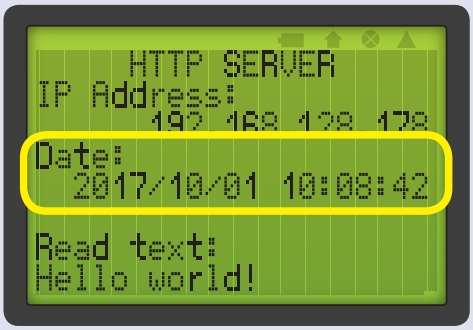
計測温度データの microSD への保存

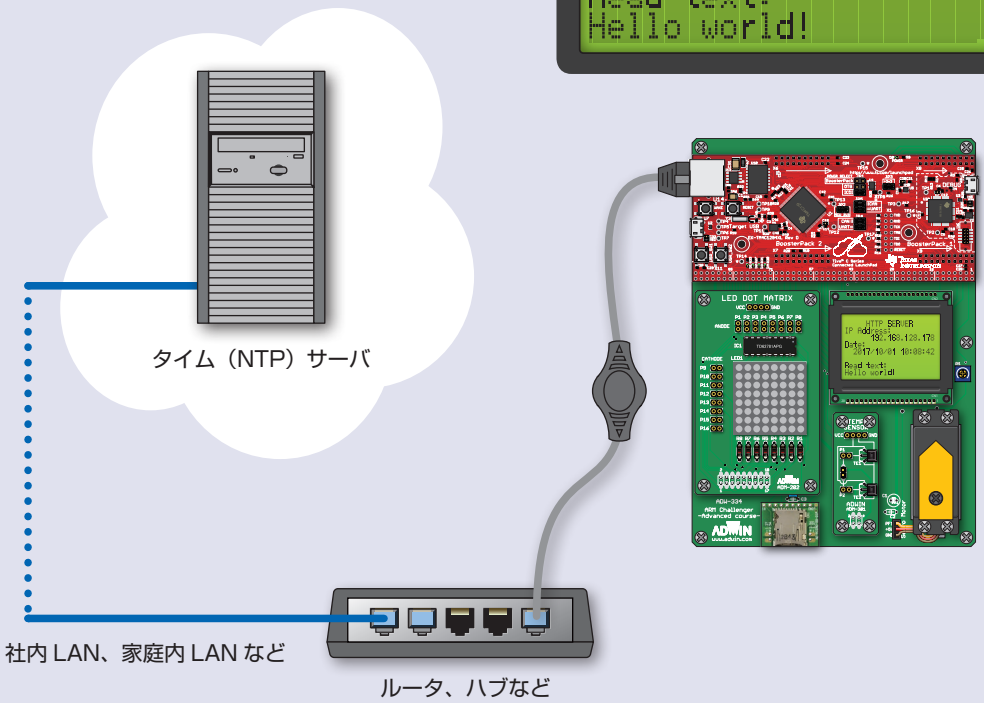
学習内容

STEP13で取得した温度データをmicroSDカードに記録する方法を学習します。
記録データには、温度を測定したときの正確な時刻が必要なので、まず、タイムサーバから時刻を取得する方法を学習します。
次に、microSD カードにデータを書き込む方法を学習します。

課題 14-1

タイムサーバから時刻を取得し、
GLCD に時刻を表示しましょう。





計測温度データの microSD への保存

SNTP モジュールの使用

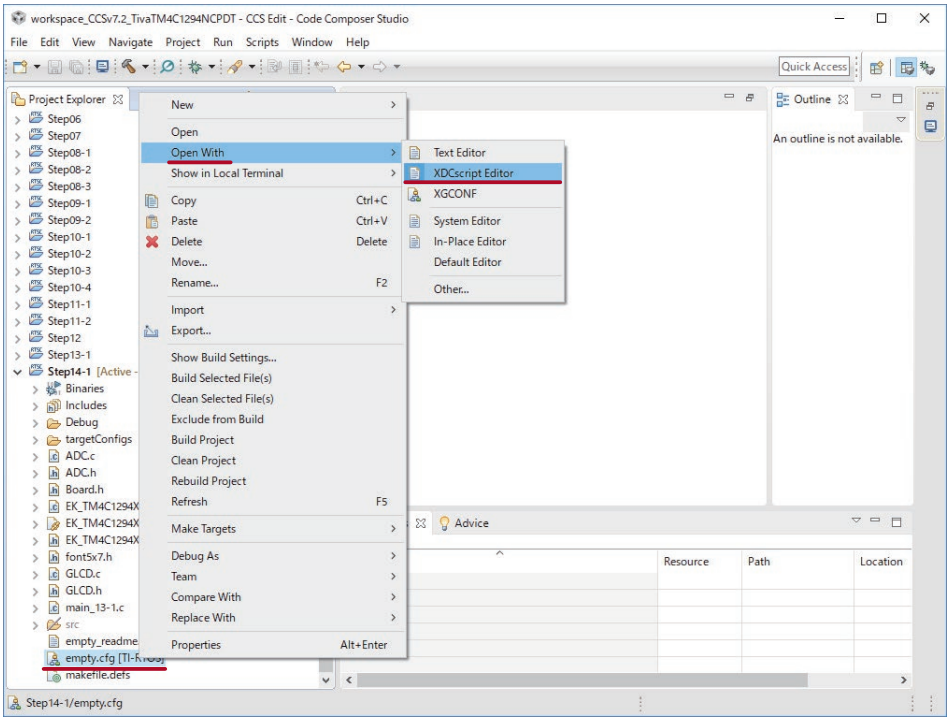
ここでは、TI-RTOS の SNTP モジュールを使って時刻合わせを行うことにします※。

なお、詳細については CCS の「Help」→「Help Contents」より「TI-RTOS for TivaC*」→「Documentation Links」→「Networking Documentation」→「TI-RTOS Network Services　User's Guide」→「SNTP Client」をご覧ください。

cfg ファイルの設定

SNTP モジュールを使うために、まずは cfg ファイルを編集します。

❶「Project Explorer」から「empty.cfg」を右クリックして「Open With」→「XDCscript Editor」で開き、cfg ファイルのソースコードを表示します。



❷ 以下の三行を末尾に追加します。

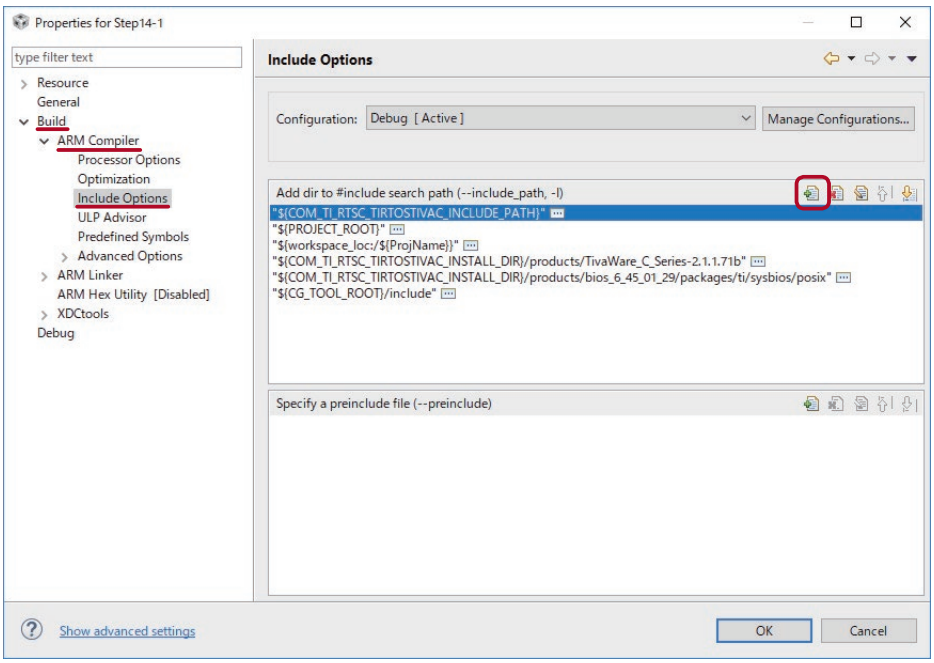
（注釈：三行目の「Global.autoOpenCloseFD = true;」は、GUI から「System Overview」→「TCP/IP (Wired)」→「Advanced」を開き、autoOpenClose を true にすることで行うこともできます）

```
var Sntp = xdc.useModule('ti.net.sntp.Sntp');
Sntp.networkStack = Sntp.NDK;
Global.autoOpenCloseFD = true;
```

※　SNTP クライアントは一般にソケットプログラミングでも作成可能ですが、本コースで用いる TI-RTOS の場合は、SNTP モジュールと干渉してしまいうまくいかないことがあるようです。

コンパイラの設定

❸「Project Explorer」でプロジェクト名を選択し、右クリックをして「Properties」を選択します。
❹「Build」→「ARM Compiler」→「Include Options」を開きます。



SNTP とは

SNTP は NTP（Network Time Protocol）と呼ばれる時刻合わせに利用されるプロトコルの簡易版です。S は Simple の S です。

NTP は、時刻を合わせたい NTP クライアントと正確な時刻を保持している NTP サーバとの間で通信するときに使用するプロトコルです。通信は UDP で、ポートの 123 番を使用します。SNTP を利用するとネットワークを通じて簡単に正確な時刻を取得することができます。

NTP サーバは、一般利用が可能なものがいくつも公開されています。以下は、国内の主な公開 NTP サーバです。

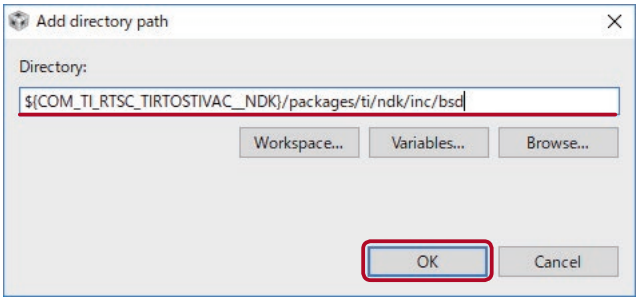
サーバ提供団体名	サーバ名	IP アドレス
NICT 情報通信研究機構	ntp.nict.jp	133.243.238. ～
NTP POOL PROJECT	jp.pool.ntp.org	各所様々

