A Practical Appraoch to the Year 2038 Problem for 32-bit Embedded Systems

Hideyuki Oe
Osaka University
hideyuki.ooe@gmail.com

Makoto Matsushita
Osaka University
matusita@ist.osaka-u.ac.jp

Katsuro Inoue
Osaka University
inoue@ist.osaka-u.ac.jp

***Abstract*— Many UNIX-based operating systems are also used for embedded systems. Many current embedded systems use 32-bit system architecture, and 32-bit versions of UNIX-based operating systems are often used. A 32-bit UNIX-based OS manages time information as a 32-bit signed integer, and it is known that an overflow of time information occurs in 2038. There are not many reports of practical approach for this problem. In this paper, we report our experience of successful modification of a 32-bit FreeBSD embedded system by changing the starting point of the UNIX time (epoch time). As a result, the system is guaranteed to be used beyond 2038 and it is now in the market.**

Keywords—The Year 2038 problem, embedded system, 32-bit system architecture, epoch time

I. INTRODUCTION

With the advancement of embedded systems, UNIX-based operating systems such as Linux and FreeBSD are generally used as the operating system [1]. These OSes are often migrated from 32-bit to 64-bit as the target system evolves [2].

However, in the development of embedded systems, major concerns are to shorten the development period and reduce costs, and new systems may be developed using the software of the old systems. In the development of embedded systems for mass production, slight differences in cost per unit have a major impact on the business. Therefore, if there is no reason to lead to user benefits, it is often decided to continue to use a low-cost 32-bit system when developing a new system.

The 32-bit version of the UNIX-based OS manages time information as 32-bit signed data on January 1st, 1970 at 0 o'clock 0 minutes 0 seconds (UTC) (called *epoch time*) [3]. This data causes digit overflow in 2038, about 68 years later [4] (hereinafter referred to as the Year 2038 problem). Systems that handle time information need to avoid various impacts caused by this problem.

Necessity to cope with the 2038 problem also depends on the operation guarantee period of the target system. For example, if the operation guarantee period is 20 years, system released in 2018 needs to be prepared for this problem.

A similar problem, failure of systems running UNIX happened on January 10, 2004 [5]. In this problem, two kinds of elapsed seconds were added without considering overflow, or the maximum number of digits of the elapsed seconds was incorrectly set. As a result, the billing program caused erroneous billing and the credit software did not work properly. Another related issue is the Year 2000 problem[6]. When dealing with the last two digits of the year in decimal, an overflow occurs once every 100 years. In this state, when the year 2000 is reached, since the year information is “00” inside the system, it is indistinguishable from the year 1900, and there is a risk of causing a calculation error. Both problems are closely related to this paper, including how to deal with them. However, there have been no concrete reports on software modification methods and their results for the 2038 problems as presented in this paper.

In this paper, we report the policy of our group's response to the Year 2038 problem and the concrete correction method. This finding can be applied to the repair of the Year 2038 problem of other similar systems, and to the time problems of related operating systems.

II. REQUIREMENTS OF DEVELOPMENT TARGET

Figure 1 shows the configuration of the development target system discussed in this paper.

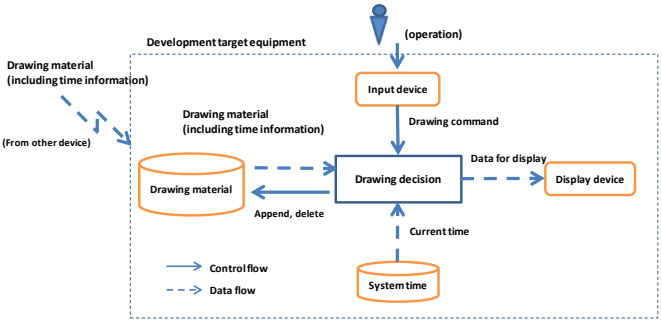


Fig. 1. Development target system.

The development target system (hereinafter referred to as the system) receives the data to be drawn by communication along with the expiration date of the data, and saves it as a drawing material with a time. It is known that the time information received by communication is notified of the correct time even after 2038. In the system, the data to be drawn is selected according to the internally managed system time and the user's operation of the system. Delete expired materials. After formatting the selected display target material, the information is displayed on the display device.

