界を記述するDFDを特にコンテキスト・ダイアグラムと呼ぶ。つまりコンテキスト・ダイアグラムはDFDの最上位層である。



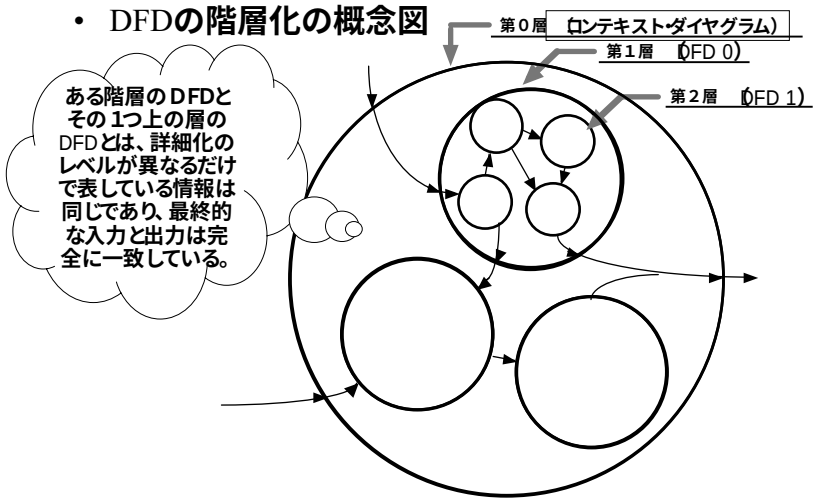
28

＜コンテキスト･ダイヤグラム改訂版＞

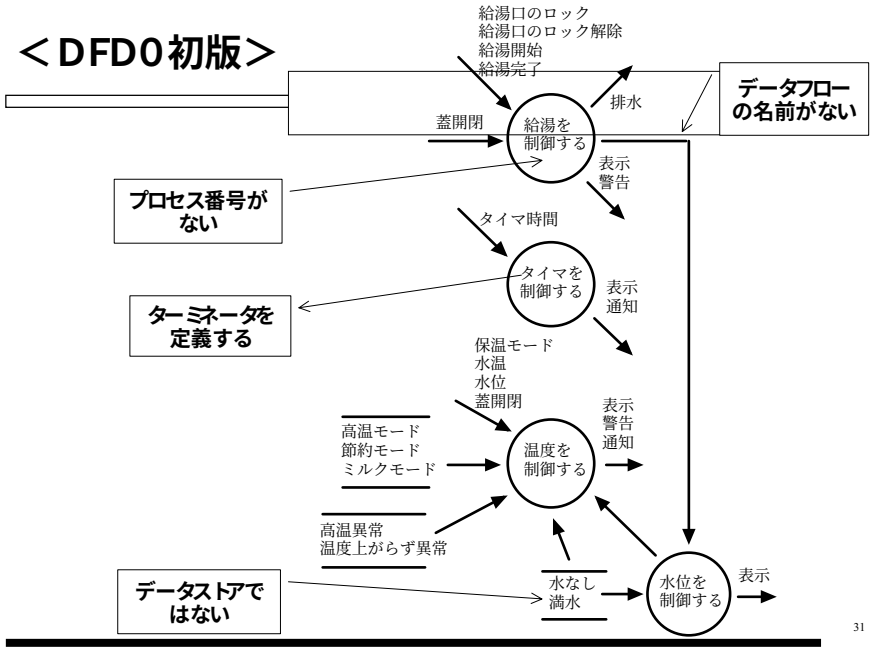


④フロー･ダイヤグラム（DFD/CFD）

- DFDの階層化の概念図



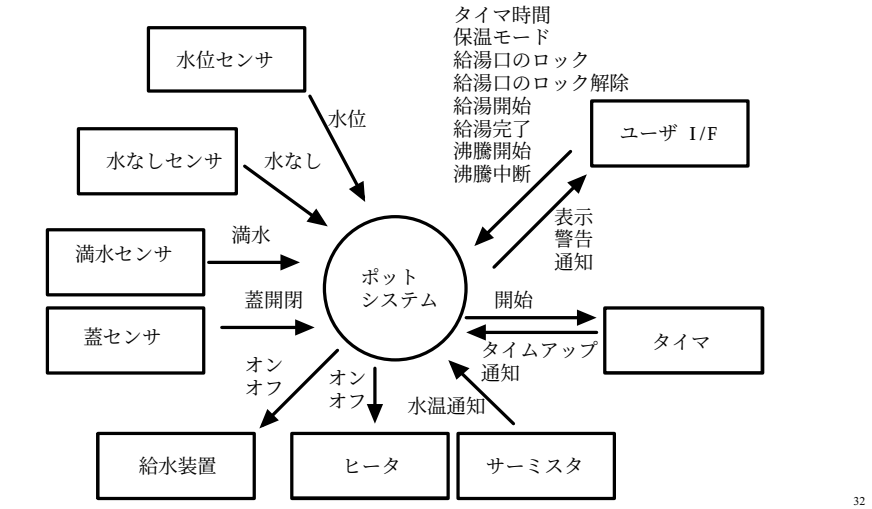
＜DFD0初版＞



31

SESSAME CONTENTS 2004

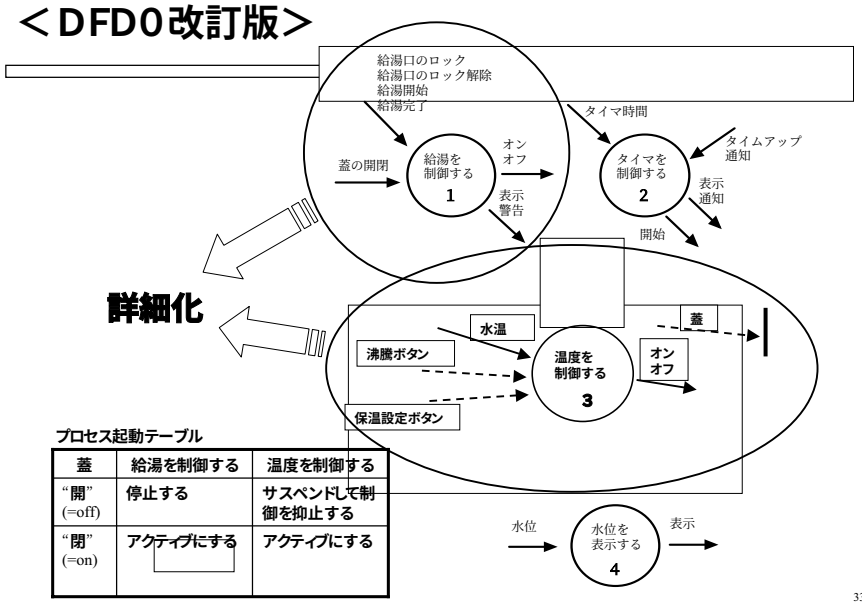
＜コンテキスト･ダイヤグラム再改訂版＞



32

SESSAME CONTENTS 2004

＜DFD0改訂版＞

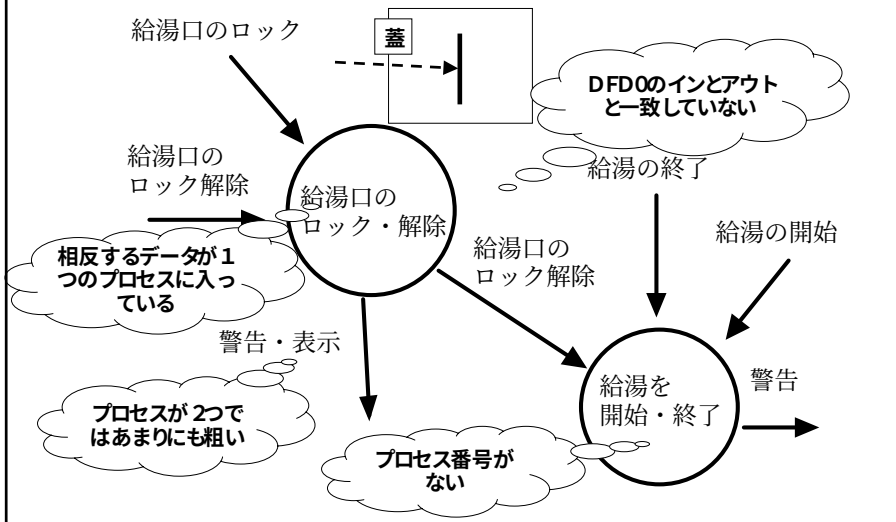


プロセス起動テーブル

蓋	給湯を制御する	温度を制御する
“開” (=off)	停止する	サスペンドして制御を抑止する
“閉” (=on)	アクティブにする	アクティブにする

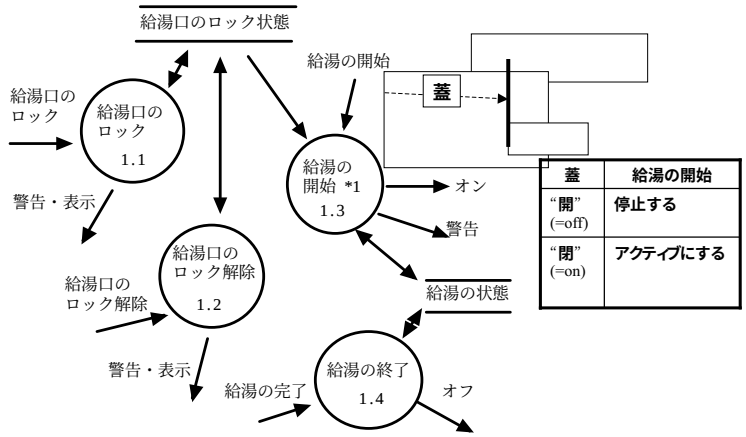
33

＜プロセス1のDFD1初版＞



34

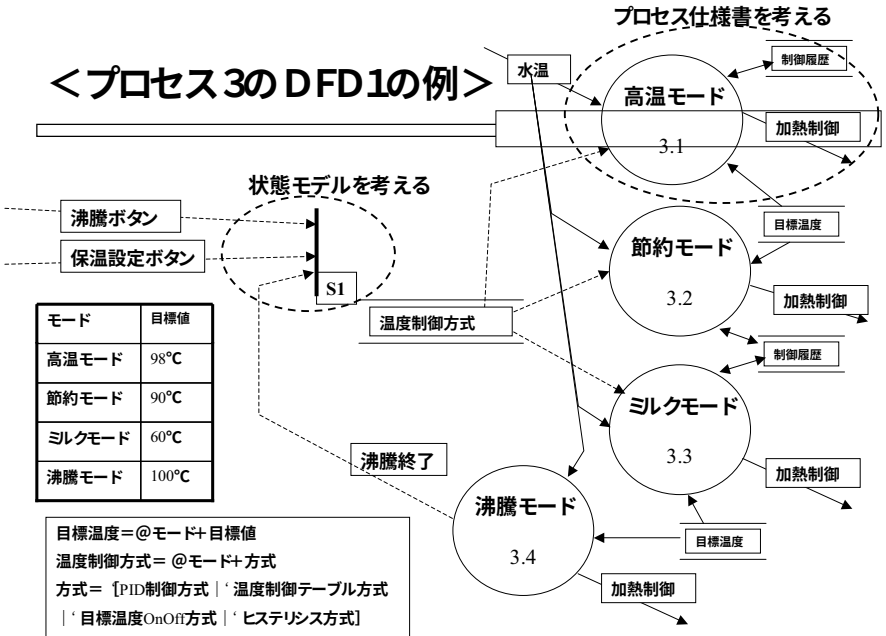
＜プロセス1のＤＦＤ1改訂版＞



蓋	給湯の開始
“開” (=off)	停止する
“閉” (=on)	アクティブにする

*1 水位と関係なく実施

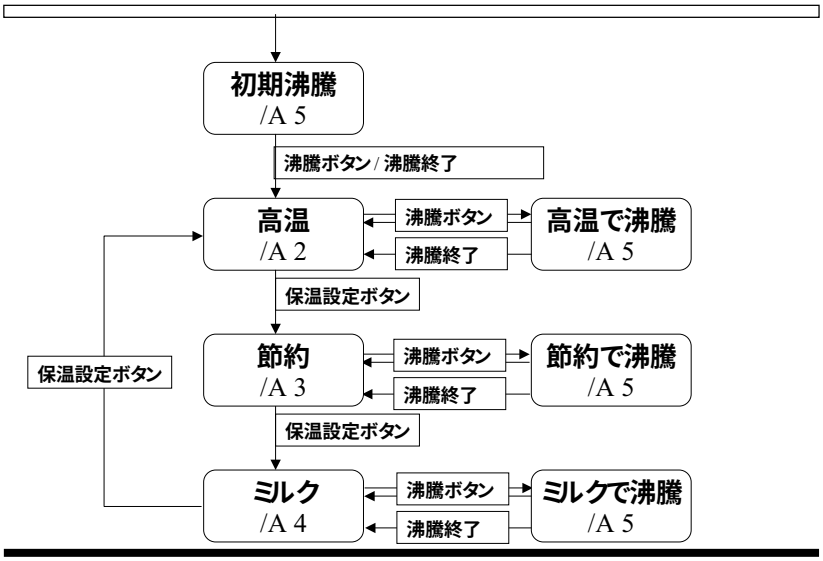
＜プロセス3のＤＦＤ1の例＞



モード	目標値
高温モード	98℃
節約モード	90℃
ミルクモード	60℃
沸騰モード	100℃

目標温度＝@モード+目標値
温度制御方式＝ @モード+方式
方式＝【PID制御方式｜‘温度制御テーブル方式｜‘目標温度OnOff方式｜‘ヒステリシス方式】
制御履歴＝前回の加熱率＋T2+T1+T0

＜S1状態モデルの例＞



37

⑤プロセス仕様書（P-SPEC）＜プロセス3.1の例＞

- 高温モードでPID制御する

制御周期ごとに、

1. //水温履歴を更新する

$T_2=T_1$

$T_1=T_0$

$T_0=$ 水温

2. //目標温度を得る

T_gを目標温度．モード＝高温モードである
目標温度．目標値に設定する

3. //PIDでの制御量を計算する

$\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

今回の加熱率＝前回の加熱率＋ΔM　//加熱率が100%以上になることは今回、考慮しない。

4. //ヒータ通電期間を制御する

制御期間×今回の加熱率　の期間中は、

加熱制御="on"

この期間が終了するタイミングで

//ヒータを切る

加熱制御="off"

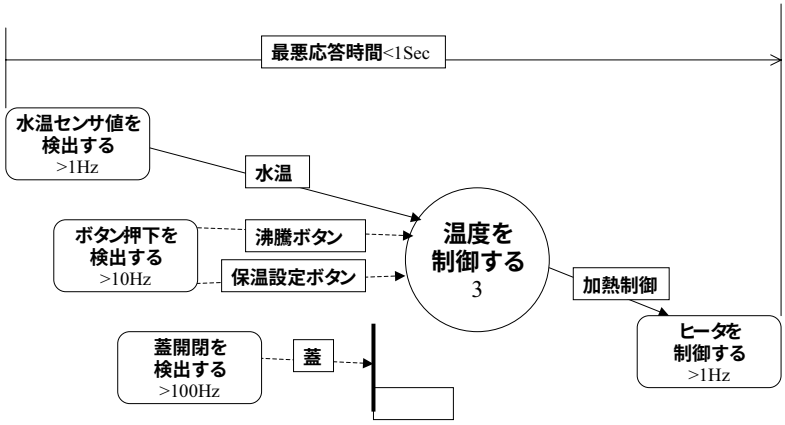
//加熱率の履歴を更新する

前回の加熱率＝今回の加熱率　//更新

38

設計の準備として

- プロセス3のDFDのI／O部を深くスケッチする



余白）

組込み向け構造分析・設計の概要（1）

坂本　直史

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2)　実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

アジェンダ

1. 何故、分析を？
2. 構造化分析と構造化設計
3. 話題沸騰ポットにおける構造化分析
4. 構造化設計概要
5. まとめ

アジェンダ

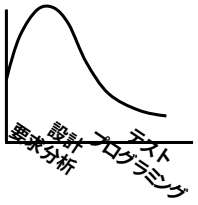
1. 何故、分析を？
2. 構造化分析と構造化設計
3. 話題沸騰ポットにおける構造化分析
4. 構造化設計概要
5. まとめ

仕様書はもらったけれど・・・・

- もらった仕様書通りに作ったけど？
- そんなことどこにも書いてないのに？
- そんな変更の可能性はちゃんと言っておいて欲しい！
- あの人の仕様書は、・・・・

⇒ みんなが経験する。でも、
- 何故そんなことが起こるか？
- どうすれば、被害を最小限に？

仕様書を作る立場から



- 客先から仕様がでてこない
- 仕様を細かく書いている時間がない
- 変更はつきもの。それより開発を進めたい。

⇒
- 昔の自分を思い出そう
  いつか通った道のはず！
- 後工程に影響が大きいから、経験があり
  給料の高い人がやってるはず！

仕様書を読む立場から

- 仕様が少し曖昧でも、わかる範囲でコーディング
- 早く動かしたい
- 仕様変更はいつものこと。指示があったら考える。

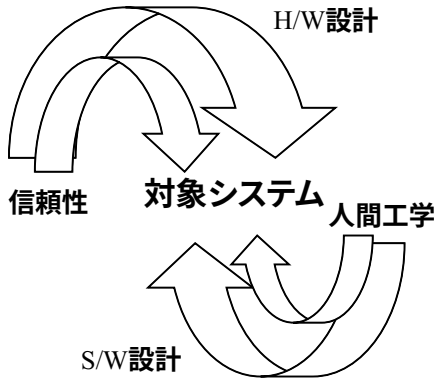
⇒
- 拡張性や保守性を考えておくことが
  自分を守る
- OJTだけでなく新しい技術の取り込むことで
  知見を広くし、一段上のエンジニアへ

要求分析

- 開発対象システムの本質を考察
  仕様の漏れ、抜け、改善点
  → 目指すシステムと
  有るべき姿の共有

- 何をどのようにする？
  → “What”を明確に

- いろいろな視点からの
  分析が必要



アジェンダ

1. 何故、分析を？
2. 構造化分析と構造化設計
3. 話題沸騰ポットにおける構造化分析
4. 構造化設計概要
5. まとめ

構造化分析と構造化設計

- S/W開発の大規模化 → 品質低下 / 保守
  この問題の解決を目指す
  設計手法であり、コミュニケーション手段である

構造化分析

- "What"を明確にする
- ユーザとシステム開発者、システム開発者同士のコミュニケーション向上

⇩

構造化設計

- 構造化分析の結果 → どのように作るか
  "What"から"How"の世界へ
- 分析で掘り起こしたモジュール間の特性を考慮
  保守性の良いプログラム構造

構造化分析

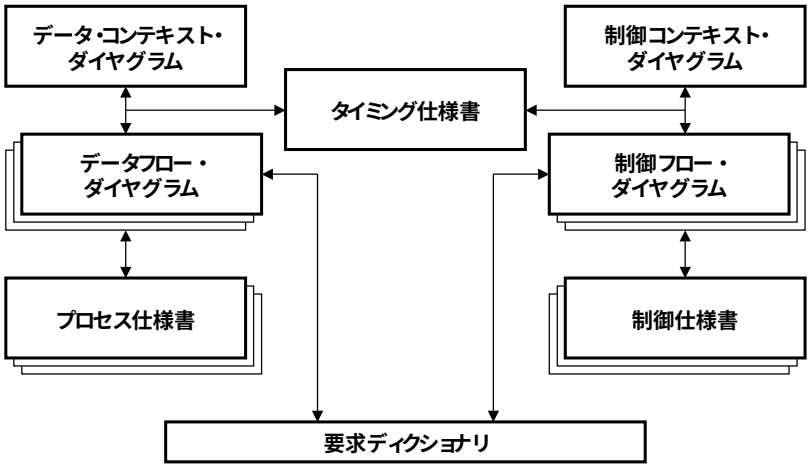
- 基本はDeMarcoに始まる構造化分析
  リアルタイム拡張としてHatleyらによる試み

- 開発手順

イベントリスト
↓
コンテキスト・ダイヤグラム
↓
フロー・ダイヤグラム
↓
プロセス仕様書

データ・ディクショナリ（随時）

Hatley/Pirbhaiによる要求モデルの構成要素

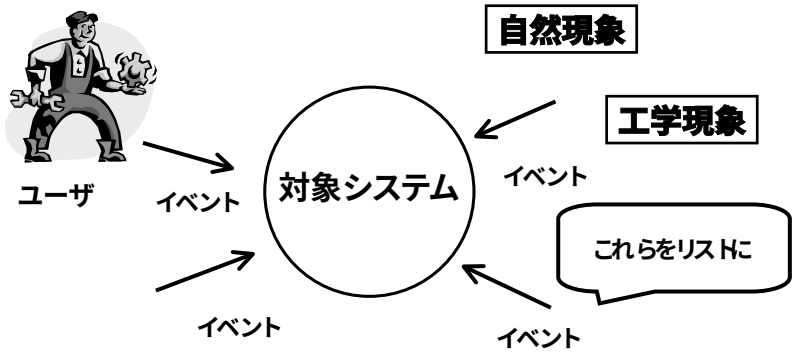


アジェンダ

1. 何故、分析を？
2. 構造化分析と構造化設計
3. 話題沸騰ポットにおける構造化分析
4. 構造化設計概要
5. まとめ

イベントリスト

- 対象システムの外部で発生する様々な事象から システムで対応しようとするものの一覧



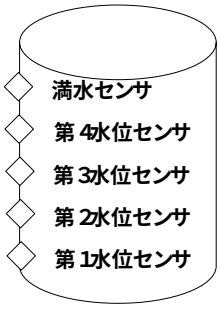
53

話題沸騰ポットのイベントリスト (1)

- 水位の変化をどう捉えるか？

- 水位の変化に関係しそうなイベント
  - 給湯
  - 給水
  - 満水の検出

- イベント候補
  - 水位センサ毎のオン/オフ
  - 給湯や給水
    - 水位の変化は副作用



54

話題沸騰ポットのイベントリスト (2)

- 対象システムの外部で発生する様々な事象から システムで対応しようとするものの一覧

イベント	スティミュラス	アクション	レスポンス
例） 保温モード 指示	保温モード	・保温モードを 設定する ・温度制御を 変更する	操作受付を通知する 温度/モードを表示する

システムが	イベントが	イベントが	イベントが
発生を制御	発生したことを	発生した時に	発生した時の
できない事象	伝える情報入力	果たすべき機能	外部への反応

55

話題沸騰ポットのイベントリスト (3)

- 水位の変化に関連するイベントを考えてみましょう

第n水位センサ
水位を検出。各センサはonの時、その位置よりも水位が高い。

アクション？

満水センサ
水位がこのポットの許容上限を越えていないかどうかを検出。
センサがonの時、水位が許容上限を越えている。

イベント？

レスポンス？

給湯ボタン
ボタンを押すと給湯口から水を排出する。押している間中は
給湯を行う。手を離すと給湯を停止する。

スティミュラス？

水位メータ
ポット内の水位を表示

56

話題沸騰ポットのイベントリスト (4)

•お湯を使いたい　　給湯する
　給湯口を開く　解除ボタン　　給湯可能にする
　給湯口を閉じる　解除ボタン　　断熱する
　湯を注ぐ　　給湯ボタン= “on”　湯が出る
　湯を注ぐ終える　給湯ボタン= “off”　湯が止まる

•ラーメンを食べたい　　調理時間を計る
　3分待とう　　タイマボタン　　最初の調理時間を設定する
　3分では足りない　タイマボタン　　調理時間延長
　やめた　　？？？　　キャンセル

人間の「心の状態」から分析する
→ リアクティブ・オートマトン法
　武蔵工業大学　松本先生

話題沸騰ポットのイベントリスト (5)

•例えば、こんな分析結果が出たとします

用語統一

実装にかかわり “How”

イベント	ステート	アクション	レスポンス
給湯スタート	給湯ボタン押下 水位検出	給湯スタート	給湯口より水排出 水位メータで水位を表示
給湯終了	給湯ボタン離上	給湯停止	給湯口より水排出停止
給水	水位センサの オン→オフ	水位センサの値により、水位メータの表示を更新する	水位メータの上昇

•どうですか？

イベント発生を伝える データ

外部から見える変化はレスポンス

話題沸騰ポットのイベントリスト (6)

- 水位センサ毎のオン/オフを1つのイベントに
  - 仕様変更に追従できない
  - 粒度が細かく問題の本質が把握しにくい
- 給湯や給水をイベントに
  - 給水はポットからは検出しにくい
  - → 給水は根本的には水位の変化
    満水や水温低下は給水に伴う副作用

⇒ 「水位変化」をイベントとする
「満水」を「水位変化」と別イベントとするか？
→ システム上重要なことはイベントとして抽出

話題沸騰ポットのイベントリスト (7)

イベント	スティミュラス	アクション	レスポンス
給湯開始	なし)	ポット内の水を給湯	操作受付を通知する 給湯口から水を排出する
給湯終了	なし)	ポット内の水の給湯を停止する	給湯口からの水の排出を停止する
水位の変化	水位	水温を保温温度にする	水位表示を更新する
満水	なし)	温度制御を停止する	沸騰解除を表示する 保温解除を表示する
水なし	なし)	温度制御を停止する	沸騰解除を表示する 保温解除を表示する

データディクショナリ

- 設計で用いる用語を登録　プログラム言語の予約語
  用語（データ名）は誰にでもわかる言葉を
  設計を通して用語を統一

- 設計の各フェーズでこまめにアップデート

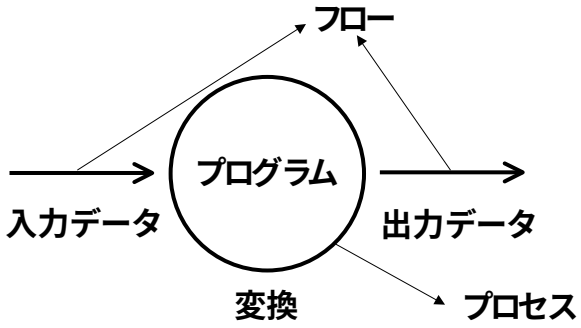
タイマ残り時間＝整数　下限：0　上限：9　単位：分
保温モード　＝[高温モード|節約モード|ミルクモード]
温度異常　　＝[高温異常|温度上がらず異常]
水位　　＝整数　下限：0　上限：4　単位：なし
高温モード　＝沸騰とほとんど同じように使うための保温モード
節約モード　＝消費電力を節約するための保温モード
ミルクモード ＝乳児の粉ミルク調乳用として使うためのモード
高温異常　　＝水温が一定以上に上がった場合
温度上がらず異常　＝水温が上がらなくなった場合

タイミング仕様書

入力	イベント	出力	イベント	対応時間
水温	110度を超える	ブザー音	高温エラー	最大 1秒
	前回検出した水温より低い	ブザー音	ヒータ動作異常	1分

データフロー・ダイヤグラムによる要求分析

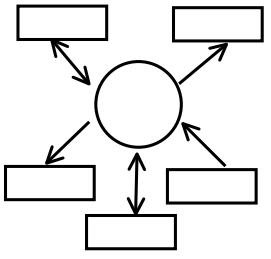
・プログラムをデータの変換と捉える



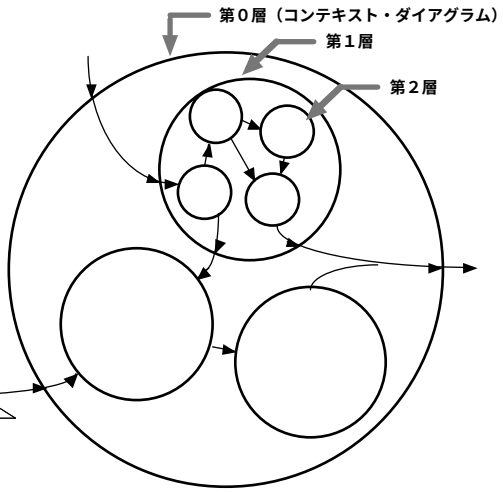
データフロー・ダイヤグラム ＝ DFD

63

分析の過程



コンテキスト・ダイヤグラム

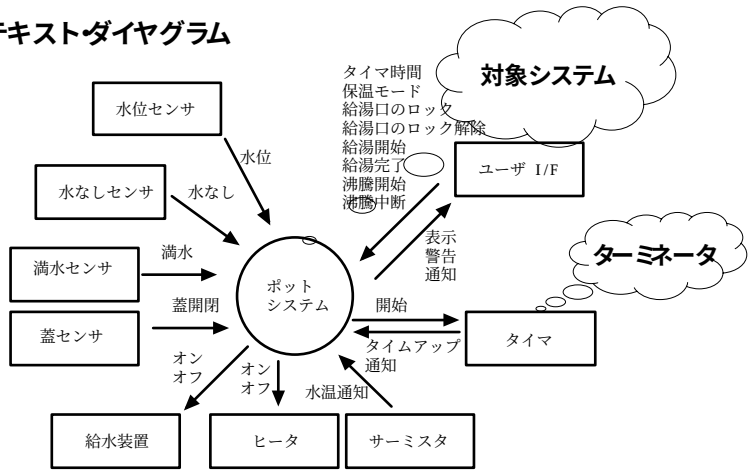


分析スタート!