計測温度データの microSD への保存

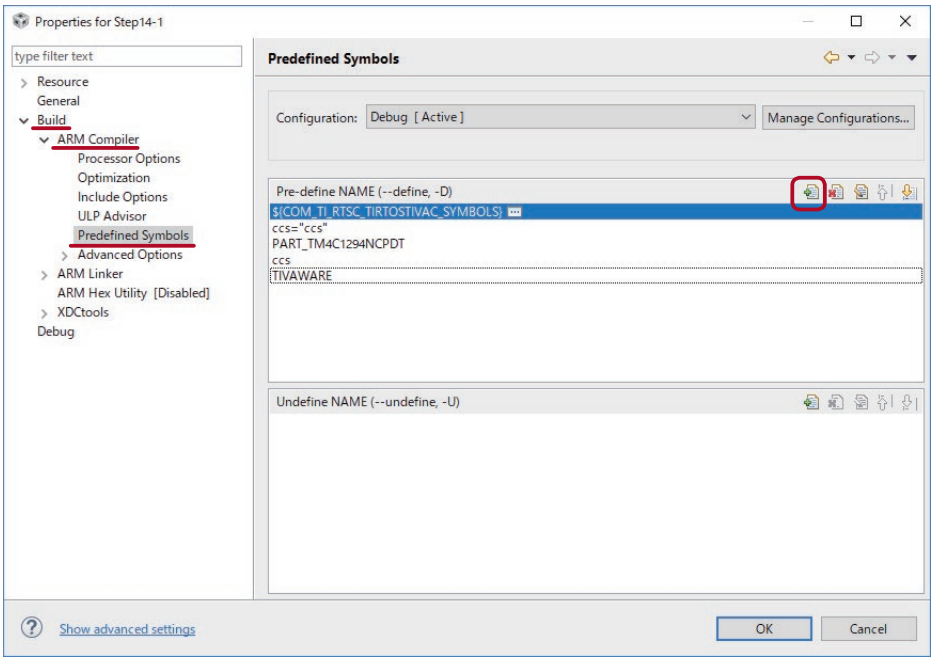
SNTP モジュールの使用

❺「Add dir to #include search path」の「Add...」アイコン をクリックし、「"${COM_TI_RTSC_TIRTOSTIVAC__NDK}/packages/ti/ndk/inc/bsd"」を追加します。

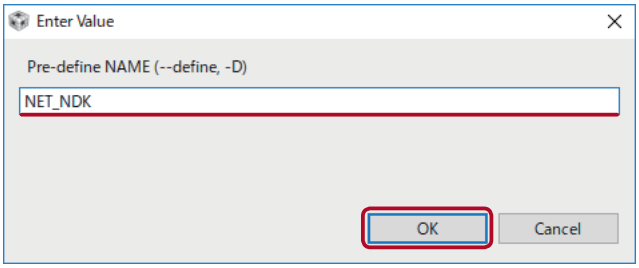
（注釈：「${COM_TI_RTSC_TIRTOSTIVAC__NDK}」は、「Variables...」ボタンから選択することもできます）



❻「Build」→「ARM Compiler」→「Predefined Symbols」を開きます。



❼「Pre-define Name」の「Add...」アイコン をクリックし、「NET_NDK」を追加します。



❽プロパティウインドウ右下の「OK」をクリックし画面を閉じます。

SNTP 用ファイルの作成

SNTP モジュールを使うには、ヘッダファイル「ti/net/sntp/sntp.h」をインクルードする必要がありますが、これは NDK のヘッダファイル「ti/ndk/inc/netmain.h」と干渉するので、SNTP 用のソースファール「SNTP.c」を別に用意することにします。

❾「Project Explorer」でプロジェクト名を選択し、右クリックをして「New」→「Source File」を選択します。

❿「New Source File」ウィンドウが開いたら、「Source file:」欄にファイル名（ここでは「SNTP.c」とします）を入力し、ウィンドウ右下の「Finish」をクリックします。

計測温度データの microSD への保存

SNTP モジュールのライブラリ関数

SNTP モジュールのライブラリ関数のうち、最低限使う必要があるのは以下のものです。

- int SNTP_start(get, set, timeUpdatedHook, servers, numservers, stacksize)

これは、SNTP クライアントを初期化して開始する関数で、この関数の実行時に NTP サーバからの時刻取得も行われます。最初の２つの引数は、それぞれ以下のような関数を指定します。

- uint32_t get(void)
- void set(uint32_t newtime)

このうち get 関数は、OS が保持している時刻を UNIX 時間※1 で返す関数を指定します。また、set 関数は OS の保持している時刻を正しい時刻に設定する関数を指定します。set 関数は、NTP サーバからの時刻取得が行われた時に呼び出され、引数 newtime には NTP サーバから取得した現在の UNIX 時間が渡されます。

SNTP_start 関数についての詳細は、ライブラリ関数の説明で触れます。

ホスト名からの IP アドレスの取得

NTP サーバの指定は、IP アドレスではなくホスト名で行うべきとされています※2。そのため、ホスト名から IP アドレスを取得する必要があります。

ホスト名の取得は、getaddrinfo 関数で行うことができます。getaddrinfo 関数の引数にホスト名を指定した場合は、対応する IP アドレスを含むそのホスト名に関する情報を addrinfo 型構造体で取得することができます。引数には IP アドレスを指定することもできます（その場合、得られる IP アドレスはもちろん引数の IP アドレスと同じものになります）。

getaddrinfo 関数についての詳細は、ライブラリ関数の説明で触れます。

※１　UTC（≒グリニッジ標準時）における 1970 年 1 月 1 日午前 0 時 0 分 0 秒からの経過秒数（ただし、うるう秒は無視）。
※２　IP アドレスで指定すると、IP アドレスが変わった際などにクライアント側で時刻情報が得られなくなるだけでなく、サーバ側にも支障を発生させてしまう恐れがあります。

計測温度データの microSD への保存

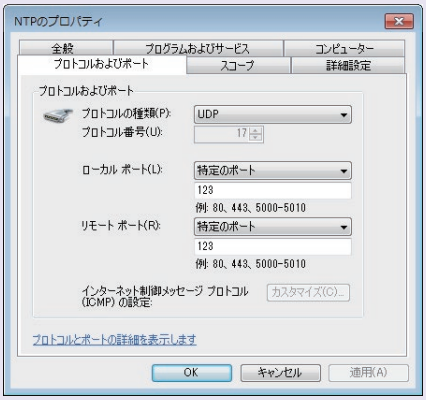
Windows パソコンを NTP サーバに設定する

外部 NTP サーバに接続できない場合、LAN 内に NTP サーバを構築する方法もあります。
以下の方法で Windows パソコンを NTP サーバに設定できます。

■ Windows 7 の場合：
コントロールパネル ＞ システムとセキュリティ ＞
Windows ファイアウォールを起動。

■ Windows 10 の場合：
設定 ＞ ネットワークとインターネット ＞
Windows ファイアウォールを起動。

以下共通
画面左側「詳細設定」をクリック。
「送信規則」をクリック ＞ 画面右側の「新しい規則」
をクリック。設定内容は右図と以下参照。



規則の種類	：ポート
プロトコルおよびポート	：UDP、特定のリモートポート 123
操作	：接続を許可する
プロファイル	：プライベート
名前	：NTP サーバ

計測温度データの microSD への保存

SNTP メッセージ

NTP サーバとの通信は SNTP メッセージと呼ばれるデータのやりとりで行われます。
SNTP メッセージのフォーマットは以下のようになっています。( ) 内の数字はビット数です。

8ビット			8ビット	8ビット	8ビット
LI(2)	VN(3)	MODE(3)	階層（8）	ポーリング間隔（8）	精度（8）
ルート遅延（32）					
ルート拡散（32）					
参照識別子（32）					
参照タイムスタンプ（64）					
開始タイムスタンプ（64）					
発信タイムスタンプ（64）					
受信タイムスタンプ（64）					
鍵識別子【任意】（64）					
メッセージダイジェスト【任意】（128）					

⚠ NTP ではポート 123 が使用されるため、NTP サーバと正常に通信できるように、ファイアウォールまたはルータ上でこのポートを開いておく必要があります。

計測温度データの microSD への保存

ここでは LI、VN、Mode、受信タイムスタンプのみ解説します。その他詳細仕様については、ネット上の公開文書「RFC1305」、「RFC2030」を参照してください。

LI 閏秒指示子 / 2bit

その日の最後の 1 分において、閏秒を挿入、削除することを示す。
00 : 警告なし (通常時はこれ)
01 : 最後の 1 分が 61 秒
10 : 最後の 1 分が 59 秒
11 : 警告

VN バージョン番号 / 3bit

SNTP および NTP バージョン番号を示す。
000 :
001 : バージョン 1
010 : バージョン 2
011 : バージョン 3 (通常の NTP サーバ)
100 : バージョン 4 (通常の SNTP サーバ)

Mode 動作モード / 3bit

動作モード示す。
000 : 予約
001 : 対称アクティブモード
010 : 対称パッシブモード
011 : クライアント
100 : サーバ
101 : ブロードキャスト
110 : NTP 制御メッセージのための予約
111 : 私的使用のための予約

受信タイムスタンプ / 64bit

サーバからクライアントに応答が発信された時刻を示す。64bit の符号無し固定小数点で表される。1900 年 1 月 1 日 0 時を基準に相対的な差を秒単位で表す。上位 32bit は整数部、下位 32bit は小数点以下を表す。

