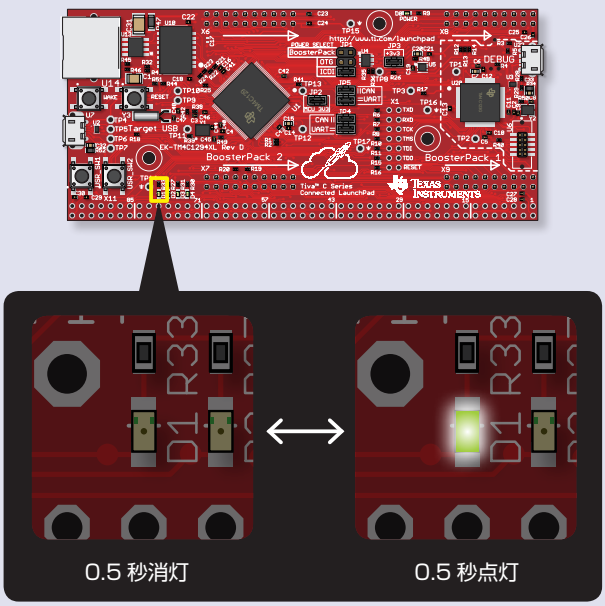
LED の点滅

学習内容

STEP 05 で実行済みですが、サンプルソースについて理解しておきましょう。
以後、LED の点滅は OS の正常動作インジケータとして利用します。

課題 06

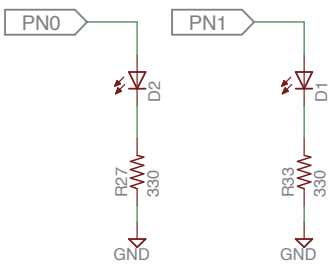
マイコンボードの LED1(D1) を 1 秒周期で点滅させましょう。



配線 06

ユーザー LED（D1 および D2）は、マイコンボード上で図のように配線されています。
High で点灯、Low で消灯になるアクティブハイの回路です。

マイコンボード	
PN0	LED_D1
PN1	LED_D2



LED の点滅

ソースコード　empty.c 03

STEP 03 で作成したサンプルプロジェクトは、マイコンボードの LED1(D1) を 1 秒おきに点灯・消灯させる（2 秒周期で点滅させる）もので、本 STEP で作りたいプログラムはこれとほとんど同じ動作になります。まずは、このサンプルプロジェクトのソースコード（プロジェクトディレクトリ直下の「empty.c」）を見てみましょう。ソースコード中のインクルードファイル、ライブラリ関数等については後ほど解説します。

empty.c

1 ～ 32 行までのコメントは省略しています

```
33 /*
34  *  ======== empty.c ========
35  */
36 /* XDCtools Header files */
37 #include <xdc/std.h>
38 #include <xdc/runtime/System.h>
39
40 /* BIOS Header files */
41 #include <ti/sysbios/BIOS.h>
42 #include <ti/sysbios/knl/Task.h>
43
44 /* TI-RTOS Header files */
45 // #include <ti/drivers/EMAC.h>
46 #include <ti/drivers/GPIO.h>
47 // #include <ti/drivers/I2C.h>
48 // #include <ti/drivers/SDSPI.h>
49 // #include <ti/drivers/SPI.h>
50 // #include <ti/drivers/UART.h>
51 // #include <ti/drivers/USBMSCHFatFs.h>
52 // #include <ti/drivers/Watchdog.h>
53 // #include <ti/drivers/WiFi.h>
54
55 /* Board Header file */
56 #include "Board.h"
57
58 #define TASKSTACKSIZE   512
59
60 Task_Struct task0Struct;
61 Char task0Stack[TASKSTACKSIZE];
62
63 /*
64  *  ======== heartBeatFxn ========
65  *  Toggle the Board_LED0. The Task_sleep is determined by arg0 which
```

```
 *  is configured for the heartBeat Task instance.
 */
Void heartBeatFxn(UArg arg0, UArg arg1)
{
    while (1) {
        Task_sleep((unsigned int)arg0);
        GPIO_toggle(Board_LED0);
    }
}

/*
 *  ======== main ========
 */
int main(void)
{
    Task_Params taskParams;
    /* Call board init functions */
    Board_initGeneral();
    // Board_initEMAC();
    Board_initGPIO();
    // Board_initI2C();
    // Board_initSDSPI();
    // Board_initSPI();
    // Board_initUART();
    // Board_initUSB(Board_USBDEVICE);
    // Board_initUSBMSCHFatFs();
    // Board_initWatchdog();
    // Board_initWiFi();

    /* Construct heartBeat Task  thread */
    Task_Params_init(&taskParams);
    taskParams.arg0 = 1000;
    taskParams.stackSize = TASKSTACKSIZE;
    taskParams.stack = &task0Stack;
    Task_construct(&task0Struct, (Task_FuncPtr)heartBeatFxn, &taskParams, NULL);

     /* Turn on user LED */
    GPIO_write(Board_LED0, Board_LED_ON);

    System_printf("Starting the example\nSystem provider is set to SysMin. "
                  "Halt the target to view any SysMin contents in ROV.\n");
    /* SysMin will only print to the console when you call flush or exit */
    System_flush();

    /* Start BIOS */
    BIOS_start();

    return (0);
}
```

