# 組込みシステムの開発事例に基づく
# 効率的な改良保守に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2021 年 7 月

大 江　秀 幸

# 内容梗概

ソフトウェア保守は近年，ソフトウェアの進化や発展とも呼ばれており，その活動も潜在・顕在的な不具合の修正や環境変化によって，受動的に行うプログラムの修正・変更活動から，より積極的に行うソフトウェアの改変活動として，初期開発と保守を分離しないで考えるという意識に変化してきている．

一方で，初期開発を既に終え，現在稼働している組込みソフトウェアでの保守プロセスに対する取組みとしては，ISO14764 に見られるように，初期開発完了後に，新規開発プロセスの設計・開発手法をそのまま用いて行うことが多い．しかし市場で稼働している組込みソフトウェアの保守には，新規開発とは異なって以下のような特徴がある．

・拡張開発を含めて考えると，ベースソフトの保守期間は 10 年を超える場合もある
・保守期間中、使用性・拡張性などを改良するための保守も行われる
・保守期間の長期化に伴い，初期開発者と保守担当者が異なる場合もあり，情報の継承が難しい
・運用環境が初期開発時と異なるため，動作条件を満たさなくなる場合がある

そのため、以下のような点に注意が必要となる．

・課題１
稼働中のシステムの仕様理解が必須である

・課題２
運用環境の変化が，仕様，設計に与える影響を理解する必要がある

・課題３
環境の変化により修正が必要な箇所を，漏れなく見つける必要がある

本論文では，システムの運用環境の変化に追従する目的で行う保守によって生じる，これらの課題に対して行った以下の事例から，有効な解決法につき議論する．

（１）組込み機器開発における 2038 年問題への対応事例
（２）32bit UNIX システムの 2038 年問題に対するプログラム修正法の提案
（３）予約受付端末の完全化保守開発事例

（1）においては，主に課題２と課題３を議論する．特に運用環境変化に伴う，修正仕様の策定過程と，設計情報が得られないソフトウェアに対して漏れの無い修正を行っている．

（2）においては，（1）の成果を踏まえて，課題２，課題３に加えて効率の良い修正箇所の発見に成功している．

（3）においては，十分な設計情報と修正仕様についての知識を持つ場合に，課題１，課題２，課題３に対応した，完全化保守を効率的に実施した事例である．

各事例では，一般的なソフトウェア保守の分類と，当該ソフトウェアの設計情報の利用可否という観点から保守の設計方針を検討し，決定した．これらの事例の中で，ソフトウェアの改良保守において，設計情報が得られない場合と得られる場合で，効率よく修正する手順を示した．前者の場合は，改良の内容を元にソフトウェアのデータと制御文を抽出し，後者の場合は，ソフトウェアモジュールの責務から修正対象箇所を抽出し，変更可能なモジュール群と慎重に変更すべきモジュール群の境界を変更しないように修正を行う．

これらの手順を用いて，個々の保守案件でそれぞれの課題への対策を行うことで，各保守案件でのソフトウェアの品質と保守効率の向上に貢献できた．

# 論文一覧

## 主要論文

[1-1] 大江秀幸, 安藤友康, 松下誠, 井上克郎, “組込み機器開発における 2038 年問題への対応事例”, デジタルプラクティス, 10(3) , pp.588-602 (2019-07).

[1-2] 大江秀幸, 松下誠, 井上克郎, “32bit UNIX システムの 2038 年問題に対するプログラム修正法の提案”, 情報処理学会論文誌, Vol.62, No.4, pp.1051-1055 (2021-04).

[1-3] 大江秀幸, 松下誠, 井上克郎,“予約受付端末の他サービス転用に向けたソフトウェア改造事例”（投稿準備中）

## 国際会議

[1-1] Hideyuki Oe, Makoto Matsushita, Katsuro Inoue, A Practical Approach to the Year 2038 Problem for 32-bit Embedded Systems, Asia BSD Con, P07A, pp.1-8, Tokyo, Japan, 2020-03.

# 謝辞

本研究の推進にあたり，常日頃より適切なご指導を賜わりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻　井上克郎教授に，心より深謝申し上げます.

本研究の推進にあたり，直接具体的なご助言とご指導を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻　松下誠准教授に心より御礼申し上げます.

本論文の執筆にあたり，有益なご教示，ご助言を賜りました大阪大学大学院情報科学研究科　楠本真二教授，伊野文彦教授に深く感謝いたします.

本論文の執筆にあたり，各種データを提供頂きました，パーソル AVC テクノロジー株式会社の安藤友康氏に心より御礼申し上げます.

本研究の推進にあたり，業務の負荷調整を快諾頂き，通学等に便宜を図って頂きました，関西デジタルソフト株式会社の沖上俊昭社長に心より御礼申し上げます.

# 目次

# 1. はじめに

一般に工業製品を作成する場合には，最初に物が作られる過程と，作られた物に何らかの変更を加える過程を明確に分けて表現する場合が多い．例えば建物の場合には新築，リフォームなどと呼び，車の場合には設計，開発，製造に対して整備，修理，改造などと呼んで区別する．

ソフトウェア開発の場合，この区別があいまいになる場合がある．国内の組込みソフトウェア開発では，新規開発に比べて機能拡張を行う改良（派生）開発が多く，10 年程度の保守が必要な場合が多い傾向がある[1-1].

図 1 に組込みソフトウェアを対象とした開発システムの経過年数，および開発システムの保守年数を示す。図中，左側のグラフは，組込みシステムの改良（派生）開発の場合に，2019 年まで保守または維持管理されていた期間（年数）と開発システム件数の分布を示し，右側のグラフは開発プロジェクト終了後に保守または維持管理しなければならない年数（製品寿命）と開発システム件数の分布を示す．

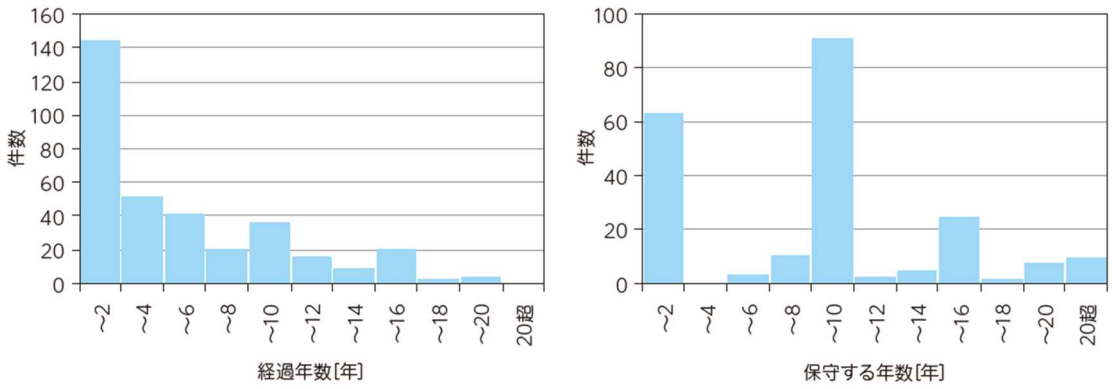


**図 1 開発システムの経過年数・保守年数**
**（組込みソフトウェア開発データ白書 2019. p.37 より転記）**

図 1 の右側のグラフから，一般に保守期間としては 10 年程度以上のものが多くあることがわかる．実際に筆者が開発に関わった組込みソフトウェア，例えば市販のテレビやレコーダー，オーディオ製品などでは，10 年以上に渡って同じソフトウェアアーキテクチャを基に機能拡張を繰り返し，製品を生み出している．

テレビなどの民生向けの組込みシステムは，大量生産を前提に開発する．製造コストを抑えるため，用いるハードウェアの調達コストを可能な限り低減させ，極力短い期間での開発

を目指す．ハードウェアは，時間と共に性能や機能向上，低価格化が進む．その中で民生向けの組込みシステムでは，ハードウェアとの並行開発も考慮して，開発スケジュールを定める[1-2].

組込みシステムでは，用いるハードウェアの仕様に合わせてソフトウェアを開発するため，ハードウェアとソフトウェアの結びつきは強くなる．例えば，複数のハードウェアを協調的に動作させて機能を実現する組込みシステムでは，採用した各ハードウェアが動作するタイミングに合わせて，ソフトウェアの制御を行うよう設計する必要がある．選択したハードウェアにより，その制御タイミングや時間的な許容誤差も変わるため，専用のソフトウェアを用意して制御することになる．

このような理由で，組込みシステムでハードウェアを変更する場合には，機能的な差分が無くても，ソフトウェアの改変が必要になる場合が多い．また，ハードウェアの変更がなく，ソフトウェアの改変によって機能追加を行う場合にも，利用しているハードウェアの仕様が設計上の制約になる場合があることに注意する必要がある．

ソフトウェア設計での工夫として，ハードウェア変更時の改変規模を小さくするため，ハードウェアに依存する部分を，他のモジュールから分離するように設計することが一般に行われている．更にハードウェア変更の有無に関わらず，ソフトウェアモジュール間の結合度を下げ，各モジュールの凝集度を上げることで改変による影響範囲を限定することができ，そうすることで，設計，実装，テスト範囲を限定し，改変にかかるコストを低減することができる．

このように設計したソフトウェアでは，改変の影響を受けない安定したソフトウェアモジュールを積極的に再利用して製品開発を行う．これにより開発量を抑制し，製品バージョンアップにかかるコストを抑制することができる．この再利用するモジュール群を 10 年以上もの長期に渡って利用する場合がある．ソフトウェアの再利用を前提とした開発では，特に再利用箇所の占める割合が多い場合，ソフトウェア開発としては新規開発とは言えない．筆者の所属する組織では，このような開発を「拡張開発」と表現しており，一般的には保守（適応保守）や進化などと表現する場合もある[1-3].

本稿では組込みソフトウェアを対象とし，一般にソフトウェア保守と呼ばれる，システム運用開始後に行うソフトウェアの改変をテーマとして企業での開発の実例を示し，より効率の良い開発方法について議論する．

## 1.1 ソフトウェア保守

本論文は組込みシステムの保守を対象にするが，本章では，ソフトウェアの保守一般について議論する．また本論文では，ソフトウェアの修正を主眼に置くため，本章においても，保守でのソフトウェア修正を念頭に記述する．

運用中のソフトウェアの改変は，一般にソフトウェア保守と呼ばれる．ソフトウェア保守のプロセスは，ISO/IEC/IEEE 14764 : 2006（以下，ISO 14764）に，体系的に示されている．

ISO 14764 では，ソフトウェア保守プロセスの準備として，保守案件の発生要因（不具合，要望，操作ミス等），発生箇所（利用者，運用者等）の整理，保守の型（タイプ）の特定等を行うことが記されている．ここでは保守の分類を挙げ，本論文で議論する保守の型（タイプ）を特定し，更に保守作業の概要を記述する．

### 1.1.1 ISO 14764 での保守の分類

ISO 14764 では，保守の型（タイプ）を 5 つに分類している．この 5 つの型は，大きく訂正保守と改良保守に分けることができる．以下にまとめる．

①訂正保守
・是正保守（corrective maintenance）：既存ソフトウェアの事後修正
・緊急保守（emergency maintenance）：計画外で一時的な既存ソフトウェアの修正
・予防保守（preventive maintenance）：既存ソフトウェアの潜在的な障害の修正

②改良保守
・適応保守（adaptive maintenance）：運用環境変化に応じて行う既存ソフトウェアの修正
・完全化保守（perfective maintenance）：改善を目的とした既存ソフトウェアの修正

ここで改良保守（maintenance enhancement）は，ISO14764 の用語定義で「新しい要求を満たすための既存ソフトウェア製品への修正（機能追加も含む）」とされている．本稿ではソフトウェアの誤りを修正する訂正保守ではなく，環境の変化に適応し，将来の改善を目的とする改良保守を対象とする．

### 1.1.2　変更作業の実際

ISO 14764 では，保守プロセスを 6 つのアクティビティで整理している．

**表 1　保守プロセスのアクティビティ**

| 番号 | アクティビティ | 概要 |
|---|---|---|
| ① | プロセス実装 | 保守計画および保守手続きの策定，修正依頼手続きおよび問題報告手続きの策定，構成管理の開始. |
| ② | 問題分析および修正の分析 | 修正依頼および問題報告の分析，問題の再現または検証，修正実施に関する選択肢の用意，承認の受領. |
| ③ | 修正の実施 | 保守対象の分析，ソフトウェアの修正，およびテスト. |
| ④ | 保守レビューおよび受入れ | 修正されたソフトウェアと修正テストの結果から，修正を承認する権限を持つ組織と共同レビューを行い，修正されたシステムの完全性を確認する. |
| ⑤ | 移行 | 移行計画の策定，利用者への通知，利用者への教育訓練，移行完了の通知，移行後影響への評価を行い，関係データ，成果物を構成管理下に保管する. |
| ⑥ | ソフトウェア廃棄 | 有用でなくなった場合に廃棄する. |

この内，本稿で議論するのは②，および③のソフトウェア修正に関係するアクティビティである．ただし，③の「修正の実施」では，保守プロセスから，ISO 12207（SLCP）での主ライフサイクルプロセスである，開発プロセスを呼び出す．よってソフトウェアの修正については，新規開発と同じプロセスとなる．ISO 14764 では，保守の型に合うように適宜，保守プロセスをテーラリングすることが推奨されている．

開発の実作業においては，③の修正の実施で行われる，保守対象の分析が重要となる．この分析は，保守対象の設計情報が得られるか否かで方法が変わる．設計情報が得られない場合のソフトウェア保守の例を 2 章，3 章で挙げる．この場合はソースコードから抽象度を上げ，設計情報を得る必要がある．また，設計情報が得られる場合のソフトウェア保守の例を 4 章で挙げる．

### 1.1.3　その他の分類

E.B.Swanson は，保守を「把握するのが困難な活動」としており，理解するためには，適切な次元の軸を設定すべきとしている．その中でソフトウェア保守の基盤として，以下の 3 つの分類を導入している[1-4].

・修正（corrective）保守：欠陥により必要な変更を行うもの
・適合（adaptive）保守：運用可能な状態を維持するための変更を行うもの
・完全化（perfective）保守：ユーザの拡張要求を満足させる進化のための変更

修正保守は, 処理の誤り, パフォーマンス障害, 設計と実装の乖離を修正するための保守, 適合保守は，データ環境や処理環境の変化に追従するための保守，完全化保守は，非効率な処理の改善や，パフォーマンスの向上を目的とした保守を例として挙げている．

この分類に従った場合，本論文の対象は適合保守，完全化保守である．

## 1.2 本研究の位置づけ

本研究は，組込みシステムの開発現場で行われている実際の保守の分析や評価を通して，今後の保守作業の生産性向上に役立てることを目的とする．

保守は, 従来の考え方を拡張して, 進化や発展と呼ぶ動きもある．ソフトウェア進化は，一旦出荷されたソフトウェアに対する変更を受け入れる仕組みや活動を指す．従来のソフトウェア保守が不具合修正を中心とする消極的な活動であったのに対し，変更を積極的に取り入れる活動がソフトウェア進化と言われている．本研究もソフトウェア進化に関する研究に位置付けられる。

ソフトウェア進化研究の分類[1-3]においては，手法，対象，目的の 3 つの視点の組合せから分類を行っている．表 2 にソフトウェア進化研究の分類（抜粋）を示す.

