デジタルプラクティス　Vol.10 No.3(July 2019)

一般投稿論文

組込み機器開発における2038年問題への対応事例

大江 秀幸[1※]　安藤 友康[1]　松下 誠[2]　井上 克郎[2]

[1]パーソルAVCテクノロジー（株）　[2]大阪大学
※2018年10月1日より関西デジタルソフト（株）に所属

UNIX系OSが組込み機器にも数多く用いられている．現在の組込み機器は32ビットのシステムも多く，32ビット版のUNIX系OSが用いられる場合も多い．32ビットのUNIX系OSでは，時刻情報を32ビット符号付き整数型で管理しており，2038年に時刻情報のオーバーフローが起きることが分かっている．この問題への具体的な対策事例の報告はあまりされていない．本稿では，2038年以降も動作保証が必要な組込み機器において，2038年を超えて機器が利用できるよう，UNIX時刻の起点を変更することで問題を回避し，製品化に成功した事例を報告する．

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[1]．これらのOSは，ターゲットシステムの進化に伴い32ビットから64ビットに移行されることも多い[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32ビットシステムの継続利用を判断する場合も多い．

32ビット版のUNIX系OSでは，時刻情報を32ビット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[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[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．たと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[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りした．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1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稿と密接な関係を持つ問題であるが，本稿で対象とする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．

本稿では，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．

2. 開発対象機器

図1に本稿で議論する開発対象機器の構成を示す．

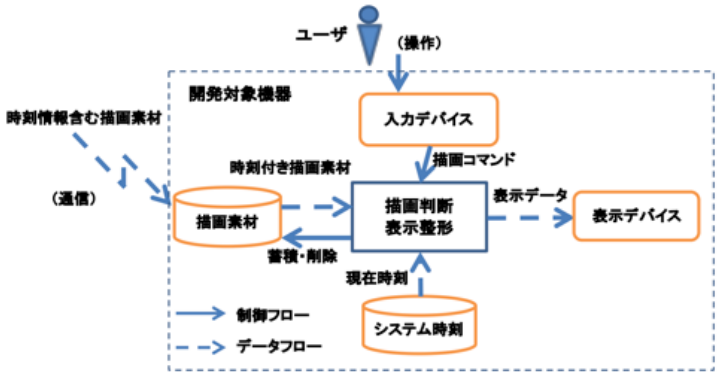

図1　開発対象機器の構成

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とも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分かっている．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2**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．

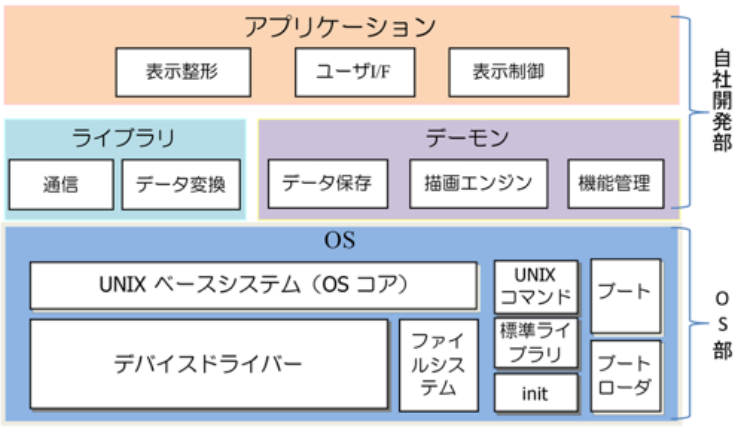

図2　アーキテクチャの概要

3. 開発背景と問題点

3.1　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32ビット 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．

3.2　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）[3]．この時刻情報の管理のため，32ビットのFreeBSDでは32ビット符号付き整数（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[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32ビット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個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個所では，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1**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，本稿では実行数を用いる．

表1　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

4. 本問題に対する対処

4.1 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2**に示す対策案を挙げ，検討した．

表2　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．たと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，量産後さまざま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2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り扱いを変更する案（（c），（d））の2案となった．これらの案の実現方法はさらに，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32ビットのままtime_t型の定義を64ビット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ともに大きくなることが予想された．案（a）の別案として，OSの64ビット化も検討されたが，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32ビットにしたまま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ないが，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とも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とも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個所を調査した．その結果を**表3**に示す．

表3　調査が必要なデータを使用する個所

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3,000カ所を超える部分で，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．加えてたとえば，time_t型のデータを32ビットの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個所すべて

で，このような使い方をする個所がな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．

4.2　修正作業の概要

4.2.1　修正仕様の検討

第4.1節に示した通り，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，曜日に影響を与えずにさら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3**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．