LED 点滅タスク

マイコンの初期設定

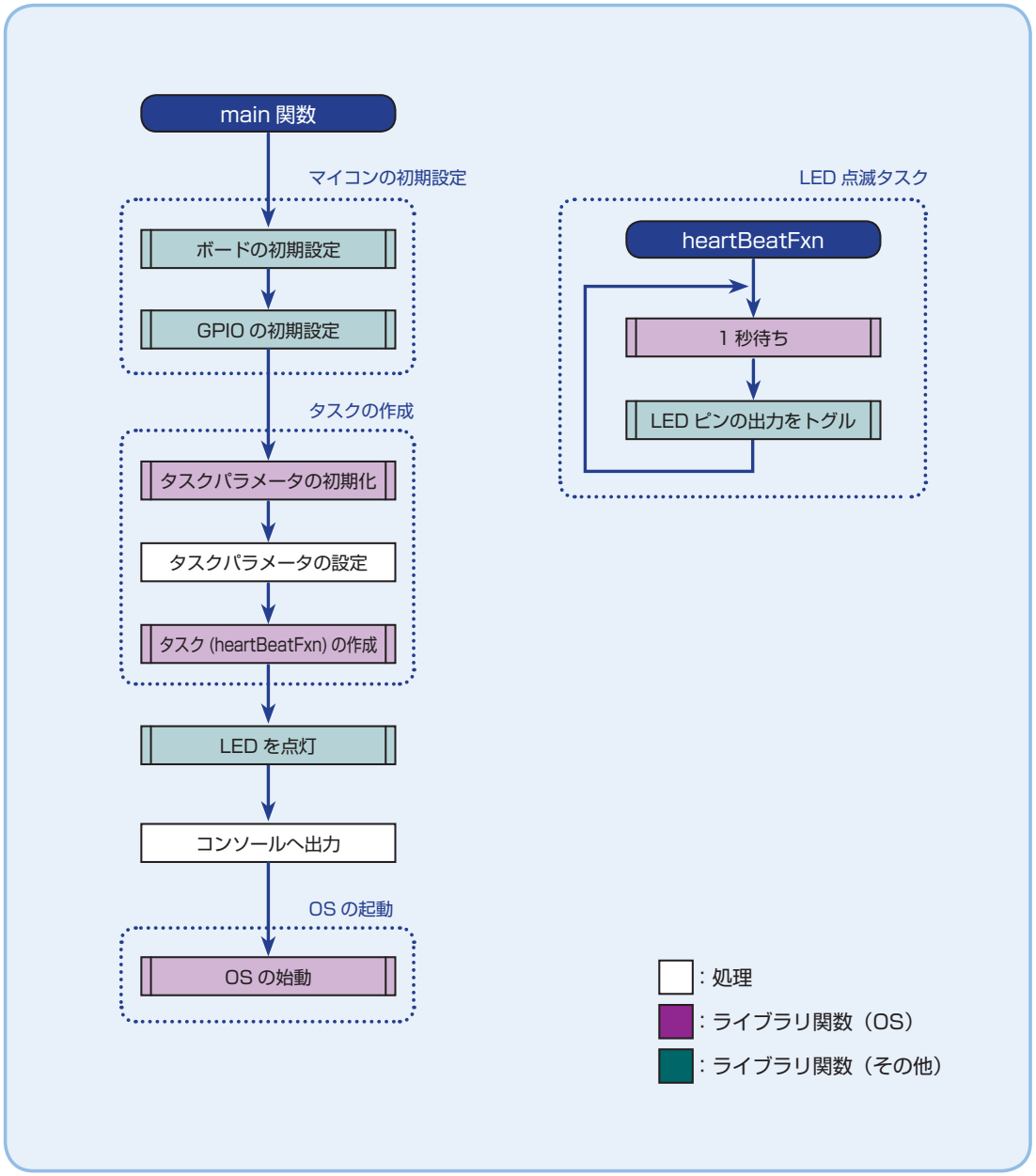
タスクの作成

ＯＳの起動

LED の点滅

フローチャート 03

フローチャートは以下のようになっています。main 関数では OS 起動までの処理を行っています。OS が実行するタスクは、ここでは 1 つだけで、「heartBeatFxn」です。



インクルードファイル 03

STEP 03 で作成したサンプルプロジェクトのソースコード中でインクルードされているファイルを解説します。
インクルードされているファイルは、テキストカーソルをファイル名部分に合わせて「F3」キーを押すことで開くことができます。

XDCtools ヘッダファイル

XDCtools は、CCS の一部として提供されているリアルタイム組み込みシステム向けのパッケージ群で、TI-RTOS で使われる重要な API も XDCtools で提供されています。

XDCtools 関連のドキュメントは、CCS の「Help」→「Help Contents」の「XCDtools *」から見ることができます。ヘッダファイルおよびソースファイルは、「C:\ti/xdctools_*_core\packages」内にあります。

xdc/std.h

```
#include <xdc/std.h>
```

「xdc/std.h」では、XDC で使う型の定義等を行っています。

xdc/runtime/System.h

```
#include <xdc/runtime/System.h>
```

デバッグ時に CCS のコンソールに文字列を表示するための関数「System_printf」および「System_flush」は、ここで定義されています。

BIOS ヘッダファイル

BIOS（TI-RTOS のカーネル部分）に関するドキュメントは、CCS の「Help」→「Help Contents」の「TI-RTOS for TivaC *」→「Documentation Links」にある「Kernel Documentation」から見ることができます。ヘッダファイルおよびソースファイルは、「C:\ti\tirtos-tivac_*\products\bios_*\packages」内にあります。

ti/sysbios/BIOS.h

```
#include <ti/sysbios/BIOS.h>
```

OS を起動する関数「BIOS_start」は、ここで定義されています。

LED の点滅

インクルードファイル 03