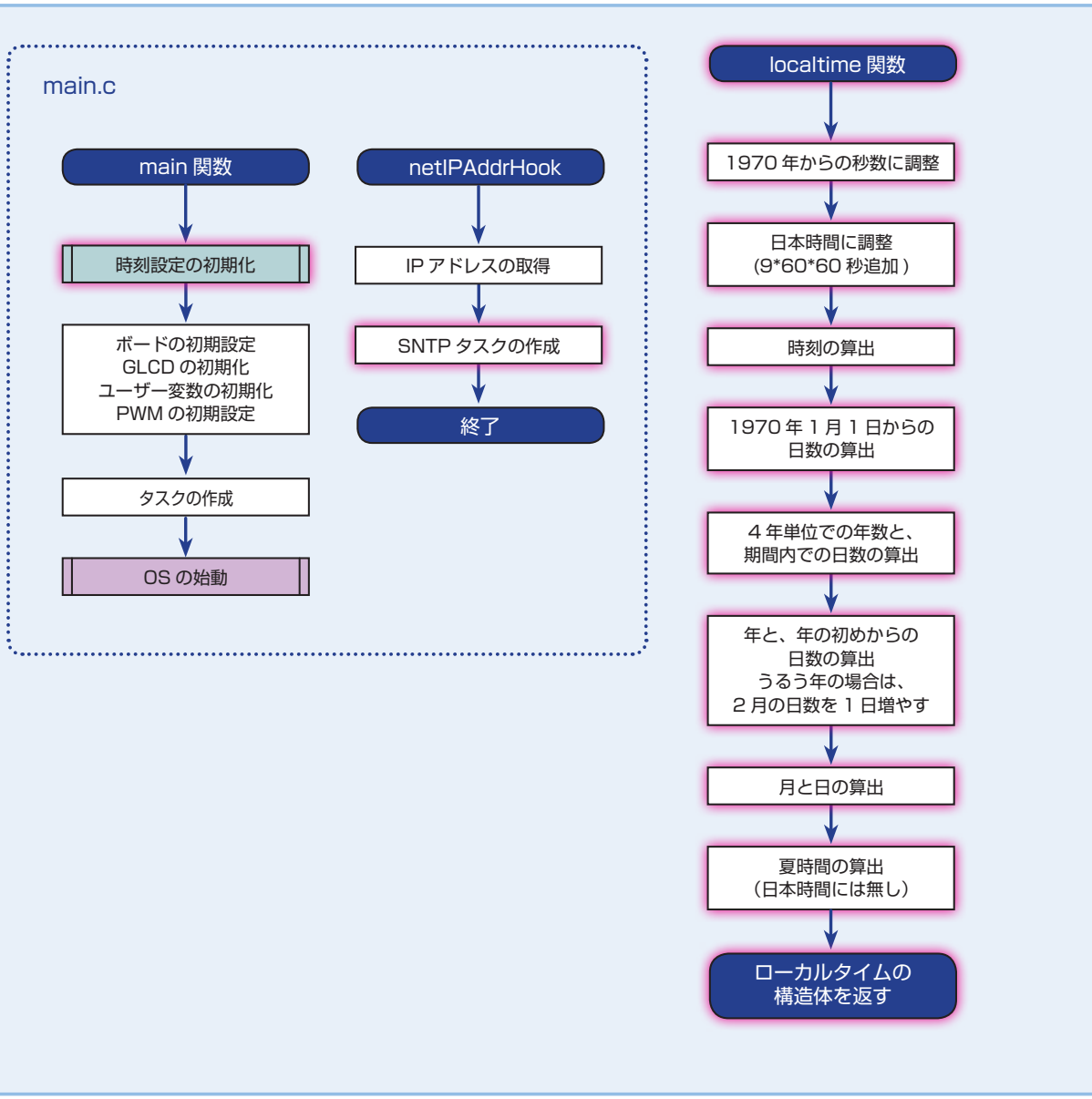
NTP サーバから時刻を取得するには、以下の手順で行います。

1. LI、VN、Mode を指定し、SNTP メッセージを NTP サーバに送信。
2. NTP サーバから SNPT メッセージを受信。
3. 受信タイムスタンプから日時を計算。

計測温度データの microSD への保存

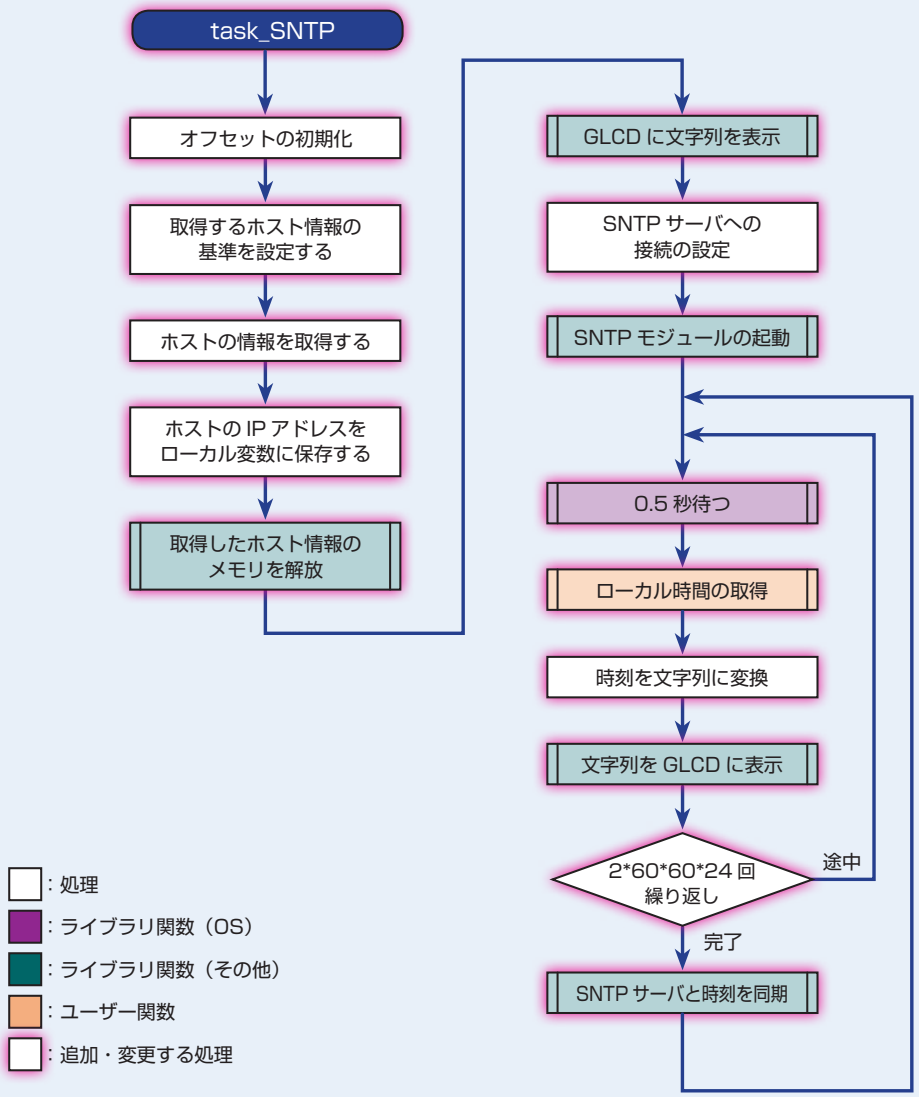
フローチャート 14-1

以下は、課題 14-1 を実現するためのフローチャート例です。localtime 関数は UNIX 時間を日本時間に変換するようにしています※。また、main 関数と netIPAddrHook 関数以外は SNTP.c ファイルに記述するものとします。その他の関数・タスクは「フローチャート 13-1」と同じです。



※ フローチャートの localtime 関数のアルゴリズムは、1970 年～ 2100 年 2 月 28 日まで有効です。
それ以降は、2100 年がうるう年ではない影響で日付がずれます。

計測温度データの microSD への保存



計測温度データの microSD への保存

型・構造体 14-1

課題 14-1 で使用する型・構造体を解説します。

なお、テキストカーソルをソースコード中の型や構造体の部分に合わせると、その型・構造体に関する情報がポップアップされます。さらに「F3」キーを押すと、その型・構造体が定義されているファイルを開くことができます。

ローカルタイム用構造体　struct tm

時刻をローカルタイムで表すための構造体で、インクルードファイル time.h で定義されています。以下のメンバ変数を持ちます。

- int tm_sec　：秒（0 ～ 59, 60, 61）※1
- int tm_min　：分（0 ～ 59）
- int tm_hour　：時（0 ～ 23）
- int tm_mday　：日（1 ～ 31）
- int tm_mon　：1 月からの月数（0 ～ 11）※2
- int tm_year　：1900 年からの年数
- int tm_wday　：曜日（0: 日、1: 月、2: 火、3: 水、4: 木、5: 金、6: 土）
- int tm_yday　：1 月 1 日からの日数（0 ～ 364, 365）※3
- int tm_isdst　：夏時間（0: 夏時間期間外、1: 夏時間実施中）※4

ネットワークアドレス用構造体　struct addrinfo

ネットワークのアドレス・ホストに関する情報を格納する構造体で、インクルードファイル netdb.h で定義されています※5。主なメンバ変数は以下のとおりです。

- int ai_family　：アドレスファミリーを指定する（IPv4 の場合は「AF_INET」を指定）
- int ai_socktype　：ソケットタイプを指定する（TCP 通信では「SOCK_STREAM」を指定し、UDP 通信では「SOCK_DGRAM」を指定する）
- struct sockaddr *ai_addr　：ソケットアドレスへのポインタ（getaddrinfo 関数が返す）

※1：　60, 61 はうるう秒用です。
※2：　1 月は「0」、2 月は「1」、12 月は「11」になります。
※3：　1 月 1 日は「0」、1 月 31 日は「30」、12 月 31 日はうるう年でない場合は「364」、うるう年の場合は「365」になります。
※4：　日本時間には夏時間は無いので、常に「0」になります。
※5：　インクルードファイル netdb.h は、ti/net/sntp/sntp.h をインクルードすると同時にインクルードされます。

計測温度データの microSD への保存

ライブラリ関数 14-1

課題 14-1 で使用する関数を解説します。
なお、テキストカーソルをソースコード中の関数の部分に合わせると、その関数に関する情報がポップアップされます。さらに「F3」キーを押すと、その関数が定義されているファイルを開くことができます。

C 言語標準

メモリ領域への指定値のセット　void *memset(void *buf, int ch, size_t n)

アドレス buf から n バイトを値 ch で埋める関数で、インクルードファイル string.h で定義されています。配列や構造体を初期化するのに便利です。

```
memset(&hints, 0, sizeof(hints));
```

引数の設定例

- &hints　：変数のポインタを指定
- 0：埋める値（初期化の場合、通常は 0 または 0xFF で埋める）
- sizeof(hints)　：埋めるサイズ（ここでは、第一引数で指定した変数 hints の大きさを指定）

IP アドレスの解決
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)

ホスト名 node の IP アドレスを、hints に基づいて取得し、res に返す関数で、インクルードファイル netdb.h で定義されています。ホスト名から IP アドレスを取得するのに使うことができます。戻り値は成功すると 0、失敗すると 0 以外の値です。

```
error = getaddrinfo(hostname, NULL, &hints, &res);
```

引数の設定例

- hostname：ホスト名の文字列を指定。
- NULL　：サービスを指定（ここでは不使用＝ NULL を指定）。
- &hints　：取得する IP アドレスに関する基準を指定した struct addrinfo 型変数へのポインタを指定（ここでは struct addrinfo 型の変数 hints へのポインタを指定している）。
- &res　：取得した IP アドレスに関する情報が収められた struct addrinfo 型の領域へのポインタを格納する変数を指定（ここでは struct addrinfo 型のポインタ res へのポインタを指定している）。