64

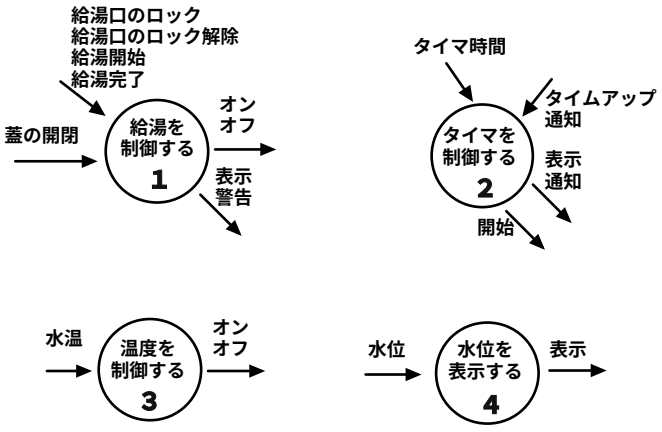
分析の入り口

・コンテキスト・ダイヤグラム



65

分析　ステップ１ ～DFD0～

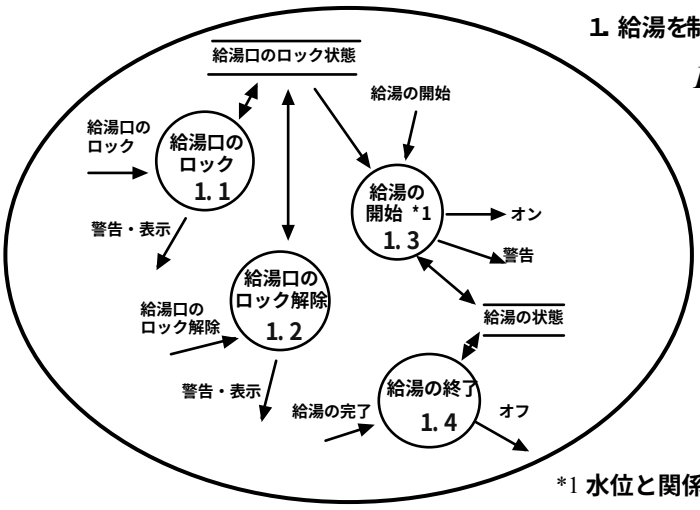


66

DFD作成の基本ルール

- 粒度と曖昧な言葉に注意
- 曖昧な言葉を使わない
  - まとめたもので、名前は少し曖昧でもよい
  - 処理する」制御する」データ」
- DFD0の上位レベルでは、いくつかのプロセスを
- DFD0でプロセスが多すぎると、まとめ直しを
- 7± 2の理論
  - イベントリストの完成度が低い
  - イベントリストにある1つのリストに対して
  - プロセスを作っていないか？

分析　ステップ２ ～DFD1～



DFD作成の指針

- 先ずは書き始める
  書いたものを見て改善
  頭の中でパッとまとまりや苦労しない

- プロセスの兵糧攻め
  プロセスに出入りするフロー数を制限
- プロセスへの均等なフローの割り当て
  整理する→フローの掌握、抽象化
  哲学者
- 名前は自ずから
  感性の世界
  芸術家

DFDを用いた分析

- 分析とは
  かみ砕き、分類し、整理し、抽象的概念で再構築する作業
  芸術家の感性と哲学者の思考
  → 良いソフトウェア開発者に要求される資質と同じ

- DFDを用いた分析
  データの変換に沿っての分析
  → ソフトウェア開発者には馴染みのフィールド

  図形表現でのビジュアル化

  WhatとHowを切り離して設計
  → 巨大化・複雑化するシステムへの対応

制御フロー・ダイヤグラム（CFD）

- 制御フローの流れを規定
  - フロー上をデータが流れればいいのか
  - 条件の組み合わせで活性化されるプロセスが違うのか

  ○上位層のシステムの状態を制御するロジックの記述
  ×プリミティブ・プロセス同士の相互作用の詳細の記述

- でも、制御は控えめに
  - DFDを最大限に、制御は最小限に
  - ×ついつい実装を意識　　モデルを抽象的なものに

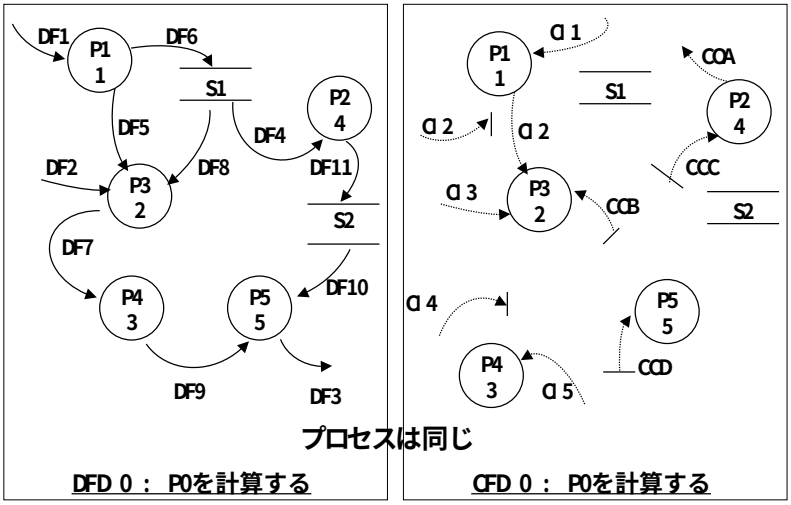
- 1つのDFDに対して制御フローを作成
  - データの流れと制御の流れを分離し明確に

CFD作成指針

- CFD作成順序
  - (1) DFD　　できるだけ
  - (2) 組み合わせコントロール　　仕方なし
  - (3) 順次処理コントロール　　最悪

- 道具立て
  - 組み合わせコントロール
    - デシジョンテーブル（真理値表）
  - 順次処理コントロール
    - 状態遷移図

DFDとCFD

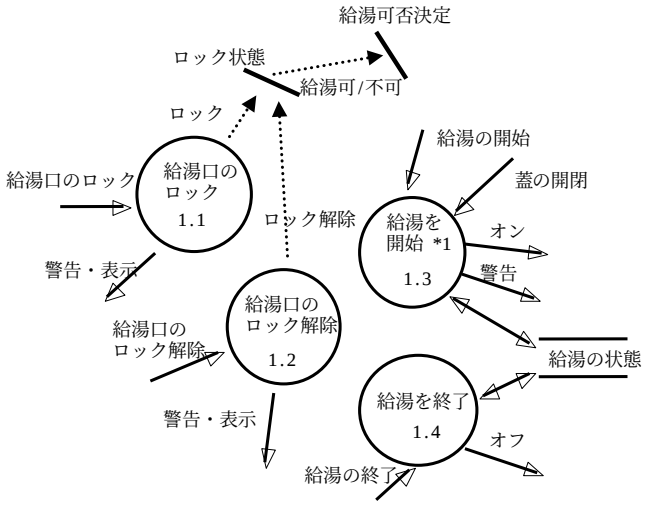


プロセスは同じ

DFD 0 ： P0を計算する　　**CFD 0 ： P0を計算する**

73

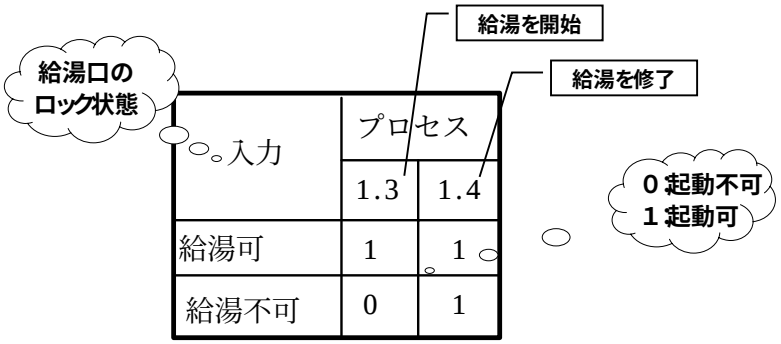
CFD



74

組み合わせコントロール

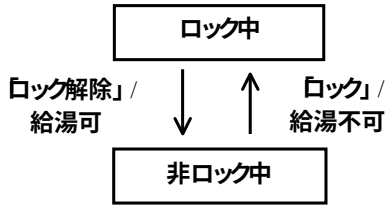
- 入出力データの全てが2値ならブール式
- 入出力データのどれかが多値ならデシジョンテーブル



75

順次処理コントロール

- 状態遷移図、状態遷移テーブル、状態遷移マトリックス
- 状態遷移図　7± 2の理論



- 状態遷移テーブル

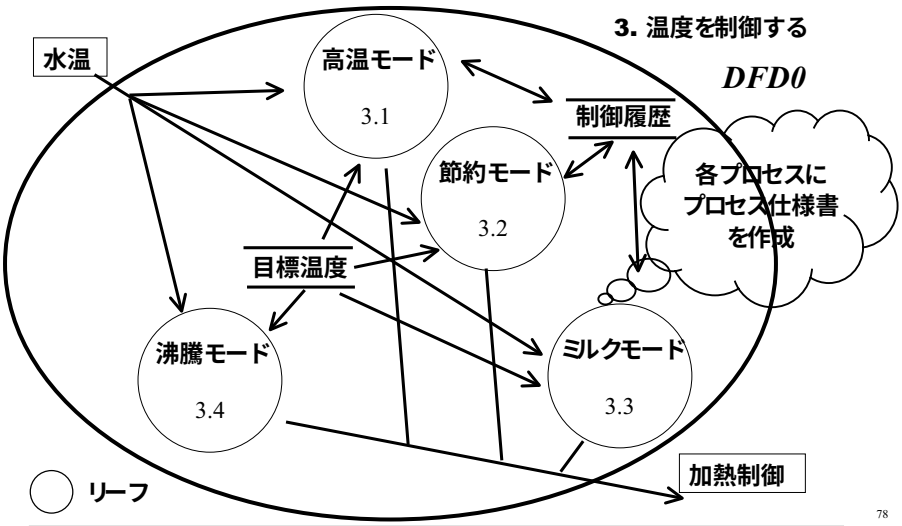
現在の状態	イベント	アクション	次の状態
ロック中	ロック解除	給湯可	非ロック中
非ロック中	ロック	給湯付加	ロック中

76

プロセス仕様書

- リーフのプロセスに対して作成
- 入力から出力への変換がなすべきことを記述
  → まさにデータフロー
- 記述方法は目的にかなっていれば自由
  文章 / 数式 / 表や図
- しかし
  疑似コードは×
  文章形式は構造化言語で
  使用する言葉と構文を決めておき、冗長性を排除）
- 局所的な記述なら、デシジョン・テーブル / 状態遷移図等もOK

温度制御とプロセス仕様書



プロセス仕様書 P3.1　～高温モードでPID制御する～

制御周期ごとに、

1. // 水温履歴を更新する

   $T_2=T_1$　$T_1=T_0$　T_0 = 水温

2. // 目標温度を得る

   T_g を目標温度　モード＝高温モードである、

   目標温度 :目標値に設定する

3. // PIDでの制御量を計算する

   $\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

   今回の加熱率＝前回の加熱率＋ΔM　// 加熱率が100%以上になることは今回考慮しない

4. // ヒータ通電期間を制御する

   制御期間×今回の加熱率の期間中は、

   加熱制御="on"

   この期間が終了するタイミングで

   // ヒータを切る

   加熱制御="off"

   // 加熱率の履歴を更新する

   前回の加熱率＝今回の加熱率　// 更新

プロセス仕様書 P3.4　～沸騰モード～

// 沸騰させてから3分間煮沸する

1. // 沸点まで連続的に加熱する

   加熱制御 = "on"

   水温 ≧ 100℃　となるまで待つ　// 異常の監視はしない

2. 以下を制御周期単位で3分間継続実行する

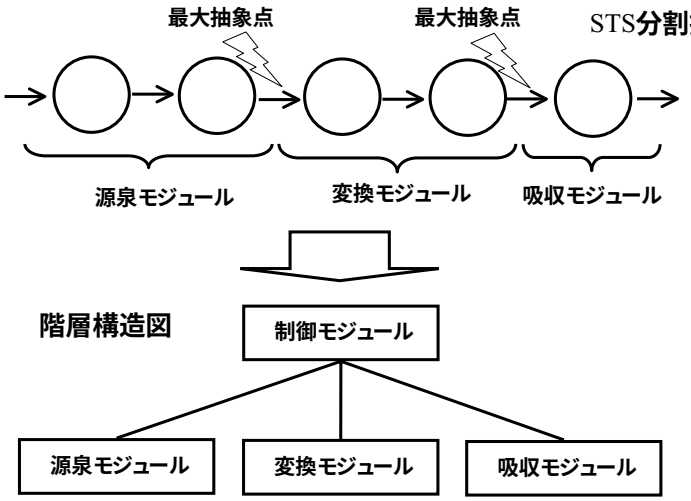
   if　水温 ≧ 100℃　加熱制御 ＝ "off"

   else　加熱制御 ＝ "on"

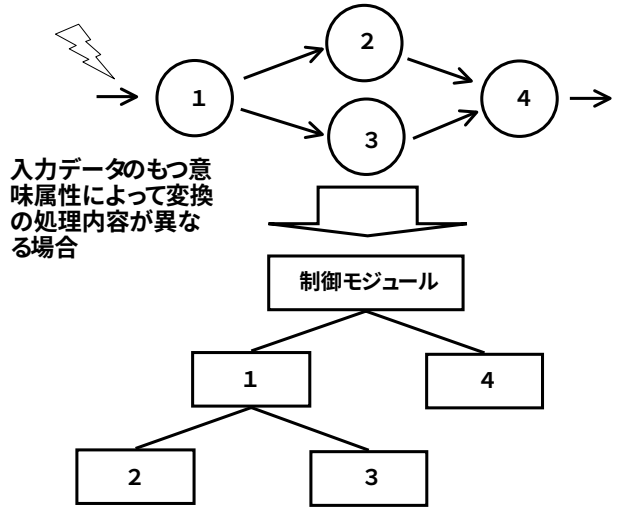
アジェンダ

1. 何故、分析を？
2. 構造化分析と構造化設計
3. 話題沸騰ポットにおける構造化分析
4. 構造化設計概要
5. まとめ

構造化分析から構造化設計へ（1）



構造化分析から構造化設計へ (2)



83

まとめ

- 分析しましょ！

  かみ砕き、分類し、整理し、抽象的概念で再構築する作業

芸術家の感性　　哲学者の思考
→ 良いソフトウェア開発者に要求される資質と同じ

- 手法は何であれ、芸は身を助く！

  分析でプロジェクトに潜む病魔をチャッチ
  → あなたの分析がプロジェクト全体を救う

  一段上のエンジニア目指して、*Let's analyze!*

84

参考

- DeMarco ：“構造化分析とシステム仕様”, 日経ＢＰ社, 1981
- Hatley and Pirbhai ：“リアルタイム・システムの構造化分析”, 日経ＢＰ社, 1989
- リアクティブ・オートマトン法：
  http://www.sft.cs.musashi-tech.ac.jp/~yhm/index.html
- 坂本 ：“ソフトウェア高信頼化のアプローチ”, 第 5回組込みシステム開発技術専門セミナー, ES-6, pp. 37-69 (2002)

余白）

組込み向け構造分析・設計の概要（2）実習/回答と補足説明

二上 貴夫

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

課題 4－ 1.

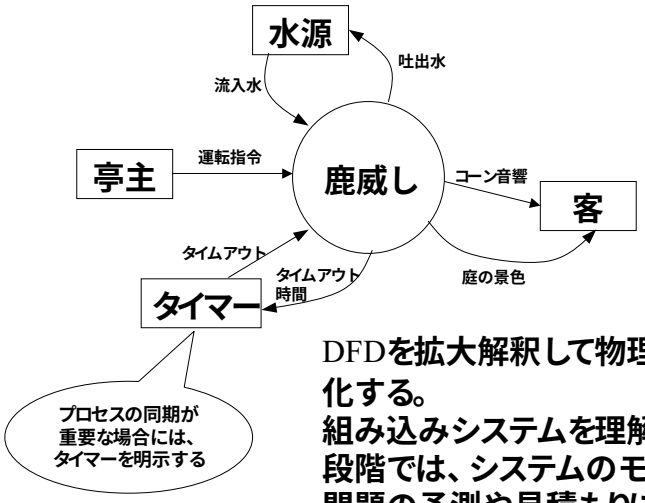
- 鹿威し開発プロジェクトの企画からSozeX001型の仕様までを読んで理解しなさい。
- 001型仕様の範囲で要求レベルのイベントを列挙しなさい。（興味のある人は、要求レベルのコンテキスト図も作成しなさい。）

イベントの列挙例

- SozeXを購入する
- 茶会を開く
- 茶会が終わる
  - 後始末をする
  - 停止することを忘れる

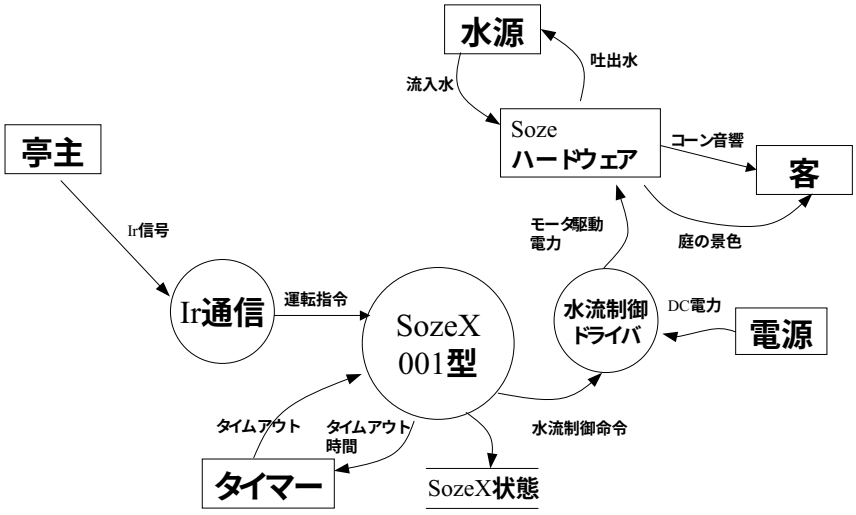
注意 :ここで示しているのはイベントのみ

システムと要求をモデル化した例



DFDを拡大解釈して物理IOもモデル化する。
組み込みシステムを理解する最初の段階では、システムのモデルを作ると問題の予測や見積もりに役立つ

001型__製品仕様の例



SozeX001型　プロセス仕様の例

SozeX状態＝“**待機中**”の場合：

運転指令 :=“運転”が着信したら、

- 水流制御命令＝“On” を出力する
- タイマにタイムアウト時間を設定する
- SozeX状態＝“運転中” として遷移する

SozeX状態＝“**運転中**”の場合：

運転指令 :=“停止”が着信したら

- 水流制御命令＝“Off” を出力する
- タイマをキャンセルする
- SozeXの状態＝“待機中” として遷移する

タイムアウト:=“Yes”が着信したら

- 水流制御命令＝“Off” を出力する
- SozeX状態＝“待機中” として遷移する

辞書__1

運転指令 ＝ ｛"運転"｜"停止"｝

流入水 ＝ 竹筒に流れ込む水のこと

吐出水 ＝ 竹筒が反転した時に出て行く水のこと

庭の景色 ＝ 客が鑑賞できるSozeXを含めた庭の景色

・・・

SozeX状態 ＝ ｛"運転中"｜"待機中"｝
状態遷移図を使わないDFDでの代替状態名

タイムアウト時間 ＝ 180分くらいを想定している
単位 :分 今後の製品展開で変更か？

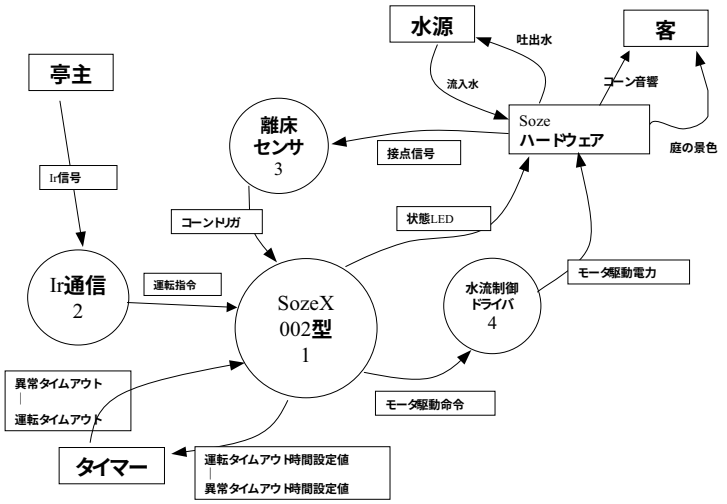
・・・

水流制御命令＝ ｛On"｜"Off"｝ Onで水が流れる

課題 4－ 2.

- SozeX002型の仕様を読んで理解しなさい。
- 製品仕様としてどのような追加が求められているか考えてコンテキスト図に加筆しなさい
- 余裕のある人は、DFD 0のレベルでプロセスを状態モデルで記述しなさい

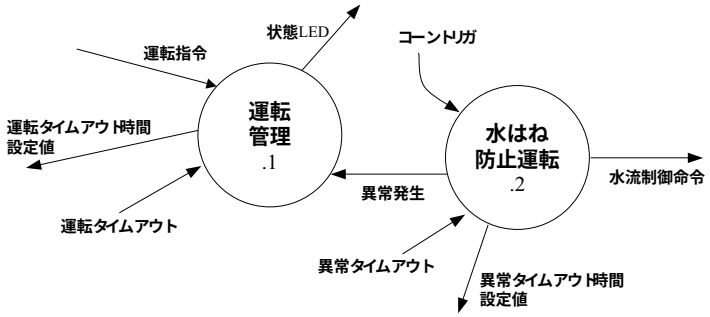
002型__製品仕様をモデル化する



95

SESSAME CONTENTS 2004

002型__DFD0



96

SESSAME CONTENTS 2004

辞書__2

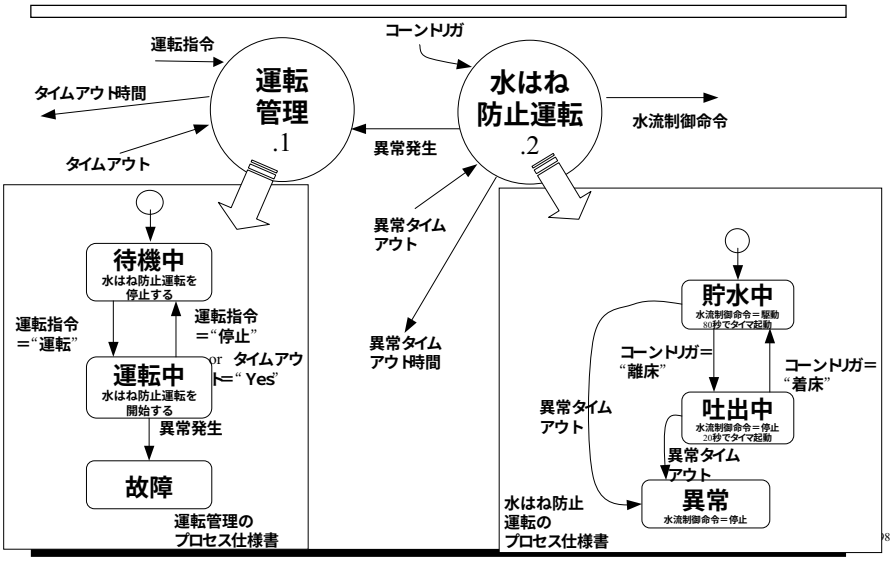
コーントリガ ＝ ｢[離床”｜“着床”]

反射光強度 ＝ 光接点センサの出力強度

接点信号 ＝ . . .

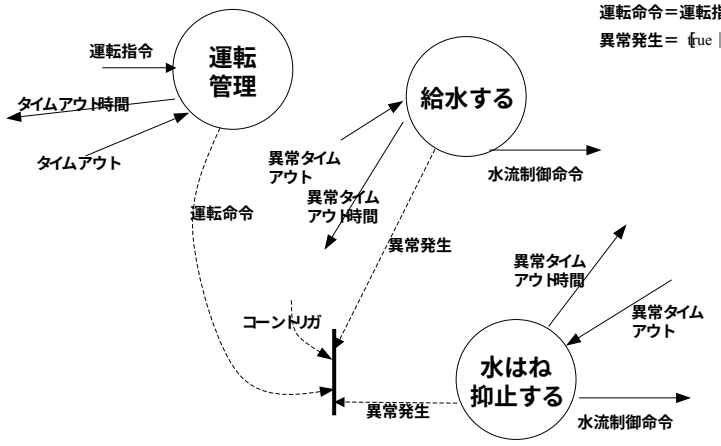
. . .