ti/sysbios/knl/Task.h

```
#include <ti/sysbios/knl/Task.h>
```

タスク関係の関数や型（構造体）が定義されています。今回のプログラムでは、以下のものがここで定義されています。

- Task_Struct
- Task_Params
- Task_FuncPtr
- Task_Params_init()

ドライバヘッダファイル

OS のドライバに関するドキュメントは、CCS の「Help」→「Help Contents」の「TI-RTOS for TivaC *」→「Documentation Links」にある「Drivers Documentation」から見ることができます。ヘッダファイルおよびソースファイルは、「C:\ti\tirtos-tivac_*\products\tidrivers_tivac_*\packages」内にあります。

ti/driver/GPIO.h

```
#include <ti/drivers/GPIO.h>
```

GPIO 関係の関数や定数は、ここで定義されています。今回のプログラムでは、以下のものがここで定義されています。

- GPIO_toggle()
- GPIO_write()

ボードヘッダファイル

Board.h

```
#include "Board.h"
```

プロジェクトディレクトリ直下にある「Board.h」では、本コースで用いるマイコンボード向けの関数が定義されています。今回のプログラムでは、以下のものがここで定義されています。

- Board_LED0
- Board_LED_ON
- Board_initGeneral()
- Board_initGPIO()

型・構造体 03

STEP 03 で作成したサンプルプロジェクトのソースコード中で使われている型・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。

Char, Void, UArg

Char、Void、UArg は、xdc（および TI-RTOS）で使われる型として「xdc/std.h」で間接的に定義されています。実体としては、「Char」型は「char」型、「Void」型は「void」型、「UArg」型は「unsigned int」型に定義されています。

詳細は CCS の「Help」→「Help Contents」から「XDCtools *」→「API Reference (cdoc)」→「all packages」→「xdc」をご覧ください。