計測温度データの microSD への保存

ライブラリ関数 14-1

領域の解放　void freeaddrinfo(struct addrinfo *res)

getaddrinfo 関数が取得した IP アドレスに関する情報を収めるために確保した領域を解放する関数で、インクルードファイル netdb.h で定義されています。引数 res には、getaddrinfo 関数の第四引数に代入されたポインタを指定します。

```
freeaddrinfo(res);
```

引数の設定例

- res　：getaddrinfo 関数で取得した IP アドレスに関する情報の領域へのポインタを指定。

時刻の取得　time_t time(time_t *timer)

現在の時刻を取得する関数で、インクルードファイル time.h で定義されています。時刻は引数 timer に代入され、戻り値としても時刻を返します。引数に NULL を設定した場合は、戻り値にのみ時刻を返します。

時刻は、多くのシステムでは UNIX 時間と呼ばれる UTC（≒グリニッジ標準時）の 1970 年 1 月 1 日午前 0 時 0 分 0 秒からの経過秒（うるう秒は無視）ですが、TI-RTOS では 1900 年 1 月 1 日午前 0 時 0 分 0 秒からの経過秒を返します。

```
time_t timer = time(NULL);
```

ローカル時刻の取得　struct tm *localtime(const time_t *timer)

ポインタ timer の指すアドレスに格納されている時刻（秒）を、ローカル時刻の構造体（struct tm）に変換してそのポインタを返す関数です。引数で指定する時刻には、一般に time 関数で取得した時刻が用いられます。

ここでは、localtime 関数は日本時間を返すものを自分で実装する必要があります。以下は、localtime 関数実装後の関数使用例です。

```
time_t timer = time(NULL);
struct tm *t_st = localtime(&timer);
```

BIOS 関係

本課題で新たに使用する BIOS ライブラリ関数は Seconds_set() と Seconds_get() で、インクルードファイル ti/sysbios/hal/Seconds.h で定義されています。

これらの関数に関する詳細は、CCS の「Help」→「Help Contents」より「TI-RTOS for TivaC*」→「Documentation Links」→「Kernel Documentation」→「TI-RTOS Kernel User's Guide」を開き、「5.4 Seconds Module」をご覧ください。

UNIX 時間の設定　Void Seconds_set(UInt32 seconds)

引数 seconds で指定した UNIX 時間をデバイスの UNIX 時間として指定します。また、秒数カウントの初期化も行います。

UNIX 時間の取得　UInt32 Seconds_get(Void)

デバイスの UNIX 時刻を取得します。なお、この関数を利用する前に、少なくとも 1 回は Seconds_set() を実行しておく必要があります。

計測温度データの microSD への保存

ライブラリ関数 14-1

SNTP 関係

SNTP モジュールを使用するためのライブラリ関数です。

詳細については CCS の「Help」→「Help Contents」より「TI-RTOS for TivaC*」→「Documentation Links」→「Networking Documentation」→「TI-RTOS Network Services User's Guide」→「SNTP Client」をご覧ください。

SNTP クライアントの初期化と開始
int SNTP_start (uint32_t(*get)(void), void(*set)(uint32_t newtime), void(*timeUpdatedHook)(void *), struct sockaddr *servers, unsigned int numservers, size_t stacksize)

sockaddr で指定した NTP サーバに接続し、引数で指定した get 関数、set 関数で時刻合わせを行うような SNTP クライアントを開始します。

NTP サーバへのアクセスによる時刻の同期は、SNTP クライアント開始時（この関数の実行時）、および後述の SNTP_forceTimeSync 関数の実行時に行われます。

```
SNTP_start(Seconds_get, Seconds_set, NULL, (struct sockaddr*)&sntpAddr, 1, 0);
```

引数の設定例

- `Seconds_get`：現在の UNIX 時間を返す関数を指定。
- `Seconds_get`：OS の時刻を設定する関数を指定。uint32_t 型の引数ひとつを持ち、この関数が呼び出される際に NTP サーバから取得した UNIX 時間が渡される。
- `NULL`　：時刻更新時のフック関数を指定（ここでは NULL ＝フック関数を使わない）。
- `&sntpAddr`　：NTP サーバを sockaddr 構造体型変数の配列（ここでは変数へのポインタ）で指定。
- `1`　：指定したサーバ（ここでは 1 つ）の数を指定。
- `0`　：SNTP クライアントタスクのスタックサイズ。0 を指定した場合はデフォルトのサイズを使用。

NTP サーバからの時刻の取得　void SNTP_forceTimeSync (void)

NTP サーバから時刻を取得し、SNTP_start 関数で設定した set(newtime) 関数を呼び出すことでシステムの時刻を更新します。

なお、NTP サーバへの負荷を避けるため、NTP サーバからの時刻取得は必要最低限にしましょう。秒単位での精度が必ずしも要求されないのであれば、NTP サーバからの時刻取得は 1 日 1 回で十分です※。

また、NTP サーバからの時刻取得は SNTP_start 関数の実行時にも行われるので、SNTP_start 関数の実行直後にこの関数を実行する必要はありません。

```
SNTP_forceTimeSync();
```

※　国立研究開発法人情報通信研究機構（NICT）の公開 NTP サービスでは、アクセス回数は 1 時間平均で 20 回を超えないように要請されています。また、Windows XP では NTP サーバとの時刻合わせは標準で 1 週間に 1 回でした。

計測温度データの microSD への保存

コーディング SNTP.c 14-1

以下は、SNTP.c のコーディング例です

SNTP.c　main.c

```
/* XDCtools ヘッダファイル */
#include <xdc/std.h>
#include <xdc/runtime/System.h>

/* BIOS ヘッダファイル */
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/hal/Seconds.h>

/* C 言語ヘッダファイル */
#include <string.h>
#include <time.h>

/* ネットワークサービスヘッダファイル */
#include <ti/net/sntp/sntp.h>

/* GLCD ヘッダファイル */
#include "GLCD.h"

/* ユーザー定義マクロ */
#define CONSOLE(...) do { System_printf(__VA_ARGS__); System_flush(); } while(0)
#define SLEEP(X) Task_sleep((X)*1000/Clock_tickPeriod)

/*
 * Unix 時間をローカルタイムに変換
 */
struct tm *localtime(const time_t *_timer)
{
    static struct tm t_st;
    char mdays[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    time_t timer = *_timer;

#if defined(__ti__)

    // TI-RTOS の time 関数は 1900 年を基準にした秒数を返すので、
    // UNIX 時間（1970 年を基準にした秒数）に変換する。
    timer -= 2208988800;
#endif

    // 時差調整（UTC+09:00）
    timer += 9 * 60 * 60;

    // 時刻の算出
    t_st.tm_hour = (timer / (60 * 60)) % 24; //「時」の取得
    t_st.tm_min = (timer / 60) % 60; //「分」の取得
    t_st.tm_sec = timer % 60; //「秒」の取得

    // 1970 年 1 月 1 日からの日数の算出
    uint32_t days = timer / (24 * 60 * 60);

```

計測温度データの microSD への保存

コーディング SNTP.c 14-1

```
51      // 曜日の算出
52      t_st.tm_wday = (days + 4) % 7;
53
54      // 4 年単位での年数の算出
55      uint32_t y4 = days / (365 * 3 + 366);
56
57      // 4 年単位内での日数の算出
58      uint32_t y4days = days % (365 * 3 + 366);
59
60      // 年と、年の初めからの日数の算出
61      t_st.tm_year = 70 + 4 * y4;
62      if (y4days < 365)
63      {
64          t_st.tm_yday = y4days;
65      }
66      else if (y4days < 365 * 2)
67      {
68          t_st.tm_year += 1;
69          t_st.tm_yday = y4days - 365;
70      }
71      else if (y4days < 365 * 2 + 366)
72      {
73          t_st.tm_year += 2;
74          t_st.tm_yday = y4days - 365 * 2;
75
76          // 2 月を 1 日増やす（うるう年）
77          mdays[1]++;
78      }
79      else
80      {
81          t_st.tm_year += 3;
82          t_st.tm_yday = y4days - (365 * 2 + 366);
83      }
84
85      // 月と、月の初めからの日数の算出
86      t_st.tm_mday = t_st.tm_yday;
87
88      // 月の算出
89      t_st.tm_mon = 0;
90      while (t_st.tm_mon < 11)
91      {
92          if (t_st.tm_mday < mdays[t_st.tm_mon])
93              break;
94          t_st.tm_mday -= mdays[t_st.tm_mon];
95          t_st.tm_mon++;
96      }
97
98      // 月の初めからの日数を月の日付に換算
99      t_st.tm_mday++;
100
101     // 夏時間の算出
102     t_st.tm_isdst = 0; // 日本時間には無し
103     return &t_st;
104 }
105 /*
106  * SNTP タスク
```

```
107  */
108 Void task_SNTP(UArg arg0, UArg arg1)
109 {
110     int i; // 汎用カウンタ
111     struct sockaddr_in sntpAddr;
112     char *hostname = "ntp.nict.jp";
113     struct addrinfo hints, *res;
114     uint32_t s_addr;
115     int error;
116 
117     // 取得するホスト情報の基準の設定
118     memset(&hints, 0, sizeof(hints));
119     hints.ai_socktype = SOCK_DGRAM;
120     hints.ai_family = AF_INET;
121 
122     // ホストの情報を取得する
123     CONSOLE("SNTP host: %s\n", hostname);
124     error = getaddrinfo(hostname, NULL, &hints, &res);
125     if (error != 0)
126     {
127         CONSOLE("getaddrinfo: %d\n", error);
128         GLCD_str(3, 0, "Cannot get IP addr...");
129         return;
130     }
131 
132     // IP アドレスを変数 s_addr に代入
133     s_addr = ((struct sockaddr_in*) (res->ai_addr))->sin_addr.s_addr;
134     CONSOLE("ip address: %d.%d.%d.%d\n", s_addr & 0xff, (s_addr >> 8) & 0xff,
135             (s_addr >> 16) & 0xff, (s_addr >> 24) & 0xff);
136 
137     // ホスト情報のメモリを解放
138     freeaddrinfo(res);
139 
140     // GLCD に文字列を表示
141     GLCD_str(3, 0, "Date:");
142 
143     // SNTP サーバへの接続の設定
144     sntpAddr.sin_family = AF_INET;
145     sntpAddr.sin_addr.s_addr = s_addr;
146     sntpAddr.sin_port = htons(123);
147 
148     // SNTP モジュールを起動（時刻が同期される）
149     SNTP_start(Seconds_get, Seconds_set, NULL, (struct sockaddr*) &sntpAddr, 1, 0);
150     while (1)
151     {
152         // 0.5 秒おきに 1 日繰り返す
153         for (i = 0; i < 2 * 60 * 60 * 24; i++)
154         {
155             // 0.5 秒待つ
156             SLEEP(500);
157 
158             // ローカル時間の取得
159             time_t timer = time(NULL);
160             struct tm *t_st = localtime(&timer);
161 
162             // ローカル時刻を文字列に変換
```