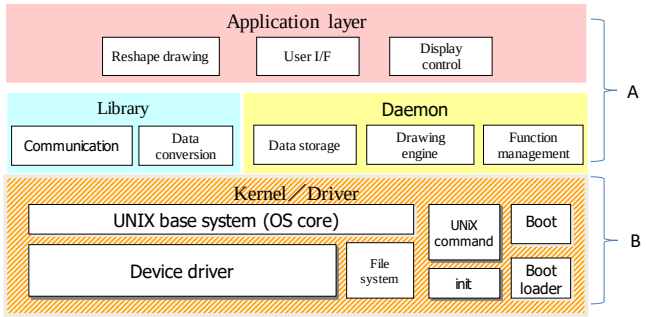


Fig. 2. Architecture overview

Figure 2 shows an overview of the software architecture of the system. Here, the in-house developed part refers to the

group of modules originally developed to realize the function of the relevant system, and the OS part refers to the group of modules exposed by OSS such as UNIX OS core and device drivers, standard libraries.

III. Development Background and Problems

A. Development Background

The system has been developed for over 10 years, repeating function expansion and model change for a fixed period. The installed OS is FreeBSD, a UNIX-based OS. At the time of initial model development of the relevant system, a 32-bit OS was mainstream, and extension development was continued until the time of model development (2017) without making major changes to the OS or architecture. This contributes to shortening of development period and cost control. The life of this product was 20 years, and even the latest sales model had to cope with the Year 2038 problem.

B. Problems and Issues

In FreeBSD, time is managed by UNIX time. The UNIX time is the number of seconds elapsed from 00:00:00 January 1st, 1970 UTC (hereinafter referred to as the 1970 origin) [3]. To manage this time information, 32-bit FreeBSD uses 32-bit signed integers (signed int). The maximum value is 0x7FFF FFFF (2,147,483,647), and when it exceeds 03:14:07 (UTC) on January 19th, 2038, about 68 years after the starting point [4]. Due to the overflow, the following problems were expected to occur on the relevant system.

(1) Information to be displayed cannot be displayed: since the drawing material with a time that does not contain the 2038 problem is received and displayed by the relevant system, if it conflicts with the time managed in the relevant system, the material cannot be displayed.

(2) The magnitude of the time is reversed because it becomes a negative value: since the most significant bit of the 32-bit signed integer is set by the overflow, it is misjudged at the location where the new / old of information is judged by the magnitude relationship.

(3) Anomalous processing works because it is a negative value: since information that does not expect a negative value by relevant system, anomalous processing is performed in places where anomalous processing is implemented for negative values.

It is necessary to identify the process that causes these problems from the source code and correct it without omission. In addition, the response to the Year 2038 problem is not the main function of the product, and there is a financial demand to minimize the cost of modification and development (collectively referred to as the development costs). The requirements are summarized below.

(a) Make corrections so as not to cause problems in product operation even after 2038.

(b) The product lifetime should be 20 years from its release.

(c) In order to reduce development costs, adopt means that can reduce the amount of development costs and the number of test steps.

(d) To avoiding confusion in the maintenance phase after development, modification of the OS kernel, drivers and other parts of the OS is not changed as much as possible (including data structures provided by the OS and API).

The volume of the development of the whole system is large, and the scope of influence varies greatly depending on the design policy. In order to reduce development costs, we would like to limit the scope of correction as much as possible. Table I shows the development volume of the entire relevant system. In this table, the total lines is the number of source code lines including comments, and the actual lines is the number of source code lines excluding comments. From here on, this paper uses the actual lines for counting source codes lines.

TABLE I. Overview of Development Volume

	Total lines (million lines)	Actual lines (million lines)
In-house development part	2.5	1.3
OS part	0.8	0.5
Total	3.3	1.8

IV. Dealing with This Problem

The type of variable that manages time information in FreeBSD is type "time_t." In order to solve the Year 2038 problem, it is necessary to correct around time_t-type variables. The developers of the system concerned conducted a study meeting and listed and considered various approaches as shown in Table II.

TABLE II. Considered Approaches

Approaches	Development volume	OS part modification	Impact range
(a) Modify time_t to 64-bit.	Huge	Needed	Huge
(b) Modify time_t to unsigned int (32-bit).	Huge	Needed	Huge
(c) Do not modify time_t, and calculate time correctly when time digits overflow occurs.	Huge	Needed	Huge
(d) Instead of modify time_t, change the "epoch" to appropriate value using wrapper functions, when we use the time related APIs.	Medium	Unnecessary	Medium

When considering the approaches, it is necessary to pay attentions to the restrictions specific to embedded systems. In addition to reducing the development cost per unit, attentions must also be paid to the maintenance cost. For example, if it is necessary to modify the software for some reason, it is generally expensive to change the software to embedded systems that operate in various usage environments after mass production. For this reason, maintenance costs become an important concern.