表 2　ソフトウェア進化研究の分類（[1-3]より抜粋）

| 視点 | 分類項目 | 説明 |
|---|---|---|
| 手法 | 形式的（Formal） | 数学的モデルに基づき厳密解を求める手法．論理と推論，形式言語と意味論，抽象解釈など |
| | 解析的（Analytic） | 言語モデルや式に基づき厳密解を求める手法．静的コード解析，動的プログラム解析，ソフトウェア計測など |
| | 実践的（Pragmatic） | 経験に基づき近似解を求める手法．経験則（heuristics），パターンや原則，知識ベースなど |
| | 実証的（Empirical） | 統計や事例に基づき近似解を求める手法．データ処理，リポジトリマイニング，事例研究（case study），メトリクスの利用など |
| 対象 | コード（Code） | ソースコード，実行コードなど |
| | モジュール（Module） | 設計仕様，設計文書，フレームワークなど |
| | アーキテクチャ（Architecture） | アーキテクチャ記述，コンポーネント，プロダクトラインなど |
| | システム（System） | システム全体，複合システム（system of systems）など |
| | 要求（Requirement） | 要求仕様，要求文書，マニュアル，ゴールなど |
| | プロセス（Process） | 開発プロセス，アクティビティ，ワークフローなど |
| 目的 | 進化の分析（Analysis） | ISO14764 の Problem and modification analysis, Process implementation に対応．変更要求の把握，進化個所の特定，進化内容の決定など |
| | 進化の実現（Implementation） | ISO14764 の Modification implementation, Migration, Retirement に対応．進化の実施，システムの移行，システムの破壊など |
| | 進化の検査（Review） | ISO14764 の Maintenance review/acceptance に対応．進化の評価，進化の検証，妥当性の確認，リリース管理など |
| | 進化の解明（Investigation） | 科学的観点（ISO14764 に対応なし）．進化の観測，進化現象の理解など |

この分類では，本研究は，手法としては解析的，対象はコードであり，目的は進化の実現である。

解析的手法でコードを対象とし，進化の実現を目的とした研究には，リファクタリングに関するものなどがある．例えば依存関係の解析によりリファクタリングを素早く，正確に行うためのツールを提案するもの[1-5]や，API を用いる場合に，過去に行われた API に対するリファクタリングを再適用するもの[1-6]等がある．本研究では，プログラムスライシング[1-7][1-8]を用いた適応保守の事例を挙げて，その有効性を議論する．

## 1.3 開発事例における課題

前述の通り，ソフトウェア保守においては，改変元の設計情報を利用できるか否かで技術課題の傾向が変わる．いずれの場合でも，企業においては開発量を極力抑えるため，可能な限りアーキテクチャの変更を避ける方針が選択される．この方針に沿うには，変更後もモジュール間の依存関係を変えないことや，各モジュールの凝集度を高く保つことが望ましいが，設計情報が利用できない場合には，これが困難となる．開発事例において，改変元の設計情報を利用できない場合とできる場合の 2 種類の事例につき検討する．

### 1.3.1　修正箇所の特定に設計情報が利用できない場合

32bit の UNIX システムでは，西暦 2038 年 1 月 19 日 3 時 14 分 7 秒（UTC）を過ぎると誤作動を起こす恐れがある．これを 2038 年問題と呼ぶ．誤作動は，時刻情報を time_t 型と呼ばれる 32 ビット符号付整数データ型で扱うことに起因して起こる．システム時刻は，1970 年 1 月 1 日 0 時 0 分 0 秒（UTC）を起点（epoch）とした経過秒で管理しており，epoch からおよそ 68 年後の 2038 年にデータの最大値(0x7FFFFFFF)を超え，桁あふれを起こしてしまうためである．

図 2 に，改造時に 2038 年問題で修正すべき対象箇所を記した概念図を示す．図中の箱がソフトウェアモジュールを表し，アプリケーションから OS・デバイスドライバまで，論理的な階層を考慮して，モジュールを配置している．

時刻情報を保存する変数や，時刻情報を取得したり加工したりする処理は，当該ソフトウェアの初期設計時の関心事として現れない場合には，専用のモジュールを用意することはあまりない．そのため図にある通り，修正対象箇所は，様々なソフトウェアモジュールに規則性なく遍在することになる．

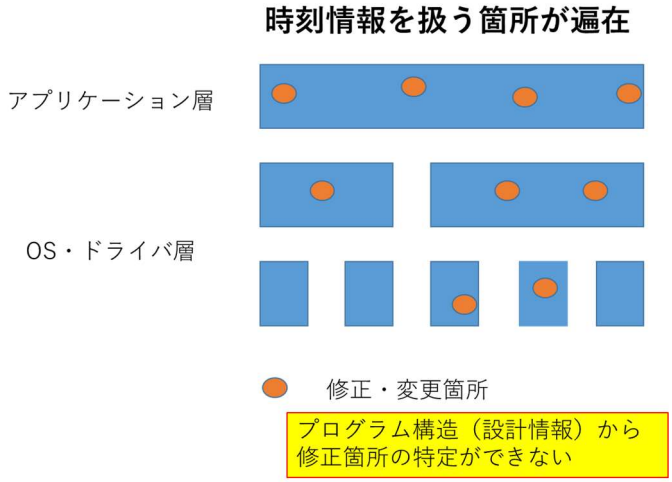


図 2　仕様変更に伴う設計修正箇所の特定（1）

2038 年問題への対応では，時刻情報のオーバーフローを回避するという明確な仕様があるものの，図 2 のようにプログラム構造を現した設計情報があったとしても，どこを修正すれば良いかを特定することは困難である．そのため，まずどこを修正すれば良いかを特定することが課題となる．この特定には，関連する変数，関数名のソフトウェア全体からの検索の他，従来から保守開発の技術として利用される，プログラムスライシング等の応用が考えられる．

また 64bit システムに移行できない場合，32bit システムのままオーバーフローを回避するには，時刻情報をどのように扱えば良いかを，動作仕様として決める必要がある．

### 1.3.2　修正箇所の特定に設計情報が利用できる場合

改変元の設計情報が利用できる場合は，仕様差分から設計変化点を抽出し，該当する箇所のプログラムを修正する．

本論文では，各種の仕様書や設計書が作成され，適切にメンテナンスされている例を挙げ，修正の具体的な手順を示す．まず改造仕様と元仕様との関連を調査し，元仕様を変更する部分と，新たな仕様としてシステムに実装する部分を明確化する．元仕様を変更する部分については，元仕様を実現している設計モジュール群を設計ドキュメント等から特定し，変更箇所（モジュール）を決定する．新たな仕様としてシステムに実装する部分は，元の設計思想を崩さないように設計構造を検討し，新たなモジュールとしてシステムに実装する．図 3 に改造前後の修正箇所特定の手順の概要を示す．図中、左側の図で改造前のプログラム構造を用いて修正対象の設計モジュールや処理を特定する．モジュールや処理の特定後，右側の図のように改造仕様を実現するモジュールの範囲を特定する．これを行うには，モジュール間

結合度や凝集度が良好に保たれ，かつ各モジュールの情報が設計情報としてメンテナンスできていることが必要である．これらの条件を満たす場合に， 1.3.1 項の例とは違って，プログラム構造が修正・改造箇所の特定に有効に利用できる．

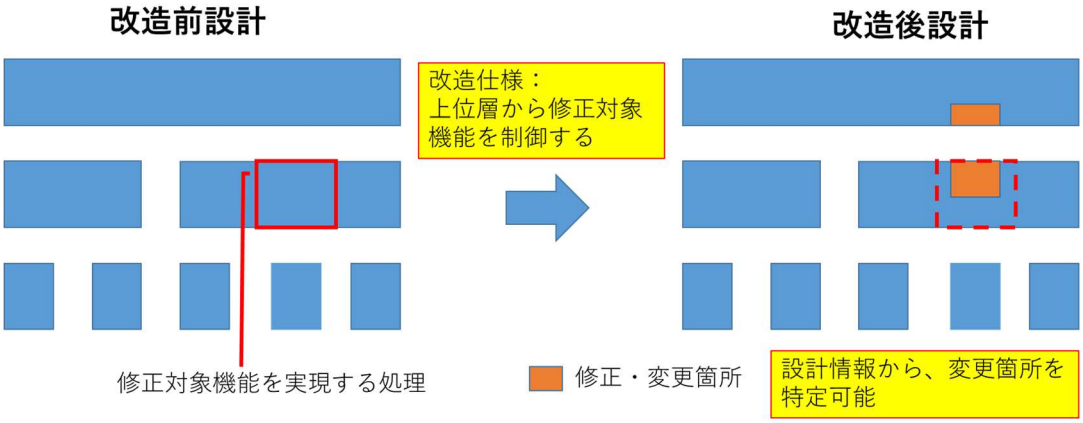


**図 3　仕様変更に伴う設計修正箇所の特定（2）**

修正・改造箇所の特定後には，これらの改造が目的とする改造仕様を満たし，改造しない部分の仕様に影響していないことを保証することが課題となり得る．

## 1.4 本論文の概要

本論文では，組込みソフトウェア開発における適応保守や進化に関わるプロジェクトで，効率よく変更要求を満たしてプログラムの修正を行う方法につき議論する．その際，要求の内容や設計ドキュメントの有無や内容等により，以下の 2 つの場合に分けることができる．

（1）設計情報を利用できない場合
（2）設計情報を利用できる場合

設計情報を利用できない場合，プログラムを何らかの方法で解析した上で，修正箇所を見つけて修正を行う必要がある．一方で設計情報を利用できる場合には，修正箇所や修正による影響範囲を設計情報から読取り，プログラム上の該当箇所を修正する．実際の開発では，どちらの場合もあり得るため，それぞれの場合につき議論する。

## 1.5 各章の構成

本論文では，設計情報が利用不可能な場合と可能な場合，それぞれの事例から効率の良いプログラム修正方法を議論する．

2 章では，設計情報の利用ができない場合に，プログラムからキーワードを元に網羅的に調査して修正箇所を見つけ，修正する方法を示す．

3 章では，2 章と同様の要求仕様で設計情報の利用ができない場合に，プログラムスライシングを応用して修正箇所を効率的に見つける方法を示す．

4 章では，設計情報が利用できる場合に，どのような設計情報をどのように用いて修正箇所を見つけるかを示す．

# 2. 組込み機器開発における 2038 年問題への対応事例

## 2.1 はじめに

組込み機器の高機能化に伴い，その OS として Linux，FreeBSD などの UNIX 系 OS が一般的に用いられている[2-1]．これらの OS は，ターゲットシステムの進化に伴い 32bit から 64bit に移行されることも多い[2-2].

しかし，組込み機器の開発では，開発期間の短縮，コスト削減を主な目的とし，旧システムのソフトウェアを活用して新しい機器の開発を行うことがある．量産を前提とした組込み機器の開発では，1 台あたりのコストのわずかな差が事業に大きな影響を与えるためである．そのため，ユーザメリットにつながる積極的な理由がなければ，新システムの開発時に低コストな 32bit システムの継続利用を判断する場合も多い.

32bit 版の UNIX 系 OS では，時刻情報を 32bit 符号付きデータとして 1970 年 1 月 1 日 0 時 0 分 0 秒（UTC）からの経過秒で管理している[2-3]．このデータは，およそ 68 年後の 2038 年に桁あふれを起こす[2-4]（以下 2038 年問題とする）．時刻情報を扱う機器では，この問題により引き起こされる不具合を回避する必要がある.

2038 年問題への対応要否は，対象機器の動作保証期間にも左右される．例えば動作保証期間を 20 年とする場合には，2018 年にリリースする機器で，この問題に備える必要がある.

類似する問題に，UNIX が稼働する機器で生じた 2004 年 1 月 10 日の障害がある[2-5]．この問題では，桁あふれを考慮せずに 2 種類の経過秒を足し合わせたり，経過秒の最大桁数を間違って設定したりしたため，料金請求プログラムが誤請求を起こしたり，与信ソフトウェアが正常に動作しなくなった．また関連した問題としては，西暦 2000 年問題がある．西暦年の下 2 桁を 10 進数で扱う場合に，100 年に一度，オーバーフローが起こる．この状態で西暦 2000 年を迎えた際，コンピュータ内部では年の情報が“00”となるため，西暦 1900 年との区別が付かず，計算誤りを引き起こす恐れがあった．いずれの問題も対処方法含めて本章と密接な関係を持つ問題であるが，本章で対象とする 2038 年問題を含め，これらの問題に対して具体的なソフトウェアの修正方法や修正結果についての報告は見当たらない.

本章では，筆者らのグループが行った 2038 年問題への対応の方針や，具体的な修正方法を報告する．本知見は，他の同様なシステムの 2038 年問題の改修や，関連する OS の時刻問題に応用することができる.

本章の事例では，修正箇所の特定に設計情報が利用できない場合につき記載する．時刻情報のオーバーフローをどのように回避するか，その修正仕様の策定と，策定した修正仕様をどのように設計に反映するかを議論する.

## 2.2 関連研究

情報システムの時刻のオーバーフローについては，かつて UNIX の 2004 年問題，および西暦 2000 年問題があった．また，2038 年問題についても本章の事例以降に発表された研究が存在する．以下，それぞれの概要につき示す．

### 2.2.1 UNIX の 2004 年問題

2004 年 1 月 10 日 13 時 37 分 4 秒（UTC）は，1970 年から 2038 年問題が発生するまでの丁度中間の時刻である．この中間時刻以降で，2038 年問題が起きる事例が報告されている[2-5]．報告されている原因には，以下のようなものがある．

(a)桁あふれを考慮せずに日付同士を足し合わせた．

(b)0.5 秒単位で時刻を認識するシステムで，最大桁数を増やさなかった．

2004 年は 1970 年から 2038 年の中間にあたり，time_t 値は 0x4000 0000 付近の値となっている．(a)は，この time_t 値 2 つを加えることで 0x7FFF FFFF を超える現象であり，(b)は time_t 値のカウントアップが通常の倍の速度で起こるため，2004 年付近でオーバーフローが発生した事例である．いずれも 2038 年問題と同じメカニズムで起こる問題であり，実際に障害として 2038 年になる前に表面化した例である．

これらに起因する障害事例としては，通話料金の課金システムで曜日の認識を誤って誤請求を行ったもの，通信プログラムが不具合を起こしたもの，ATM が一部正常に利用できなくなったものなどがある[2-5]．

現象発生を事前に対策できた例もあり，ユーザに対策ソフトへのバージョンアップを呼びかけて問題の極小化に成功している[2-5]．本章が対象とした組込みソフトの場合も，事前の対策は可能であり，対策により問題の極小化を図ることができる．

### 2.2.2 西暦 2000 年問題

西暦 2000 年問題は，4 桁の年情報を 10 進数の下 2 桁で表現するために，システム内部でも年情報を 10 進数 2 桁で管理するため，西暦 2000 年を迎えた際に管理領域のオーバーフローを起こす問題である[2-6]．2000 年問題が引き起こす可能性のある問題例としては，以下のようなものがある．

(a)日付計算の間違い
例：1996 年から 2000 年までの期間：0-96=-96．

(b)日付比較の間違い
例：0<96 より，1996 年の方が新しいと誤判断．

(c)入出力による誤り
例：2000 年以降のデータを 1901 年より古いデータとして登録.

これらの問題は，2 桁を 4 桁に拡張する， 2 桁のまま 10 進数 2 桁以上の数を表現することなどで回避可能とされていた[2-6].

2000 年問題は，事前に問題点やリスク，影響範囲が指摘され，国際的にも大きな関心事となっていた．国内においても，各企業で対応計画を策定，予算も確保して対応にあたり，国は金融，エネルギー，情報通信，交通，医療などの各分野毎に対応の進捗状況を確認して正確な情報の開示に努めた.

このような事前の対策の結果，大きな混乱は発生しなかった．2000 年 1 月 5 日午前中現在の情報では，2000 年問題関連の不具合は，比較的軽微な 27 件の問題発生にとどまった[2-7].

2000 年問題では，機器内で管理すべき時刻情報が time_t 値以外に複数個所存在するおそれがあり，本章で報告したライブラリの入出力データの変換だけでは解決できない可能性がある.

### 2.2.3 2038 年問題の最近の研究事例

（1）Detecting and analyzing year 2038 problem bugs in user-level applications.[2-8]
2012 年から 2018 年の 7 月 1 日～7 月 10 日に GitHub にアップロードされた全ての C ベースのプロジェクト（32,921 プロジェクト）を対象に 2038 年問題を含むかを分析し，7.35%にバグがあることを検出した．2038 年問題の検出をテーマとした研究である．検出されたうち 6 つのプロジェクトに修正パッチを送信し、承認された。

（2）Avoiding Year 2038 Problem on 32-bit Linux by Rewinding Time on Clock Synchronization.[2-9]
時刻同期ソフトウェアが受信した時刻を巻き戻すように制御し，NTP サーバ，PTP マスタと同期しているときに，正しく動作できることを確認した事例が紹介されている.