タスクの構造　Task_Struct

タスクの構造に関する情報を保持する構造体で、「ti/sysbios/knl/Task.h」で間接的に定義されています。メンバ変数を直接操作することはありません。

タスクのパラメータ　Task_Params

タスクの設定値や、その他のパラメータに関する構造体で、「ti/sysbios/knl/Task.h」で間接的に定義されています。主なメンバ変数として、以下のものがあります。

- arg0　：タスクの関数に渡す第一引数を設定できます。
- arg1　：タスクの関数に渡す第二引数を設定できます。
- priority　：タスクの優先度を設定できます。
  優先度は 1 ～ 15 の整数で設定し、数値が大きいほど優先度が高くなります。
- stack　：スタックのポインタを指定します。
- stackSize　：スタックサイズを指定します。

詳細は CCS の「Help」→「Help Contents」から「TI-RTOS for TivaC *」→「Documentation Links」→「Kernel Documentaion」→「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」→「package ti.sysbios.knl;」→「module Task;」→「typedef struct Task_Params ...」をご覧ください。

タスク関数の型　Task_FuncPtr

タスク関数の型で、「ti/sysbios/knl/Task.h」で間接的に定義されています。実体は以下のとおりです。

```
typedef Void (*Task_FuncPtr)(UArg,UArg);
```

LED の点滅

ライブラリ関数 03

STEP 03 で作成したサンプルプロジェクトのソースコード中で使われているライブラリを解説します。

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。

ボード関係

ボード関係の関数は、プロジェクトディレクトリ直下の「Board.h」で定義されていますが、このファイルではマクロで関数名を置き換えているだけで、関数の実体は同じくプロジェクトディレクトリ直下の「EX_TM4C1294XL.c」（ヘッダファイルは「「EX_TM4C1294XL.h」」）にあります。

Tiva シリーズマイコンボードを使う場合、関数ひとつで必要な設定を行ってくれるので便利です。ただし、関数で用いる定数名とボード上のパーツ名がずれている場合もあるのでご注意ください。

ボードの初期設定　void Board_initGeneral(void)

ボードの初期設定を行います。関数の実体は「EX_TM4C1294XL.c」中の「EK_TM4C1294XL_initGeneral」（149 行目～）であり、TivaWare のドライバライブラリ関数である SysCtlPeripheralEnable 関数で I/O ポートをすべて有効化しています※1。

本 STEP の範囲内では、以下の関数（I/O ポート N を有効化）を実行しているのと同じ意味になります。

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
```

なお、SysCtlPeripheralEnable 関数についての詳細は「C:\ti\tirtos_tivac_*\products\TivaWare_C_Series-*\docs\SW-TM4C-DRL-UG-*.pdf」の「SystemControl」→「API Functions」→「SysCtlPeripheralEnable」をご覧ください。もしくは、「No.07 ARM チャレンジャー入門編」をご参照ください。

ボードの GPIO の初期設定　void Board_initGPIO(void)

ボードの GPIO の初期設定を行います。この関数では、LED1(D1) を制御する PN1 ピンや、LED2(D2) を制御する PN0 ピンが出力に設定されます※2。この関数の実体は「EX_TM4C1294XL.c」中の「EK_TM4C1294XL_initGeneral」（326 ～ 330 行目）であり、内容としては「GPIO_init」関数を呼び出しているだけですが、この関数は直前に定義されている構造体変数「GPIOTiva_config」（315 ～ 321 行目）に基づいて GPIO の設定を行います。

より詳細な内容については STEP 10 で触れますが、本 STEP の範囲内では以下の関数（PN1 を出力に設定）を実行しているのと同じ意味になります。

```
GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1);
```

なお、GPIOPinTypeGPIOOPutput 関数についての詳細は「C:\ti\tirtos_tivac_*\products\TivaWare_C_Series-*\docs\SW-TM4C-DRL-UG-*.pdf」の「GPIO」→「API Functions」→「GPIOPinTypeGPIOOutput」をご覧ください。もしくは、「No.07 ARM チャレンジャー入門編」をご参照ください。

ドライバ関係

GPIO 関係のドライバ関数は「ti/driver/GPIO.h」で定義されています。これらの関数は、「EX_TM4C1294XL.c」で定義されている構造体変数「GPIOTiva_config」(315 ～ 321 行目) に基づいて GPIO を制御します（詳細は STEP 10 で触れます）。

GPIO 関係のドライバ関数についてのドキュメントは、CCS の「Help」→「Help Contents」の「TI-RTOS forTivaC *」→「Documentation Links」→「Drivers Documents」→「TI-RTOS Drivers Runtime APIs(doxygen)」→「GPIO.h」にあります。

ピンの書き込み　void GPIO_write(unsigned int index, unsigned int value)

ピン index に値 value を書き込みます。サンプルプログラム中では、