計測温度データの microSD への保存

コーディング SNTP.c 14-1

```
163             char tstr[20];
164             tstr[0] = (t_st->tm_year + 1900) / 1000 + 0x30;
165             tstr[1] = ((t_st->tm_year + 1900) / 100) % 10 + 0x30;
166             tstr[2] = ((t_st->tm_year + 1900) / 10) % 10 + 0x30;
167             tstr[3] = (t_st->tm_year + 1900) % 10 + 0x30;
168             tstr[4] = '/';
169             tstr[5] = (t_st->tm_mon + 1) / 10 + 0x30;
170             tstr[6] = (t_st->tm_mon + 1) % 10 + 0x30;
171             tstr[7] = '/';
172             tstr[8] = (t_st->tm_mday) / 10 + 0x30;
173             tstr[9] = (t_st->tm_mday) % 10 + 0x30;
174             tstr[10] = ' ';
175             tstr[11] = (t_st->tm_hour) / 10 + 0x30;
176             tstr[12] = (t_st->tm_hour) % 10 + 0x30;
177             tstr[13] = ':';
178             tstr[14] = (t_st->tm_min) / 10 + 0x30;
179             tstr[15] = (t_st->tm_min) % 10 + 0x30;
180             tstr[16] = ':';
181             tstr[17] = (t_st->tm_sec) / 10 + 0x30;
182             tstr[18] = (t_st->tm_sec) % 10 + 0x30;
183             tstr[19] = '\0';
184 
185             // 文字列を GLCD に描写
186             GLCD_str(4, 2, tstr);
187         }
188 
189         // SNTP サーバと時刻を同期（1 日に 1 回）
190         SNTP_forceTimeSync();
191     }
192 }
193 
```

計測温度データの microSD への保存

コーディング main.c 14-1

以下は、main.c 13-1 のソースコードを元にしたコーディング例です。■は main.c 13-1 から追加した部分です。

SNTP.c　main.c

```
1  /* XDCtools ヘッダファイル */
2  #include <xdc/std.h>
3  #include <xdc/runtime/System.h>
4
5  /* BIOS ヘッダファイル */
6  #include <ti/sysbios/BIOS.h>
7  #include <ti/sysbios/knl/Task.h>
8  #include <ti/sysbios/knl/Clock.h>
9  #include <ti/sysbios/hal/Seconds.h>
```

この間の行に変更はありません

```
36 /*
37  * ユーザータスクの優先度
38  */
39 #define TASK_LED_PRIO 1
40 #define TASK_GLCD_PRIO 9
41 #define TASK_FATFS_PRIO 10
42 #define TASK_LDM_PRIO 11
43 #define TASK_SNTP_PRIO 6
44
45 /*
46  * ユーザースタックのサイズ
47  */
48 #define TASK_LED_STACK 512
49 #define TASK_GLCD_STACK 512
50 #define TASK_FATFS_STACK 1024
51 #define TASK_LDM_STACK 512
52 #define TASK_SNTP_STACK 2048
53
54 /*
55  * ユーザータスクの構造体
56  */
57 Task_Struct taskLEDStruct;
58 Task_Struct taskGLCDStruct;
59 Task_Struct taskFATFSStruct;
60 Task_Struct taskLDMStruct;
61 Task_Struct taskSNTPStruct;
62
63 /*
64  * ユーザータスクのスタック
65  */
66 Char taskLEDStack[TASK_LED_STACK];
67 Char taskGLCDStack[TASK_GLCD_STACK];
68 Char taskFATFSStack[TASK_FATFS_STACK];
69 Char taskLDMStack[TASK_LDM_STACK];
```

計測温度データの microSD への保存

コーディング main.c 14-1

```
 70 Char taskSNTPStack[TASK_SNTP_STACK];
```

この間の行に変更はありません

```
277 /*
278  * ユーザータスク
279  */
 .    .
 .    .
 .    .
533 /*
534  * SNTP タスク
535  */
536 Void task_SNTP(UArg arg0, UArg arg1);
537
538 /*
539  * フック関数
540  */
541
542 Void netIPAddrHook(IPN IPAddr, uint IfIdx, uint fAdd)
543 {
544     Task_Params taskParams;
545
546     /* IP アドレスの取得 */
547     IP = IPAddr;
548     CONSOLE("IP アドレス : %x\n", IP);
549
550     CONSOLE("SNTP タスクの作成 \n");
551     Task_Params_init(&taskParams);
552     taskParams.stackSize = TASK_SNTP_STACK;
553     taskParams.stack = &taskSNTPStack;
554     taskParams.priority = TASK_SNTP_PRIO;
555     Task_construct(&taskSNTPStruct, (Task_FuncPtr)task_SNTP, &taskParams, NULL);
556 }
```

この間の行に変更はありません

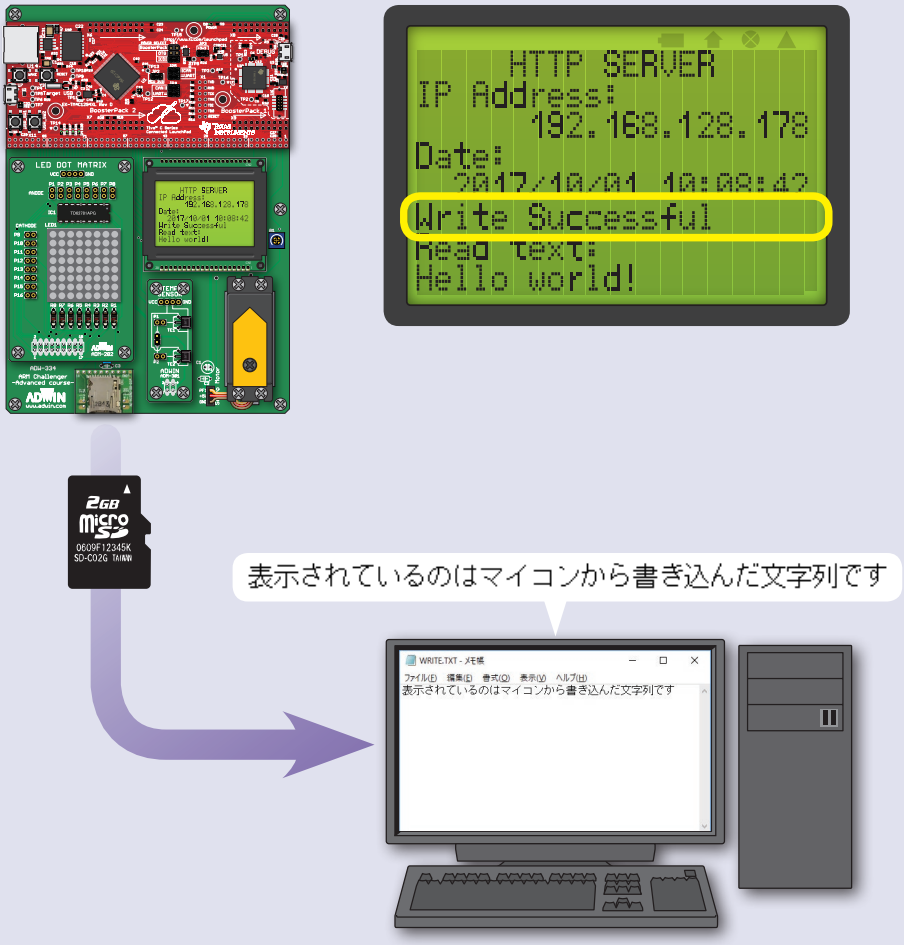
```
628 /*
629  * メイン
630  */
631
632 int main(void)
633 {
634     Task_Params taskParams;
635     PWM_Params pwmParams;
636
637     // 時刻設定の初期化
638     Seconds_set(0);
639
640     // ボードの初期設定
```

これ以降の行に変更はありません

計測温度データの microSD への保存

課題 14-2

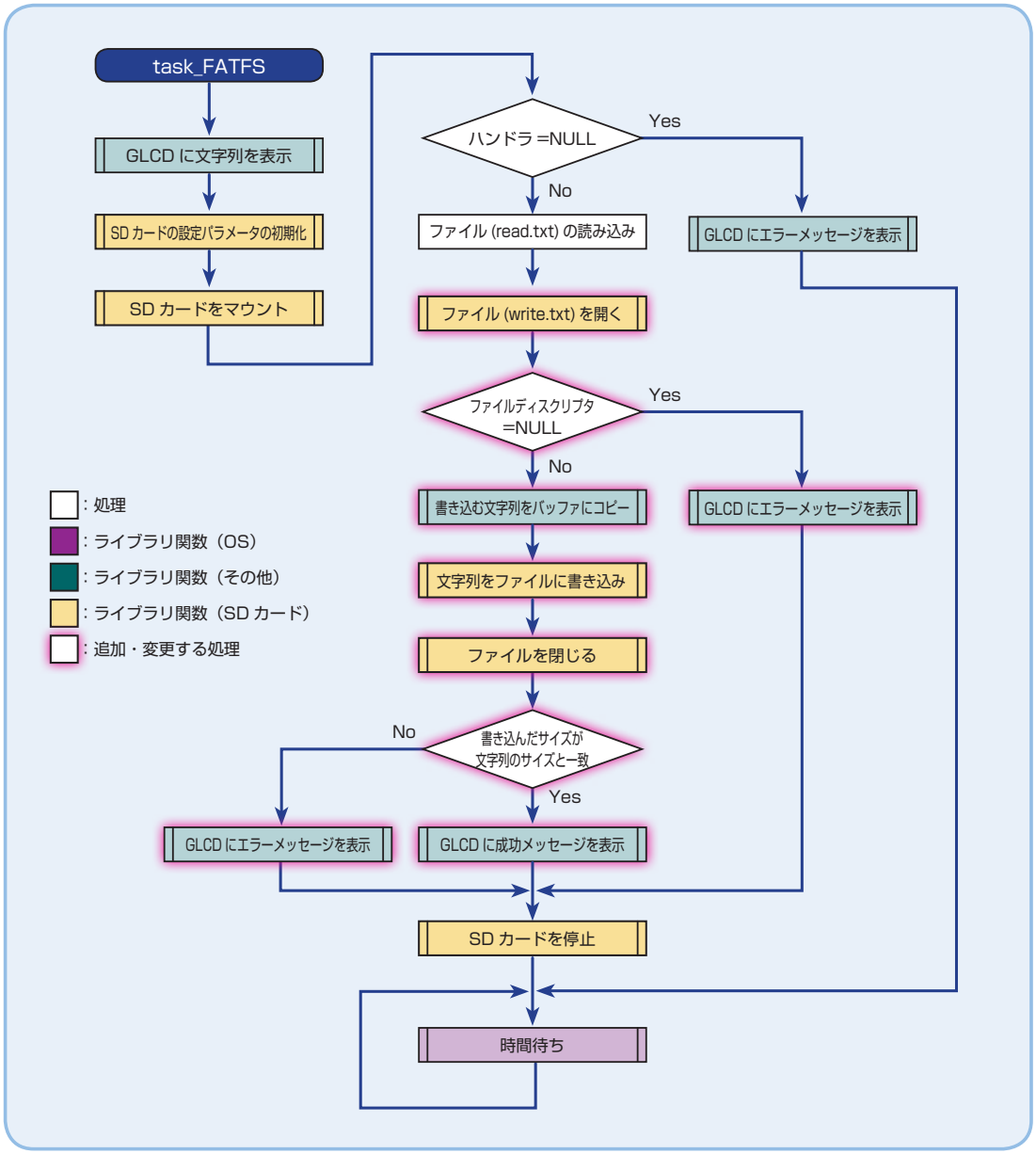
1. マイコンから、テキスト形式で microSD カードにデータを保存しましょう。
2. 書き込み成功なら GLCD に「Write Successful」と表示し、失敗した場合は「Write Error」と表示しましょう。
3. 書き込み成功後、microSD カード内のデータをパソコンで確認しましょう。