## 2.3 開発対象機器

図 4 に本章で議論する開発対象機器の構成を示す.

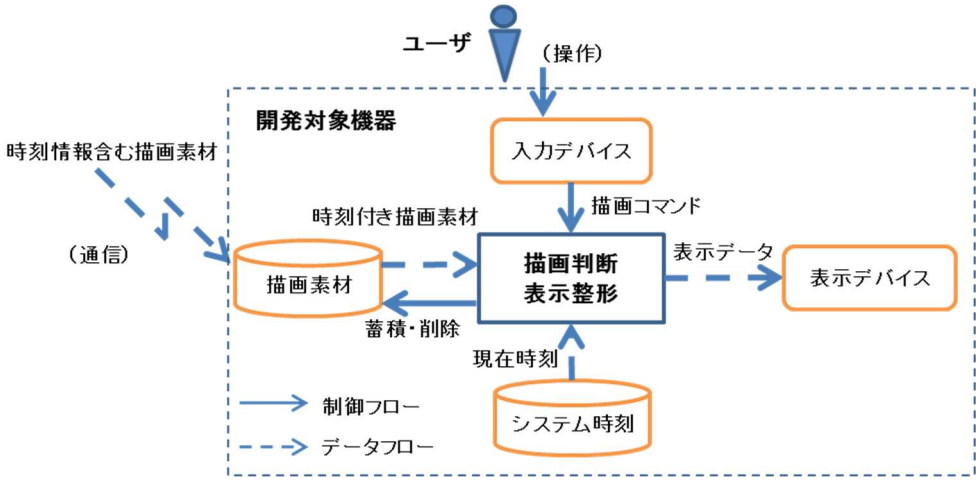


**図 4　開発対象機器の構成**

開発対象機器（以降，当該機器と呼ぶ）は，描画するデータを，そのデータの有効期限と共に通信により受信し，時刻付きの描画素材として保存する．ここで通信により受信する時刻情報は，2038 年以降であっても正しい時刻が通知されることがわかっている．当該機器では，内部で管理するシステム時刻と当該機器を扱うユーザの操作によって，描画すべきデータを選択する．期限の切れた素材は消去する．選択された表示対象の素材は整形を行った後，表示デバイスで情報を表示する．

当該機器のソフトウェアアーキテクチャの概要を図 5 に示す．ここで自社開発部は，当該機器の機能実現のために独自に開発したモジュール群を指し，OS 部は UNIX OS コアとデバイスドライバ，標準ライブラリなど，OSS で公開されているモジュール群を指す．

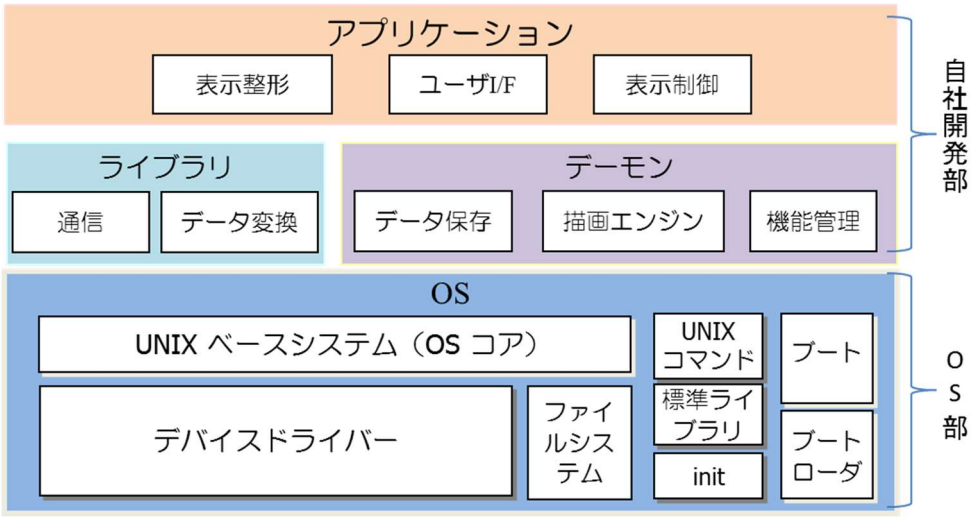


**図 5　アーキテクチャの概要**

## 2.4 開発背景と問題点

### 2.4.1　開発背景

当該機器は，機能拡張とモデルチェンジを一定期間で繰り返し，10 年以上開発を継続してきた．搭載している OS は UNIX 系 OS の FreeBSD である．当該機器の初期モデル開発時は 32bit OS が主流であり，当該モデル開発時点（2017 年）まで，OS やアーキテクチャなどを大きく変更せず，拡張開発を継続している．このことが開発期間の短縮やコスト抑制に貢献している．この製品の寿命は 20 年であり，直近販売のモデルであっても 2038 年問題に対処する必要があった．

### 2.4.2　問題点と課題

FreeBSD では，時刻を UNIX 時刻で管理している．UNIX 時刻は，1970 年 1 月 1 日 0 時 0 分 0 秒 UTC（世界標準時）を起点とする経過秒数である（以下，1970 年起点と呼ぶ）[2-3]．この時刻情報の管理のため，32bit の FreeBSD では 32bit 符号付き整数（signed int）を用いている．最大値は 0x7FFF FFFF（2,147,483,647）であり，起点から約 68 年後の 2038 年 1 月 19 日 3 時 14 分 7 秒（UTC）を超えると桁あふれを起こす[2-4]．桁あふれにより，当該機器では以下の問題の発生が想定された．

（1）表示すべき情報が表示されない
2038 年問題を含まない時刻付きの描画素材を受信，表示するため，当該機器内で管理する時刻と矛盾する場合，素材を表示できなくなる．

（2）負値になるため時刻の大小が逆転する
桁あふれにより 32bit 符号付き整数の最上位ビットが立つため，値の大小関係で情報の新旧を判断している箇所で誤判断してしまう．

（3）負値になるため異常系処理が動作する
負値を期待しない情報であり，負値に対して異常処理を実装している箇所では，異常処理を実行してしまう．

これらの問題を発生させる処理をソースコードから特定し，漏れなく修正する必要がある．

また 2038 年問題への対応は，製品の主たる機能ではない要件であり，修正や開発のコスト（まとめて開発コストと呼ぶ）は極力抑えたいという要望がある．要求を以下にまとめる．

(a)2038 年を超えても製品動作に不具合を起こさないように修正すること．

(b)製品寿命は 20 年とすること．

(c)開発コスト低減のため，開発量やテスト工数が少なくできる手段を採用すること．

(d)開発後の保守での混乱を避ける意味でも，OS のカーネル，ドライバ等，OS 部の修正は可能な限り行わない（OS 提供のデータ構造，API の修正含む）．

当該機器は，機器全体の開発規模が大きく，設計方針次第で影響範囲が大きく異なってくる．開発コスト削減のためには，極力修正範囲を限定したい．機器全体の開発規模を表 3 に示す．ここで，総行数はコメント行含むソースコードの行数，実行数はコメントを除くソースコードの行数である．以降，本章では実行数を用いる．

**表 3　開発規模の概要**

| 対象 | 総行数（百万行） | 実行数（百万行） |
|---|---|---|
| 自社開発部 | 2.5 | 1.3 |
| OS 部 | 0.8 | 0.5 |
| 全体 | 3.3 | 1.8 |

## 2.5 本問題に対する対処

時刻の経過により，time_t 型変数のオーバーフローが起こるのを回避するため，オーバーフローが起こる可能性のあるソースコード上の場所を特定して修正する必要がある．ただし，抽出した全てを修正対象とすると，開発規模が膨大になるため，修正規模を極力小さくするような工夫が必要となる．

### 2.5.1 対策の検討

FreeBSD で時刻情報を管理する変数の型は time_t 型である．2038 年問題の解決のためには，time_t 型変数周辺での修正が必要となる．当該機器の開発者で検討会を行い，表 4 に示す対策案を挙げ，検討した．

**表 4 検討した対策案**

| 対策案 | | 開発量見込み | OS 部修正有無 | 修正による影響範囲見込み |
|---|---|---|---|---|
| (a) | time_t 型を 64bit にする | 大 | 有 | 大 |
| (b) | time_t 型を unsigned int (32bit)にする | 大 | 有 | 大 |
| (c) | time_t 型を変更せず，年月日変換・大小比較する箇所で桁上がりを考慮する | 大 | 有 | 大 |
| (d) | time_t 型を変更せず，ラッパー関数を用いて起点（epoch）を変更してシステム時刻を管理する | 中 | 無 | 中 |

対策案を検討する際には，組込み機器特有の制約事項を考慮する必要がある．1 台あたりの開発コスト低減の他にも，メンテナンスコストにも注意が必要である．例えば何らかの理由でソフトウェアの修正が必要となった場合，一般に，量産後様々な利用環境で動作する組込み機器へのソフトウェアの変更にはコストがかかる．このため，メンテナンスコストは重要な関心事となる．

表 4 の案は，大きく分けて，time_t 型を変更する案（(a)，(b)）と time_t で管理される値の取扱いを変更する案（(c)，(d)）の 2 案となった．これらの案の実現方法は更に，OS 部への変更を前提とするもの（(a)～(c)）と，自社開発部の修正を前提とするもの（(d)）に分かれる．

案(a)は，OS は 32bit のまま time_t 型の定義を 64bit に変更する案である．OS 定義のデータ型のビット幅を変更するので，ソフトウェア全体で変更・確認作業が必要になる．このため，開発量，修正による影響範囲共に大きくなることが予想された．案(a)の別案として，OS の 64bit 化も検討されたが，開発工数が案(a)より増大するため選択できなかった．

案(b)は，time_t 型を 32bit にしたまま signed int から unsigned int に変更する案である．案(a)と違いビット幅の変更は無いが，OS 定義のデータ型を変更するので，ソフトウェア全体で変更・確認作業が必要になることは変わらない．このため，開発量，修正による影響範囲共に大きくなることが予想された．

案(c)は，time_t 型は変更しないで，値を比較したり読み出す際，桁上がりが起こっている場合に，2038 年以降として扱う案である．time_t 型の変数を扱う，ソフトウェア全体で変更・確認作業が必要になる．このため，開発量，修正による影響範囲共に大きくなることが予想された．

案(d)は，本プロジェクトで採用した案である．time_t 型は変更せず，起点（epoch）を 1970 年より後にずらして扱うことにより，ずらした分だけオーバーフローの時期を 2038 年より遅らせる案である．起点変更と，OS 部提供の API 呼び出しを実行するラッパー関数を用意し，必要に応じて利用することで，OS 部を一切変更することなく機能を実現する．開発量，影響範囲ともに OS 部を変更する案よりは小さくなる．

さらに，各案の修正規模把握のため，time_t 型を明示的に使用する箇所を調査した．その結果を表 5 に示す．

**表 5　調査が必要なデータを使用する箇所**

| 使用箇所 | 箇所数 |
|---|---|
| OS 部（デバイスドライバ，OSS，標準ライブラリ含む） | 2546 |
| 自社開発部 | 945 |

time_t 型そのものを変更する案(a)，案(b)では，OS 部を含めた 3000 箇所を超える部分で，データがどのように使用されるかを調査する必要がある．加えて例えば，time_t 型のデータを 32bit の signed int 型データにキャストして何らかの時刻計算を行っている場合，time_t 型の拡張だけでは 2038 年問題は解決しない．定義されたデータを使用する箇所全てで，このような使い方をする箇所が無いかを確認する必要がある．また OS 部については，自社外でメンテナンスのため変更される可能性があり，OS 部を自社独自に修正した場合には，OS のメンテナンスへの追従にもコストを見込む必要がある．

このように，OS 部の変更が必要な案については，変更による影響が広範囲に及ぶため，限られた期間での修正対応と動作保証は困難であると判断した．また，メンテナンスコストも考慮して，OS 部の変更を前提としない案(d)を選択することとした．

### 2.5.2　修正作業の概要

#### 2.5.2.1　修正仕様の検討

2.5.1 項に示した通り，修正設計としては 1970 年の起点から一定期間遅らせる案を採用した．ここで，遅らせる期間を決める必要がある．当該機器は，製品寿命を 20 年としている．遅らせる期間を 20 年以上とすることで，オーバーフローが起こるのは 2058 年以降となる．そのため 2018 年以降，2038 年までのどの時点からのシステムの稼働にも対応することができる．

また遅らせる期間は，閏年を考慮すると 4 の倍数とすべきである．遅らせる期間を 28 年とすれば，2099 年までは曜日まで含めてカレンダーが一致することが分かっており，時刻から曜日の情報を得る際にも修正は不要となる．28 年の倍数を使えば，曜日に影響させずに更にオーバーフローの時期を遅らせることができる．現在起点となっている 1970 年の 28 年後は 1998 年であるが，1998 年の 28 年後は 2026 年であり，2026 年を起点とした場合は 2018 年現在を表現することができない．

以上より，遅らせる期間は 28 年と決定し，1998 年 1 月 1 日 0 時 0 分 0 秒（UTC）を起点（1998 年起点と呼ぶ）とすることとした．図 6 に 1970 年起点，1998 年起点それぞれの time_t 型変数値の違いを示す．

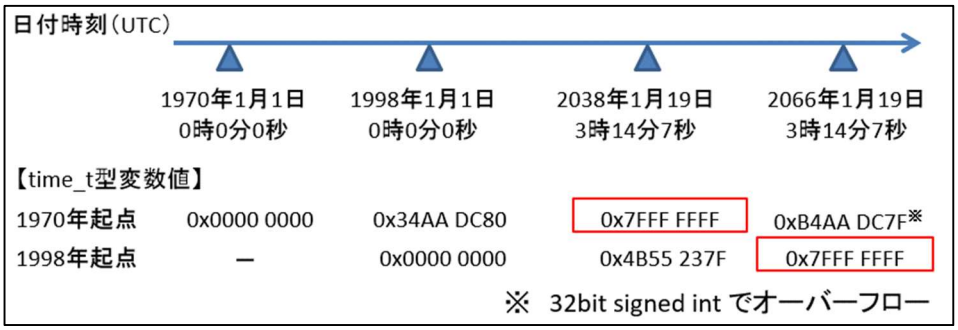


**図 6　起点変更と time_t 型変数値**

2.5.1 項で示したように，今回の手法は OS 部を変更しないため，起点変更の実現には，time_t 型の値を 1970 年起点と解釈する OS 提供ライブラリの使用に注意が必要となる．システム外部の time_t 型以外の値と，システム内部の time_t 型の値を OS 提供ライブラリを用いて変換する場合に，自社開発部で起点変更した time_t 型の値との整合が取れなくなるためである．これを，値の扱い方に応じた起点解釈の変更により回避する．

（1）起点解釈の変更が必要な箇所

システム外部の時刻情報とシステム内部の time_t 値を，OS 部のライブラリを用いて変換する箇所で，起点解釈の変更が必要となる．修正前の自社開発部には，システム外部から入力される文字列の時刻情報を，time_t 型の値に変換する処理が存在した．文字列から数値（unsigned int 型）の時刻情報を得るために，この処理を流用することにしたため，変換後の時刻情報は 1970 年起点の時刻となる．この値から，システム内部の time_t 型の変数値を作るために OS 提供のライブラリ関数 settimeofday を用いると，1970 年起点の誤った値が time_t 値に設定される．1998 年起点として正しい time_t 値を作るには，起点解釈の違いを吸収する変換が必要となる．また逆に 1998 年起点の time_t 値から正しい年月日等の要素別時刻情報を得る際にも，変換が必要となる．そのため，図 7 に示す，起点解釈変更のための変換を行うことにした．変換は，次のように行う．

外部の時刻から，1998 年起点のシステム時刻の値を得るには，28 年分の時間（0x34AADC80）を減算する．反対に，1998 年起点のシステム時刻から正しい時刻の値を得るには，28 年分の時間を加算する．