```
GPIO_write(Board_LED0, Board_LED_ON);
```

のように使われており、第一引数は書き込みを行うピン、第二引数は書き込む値を表します。第一引数の「Board_LED0」は、ここではボード上の LED1(D1) を制御する PN1 ピンに対応します※3。第二引数の「Board_LED_ON」は、「EK_TM4C1294XL.h」で定義されており、実体は整数「1」です。

本 STEP では、以下の関数（PN1 ピンに 1 を書き込み）を実行しているのと同じ意味になります。

```
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, GPIO_PIN_1);
```

なお、GPIOPinWrite 関数についての詳細は「C:\ti\tirtos_tivac_*\products\TivaWare_C_Series-*\docs\SW-TM4C-DRL-UG-*.pdf」の「GPIO」→「API Functions」→「GPIOPinWrite」をご覧ください。もしくは、「No.07 ARM チャレンジャー入門編」をご参照ください。

※1　A ～ T の各ポート（I、O は紛らわしいので無し）を有効化する処理になっています。ただし、R ～ T ポートは本コースのマイコンである TM4C1294NCPDT には存在していません。
※2　なお、LED3(D3) と LED4(D4) はイーサネットの監視用（STEP 08 参照）で、ここでは扱いません。
※3　ボード上の LED2(D2) を制御する PN0 ピンは、「Board_LED1」または「Board_LED2」と指定します。イーサネット監視用の「LED3(D3)」、「LED4(D4)」はここでは扱いません。

LED の点滅

ライブラリ関数 03

ピンのトグル　void GPIO_toggle(unsigned int index)

ピン index の値をトグルします。サンプルプログラム中では、

```
GPIO_toggle(Board_LED0);
```

のように使われており、引数は書き込みを行うピンを表します（GPIO_write と同様です）。本 STEP では、以下の処理と同じ意味になります。

```
int32_t value = GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1);
value ^= GPIO_PIN_1;
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, value);
```

なお、GPIOPinRead 関数についての詳細は「C:\ti\tirtos_tivac_*\products\TivaWare_C_Series-*\docs\SW-TM4C-DRL-UG-*.pdf」の「GPIO」→「API Functions」→「GPIOPinRead」をご覧ください。もしくは、「No.07 ARM チャレンジャー入門編」をご参照ください。

コンソール関係

「xdc/runtime/System.h」で定義されている XDC のライブラリ関数には、デバッグモードで CCS のコンソールに文字列を表示させる関数が含まれています。これらは、プログラムの動作確認を行う際に便利です。

詳細は CCS の「Help」→「Help Contents」から「XDCtools *」→「API Reference (cdoc)」→「all packages」→「xdc.runtime」→「System」をご覧ください。

文字列の書き出し　Int System_printf(CString fmt, …)

コンソールに表示する文字列のデータを、文字列表示用の内部バッファに書き出します。書き出されたデータを実際にコンソールに表示するには、後述の System_flush 関数を実行する必要があります。

使い方は C 言語の printf 関数とおおむね同じです（書式に多少制限があります。なお、printf 関数についての詳細は、C 言語の入門書等をご参照ください）。バッファサイズは 128 バイトで、それを超える場合は古いデータから上書きされます。

書き出された文字列の表示　System_flush()

文字列表示用の内部バッファに書き出されたデータをコンソールに表示し、同時に内部バッファをクリアします。

OS 関係

以下は、RTOS の設定や起動に関する関数です。

これらの関数のドキュメントは CCS の「Help」→「Help Contents」の「TI-RTOS for TivaC *」→「Documentation Links」→「Kernel Documentation」→「TIRTOSKernel Runtime APIs and Configuration (cdoc)」以下にあります。

タスクの設定パラメータの初期化　Task_Param_init(Task_Params *params)

「ti/sysbios/knl/Task.h」で定義されており、タスクの設定パラメータを初期化します。タスクの設定パラメタは、最初にこの関数で初期化する必要があります。サンプルプログラム中では、