計測温度データの microSD への保存

フローチャート 14-2

以下は、課題 14-2 を実現するためのフローチャート例です。SD カードへの書き込みの手順は、読み込みの手順（STEP 09）とほとんど同じです。書き込むファイル名は、ここでは「write.txt」としています※。その他の関数・タスクは「フローチャート 14-1」と同じです。



※ FAT では大文字と小文字は区別されないので、実際に作成されるファイル名は大文字の「WRITE.TXT」になります。

ライブラリ関数 14-2

課題 14-2 で使用する関数を解説します。

なお、テキストカーソルをソースコード中の関数の部分に合わせると、その関数に関する情報がポップアップされます。さらに「F3」キーを押すと、その関数が定義されているファイルを開くことができます。

C 言語標準

文字列のコピー　char *strcpy(char *s1, const char *s2)

文字列 s2 を s1 にコピーする関数で、string.h で定義されています。文字列 s1 の大きさは、文字列 s2 の長さにヌル文字を加えた大きさが確保されている必要があります。

```
strcpy(buffer, "Hello World!");
```

引数の設定例

- buffer　：コピーした文字列を格納する配列。
- "Hello World!"：コピーする文字列。
  文字数 +1 が第一引数で指定した配列の長さを超えないよう注意すること。

ファイルへの書き込み　size_t fwrite(const void *buf, size_t size, size_t n, FILE *fp)

ファイルディスクリプタ fp のファイルに、アドレス buf で始まるデータを size バイトずつ n 個書き込みます。戻り値は実際に書き込んだ個数です。

```
bytes = fwrite(buffer, 1, strlen(buffer), fd);
```

引数の設定例

- buffer　：書き込むデータの最初のアドレス（ここでは、文字列 buffer）。
- 1　：1 バイトずつ書き込み（1 を指定すると、関数の戻り値が書き込んだバイト数になる）。
- strlen(buffer)：書き込む個数（1 バイトずつ書き込むことにしているので、ここでは書き込むバイト数＝文字列 buffer の長さを指定している）。
- fd　：書き込むファイルのファイルディスクリプタ。

計測温度データの microSD への保存

コーディング main.c 14-2

以下は、main.c 14-1 のソースコードを元にしたコーディング例です。■は main.c 14-1 から追加した部分です（■はコメントの変更です）。ここではバッファを読み込み・書き込みに使うので、わかりやすいようにバッファ名を STEP 09 のものから変更しています。

main.c

これ以前の行に変更はありません

```
328 /*
329  * FATFS タスク
330  */
331 Void task_FATFS(UArg arg0, UArg arg1)
332 {
333     SDSPI_Handle sdspiHandle; // SD カードマウントのハンドラ
334     SDSPI_Params sdspiParams; // SD カードマウントのパラメータ
335
336     FILE *fd; // ファイルディスクリプタ
337
338     char buffer[128]; // ファイル読み込み・書き込みバッファ
339     unsigned int bytes; // ファイル読み込み・書き込みサイズ
340
341     CONSOLE("FATFS タスクの開始 \n");
342
343     // GLCD に文字列を表示
344     GLCD_str(6, 0, "Read text:");
345
346     // 他のタスクが SD カードを使っている場合は使い終わるまで待つ
347     while (lockSD != 0) SLEEP(100);
348
349     // SD カードの使用を開始
350     lockSD = 1;
351
352     // SD カードの設定パラメータの初期化
353     SDSPI_Params_init(&sdspiParams);
354
355     // SD カードをマウント
356     sdspiHandle = SDSPI_open(Board_SDSPI0, 0, &sdspiParams);
357
358     // エラーチェック
359     if (sdspiHandle == NULL)
360     {
361         CONSOLE(" マウントに失敗しました！ \n");
362         // GLCD にエラーメッセージを表示
363         GLCD_str(7, 0, "Mount error");
364     }
365     else
366     {
367         // ファイルを開く
```

書き込みにも使うので、わかりやすいようにバッファの変数名を変更

```
368         fd = fopen("fat:0:read.txt", "r");
369
370         // エラーチェック
371         if (fd == NULL)
372         {
373             CONSOLE(" ファイルを開けませんでした！ \n");
374             // GLCD にエラーメッセージを表示
375             GLCD_str(7, 0, "File open error");
376         }
377         else
378         {
379             // ファイルから文字列を読み込み
380             bytes = fread(buffer, 1, sizeof(buffer) -1, fd);
381             buffer[bytes] = '\0';
382
383             // GLCD に読み込んだ文字列を表示
384             GLCD_str(7, 0, buffer);
385
386             // ファイルを閉じる
387             fclose(fd);
388         }
389
390         // ファイルを開く
391         fd = fopen("fat:0:write.txt", "w");
392
393         // エラーチェック
394         if (fd == NULL)
395         {
396             CONSOLE(" ファイルを開けませんでした！ \n");
397
398             // GLCD にエラーメッセージを表示
399             GLCD_str(7, 0, "File open error");
400         }
401         else
402         {
403             // 書き込む文字列をバッファにコピー
404             strcpy(buffer, "Hello World!");
405
406             // バッファの文字列をファイルに書き込み
407             bytes = fwrite(buffer, 1, strlen(buffer), fd);
408
409             // ファイルを閉じる
410             fclose(fd);
411
412             // 書き込みエラーチェック
413             if (bytes == strlen(buffer))
414             {
415                 GLCD_str(5, 0, "Write Successful");
416             }
417             else
```

計測温度データの microSD への保存

コーディング main.c 14-2

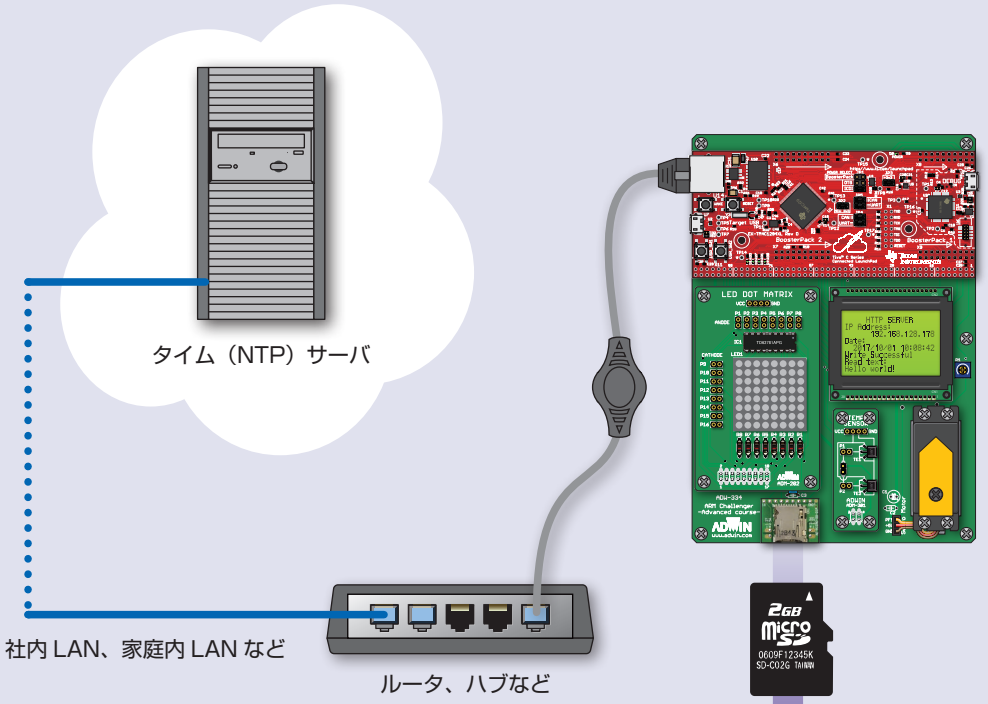
```
418             {
419                 GLCD_str(5, 0, "Write Error");
420             }
421         }
422
423         // SD カードを停止
424         SDSPI_close(sdspiHandle);
425     }
426
427     // SD カードの使用を終了
428     lockSD = 0;
429
430     while (1)
431     {
432         // 時間待ち
433         SLEEP(1000);
434     }
435 }
```

これ以降の行に変更はありません

計測温度データの microSD への保存

課題 14-3

温度を計測し、一定時間（1 分）毎に時刻とともに microSD カードに保存しましょう。
書き込み完了後、microSD カード内のデータをパソコンで確認しましょう。
なお、温度は 1 秒毎に 60 回（1 分間）計測したものを平均するものとします。



「log.txt」書き込みデータ例

日付	時刻	温度
2017/10/01	10:08	27.4
2017/10/01	10:09	27.3
2017/10/01	10:10	27.4
2017/10/01	10:11	27.3
：		