（2）起点解釈の変更が不要な箇所

図 7 の起点解釈変更後の破線矢印に示すように，システム内部の自社開発部と OS 部でのみ利用する time_t 値を扱う箇所では，自社開発部，OS 部の双方で 1998 年起点として扱うことができるため，起点解釈の変更は不要である．2 つの時刻の差分を計算したり，時刻の新旧を判断する場合には，起点の同じ時刻情報同士を比較すれば良いためである．例えば 2018 年 5 月 1 日（UTC）と同年 5 月 22 日の差は，1970 年起点でも 1998 年起点でも同じ 3 週間であるため，time_t 型変数をそのまま比較すれば良い．

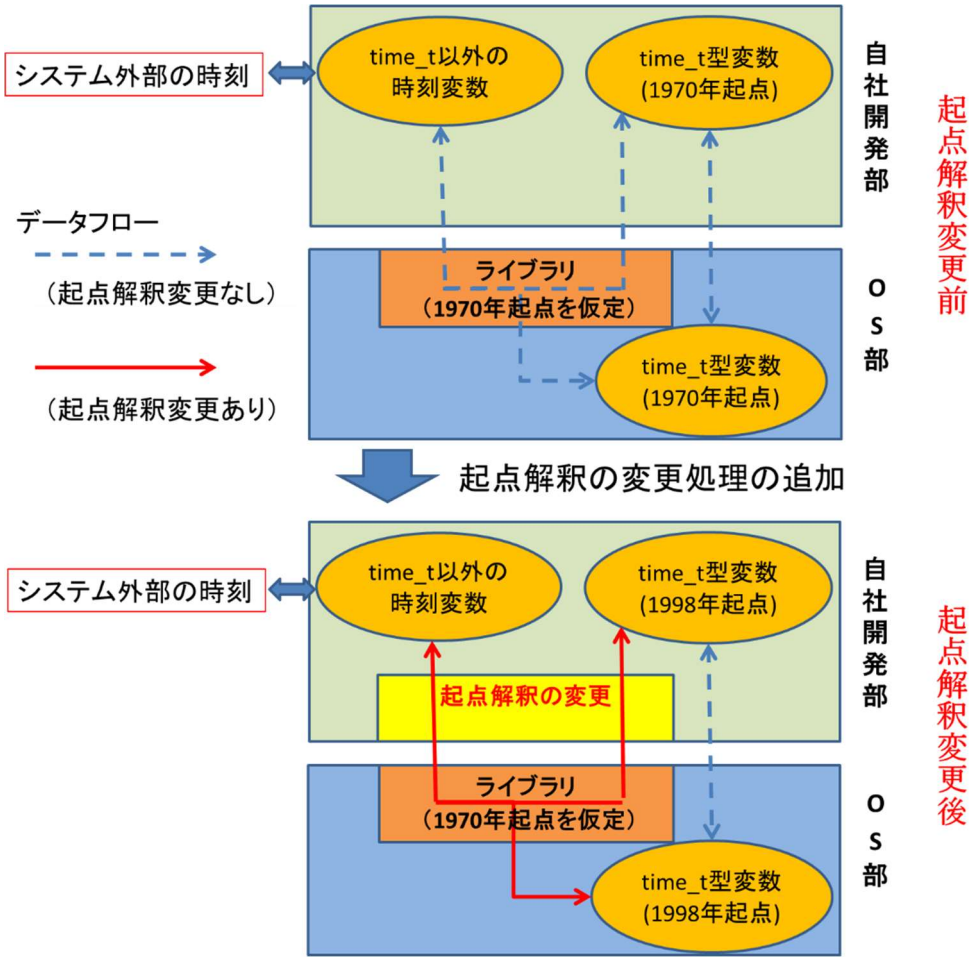


図 7　時刻の扱い方の変更

上記の設計案実現のため，時刻を扱うデータ，関数のそれぞれで，修正時の影響調査が必要である．

#### 2.5.2.2　変数の調査

今回採用した案(d)では起点変更を行うため，2038 年ではオーバーフローは発生しない．定義型を変更しないため，time_t 型から int 型等へのキャストを行っていたとしても，2038 年問題は発生しない．図 5 の自社開発部でシステム時刻を 1998 年起点の値となるように制御すれば，OS 部も含めてオーバーフローしない時刻のみを扱うことができる．そのため，time_t 型の変数を使用している箇所の調査は，OS 部については必要はなく，自社開発部での 945 箇所の調査のみで良い．

ただし，修正箇所の特定には，time_t 型を直接宣言している変数に加えて，構造体のメンバに time_t 型を使用している場合についても調査が必要である．表 6 に示す型のデータが調査対象となる．

**表 6　調査が必要なデータ型**

| 項番 | データ型 | 概要 |
|---|---|---|
| 1 | time_t | システム時刻を保存するためのデータ型．標準 C ライブラリで定義され，32bit FreeBSD では，32bit 符号付き整数型で定義されている． |
| 2 | struct timeval | time_t 型変数(tv_sec)と suseconds_t 型変数(tv_usec)をメンバに持つ構造体．tv_sec では秒を，tv_usec ではマイクロ秒の情報を格納する． |
| 3 | struct tm | 年，月，日，曜日などの要素別時刻情報をメンバとし，各メンバを int 型で格納する構造体. |

自社開発部でのこれらのデータにつき，time_t 型の変数はオーバーフローを起こさないよう，1998 年起点のデータを利用するように調査検討した．struct timeval，　struct tm 型の変数ではどのようなデータで管理すべきかも調査した．調査の結果，ライブラリ以外で修正必要な箇所は無いと判断した．

### 2.5.3　関数の調査

時刻を扱う関数を使用する全ての箇所につき調査を行った．関数調査では，標準 C ライブラリ，および UNIX 標準ライブラリ（以降，単にライブラリと呼ぶ）について，各ヘッダファイル（time.h）で定義された関数を調査対象とした．調査対象の関数の内，当該機器で使用している関数については修正対応が必要となる．OS 部の修正は行わないため，各ライブラリの呼び出し元で修正を行う方針とした．表 6 に調査必要なライブラリ関数の例を示す．

それぞれの関数の引数，戻り値につき，time_t 型の変数はオーバーフローを起こさないよう 1998 年起点のデータを設定するようにし，struct tm 型の変数ではどのようなデータを管理すべきかを調査した．

調査結果から全ての修正対象データ，関数をチェックし，時刻データの取得や保存などを行うライブラリに，新規にラッパー関数を作成する方針とした．ラッパー関数では，28 年を time_t 値に対して加減算した上で，OS 部の提供するライブラリを呼び出す．自社開発部からは，作成したラッパー関数を呼び出す．これにより，OS 部を一切変更せず，かつ元システムの修正範囲も極力小さくして，案(d)を実現することを目指した．

調査の結果，表 7 のラッパー関数作成列に〇印で示す 12 個のライブラリに対して，ラッパー関数を設けることとした．

## 表 7　調査が必要なライブラリ関数の例

| 項番 | 関数名 | ラッパー関数作成 |
|---|---|---|
| 1 | clock_t clock(void) | － |
| 2 | int clock_gettime(clockid_t, struct timespec *) | － |
| 3 | char *ctime(const time_t *) | ○ |
| 4 | char *ctime_r(const time_t *, char *) | ○ |
| 5 | double difftime(time_t, time_t) | － |
| 6 | int gettimeofday(struct timeval *, struct timezone *) | ○ |
| 7 | struct tm *gmtime(const time_t *) | ○ |
| 8 | struct tm *gmtime_r (const time_t *, struct tm *) | ○ |
| 9 | struct tm *localtime(const time_t *) | ○ |
| 10 | struct tm *localtime_r (const time_t *, struct tm *) | ○ |
| 11 | time_t mktime(struct tm *) | ○ |
| 12 | int nanosleep(const struct timespec *, struct timespec *) | － |
| 13 | int setitimer(int, const struct itimerval *, struct itimerval *) | － |
| 14 | int settimeofday(const struct timeval *, const struct timezone *) | ○ |
| 15 | size_t strftime(char *, size_t, const char *, const struct tm *) | ○ |
| 16 | time_t time(time_t *) | － |
| 17 | time_t timegm (struct tm *) | ○ |
| 18 | time_t timelocal (struct tm *) | ○ |
| ・<br>・<br>・ | ・<br>・<br>・ | ・<br>・<br>・ |
| 60 | void tzsetwall(void) | － |

### 2.5.4　修正設計概要

図 8，および図 9 に，修正前後の設計概要を示す．システム内部の時刻を 28 年ずらすため，外部時刻とのインタフェースを持つ箇所では補正が必要となる．

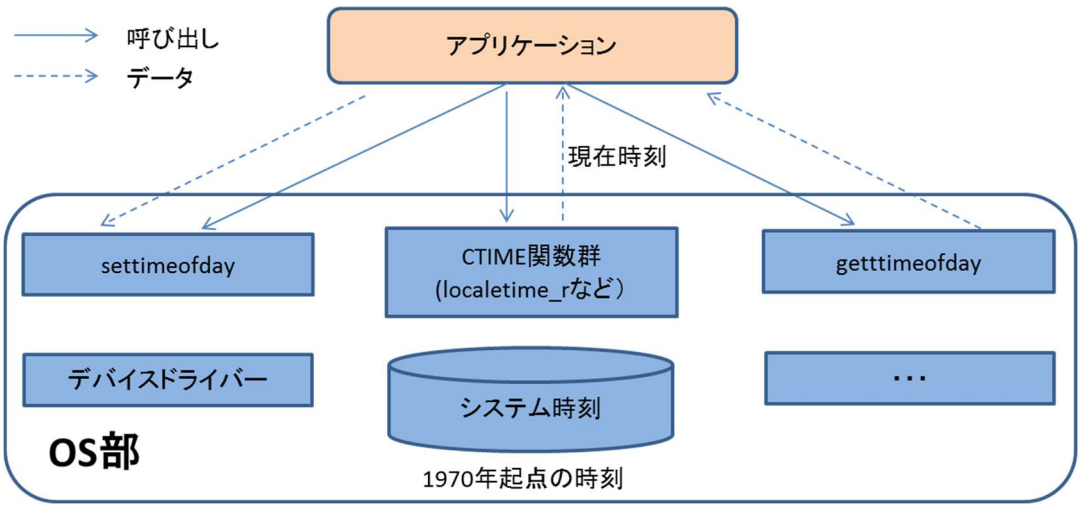


図 8　修正前の設計概要

図 8 に，修正前のシステム時刻の管理の概要を示す．システム時刻は time_t 型のデータで管理される．2038 年まではオーバーフローを起こさないため，管理された 32bit のデータから，年月日等の要素別時刻情報を取り出す．取り出した値は，アプリケーションが用いる．

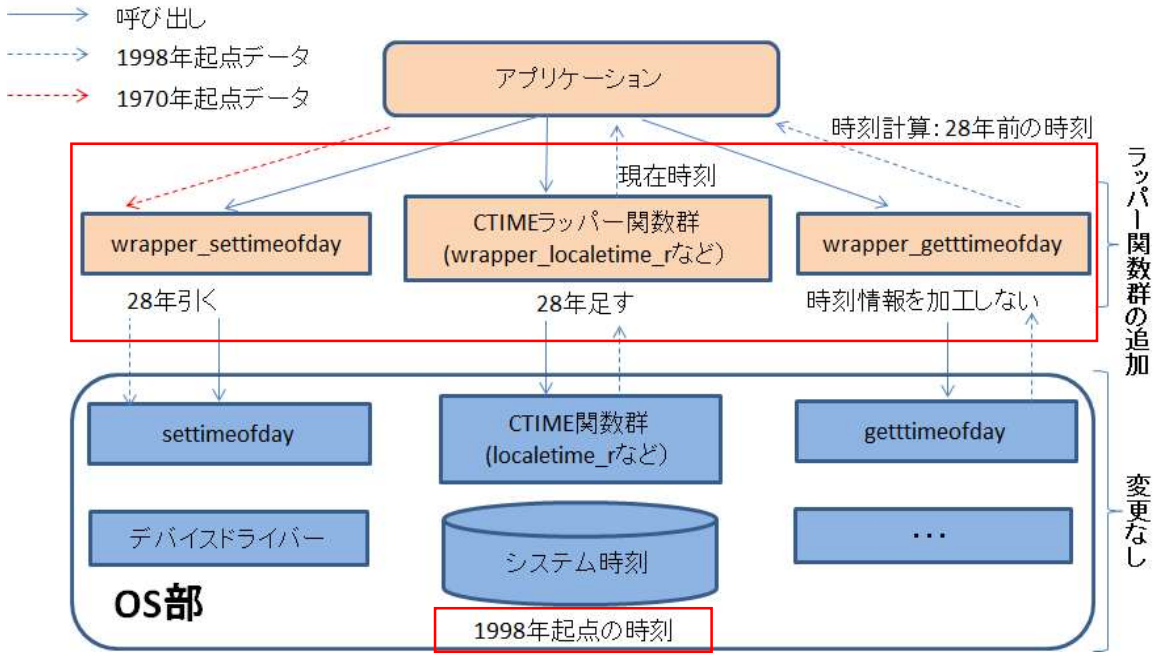


図 9　修正後の設計概要

図 9 には修正後の設計概要を示す．修正のため，時刻情報を扱う OS のライブラリ対してラッパー関数を用意している．アプリケーションからは，ラッパー関数を介して OS のライブラリを利用する．

当該機器のシステム時刻は 1998 年起点の time_t 値，つまり 1970 年起点の解釈から 28 年前を示す time_t 値で管理する．これにより，オーバーフローの発生を 2066 年まで遅延させることができる．年月日など要素別時刻情報や，時刻情報を文字列で返却する CTIME 関数群では，主に機器外部に対して用いる時刻のため，ラッパー関数で 1998 年起点時刻への正しい解釈への変更（プラス 28 年）を行う．時刻情報を time_t 型の 32bit 値で返す gettimeofday()のような関数では，ラッパー関数では値をそのまま返す．また，通信で受信する時刻情報についても，1998 年起点の時刻に解釈を変更して保存する．

まとめると，当該機器のシステム時刻を 1998 年起点とし，time_t 値から要素別時刻情報を求めたり，要素別時刻情報から time_t 値を求めるライブラリを使用する際には，ラッパー関数を用いて適宜起点の解釈を変更する．上記により OS 部を一切変更することなく，当該機器のシステム時刻を 1998 年起点の時刻で管理することができる．

### 2.5.5　実装例

以下に実装例を示す．まず，起点変更時間である 28 年を，年と秒それぞれで定義する．

```
#define DATE_OFFSET_YEAR    28       /* 28years */
#define LEAP_DAYS   (DATE_OFFSET_YEAR >> 2)
#define DATE_OFFSET_SEC     ((DATE_OFFSET_YEAR * 365 + LEAP_DAYS) * 60 * 60 * 24)
・・・
```

**図 10　起点変更時間の定義**

DATE_OFFSET_YEAR は起点変更時間の 28 年であり，LEAP_DAYS は 28 年の期間内の閏日の日数を示す．これらの値から計算される DATE_OFFSET_SEC は，起点変更時間の 28 年間（閏日含む）を秒で表した値であり，883,612,800 秒となる．これらの値を用いてラッパー関数を構成する．

図 11 に，settimeofday 関数に対するラッパー関数を示す．この関数ではシステム時刻をライブラリの settimeofday 関数を利用して設定するが，その際に 1998 年起点とするために，28 年分時間（秒）を減算して設定する．なお，第 1 引数の wrapper_timeval 型は，timeval 構造体のメンバである time_t 型の変数（tv_sec）を，unsigned int 型に変更して作成した型である．

```
int wrapper_settimeofday(const wrapper_timeval * tv , const struct timezone * tz)
{
 struct timeval tv_tmp, *tvp;
 int ret;
 if (NULL == tv)
 {
  tvp = NULL;
 }
 else {
  tv_tmp.tv_sec = (time_t)(tv->tv_sec - DATE_OFFSET_SEC);
  tv_tmp.tv_usec = tv->tv_usec;
  tvp = &tv_tmp;
 }
 ret = settimeofday(tvp, tz);
 return ret;
}
```

**図 11　settimeofday 関数に対するラッパー関数**

システム時刻の要素別時刻情報を取得する関数では，取得した要素を表示等に用いている．正しい時刻を表示するためには OS 等ライブラリで得られる要素別時刻情報に 28 年加算する必要がある．図 12 に localtime_r 関数に対するラッパー関数を示す．

```
struct tm* wrapper_localtime_r (const time_t *clock, struct tm *result)
{
 struct tm *tm_tmp;
 if ((tm_tmp = localtime_r(clock, result)) != NULL)
 {
  result->tm_year += DATE_OFFSET_YEAR;
 }
 return tm_tmp;
}
```

**図 12　localtime_r 関数に対するラッパー関数**

なお，settimeofday()関数と対になる gettimeofday()関数では，time_t ポインタ型の引数を取り，現在時刻が取得できるが，time_t 型の変数にはこのシステム内部では 1998 年起点の情報を保存している．このシステムでは 1998 年起点の情報を扱うためラッパー関数（wrapper_gettimeofday）は用意したが，gettimeofday()関数が返す 1998 年起点の値をそのまま利用し，特に起点解釈の変更は行わない．

## 2.6 修正作業

### 2.6.1　テスト結果

表 8 に単体テスト，結合テストの項目数とそのテストでの不具合項目数を示す.