```
Task_Params_init(&taskParams);
```

のように使われており、taskParams に設定パラメタの初期値が代入されます。

詳細は「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」から「package ti.sysbios.knl;」→「module Task;」→「Void Task_Params_init(Task_Params *params);」をご覧ください。

タスクのインスタンスを作成 Task_construct(Task_Struct *structP,Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb)

「ti/sysbios/knl/Task.h」で定義されており、タスクのインスタンスを作成します。サンプルプログラム中では、

```
Task_construct(&taskLEDStruct, (Task_FuncPtr)task_LED, &taskParams, NULL);
```

のように使われており、関数 heartBeatFxn の処理を行うタスクを taskParams の設定に従って作成します。

詳細は「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」から「package ti.sysbios.knl;」→「module Task;」→「Void Task_construct(Task_Struct *structP, Task_FuncPtr fxn,const Task_Params *params, Error_Block *eb);」をご覧ください。

LED の点滅

ライブラリ関数 03

現在のタスクの遅延　Task_sleep(UInt32 nticks)

「ti/sysbios/knl/Task.h」で定義されており、現在のタスクを、nticks で指定したティックだけ遅延させます（待ち状態にします）。1 ティックの長さは cfg ファイル（本 STEP では empty.cfg）で指定されており、デフォルトでは 1000 マイクロ秒（＝ 1 ミリ秒＝ 1000 分の 1 秒）です。サンプルプログラム中では、