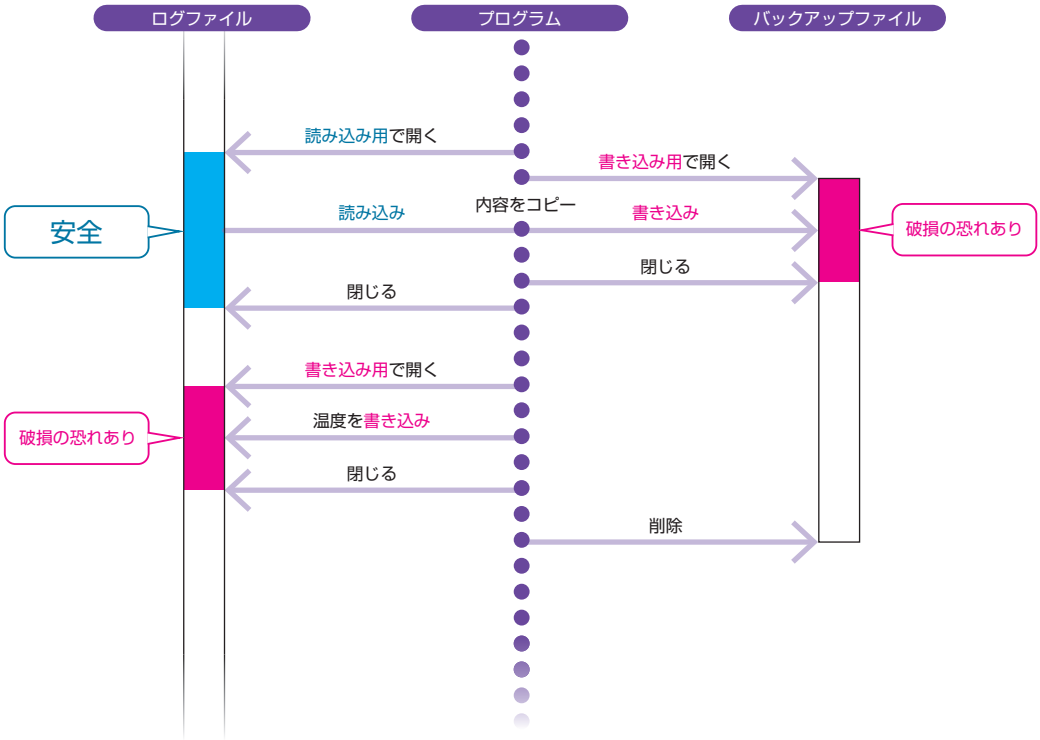
計測温度データの microSD への保存

SD カードへの書き込み時の注意

SD カードへの書き込みを行っている途中に、SD カードを取り出したりマイコンボードの電源を OFF にしたりする等、何らかの事情でファイルへの書き込みが中断されると、書き込み中のファイルが破損し中身が消えてしまうことがあります。

データの損失を避けるには、書き込みが終わるまで SD カードを抜き差ししたり電源を OFF にしたりできない仕組みを作るか、書き込み中はバックアップファイルを作り、万が一の際はバックアップファイルからデータを復元できるようにしておく必要があります。

以下は、書き込み時のバックアップ例です。書き込みが途中で中断した場合であっても、ログファイル・バックアップファイルの少なくとも一方にはデータが破損せずに残ります。



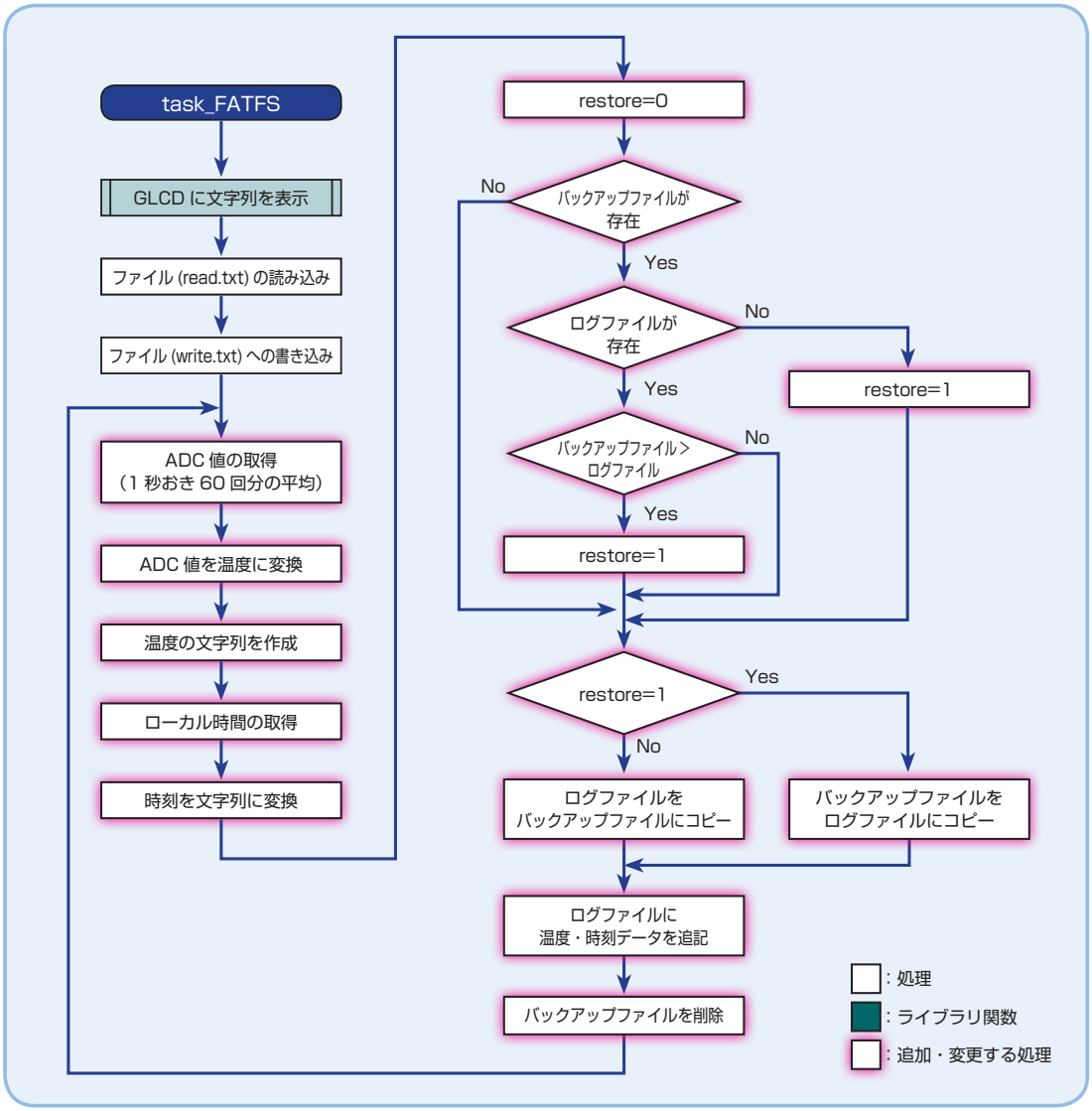
計測温度データの microSD への保存

フローチャート 14-3

以下は、課題 14-3 を実現するためのフローチャート例です。ここでは、

- ログファイルへの書き込み前にバックアップファイルを作成。
- バックアップファイルが存在し、ログファイルよりバックアップファイルのサイズのほうが大きい時、バックアップファイルからデータを復元。

といった手順によって、SD カードへの書き込み時のデータ損失を防ぐようにしています。その他の関数・タスクは「フローチャート 14-2」と同じです。



計測温度データの microSD への保存

ライブラリ関数　14-3

課題 14-3 で使用する関数を解説します。課題 14-3 で新たに使用する関数は、いずれも stdio.h で定義されている C 言語標準の関数です。

なお、テキストカーソルをソースコード中の関数の部分に合わせると、その関数に関する情報がポップアップされます。さらに「F3」キーを押すと、その関数が定義されているファイルを開くことができます。

C 言語標準

ファイル中の指定位置に移動　int fseek(FILE *fp, long offset, int origin)

ファイルディスクリプタ fp のファイル中の位置を、origin の示す位置から offset だけずれた位置に移動します。戻り値は正常時に 0、異常時に 0 以外の値です。

```
fseek(fd, 0L, SEEK_END);
```

引数の設定例

- fd　：ファイルディスクリプタ
- 0L　：指定位置からの移動無し
- SEEK_END　：ファイル中の位置（ファイル末尾は SEEK_END、ファイル冒頭は SEEK_SET、現在位置は SEEK_CUR）

ファイル中の現在位置の取得　long int ftell(FILE *fp)

ファイルディスクリプタ fp の現在のファイル中の位置を返します。以下は、fseek 関数と組み合わせてファイルのサイズを取得する例です。

```
fseek(fd, 0L, SEEK_END);
size = ftell(fd);
```

ファイルの終端の確認　int feof(FILE *fp)

ファイルディスクリプタ fp の現在のファイル中の位置が、ファイル終端に達したかどうかをチェックします。達した場合は 0 以外の値を、達していない場合は 0 を返します。

ファイルから 1 文字読み込み　int fgetc(FILE *fp)

ファイルディスクリプタ fp の示すファイルから 1 文字読み込んで返します。

ファイルに 1 文字書き込み　int fputc(int c, FILE *fp)

ファイルディスクリプタ fp の示すファイルに、データ c を書き込みます。書き込みに成功した場合は書き込んだ c を、失敗した場合は EOF（=-1）を返します。

ファイルの削除　int remove(const char *filename)

filename で指定されたファイルを削除します。削除に成功した時は 0 を、失敗した時は 0 以外の値を返します。

```
remove(log_back);
```

引数の設定例

- log_back　：削除するファイル名の文字列（ここでは、ファイル名の文字列を格納した配列）。

計測温度データの microSD への保存

コーディング main.c 14-3

以下は、main.c 14-2 のソースコードを元にしたコーディング例です。■は main.c 14-2 から追加した部分です。

main.c

```
/* XDCtools ヘッダファイル */
#include <xdc/std.h>
#include <xdc/runtime/System.h>

/* BIOS ヘッダファイル */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/hal/Seconds.h>

/* C 言語ヘッダファイル */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

この間の行に変更はありません