**表 8　テスト項目数と結果**

| | テスト項目数 | 不具合項目数 |
|---|---|---|
| 単体テスト | １６５ | ０ |
| 結合テスト | １０３ | １ |

単体テストは，関数毎に作成しているフローチャートに基づき，パス網羅のテストと関数出力のチェックを実施した．結合テストでは，当該機器外部から入力する 2038 年以降の時刻データを用意し，当該機器の状態やユーザ操作の組み合わせと共に機能が動作するユースケースを確認した.

結合テストで発生した 1 か所の不具合項目は，エラーで返却される値である“-1”を時刻情報であると誤って判断して，別の時間との足し算を実行したため，所望の動作をしないというものであった．他に不具合項目は無く，当初の案通りの設計，実装が計画通りに行えた.

### 2.6.2　実績工数

ここで開発時の規模と工数を示す．なお，2038 年問題への対応は他の機能拡張などを含むプロジェクト全体の要件の一部であり，純粋に 2038 年問題だけの作業工数（ここでは単に作業工数と呼ぶ）を得ることはできなかった．一方で，プロジェクト完了後の意見交換において，本問題の対応に特段の困難が発生せず，他の要件の開発作業と同程度の作業効率であったとの開発担当者の意見があった．このことから，プロジェクト全工程での開発効率（step/人時）と 2038 年問題に要した開発及び修正規模（合わせて修正規模と呼ぶ）を用いて，作業工数を算出できると判断した．作業工数は次式により算出する.

$$\text{作業工数} = \frac{\text{修正規模}}{\text{プロジェクト全工程での開発効率}}$$

表 9 に各ライブラリ関数に対して，作成したラッパー関数の規模，そのライブラリ関数を呼び出している箇所数，呼び出しをラッパー関数に変更するための修正規模を示す.

なお，自社では修正規模を求める際，修正による影響範囲も加味することとしている．具体的には，修正によりテストを行う必要のあるソースコードの範囲を影響範囲としている．本プロジェクトでは単体テストの最小単位を関数としているため，ラッパー関数の呼

び出し元関数をテスト対象とした．よって修正規模は，ラッパー関数を呼び出している関数全体の行数としている．

**表 9　修正規模**

| 項番 | ライブラリ名 | ラッパー関数規模（行） | 呼出し箇所数 | 呼出し部の修正規模（行） |
|---|---|---|---|---|
| 1 | ctime | 11 | 0 | 0 |
| 2 | ctime_r | 11 | 0 | 0 |
| 3 | gettimeofday | 6 | 25 | 1262 |
| 4 | gmtime | 9 | 0 | 0 |
| 5 | gmtime_r | 9 | 1 | 21 |
| 6 | localtime | 9 | 2 | 12 |
| 7 | localtime_r | 9 | 30 | 1131 |
| 8 | mktime | 22 | 10 | 307 |
| 9 | settimeofday | 15 | 2 | 127 |
| 10 | strftime | 81 | 2 | 48 |
| 11 | timegm | 22 | 1 | 28 |
| 12 | timelocal | 22 | 0 | 0 |
| 13 | 共通関数等※ | 38 | － | － |
| 合計 | | 264 | 73 | 2936 |
| 修正規模 | | 264 + 2936 = 3200 行 | | |

※複数のラッパー関数で共用する関数と include，define 文を含む．

また表 9 中には，呼出し箇所数が 0 の関数も記載している．これは，同じソフトウェアからコンパイルスイッチにより複数の機器（モデル）を開発できるようにしていることに起因する．本プロジェクトでは使用しないが，別モデルの開発プロジェクトでは使用する関数を表 9 の使用関数に含め，本プロジェクトでの修正規模は，0 としている．

次に，2038 年問題を含む本プロジェクトでの開発規模を，その規模の開発に費やした工数で割り，プロジェクト全工程での開発効率を求める．本プロジェクトでの開発効率は，以下の通りであった．

開発効率：3.45 step/人時

前述の修正規模と上記の開発効率より，2038 年問題の修正対応に要した工数は，

3200 / 3.45 = **927.54 人時**

となり，1 日 8 時間，月 20 日間と仮定して人月であらわすと，

927.54 / (8*20) = **5.80 人月**

となる．

## 2.7 議論

### 2.7.1 ソフトウェアにおける日時の扱い

通常，計算機システムでは，タイマーなどによる定期的な割り込み信号やネットワークから得られる情報などに基づき，起点時刻からの経過時間を OS が管理している．この経過時間は，必要に応じて OS 自身や他のアプリケーションプログラムに，経過時間そのままの形や，人間が理解できる暦表示（例えば 2018 年 9 月 24 日午前 11 時 34 分 56 秒など）に変換して提供される．経過時間を記憶する時間型の変数が十分な語長を有していれば，長大な経過時間を表現することができる．しかし，古い時代に設計された OS や計算機システムでは，長期の展望が欠けていたり，ハードウェアやコスト制約が厳しいなどの理由で，不十分な語長で開発，利用されたものが多数あり，本章で議論したような対応が必要となっている．

近年，このような問題は広く認識されつつあり，OS やプログラミング言語レベルで対応する動きがある．例えば，多くの OS においては，2.5.1 項における対策案(a)，すなわち，経過時間を 64bit で表現する対策が取られている．Microsoft Windows では，経過時間を time_t 型で表現しており，Visual C++ 2005 より前のバージョンの Visual C++と Microsoft C/C++ではそれが 32bit であったため同様な問題が存在していたが，現在は 64bit で表現されている[2-10]．また，NetBSD も対応する全アーキテクチャで time_t を 64bit で表現するよう修正が行われた[2-11]．Apple の macOS や iOS では，時間を 2001 年 1 月 1 日 UTC からの経過秒数で表現しており[2-12]，現在は 64bit アプリケーションしか許容しないよう規約で定めている[2-13][2-14]．このように，経過時間を 64bit で表現する方法は，実用上問題を解決する確実な方法であるが，ハードウェアや API の変更を伴うため，今回のような組込みシステムでは安易に適用しにくい．FreeBSD では，32bit OS と 64bit OS が同時に保守されており，引き続き 32bit OS も利用し続けることができるが，本章で述べたような 2038 年問題が生じる．64bit OS への移行はコスト制約上難しく，本プロジェクトでは採用しなかった．

このように，本プロジェクトのような 64bit OS への移行が困難な組込みシステムには，本章での知見が役立つものと思われる．

### 2.7.2 修正効率

本プロジェクトでは，変更箇所を小さくするように設計方針を定めた．仮に，OS 部の変更を含めた time_t 型変数のオーバーフローの回避を行った場合，修正が必要な箇所は，表 5 の調査結果を用いて概算すると，

2546 / 945 ＝ 2.69 倍

となる．修正難易度に差は無いと考えられるため，必要工数は，

5.80 人月 × 2.69 ＝ 15.60 人月

かかることになる。設計方針の選択により，約 2.7 倍の違いが発生することになる．

## 2.8 修正作業の評価

プロジェクト完了後に，開発メンバによる本プロジェクト活動に関する振り返りの会議を実施した．会議では，2038 年問題については事前に修正箇所や影響範囲の確認を十分に行ったため，大きな混乱が無かったことを確認した．

対応手段について，特に不明な点がなく作業規模や作業量が開発者に想像できたため，あらかじめ設定したスケジュールで対応を終えることができた．OS や標準ライブラリの改変も行っていないため，今後の OS，標準ライブラリのアップデート時にも本問題に関して特別な対応を行う必要はない．完成したソースコードは，リリースされた製品に組み込まれて正常に動作している．

なお，今回選択した手法では 1998 年以前を表現することはできない．そのため，変更した起点以前の情報を管理する機器では選択できない手法である．また，epoch を変更する前後でファイルを作成して参照する場合には，ファイルの時刻の扱いを適切に行う必要がある．当該機器は，現在と未来の情報のみを扱うため，この手法が選択できた．今回の事例では，システム時刻を設定する際に，wrapper_settimeofday 関数の呼出し元で 1970 年起点の時刻を扱う箇所がある．システム時刻の設定で 1970 年起点の時刻を使用する場合は，漏れなく洗い出し，オーバーフローの可能性を検討する必要がある．今後 2066 年に同様な問題を生じるが，本手法を用いればラッパー関数の変更により，簡単に対応が可能である．また 2.5.1 項で検討した案(a)～案(c)については，対応工数とメンテナンスコストのため，当該機器の開発では選択できなかった．

時刻情報の取り扱いや時刻情報の入出力など，当該システム特有のものではなく，組込み以外のコンピュータシステムでも一般的に用いられるものである．またラッパー関数を用いる今回の手法についても，特にこのシステムでなければ実現できないものではない．選択した手法はこの機器以外でも適用可能である．

## 2.9 おわりに

本プロジェクトにおいては，使用可能な工数や納期の制約により，案(d)に基づき，システム時刻の起点を 28 年遅らせ 1998 年とする手段を選択した．調査範囲や調査内容を明確にした上で，地道な調査を実施し，最終的にテストで 2038 年以降の環境を用意し，問題が発生しないことを確かめることができた．

日付，時刻のオーバーフロー問題は，64bit のシステムでは現実的には検討の必要がない[2-15]．現在稼働している 32bit のシステムを 64bit 化する計画がある場合は， 2038 年までにオーバーフローが起こらないことを確認の上，計画に従って移行すれば問題とはならない．しかし，64bit 化を予定していない 32bit の比較的廉価なシステム，かつ稼働期間の長い 2038 年時点で稼働しているシステムでは，何らかの対策を打つ必要がある．

本章で報告した方法は，この問題に対して比較的安価に対応できるので，機器での 2038 年問題への対策案の検討，開発期間・コストの見積もり等に役立てば幸いである．他の情報機器の 2038 年問題にも適用していきたい．

# 3. 32bit UNIX システムの 2038 年問題に対するプログラム修正法の提案

## 3.1 はじめに

32bit の UNIX ベースの OS では，時刻情報を time_t 型と呼ぶ 32 ビット符号付き整数データとして 1970 年 1 月 1 日 0 時 0 分 0 秒（UTC）を起点（epoch）とした経過秒で管理している[3-1]．1970 年からおよそ 68 年後の 2038 年にデータの最大値を超えて桁あふれを起こすため[2-4]，アプリケーションの動作にさまざまな影響を及ぼす恐れがある（以下 2038 年問題とする）[2-5].

この問題への対策として，2 章では epoch を 1970 年から 1998 年に変更し，桁あふれの発生を 2038 年から 2066 年に遅らせる方法を提案した[3-2]．この提案は実際に 20 年間動作保証する組込みシステム開発に適用され，2020 年現在，市場で稼働している．

しかしこの変更を実現するためには，時刻に関わるライブラリ関数，それぞれの入出力値や一時的に時刻情報を保存する変数の全てをソースコードから抽出し，それぞれ修正対象箇所かどうかを人手で確認する必要があった．

本章では，ソースコードからこの修正箇所を効率良く特定する手順を一般化した．まず，開発対象から修正箇所を抜き出すためにプログラムスライシング手法[1-7] [1-8]を適用する．その後，得られたスライスより，対象とするアプリケーションの動作条件，制約事項により修正箇所を絞り込む．本章では，実際にこの手順を FreeBSD 上の 3 つのコマンド date, stat, touch）に適用し，その効果を確認した．

本章の事例も，2 章同様に修正箇所の特定に設計情報が利用できない例である．策定した修正仕様を設計，実装に反映するため，これまで報告されていない，2038 年問題に対するプログラムスライシング手法の適用効果を議論する．

## 3.2 提案手法

### 3.2.1 問題点と課題

2038 年問題は，32bit システムにおいて 32bit の符号付整数型で扱う時刻管理情報が桁あふれを起こすことが原因で起こる．この問題への対処法として，①time_t 型を 64bit 化する，②time_t 型を 32bit 符号無し整数型に変更する，③time_t 型変数の評価箇所で桁上がりを考慮して 32bit に収める，④epoch を変更し，影響を受ける部分をラッパー関数等で修正するという 4 つの方法が考えられた[3-2].

筆者らが行った既報の提案では，開発量や修正による影響範囲の観点から，④を採用した．この選択により，標準ライブラリやデバイスドライバ，OS 等の変更を一切行わず，アプリケーション部の修正だけで対応が可能であった．

本章ではこの修正手順を明確化，一般化し，修正対象の関数や命令を効率よく見つける手法を提案する．

### 3.2.2　概要

修正対象を効率よく見つける手法を導くため，FreeBSD のツールを対象に 2038 年問題の修正方法を検討した．対策方針は既報の開発と同様に，時刻情報のオーバーフロー回避のため，　epoch の起点をちょうど 28 年後ろにずらして，1998 年 1 月 1 日 0 時 0 分 0 秒（UTC）に変更することとした．また修正による影響範囲を小さくするため，3.2.1 項で示したものと同様の修正方針とし，デバイスドライバ，標準ライブラリや OS 部の改変は行わないこととした．図 13 に手順を示す．

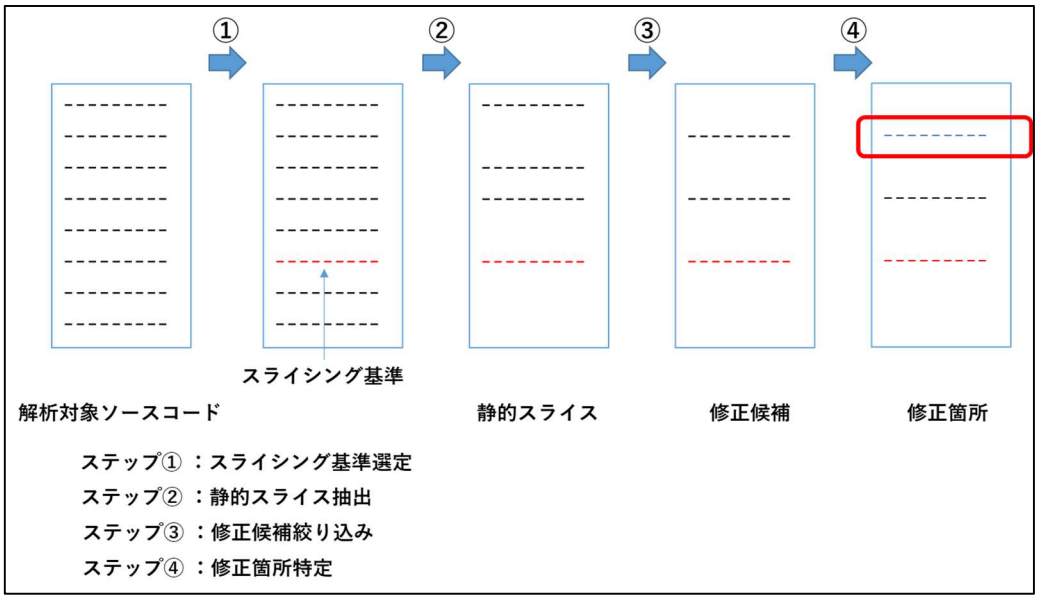


**図 13　修正箇所の特定手順**

**ステップ①　スライシング基準の選定**

time_t 型変数が該当プログラムの実行系列で最後に利用される箇所を見つけて，スライシング基準とする．実行系列での最後の利用箇所については，対象によっては一意に決まらない場合もあるため，コードレビュー等で最後に利用される可能性のある命令を全て抽出する．