SozeX 002型のOO風味のデマルコDFD0例



純ハトレー・ピルバイのモデル例

辞書：
運転命令＝運転指令の制御フロー版
異常発生＝［true｜false］



余白）

組込み向け構造化設計（1）

山田 大介

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

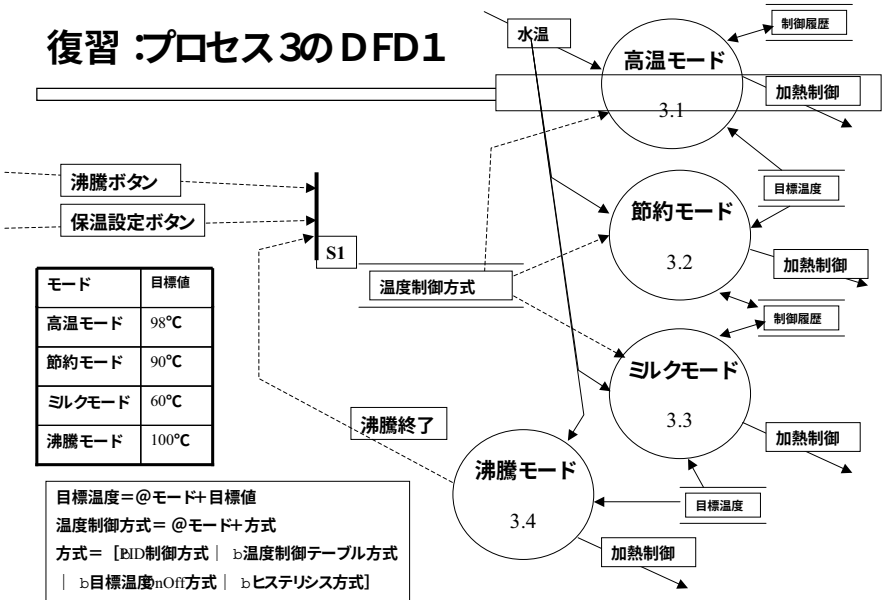
アジェンダ

1. **構造化設計**
   - 分析モデルから設計の構造図へ
2. **設計の品質**
   - モジュール分割、凝集度と結合度
3. **アーキテクチャ設計**
   - 非機能要件の設計
4. **モジュール仕様**
   - 構造図からモジュール仕様へ
5. **仕様変更への対応**
   - 長持ちする設計

アジェンダ

1. **構造化設計**
2. **設計の品質**
3. **アーキテクチャ設計**
4. **モジュール仕様**
5. **仕様変更への対応**

復習 :プロセス3のＤＦＤ1



モード	目標値
高温モード	98℃
節約モード	90℃
ミルクモード	60℃
沸騰モード	100℃

目標温度＝@モード+目標値
温度制御方式＝ @モード+方式
方式＝［PID制御方式｜ b温度制御テーブル方式
｜ b目標温度nOff方式｜ bヒステリシス方式］
制御履歴＝前回の加熱率＋T2+T1+T0

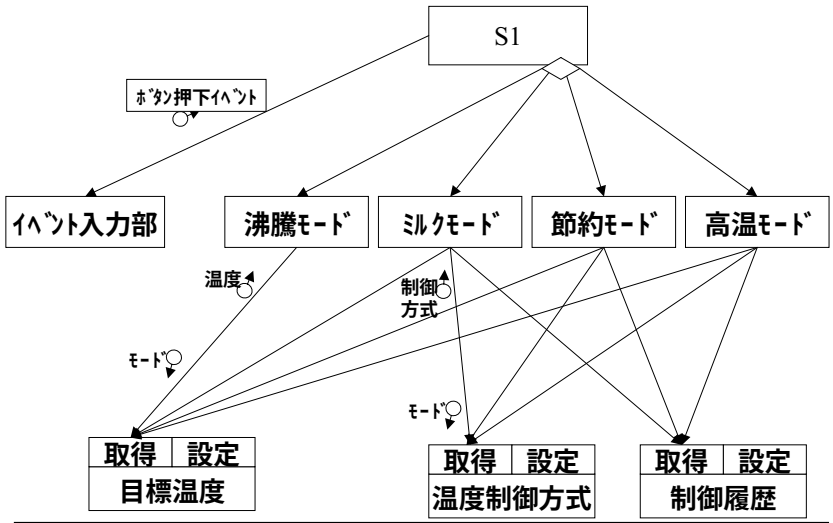
構造化設計の手順

- 構造化分析のDFDに基づき、
- 最大抽象点を見つけて、源泉－変換－吸収スタイルを作る
  - その際に、ボスとして制御モジュールを作る
- 制御バーをモジュールに置き換える
  - ディスパッチャになる
- その他のバブルをモジュールに置き換える
  - そのバブルの下に、下の階層のバブルがぶら下がっていく
- 外部データの入力モジュールを追加する
- データの入出力を考える
  - データのアクセスにはデータ隠蔽モジュール

次に示す2つの案が簡単に設計できる

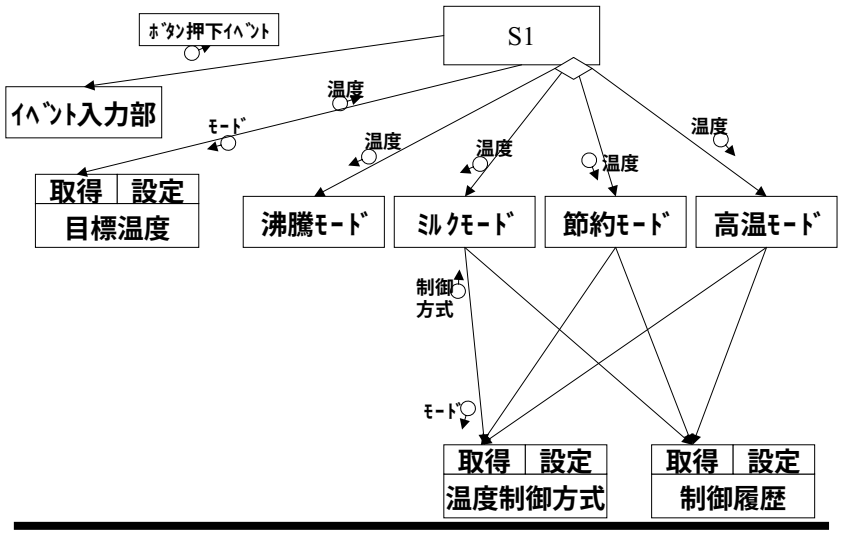
105

案１



106

案２



107

構造図の要点

- 構造図は、分析時のDFDからある程度機械的に作成できる
  - 最大抽象点を見つけることにテクニックは必要
    - 最も本質的な形で入力を表現しているデータフローと、最も本質的な形で出力を表現しているデータフローで囲まれた部分
- あとは、次に示す設計基準（凝集度と結合度）により、モデルの選択と、若干の構造見直しを行う

108

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

設計の品質

- モジュール分割
- モジュールの凝集度（コヒージョン）
- モジュール間の結合度（カップリング）

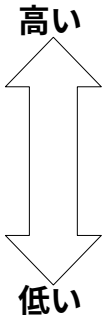
モジュール分割

- モジュールサイズの適正化
  - 1ページの半分程度（50行）
  - 実装が単純になる
- システムの明解さ
  - 均衡型システム（左右均衡、上下は論理－物理）
  - モジュールの関係だけでシステムが理解できる
- 重複の極小化
  - 同じ機能を複数のモジュールに持たせない
- 情報隠蔽
  - モジュールで取り扱うデータを隠蔽する

凝集度（コヒージョン）

- 凝集度は「高い」ほど保守性が良い

1. 機能的
2. 逐次的
3. 通信的
4. 手順的
5. 一時的
6. 論理的
7. 偶発的



良い凝集度

1. 機能的
   - 単一の目的を持った機能、情報、責務。
2. 逐次的
   - 同一のデータに対する、
   - 関連が強く必然的な順番を有す機能集合。
3. 通信的
   - 同一のデータに対する、
   - 複数目的の機能集合。

以上が、凝集度が高く保守しやすいモジュール

中程度の凝集度～悪い凝集度

4. 手順的
   - 関連の薄い複数データに対する、順次機能。
5. 一時的
   - 同一時間に発生する、互いに関係のない機能集合。
6. 論理的
   - 外部から利用する時に便利な機能の寄せ集め。
7. 偶発的
   - まったく相関のない機能の寄せ集め。
   - 実装制約（HOW項目）を意識してしまうと、、、

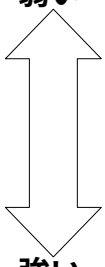
時の経過とともに、

モジュールの存在理由が曖昧になっていく

結合度（カップリング）

- 結合度は「弱い」ほど保守性が良い

1. データ結合
2. 構造体結合
3. バンドリング結合
4. 制御結合
5. ハイブリッド結合
6. 共通結合
7. 内容結合



良い結合度

1. データ結合
   - 構造を持たない単一のデータ。
2. 構造体結合
   - 同一の目的を持つデータメンバの集合。
3. バンドリング結合
   - 複数のフィールドから構成される構造体。
   - 使い方自由の万能構造体は良くない。

以上が、結合度が弱く、
理解しやすく、波及作用が限定的なモジュール間結合

中程度の結合度～悪い結合度

4. 制御結合
   - 相手のモジュールを制御するフラグ。
5. ハイブリッド結合
   - 値の範囲により意味の異なるデータやフラグ。
6. 共通結合
   - グローバルデータ。
7. 内容結合
   - 相手のモジュールの内部を参照。（アセンブラ）

変更すると、思わぬところに副作用が、、、

分析の効用

分析を行うことにより、

- 要求の曖昧個所や矛盾点に気づくだけでなく
- ある程度機械的に設計が可能となる
- さらに、「凝集度」が高く、「結合度」が弱い、という高品質な設計も手に入る

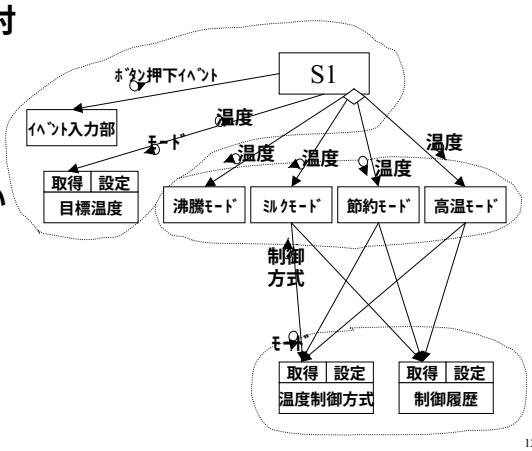
上流工程で品質を作り込もう

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

アーキテクチャ設計

- システム外部からのイベント
  - 制御スレッドの検討
  - 割り込みの利用
- 並行処理
  - パッケージ分割
    - 構造図で破線囲い
  - サブシステム分割
    - 人員配置にも



非機能要件

- 時間的制約
  - 応答時間
    - ハードリアルタイム／ソフトリアルタイム
- リソースの制約
  - ROM／RAM
  - CPUパワー
- 環境上の制約
  - 開発環境、工場検査、市場メンテナンス
- フォールトトレランス
  - 過負荷、フェイルセーフ

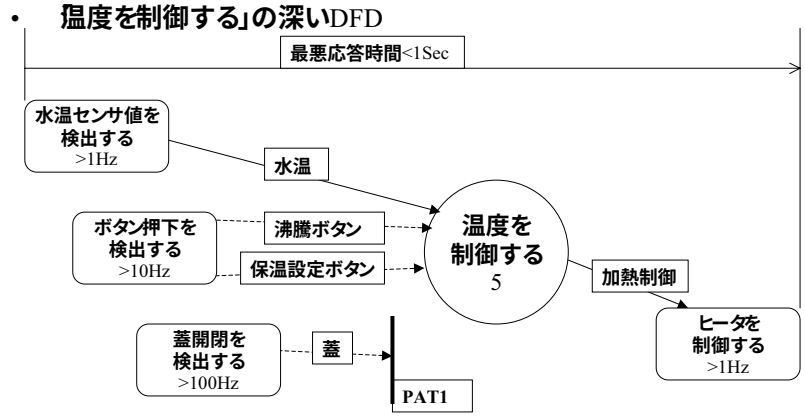
非機能要件 :ポットの例

• レートモノトニック要求		
– ヒータのOnOff制御（PID制御は、1秒毎）	精度±20mSec	ラインクロック
• イベントドリブン要求		
– 蓋の開閉検出	必須	ハード割り込み
– 温度異常検出	不用	
– 満水と水なし検出	不用	
• データ構造要求		
– 温度履歴の保持方法		3点までのリニアバッファ
• 復帰のメカニズム		invocationで対応
• 製品対応要求		
– ROM実装量を最小化＋保守容易化		SCM*環境で開発
• エラー処理メカニズム要求		
– エントリ引数チェックとトラップメカニズム		コーディングルールで対応

*SCMとは、ソフトウエア構成管理

非機能要件 ポットの例

- 「温度を制御する」の深いDFD



蓋の開閉を割り込み処理

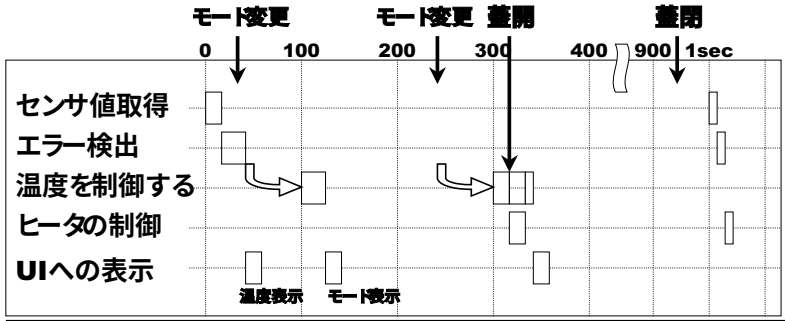
ボタン押下と水温検出を周期制御

123

SESSAME CONTENTS 2004

タイミングと並行性の検討

- センサ値取得が、1sec周期
- ボタン押下（モード変更）の検出が、100msec周期
- 蓋の開閉は、割り込み処理



124

SESSAME CONTENTS 2004

タイミングと並行性の検討

- タイムリー性
  - ワーストケースを積み上げてみる
  - ハードリアルタイム　　ハードウエア処理の遅延がないか？
  - ソフトリアルタイム　　ユーザへの応答性は大丈夫か？
- 並行性
  - 処理時間がかかるもの、ウエイトが発生するもの、の抽出
  - スケジューリング方式を検討
    - イベントドリブン、レートモノトニック
  - 共有するリソースやデータが存在するか？
    - 何らかの排他処理

ここあたりに経験に基づく暗黙の知識が？

- スケジューリング方式の選定
- パフォーマンスチューニング
- ROMサイズのオプティマイズ
- RAMのセクション設定、スタック量の見積もり
- フォールトトレランス

など？

皆さんが普段留意している事項を書きとめてください。
可能であれば失敗事例もお願いします。

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

モジュール仕様

- モジュール単位のインターフェイス仕様と機能仕様
- インターフェイス仕様
  - そのモジュールが「何をすべきか」をインターフェイス仕様で定義
  - インターフェイス契約
- 機能仕様
  - そのモジュールが「どのように実現するか」を擬似コードで定義
  - 分析時のP-SPECを利用可能

モジュール仕様一覧の例

- 割り込み処理
  - 蓋開で、ヒーター制御を停止
- 制御クロック算出
  - ラインクロックでの制御周期計算
- 「温度を制御する」のディスパッチャ
- 「温度を制御する」のイベント入力部
- 「温度を制御する」の高温モード
- 「温度を制御する」のミルクモード
- 「温度を制御する」の沸騰モード

モジュールのインターフェイス仕様の例

名　称：「温度を制御する」の高温モード
目　的：高温で保温するときの温度制御
引　数：なし
戻り値：なし

モジュールの機能仕様の例

- **擬似コード**
  - P-Specより抜粋

制御周期ごとに、

1. //水温履歴を更新する

$T_2=T_1$

$T_1=T_0$

$T_0=$ 水温

2. //目標温度を得る

Tgを目標温度.モード=高温モードである
目標温度.目標値に設定する

3. //PIDでの制御量を計算する

$\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

4. //ヒータ通電期間を制御する

制御期間×今回の加熱率　の期間中は、

加熱制御="on"

この期間が終了するタイミングで

ここにはちょっとした罠が、、、

加熱制御="off"

//加熱率の履歴を更新する

前回の加熱率＝今回の加熱率　//更新

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

仕様変更 :ポットの例、例えば、

- ミルクモードのときは、「62度を上限値、58度を下限値」という仕様変更
- いきなりプログラミングした場合、
  - プログラムの流れを順を追って読み、
  - ソースコード上で変更個所を特定する
  - スパゲッティになってしまっている場合は、
  - 変更箇所の特定すらできないことも
  - また変更による副作用も恐い
- 設計図がある場合、
  - 構造図から変更モジュールが特定できる
  - 外部のインターフェイスを変えないように、内部の処理を変えよう

62度を超えたぞ、
ミルクモードだから、
一度沸騰させているし、
ヒータのエラーにもなっていないから、
ここでヒータを停止させよう。
でもこの手前の分岐は何かな？？
今ヒータ止めてもいいのかな？？

⇨ **変更箇所の特定は素早く、修正は局所的**

仕様変更への対応

- プログラムの流れを追って修正（増築）
  - ソースコードを見ないと修正個所が特定できない
  - やがてスパゲッティ化
  - ソースコードだけが信頼できる

老舗温泉旅館

- 設計図から修正個所を特定
  - 修正個所が局所化される
  - 制御された修正

工学的建築
（基礎がしっかり）

仕様変更への対応

- 長持ちする設計」が良い設計
  - 固定変動分離
    - 外部へのインターフェースを固定
    - 内部処理は変動容易
    - 固定部から変動部を呼び出して、変動部だけを修正
- 設計図でレビューしましょう
  - ソースコードを追わないと分からない
    - 作った人しか分からない
    - さらに、副作用に関しては未知
  - 抽象度の高い設計図でのレビューが効果的

まとめ

1. 構造化設計
   - 分析さえしっかりすれば設計は簡単。分析しましょう！
2. 設計の品質
   - 高い凝集度と弱い結合度で、保守しやすいソフトウエアへ！
3. アーキテクチャ設計
   - いろいろ暗黙の知識がありそうですね。
   - 失敗事例などの知識を共有化する仕組みを作りましょう！
4. モジュール仕様
   - 構造図さえあればここも簡単。但し、状態やモードに注意。
5. 仕様変更への対応
   - 長持ちする設計」を目指し、設計図でレビューしましょう！

以上が、設計でした。
次はもちろん（やっと？）プログラミングです。

その設計は、変更に耐えられますか？

参考

- Meilir Page-Jones：“構造化システム設計への実践的ガイド”，近代科学社，1991

組込み向け構造化設計（2）
実習/回答と補足説明

二上 貴夫

設計

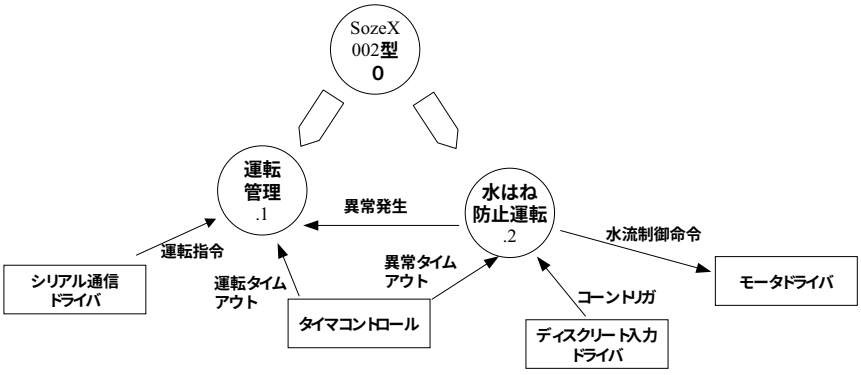
1. DFDモデルの中での並行性を数える（スレッド数）
2. 主要なデータ構造を考える
3. 個々のスレッドに対して実装の仕組みを割り振る
   （プロセッサ、タスク、割り込み、ポーリング関数）
4. 共通アクセスデータについて一貫性を検討する
5. 個々のスレッドに対して構造化設計を実施する

課題6－１

- 002型仕様のコンテキスト図からスレッドを考えなさい。
- 各スレッドについてストラクチャーを考えなさい。

SozeX002設計　スレッドの例

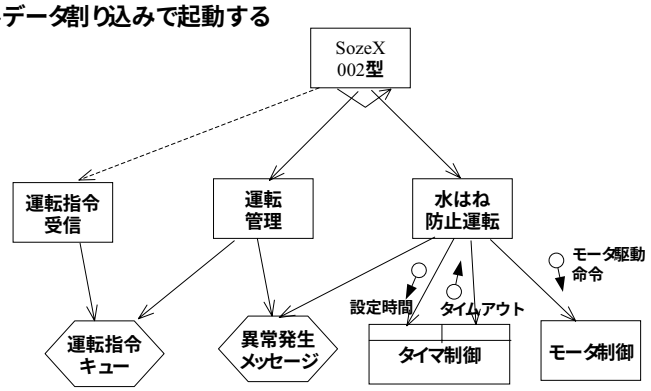
- スレッド
  - 亭主からの運転開始／停止要求
  - 水流の開始／停止



S002ホスト設計例

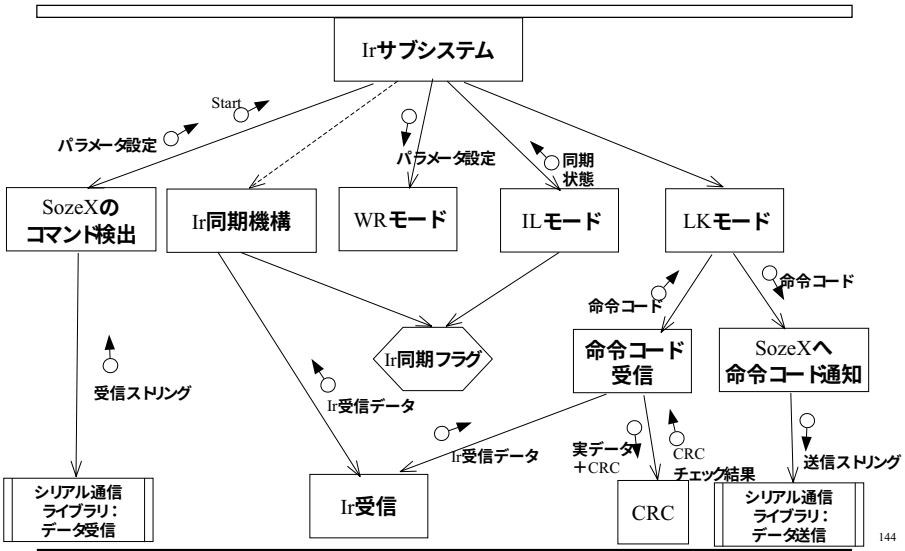
・スレッド2本をCコードのイテレーションで扱う

・イテレーションで運転指令をロストしないように運転指令キューを置き、運転指令受信ドライバをシリアルデータ割り込みで起動する



143

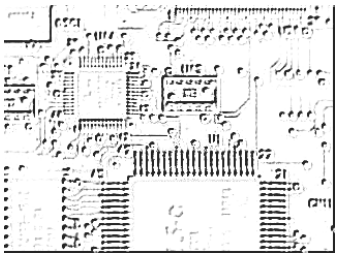
Irサブシステム設計例



144

プログラミング　組込み用語基礎知識

三浦　元



１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2)　実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

おしながき

1. プログラムはどのように動くか
   – ランタイム構造とスタック
2. 配列の実態
3. スタートアップ – main() 以前
4. 周辺デバイス
   – 周辺デバイスへのアクセス, ラッチとゲート
5. エッジ・トリガとレベル・センス
6. 組み込みプログラムの構造
   – ポーリングと割込み, 同期呼出しと非同期呼出し
7. volatile – 変数と最適化
8. ハードウェアとのお付き合い

概論 – 組込みソフトとビジネスソフトの違い？

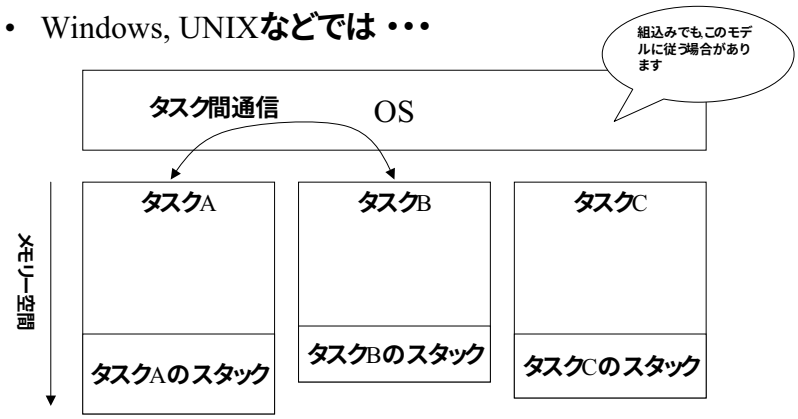
- ハードウェアと密接に関わる
  - ハードウェアの、またはハードウェアを介した事象・状態の読み取り・取得
  - ハードウェアの制御（レスポンス、アクション）
  - メモリーフットプリントや消費電力など、ハードウェアを意識したプログラミングが要求されることが多い
- リアルタイムプログラムが多い
- OSを使わないことが多い
  - OSを使う場合でも、プログラムの実行は特有の手続きが必要
- 稼働環境（物理環境）がいろいろ
  - 実行論理環境は比較的安定
- 品質・信頼性に、より高度なものが求められる
- 出荷したらめったにアップデートできない

1. プログラムはどのように動くか

- スタックとのおつきあい
- 関数呼び出しとスタック
  - スタックが保持する情報
  - 関数呼び出しの深さとスタック
  - スタックのオーバーフロー

プログラムのランタイム構造

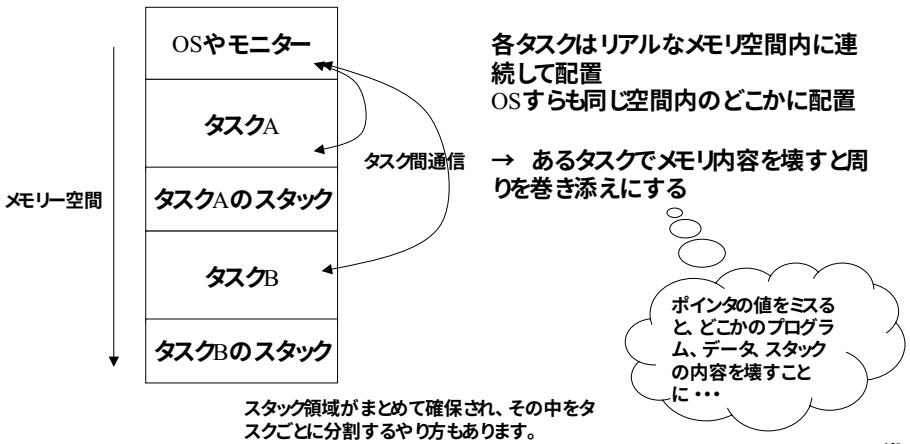
- Windows, UNIXなどでは・・・



各タスクは独立した仮想メモリ空間内で動作
→ 個別のタスクでメモリ内容を壊しても、そのタスクが死ぬだけ

プログラムのランタイム構造

- 多くのリアルタイムOSでは・・・



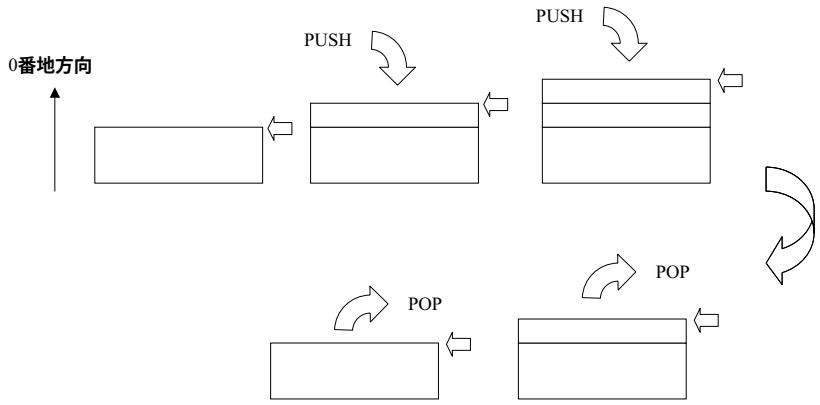
各タスクはリアルなメモリ空間内に連続して配置
OSすらも同じ空間内のどこかに配置

→ あるタスクでメモリ内容を壊すと周りを巻き添えにする

ポインタの値をミスると、どこかのプログラム、データ、スタックの内容を壊すことに・・・

スタック領域がまとめて確保され、その中をタスクごとに分割するやり方もあります。

スタックの使い方



スタックに積まれた途中のデータはPOPできない。
スタックの先頭位置（⇦）は、スタックポインタで常に管理されている。

151

スタックの使われ方 (1)

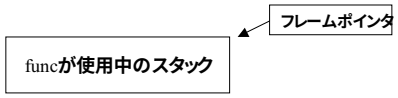
1.