```
/*
 * FATFS タスク
 */
Void task_FATFS(UArg arg0, UArg arg1)
{
    SDSPI_Handle sdspiHandle; // SD カードマウントのハンドラ
    SDSPI_Params sdspiParams; // SD カードマウントのパラメータ

    int i; // 汎用カウンタ

    FILE *fd; // ファイルディスクリプタ
    FILE *fd_back; // バックアップファイル用ファイルディスクリプタ

    char buffer[128]; // ファイル読み込み・書き込みバッファ
    unsigned int bytes; // ファイル読み込み・書き込みサイズ

    char log[] = "fat:0:log.txt";           // ログファイル
    char log_back[] = "fat:0:log_b.txt";    // バックアップファイル

    const int sampling = 60;   // 温度データのサンプリング数
    unsigned int data;         // ADC データ格納用変数
    int temp;                   // 温度の絶対値
    char minus;                 // 温度の符号

    char Temp[6];   // 温度の文字列データ格納用配列
    char tstr[17];  // 日時の文字列データ格納用配列

```

```
356     int restore;   // バックアップファイルからの復旧フラグ
357
358     CONSOLE("FATFS タスクの開始 \n");
```

この間の行に変更はありません

```
447     while (1)
448     {
449         /*
450          * 温度データの文字列の作成
451          */
452
453         // 温度データの取得（1 秒おき 60 回分の平均を取る）
454         data = 0;
455         for (i = 0; i < sampling; i++)
456         {
457             // ADC が使用中の場合は終わるまで待つ
458             while (getADCStatus() == 1)
459                 SLEEP(10);
460             // ADC 値の取得
461             data += getADC(ADC_P1);
462             CONSOLE("ADC 平均値 : %d (%d/%d)\n", data / (i + 1), i + 1, sampling);
463             // 1 秒待つ
464             SLEEP(1000);
465         }
466         data /= sampling;
467
468         // ADC 値を温度（0.1℃単位）に変換
469         temp = (data * 3.3 * 1000 / 4096 - 600);
470
471         // 温度の絶対値と符号を取得
472         if (temp < 0)
473         {
474             temp = -temp;
475             minus = 1;
476         }
477
478         // 絶対値が 999 以上の場合は 999（± 99.9℃）とする
479         if (temp > 999)
480             temp = 999;
481         CONSOLE("Temp: %d\n", temp);
482
483         // 温度の文字列を作成
484         Temp[4] = '0' + temp % 10;          // 小数点第 1 位
485         Temp[3] = '.';                      //「.」
486         Temp[2] = '0' + (temp / 10) % 10;   // 1 の位
487         if (temp / 100 > 1)                 // 10 の位および符号
488         {
489             Temp[1] = '0' + (temp / 100) % 10;
490             Temp[0] = ' ';
```

計測温度データの microSD への保存

コーディング main.c 14-3

```
            if (minus == 1)
                Temp[0] = '-';
        }
        else
        {
            Temp[1] = ' ';
            if (minus == 1)
                Temp[1] = '-';
            Temp[0] = ' ';
        }
        Temp[5] = '\0';

        /*
         * 現在時刻の文字列の作成
         */

        // ローカル時間の取得
        time_t timer = time(NULL);
        struct tm *t_st = localtime(&timer);

        // 時刻を文字列に変換
        tstr[0] = (t_st->tm_year + 1900) / 1000 + 0x30;
        tstr[1] = ((t_st->tm_year + 1900) / 100) % 10 + 0x30;
        tstr[2] = ((t_st->tm_year + 1900) / 10) % 10 + 0x30;
        tstr[3] = (t_st->tm_year + 1900) % 10 + 0x30;
        tstr[4] = '/';
        tstr[5] = (t_st->tm_mon + 1) / 10 + 0x30;
        tstr[6] = (t_st->tm_mon + 1) % 10 + 0x30;
        tstr[7] = '/';
        tstr[8] = (t_st->tm_mday) / 10 + 0x30;
        tstr[9] = (t_st->tm_mday) % 10 + 0x30;
        tstr[10] = ' ';
        tstr[11] = (t_st->tm_hour) / 10 + 0x30;
        tstr[12] = (t_st->tm_hour) % 10 + 0x30;
        tstr[13] = ':';
        tstr[14] = (t_st->tm_min) / 10 + 0x30;
        tstr[15] = (t_st->tm_min) % 10 + 0x30;
        tstr[16] = '\0';
        CONSOLE("log: %s %s\n", tstr, Temp);

        /*
         * ログファイルのチェックとバックアップ
         */

        // 復旧フラグをクリア
        restore = 0;

        // 他のタスクが SD カードを使っている場合は使い終わるまで待つ
        while (lockSD != 0) SLEEP(100);

```

```
541         // SD カードの使用を開始
542         lockSD = 1;
543 
544         // SD カードの設定パラメータの初期化
545         SDSPI_Params_init(&sdspiParams);
546 
547         // SD カードをマウント
548         sdspiHandle = SDSPI_open(Board_SDSPI0, 0, &sdspiParams);
549 
550         // エラーチェック
551         if (sdspiHandle == NULL)
552         {
553             // SD カードの使用を終了
554             lockSD = 0;
555             CONSOLE(" マウントに失敗しました！ \n");
556 
557             // GLCD にエラーメッセージを表示
558             GLCD_str(7, 0, "Mount error");
559             continue;
560         }
561 
562         // ログファイルを読み込み専用で開く
563         fd = fopen(log, "r");
564 
565         // バックアップ用ファイルを読み込み専用で開く
566         fd_back = fopen(log_back, "r");
567 
568         // バックアップファイルが存在する場合
569         if (fd_back)
570         {
571             if (!fd)
572             {
573                 // ログファイルが存在しない時、バックアップファイルからの復旧を行う
574                 restore = 1;
575             }
576             else
577             {
578                 // 各ファイルの末尾に移動
579                 fseek(fd, 0L, SEEK_END);
580                 fseek(fd_back, 0L, SEEK_END);
581 
582                 CONSOLE(" ログ : %d, バックアップ : %d\n", ftell(fd), ftell(fd_back));
583 
584                 // バックアップファイルほうがファイルサイズが大きい時、
585                 // バックアップファイルからの復旧を行う
586                 if (ftell(fd_back) > ftell(fd))
587                 {
588                     restore = 1;
589                 }
590                 // 各ファイルの先頭に移動
```

計測温度データの microSD への保存

コーディング main.c 14-3

```
591                 fseek(fd, 0L, SEEK_SET);
592                 fseek(fd_back, 0L, SEEK_SET);
593             }
594         }
595 
596         // 復旧フラグが立っている時、バックアップファイルをログファイルにコピーし、
597         // そうでない場合、ログファイルをバックアップファイルにコピーする
598         if (restore)
599         {
600             CONSOLE(" ログファイル破損。復旧を開始 \n");
601 
602             // ログファイルをいったん閉じ、書き込み専用で開く
603             if (fd != NULL)
604                 fclose(fd);
605             fd = fopen(log, "w");
606 
607             // バックアップファイルをログファイルにコピーする
608             if (fd)
609             {
610                 while (1)
611                 {
612                     if (feof(fd))
613                         break;
614                     fputc(fgetc(fd_back), fd);
615                 }
616             }
617             else
618             {
619                 CONSOLE(" ログファイルを復旧できません！ \n");
620             }
621         }
622         else
623         {
624             // ログファイルが存在する場合、ログファイルをバックアップ
625             if (fd)
626             {
627                 CONSOLE(" ログファイルをバックアップ \n");
628 
629                 // バックアップファイルをいったん閉じ、書き込み専用で開く
630                 if (fd_back != NULL) fclose(fd_back);
631                 fd_back = fopen(log_back, "w");
632 
633                 // ログファイルをバックアップファイルにコピーする
634                 if (fd_back)
635                 {
636                     i = 0;
637                     while (1)
638                     {
639                         if (feof(fd))
640                             break;
```

```
641                         fputc(fgetc(fd), fd_back);
642                     }
643                 }
644                 else
645                 {
646                     CONSOLE("バックアップできません！\n");
647                 }
648             }
649         }
650 
651         // ファイルを閉じる
652         if (fd != NULL)
653             fclose(fd);
654         if (fd_back != NULL)
655             fclose(fd_back);
656 
657         /*
658          * ログファイルへの温度の記録
659          */
660 
661         // ログファイルを追記専用で開く
662         fd = fopen(log, "a");
663 
664         // エラーチェック
665         if (!fd)
666         {
667             CONSOLE("ログファイルを開けませんでした！\n");
668 
669             // GLCD にメッセージ表示
670             GLCD_str(5, 0, "File open error");
671         }
672         else
673         {
674             // 時刻の文字列をファイルに書き込み
675             bytes = fwrite(tstr, 1, strlen(tstr), fd);
676 
677             // 1 文字空ける
678             if (fputc(' ', fd) != EOF)
679                 bytes++;
680 
681             // 温度の文字列をファイルに書き込み
682             bytes += fwrite(Temp, 1, strlen(Temp), fd);
683 
684             // 改行
685             bytes += fwrite("\r\n", 1, 2, fd);
686 
687             // ファイルを閉じる
688             fclose(fd);
689 
690             // 書き込みエラーチェック
```

計測温度データの microSD への保存

コーディング main.c 14-3

```
691             if (bytes == strlen(tstr) + strlen(Temp) + 3)
692             {
693                 GLCD_str(5, 0, "Write Successful");
694             }
695             else
696             {
697                 GLCD_str(5, 0, "Write Error");
698             }
699         }
700
701         /*
702          * バックアップファイルを削除
703          */
704         remove(log_back);
705
706         // SD カードを停止
707         SDSPI_close(sdspiHandle);
708
709         // SD カードの使用を終了
710         lockSD = 0;
711     }
712 }
```

これ以降の行に変更はありません

ARM サーバを LAN 外部からアクセスできるようにする

LAN 内に構築された ARM サーバを外部に公開するには、ルータの設定が必要です。
固定グローバル IP を取得している場合は、ルータの NAT（Network Address Translation）や IP マスカレード（Internet Protocol Masquerade）機能を使えば、外部公開可能です。
ルータの設定については、お使いのルータの説明書をご覧ください。

固定グローバル IP を取得していない場合は、ダイナミック DNS サービスを利用する方法もあります。

以上で「ARM チャレンジャー応用編」の解説は終了です。

ARM プロセッサを搭載したサーバは、巨大なデータベースサーバやいくつものサーバソフトウェアが稼働する Windows サーバに比肩するものではありません。
しかし、低消費電力を売りにするミニマムなサーバの需要は、IoT 時代の現代においては、今後ますます増えることが予想されます。消費電力の低さは ARM プロセッサの特徴です。今まで考えられなかったものにサーバが組み込まれ、ネットにつながることが可能になるのです。皆さんなら何にサーバを組み込みますか？