ステップ②　静的スライスの抽出

求めたスライシング基準より，データ依存行，制御依存行を求めて，静的スライスを得る．ただし，OS 部，標準ライブラリ，デバイスドライバ等の関数の呼出し文については，入出力の仕様からステップ③と同様の判断基準で，静的スライスとして含めて更に伝搬させるか否かを判断する．静的スライスの抽出には，CodeSonar[3-3]等のツールを用いると効率的だが，本研究では対象が比較的小規模なので目視により行った．

ステップ③　修正候補の絞り込み

ステップ②で得られた静的スライスの各文（スライス文と呼ぶ）から修正箇所を絞り込むため，2038 年問題に関係しないスライス文を対象から除外する．除外する箇所は，呼び出し関数，アプリケーションの機能，処理内容の 3 つの視点で選定する．

（1）呼び出す関数の機能

時刻情報を扱わない関数呼び出しを持つスライス文は，対象外とする．例えば実行制御に関わっている場合でも，時刻情報を扱わない関数は対象外とする．

（2）アプリケーションの機能

アプリケーション中の特定の機能が，直接的，間接的に時刻情報に影響しない場合には，その機能を実現しているスライス文を修正対象外と判断する．例えばコマンドライン引数を取るアプリケーション中で，時刻情報の更新に影響しない引数を処理するスライス文は，修正対象外とする．

（3）スライス文の処理内容

time_t 型変数への，符号付き整数型 32bit を超えない定数の代入箇所は，修正対象外と判断する．例えば time_t 型変数への初期値 0 の代入箇所は，修正対象外とする．

ステップ④　修正箇所の特定とその修正方針

修正候補の各文に対して，修正箇所の特定を以下の手順で行う．

（1）標準ライブラリ関数の呼び出し

time_t 型変数に関連するライブラリ関数は，ラッパー関数から間接的に呼び出すように変更する．関数 mktime を例に示す．mktime は年月日と時刻の情報を表す struct tm 型の値の引数を取り，epoch からの経過秒である time_t 型の値を返す関数である．アプリケーションから mktime を直接呼び出す代わりに，mktime のラッパー関数（関数 wrapper_mktime とする）を用意し，wrapper_mktime 内で，epoch の起点変更分（28 年）を年の情報から減算した上で，mktime を呼び出すようにする．

（2）時刻情報を評価する箇所

時刻情報を直接評価する命令がある場合は，epoch 変更前の時刻情報を扱う箇所か，epoch 変更後の時刻情報を扱う箇所かを識別する．その上で，epoch 変更後の時刻情報を扱う場合には，起点変更分（28 年）を考慮した評価に改める．

## 3.3 修正候補の絞り込みの具体例

FreeBSD のコマンド touch を例に，本手法のステップ③である修正候補の絞り込みについて具体的に示す．

（1）呼び出す関数の機能

関数 main から呼び出す関数の中には，2038 年問題の解決に無関係な関数もある．このような関数の例として，図 14 に main から呼び出す関数 stime_darg の抜粋を示す．332 行目，および 335 行目で関数 isdigit を呼出しているが，isdigit は入力の文字列が数値かどうかを評価する関数であり，時刻の情報は扱わない．よって，isdigit の呼出し部分および isdigit 自体は修正対象外とする．

```
332     if ((*p == '.' || *p == ',') && isdigit((unsigned char)p[1])) {
333             p++;
334             val = 100000000;
335             while (isdigit((unsigned char)*p)) {
336                     tvp[0].tv_nsec += val * (*p - '0');
```

図 14　対象外の関数呼出しの例

（2）アプリケーションの機能

図 15 に関数 main の抜粋を示す．87 行目の関数 timeoffset は，-A オプションを指定した際に呼ばれる関数であり，経過秒を返す関数である．時刻情報ではあるが，年単位の情報を扱わないため，修正対象から除外する．

```
79    Aflag = aflag = cflag = mflag = timeset = 0;
80    atflag = 0;
81    ts[0].tv_sec = ts[1].tv_sec = 0;
82    ts[0].tv_nsec = ts[1].tv_nsec = UTIME_NOW;
83
84    while ((ch = getopt(argc, argv, "A:acd:fhmr:t:")) != -1)
85      switch(ch) {
86      case 'A':
87        Aflag = timeoffset(optarg);
88        break;
```

図 15　アプリケーション機能による確認例

（3）スライス文の処理内容

図 14 の 79 行目から 82 行目のように，固定の初期値を変数に代入する処理では，年情報のオーバーフローは起こらないため，修正対象から除外する．

## 3.4 修正箇所の特定の具体例

3.3 節と同様に，本手法のステップ④である修正箇所の特定について，具体的に示す．

（1） 標準ライブラリ関数の呼び出し

図 16 に関数 main より呼び出される関数 stime_arg1 を示す．272 行目に，当該関数内で最後に時刻を扱う標準ライブラリ関数である mktime がある．この関数については，図 17 のようにラッパー関数 wrapper_mktime を用いて epoch の起点を変更する．これにより，main では epoch を修正した時刻を扱えるようになる．このように各ライブラリ関数について，年でのオーバーフローの発生が起こり得るかどうかを確認し，オーバーフローの可能性が無い場合には修正対象としない．

（2） 時刻情報を評価する箇所

図 16 に示す関数 stime_arg1 では，254 行目に t->tm_year に値を設定する命令がある．この命令が修正対象かどうかを検討する必要がある．この命令実行前には time_t 型の変数値に対して加減算を行っていない．時刻を修正していないため，yearset と比較する値も修正前のプログラムと同等であり修正対象外とする．

```
219  static void
220  stime_arg1(const char *arg, struct timespec *tvp)
221  {
     ...
252              yearset = ATOI2(arg);
253              if (yearset < 69)
254                 t->tm_year = yearset + 2000;
255              else
256                 t->tm_year = yearset + 1900;
257                 }
258              t->tm_year -= 1900;   /* Convert to UNIX time. */
259           /* FALLTHROUGH */
260       case 8:                 /* MMDDhhmm */
261              t->tm_mon = ATOI2(arg);
     ...
271       t->tm_isdst = -1;     /* Figure out DST. */
272       tvp[0].tv_sec = tvp[1].tv_sec = mktime(t);
273       if (tvp[0].tv_sec == -1)
274          goto terr;
     ...
281  }
```

**図 16　関数 stime_arg1**

```
219   static void
220   stime_arg1(const char *arg, struct timespec *tvp)
221   {
      ...
271           t->tm_isdst = -1;      /* Figure out DST. */
272           tvp[0].tv_sec = tvp[1].tv_sec = wrapper_mktime(t);
273           if (tvp[0].tv_sec == -1)
274               goto terr;
      ...
281   }
      ...
...   static time_t
      wrapper_mktime(struct tm *ptm)
      {
          time_t ret;

          ptm->tm_year -= 28;
          ret = mktime(ptm);

          return ret;
      }
```

図 17　stime_arg1 の修正箇所

## 3.5 修正後の実行例

FreeBSD のコマンド touch，stat，date のソースコードを修正した．これらの実行結果を，図 18，図 19 に示す．

```
ooe@FreeBSD:~/test % touch -t 204001010000 aaa
touch: out of range or illegal time specification: [[CC]YY]MMDDhhmm[.SS]
```

```
ooe@FreeBSD:~/test % touch.new -t 204001010000 aaa
ooe@FreeBSD:~/test % stat.new -F aaa
-rw-r--r-- 1 ooe ooe 0 Jan  1 00:00:00 2040 aaa
ooe@FreeBSD:~/test %
```

図 18　実行結果（touch, stat）

修正前のコマンド touch で 2040 年 1 月 1 日 0 時 0 分のファイルを作成しようとすると，エラーが返る．修正したコマンド touch.new で 2040 年 1 月 1 日 0 時 0 分のファイルを作成した場合，ファイルの作成に成功し，同様に 2038 年問題に対応したコマンド stat.new でファイルの作成年月日を確認すると， 2040 年 1 月 1 日 0 時 0 分となっており，正常に実行できたことが確認できる．

コマンド date.new の実行結果を図 18 に示す．date.new では，システム時刻をあらかじめ 1998 年起点の時刻に変更する（28 年分減算する）．修正前のコマンド date で現在時刻を表示させると，2020 年現在の 28 年前である 1992 年を表示する．epoch を修正した date.new

では，期待通りに 2020 年を表示する．また修正前の date で 2040 年 1 月 1 日 0 時 0 分に日付を変更しようとするとエラーが返り，修正した date.new では 2038 年を超える 2040 年に正しく変更できる．

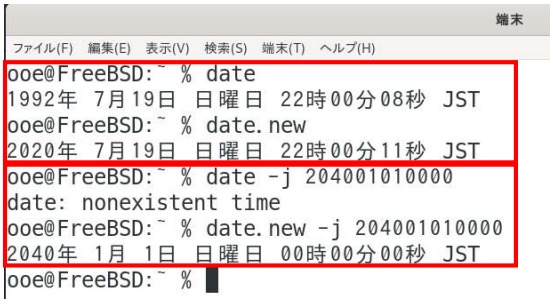


図 19　実行結果（date）

## 3.6 議論

2038 年問題に対応する際，修正コストを極力抑えるため，修正規模を小さく，またなるべく簡単に対応したい．既報のやり方で 2038 年問題に対応する場合，影響する可能性のある，プログラム中の全ての箇所を確認する．複雑なことは行わない代わりに，標準ライブラリ関数も含めた確認が必要だったため，確認の必要なプログラム規模は大きく，調査コストが大きくなった．調査範囲を絞り込むため，grep 等の正規表現を用いた検索手法の利用も考えられるが，int 型で宣言した一時変数に time_t 型の値を代入する場合や，ポインタ変数からエイリアスとしてアクセスする場合などには，修正対象箇所として抽出できない．

プログラムスライシングを用いて修正対象箇所を限定すれば，調査するプログラムの規模を抑えた上で修正箇所を漏れなく抽出できる．しかし，対象とするプログラムによっては，プログラムスライシングでは修正候補箇所の絞り込みが不十分な場合があり，確認の必要なプログラム規模が大きくは減少しないおそれがある．

そこで提案手法では，修正候補のスライス文からプログラムで 2038 年までの時刻情報を扱う箇所と，時刻に影響するライブラリ関数をコールする箇所に着目し，それ以外の箇所は修正対象外として確認対象から外すことにした．提案手法により，確認が必要な箇所が減少する様子を表 1 に示す．表中，静的スライス結果は手順のステップ②の結果，残るソースコードの行数であり，修正候補の確認行数はステップ③の結果，残るソースコードの行数である．

表 10　提案手法による調査規模の減少

| コマンド | 元の行数* | 静的スライス結果 | 修正候補の確認行数 | 修正行数 | 追加行数 |
|---|---|---|---|---|---|
| touch | 303 | 273 (10%) | 178 (41%) | 18 | 15 |
| date | 871 | 564 (35%) | 258 (70%) | 11 | 35 |
| stat | 771 | 565 (27%) | 249 (68%) | 3 | 10 |

*空行，コメント行除く．() 内は元の行数からの削減率.

表 10 に示す通り，コマンド touch で 41%，コマンド date で 70%，コマンド stat で 68%，確認が必要なプログラムの量を減少させることができた．より大規模なプログラムで，時刻情報に関わるプログラム規模の割合が相対的に下がる場合には，更に大きな効果を見込むことができる．

また，touch と同様に date でも関数 mktime を用いており，touch で作成したラッパー関数がそのまま利用できる．時刻を扱うライブラリ関数のラッパー関数群をあらかじめ用意すれば，更に修正コストを抑えた 2038 年問題の解決が期待できる．

一方で本手法は，プログラムスライシングを前提とした手法であるため，スライシング基準を適切に設定できないと，修正対象から漏れてしまう．シグナルハンドラ中の処理などは，スライシング基準選定時に別に確認する必要がある．

なお，Linux カーネル 5.6 では，time_t 型変数の 64bit 化が行われている（2020 年 5 月現在）[3-4]．しかし，time_t 型の変数の 64bit 化が行われた場合でも，32bit システム下で作成したファイルの属性等を含めて考えた場合，単純には 2038 年問題は解決しない．特に稼働中のシステムに対するモダナイゼーションを検討する際，コスト面等の事情によりハードウェア更新やリビルドなどの大規模な修正が選択できない場合には，対象アプリケーションのみを修正対象とする本手法が代替案となり得る．

## 3.7 おわりに

32bit Unix システムの 2038 年問題の解決法の一つを提案し，FreeBSD 上の 3 つのツールに適用して，その効果を確認した．

2038 年問題の解決にプログラムスライシングを適用し，得られたスライスから更に調査範囲を絞る手順により，調査が必要なプログラムの規模を小さくすることができた．これにより，2038 年問題の対応の効率化が期待できる．

プログラムスライシングを利用する際には，C 言語の場合にはある変数を，別の複数のポインタ変数から参照するエイリアス問題に注意が必要となる．本章の事例ではスライスを人の目で確認しており，当該問題は見られなかった．

2038 年問題への対応方法はいくつか考えられるが，特に対応コスト抑制の優先度が高い場合には，本章で紹介した手法が，有力な選択肢となり得る．

また，本章で示した改良保守の効率化の手順は，2038 年問題以外にも有効である．2038 年問題の場合には，time_t 型変数のオーバーフローを関心事として修正箇所を特定したが，各保守案件での関心事をソースコード上から見つければ，同じ手順でソースコードの調査規模を削減でき，改良保守効率の向上が期待できる．

# 4. 予約受付端末の他サービス転用に向けたソフトウェア改造事例

## 4.1 はじめに

ソフトウェアの改造，再利用により当初の目的外の別の製品を開発する中で，改造元のソフトウェアを修正した完全化保守の例を示し，設計情報を用いた効率の良い保守開発について議論する．

本章の事例は，修正箇所の特定に設計情報が利用できる例である．設計情報を用いることで，修正箇所の特定効率がどのように変わるかを議論する．

当初，商業施設の予約受付端末として開発した製品（以降，これを**予約受付端末**と呼ぶ）を用いて，別の用途で用いる自動券売機を開発した（以降，**券売機**と呼ぶ）．図 20 に物理的な機器構成の概要を示す．

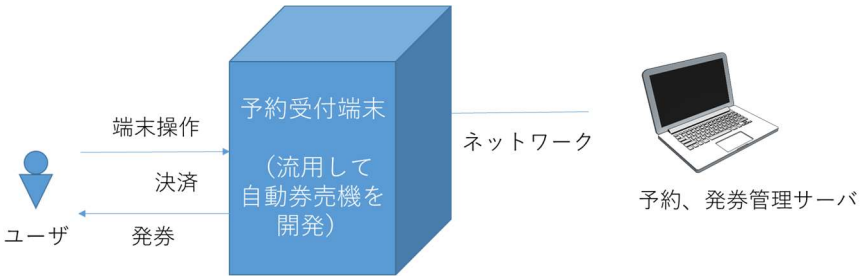


**図 20　機器構成概要**

図のユーザが端末操作を通して購入対象等を選択し，端末は予約情報や発券に必要な情報をサーバに問合せ，購入可能な場合にはユーザに決済手続きを促し，チケット等を発券する．

施設の予約受付では，ユーザは，予約受付端末を用いてあらかじめ予約した施設を選択し，選択した内容に応じて現金やクレジットカードでの決済を行う．端末では購入内容に応じて各種チケットやレシートの印刷を行う．この一連の手続きは，券売機の場合は，目的のチケット種類を選択させ，決済を行い，チケットを印刷するという一連の流れと同じである．図 21 に，2 つのサービスの共通機能を示す．