The proposals in Table II are roughly divided into two: proposals for changing time_t ((a), (b)) and proposals for changing the handling of values managed by time_t ((c), (d)). It became the methods for realizing these proposals are further divided into those that assume a change to the OS part ((a), (b), (c)) and those that assume a modification of the in-house development part ((d)).

Plan (a) is to change the definition of time_t 64-bit while the system architecture is still 32-bit. Since the bit width of the data type of the OS definition is changed, it is necessary to change and check the entire software. For this reason, it is expected that the amount of development and the scope of the impact of correction will both increase. As an alternative to the proposal (a), 64-bit OS was also considered, but it could not be selected because the development man-hours would increase compared to the proposal (a).

Plan (b) is to change from signed int to unsigned int while making the time_t 32-bit. Unlike the plan (a), there is no change in bit width, but since the data type of the OS definition is changed, the change / confirmation work is required for the entire software. For this reason, it is expected that the amount of development and the scope of the impact of correction will both increase.

Plan (c) is to treat time_t-type variables as 2038 or later if a carry occurs when comparing or reading values. It is necessary to change and check the entire software that handles time_t-type variables. For this reason, it is expected that the amount of development and the scope of the impact of the correction will both increase.

Plan (d) is the plan adopted in this project. The type of time_t does not change, and by shifting the starting point (epoch) from 1970; it is a plan to delay the overflow timing from 2038 by the amount shifted. With a wrapper function that changes the starting point and calling the API provided by the OS part and using it as needed, the plan could be implemented without changing the OS part at all. Both the amount of development and the scope of influence are smaller than the proposal to change the OS part.

In addition, we investigated locations where time_t was explicitly used to grasp the revision volume of each proposal. The results are shown in Table III.

TABLE III. WHERE WE NEED TO INVESTIGATE USING THE TIME VALUE

Where used	Number of places
OS part (including device drivers, OSS included, standard libraries)	2546
In-house development part	945

In plans (a) and (b) that change the time_t itself, it is necessary to investigate how data is used in more than 3,000 locations including the OS part. In addition, for example, if data of time_t is cast to 32-bit signed int data and any time calculation is performed, extending the time_t alone will not solve the Year 2038 problem. It is necessary to make sure that there is no place to use such data. Furthermore, the OS part may be changed for maintenance outside the company, and if the OS part is modified in-house, it is necessary to allow for the cost to follow the OS maintenance.

It was judged that it is difficult to take corrective action and guarantee the operation in a limited period of time, as the impact of the change on the proposal requiring changes in the OS part is widespread to the whole source codes. We chose plan (d) that does not presuppose a change in the OS part, in consideration of the maintenance cost.

V. THE CORRECTION WORK

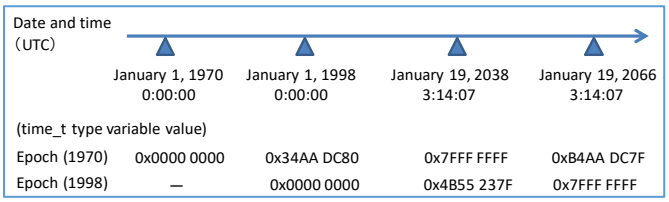
A. *Examination of Correction Specifications*

As shown in Section IV, the modified design adopted a plan to delay for a certain period from the start of 1970. Here, it is necessary to decide the delay period. The system has a product life of 20 years. By setting the delay period to more than 20 years, overflow will occur after 2058. Therefore, it is possible to cope with system operation from any time from 2018 to 2038.

Also, the delay period should be a multiple of 4, considering the leap year. If the delay period is 28 years, it is known that the calendar matches up to the day of the week until 2099, and no correction is necessary even when obtaining the information of the day from the time. If multiples of 28 years are used, the overflow period can be further delayed without affecting the day of the week. 28 years after 1970, which is the current starting point, is 1998, but after 28 years in 1998 is 2026. If 2026 is the starting point, it could not be expressed of year 2020.

From the above, the delay period was determined to be 28 years, and it was decided to use January 1st, 1998 00:00:00 (UTC) as the starting point (referred to as the 1998 starting point). Figure 3 shows the difference between the time_t-type variable values for the 1970 and 1998 start points.

Fig. 3. time_t variable value when change the meaning of the epoch



As shown in Section IV, this plan does not change the OS part, so in order to change the epoch time, it is necessary to pay attention to the use of OS provided libraries that interpret the value of time_t-type variables as the start point of 1970. When converting values between other than the time_t outside the system and the time_t internal to the system using the OS provided libraries, there is no match to the time_t-type values that were originally changed by the in-house development part. This is avoided by changing the interpretation of the origin according to how to handle the value.