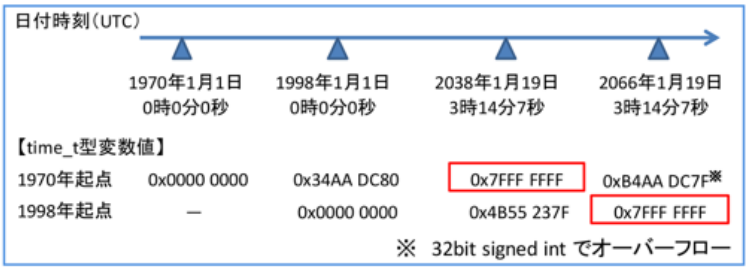

図3　起点変更とtime_t型変数値

第4.1節で示したように，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個所

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個所で，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4**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．

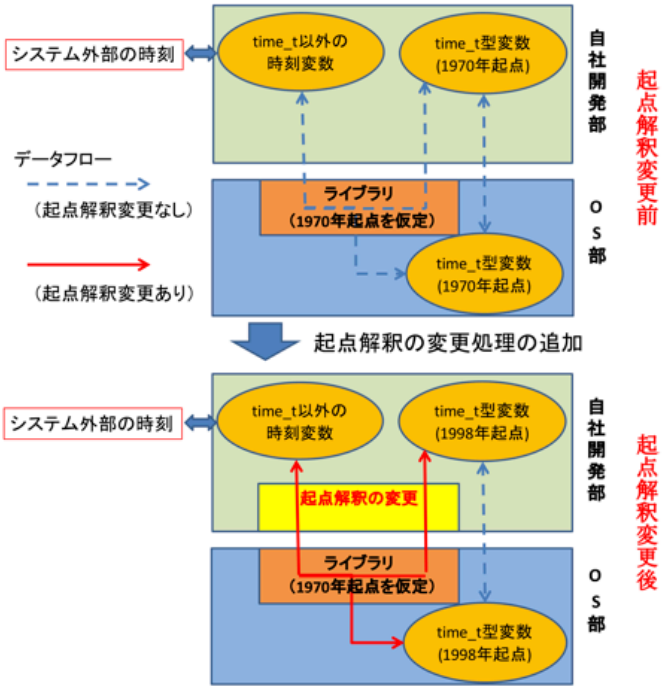

図4　時刻の扱い方の変更

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（0x34AA DC80）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個所

図4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値を扱う個所では，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よいためである．たとえば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よい．

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．

4.2.2　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2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個所の調査は，OS部については必要はなく，自社開発部での945カ所の調査のみでよい．

ただし，修正個所の特定には，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4**に示す型のデータが調査対象となる．

表4　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個所はないと判断した．

4.2.3　関数の調査

時刻を扱う関数を使用するすべての個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．調査対象の関数のうち，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5**に調査必要なライブラリ関数の例を示す．

表5　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 *)	○
・ ・ ・	・ ・ ・	・ ・ ・
60	void tzsetwall(void)	−

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．

調査結果からすべ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5のラッパー関数作成列に○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．

4.2.4　修正設計概要

図5，および**図6**に，修正前後の設計概要を示す．

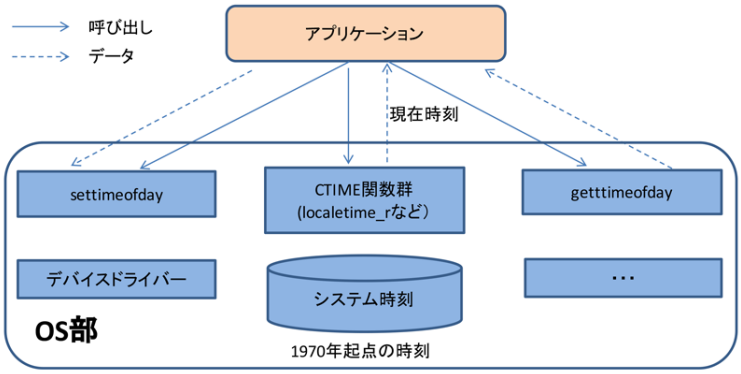

図5　修正前の設計概要

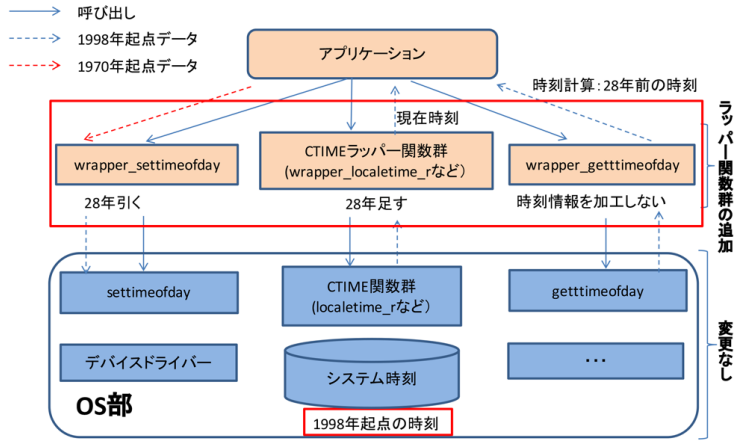

図6　修正後の設計概要

図5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32ビットのデータから，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6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32ビット値で返す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．