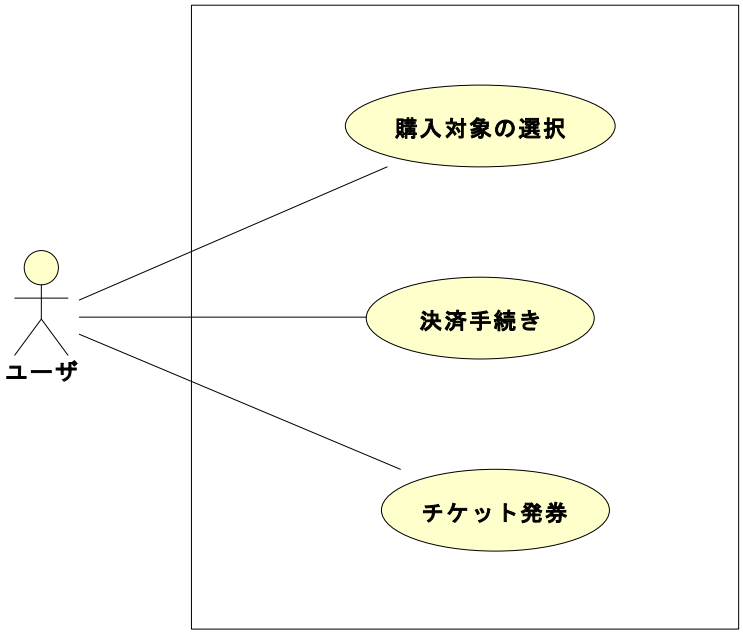


図 21　2 つのサービスの共通機能

購入対象を選択し，決済手続きを行い，チケットを発券する一連の機能の中で，製品毎に大きく異なる部分は，主に購入対象の選択機能である．現金，クレジット決済やチケットの発券で，必要なハードウェアを動作させる順序やタイミング等は共通である．このため，製品毎に大きく異なる部分のみサービスにあわせて開発し（可変部），共通部分は固定部として流用できるようなソフトウェアのアーキテクチャを採用した．

ここで，当初想定のなかった問題が発生し，可変部と固定部の切り分けを見直す必要が生じた．本章では，ソフトウェア構造の理解を用いて効率的にこの問題を解決した事例を示す．

## 4.2 ソフトウェア構成

対象システムは，あらかじめ製品の複数サービスへの適用を想定しており，ソフトウェアのアーキテクチャとしては，Layers アーキテクチャ[4-1]を採用していた．サービスにより異なる購入対象の選択機能は，UI（アプリケーション）層で実現する．図 22 に，当該製品のソフトウェア構成図を示す．

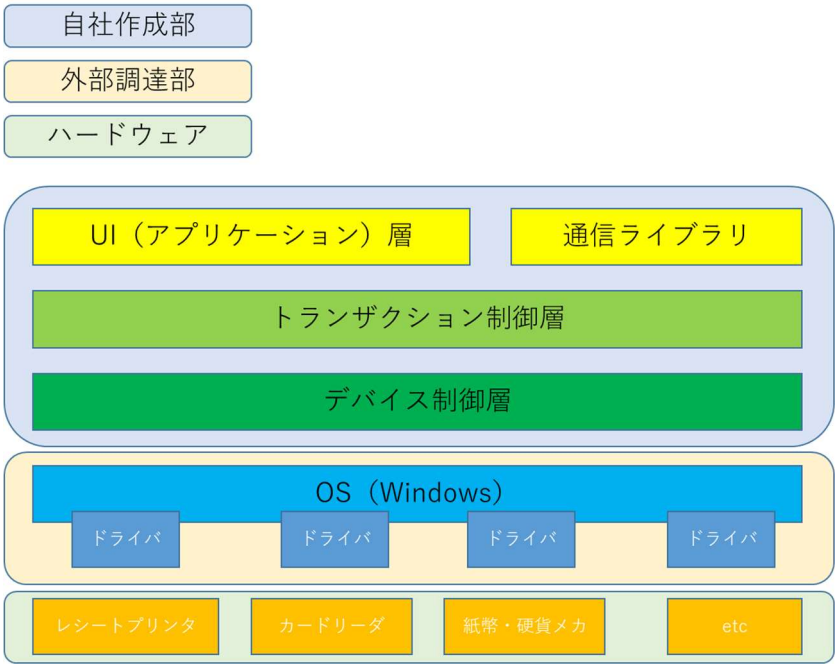


**図 22　ソフトウェア構成図**

図中の自社開発部の各ブロックの主な役割を以下に示す.

表 11　各ブロックの主な役割

| ブロック名 | 役割概要 |
|---|---|
| UI 層 | 購入対象のサービスをユーザに選択させる. |
| 通信ライブラリ | サービス毎に必要となる各種サーバとの接続を担う. |
| トランザクション制御部 | UI 層より指示を受け，現金収受やカードの読込み，発券操作などハードウェアを動作させる一連の手順やタイミング制御を担う. |
| デバイス制御部 | 各デバイスを制御する. |

各ソフトウェアブロックで，UI 層は可変部と位置付けており，サービスに応じて積極的に変更するブロックである．トランザクション制御部，デバイス制御層，OS，各種ハードウェア（デバイスドライバ含む）を製品の共通（固定）部として位置付けており，ここは極力変更しないことを設計方針として定めている．このことにより，拡張開発時の改変箇所を限定し，短納期，低コストでのサービス提供を行っている.

## 4.3 インタフェース例

UI 層からの指示により，トランザクション制御層やデバイス制御層がどのように動作するかを，現金決済処理を例に示す.

図 23 に，現金による決済処理を例に，UI（アプリ）層と，トランザクション制御層間のインタフェースを示す.

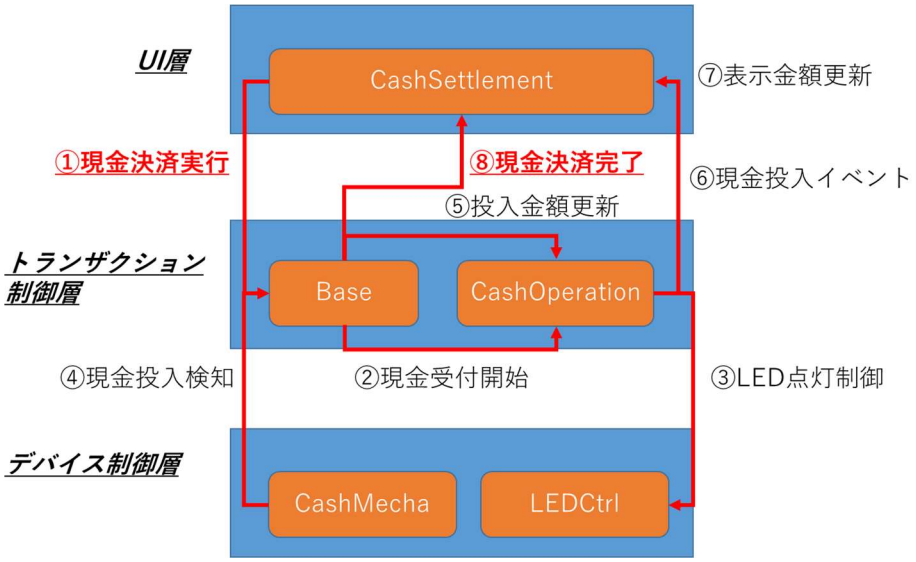


**図 23　現金決済処理時のインタフェース例**

UI 層では，ユーザの画面操作に従い，トランザクション制御層に対して現金決済処理を同期呼び出しで実行する．トランザクション制御層では，LED 点灯制御を行い，デバイス制御層からの現金挿入を待つ．デバイス制御層では，ユーザが現金を挿入した際，トランザクション制御層に対して現金投入検知メッセージを送る．トランザクション制御層では現金投入検知メッセージを受けて，UI 層に現金投入イベントを通知する．現金投入イベントの通知を受け，UI 層では，投入金額の表示の更新を行う．決済金額・投入金額の比較やつり銭の計算，返却等は，デバイス制御層からのメッセージを受け，トランザクション制御層が，自身で持つ状態で管理しながら完結し，決済完了状態に移行した後，UI 層に制御を返す．

まとめると，UI 層とトランザクション制御層の間の基本のインタフェースは，

（1）UI（アプリ）層からトランザクション制御層へのサービス要求

（2）トランザクション制御層から UI（アプリ）層へのイベント通知（非同期）

（3）トランザクション制御層から UI（アプリ）層へのサービス結果通知

から成り立っている．（1）のサービス要求は，同期インタフェースであり，要求の結果が（3）として返る．（2）は，サービス中の状態変化などのデバイス制御層やトランザクション制御層に起因する通知であり，通知を受けた UI（アプリ）層では，通知内容に応じて表示の更新を行っている．

図 24 にトランザクション制御層が公開するサービスの一覧を示す

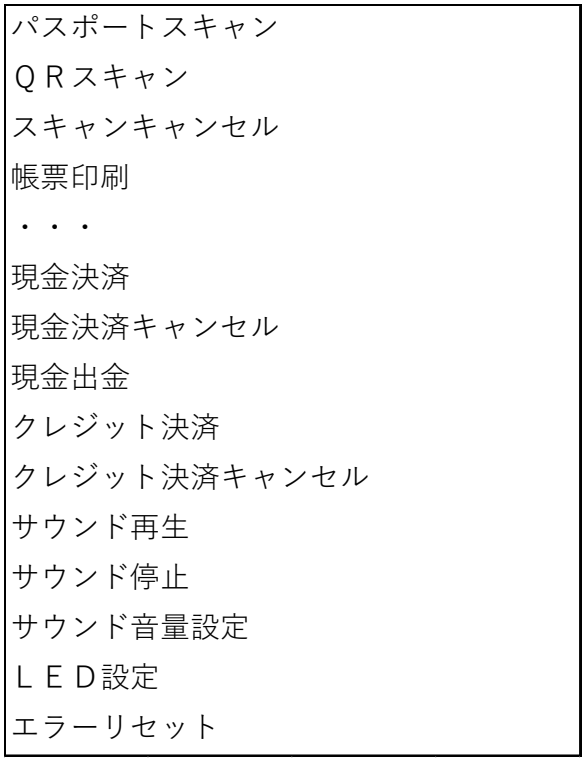


**図 24　トランザクション制御層が公開するサービス（抜粋）**

## 4.4 問題点と解決方法

あるサービス向けに開発した製品を別のサービス向けの製品に適用する場合には，最初の製品開発時には問題とならなかったことが問題となる場合がある．以下に問題点の例と解決方法を示す.

### 4.4.1　問題点

予約受付端末でレシートを複数枚印字する場合には，枚数分の画像をあらかじめ１つのリストにする．その上で，UI 層からトランザクション制御層の印字 API を 1 度だけ用いて画像リストを渡し，トランザクション制御層では，画像印字の間，ブザーを鳴動するように制御している.

一方，券売機ではレシートプリンタを用いて券の発行を行っており，複数枚の券を発行中に異常が発生した場合にも異常発生前の発券を有効とするため，券を 1 枚ずつ印刷する仕様であった．この仕様を実現するため，複数枚の券を発行する場合には，枚数と同じ回数だけ連続でトランザクション層の印字 API を呼び出す設計とした.

この結果，券売機ではブザー鳴動制御（開始・停止）が発券毎に複数回実行され，ブザーが不規則に鳴ってしまう現象が発生した．問題の現象を図 25 に示す．

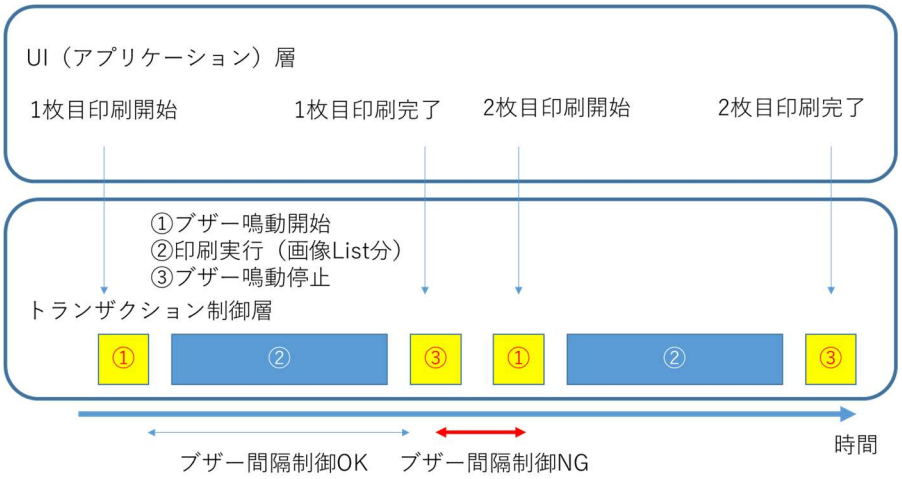


**図 25　問題の現象**

印刷のための API を連続で呼び出す想定は予約受付端末にはなく，印刷とブザー鳴動を同時に制御する設計となっていることが，券売機の開発では問題となった．将来的に，券売機の考え方に近い製品開発時にもこのことが問題となる可能性があるため，改造を行うことにした．

### 4.4.2　解決方法

プログラムの調査では，調査箇所を計画的に定め，調査結果を記録するといった，秩序だった体系的な方法が修正箇所を見つけるのに有効とされている[4-2]．ここでもアーキテクチャの情報を利用して，調査箇所を特定することとした．

ブザーの開始，終了制御とサウンドデバイス制御を全てトランザクション制御層で行っていることはわかっているため，トランザクション制御層で修正が必要なことがわかる．更に印刷とブザー鳴動の処理を行うモジュールは Transaction_X にのみ存在するため，このモジュールを修正対象として特定できる．

まず UI 層が，印刷枚数（印字 API の呼び出し回数）が複数の場合にブザー制御をする旨を，トランザクション制御層に新設するブザー制御のラッパーモジュールに設定する．トランザクション制御層では，ブザー鳴動制御を直接呼び出さず，ラッパーモジュールを介して呼び出す．ラッパーモジュールでは UI 層から設定された状態に応じて，ブザー制御を行うか否かを判断して制御を行う．図 26 に改造前後のソフトウェア構造を示す．

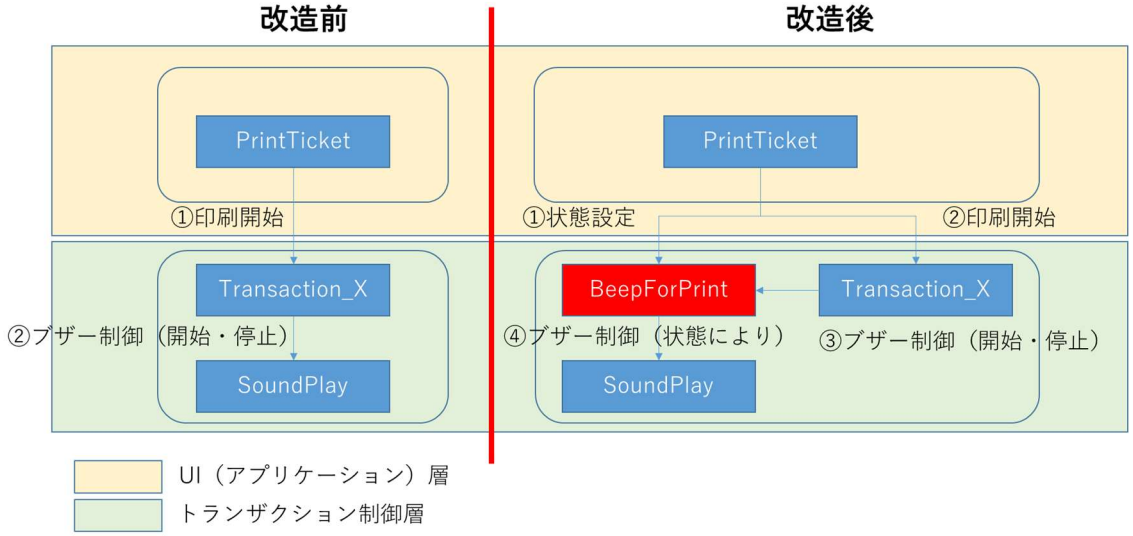


**図 26　改造前後のソフトウェア構造**

これにより，問題現象発生時（図 25）の 1 枚目のブザー停止制御から最終枚目のブザー開始制御まで，ブザーの開始・停止制御を行わないようにすることができ，ブザー鳴動を規則的に行わせることができる．