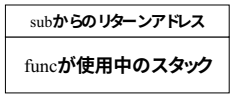
```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

2.

```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

引数をスタックに積む場合もあります



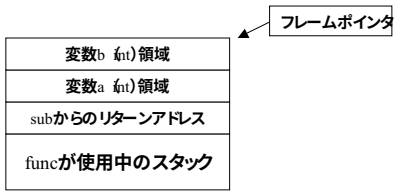


152

スタックの使われ方 (2)

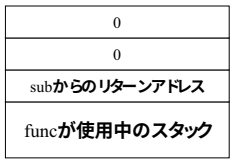
3.

```
int func()
{
    int k;
    k = sub( 2 );
}
⇨ int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```



4.

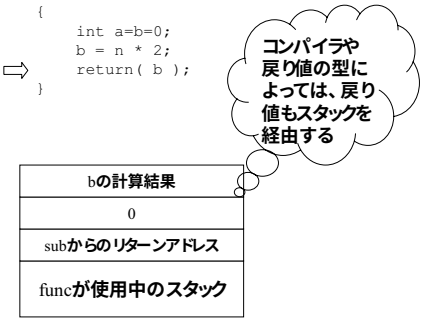
```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
⇨   b = n * 2;
    return( b );
}
```



スタックの使われ方 (3)

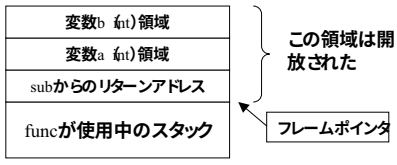
5.

```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
⇨   return( b );
}
```



6.

```
int func()
{
    int k;
    k = sub( 2 );
⇨ }
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

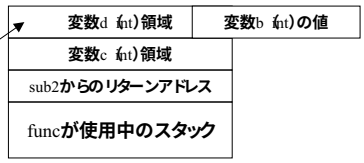


スタックが原因の トラブル (1)

```
int func()
{
    int i, j;
    i = sub( 2 );
    j = sub2( i );
}
int sub( int n )
{
    int a,b;
    b = n * 2;
    return( b );
}
int sub2( int m )
{
    int c, d;
    c = m + d;
    return( c );
}
```

計算結果がおかしい？？

sub() が使っていたスタック領域
続いてsub2()が使用

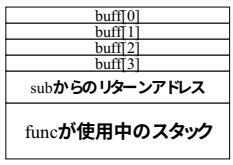


初期化していない

スタックが原因の トラブル (2)

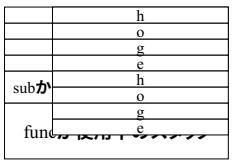
1.

```
int func()
{
    sub( "hogehoge" );
}
int sub( char c[] )
{
    char buff[4];

    strcpy(buff, c);

    return 0;
}
```



2.

```
int func()
{
    sub( "hogehoge" );
}
int sub( char c[] )
{
    char buff[4];

    strcpy(buff, c);

    return 0;
}
```



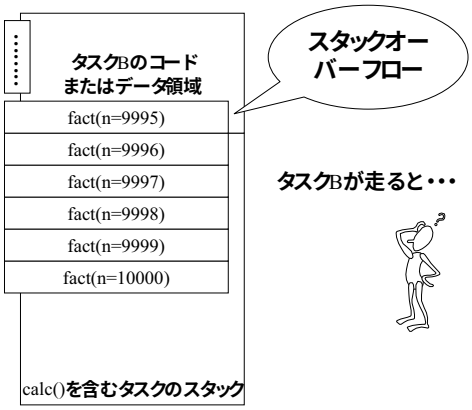
3.

次の瞬間の行き先は・・・

バッファオーバーフロー

スタックが原因のトラブル (3)

```
int calc()
{
    fact( 10000 );
}
int fact( int n )
{
    if ( n == 1 )
        return 1;
    return( fact( n-1 ) * n );
}
```



スタックに関する定石

- スタック変数をはじめとした変数の初期化を忘れない
- バッファへの書き込みは必ずサイズチェックを行い、バッファサイズを超えて書き込みを行わないようにプログラムで制御する
- タスクごとの最大スタック使用量を見積もり、資源の枠内で余裕を持ってスタックサイズを設定する

格言

挙動不審なプログラムの裏にスタックあり

159

2. 配列の実態

- **配列はメモリー上でどうなっている？**
- **intの配列の場合**

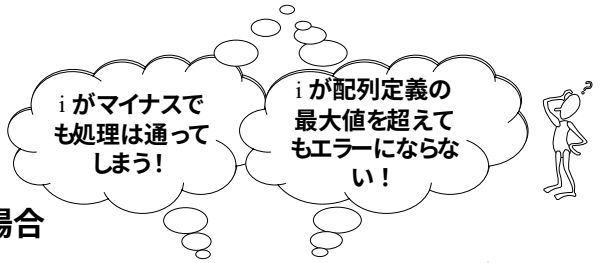
int iarray[4];

0
1
2
3

intの語長

iarray[i]へのアクセスは、Cコンパイラは次のように扱う・・・

(int)*(&iarray[0] + i * sizeof(int))



- **structの配列の場合**

```
typedef struct {
    int i1, i2;
    long lo;
} st_def;

st_def   sarray[10];
```

sarray[i]へのアクセスは、Cコンパイラは次のように扱う・・・

(st_def)*(&sarray[0] + i * sizeof(st_def))

160

つまり配列とは・・・

- 配列と同じ型情報をもった、配列要素個々の先頭のアドレスを指し示すポインタ の集まりである。

そして

- コンパイラは、ランタイムにポインタの指し示す先の正当性 ・妥当性の検証までは面倒を見てくれない。

つまり

- インデックスの正当性 ・妥当性の検証はプログラムロジックで記述しなければならない

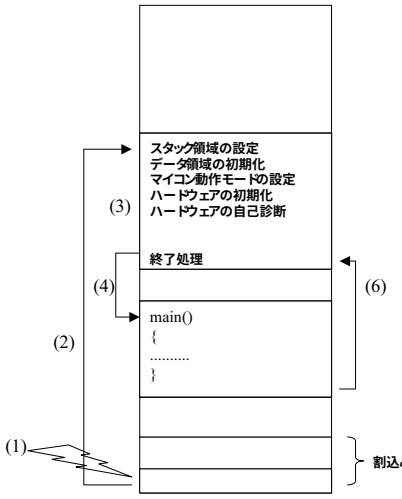
格言

配列のインデックスには悪魔が潜む

3. スタートアップ – main()以前

- 実行の前に：
  - 私の前に道は無い、私の後に道ができる
    - 誰が道を整備するのか？
    - 誰が処理をmain()の玄関口まで導くのか？
  - 環境を整えなければ動けない・・・・
    - ハードウェア初期化
      - 周辺デバイスの初期化、自己診断指示
      - メモリー初期化
    - データ領域の初期化
      - スタック領域の確保
    - マイコンの動作モード設定

プログラム実行の流れ



(1) リセットによる強制割り込みが入る

(2) 割り込みベクトル領域に書かれた番地にジャンプする

(3) スタートアップルーチンの処理を実行する

(4) main() を呼び出す

(5) アプリケーションプログラムを実行する

(6) アプリケーションプログラムを終了する

もう少し詳しく見てみよう

- メモリマップ
  - プログラムとデータをメモリ中でどのように配置するかを設計する
    - プログラム
      - プログラムはROMまたはFlash
        - » ROM/Flash上のコードを直接実行するか
        - » 一旦RAMに展開してから実行するか
      - ハードディスクにある場合はRAMに展開
    - データ
      - 定数はROMまたはFlash
        - » ROM/Flash上の値を直接参照するか
        - » 一旦RAMに展開してから参照するか
      - static, globalなどの変数はRAM
        - » ROM/Flash上の値をRAMに展開する必要がある
        - » 変数領域の初期化

変数はどこにある？

- 自動変数はスタック
- global,static宣言の変数はRAM
- register宣言の変数はCPUレジスタ
  - コンパイラの最適化機能により、特に宣言していなくてもregisterに割り付けられることも。
  - むやみにregister宣言をしても、資源は限られている。
    - registerが使えなければ、スタック変数に振り替えられる
      宣言したのにパフォーマンスがあがらないけれど・・・・・？）

プログラムの配置？

- どこに配置するかを設計時に明確にする
  - ROM領域　不揮発性だがアクセスが遅い
  - Flash領域　書き換え可能、不揮発性だが書き換え回数に制限がありアクセスが遅い
  - DRAM領域　大容量 書き換え可能 そこそこ高速だが揮発性
  - SRAM領域　高速だが揮発性、コスト高
    - バッテリーバックアップが付くと不揮発領域にできる
    - ここを使用する場合には、スタートアップの中でコードコピーを記述する必要がある
- インスタンス生成のコスト
  - インスタンスを生成する＝ROMからRAM領域にコードをコピーし初期化すること
    - CPU資源を使用する、時間がかかる、コンストラクトデストラクトを繰り返すとメモリーが虫食いになる・・・
  - スタートアップの中で、必要なインスタンスを生成しておくことも。

ちょっと休憩 - ROM　SESSAME組込み用語集 http://www.sessame.jp/ より

ROMは読出し専用のメモリで、書き込まれた内容は電源を切った状態でも保存されます。読出しは通常CPU等のバスマスタからリードサイクルで行うことができますが、書込みや消去はライトサイクルでは行うことができません 正確には全く出来ないもの、数ミリ秒の時間が必要なもの、特別な電圧が必要なもの、手順が複雑なもの等があります)。ROMにはEPROM、ワンタイムプログラマブルROM、EEPROM，フラッシュメモリ、マスクROM等の種類があり、それぞれの特徴に適したところに使用します。EPROM、EEPROM，フラッシュメモリは内容を消去して再書込みができますが、書込み回数には制限があり種類によってその保証回数は数百～数十万回とさまざまです。また、書き込まれた内容の保証期間も十数年～百年程度とメーカによりばらつきがありますので長期間使用する用途には注意が必要です。

EPROM

- EPROMは紫外線消去可能なROMで UV-EPROMと書かれることもあります。EPROMには石英ガラスの透明な窓がついていて、ここに紫外線を照射することでメモリの内容を消去することができるようになっています。書込み後はこの窓に遮光ラベルを貼って書いた内容を保護することが必要です。EPROMへの書込みにはROMライタ(ROM プログラマー)と呼ばれる専用ツールがよく使用されます。EPROMには書き込み回数の上限はありますが、実際には紫外線消去の手間が発生するので、その時間的な制約で書き込み制限回数を超えて使用されることはほとんど考えられません。もっとも、消去／再書き込みをするときには普通機械的にソケットから抜き差しするのでその頻度が多いとICのピン磨耗により破損する可能性が大きいと考えられます ソケットも通常多くても数十回程度の抜き差ししか保証されていませんので注意が必要です)。EPROM部品自体には石英ガラスが必要なのでその分コストアップになり、また実装には通常ソケットが使用されるのでそこにもコストがかかります。しかし バージョンアップ等のROM交換が現場でできることやEPROM自体が再利用できるメリットがあるので以前はよく使われていました。現在ではフラッシュメモリが代わりに多く使用され需要を伸ばしています。

ちょっと休憩- ROM (2)

ワンタイムプログラマブルROM(OTPROM)

- EPROMから石英ガラスの窓を無くした物で、1回だけ書込みが可能なROMです。それを除けばEPROMとほぼ同様です。石英ガラスの窓がない分EPROMよりコストが下がっています。ROMによっては同じ形状で窓付きのものと窓無し（ワンタイム）の両方発売されている場合と窓無し（ワンタイム）しか発売されてない場合があります。小規模のワンチップマイコンでコストや実装面積が優先されるような場合窓無しタイプしかない製品があります。窓無し（ワンタイム）しかないものは開発段階ではある程度使い捨てとなりますが、開発・生産・保守のライフサイクルを通じて書き換えの割合が少ない場合に使用されています。

EEPROM

- 電気的消去可能なROMで、実装された状態で読み書きが可能なメモリです。書込みはバイト単位（容量の大きなものではページと呼ばれる連続した領域）で行うことができて、CPUからのライトサイクルやコマンドで書き込みを行った後、書込みが完了するまでに数ミリ秒の時間を待つ必要があります（書き込み中は他の場所にデータを書くことはできません）。尚、消去は書込みの際に自動的に行われるので特に意識する必要はありません。EEPROMに書き込まれる内容は組み込まれる装置の動作に必要なパラメータ等、ユーザが電源を切っても保存しなければならないデータの格納に主として用いられます。書き込み回数に制限があるので（数十万回程度）、書き換えが多い用途には同じアドレスにメーカの指定する制限回数以上書込みをしないようなアルゴリズムが必要です。EEPROMにはシリアルEEPROMと呼ばれる種類があります。シリアルEEPROMにはアドレスバス・データバスといった信号は無く、数本の信号線（クロック、シリアルデータ等）を制御することで読み書きができるようになっています。シリアルEEPROMのメモリ配列をメモリ空間・I/O空間に配置することはできませんが、小型の装置で利用できる8ピン程度のDIPやSOP等のパッケージ製品もあります。

ちょっと休憩- ROM (3)

マスクROM

- マスクROMは部品の段階ですでにデータ（プログラム）が書かれているメモリで、消去／書込みは全く不可能です。実際にはICの回路上でデータ（プログラム）がパターンとして作りこまれていて変更することは出来ませんので他のメモリとは概念的に大きく異なると考えて差し支えないでしょう。しかし、部品としてのコスト（1ビット当たりの価格）はROMの中でも低い方です（最も低いのはNAND型フラッシュメモリと言われてますがストレージ以外の用途ではマスクROMはほぼ最低価格です）。ROMに書くデータ（プログラム）が確定し将来変更の可能性がない場合に使用されています。イニシャルコスト（数十万円以上）が必要ですから、部品単価・装置の生産台数を考慮して採用するか否かの判断をする必要があります。

フラッシュメモリ

- フラッシュメモリはEEPROMから改良されたもので、1ビットの記憶セルを構成するトランジスタを2つから1つにすることでより大きな容量を得ることが可能となり、また、消去と書込みをブロック単位で行うようにしたことでEEPROMに比べて書き込みが速く、消費電力が小さくなっていることが特徴です。フラッシュメモリは従来のEPROMやOTPROMのようにプログラム等の固定的なデータメモリだけでなく、同時に一部をEEPROMのようにパラメータ保存領域として使うことも可能です。さらに、オンボードでプログラム書き換えができることやその書き換えが遠隔でも可能になること等メリットが大きく、最近のコストダウンと相まってその需要を大きく伸ばしています。
フラッシュメモリは大別してNOR型とNAND型があります（他の種類もありますがこの2つが現在の主流です）。
NOR型はランダムアクセス・高速アクセスが可能なので通常CPUのプログラムメモリによく使用されています。
NAND型はランダムアクセスが不可能で、またNOR型よりもアクセススピードが低速ですが、1ビット当たりの単価がNOR型よりも安いのが特徴です。欠点としてNAND型は読出し時の信頼性が低くビットエラーを起こすことがあります。そこでパリティのような補助的なビットをいくつか付加してエラー検出／エラー訂正を行うなどの工夫をして信頼性を向上させています。NAND型はストレージデバイスへの用途として（ハードディスクの代わりになるものとして）開発され、スマートメディア等の記憶素子に使われています。

ハードウェアの初期化

- ハードウェアリセットいっぱぁつ！ ・・・というわけにはいかない
  - 電気屋さんとのコミュニケーションが重要
    - CPUリセットと同時に、周辺デバイスはどのような振る舞いをするのか？ をよく知る必要がある。
  - その上で
    - 周辺デバイスに対してソフトウェアで何を設定（初期設定）すべきなのか、をシステム仕様から理解する。
    - 初期設定のプログラムをスタートアップ部に漏れなく記述する。
  - ハードウェアの自己診断結果を判断する
    - ハードウェアによっては、リセット投入後または初期化により自動的に自己診断を行うものもあり、その自己診断を意識してプログラムする必要もある。
    - 詳しくは電気屋さんの仕様書または周辺デバイスのデータシートをよく読むこと。

格言

スタートアップに始まりスタートアップに終わる。

リセットとは、責任を伴う行為である。

4. 周辺デバイス

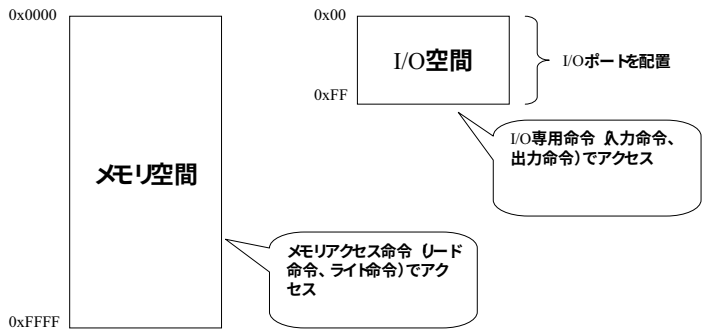
- 周辺デバイスはCPUから見て、入出力（I/O）の概念で扱われる。
- CPUと周辺デバイスは論理的にどうつながっているか？
  - 専用のI/O空間を持つ方式
  - メモリ・マップドI/O方式

どのような命令で入出力が実行されるか？

- ポート
  - 周辺デバイスとの間でデータの積み下ろしを行う「港/窓口」
- 通信
  - シリアルデータはデバイスでどのように扱われるか？

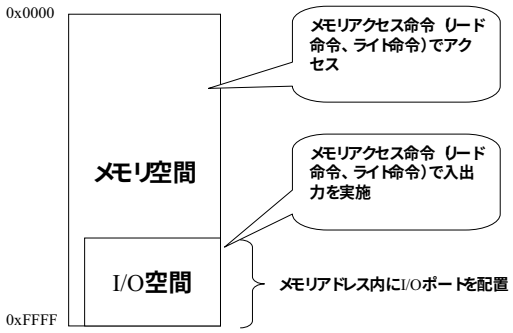
専用のI/O空間を持つ方式

- メモリ空間（アドレス）と、I/O空間（アドレス）を別に管理する方式



メモリ・マップドI/O方式

- メモリ空間内にI/Oを配置する方式



I/O（ポート）へのアクセス

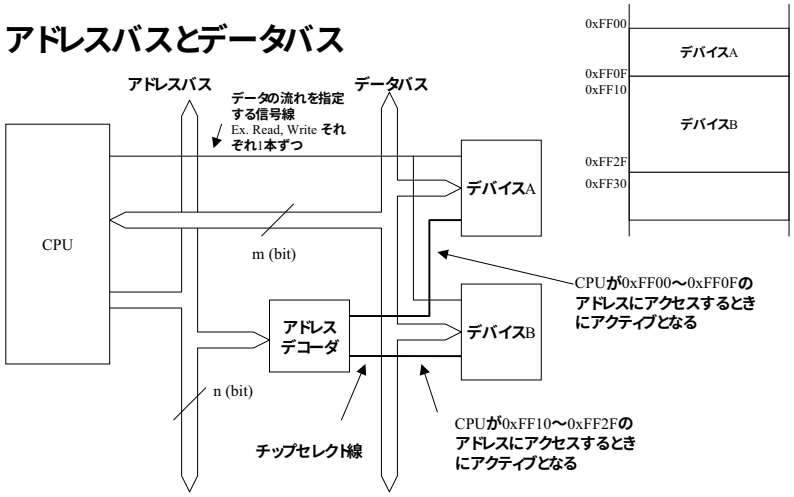
- ポートにはアドレスがある
  - I/O空間のアドレス or メモリ空間のアドレス
- ポートにはデータ型がある
  - 8bit 幅アクセス、16bit幅アクセス・・・・
- つまり、ポートへのアクセスでは、型指定が重要

```
unsigned int *device_A_port1 = (unsigned int *)0x1200;

unsigned char *device_B_port3 = (unsigned char *)0x1201;
```

- 同じアドレスでも、READとWRITEでは違う機能にアクセスする（メモリーはアドレスが同じならR/Wは同じメモリセルが対象）

汝、如何にしてデバイスにアクセスするや？(1)

- アドレスバスとデータバス



177

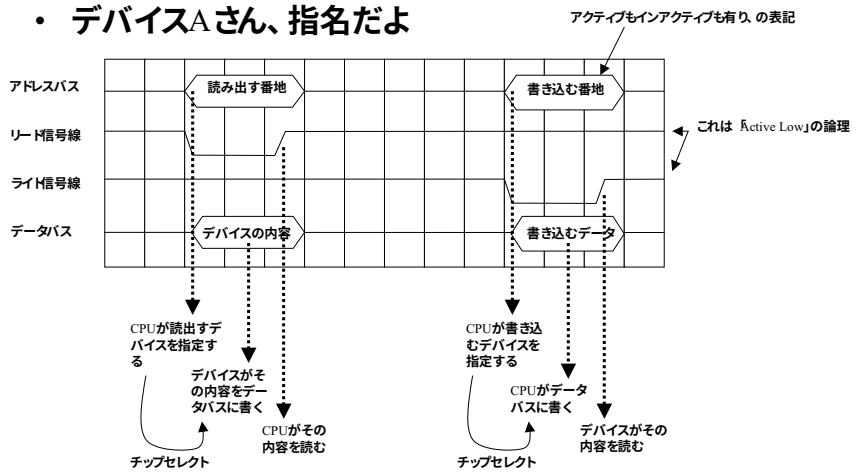
汝、如何にしてデバイスにアクセスするや？(2)

- アドレスバスとデータバス
  - アドレスバスが16bit幅であるCPUは、2^{16}=0xFFFF だけのアドレスを指定可能
  - 実際にすべてのアドレス空間にメモリやデバイスが隙間無く配置されることは少ない・・・ハード仕様書をよく読むことが重要
  - データバスが16bit幅であるCPUは、1回のREAD/WRITE命令（またはIN/OUT）で2バイト幅のデータ(short)を転送できる。

178

汝、如何にしてデバイスにアクセスするや？(3)

- デバイスAさん、指名だよ



179

汝、如何にしてデバイスにアクセスするや？(4)

- ご指名でないデバイスは玄関のドアを閉じる
  - トライ・ステート（Tri-State）
    - High H デジタル論理の「高」状態
    - Low L デジタル論理の「低」状態
    - High Impedance Z 接続していないのと同様の状態
- つまり
  - チップ・セレクトがアクティブでなければ、デバイスはデータバスをHigh Impedance状態にして「しらんぷり」を決め込む。

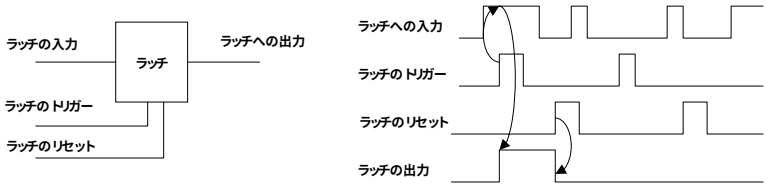
「混信」を防ぐだけでなく、ファンアウト性能のからみもある。

聴衆が少なければ小さな声で良いが、聴衆が増えると大きな声を出さないと通らない

180

ラッチとゲート(入出力)

- プログラムは、見に行った時点の「状態」しか知りえない・・・
- ある時点のスナップショットを「保持」するラッチ
  - ソフトウェアトリガまたは定期的なクロックでスナップショットを取得する
  - 保持した値をリセットするには明示的に「リセット」を指示する



- 状態をリアルタイムにスルーするゲート
  - 読みに行った「瞬間」の状態を取得する

I/Oマップ– I/Oの仕様書

入力の記述例

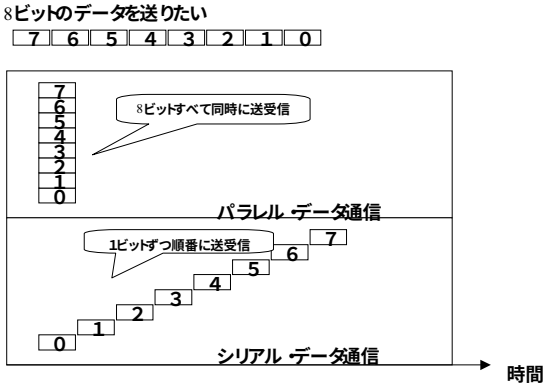
番地	bit	周辺装置	デバイス	注釈
0xFF00	0(LSB)	スイッチA	ゲート	0: 押されている 1: 押されていない
	1	スイッチB	ゲート	0: 押されている 1: 押されていない
	••••			
	7(MSB)	スイッチH	ゲート	0: 押されている 1: 押されていない
0xFF01	0 - 7	パラレル	ラッチ	
••••				

出力の記述例

番地	bit	周辺装置	デバイス	注釈
••••				
0xFF01	0	パラレル	ラッチ	ラッチトリガー
	1	パラレル	ラッチ	ラッチのリセット
••••				

通信　：シリアルとパラレル

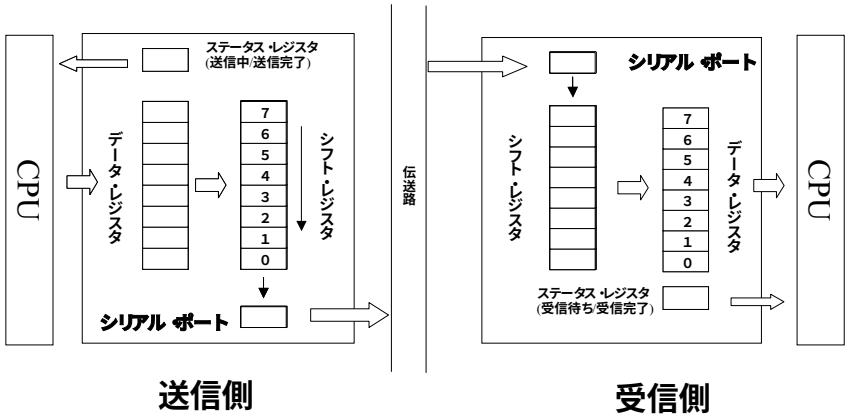
- **シリアル通信 :**RS232C, Ethernet, USB, IEEE1394 etc.
- **パラレル通信 :内部バス**, IDE, **セントロニクス**, SCSI etc.