(1) Locations where the change of origin interpretation is necessary

At the point where the time information outside the system and the time_t value inside the system are converted using the libraries of the OS part, it is necessary to change the starting point interpretation. In the in-house development part before the correction, there was a process to convert time information of a character string input from the outside of the system into a time_t-type value. Since this process is diverted to obtain time information of numeric value from a character string (unsigned int), the converted time information is the time of the 1970 origin. From this value, if you use the OS-provided library function settimeofday to create a value of time_t-type inside the system, an incorrect value starting from 1970 is set. In order to make the correct time_t-type value as the 1998 start point, a transformation that treats the difference of the start point is required. Conversely, another transformation is required to obtain a time information such as the correct date from the time_t value starting from 1998.

Therefore, we decided to perform the transformation for changing the starting point interpretation shown in Figure 3. The conversion is performed as follows. To obtain the value of the system time starting from 1998 from the external time, subtract the 28-year time (0x34AADC80). On the contrary, in order to obtain the correct time value from the system time starting from 1998, the time for 28 years is added.

(2) Locations where no change of origin interpretation is required

As shown in the broken line arrow in "after modification" of Figure 4, it is not necessary to change the starting point interpretation where the time_t-type value is used only in the in-house development part or in the OS part, since the time is treated as the year 1998 in both cases. It is not necessary to change the starting point interpretation because in order to calculate the difference between the two times or to determine the time of day, it is sufficient to compare the same time information that share the same starting point. For example, the difference between May 1st, 2018 (UTC) and May 22nd is the same three weeks for both the 1970 and 1998 starting points, so you can compare the time_t-type variables as they are.

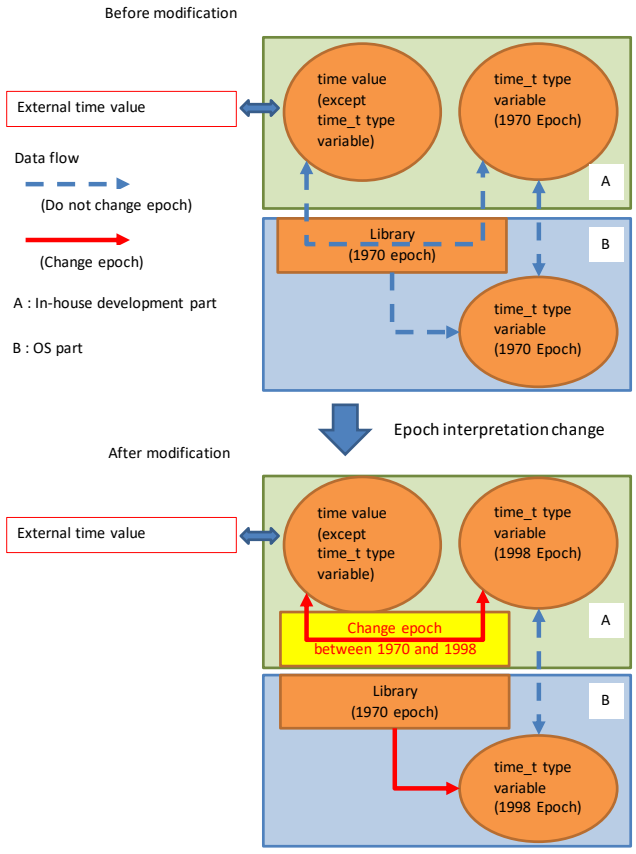


Fig. 4. Changing the meaning of the time_t variable value

In order to realize the above design proposal, it is necessary to investigate the influence of correction for each of data and functions that handle time.

B. Survey of Variables

In the case of the proposal (d) adopted this time, an overflow will not occur in 2038 because the starting point is changed. The problem does not occur in 2038, even if casting from time_t to int etc., because the definition type is not changed. If the in-house development part in Figure 2 controls the system time to be the value starting from 1998, it is possible to handle only the time that does not overflow, including the OS part. Therefore, it is not necessary to investigate the part using the time_t-type variable for the OS part, and it is sufficient to investigate only 945 parts in the in-house development part.

However, in addition to the variable that directly declares the time_t, it is also necessary to investigate the case where the structure members use the time_t-type in order to specify the location to be modified. The types of data shown in Table IV are to be surveyed.

TABLE IV. DATA TYPES THAT NEED TO BE INVESTIGATED

Number	Type	Abstraction
1	time_t	Data type for storing system time. It is defined in the standard C library, and is defined as 32-bit signed integer in 32-bit FreeBSD.
2	struct timeval	A structure that has a time_t-type variable (tv_sec) and a suseconds_t-type variable (tv_usec) as members. tv_sec stores seconds, and tv_usec stores microseconds.
3	struct tm	A structure that stores time information for each element such as year, month, day, day of the week as members, and stores each member as an int.