UI 層では印刷枚数に関する情報を持っている．ブザーの鳴動を印刷開始から全ての対象物の印刷終了までとするように UI 層から状態を設定する．トランザクション制御層では UI 層が設定した状態に応じてブザー鳴動制御を行う．Transaction_X では呼び出し先が変わるだけで，ブザー鳴動の ON/OFF 制御の決定には関与しない．BeepForPrint で管理する状態は，デフォルトでは Transaction_X のブザー鳴動指示に従う状態とするが，複数枚の印刷で UI 層が 1 枚ずつ印刷指示を行う場合のみ，UI 層が状態を変更する．図 27 に，この設計により問題現象がどのようにして解決するかを示す．

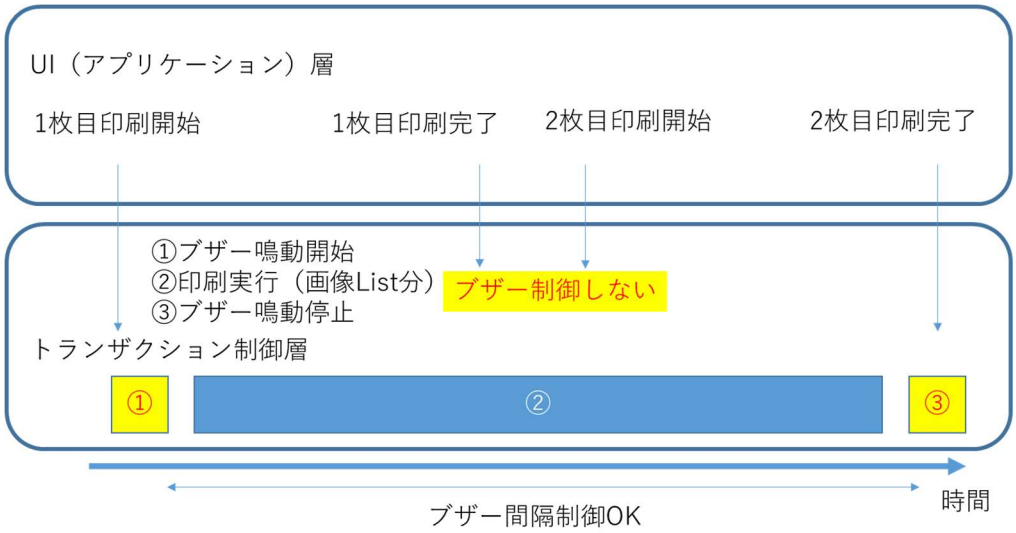


**図 27　問題現象の解決**

このようにすることで，予約受付端末で UI 層から画像リストを送る印刷指示にも影響を与えないため，改造したトランザクション制御層を製品共通で用いることができる．

## 4.5 保守の手順

本開発のように、設計情報がある場合の保守を以下に手順化する．

### 4.5.1　ソフトウェア改変の動機

本開発の例では，予約受付端末のトランザクション制御層を用いるとブザー鳴動のタイミングずれが起こるが，券売機用に適切に変更すれば，問題とはならない．しかし，ビジネス観点ではトランザクション制御層を全てのサービスで共通のものとすることで，効率よく開発を遂行したいという要求がある．そのため，トランザクション制御層を予約受付端末と券売機の両方で利用できるように改良し，将来的なサービスの多様性に備える必要があった．ここでのプログラムの改変の動機は，ISO14764 での完全化保守にあたる．予約受付端末の観点では，印刷時のブザー鳴動以外には変更の影響がないようにしたいため，極力設計構造の変更は行わないことにした．このように，改変の動機から要求仕様や修正方針を決定する．

### 4.5.2　変更箇所の特定

まずは，修正対象が設計上可変でない，不変のモジュール（層）に及ぶものかどうかを判断する．本開発事例の場合，各層の役割が明確なため，改変が必要なモジュールや，改変した場合の影響範囲が容易に特定できる．本開発事例ではブザー鳴動の開始，停止を行う箇所を修正する必要があり，その責務を担う箇所はトランザクション制御層である．このモジュールは，複数の製品で同じものを用いたいため，なるべく変更を行わない，不変のモジュールとして扱っている．

### 4.5.3　修正方針の決定

不変モジュールを変更する場合，可変モジュールの変更と比べて，従来動作していた機能に悪影響を及ぼす（デグレード）の可能性が高くなる．本開発では UI（アプリケーション）層は設計構造上，変更することを前提とした可変モジュールであり，この層を変更してもシステムの基本機能が成立するように設計されている．一方でトランザクション制御層は，変更を前提としておらず，変更した内容が層内の他のモジュールに影響する場合，修正対象機能以外に影響する恐れがある．

不変モジュールの変更時には，変更の影響範囲を極力小さくする方策を検討する．本開発事例では，UI（アプリケーション）層から利用する API 単位で，複数の API で実現する機能に修正の影響が極力及ばないよう，修正箇所の特定を行う．Transaction_X は，印刷機能を実現するモジュールであり，この部分の修正は，印刷機能のみに影響する．またアダプター（BeepForPrint）を用意することで，Transaction_X 内でも，従来の機能を実現するモジュールを大きく変えることなく，要求仕様を実現している．

このような手順で変化点を局所化することで，設計，実装，テストの規模を抑えて効率良く開発を遂行できる．

### 4.5.4　ISO14764 との本手順の関係

ISO 14764 でのソフトウェアの設計，開発では，ISO 12207 の開発プロセスに記載された新規開発の方法を参照している．実務上の手順としては，保守内容の要求仕様化，および設計方針を決定するため，ソフトウェア改変の動機を明確化し，要求仕様と設計方針が確定した後は，設計変化点を特定する．特定した設計変化点について変更設計を行い，それに基づき実装を行う．

## 4.6 おわりに

本章で紹介したソフトウェアは，初期設計時点から，極力変更しないモジュール群と，提供サービスによって積極的に変更を許容するモジュール群を分離しており，かつ Layers アーキテクチャでシンプルに実現しているため，変更箇所や影響範囲を容易に特定できた．このように，設計書で固定的に扱うモジュール群と，可変的に扱うモジュール群が区別され，その間のインタフェースが管理できていれば，改良保守時に修正箇所と修正による影響範囲を局所化でき，保守対応時間を短縮することができる．本章の例では，慎重に変更すべきトランザクション制御層に新たなモジュールを追加することになったが，アプリケーション層がこのモジュールを利用するか否かの判断を行う形としているため，初期設計の思想を壊していない．設計レビューでこの修正についても関係者で周知している．現在はこの修正の入ったソフトウェアが，各種サービス向けの製品開発のベースとなっており，市場にて既に 2 つの異なるサービスで稼働している．

# 5. まとめ

ソフトウェアの主に改良保守の具体的事例で，その課題と解決方法について議論してきた．1 章で見たように，組込みソフトウェア開発は保守プロジェクトが 90%以上を占めるが，保守のための手法や開発手順は十分に議論されていない．そこで保守の効率化を目指し，調査規模，修正規模を小さくできる方法を示し，成果を確認した．

2 章で見たように，保守によって解決したい課題は明確であっても，改変後の仕様の検討に注意が必要な場合がある．また仕様が明確化された後，その仕様を実現するためにどこを変更すべきか，設計情報が使えない場合と使える場合で，開発効率は大きく異なる．

設計情報が使えない 2 章の例のような場合には，ソースコードから修正箇所を特定する必要がある．修正方針の検討に先立ち，修正規模をあらかじめ調査することで，修正箇所を局所化できるようにして，対応にかかるコストの低減を図った．しかしその際，grep のように文字列一致での検索によって修正箇所を特定するには，時刻を扱うライブラリや API の検索など，変数文字列とは別の観点での検討も併用する必要がある．

一方で，3 章のようにプログラムスライシング等の技法に加えて，アプリケーション仕様の観点で絞り込みを使うことで，一部，割り込みハンドラ処理などの例外を除き，ソースコードから対象箇所を一定の作業手順により、特定することができる．この作業は，プログラムスライシングを扱えるツールを利用することによって効率化できるが，今後はスライスから時刻の設定に関係しないもの（目的としないもの）を自動的に取り除くことで，更なる効率化を目指す予定である．

また 4 章に記載した例では，設計情報が修正箇所の特定に有効な例を示した．Layers アーキテクチャを利用して，製品群に共通のソフトウェアモジュールと製品毎に異なるソフトウェアモジュールを分離することで，変更の必要なモジュールを特定しやすく工夫している．更に，各モジュールで担う機能の情報から，修正すべき正確な箇所を効率よく見つけることができる．このような構造にすることにより，ハードウェアとソフトウェアの共通部分を流用し，ソフトウェアの可変部分を変更することで，決済機能を持ち，チケット等を発行できる製品成果物を効率よく作成していく予定である．

近年のソフトウェア保守は，動作を当初仕様に適合させる，単なる不具合の修正のみを対象としていない．ソフトウェアを継続的に利用する中で，変化する運用環境への適合や，ユーザ要求機能の追加，当初計画しなかった他システムとの接続，連携などにも対処する必要がある．このため保守では，変更要求の早期反映とシステムの安定稼働という，相反する要望を同時に実現することが求められる場合もある．

このように複雑な保守の活動に対応すべく，DevOps など，新規開発と運用を同時に検討する考え方も生まれた[5-1]．継続的構築，デリバリーの実現やコミュニケーションを強化するツール類を用いて，保守に関わる組織やプロセスを改善しようとする取組みと考えることができる．DevOps の考え方は，新規開発と保守，運用の間に存在する組織やプロセスの違いを解消し，上手く連携させて開発活動をサポートするものである．

一方で組織間のコミュニケーションやプロセスの変革ではなく，ソフトウェアの設計開発技術から保守の課題を考えると，ISO14764 の内容から大きな進歩はなく，現在でも新規開発で用いられる手法に依存している．最初から DevOps で提唱されるような体制を取らずに開発された多くのソフトウェアでは，現在でも開発現場において，案件毎に最適な方策を模索しながら解決している状況である．

例えば「2025 年の崖」[5-2]に備えるデジタルトランスフォーメーション（DX）への動きの中で，サイロ化，ブラックボックス化した旧来のソフトウェアを置き換えたり，発展させたりする困難さも，そうしたソフトウェアを当初計画外の形で修正したり発展させるための，有効なソリューションが現時点では無いためと考えられる．

既存のソフトウェアを部分的，概要的に表現した設計情報を使って，品質を確保したまま効率よく発展させる手法の研究は，こうしたニーズに応えるものである．本論文で示した内容は，十分に一般化はされていないものの，上記課題の解決に必要な設計開発技術の，現実的で有効な使用方法の一例となっている．今後は，稼働しているソフトウェアの品質を確保したまま効率よく拡張，発展させるための設計開発技術の一般化に向けて，研究を進めて行きたい．

# 参考文献


[1-1] 組込みソフトウェア開発データ白書 2019, 独立行政法人 情報処理推進機構 社会基盤センター(2019), p.37.
[1-2] 【改訂版】組込みソフトウェア向け開発プロセスガイド, 独立行政法人 情報処理推進機構 ソフトウェア・エンジニアリング・センター(2007), pp.224-225.
[1-3] 大森隆行, 丸山勝久, 林晋平, 沢田篤史, ソフトウェア進化研究の分類と動向, コンピュータソフトウェア, Vol.29, No.3, pp. Aug. 2012.
[1-4] E.Burton Swanson, “The Dimensions of Maintenance”, in IEEE 2nd International Conference on Software Engineering Proc. (San Francisco), pp.492-497, October 13-15(1976).
[1-5] Murphy-Hill, E. and Black, A. P., Breaking the Barriers to Successful Refactoring, Observations and Tools for Extract Method, in Proc. Int'l Conf. Software Engineering, ICSE'08, 2008, pp.421-430.
[1-6] Henkel, J. and Diwan, A. CatchUp!, Capturing and Replaying Refactorings to Support API Evolution, in Proc. Int'l Conf. Software Engineering, ICSE'05, 2005, pp.274-283.
[1-7] 下村隆夫, “プログラムスライシング技術と応用”, 共立出版 (1995).
[1-8] M. Weiser, “Program Slicing, “ in IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp.352-357, July 1984, doi: 10.1109/TSE.1982.5010248.

[2-1] Eric Brown, Embedded Linux Keeps Growing Amid IoT Disruption, Says Study, Linux.com News (2015).
[2-2] Apple, 64-bit Transition on macOS, Apple Developers News and Update, https://developer.apple.com/news/?id=0411018a (2018).
[2-3] time(3)解説, FreeBSD11.1, 3-Subroutines (2003).
[2-4] G.Holzmann, Out of Bounds, IEEE Software, Vol32, No.6, pp.24-26 (2015).
[2-5] 鈴木孝知, 中村建助 : 「西暦 2038 年問題」でトラブル相次ぐ, 日経コンピュータ 2004-4-1, http://tech.nikkeibp.co.jp/it/members/NC/ITARTICLE/20040325/1/ (2004).
[2-6] 横田隆夫, 西暦 2000 年問題の意味と対応策, コンピュータソフトウェア, Vol.13, No.5, pp.412-419 (1996).
[2-7] 内閣コンピュータ西暦二千年問題対策室, コンピュータ西暦２０００年問題に関する報告書 (2000).
[2-8] Keita Suzuki, Takafumi Kubota, Kenji Kono, Detecting and analyzing year 2038 problem bugs in user-level applications, 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), pp65-6509 (2019).

[2-9] Ryo Okabe, Jun Yabuki, Masakatsu Toyama, Avoiding Year 2038 Problem on 32-bit Linux by Rewinding Time on Clock Synchronization, 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vol1, pp1019-1022 (2020).
[2-10] Microsoft, 時間管理, https://msdn.microsoft.com/ja-jp/library/w4ddyt9h.aspx.
[2-11] NetBSD Foundation, Announcing NetBSD 6.0, https://www.netbsd.org/releases/formal-6/NetBSD-6.0.html.
[2-12] Apple, NSDate - Foundation | Apple Developer Documentation, https://developer.apple.com/documentation/foundation/nsdate.
[2-13] Apple, 64-bit Requirement for Mac Apps, https://developer.apple.com/news/?id=06282017a.
[2-14] Apple, 64-bit Apps on iOS 11, https://developer.apple.com/news/?id=06282017b.
[2-15] S. Harshini, K.R. Kavyasri, Digital World Bug, Y2k38 an Integer Overflow Threat-Epoch, International Journal of Computer Sciences and Engineering, Vol.5(3), Mar 2017, E-ISSN: 2347-2693 (2017).

[3-1] time(3), FreeBSD 11.1-RELEASE Library Functions Manual (2017).
[3-2] 大江秀幸, 安藤友康, 松下誠, 井上克郎, “組込み機器開発における 2038 年問題への対応事例”, デジタルプラクティス, 10(3) , pp.588-602 (2019).
[3-3] CodeSonar, https://www.grammatech.com/codesonar-cc (2020 年 7 月 25 日閲覧).
[3-4] Steven J. Vaughan-Nichols, “「2038 年問題」への対応進む Linux”. https://japan.zdnet.com/article/35149495/ (2020 年 07 月 19 日閲覧).

[4-1] F.ブッシュマン, R.ムニエ, 他, “ソフトウェアアーキテクチャ ソフトウェア開発のためのパターン体系”, 株式会社近代科学社(2000).
[4-2] Robillard, M. P, Coelho, W, and Murphy, G. C., How effective developers investigate source code: An exploratory study (2004).

[5-1] Ingo M. Weber, Leonard J. Bass, Liming Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional (2015).
[5-2] 経済産業省 デジタルトランスフォーメーションに向けた研究会, DX レポート～IT システム「2025 年の崖」の克服と DX の本格的な展開～, https://www.meti.go.jp/shingikai/mono_info_service/digital_transformation/pdf/20180907_03.pdf (2018).