183

SESSAME CONTENTS 2004

シリアル通信の仕組み



184

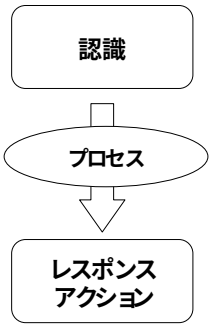
SESSAME CONTENTS 2004

格言

隣人をよく理解することはトラブル防止への第一歩

5. エッジ・トリガとレベル・センス

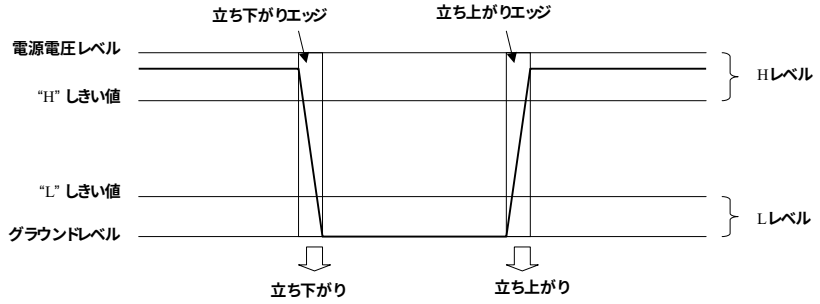
- 現象を、デバイスを通してどのように受け取るか？
  - 入力
    - スイッチ入力
    - LSIの状態入力
    - 通信データ入力
  - 入力には、次の3種類がある。
    - 変化したことを読み取る
    - 読取りにいった瞬間の状態を読む
    - ある時点の状態を保持し、それを後から読み取る



- デバイスを通してどのようなデータ出力を行うか？
  - データバス上の「出力状態」は瞬間
  - 瞬間の出力で良いか、瞬間の状態を保持する必要があるか？

変化したことを読み取る - エッジ センス

- おや、立とうとしているね・・・
  - 信号の 変化」を捕らえる
- 立ち上がりエッジと立下りエッジ



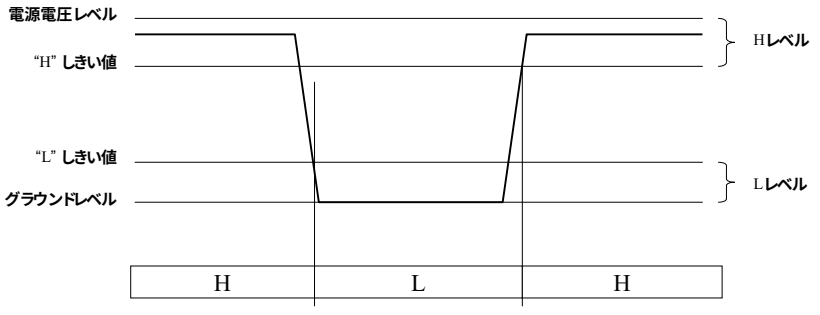
クロックはエッジトリガ方式として使用される信号の代表的なものです。各種信号の同期を取るのに用いられます。
割り込みに関して、レベルセンス／エッジトリガをソフトウェアで選択設定できるマイコンがあります。いずれに設定するかは、割込みの性質によって変わってきます。
センサやスイッチ等の変化を割込みに使用する場合はエッジトリガ方式が一般的です。

187

SESSAME CONTENTS 2004

読取りにいった瞬間の状態を読む - レベル センス

- うん、君は今座っているね
  - 信号の 状態」を捕らえる



回路では、チップセレクト R/W信号等の制御信号でレベルセンス方式がしばしば用いられます。

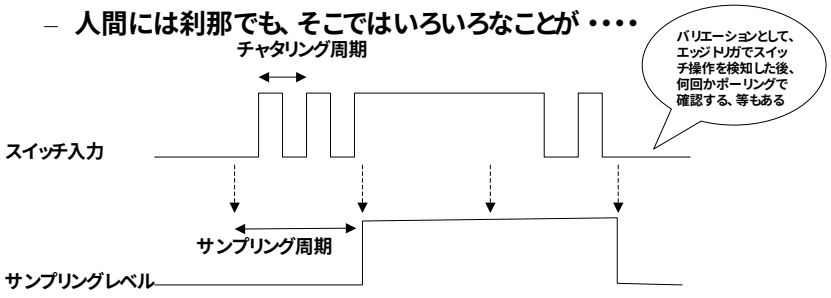
PCIバスで採用されているように、複数デバイスで割り込みが共有されているような場合にはレベルセンス方式が使用されることがあります。

188

SESSAME CONTENTS 2004

君、玄関のベルを2回押した？

- チャタリング
  - スイッチやリレーの接点が、自身の持つバネ性によりバウンスしてしまい、信号が短い周期でH/Lを行き来すること。
  - 人間には刹那でも、そこではいろいろなことが ・・・・



チャタリング周期よりもサンプリング周期を長く設定することでチャタリングの影響を回避できる。
ただし、サンプリング周期を長くしすぎると、操作に対する反応の遅れを生み、操作感を損なってしまうので注意が必要。

ある時点の状態を保持し、それを後から読む – ラッチ

- ラッチの実施方法
  - 特定の信号線の立ち上がりエッジ／立下りエッジでデータをラッチする
  - 定期的にデータをラッチする
    - クロック信号のエッジでラッチ
    - ソフトウェアでラッチストローブを出力
    - など

変化する状態や信号を処理しなければならない場合に、ラッチはとても便利です。

ただし、定期的なラッチも、プログラムの処理状況によって1サイクル分読み漏らしをしてしまったりする危険性が無いとはいえません。
そのような場合はラッチではなくDMA転送でゲートから読み込んだ値をメモリの特定領域に連続して格納することもあります。読み込むデータの周期、読み漏らしリスクの程度などにより、最適なデータ入力の回路・手法を選定するようにしましょう。

格言

変化点か状態か、それが問題だ

入出力の定石

- ハードウェア仕様をよく理解する
  - 周辺デバイスの初期化、アクセスに際して必要な制御、それらにハードウェアとして要する時間　などをよく理解する。
- 入出力のデータ幅、データ型に注意する
- 入出力がラッチなのか、ゲートなのかをよく理解し、
  - ゲートの場合は事象の発生として扱う最低（and/or 最高）の時間幅を明確にし、
  - 入出力の原因となる事象とあわせて、ソフトウェアが得た信号の扱いを適切にプログラミングする。

6. 組込みプログラムの構造

- プログラムではあれこれしながら、何やかや ・・・
  - タイムリな処理
  - ハードウェアからの状態取得
  - ハードウェアへの出力
  - ・・・・・

いくつもの皿回しをしながらお客さんと会話をし・・あぁ、サインをねだられちゃった・・おっと皿の回転が弱まったからエネルギー供給、っと・・あぁそうそう、サインが途中だった・・・えっ、今度は皿じゃなくてやかんだって！？

- 周辺デバイスからの状況をどのような仕掛けで入手するか
  - ポーリングを使用する
  - 割り込みを使用する

- プログラムの実行構造をどのようにするか
  - 同期呼び出し
  - 非同期呼び出し

ポーリング – 御用聞きモデル

- 各家庭を訪問して 何かご入用のものはありますか」と注文を聞き、それを届ける 御用聞き」
  - 訪問の順序は決まっている
  - 訪問の途中の各家庭での注文の有無、急ぎの配達のために一旦店に戻ったりという事象発生の有無などにより、訪問の時刻が保障されない
  - 各家庭からの注文に即時対応できない

```
main()
{
    初期化;
    while(1) {
        事象Aを調べる;
        事象Aが条件を満たしていたら処理する;
        事象Bを調べる;
        事象Bが条件を満たしていたら処理する;
        ・・・・・・・
    }
}
```

ポーリング周期の要件を明確にする。
全事象への処理時間が周期要件よりも長くならないようにする。
ことが重要！

割込み – 電話注文モデル

- 電話で注文を受け、焼きたてピザを届けるピザ屋
  - 電話が入るまでは下ごしらえなどの別処理をしている
  - 電話が入ると、それを待ち行列につなぎ、あるいは優先度の組換えをしながら約束のタイミング（配達納期）に間に合うように注文に従った処理を実施
  - 下ごしらえ中に電話が入ると、「どこまで下ごしらえをした」メモを書き、受話器を取る

```
main()   // 割り込まれる側
{
    初期化;
    while(1) {
        割込み禁止にする;
        割り込まれてはならない処理;
        割り込みを許可する;
        割り込まれても良い処理;
    }
}
```

```
interrupt()  // 割り込み処理
{
    割込み処理で使う資源を退避する;
    割り込みに応じた処理;
    退避していた資源の復帰;
}
```

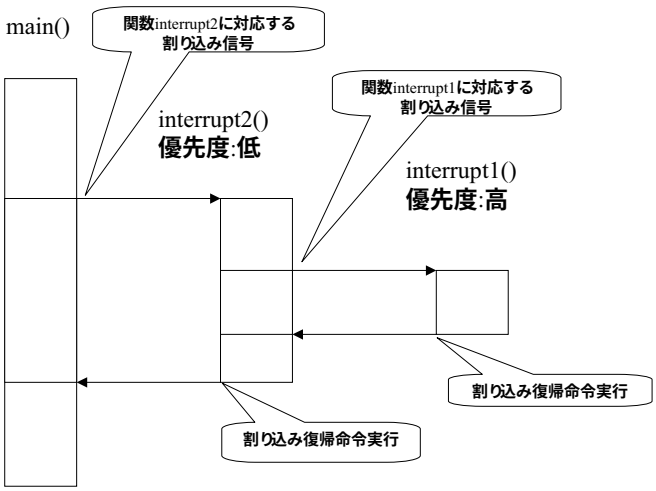

割込みも交通整理が必要

- 割込みのおきて
  - 割込みが発生した際に、処理中の作業のメモ（プログラムカウンタ、スタックポインタ等：コンテキスト）を保存する
  - 割込み処理を実行する
  - コンテキストを復元する（i.e. もとの作業に戻る）
- 割り込みの交通整理
  - 割込み優先度を考慮する
    - 割込みには優先度が付与できる
      - 割込みが重複した場合、優先度のより高い割込みが先に処理される
  - マスク可能割込み/マスク不能割込みを使い分ける
    - 割込みの受付を禁止／解除できる割込み　：マスク可能割込み
    - どのような状態下であっても受けなければならない割込み　：マスク不能割込み

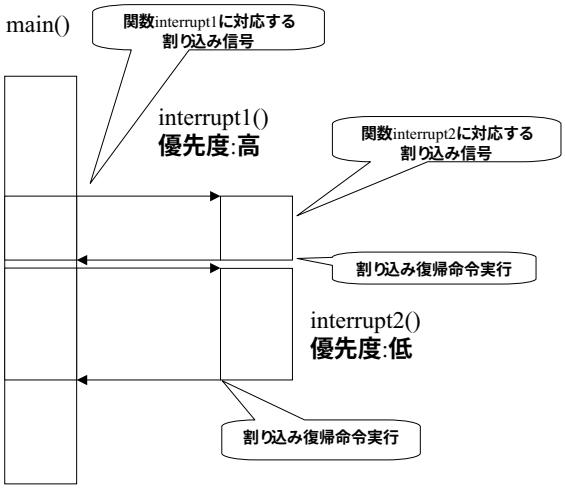
注文での長電話はご法度

- ピザの注文の電話で長電話をしてはいけない
  - 電話（割込み処理）は短く簡潔に
  - 急ぎの用件は割込み処理の中で、受け付けてから実処理まで余裕のあるものは別処理で
    - 割込みの発生したこと、詳細情報などをフラグ等にセットし、通常処理に受け渡す
  - 割込み応答の時間、割込みが発生してからそれに応じた処理が開始されるまでの最遅時間（ワーストケース）をきちんと抑えること
- お得意様や遠方のお客様の処理は早く
  - 割込みには優先度を設定する
  - キャッチホン（多重割込み）で優先度の高い割込みを先に扱う

キャッチホンをうまく使う - 多重割込み



多重割込み – お得意様は神様です



199

割り込み処理の定石

- 割込みで対応すべき入力／イベントを洗い出す
- それらについて性格を分析し、明確に定義する
  - 発生頻度、重要度 緊急度はどれくらいか
  - 発生したら必ず受けなければならないものか
    - マスク可能か、不可能か
  - 発生した場合の対応処理にはどのくらいの時間が必要か
    その時間のばらつきはどれくらいか オーバーしてしまう可能性はどれくらいあるか）
  - 割込み発生から処理開始までの遅延はどの程度許されるのか、そのマージンはどの程度か
  - 受け付けた後、メインの処理の性格が変わるようなものか
    - 受け付けて処理した後、元の仕事に戻るかどうか
    - 元の仕事に戻らない場合、元の仕事に与える影響はどういったものか
  - 割込み処理の中で使用するスタック量はどれくらいか
- 割込みルーチン内で処理すべきもの、メインの処理内で扱うべきものを明確にする

200

処理の実装

- 例えばエアコンのボタン操作を受け付ける処理は
  - ポーリングを繰り返してボタン押下を検知する
  - ボタン押下をハードウェアからの割り込みとして検知する
  - OSレベルに任せる

など、さまざまな実装方法がありえる。

Windowsでのボタンクリック処理など

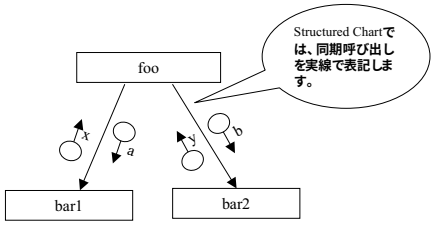
- 処理それぞれで必要とされる要件を明確にする
- 同期呼び出し、非同期呼び出しをうまく使い分ける

ことが重要

同期呼び出し

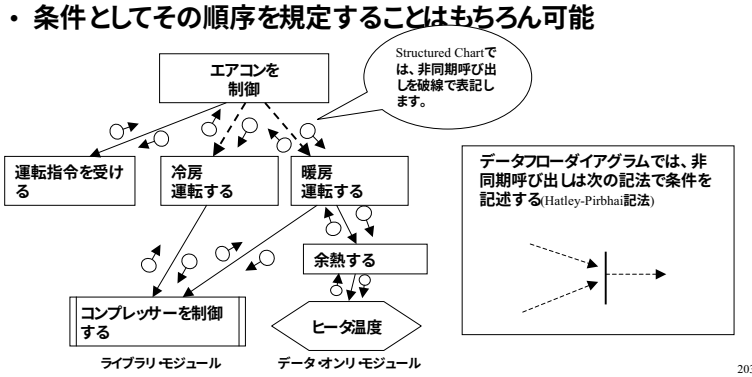
- シングルタスク処理のイメージ
- JISフローチャートのように、手順どおりに：
  - 呼び出しは、直ちにそれを実行開始することを意味し
  - 「呼び出す／結果を受け取る」を必ずペアで扱う　つまり
  - 先の結果を受け取るまでは、次の処理を呼び出すことができない

```
foo()
{
    ....
    x = bar1( a );
    y = bar2( b );
    z = x/y;
}
```



非同期呼び出し

- 処理の起動の直後に処理が必ずしも実施されない
  - 必要な条件がそろうまで待ち、条件が揃った時点で実行
  - 個々の処理は、その実行順序が規定されない
    - 条件としてその順序を規定することはもちろん可能



格言

締め切りは待ってくれない

へたな鉄砲は数を撃っても当たらない

7. volatile – 変数と最適化

- コンパイラの最適化によるおせっかい迷惑を防ぐための宣言
  - 例えば、周辺デバイスのレジスタの中身を確認する時、現在最新の状態を読み直す動きをさせるようにコンパイラに指示する機能。
  - 例えば、左のソースコードはコンパイラの最適化機能により右のように扱われる

```
int data;
.....
while(1) {
      if ((data&4) != 0) {
            処理1;
      }
}
.....
```

結果のあらかじめわかっている判定をループの度にいちいち実施するのは処理を遅くするだけ！

=

```
int data;
.....
if ((data&4) != 0) {
      while(1) {
            処理1;
      }
}
.....
```


最適化による予期せぬ振る舞い

- 周辺デバイスのレジスタをポーリングで読み込むような場合にvolatile宣言を付けないでいると、コンパイラによっては最適化オプションが災いして、再度レジスタを読むという作業を省いて機械語に変換してしまう。

```
const unsigned char *port;
port = 0xfd0c;
.....
while(1) {
      /* レジスタのbit2をチェック */
      if (((*port)&4) != 0) {
            処理1;
      }
}
.....
```

結果のあらかじめわかっている判定をループの度にいちいち実施するのは処理を遅くするだけ！

と、コンパイラは判断してしまう・・・

=

```
const unsigned char *port;
port = 0xfd0c;
.....
if (((*port)&4) != 0) {
      while(1) {
            処理1;
      }
}
.....
```

期待した動作をしない・・・

volatileの使用

```
volatile unsigned char *port;
port = 0xfd0c;
.....
while(1) {
      /* レジスタのbit2をチェック */
      if (((*port)&4) != 0) {
          処理1;
      }
}
.....
```

コンパイラは、volatile宣言された変数は最適化の対象外であると判断します。

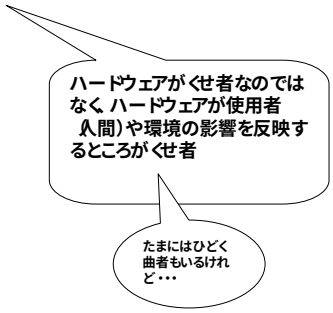
- ポートから読み込む値だけでなく割込み処理や他のタスクから書き換えられる可能性のある変数をチェックするような場合、volatile宣言の考慮が必要！

格言

シルクハットの中のハンカチは
突然ぞうきんに変わる・・かも

8. ハードウェアとのお付き合い

- ハードウェアは曲者だらけ
  - タイミング
  - 順序
  - 前準備、後処理
  - 本当にその値？？
  - ノイズとのお付き合い
  - 発熱があると・・・・

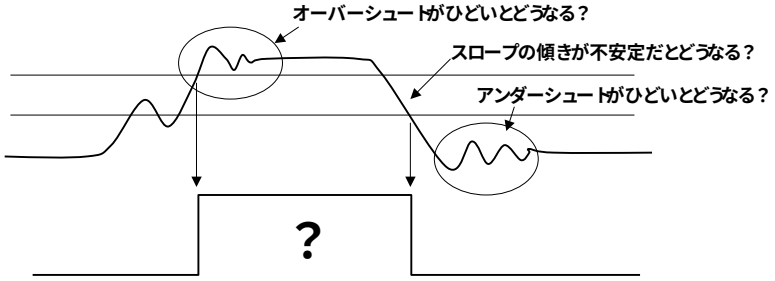


デジタルはアナログより簡単？

- デジタルの波形
  - 矩形波形は、無限の周波数（高調波）成分を含む
- デジタル信号の伝送路は、抵抗がゼロではなく、隣の配線とコンデンサ要素を形成したり、自分自身がコイル的な要素を持ったりする。
  - 交流的抵抗成分「インピーダンス」を持つ
- インピーダンスを持つことにより、信号線は高周波ほど伝送特性が劣化する、
- また、信号線は隣接する信号線上の信号に影響を与える（クロストーク）、つまり
- 信号線を通るデジタル波形は、理想的な形ではなく、崩れている。

デジタルはアナログより簡単？

- デジタルの「グレーゾーン」
  - デジタル回路も、内部はとてもアナログ的
  - デジタルはノイズに強いって！？
  - デジタルは時間に正確か？



また、ディジタル回路で用いられるパーツには、Hiレベルと Lowレベル 2つのしきい値があります。
各パーツのデータシートには、Hiレベル／Lowレベルの入出力電圧に関する記載があります。
中間レベルでは、Hi/Low不定です。

211

ハードウェアは確実？

- ソフトウェアが状態を出力したものが出力デバイスに直ちに反映される・・・・・とは限らない。
- 周辺デバイスなどのハードウェアの状態変化を直ちに 間違いなくソフトウェアで読み取ることができる・・・・・とは限らない。
- ハードウェアの状態は動作環境によって不変で安定している・・・・・とは限らない。

信号やハードの振る舞い、タイミングに関する信頼性がどの程度実現されるべきか、は、製品によって異なります。
ハードウェアの「癖」に対応するための定石をいくつも身につけておき、さまざまな信頼性要求に応えられるようになりましょう。

212

なめるなよ、ノイズ

- 内部で発生するノイズ
- 外部から侵入してくるノイズ
  - 電源から
  - アースを取ったラインから
  - 電磁波として
  - 人間を経由して
    - 誘導
    - 静電気
- ノイズの影響を受けやすい回路配置がありうるが、製品企画上ハード屋さんがそれ以外に選択できない場合がある
  - そのような場合は、ノイズによる誤動作を防ぐ判断ロジックをソフトの側で組む必要がある

温度によって、湿度によって、操作者によって、気分によって（らそ）、つど変わる

しかし、システムへの論理的な不安定要因であるのは間違いない

百推は一見にしかず

- 動作が不安定な場合、問題領域のソースコードや実際のスタックの状態を見直すのは当然ですが、ソースコードにミスが見つからない場合は、信号の状態をオシロスコープやロジックアナライザで実際に見てみましょう。
- 百の推測すべてに思いを致して悩むより、推測を絞り、その推測に基づいて「システム」を実際に覗いて見ましょう。

ハードウェア任せ、ではなく、より良い・信頼性の高い製品にするために、ソフトとして何ができるかを考えることは重要なことです。

格言

知彼知己　百戦不殆
彼を知り、己を知れば、百戦して危うからず）

トラブルは境界に潜む

自分と他メンバー
ハードとソフト
製品の内側と外側
チーム内と外
発注元と開発者 etc.

「不安定」には論理的な原因がある。
その原因をハードで対処できない場合は、ソフトで対処しなければならない

システム屋への道

- 組込みシステムは、メカ屋さん、エレキ屋さん、ソフト屋さんの努力の結果世に出ます。
- それぞれの担当領域でのエキスパートとなるのは当然として、しかし他の担当領域に興味を示さなかったり、「あとは君らの仕事、あっしの預かり知らぬことで」というのでは良い品質の製品を世に出すことは難しいでしょう。
- エキスパートにならないまでも、他の領域のシゴトが理解できるようになりましょう。
- 「ソフト屋育ち、メカ・エレキもそれなりに分かる」システム屋さんは、製品の要求分析やシステム設計の時に核となれる存在です。こういったメンバーが上流工程できっちり仕事をすると、その後の仕事が楽になると思います。
- ぜひ、システム屋を目指してください。

ソフトウェアテストの概要

西 康晴

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

目次

- テストは大事
  - バグの影響はとんでもなく大きいのだ
  - テストが上手になるとバグの無い開発ができる
- テストの進め方
  - テストのフェーズとプロセス
  - テストに必要な視点
- テストの設計
  - 単体テスト・結合テスト・機能テスト・システムテスト
- テストしやすい開発をしよう
  - テストが上手になるとバグの無い開発ができる

バグの影響はとんでもなく大きいのだ

- 300万行のうち1行余計だったせいで、AT&Tは11億ドルの損害
  - エラー回復コードがバイパス
  - 長距離電話が9時間に渡り話し中になった
- ブレーキシステムのバグで、GMにリコール？
  - 停車距離が15～20m伸びてしまった
  - 350万台がリコール、数百億ドルの損害
- ARIANE5ロケットは爆発
  - オーバーフローが原因
  - 4億ドルの損害

219

SESSAME CONTENTS 2004

テストは簡単？

- テストなんて、仕様を確認するだけじゃないか
- ちゃんと作れば、テストなんていらない
- テスターは、力の無い奴にやらせればいい
- テスターはあら探しをするからムカつく

220

SESSAME CONTENTS 2004

テストとデバッグは同じじゃないの？

- デバッガ上で動かしてみるのはテストではない
  - バグを「いぶり出す」ために知恵を捻って設計すべし
  - 動きやすいテストをしてはダメよ
  - バグが見つかって始めてテストは成功なのだ

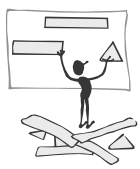
テストが上手になってくると
そもそもバグのない開発ができるようになる

221

テストはどうやって進めればいいの？

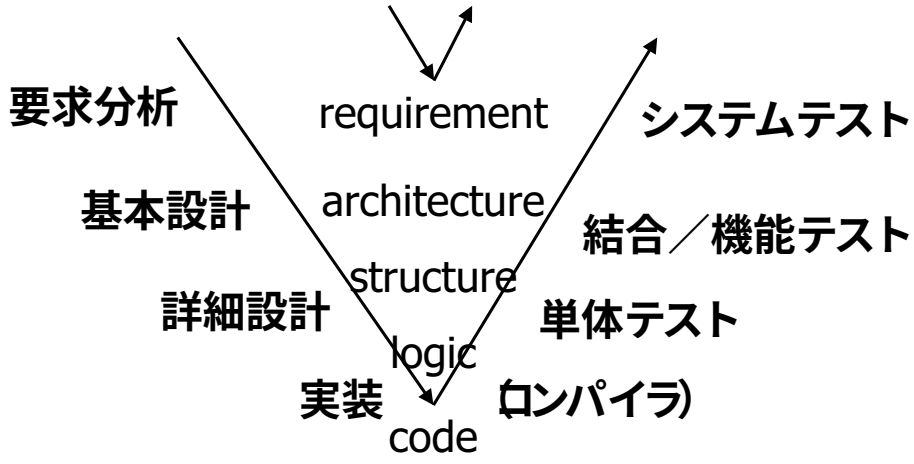
- 組込みソフトのテストのフェーズ
  - 下位Vモデル
    - 単体テスト／結合テスト／機能テスト／システムテスト
  - 上位Vモデル
    - シミュレータテスト／実機テスト／シミュレータテスト
  - リグレッションテスト（回帰テスト）
    - 修正や変更の副作用で入り込んだ新たなバグも
      きちんと見つけて取り除こう

行き当たりばったりではなく
きちんと「設計」しよう



222

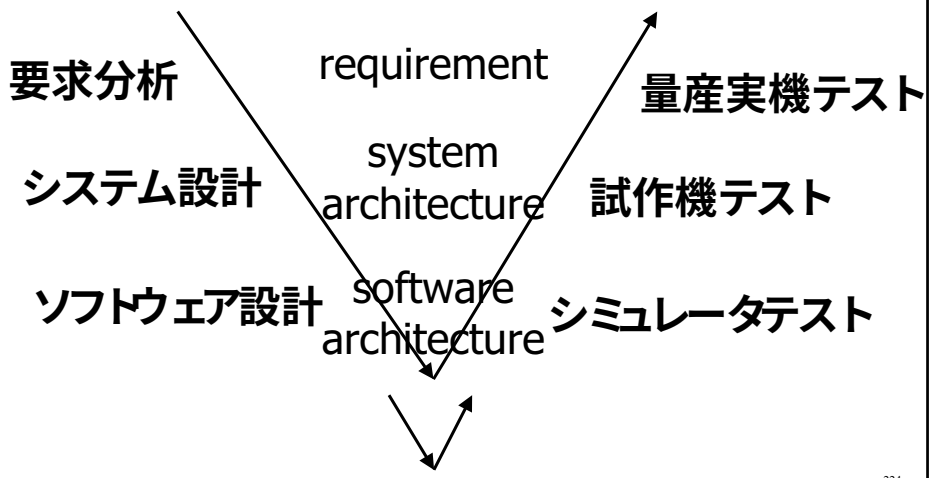
組込みソフトのテストのフェーズ：下位Vモデル



223

SESSAME CONTENTS 2004

組込みソフトのテストのフェーズ：上位Vモデル



224

SESSAME CONTENTS 2004

テストを設計する時には何を意識すればいいの？

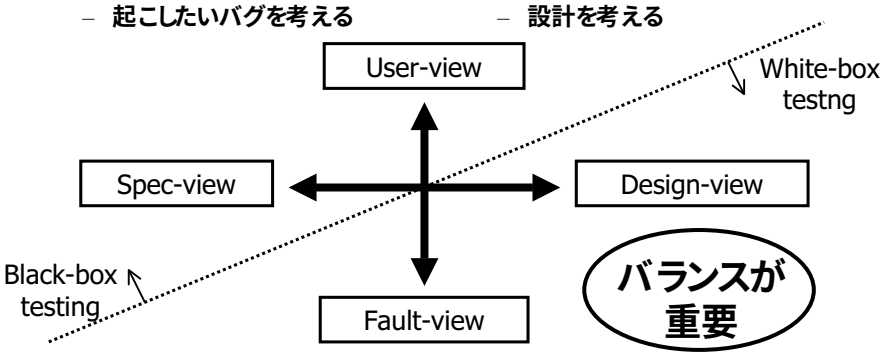
- 網羅
  - テスト漏れにはバグが潜んでいるかもしれない
    - どれくらい網羅したかを測る :カバレッジ
      - 制御パステスト
      - 機能網羅テスト
- ピンポイント
  - バグの多そうなところを狙う
    - 先人の知恵
      - 境界値テスト
      - ストレス系テスト
    - 過去の経験
      - バグの分析をして自分の弱いところを把握しよう

期待結果を
必ず記述すること

225

テスト設計に必要な視点

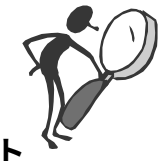
- User-view
  - ユーザが何をするかを考える
- Fault-view
  - 起こしたいバグを考える
- Spec-view
  - 仕様を考える
- Design-view
  - 設計を考える



226

モジュールをテストしよう:単体テスト

- どんなテスト?
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 詳細設計やプログラミングに着目したテスト
- どんな手法でテストするの?
  - 境界値テスト
    - モジュールの引数などに与えるテストデータを はじっこの値 境界値)にするテスト
  - 制御パステスト
    - モジュール内のロジックを全て通すように テストデータを与えるテスト



両方とも必要!