For these data in the in-house development part, we investigated to use data starting from 1998 so that time_t-type variables would not cause overflow. We also investigated what kind of data should be managed for struct timeval-type and struct tm-type variables. As a result of the investigation, it was judged that there was no need for correction other than the libraries.

C. Survey of Functions

The survey was conducted at every point where a function that handles time was used. In the function survey, the functions defined in each header file (time.h) were surveyed for the standard C library and the UNIX standard libraries (hereinafter simply referred to as the libraries). Among the functions to be surveyed, it is necessary to take corrective action for the function used by the relevant system. Since no modification of the OS part is made, it is decided to make corrections at each library caller. Table V shows an example of the libraries functions required for investigation.

We set data starting from the year 1998 so that time_t-type variables do not cause overflow for each function argument and return value and investigated what data should be managed with struct tm-type variables.

We decided to create a new wrapper function in a library that checks all correction target data and functions found by the survey and acquires and saves time data. The wrapper function adds or subtracts 28 years from the time_t-type value and then calls the libraries provided by the OS part. The in-house development part calls the wrapper function. As a result, we aimed to realize proposal (d) without changing the OS part at all and minimizing the correction range of the original system.

As a result of investigation, we decided to provide wrapper functions for the 12 libraries indicated by “Required” in the wrapper function creation column of Table V.

TABLE V. EXAMPLES OF LIBRARY FUNCTIONS THAT NEED TO BE INVESTIGATED

Number	Function name	Wrapper function creation
1	clock_t clock(void)	—
2	int clock_gettime(clockid_t, struct timespec *)	—
3	char *ctime(const time_t *)	Required
4	char *ctime_r(const time_t *, char *)	Required
5	double difftime(time_t, time_t)	—
6	int gettimeofday(struct timeval *, struct timezone *)	Required
7	struct tm *gmtime(const time_t *)	Required
8	struct tm *gmtime_r(const time_t *, struct tm *)	Required
9	struct tm *localtime(const time_t *, struct tm *)	Required
10	struct tm *localtime_r(const time_t *, struct tm *)	Required
11	time_t mktime(struct tm *)	Required
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 *)	Required
15	size_t strftime(char *, size_t, const char *, const struct tm *)	Required
16	time_t time(time_t *)	—
17	time_t timegm(struct tm *)	Required
18	time_t timelocal(struct tm *)	Required
·	·	·
·	·	·
60	void tzsetwall(voiod)	—

D. *Outline of Correction Design*

Figures 5 and 6 show the design outline before and after correction.

Figure 5 outlines the management of the system time before correction. System time is managed by data of time_t. Since overflow does not occur until 2038, element-specific time information such as number of month is extracted from the 32-bit data. The application uses the extracted value.

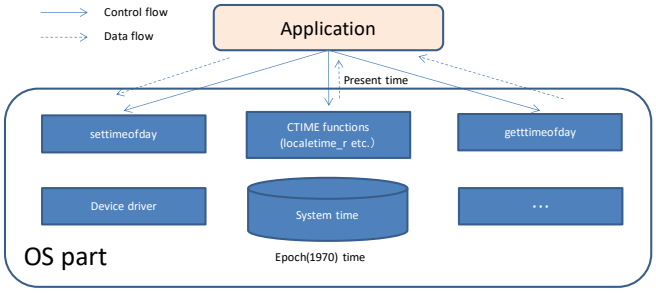


Fig. 5. Design overview before modification

Figure 6 shows the outline of the design after correction. In this correction, a wrapper function is prepared for the libraries that handles time information. Applications use the libraries via wrapper functions.