4.2.5　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7　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8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8　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9**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9　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．

5. 修正作業

5.1　テスト結果

表6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6　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，関数ごと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合せととも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カ所の不具合項目は，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．他に不具合項目はなく，当初の案通りの設計，実装が計画通りに行えた．

5.2　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および修正規模（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7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個所数，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．

表7　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．

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．

また表7中には，呼出し個所数が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7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，

3,200 / 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．

6. 議論

6.1 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（たとえば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，本稿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．たとえば，多くのOSにおいては，第4.1節における対策案（a），すなわち，経過時間を64ビット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32ビットであったため同様な問題が存在していたが，現在は64ビットで表現されている[9]．また，NetBSDも対応する全アーキテクチャでtime_tを64ビットで表現するよう修正が行われた[13]．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[10]，現在は64ビット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[11][12]．このように，経過時間を64ビットで表現する方法は，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，32ビット OSと64ビット OSが同時に保守されており，引き続き32ビット OSも利用し続けることができるが，本稿で述べたような2038年問題が生じる．64ビット 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64ビット OSへの移行が困難な組込みシステムには，本稿での知見が役立つものと思われる．本稿では，UNIXの2038年問題を対象として議論したが，第6.2節，第6.3節では，関連した時間の取り扱いの問題についても述べる．

6.2 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ちょうど中間の時刻である．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[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[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[5]．本稿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．

6.3 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桁で表現し，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[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[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，医療などの各分野ごと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[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恐れがあり，本稿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．

6.4　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個所や影響範囲の確認を十分に行ったため，大きな混乱がな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．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個所がある．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．また第4.1節で検討した案（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．

7. 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，64ビットのシステムでは現実的には検討の必要がない[8]．現在稼働している32ビットのシステムを64ビット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，64ビット化を予定していない32ビット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．

本稿で報告した方法は，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．

謝辞　本プロジェクトに関するデータ提供等をいただいた，パーソルAVCテクノロジー（株）開発本部 第2技術部の開発者諸氏に深く感謝する．

参考文献

1）Brown, E. : Embedded Linux Keeps Growing Amid IoT Disruption, Says Study, Linux.com News (2015).
2） Apple : 64-bit Transition on macOS, Apple Developers News and Update, https://developer.apple.com/news/?id=0411018a (April 11, 2018)
3）time（3）解説, FreeBSD11.1, 3-Subroutines (2003).
4）Holzmann, G. : Out of Bounds, IEEE Software, Vol32, No.6, pp24-26 (2015).
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)
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).
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).
8）Harshini, S. and Kavyasri, K. R. : 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).
9）Microsoft, 時間管理, https://msdn.microsoft.com/ja-jp/library/w4ddyt9h.aspx
10）Apple, NSDate - Foundation | Apple Developer Documentation, https://developer.apple.com/documentation/foundation/nsdate
11）Apple, 64-bit Requirement for Mac Apps, https://developer.apple.com/news/?id=06282017a
12）Apple, 64-bit Apps on iOS 11, https://developer.apple.com/news/?id=06282017b
13）NetBSD Foundation, Announcing NetBSD 6.0, https://www.netbsd.org/releases/formal-6/NetBSD-6.0.html

大江 秀幸（正会員）hideyuki.oe@digitalsoft.co.jp
1991年関西大学工学部金属工学科卒業．同年よりNECテレコムシステム（現NEC通信システム）株式会社，2002年より松下AVCマルチメディアソフト（現パーソルAVCテクノロジー）株式会社にて主に組み込みソフトウェア開発に従事．2018年10月より関西デジタルソフト株式会社に所属．技術士（情報工学部門）．

安藤 友康（非会員）ando.tomoyasu@kk.jp.panasonic.com
2007年神戸高専 電気工学科卒業．同年パナソニックAVCマルチメディアソフト（現パーソルAVCテクノロジー）株式会社に入社．以来，組み込みソフトウェア開発に従事．

松下 誠（正会員）matusita@ist.osaka-u.ac.jp

1998年大阪大学大学院基礎工学研究科博士後期課程修了．同年同大学基礎工学部情報工学科助手．2002年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助手．2005年同専攻助教授．2007年同専攻准教授．博士（工学）．リポジトリマイニング，プログラム解析の研究に従事．情報処理学会，日本ソフトウェア科学会，ACM各会員．

井上 克郎（正会員）inoue@ist.osaka-u.ac.jp

1984年大阪大学大学院基礎工学研究 科博士後期課程修了（工学博士）．同年大阪大学基礎工学部情報工学科助手．1984年～1986年，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991年大阪大学基礎工学部助教授．1995年同学部教授．2002年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．

投稿受付：2018年6月1日
採録決定：2019年2月6日
編集担当：高田広章（名古屋大学）