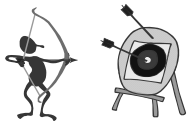
境界値にはバグが多い

- 境界値や限界値の周りにはバグがたくさん潜んでいる
  - 条件文の間違い:どんな間違いが多い?
    例) ○:if (a>0) then　　×:if (a≧0) then
    のように間違えることが多い。
    境界値であるa=0でテストすればバグが見つかる
    - if文などの条件文やwhile文などのループには、 境界ずれや一つ違いのバグが多い
  - リソースのリミット:テスト文字列の長さは?
    例) 10バイトの固定長文字列を入力するプログラムの場合 とても長い文字列を入力すれば落ちるかもしれない
    - 大きさを持ったデータ構造には、バッファオーバーフローなど 容量の限界値ギリギリの処理を忘れているバグが多い

バグが多いところを狙おう 境界値テスト

- 同値クラスを見落とすと、ゴソッとテストがモレる
  - 境界値や限界値を抽出する範囲を「同値クラス」と呼ぶ
    - モジュールの引数、返値の境界値
    - if文の条件式など内部の境界値
    - テンポラリファイルの容量などI/Oの境界値
  - 正常境界値だけでなく異常境界値もテストする
  - 境界値を考えるのをサボっちゃダメよ

どんなときでも
境界値を考えよう



229

境界値テストの練習をしてみましょう

- 以下のモジュールのテストを設計してみましょう
  - int型の引数aがあります
  - モジュール内にはif (a<0) とif (7<a)という2つの条件文があります

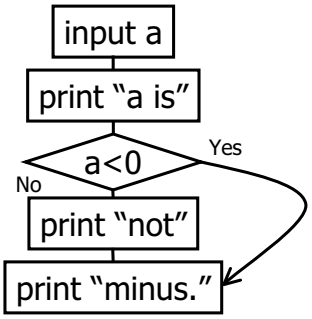
- まず同値クラスを挙げてみましょう。
- 次に同値クラスの境界値を挙げてみましょう
- 最後に、どんなバグが見つかるかを考えてみましょう

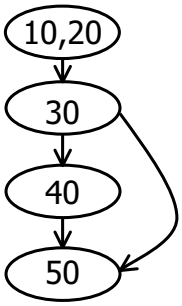
230

ロジックを網羅しよう:制御パステスト

- ロジックを網羅して全てきちんとモレなくテストしよう
  - フローグラフを描いてロジックを抜き出す
    - フローグラフ:ノードとリンクで書かれた図
    - フローチャートや状態遷移図はフローグラフ

```
10  input a
20  print "a is"
30  if (a<0) goto 50
40  print "not"
50  print "minus."
```

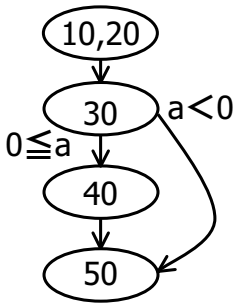




231

制御パステストの設計 :パスの網羅

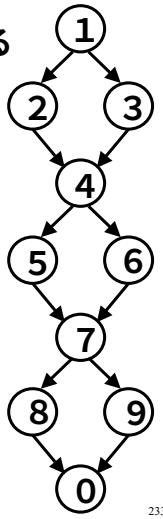
- パスとは？
  - フローグラフ上の経路
  - プログラムのロジック
  - パスの数がテストの数に比例する
- パスを全て網羅するように
  テストを設計する
  - まずパスの一覧表を作る
    - 10→20→30→40→50
    - 10→20→30→50
  - 次にパスを通るデータを実装する
    - 10→20→30→40→50: a=0
    - 10→20→30→50 : a=-1



232

どこまでテストすればいいのかな？

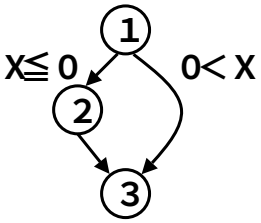
- 全てのパスを組み合わせようとすると膨大になる
  - if文の数の累乗で増えていく
  - if文に含まれるandやorの数の累乗でも増えていく
    - 右のグラフでは8本のパスが必要となる
- 最低でもif文の両側の分岐は1度テストしよう
  - 条件網羅（C1基準）
- 組み合わせが増えないように気を付けて開発する必要がある
  - KISS: Keep It Simple, Stupid!



参考：命令網羅（C0基準）

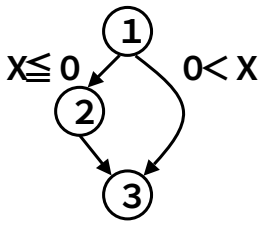
- どれだけ網羅できているか、を「カバレッジ」と呼ぶ
  - フローグラフのカバレッジ
  - 機能カバレッジ
- 全ての命令を一度実行すればいいような気がする
  - 命令網羅／ノード網羅／C0基準
- 右のフローグラフでは
  テストケースは1つでよい？

  ┌A：1→2→3
  └　　X＝0
- 明らかにテストが足りない
  - if (9<x) とタイプミスしていても分からない



参考 条件網羅（C1基準）

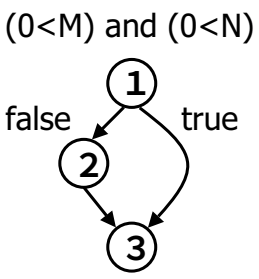
- 少なくとも条件文のtrueとfalseは網羅しなくてはならない
  - 条件網羅／リンク網羅／C1基準
  - 右のフローグラフでは、
    テストケースは以下の 2つになる
    - A：1→2→3
      X＝0
    - B：1→3
      X＝1



  - if (9<x) とタイプミスしていても分かる
    - 両方とも左に分岐するのでバグであると分かる

参考 複合条件網羅（C2基準）

- 組み合わせていない単一の条件式を
  すべて網羅しないとテストできたことにならない
  - 右のフローグラフでは、
    テストケースは以下の 4つになる
    - A：1→2→3　（falseの場合）
      (M＝-1, N＝-1)
      (M＝ 1, N＝-1)
      (M＝-1, N＝ 1)
    - B：1→3　　（trueの場合）
      (M＝ 1, N＝ 1)

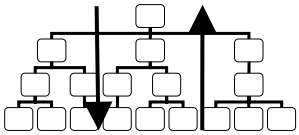


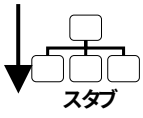
設計をテストしよう:結合テスト

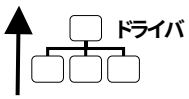
- どんなテスト?
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 概要設計や構造設計に着目したテスト
    - 状態遷移設計やモジュール間構造設計に着目したテスト
- どんな手法でテストするの?
  - トップダウンテスト/ビッグバンテスト
    - モジュール間の結合のミスを探すテスト
  - 状態遷移パステスト/状態遷移マトリクステスト
    - 状態遷移図や状態遷移マトリクスのミスを探すテスト
  - など

モジュール間のインターフェースのテスト:結合テスト

- モジュールをつなげる時のテスト
  - モジュールの結合順序を決める
  - ドライバやスタブを作成する
  - インターフェースをテストする
- トップダウンテスト
  - 上位のモジュールから先に作成し、結合していく
  - スタブが必要
- ボトムアップテスト
  - 下位のモジュールから先に作成し、結合していく
  - ドライバが必要
- ビッグバンテストは禁止である

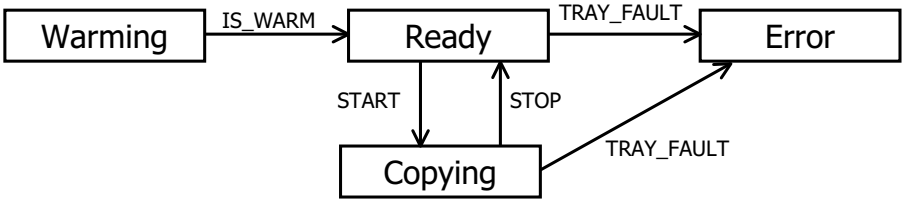


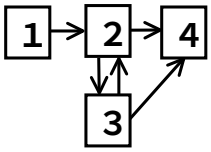




イベントを順番に起こそう :状態遷移パステスト

- イベントを順番に起こしてテストすることで
  状態遷移が正しく実現されているかをチェックしよう





パスの数は何本必要？

- 1→2→4
- 1→2→3→2→4
- 1→2→3→2→3→2→4
- 1→2→3→4
- 1→2→3→2→3→4
- 1→2→3→2→3→2→3→4

239

イベントと状態を網羅しよう :状態遷移マトリクステスト

- 状態遷移マトリクスをモレなくテストすることで
  状態遷移が正しく実現されているかをチェックしよう

何が起こるか分からない
↓
バグの可能性が高い

	S / E	起動処理中 (Warming)	スタンバイ中 (Ready)		
		0	1		
スタンバイ完了 (IS_WARM)	0	→Ready	×	×	×
コピーボタン押下 (START)	1	／	→Copying	?	／
コピー完了 (STOP)	2	×	×	→Ready	×
エラー発生 (TRAY_FAULT)	3	?	→Error	→Error	×

→ 遷移
／ 無視
× ありえない

- 状態遷移マトリクステストを設計すると、
  矛盾や抜けのある遷移もレビューできる

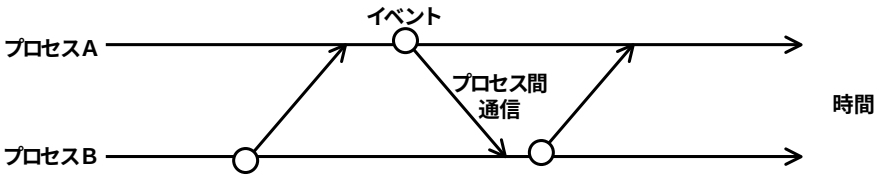
240

状態遷移テストの注意点

- 状態遷移パステストと状態遷移表テストは使い分ける
  - リンク網羅ならばカバレッジとしては等価
  - 状態遷移パステストは、ユーザの操作などと対応させてテストしやすい
  - 状態遷移表テストは、異常なイベントへの対応が網羅的にテストできる
- 状態遷移図そのものの間違いに気を付ける
  - 仕様書をよく読んで検討する
    - 状態や遷移のモレ／あいまいな遷移条件
- 時間に関する状態には気を付ける
  - ある一定時間だけ初期化処理の状態になる、など
  - その状態でいられる最小時間と最大時間を明確にする
- モデルと実装の差異に気を付ける
  - モデル上では独立な状態でも、履歴に依存するかもしれない
  - モデル上では瞬時に遷移することになっているが、実際にはほんの少しでも時間がかかるため、ありえないイベントが発生する

241

パステストの応用 :シーケンス図のテスト

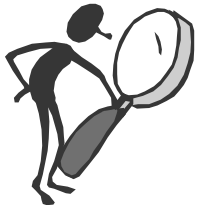


- C0基準 :イベント網羅
  - 一つ一つのイベントがちゃんと発生するか
- C1基準 :イベント間通信網羅
  - 一つ一つのプロセス間通信がちゃんと受け取れるか
- イベント発生可能性時刻の同定
  - 境界値テストで明らかにする
- フローグラフで表現できる設計は全てパステストが可能

242

コンパイル後に改めてテストしよう 機能テスト

- どんなテスト?
  - 開発者だけでなく、テスト担当者や品質保証部門も行うテスト
  - 組み上がったソフトウェアレベルのテスト
  - 機能仕様に着目したテスト
- どんな手法でテストするの？
  - 機能網羅テスト
    - 機能一覧を作って網羅するテスト
    - 簡単そうだが意外にやらない
  - 境界値テスト
    - 機能のパラメータに着目した境界値テスト
    - 仕様のバグが見つかる



243

機能網羅テスト

- ソフトウェアの持つ機能を漏れなくテストする
  - 表を作って機能を全て書き、OKになったらチェックする
    - 基本中の基本だが面倒がって行わないことが多い

機能	*チェック欄*
機能１	レ
機能２	
機能３	レ
機能４	
機能５	レ
機能６	

244

マトリクス網羅テスト

- 2次元の組み合わせを網羅する場合のテスト
  - 機能×データ)、状態×イベント)など
    - 行と列の組み合わせが網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	レ			
条件2			レ	
条件3	レ			
条件4				レ
条件5		レ		
条件6				

    - ありうる組み合わせ」が網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	○ レ			
条件2	○	○	○ レ	
条件3	○	○ レ		
条件4	○	○	○ レ	○
条件5		○	○	○ レ
条件6				○

245

境界値とバグ

- 境界値や限界値の周りにはバグがたくさん潜んでいる
- バグが多いところを狙ってテストしよう 境界値テスト
- 仕様の境界値をどんどんテストしよう
  - 境界値を考えると、仕様があいまいなことに気付く
    - 4月1日生まれは早生まれ？遅生まれ？
      - 民法第143条 :満年齢は起算日に応当する前日をもって満了する
      - 学校教育法第22条 :満6歳に達した日の翌日以降における最初の学年

**境界値テストは
なるべく上流で設計して
仕様のあいまいな部分の
レビューに使おう**

246

色々いじめてみよう:システムテスト

- どんなテスト?
  - 開発者だけでなく、テスト担当者や品質保証部門も行うテスト
  - ソフトウェアを使うユーザレベルのテスト
  - 要求仕様に着目したテスト
- どんな手法でテストするの？
  - ストレス系のシステムテスト
    - ソフトウェアに負荷をかけるテスト
    - よくバグが見つかるので手抜きしないでテストする
  - 環境系のシステムテスト
    - 相性や互換性の問題を見つけるテスト
    - データだけでなく、電気や熱の流れなどにも気を付ける
  - 評価系のシステムテスト
    - どれくらい堅牢か、どれくらいユーザが満足するかの評価
    - ユーザに使わせるだけでなく、きちんと考えて設計する

参考 :ストレス系のシステムテスト

- ボリュームテスト
  - 大きなデータ、たくさんのデータを与えるテスト
    - 巨大な表のあるページで某Webブラウザがクラッシュ
- ストレージテスト
  - ディスクやメモリなどを残り少ない状態で使うテスト
    - メモリ12Mで動かしたら某OSがブルーサンダー
- 高頻度テスト
  - 短時間にたくさん処理させたり同時に処理させるテスト
    - ある種のパケットを大量に投げると某Webサイトが停止
- ロングランテスト
  - 長時間実行させるテスト
    - 某ターミナルエミュレータは長時間の使用でブルーサンダー

参考 環境系のシステムテスト

- 構成テスト
  - 一緒に利用している他のソフトやハード「から」悪影響を与えられる問題を検出するテスト
    - 某ビデオドライバをインストールするとテスト対象であるWindowsが起動しなくなる
- 両立性テスト
  - 一緒に利用している他のソフトやハード「に」悪影響を与える問題を検出するテスト
    - テスト対象である某Webブラウザをインストールすると某ワープロが起動しなくなる
- 互換性テスト
  - 他のソフトやハードとデータなどの交換をさせるテスト
    - 某ワープロで枠を作ると、異なるバージョンでは位置ズレを起こす

249

参考 評価系のシステムテスト

- 障害対応性テスト
  - 電源を抜くなどの障害を起こして復旧性などを評価するためのテスト
    - 某PCは電源を抜いてスリープさせるとHDDがクラッシュ
- セキュリティテスト
  - 機密保護などの穴を突いてセキュリティを評価するためのテスト
    - 某ウィルスチェッカは、特定のURLで管理機能を奪取できる
- ユーザビリティテスト
  - 操作性や視認性などを評価するためのテスト
    - 某統計ソフトのインストーラは、OKとCancelが逆の場合があり、インストール不能だとユーザに判断されてしまう
- ユーザ操作テスト
  - ユーザが満足しているかを評価するためのテスト

250

Testability の高い開発を意識しよう

- テストが上手になると、そもそもバグのない開発ができるようになる
  - テストを設計すると、構造を見直したり曖昧なところを明確にすることになるので、気付かなかったバグに気付くようになる
  - テスト設計が簡単なプログラムは、設計や実装もシンプルなので、バグが入り込みにくくなる
  - テストが実施しやすいプログラムは、実施できるテストの量も増えるので、バグが見つかりやすい
- テストが容易な製品設計を、テスト容易化設計と呼ぶ
  - Testability DesignやDFT (Design For Test)とも呼ぶ
  - プログラムの内部に依存関係があると、その分だけ組み合わせが必要になるので、きちんとモジュール化を行う
  - プログラムの出力や内部状態の変化が分からないとバグを見逃してしまうので、なるべく簡単に分かるようにしておく

251

まとめ

- テストをする時に意識すべきこと
  - 網羅：モレなくテストする
  - ピンポイント：バグが多いところを狙ってテストする
- テストとは
  - 知恵を絞って設計すべし
  - バグが見つかって初めてテストは成功なのだ
  - いろいろなテスト手法がある
- テストが上手になってくると、そもそもバグのない開発ができるようになる

テストを意識して開発することで
はじめからバグの無いソフトを作ろう

252

ソフトウェアテストの概要
～話題沸騰ポットに対するテストの実践～

大野 晋

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

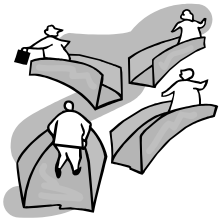
付録．話題沸騰ポットのシミュレーション

テストは難しい

- 通常、プログラム開発の工数の半分以上をテストに費やしている
- しかもその大半をテストの実施ではなく、デバッグに費やす
- テスト時間のほとんどは様々な原因のテストのやり直し
- 体系的にテストを計画し、実践することで工数の大半を削減できる
- しかし、実際には行き当たりばったりにテストが実施され、時間が浪費される

立場によってもテストの内容が違ってくる

- プログラム作成者のテスト:意図した通りかどうかの確認
- テスターのテスト:バグのたたき出し
- 品質保証（QA)のテスト:品質の確認、完成の最終確認



255

プログラム作成者のテスト

設計者としてできたものの動作を保証するためにしていたテストは。

⑴ 共用ルーチン、マクロなど共有物（部品）のテスト
⑵ モジュールレベルの単体テスト（C0,C1：100%はマナー！）
⑶ モジュールを組みたてた組み合わせテスト（機能仕様書、詳細仕様書レベル）
⑷ 全部組み立てた機能テスト（機能の動作確認）
⑸ パラメタを組み合わせた組み合わせ条件のテスト
⑹ 境界条件テスト
⑺ エラーケースのテスト
⑻ 限界テスト
⑼ 他のプログラムなどとの組み合わせテスト
⑽ 使用条件を考えたユーザイメージの機能評価

256

テスターのテスト

デバッグを目的としたテスト
⑴ 全体の機能に対する機能テスト
⑵ パラメタを組み合わせた組み合わせ条件のテスト
⑶ 境界条件テスト
⑷ エラーケースのテスト
⑸ 限界テスト
⑹ 他のプログラムなどとの組み合わせテスト
⑺ 使用条件を考えたユーザイメージの機能評価

257

品質保証（QA）としてのテスト

限られた時間内でプログラムの品質を保証するための検査
⑴ 主力機能をサンプリングした機能テスト（機能の確認）
⑵ パラメタを組み合わせた組み合わせ条件のテスト
⑶ 境界条件テスト
⑷ エラーケースのテスト
⑸ 限界テスト
⑹ 他のプログラムなどとの組み合わせテスト
⑺ 使用条件を考えたユーザイメージの機能評価
で、特に ⑶ ⑷ ⑸ ⑹ ⑺ は大切。
しかし、本当に大切なのは、バグが出た場合の要因分析！

258

さて、テストを始めよう:テストの準備

•まず、計画（戦略）を立てる

•そして、テストの準備をする
⇒例えば、ポットのテストにはこんなものが必要！

1. ポットのシミュレータ
2. ポットへの温度計設置
3. センサの作動状態がわかるような仕掛け
4. ポットのデバッグ環境

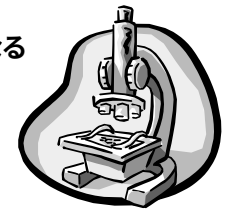
259

テストケースを作る（1）

- 機能の一覧を作成する
- それぞれの機能の同値クラス分析と境界値分析を行う

同値クラスとは…
「2つのテストを実行して同じ結果を期待するとき、2つは同値であるという。」（Kaner他）

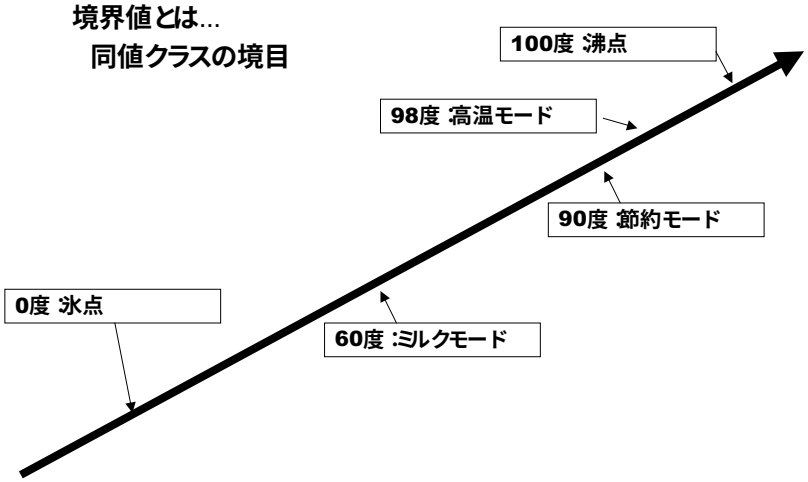
例えば、温度制御のテストを考える場合
•水温：10度、20度、36度、50度は同値クラスになる



260

テストケースを作る (2)

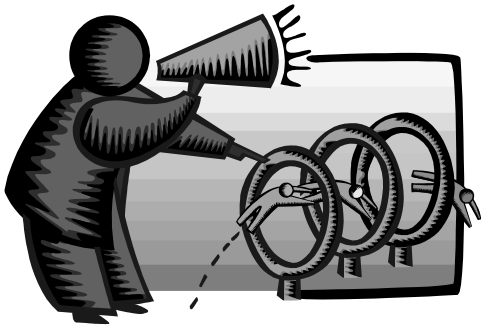
境界値とは...
同値クラスの境目



261

SESSAME CONTENTS 2004

さあ、各モードのテストケースを挙げてみよう



262

SESSAME CONTENTS 2004

テストケースを作る（3）

- 保温モード:高温
  - 100度
  - 98度
  - 95度
  - 80度
  - 0度
- 保温モード:節約
  - 100度
  - 98度
  - 90度
  - 80度
  - 0度
- 保温モード:ミルク
  - 100度
  - 98度
  - 60度
  - 30度
  - 0度

ほんの1例。

どこまで、確認するかでリストアップする温度も増えていく!