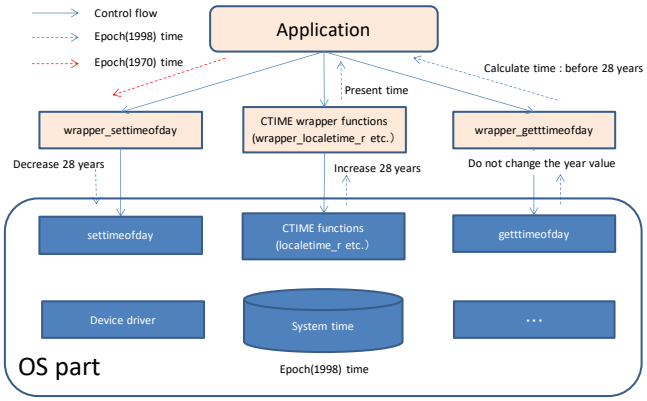


Fig. 6. Design overview after modification

The system time of the relevant system is managed with the time_t-type value starting from 1998, that is, the time_t-type value that adds 28 years from the interpretation of the 1970 starting point. This makes it possible to delay the occurrence of overflow until 2066. In the CTIME function group that returns the time information for each element such as year, month and day, and the time information as a character string, the wrapper function changes the interpretation to the 1998 start time correctly because the time is mainly used outside the relevant system (plus 28 years). In functions like gettimeofday that return time information as a 32-bit value of time_t, the wrapper function returns the value as it is. In addition, the time information received by communication is also stored after changing the interpretation to the time of the year 1998.

In summary, the system time of relevant system is set to year 1998 as the epoch. When using the libraries that calculates the time information for each element from the time_t-type value or the time_t-type value from the time information for each element, change the interpretation of the starting point appropriately using the wrapper function. From the above, the system time of relevant system can be managed at the time starting from 1998 remain the OS part unchanged.

E. *Implementation Example*

An implementation example is shown below. First, we define 28 years, which is the start change time, in years and seconds (Figure 7).

```
#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)
...
```

Fig. 7. Definition of the time value when change the epoch

```
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;
}
```

Fig. 8. The wrapper function of “settimeofday”

DATE_OFFSET_YEAR is 28 years of the start change time, and LEAP_DAYS shows the number of days of the leap day within the 28-year period. DATE_OFFSET_SEC calculated from these values is a value representing 28 years (including the last day) of the start change time in seconds, which is 883,612,800. A wrapper function is defined using these values.

Figure 8 shows the wrapper function for the settimeofday function. In this function, the system time is set using the libraries' settimeofday function, but at that time, it is set by subtracting 28-year time (seconds) in order to make it the year 1998. The first argument, with "wrapper_timeval" as type, is a type defined as like struct timeval, but its member tv_sec is defined as unsigned int instead of time_t.

The function that acquires element-specific time information of system time uses the acquired element for display etc. In order to display the correct time, it is necessary to add 28 years to elemental time information obtained by the libraries such as OS. Figure 9 shows a wrapper function for the localtime_r function.

```
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;
}
```

Fig. 9. The wrapper function of "localtime_r"

Note that the gettimeofday function, which is paired with the settimeofday function, takes an argument of the pointer of time_t and can obtain the current time, but the time_t-type variable stores the information from 1998 in the system. In this system, a wrapper function (wrapper_gettimeofday) was prepared to handle the information of the 1998 origin, but the value of the 1998 origin returned by the gettimeofday function is used as is, and the interpretation of the origin is not changed.

VI. The Work Results

A. Test Phase

Table VI shows the number of items of unit test and integrated test, and the number of defect items in the tests.

TABLE VI. THE NUMBER OF THE TEST ITEMS AND THOSE RESULT

	Test items	Number of defects
Unit test	165	0
Integrated test	103	1

In the unit test, path coverage tests and function output checks were performed based on the flowchart created for each function. In the integrated test, we prepared time data from 2038 or later to be input from the outside of the relevant system and confirmed the use cases where the function operates with the combination of the system status and the user operation.

The integration test detected one defect. The content of the defect was that the value "-1" returned by the error was incorrectly recognized to be the actual time information, and addition with another time data was calculated so that the desired operation was not performed. There were no other failure items, and the design and implementation as originally planned were performed as planned.

B. Actual Efforts

Here we show the volume and man-hours at the time of development. It should be noted that the response to the 2038 problem was a part of the requirements of the entire project including other enhancements, and it was not possible to obtain the man-hours for purely the 2038 problem. On the other hand, in the exchange of opinions after the project was completed, there was an impression from the development staff that there was no particular difficulty in dealing with this issue, and the work efficiency would be at the same level as the development work for other requirements. From this, it was judged that the number of man-hours could be calculated using the development efficiency (step / man-hours) in all project steps and the development and correction volume required for the 2038 problem (collectively called the correction volume). The work man-hours are calculated by the following equation.

$$Man\ hours = \frac{\text{Correction volume}}{\text{Development efficiency in all project processes}}$$

Table VII shows, for each library function, the volume of the created wrapper function, the number of places where the libraries' function is called, and the correction volume for changing the call to the wrapper function.

TABLE VII. THE VOLUME OF THE CORRECTION

#	Library function name	Created lines of wrapper function	Number of call locations	Modification lines of calling part
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	Common functions etc.※	38	-	-
Total		264	73	2936
Modification lines		264 + 2936 = 3200		

* Includes functions shared by multiple wrapper functions and "include" and "define" statements.

In addition, when determining the volume of correction, the company will also consider the affected code by the correction. Specifically, the range of source code that needs to be tested by modification is defined as the affected code. In this project, since the minimum unit of the unit test is a function, the calling function of the wrapper function is tested. Therefore, the correction volume is the number of lines of the whole function include calling the wrapper function.

Table VII also shows functions with zero calls. This is because multiple systems (system models) can be developed with the same software. The functions used in another model

development project are included in the functions in Table VII although not used in this project, and in such case the correction volume in this project is 0.

Next, the volume of development in this project including the 2038 problem is divided by the number of man-hours spent on development of that volume and calculate the development efficiency in all steps of the project. The development efficiency in this project was as follows.

Development efficiency: 3.45 steps / man-hours

From the above correction volume and the development efficiency, the man-hours required for the correction of the Year 2038 problem are as follows:

3,200 / 3.45 = 927.54 man-hours.

And if it is 8 hours a day, 20 days a month, it becomes:

927.54 / (8 * 20) = 5.80 man-months.

VII. Discussions

A. Handling of Date and Time in Software

Generally, in computer systems, the OS manages the elapsed time from the start time based on periodic interrupt signals from timers or information obtained from the network. This elapsed time may be displayed as it is in the OS itself or other application programs as it is, or as it is converted to a human-readable string (for example, 11:34:56 am, September 24, 2018). If the time_t-type variable that stores the elapsed time has an enough word length, a long elapsed time can be represented. However, many operating systems and computer systems designed in the old days, have been developed and used with insufficient word length due to lack of long-term prospects and severe hardware and cost constraints. In this case, a workaround as discussed in this paper is necessary to cope with a long elapse time.

In recent years, such problems are becoming widely recognized, and there is a corresponding movement at the OS and programming language level. For example, in many OSs, plan (a) described in Section IV, an approach that expresses the elapsed time in 64-bit, is taken. In Microsoft Windows, elapsed time is represented by time_t, and in Visual C++ versions before Visual C++ 2005 and Microsoft C / C++, there was a similar problem because it was 32-bit length, but now expressed in 64-bit [9]. Also, NetBSD has been modified to represent time_t in 64 bits on all supported architectures [13]. On Apple's macOS and iOS, time is expressed as the number of seconds since January 1st, 2001 UTC [10], and it is currently defined by regulations to allow only 64-bit applications [11] [12]. Thus, expressing the elapsed time in 64-bit is a reliable way to solve the problem in practice, but it involves hardware and API changes, so it is difficult to apply it easily in an embedded system like the relevant system. In FreeBSD, a 32-bit OS and a 64-bit OS are maintained at the same time, and the 32-bit OS can be used continuously, but the 2038 problem described in this paper will occur. Migration to a 64-bit OS is difficult due to cost constraints and was not adopted in this project.

Thus, for embedded systems that are difficult to migrate to 64-bit OSs like this project, the findings in this paper will be useful. In this paper, we have discussed UNIX's Year 2038 problem, but in Sections VII.B and VII.C we will also discuss similar time handling issues.

B. The UNIX's Year 2004 Issues

The time 13:37:04, January 10th, 2004, (UTC) is a halfway between the year 1970 epoch and year 2038 rollover. After this time, cases have been reported where the Year 2038 Problem occurs [5]. The reported causes are as follows.

(a) The dates are added together without considering the overflow.

(b) In a system that recognizes time in units of 0.5 seconds, the maximum number of digits was not increased.

The year 2004 is in the middle of the year 1970 to 2038, and the time_t-type value is around 0x4000 0000. (a) is a phenomenon that exceeds 0x7FFF FFFF by adding two time_t-type values, and (b) is an example where an overflow occurs around 2004 because the count-up of the time_t-type value occurs at twice the normal speed is. Both are problems that occur with the same mechanism as the Year 2038 Problem, and in fact are examples that occurred before year 2038 as an obstacle.

Examples of failures caused by these are those in which the day of the week was incorrectly identified in the call charge billing system, the communication program was failed, and some ATMs could not be used properly [5].

There are also cases where the occurrence of the phenomenon has been prevented in advance, and the user has been successful in minimizing the problem by calling for upgrading to the countermeasure software [5]. Even in the case of embedded software targeted in this paper, we have recognized the problem and taken the action in advance, so that we could minimize the problems.

C. The Year 2000 Problem

The Year 2000 problem is to represent four-digit year information in the last two digits of the decimal number, and the system also manages internally year information in the two-digit decimal number. It is a problem that causes an overflow after year 2000 [6]. The following is an example of problems that can be caused by the Year 2000 problem.