テストケースを作る（4）

テストケースを作るには

- 機能を同値分析、境界分析する
- 抽出した機能のテスト値を組み合わせてテストケースを作成する
- 実験計画法の直交表を用いることで、テストケースを最適化できることもある

テストケースを作る（5）

機能から視点を変えてテストケースを考える

- エラー処理
- 性能
- 割り込み
- Ｉ／Ｏ
- 温度
- スイッチの種類とタイミング
- 過去の失敗

など

これらを同値分析、境界値分析などを行ってテストケースを作成する

テストを測る

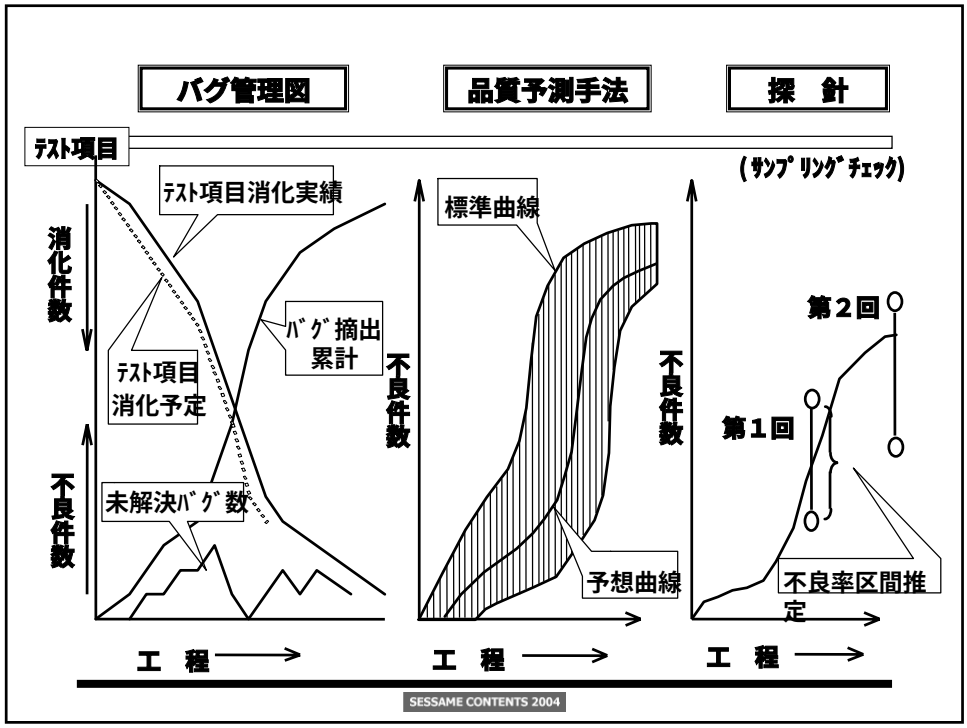
- テストの結果をまとめることでいろいろなことが見えてくる
- テストのまとめ方には

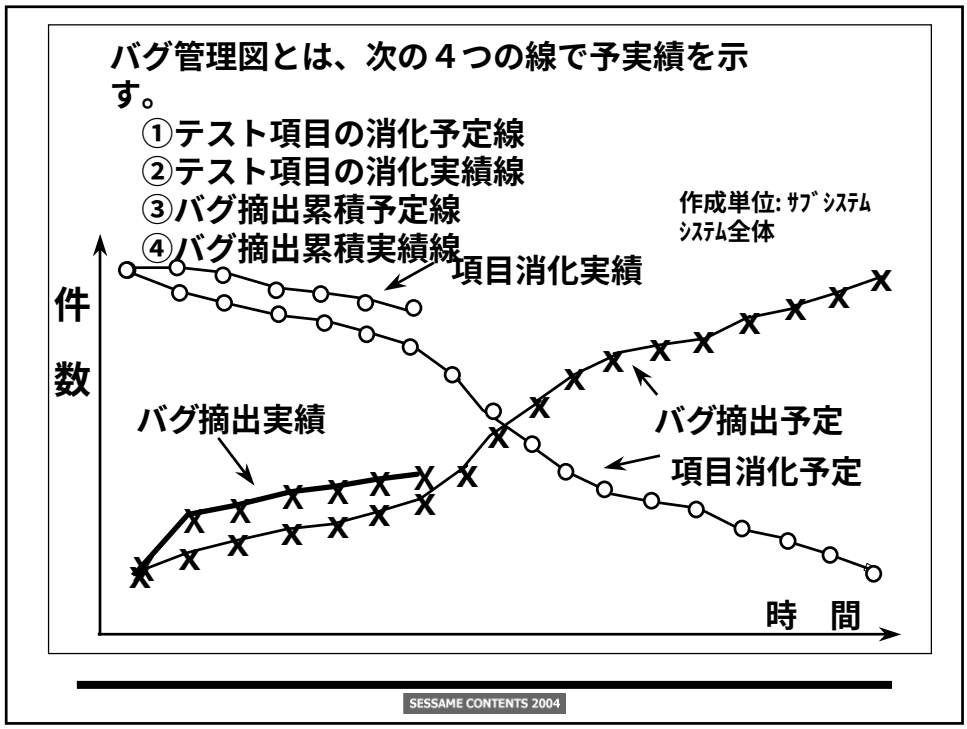
✧ バグを1件ずつ分析するためのレポートを書く

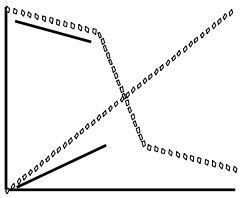
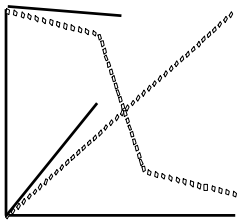
✧ レポートの一覧をもとにバグの傾向を分析する

✧ バグの摘出とテストの進捗からプログラムの品質を分析する

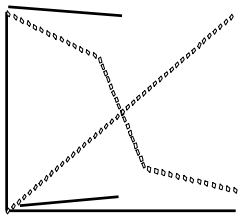
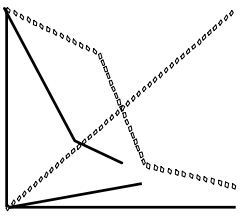
⇒次にテストの進捗傾向の見方の例を示そう


バグ管理図
品質予測手法
探　針
ﾃｽﾄ項目
(ｻﾝﾌﾟﾘﾝｸﾞﾁｪｯｸ)
ﾃｽﾄ項目消化実績
消化件数
ﾊﾞｸﾞ摘出累計
ﾃｽﾄ項目消化予定
不良件数
未解決ﾊﾞｸﾞ数
標準曲線
不良件数
予想曲線
不良件数
第２回
第１回
不良率区間推定
工　程
工　程
工　程
SESSAME CONTENTS 2004


バグ管理図とは、次の４つの線で予実績を示す。
①テスト項目の消化予定線
②テスト項目の消化実績線
③バグ摘出累積予定線
④バグ摘出累積実績線
作成単位: ｻﾌﾞｼｽﾃﾑ
ｼｽﾃﾑ全体
項目消化実績
件数
バグ摘出実績
バグ摘出予定
項目消化予定
時　間
SESSAME CONTENTS 2004

項番	組合せ		バグ管理図	考えられる問題要因
	項目消化	バグ摘出		
1	Ｐ１ 正常	Ｂ１ 正常		特になし ＊架空の報告の可能性あり
2	Ｐ２ 消化停滞	Ｂ３ バグ多発		①バグの作りこみ量が多い ②仕様変更、中途追加でバグを作り込み

バグ管理図の見方（その１）

項番	組合せ		バグ管理図	考えられる問題要因
	項目消化	バグ摘出		
3	Ｐ２ 消化停滞	Ｂ２ 摘出不足		①デバッグ／テスト環境の不備 ②開発要員不足 ③同一テストをしている
4	Ｐ３ 急激消化	Ｂ２ 摘出不足		①テスト項目の質が低い ②テストの実施が一部の機能に偏っている ③品質が良い（？）

バグ管理図の見方（その２）

テスターの視点 (1)

普通のテストはこのように進んでいく。
⇒ただ、バグを良く見つける人と見つけられない人がいる

- 良いテストができるひと
  - 気のつく人？
  - 気配りのできる人？
  - よく仕様を知っている人？
  - 品質保証部の人？
  - お客さん？
  - 心配性の人？

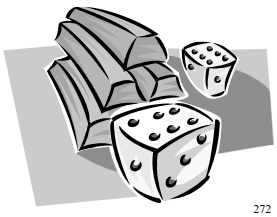
271

テスターの視点 (2)

単純に「目に見える仕様」をテストしてもバグはなかなか見つからない！

⇒例えば、「話題沸騰ポット」をテストするとしたら、どのようなポイントをテストするだろうか？

仕様書からこんな点をテストしたいと思うポイントを挙げてみよう！



272

テスターの視点 (3)

(1)気圧が低く、沸点が下がっている場合
(2)高温の水を沸かす
(3)低温の水 (氷)を沸かす
(4)水量センサーの誤作動 :空瓶センサーがoff
(5)センサー面での水のゆれ
(6)蓋を開けるタイミング :ヒーターの切り替えと蓋の開け閉め
(7)加熱中の水量変化
(8)水以外の液体
(9)外気温の影響
(10)ボタンの同時押し
(11)周波数による違い
⇒本来、考えられるべき事項 (文章にならない仕様)
⇒良いテストとは、文章以外の仕様を見つけ、それを確かめること
⇒特に繊細な制御をするプログラムの場合、制御に抜けが生じやすい

273

テスターの視点 (4)

- 過去のバグを見直すことで、考えておくべき視点を知ることができる
- バグレポートは知識の宝庫！
- 蓄積がない場合には、書籍の付録のバグ一覧が有効！
- 「書いてある仕様」でプログラムが間違っていることは本当に少ない！
- 「書いていない仕様」で問題を抱えていることは多いが、バグなのか、その通りで良いのか、の判断は難しい！

274

まとめ

- ソフトウェアのテストのやり方は単一のパターンではなく、目的によってやり方が異なる
- テストの視点とは、設計者と同じ視点である

テスターの言い分：

- 良いテスターの視点を設計の早い時期に役立てることで効率よく、品質の良いプログラムが出来上がる
- だから、テストでバグ出しをするよりもレビューや机上見直しが効率的！

275

SESSAME CONTENTS 2004

余白）

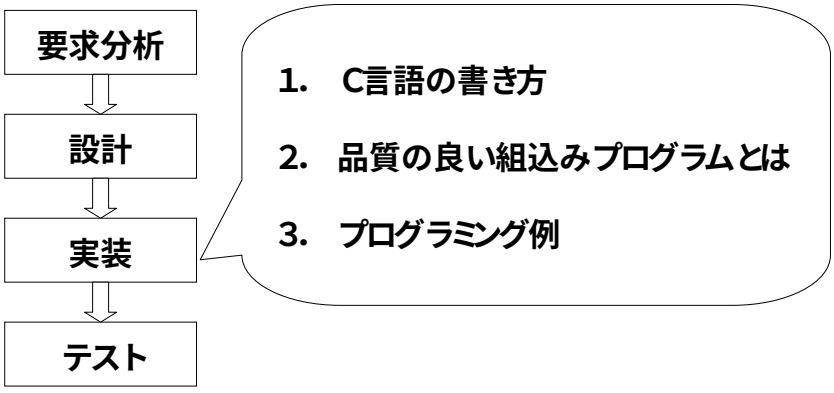
276

SESSAME CONTENTS 2004

プログラミング実習への説明

上原 慶子／三宅 貴章

アジェンダ



1．C言語の書き方

1.1 C言語の特徴

1.2 演算子の優先順位

1.3 if文に注意！

1.4 わかりやすいプログラムの文体

1.1 C言語の特徴

1. 関数を基本とした構造化プログラミング

main｛｝関数がトップ。プログラムは関数の集合体

if else文、while文、for文・・構造化に適した制御文

2. 自由度と簡潔性

変数名などの単語の綴り内以外であれば、空白・改行は自由

豊富な演算子により、さまざまな書き方が可能

例）

```
i= i+ 1;    i= 1;    i++;
```

1.1 C言語の特徴

3. 不定（コンパイラ依存）

同じソースコードであってもターゲットによって動作が異なる場合有。
コンパイラによってコード生成も異なる。
→コーディング規約が必要。

4. 自動変数と静的変数

自動変数：スタック上に領域を確保。宣言した関数内で一時的に有効
静的変数：メモリ上に領域を確保。関数実行後も値を保持

5. アセンブリ言語に近い

ビット演算子やポインタなど、アセンブリ言語に近い記述が可能
オブジェクトコードは他の高級言語に比べコンパクト

281

1.2 演算子の優先順位

演算子には、優先順位が決められている。
括弧を活用して誤解を防ごう。

例） if（byte0 & 0x08 == 0）{・・・

演算子 & と== では、== の優先順位が高い。
この記述の場合、0x08 == 0の比較結果（偽：0）とbyte0を
AND演算するため、条件式は常に（偽：0）となってしまう。

正しくは、・・・
if（（byte0 & 0x08）== 0）{・・・

282

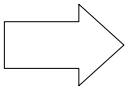
1.3 if文に注意

if文は障害を起こしやすい

よくある誤り

1. =と==と!=との間違い
2. &&と&, ||と|, ==と=との間違い
3. 演算子の優先順位の勘違い（前ページで紹介）
4. ifとelseの対応

```
if(k1 == 0)                    if(k1 == 0)
    if(k2 == 0)                    if(k2 == 0)
        i = i + 1;                     i = i + 1;
else                           else
    i = i + 2;                     i = i + 2;
```



283

SESSAME CONTENTS 2004

1.4 わかりやすいプログラムの文体

1. わかりやすい変数名・関数名

外部変数と内部変数の違いが変数名からわかる
外部変数は名前から使用目的がわかる／内部変数は短く
関数名は、動詞＋名詞のスタイルで！

2. 構造がわかるインデント（繰り返しや判断など）

```
for(i = 0; i < 10; i++);       for(i = 0; i < 10; i++)
if(k2 == 0)                        ;
i = i + 1;                     if(k2 == 0)
                                   i = i + 1;
```

284

SESSAME CONTENTS 2004

1.4 わかりやすいプログラムの文体

3. 括弧の活用 と 空白

括弧を使って、優先順位による障害を防ぐ

[や演算子の前後に空白を入れて読みやすく

4. 式は自然な書き方で (は判りにくい)

```
if( !(k > 10) )   ⇒   if(k <= 10)
```

5. 一貫性をもたせる

気まぐれな記述は他人にわかりにくい ＝ バグのもと

慣用表現で記述の統一を図る

```
for(i=0; i<n;)                for(i=0; i<n; i++)
    data[i++] = 0;     ⇒          data[i] = 0;
```


1.4 わかりやすいプログラムの文体

6. 数字の記述は避ける

その意味 根拠がわかる名前をつける

sizeof演算子の活用 ・・配列サイズの取得時など

7. コメントのつけ方

コードを見てわかるコメントは不要

変数や定数、関数などには積極的にコメントをつける

変数 定数 意味、とる値など

関数 処理の概要、引数、戻り値など

ソースとの相違が無いように・・・

2．よい組込みプログラムとは

2.1 品質のよいプログラムとは

2.2 よいプログラムを作る注意点

2.3 McCabeの複雑度

2.1 品質のよいプログラム

- ISO/IEC9126の品質特性
  ★がプログラミング時の注意点

機能性
★効率性・・実行時間が短く、資源の使用が少ない
★信頼性・・プログラムの障害発生率が低い
★保守性・・障害修正や機能追加変更が容易
使用性
★移植性・・他の環境でも動作可能

2.2 品質のよいプログラムにするためには

・おもな注意点

効率性・書き方によるオブジェクトサイズの違いを習得する。(コンパイラにも依存)
信頼性・コーディングミスを起しにくい文体
保守性・わかりやすいプログラム
移植性・標準的な書き方とコンパイラ依存を極力排除した記述

2.3 McCabeの複雑度

- 判定文が多いとプログラムの理解に時間がかかる。
- プログラムの複雑度とはプログラムがどの程度複雑でわかりにくいか? を数値で表す.
- プログラムの複雑度の計量法でよく使われているのが、McCabeの閉路複雑度

McCabeの複雑度=判定数 + 1

McCabeの閉路複雑度 <=20 わかりやすい
McCabeの閉路複雑度 >20 理解に時間がかかる
McCabeの閉路複雑度 >50 作成者以外にはほとんど理解不可能
複雑度を下げるために関数分割をするのはよく考えてから

3．プログラミング例

3.1 状態遷移マトリクス例

3.2 プログラム例その1

3.3 プログラム例その2

3.4 プログラム例の特徴

3.1 状態遷移マトリクス例

状態番号	状態／事象	保温設定ボタン	沸騰ボタン	初期沸騰検出	沸騰終了
1	水位不足	変化なし	変化なし	変化なし	×
2	加熱途中	→7	変化なし	→3	×
3	初期沸騰	→7	→4	×	→4
4	高温保温	→7	→5	×	×
5	高温沸騰前	→8	変化なし	→6	×
6	高温沸騰中	→9	→4	×	→4
7	節約保温	→10	→8	×	×
8	節約沸騰前	→11	変化なし	→9	×
9	節約沸騰中	→12	→7	×	→7
10	ミルク保温	→4	→11	×	×
11	ミルク沸騰前	→5	変化なし	→12	×
12	ミルク沸騰中	→6	→10	×	→10

3.2 プログラム例その1(switch文)

各状態と事象をそれぞれ, 判断文で判定し, 関数を呼び出す

```
switch (status){
   case  RETAINING :  /* 高温保温中 */
      if ((cur_switch & BOIL_SWITCH) != 0) {/* 沸騰*/
         status = BOILING; /* 沸騰モードへ遷移 */
         prboil( ); /* 沸騰処理呼び出し*/
         }
      else  if ((cur_switch & RETAIN_SWITCH) != 0){/* 保温 */
         status = ECONOMY; /* 節約モードへ遷移 */
         preco( ); /* 節約処理呼び出し*/
         }
   •••
```


3.3 プログラム例その2(関数ポインタの定義)

各状態と事象をそれぞれ, 二次元の配列で考え, 関数ポインタを使って呼び出す

```
struct  functbl {
   void   (*fp)(void); /* 呼びだす処理関数のアドレス */
   mode_type  status; /* 遷移する状態 */
};
mode_type   status;
mode_type   nextstatus;
#define   MAXSTATUS  12
#define   MAXEVENT    5
struct functbl datatbl[MAXSTATUS][MAXEVENT] = {
                { {prboil , BOILING} , {preco, ECONOMY},   •••}
};
```

3.3 プログラム例その2(関数ポインタによる呼び出し)

各状態と事象をそれぞれ, 二次元の配列で考え, 関数ポインタを使って呼び出す

```
void   (*callp)(void) {
   for (; ;)  {  /* 無限ループ */
      /*イベント待ち処理およびevent領域へのイベント番号の設定 省略 */
      ;
      callp = functbl[status][event].fp; /* 呼び出す関数の設定 */
      nextstatus = datatbl[status][event].status; /* 次の状態の設定*/
      (*callp)();  /* 状態とイベントに対応した処理の呼び出し*/
      status = nextstatus; /* 状態の変更 */
   }
}
```

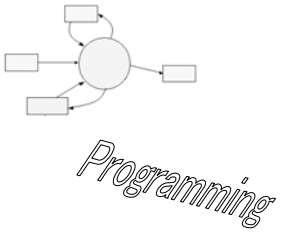

3.4 プログラム例の特徴

1.switch-case文
- 小さなマトリクス向き
- 可読性高いが、モレ・ヌケのリスクあり。

2. 関数ポインタ
- 大規模なマトリクスに有効
- 状態・条件にかかわらず処理時間が一定
- 状態・イベントの追加や遷移仕様の変更が容易
- 変数によってアドレッシングするため、暴走のリスクあり

プログラミング　実習

森　孝夫



演習の進め方

演習①

- 鹿威しIrレシーバーサブシステム開発仕様書「4. プロセス仕様」を見て、データ受信プロセスの状態遷移マトリクスを作成しましょう
  - 仕様書を理解し、吟味しましょう
  - マトリクスによる整理を行いましょう

演習②

- 作成した状態遷移マトリクスから、鹿威しIrレシーバーサブシステムのプログラムを作成しましょう
  - 状態モデルを実現する整理されたコードを記述しましょう

演習②のフレームワーク

```
/* 以下の状態定義、イベント定義、処理が定義されているものとします */

typedef enum {
    WRITE, IDLE, LOCK, STATE_NUM
} mode_type;
typedef enum {
    EVENT_WRITE, EVENT_START, EVENT_RECEIVE_ERROR,
    EVENT_LEAD_DETECT, EVENT_DATA_READY, IR_EVENT_NUM
} event_type;

void dummy(void) { }
/* エントリ処理 */
void Idle(void) { /****/ }
void Write(void) { /****/ }
void Lock(void) { /****/ }

/* エントリ処理以外にも、記述する処理が必要かもしれません */
```


演習②のフレームワーク

```
/* 関数ポインタを用いる時は、ここにテーブルを記述して下さい */
/* この時、テーブルの縦と横がテーブルと一致するように工夫しましょう */

int main(void)
{
    int state = STATE_WRITE;

    for (;;) {
        event = ******;  /* ここでイベントが確定します */
                         /* 同時イベントの発生は ... 後で考えましょう */
        if (event != IR_EVENT_NUM) {

            /* 本日はこの中の状態遷移プログラム記述に集中して下さい */

        }
    }
}
```

鹿威しIrレシーバーサブシステム開発仕様書 目次

1. 目的

このドキュメントは、鹿威しシステムを赤外線リモコンで制御するために必要な、Irレシーバサブシステムの仕様を示します。

本サブシステムの役割、位置づけについては、鹿威しシステム全体のDFDなどをご確認下さい。

2. 機能要件

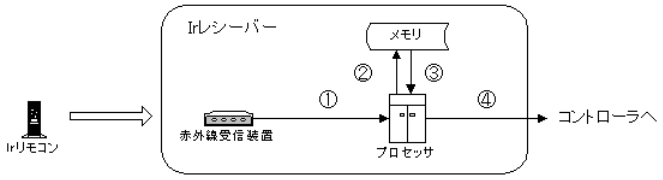
今回設計するIrレシーバサブシステムの機能要件は以下の通りです。

- 本仕様書 「通信データフォーマット」で示す、Irリモコンからの信号を受信、解釈する。
- 解釈後、コントローラに対して命令コードを出力する。
- 将来の利用範囲拡大を考慮する。

3. 概要

3.1 全体構成

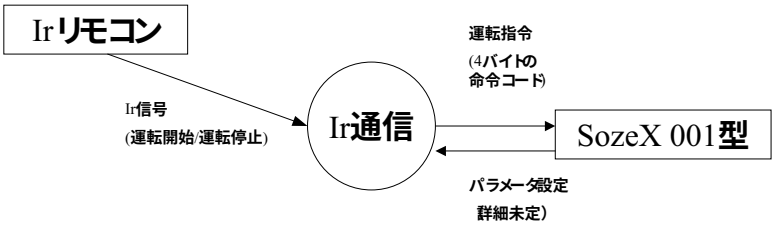
Irレシーバサブシステムの全体構成を以下に示します。



プロセッサにはSFRを持つCPUを使用します。赤外線受信装置からの信号は、プロセッサのSFRポートに接続します。
RTOSは使用しません。

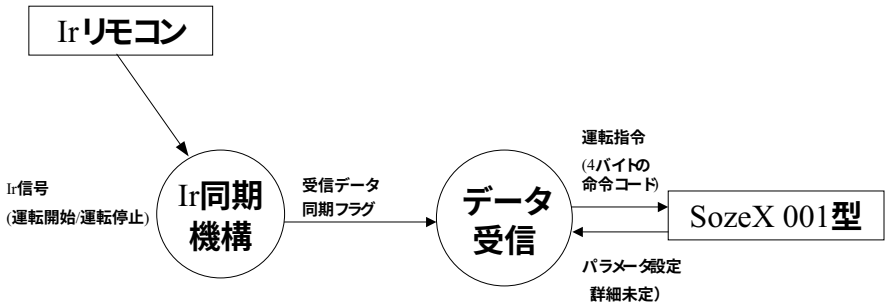
3.2 モデル – 1

将来の利用範囲拡大を考慮し、ホストであるSozeX 001型からのパラメータ設定機能を追加し、モデル化します。



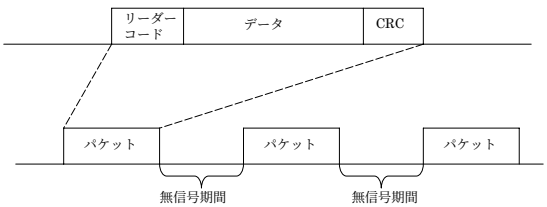
3.2 モデル – 2

Ir信号の同期を行い信号をデータ化する部分と、そのデータを解釈して
SozeXに適切な通知を行う部分に分離してモデル化します。



3.3 通信データフォーマット

Irリモコンは、ユーザーの操作時に以下のデータを送付します。
したがって、これがそのまま受信データフォーマットとなります。



リーダーコードは、Irシリアル通信におけるビット同期を取るための信号です。データは4バイト、その後に1バイトのCRCコードが続きます。
リーダーコード、データ、CRCの一塊を「パケット」と呼びます。パケットとパケットの間には、必ず一定時間以上の無信号期間を設けます。

4. プロセス仕様

4.1 動作と状態の考察

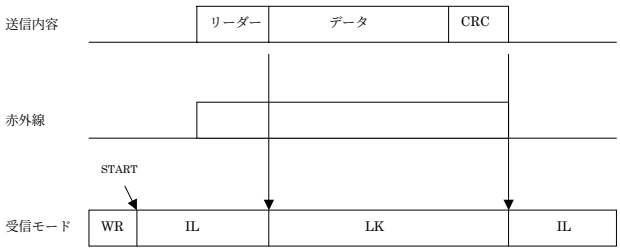
4.2 受信状態遷移図

4.1 動作と状態の考察 - 1

Irサブシステムの状態は、通常状態と、受信機のパラメータを設定する状態に大別できます。
また、通常状態は、データ受信をしている状態と、データを待っている状態の2つに大別できます。
したがって、Irサブシステムは以下の3モードで動作します。

名称	略称	モード
WRITE	WR	受信機のパラメータ設定モード
IDLE	I	リーダーコード待ちモード
LOCK	LK	データ受信モード

4.1 動作と状態の考察 - 2



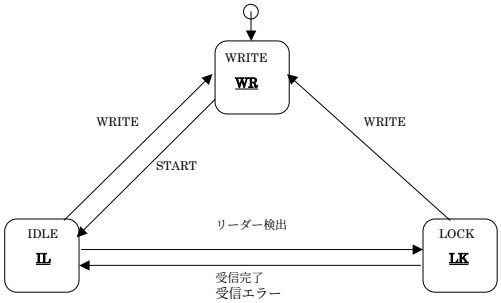
STARTとは、SozeX 001からの開始OKの通知であるとします。

WRモードは、SozeX 001からの命令待ち受け状態です。
今開発では、受信したデータを命令コードと見なし、そのままSozeX 001型に送信します。

311

4.2 受信状態遷移図

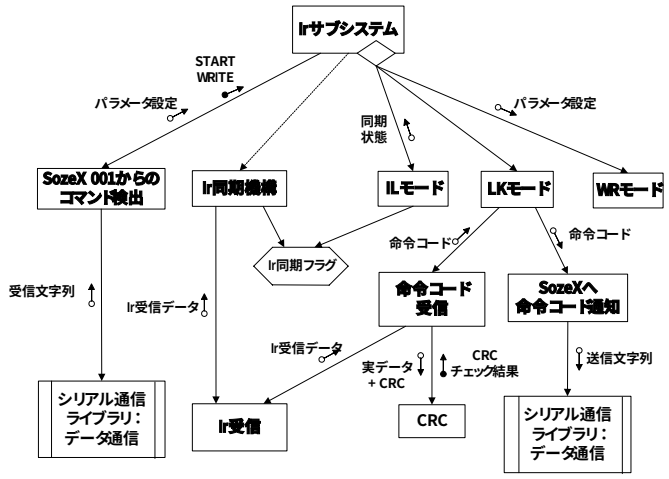
「データ受信」モジュールの状態遷移図を以下に示します。



データ受信モジュールは、矢印上に記載されたイベントを受けて状態を切り替えます。
WRITE、STARTは、SozeX 001からのイベント、
それ以外は「データ受信」モジュールが自ら発行するセルフイベントです。

312

5. モジュール構成図



313

6. 開発用ライブラリ仕様

6. 1 SozeX 001型への通知

6. 2 SozeX 001型コマンド検出

6. 3 データ受信開始準備

6. 4 Ir同期、受信イベント検出

6. 5 命令コード検出

314

6.1 SozeX 001型への通知

int SendMsgToSozeX001 (const char inst[])		
機能		
	SozeX 001型に命令コードを送信する。	
引数		
	inst	命令(instruction)コード
戻り値		
	0	成功
	負数	エラーコード
ヘッダ	SozeX_IF.h	

315

6.2 SozeX 001型コマンド検出

int GetHostCmd (void)		
機能		
	SozeX 001からのイベントを取得する。	
引数		
	なし	
戻り値		
	0	コマンド検出できず
	1	WRITE
	2	START
ヘッダ	SozeX_IF.h	

316

6.3 データ受信開始準備

void InitDataSynchronizer(char *dataBuf, int receiveSize, int *leadDetectF, int *receiveErrorF, int *dataReadyF)		
機能		
	Ir信号の受信開始準備を行う。	
引数		
	dataBuf	データ、CRCを受信するバッファ
	receiveSize	受信するデータのビット長指定
	leadDetectF	リーダーコードを受信したかどうかのフラグへの参照
	receiveErrorF	受信エラーフラグへの参照
	dataReadyF	データ受信完了フラグへの参照
戻り値		
	なし	
ヘッダ	IrSync.h	

317

6.4 Ir同期、受信イベント検出

void GetIrEvent(int *leadDetectF, int *dataReadyF, int *receiveErrorF);		
機能		
	Ir同期、受信イベントを検出。	
引数		
	leadDetectF	リーダーコードを受信したかどうかのフラグへの参照
	dataReadyF	データが揃い、CRCチェックまで完了したかどうかのフラグ
	receiveErrorF	データ受信中に受信エラーが発生したかどうかのフラグ
戻り値		
	なし	
ヘッダ	IrSync.h	

318

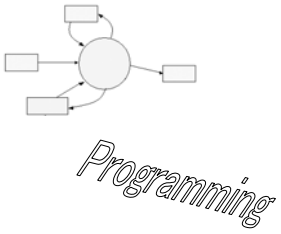
6.5 命令コード検出

void ReceiveIrCode(char *dataBuf)		
機能		
	Irから受信、CRCチェックが完了した命令コードを受信する。	
引数		
	dataBuf	データ、CRCを受信するバッファ
戻り値		
	なし	
ヘッダ	IrSync.h	

余白）

プログラミング　実習/回答と補足説明

森　孝夫



演習①の回答　状態遷移マトリクスの例

→ 遷移
／ 無視
× ありえない

状態 / イベント	WRITE (受信機のパラメータ設定モード)	LOCK (データ受信モード)	IDLE (リーダーコード待ちモード)
WRITE (SozeXから)	／	→WRITE	→WRITE
START (SozeXから)	→IDLE	／	／
リーダー検出 (セルフイベント)	×	×	→LOCK
受信完了 (セルフイベント)	×	→IDLE	×
受信エラー (セルフイベント)	×	→IDLE	×

演習②の回答例(1)

```
typedef enum {
    WRITE, IDLE, LOCK, STATE_NUM
} mode_type;
typedef enum {
    EVENT_WRITE, EVENT_START, EVENT_RECEIVE_ERROR,
    EVENT_LEAD_DETECT, EVENT_DATA_READY, IR_EVENT_NUM
} event_type;

void dummy(void) { }
void Idle(void) { /****/ }
void Write(void) { /****/ }
void Lock(void) { /****/ }
```


演習②の回答例(2)

```
struct  functbl {
    void   (*fp)(void); /* 呼びだす処理関数のアドレス */
    mode_type  status; /* 遷移する状態 */
};
/* state transition table */
struct functbl stt[IR_EVENT_NUM][STATE_NUM] = {
        /*  WRITE */     /* LOCK */      /* IDLE */
    { { dummy,WRITE }, {Write, WRITE}, {Write, WRITE} },
    { { Idle, IDLE  }, {dummy, LOCK }, {dummy, IDLE } },
    { { dummy,WRITE }, {dummy, LOCK }, {Lock,  LOCK } },
    { { dummy,WRITE }, {Idle,  IDLE }, {dummy, IDLE } },
    { { dummy,WRITE }, {Idle,  IDLE }, {dummy, IDLE } }
};
```

演習②の回答例(3)

```
int main(void)
{
    mode_type  status = WRITE, nextstatus = WRITE;
    event_type event = IR_EVENT_NUM;
    void (*callp)(void); /* 呼びだす処理関数のアドレス */

    for (;;) {
        event = *****;  /* ここでイベントが確定します */
        if (event != IR_EVENT_NUM) {
            callp = stt[event][status].fp;
            nextstatus = stt[event][status].status;
            (*callp)();
            status = nextstatus; /* 状態の変更 */
        }
    }
}
```


余白）

ソフトウェアテスト実習

西 康晴

１．SESSAMEの紹介およびコースの概要
２．開発課題と失敗事例の解説
３．組込み向け構造化分析の例・設計の概要(1)
４．組込み向け構造化分析の例・設計の概要(2)
　実習/回答と補足説明
５．組込み向け構造化設計(1)
６．組込み向け構造化設計(2) 実習/回答と補足説明

７．プログラミング　組込み用語基礎知識
８．ソフトウェアテストの概要
９．プログラミング実習への説明
１０．プログラミング　実習
１１．プログラミング　実習の回答と補足説明
１２．ソフトウェアテスト　実習
１３．ソフトウェアテスト　実習/回答と補足説明

付録．話題沸騰ポットのシミュレーション

鹿威しIrレシーバーサブシステムのテストを設計しましょう

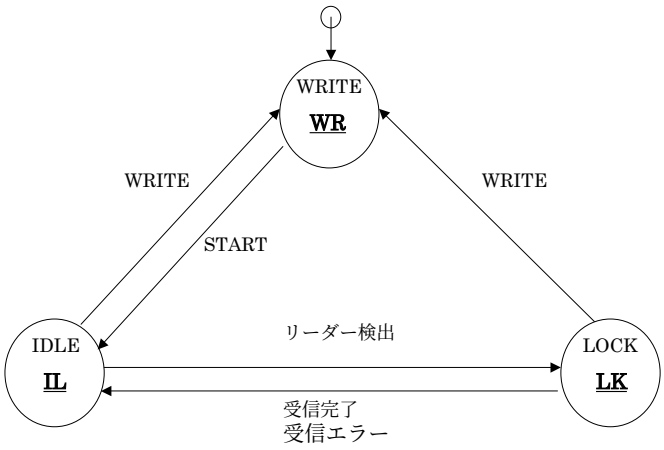
演習①

- 鹿威しIrレシーバーサブシステム開発仕様書を見てテストを設計してみましょう
  - 通信データの境界値テストを設計しよう
  - 状態遷移設計に関するテストを設計しよう

演習②

- 鹿威しIrレシーバーサブシステムのプログラム(int main(void))のテストを設計してみましょう
  - 制御パステストを設計しよう

Irレシーバーサブシステムの状態遷移図



329

SESSAME CONTENTS 2004

Irレシーバーサブシステムのソースコード

```
          for (;;) {
              /* determine event */
0             event = IR_EVENT_NUM;
              hostCmd = GetHostCmd();
1             if (hostCmd == WRITE)
2                 event = EVENT_WRITE;
3             else if (hostCmd == START)
4                 event = EVENT_START;
              else {
5                 GetIrEvent(&leadDetectF, &dataReadyF, &receiveErrorF);
                  if (receiveErrorF)
6                     event = EVENT_RECEIVE_ERROR;
7                 else if (leadDetectF)
8                     event = EVENT_LEAD_DETECT;
9                 else if (dataReadyF)
10                    event = EVENT_DATA_READY;
              }

11            if (event != IR_EVENT_NUM) {
                  callp     = dataTbl[current_mode][event].fp;
                  next_mode = dataTbl[current_mode][event].mode;
12
                  (*callp)();

                  current_mode = next_mode;
              }
13        }
```

330

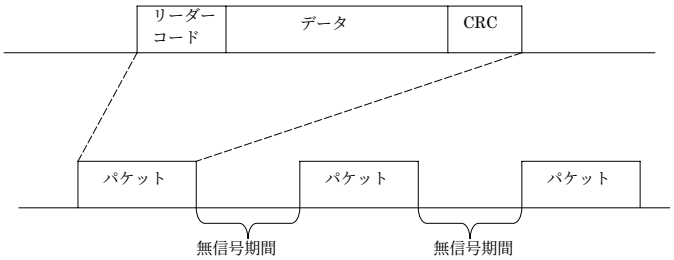
SESSAME CONTENTS 2004

ソフトウェアテスト実習/回答と補足説明

西 康晴

通信データフォーマット

- パケットと無信号期間がある
  - リーダーコードは不明
  - データは 4バイト
  - CRCは 1バイト
  - 無信号期間は不明



境界値テストの設計

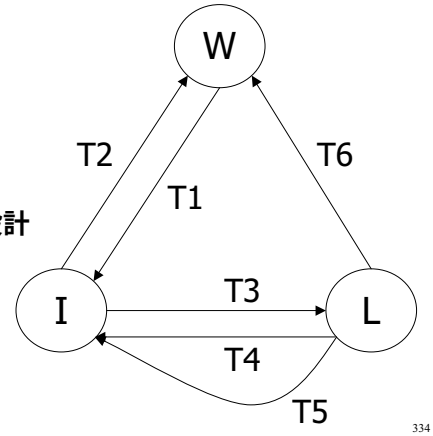
- データ部の境界値テスト
  - 4バイトのデータ部を送る→正常
  - 4バイト以上のデータ部を送る→はみ出てCRCエラーとなる
    - 実際にはバッファオーバーフローが起きるかもしれない
  - 4バイト以下のデータ部を送る→足りずにCRCエラーとなる
- CRC部の境界値テスト
  - 1バイトのCRC部を送る→正常
  - 1バイト以上のCRC部を送る→無視される
    - 実際にはバッファオーバーフローが起きるかもしれない
  - 1バイト以下のCRC部を送る→CRCエラーとなる
- リーダーコード／無信号期間の境界値テスト
  - 境界値が分からない場合は問い合わせる
    - 分からない仕様はバグの温床

333

SESSAME CONTENTS 2004

状態遷移パステストの設計

- 遷移リンクの抽出
  - T1: W→I
  - T2: I→W
  - T3: I→L
  - T4: L→I 受信完了）
  - T5: I→L 受信エラー）
  - T6: L→W
- リンク網羅での状態遷移パスの設計
  - P1: T1→T2
  - P2: T1→T3→T4
  - P3: T1→T3→T5
  - P4: T1→T3→T6

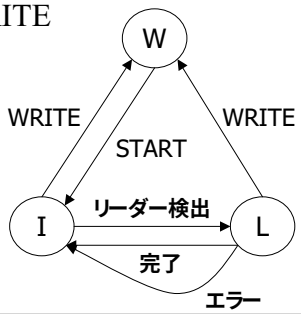


334

SESSAME CONTENTS 2004

状態遷移パステストの設計

- イベントに直してテスト項目を記述する
  - P1: START→WRITE
  - P2: START→リーダー検出→受信完了
  - P3: START→リーダー検出→受信エラー
  - P4: START→リーダー検出→WRITE
- 期待結果を記述する
  - P1: WRITE状態
  - P2: データを送出してIDLE状態
  - P3: IDLE状態
  - P4: WRITE状態



335

状態遷移マトリクス

→ 遷移
／ 無視
× ありえない

状態 / イベント	WRITE (受信機のパラメータ設定モード)	LOCK (データ受信モード)	IDLE (リーダーコード待ちモード)
WRITE (ホストから)	／	→WRITE	→WRITE
START (ホストから)	→IDLE	／	／
リーダー検出 (セルフイベント)	×	×	→LOCK
受信完了 (セルフイベント)	×	→IDLE	×
受信エラー (セルフイベント)	×	→IDLE	×

336

状態遷移マトリクステストの設計

- 各状態でのイベントをそれぞれテストする
  - 無視されるイベントも確認する
    - 発生しうるイベントは 6つ
    - 無視されるイベントは 3つ
  - ありえないイベントはデザインレビューで確認する
    - ありえないイベントは 6つ
  - テストドキュメントを作成するに越したことはないが、経験が無いなら状態遷移マトリクスにOKを書く
    - NGはきちんと不具合報告書で報告すること
- イベントの排他発生を確認する
  - WRITE / START イベントは排他
  - リーダー検出 / 受信完了 / 受信エラー イベントは同時に発生する可能性がある
    - 同時に発生した時にフェイルセーフになるかどうかを確認する必要がある

OK結果の記録も重要

337

状態遷移マトリクステストの設計

- 状態遷移マトリクスからテスト項目を設計する
  - 発生しうるイベントのテスト項目を設計する
    - WRITE状態でSTARTイベントを起こし、IDLE状態に遷移する
    - LOCK状態でWRITEイベントを起こし、WRITE状態に遷移する
    - LOCK状態で受信完了イベントを起こし、データを送出してIDLE状態に遷移する
    - LOCK状態で受信エラーイベントを起こし、IDLE状態に遷移する
    - IDLE状態でWRITEイベントを起こし、WRITE状態に遷移する
    - IDLE状態でリーダー検出イベントを起こし、LOCK状態に遷移する
  - 無視されるイベントのテスト項目を設計する
    - WRITE状態でWRITEイベントを起こし、無視されることを確認する
    - LOCK状態でSTARTイベントを起こし、無視されることを確認する
    - IDLE状態でSTARTイベントを起こし、無視されることを確認する
  - 同時に発生するイベントのテスト項目を設計する
    - 各状態でリーダー検出、受信完了、受信エラーイベントを起こし、受信エラー動作が起こる 受信完了動作が起こらない）ことを確認する

338

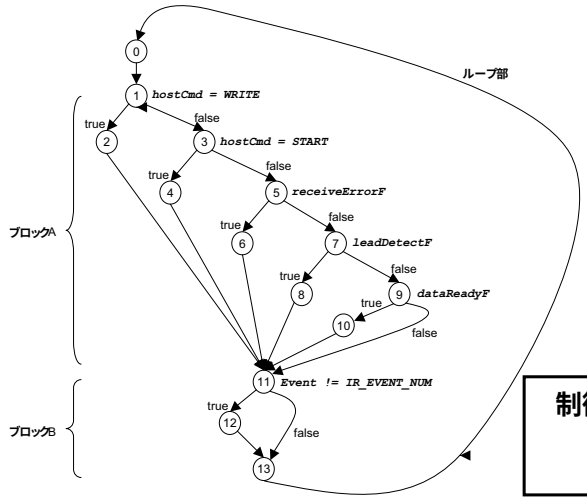
Irレシーバーサブシステムのソースコード

```
          for (;;) {
              /* determine event */
0             event = IR_EVENT_NUM;
              hostCmd = GetHostCmd();
1             if (hostCmd == WRITE)
2                 event = EVENT_WRITE;
3             else if (hostCmd == START)
4                 event = EVENT_START;
              else {
5                 GetIrEvent(&leadDetectF, &dataReadyF, &receiveErrorF);
                  if (receiveErrorF)
6                     event = EVENT_RECEIVE_ERROR;
7                 else if (leadDetectF)
8                     event = EVENT_LEAD_DETECT;
9                 else if (dataReadyF)
10                    event = EVENT_DATA_READY;
              }

11            if (event != IR_EVENT_NUM) {
                  callp     = dataTbl[current_mode][event].fp;
                  next_mode = dataTbl[current_mode][event].mode;
12
                  (*callp)();

                  current_mode = next_mode;
              }
13        }
```

339

SESSAME CONTENTS 2004

Irサブシステムの制御フローグラフ



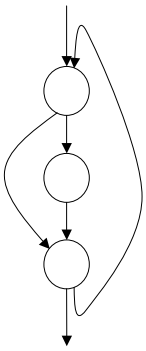
制御フローグラフには処理の詳細は記述しない

340

SESSAME CONTENTS 2004

なぜリンク網羅が必要かを理解する

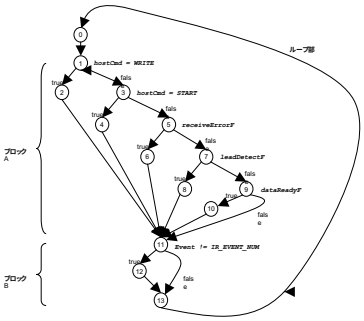
- ノード網羅は ◯ しか網羅しないのでバグ見逃しが起こる
  - 9→11／11→13のような「飛び越し」をテストできない
  - ループもテストできない
- リンク網羅は ⟶ を網羅する
  - 飛び越しもループもテストできる
  - パスもテストできる
- ループをテストするときは「0回、」1回、2回、たくさん、のテストをする
  - 要はリンク網羅である



341

3つの部分に分けてパスを抽出する

- ブロックAの部分パス
  - A1: 0→1→2→11
  - A2: 0→1→3→4→11
  - A3: 0→1→3→5→6→11
  - A4: 0→1→3→5→7→8→11
  - A5: 0→1→3→5→7→9→10→11
  - A6: 0→1→3→5→7→9→11
- ブロックBの部分パス
  - B1: 11→12→13
  - B2: 11→13
- ループ部の部分パス
  - L0: 「ループ0回」無し
  - L1: 0→1…→13 「1回」
  - L2: 0→1…→13→0→1…13 「2回」
  - L∞: 0→1…→13→0→1…13→・・・→0→1…13→0→1…13



342

組み合わせを検討する

- ブロックAとブロックBに依存関係があるか
  - A1～A5は必ずB1に、A6は必ずB2に分岐する
- ブロックA/Bとループ部に依存関係があるか
  - ブロックの履歴に動作が依存しないか
    - 過去のブロックA（B）とブロックB（A）
    - 過去の自分自身
  - 今回は依存しないと想定する
- デザインレビューや
  コードインスペクションを行う
  - 保守や拡張、後継機種開発の時にも気を付ける

テストパスの設計

- L1のパス:
  L1-1: A1→B1
  L1-2: A2→B1
  L1-3: A3→B1
  L1-4: A4→B1
  L1-5: A5→B1
  L1-6: A6→B2
- L2:
  - 履歴による依存関係はないので、
    L1-1～L1-6を適宜組み合わせればよい
    - 依存関係がある「疑い」がある場合は
      6つのパスを全て組み合わせなければならない

設計したテストパス

A1→B1（WRITE）
A2→B1（START）
A3→B1（受信エラー）
A4→B1（リーダー検出）
A5→B1（受信完了）
A6→B2→A1→B1
（イベントなし→WRITE）

テスト実施結果の確認方法

- ICE／デバッガでテストする場合
  - ループごとに実行を停止させて
    変数 event とcurrent_mode（next_mode）をウォッチする
  - printfなどアサーションコードを入れる
- モジュールの引数／返値でテストする場合
  - LOCK 状態で dataReadyF を発生させて
    テスト結果を確認する必要がある
    - こうなるとコードからテストケースを起こすのは
      至難の業となる
    - コードが設計と同じ構造をしていれば、
      状態遷移パスで考えることができて楽である
- モジュールをブラックボックスにしないように気を配る
  - テスト結果を観測しやすい設計や実装を意識する

ソフトウェアテスト実習のまとめ

- それぞれのテストの特徴を掴もう
  - 境界値テストでは、通信データのテストが設計できた
  - 状態遷移パステストでは、イベントの順序のテストが楽に設計できた
  - 状態遷移マトリクステストでは、同時イベントのテストが設計できた
  - 制御パステストでは「イベントなし」のテストが設計できた
- テストの設計と同時に、仕様レビュー／デザインレビュー／
  コードレビューを行っていることを肝に銘ずる
  - 期待結果の導出と、不明な期待結果の問い合わせを忘れない
  - テスト設計の時点で、仕様や設計、実装のバグを見つけよう
  - テストの組み合わせを減らす設計やレビューを心がける
  - テスト結果を観測しやすい設計や実装を意識しよう

**テストが上手になってくると、
そもそもバグのない開発が
できるようになる**

付録: 話題沸騰ポットのシミュレーション

山崎 辰雄

アジェンダ

1. なぜ、ソフトウエア・シミュレータ?
2. どこまでシミュレートするの?
3. どうなっているの?
4. なにがいいの?
5. 気をつけなければならないこと
6. デモンストレーション
7. つぎはどこへ?
8. まとめ

なぜ、ソフトウエア・シミュレータ? (1)

SESSAMEでポットを作ることになりました。
はじめは、LEGO　マインド・ストームを使うつもりでした。

- 理由
  - 開発環境がそろっている
  - 比較的安価 (約4万円)
  - くじけても、子供のおもちゃになる
- 買ってはみたけれど
  - オプションで、温度センサはあるけど、ヒーターがない
  - 水にはつけられないよね
  - やっぱり、お札が飛んでいくのはイヤ

349

なぜ、ソフトウエア・シミュレータ? (2)

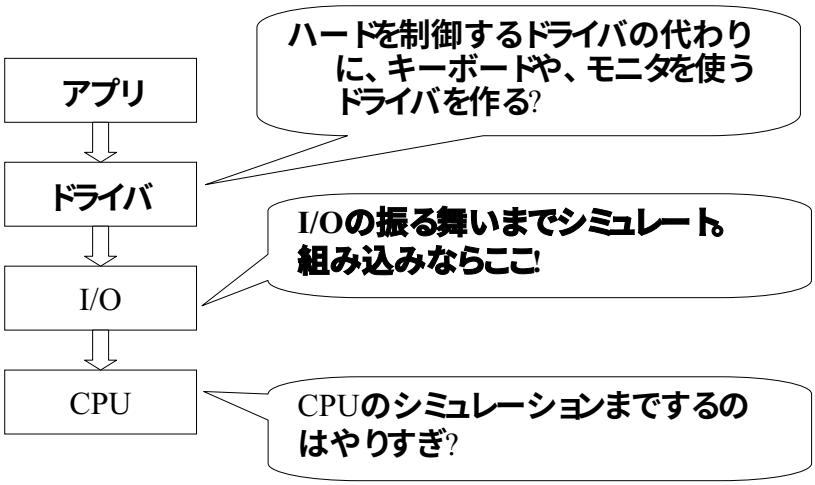
ワンボード・マイコンで作ることにしました。

- 理由
  - メンバーに提供していただきました
  - 作り込めば、なんでもできる
- 作業に取りかかりましたが
  - ハード仕様書を書いた
  - 回路図もできた
  - でも、なかなか形にならない
  - ちょっと違うマイコンのCも勉強しなくちゃ
  - このセミナーの日程が迫ってきましたぁ
    - ← 仕事で苦しんでるのそのまんま

ソフトで
つくってしまえ

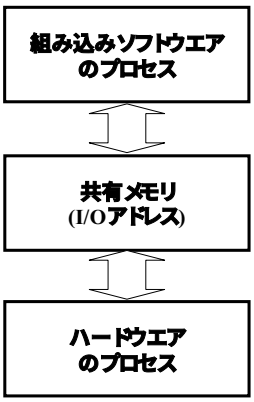
350

どこまでシミュレートするの?



351

どうなっているの?



共有メモリを用意して、
二つのプロセスから読み書きする。(I/O空間のつもり)

片方のプロセスは、
ハードウエアをシミュレート
もう一つのプロセスが、
組み込みソフト

352

なにがいいの?

- ハードウエアがなくても開発開始できる
- ハード屋さんに頼まなくても自分で拡張できる
- お財布に優しい
- 本当のハードウエア同様にI/Oアクセスできる
- ハードウエアが手に入ったら、ポインタの変更だけで対応できる
- シミュレータは別プロセスなのでリンク不要
- 組み込みソフトの変数も、共有メモリにおけば他のプロセスを使ってモニタできる。

でも、いいことばかりじゃないよ。

気をつけなければならないこと

- シミュレートしたところまでしか確認できません
- CPUまで、シミュレートしていません。つまりマイコンのCの仕様と、シミュレータのCの仕様は違うかも
- 実時間処理はやっぱり違う
- 外部環境も何らかの形でシミュレートしなければ。。
- ソフトはもちろん、ハードも設計できる人でないとよいモデルが作れない
- 物理現象の知識もいる。ポットだと、水にヒーターを通して電力をある時間与えると何度温度が上昇するのかな????

デモンストレーション

動かすのにUnixが必要です。

- 共有メモリ
- シグナル
- ノン・ブロックI/O (BSD)
- マルチ・プロセス
- cursesライブラリ
- マルチ・ターミナル

レシピはSESSAME Webサイト

http://blues.tqm.t.u-tokyo.ac.jp/esw/

をごらんください。

つぎはどこへ

- ***商品企画***

  プロトタイプ作って動かして商品を提案しましょう

- ***協調設計***

  ソフト屋自らソフトの作りやすいハードを提案しましょう

- ***ハードウエア知識***

  シミュレータを作ることでハードの立場からも考えることができるようになります。組み込みは、ハード&ソフトです

まとめ

シミュレータの例として、
I/Oレベルでハードウエアをシミュレートする手法を
簡単に紹介させていただきました。

工夫次第で、より快適な開発&テスト環境が得られます。
組み込みを極めるため
“ハードウエア”にも手を出してみられませんか。

余白）

OpenSESSAME Seminor

組込みソフトウェア技術者・管理者向けセミナー

初級者向けテキスト

2002 **年** 10 **月** 15 **日 初版 第** 1 **刷発行**
2003 **年** 10 **月** 29 **日 初版 第** 2 **刷発行**
2004 **年** 3 **月** 19 **日 第** 2 **版 第** 1 **刷発行**
2004 **年** 4 **月** 30 **日 第** 3 **版 第** 1 **刷発行**
2004 **年** 6 **月** 17 **日 第** 4 **版 第** 1 **刷発行**

著 者 上原慶子、大野晋、坂本直史、鈴木圭一、須田泉、西康晴、
二上貴夫、三浦元、三宅貴章、森孝夫、山田大介、山崎辰雄

編集・発行 組込みソフトウェア管理者・技術者育成研究会
（SESSAME）
http://www.sessame.jp