(a) Date calculation error: period from 1996 to 2000: 0-96 = -96 years.

(b) Mistakes in date comparison: from 0 < 96, it is misjudged that 1996 is newer than 2000.

(c) Errors due to input / output: register data in year 2000 as data older than year 1901.

These problems can be avoided by extending the two digits to four digits, or by expressing the number with two or more decimal numbers with two digits [6].

The Year 2000 problem was pointed out in advance in terms of problems, risks, and scope, and was a major concern globally. Even in Japan, each company formulates a response plan, secures a budget, and responds by confirming the progress of the action in each field such as finance, energy, telecommunications, transportation, medical care, etc. Efforts were made to disclose information.

As a result of such prior approaches, no major confusion occurred. According to the information as of the morning of January 5, 2000, there were only 27 relatively minor problems related to the Year 2000 problem [7].

In the Year 2000 problem, there is a possibility that time information to be managed in the system may exist in multiple places other than the time_t-type value, and there is a possibility that it cannot be solved only by conversion of input / output data of the libraries reported in this paper.

D. Evaluation of Corrective Work

After the project was concluded, the development members held a review meeting on the project activities. At the meeting, we confirmed that there were no major confusions, as the correction points and the scope of impact were sufficiently confirmed in advance for the Year 2038 Problem.

Regarding our approach here, the developers were able to depict the work volume and the amount of work without any particular points of uncertainty, so we could finish the response according to our schedule. Since neither the OS nor the standard libraries have been modified, there is no need to take any special action on this issue when updating the OS or standard libraries in the future. The completed source code is incorporated into the released product and is operating normally.

The method selected this time could not express time before year 1998. Therefore, this method cannot be selected by systems that manage information before the changed start point. This method can be selected in the relevant system since the system handles only current and future information. In this case, when setting the system time, there is a part that handles the time starting from 1970 at the caller of the wrapper_settimeofday function. When setting the system time, when using the time starting from the year 1970, it is necessary to check thoroughly the possibility of overflow. Although similar problems will occur in 2066, it is possible to easily re-apply the method by changing the wrapper function used in this method. In addition, the proposals (a) to (c) examined in Section IV could not be selected in the development of the system because of the corresponding man-hours and maintenance costs.

The handling of time information and the input and output of time information are not specific to the system and are generally used in computer systems other than embedded systems. Also, the wrapper function using in the method could be implemented on another systems. Therefore the selected method is applicable to systems other than this one.

VIII. CONCLUSION

In this project, due to constraints of available man-hours and delivery date, we chose a method to set the starting point of the system time to 28 years delayed 1998 based on the plan (d) in Section IV. After clarifying the scope of the survey and its results, we conducted a steady survey and finally prepared the environment after 2038 in the test and confirmed that no problems occurred.

Date and time overflow problems do not have to be considered practically on 64-bit systems [8]. If you plan to convert a currently operating 32-bit system to 64-bit, it will not be a problem if you make sure that no overflow will occur by 2038 and migrate according to the plan. However, for a relatively inexpensive 32-bit system that is not scheduled to be 64-bit and a system that is operating as of year 2038, which has a long operation period, some approaches need to be taken.

Since the method reported in this paper can cope with this problem relatively inexpensive, it would be fine if it could be useful for examining our approach for the Year 2038 problem with system and for estimating the development period and cost on other systems. We want to apply it to the 2038 problem of other information system.

ACKNOWLEDGMENT

We deeply appreciate the developers of 2nd Engineering Department at Persol AVC Technology Co., Ltd. for providing data related to this project.

REFERENCES

[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] FreeBSD: time(3), FreeBSD 11.1-RELEASE manual, 2003.

[4] Holzmann, G.: Out of Bounds, IEEE Software, Vol32, No.6, pp24-26, 2015.

[5] Takatomo Suzuki, Kensuke Nakamura : Troubles in “year 2038 problem”, Nikkei Computer (2004-4-1), http://tech.nikkeibp.co.jp/it/members/NC/ITARTICLE/20040325/1/, 2004 (in Japanese).

[6] Takao Yokota : Meaning of the year 2000 problem and countermeasures, Computer Software, Vol.13, No.5, pp.412-419, 1996 (in Japanese).

[7] Cabinet Computer Year 2000 Problem Management Office, “Report on the year 2000 problem”, https://www.kantei.go.jp/jp/pc2000/houkokusyo/honbun.html, 2000 (in Japanese).

[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: Microsoft Docs: Time Management, https://docs.microsoft.com/en-us/cpp/c-runtime-library/time-management

[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