```
Task_sleep((unsigned int)arg0);
```

のように使われており、arg0 で指定した時間（ここでは 1000 ティック＝ 1 秒）だけスリープします。

詳細は「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」から「package ti.sysbios.knl;」→「module Task;」→「Void Task_sleep(UInt32 nticks); 」をご覧ください。

システムの始動　BIOS_start()

「ti/sysbios/BIOS.h」で定義されており、システム（OS）を始動します。main 関数内で、他のすべての初期設定を行った後に呼び出します。また、main 関数内でこの関数以降に書かれている処理は実行されることはありません※。

詳細は「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」から「package ti.sysbios;」→「module BIOS;」→「Void BIOS_start();」をご覧ください。

タスクの関数

タスクの処理内容は、以下のような、引数をふたつ持つ戻り値無しの関数で記述します（先述のとおり、UArg 型と Void 型は、実体はそれぞれ unsigned int 型および void 型です）。

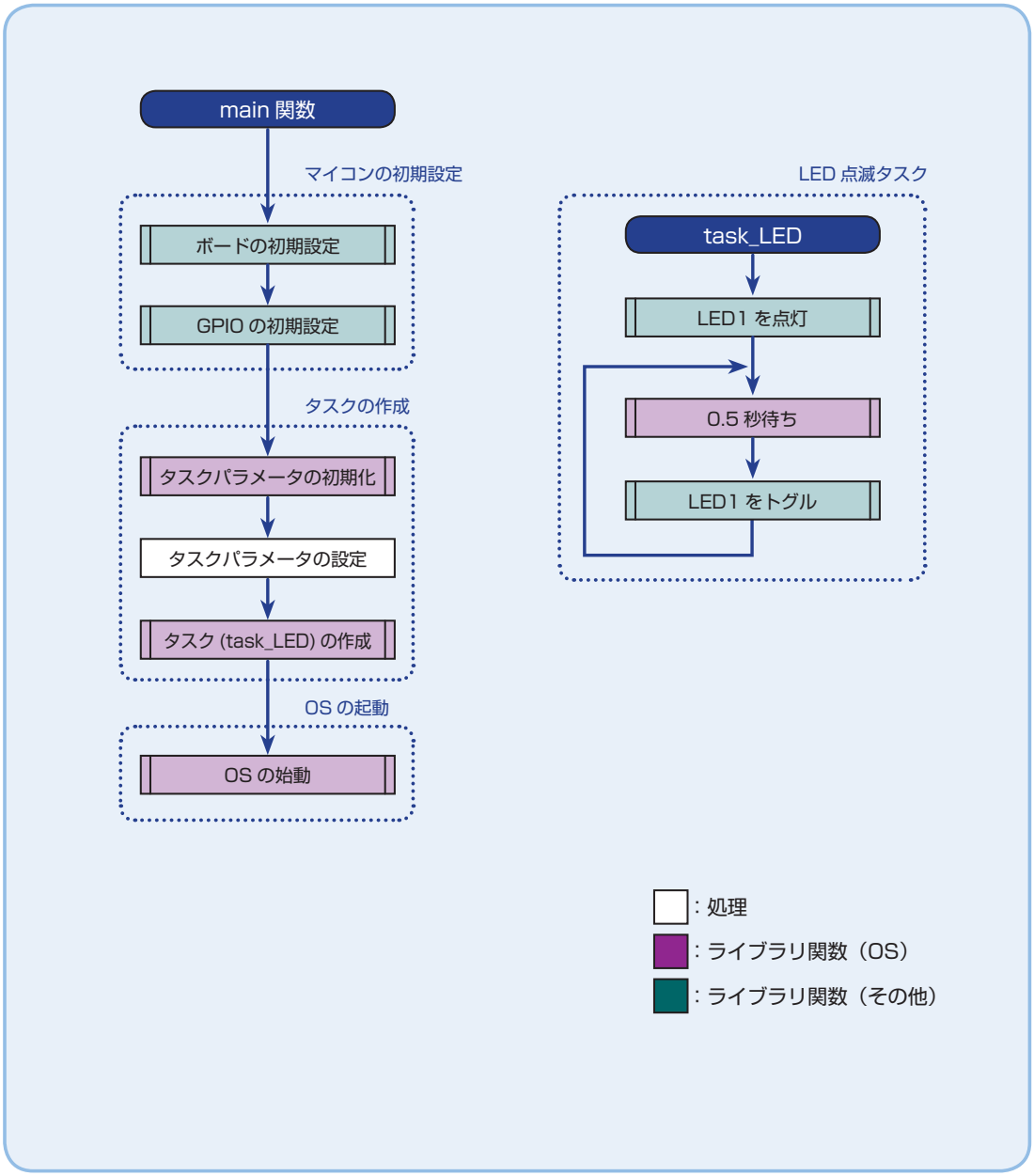
```
Void taskFxn(UArg arg0, UArg arg1) {
    /* タスクの処理内容 */
}
```

ふたつの引数には、ポインタや正の整数を格納することができ、Task_Param 構造体のメンバ変数 arg0 および arg1 で指定します。サンプルプログラム中では、第一引数（arg0）で LED の点滅間隔を指定するようになっています。

※　サンプルプログラム中では、BIOS_start(); の後に「return (0);」という命令が書かれていますが、これは main 関数が int 型の戻り値を持つために形式的に書かれているものであり、実際に処理がここまで達することはありません。

フローチャート 06

ここで、あらためて本 STEP の課題を実現するためのフローチャートを考えてみましょう。サンプルソースのものを、待ち時間のみ変更してもかまいませんが、以下ではその他にも若干変更を加えています。



LED の点滅

インクルードファイル 06

課題 06 の実現にあたり、追加でインクルードすると便利なファイルです。

BIOS ヘッダファイル

XDCtools は、CCS の一部として提供されているリアルタイム組み込みシステム向けのパッケージ群で、TI-RTOS で使われる重要な API も XDCtools で提供されています。

ti/sysbios/knl/Clock.h

```
#include <ti/sysbios/knl/Clock.h>
```

クロック関係の関数や定数（例えば 1 秒のティック数を表す Clock_tickPeriod 等）が定義されています。

定数 06

課題 06 の実現にあたり、追加で使用すると便利な定数です。

Clock_tickPeriod

「ti/sysbios/knl/Clock.h」で定義されている定数で、1 ティックが何マイクロ秒であるかを示しています。TI-RTOS の初期設定では 1 ティックは 1000 マイクロ秒であり、Clock_tickPeriod も 1000 を表します。時間関係の処理を行いたい時は、この定数を使うと便利です。

詳細は CCS の「Help」→「Help Contents」から「TI-RTOS for TivaC *」→「Documentation Links」→「Kernel Documentation」→「TI-RTOS Kernel Runtime APIs and Configuration (cdoc)」→「package ti.sysbios.knl;」→「module Clock;」→「extern const UInt32 Clock_tickPeriod;」をご覧ください。

ユーザー定義マクロ 06

文字列の書き出しやタスクの遅延は、提供されている関数そのままでは、毎回使うのにやや煩わしさがあります。以下のようなユーザー定義マクロを使うのも良いでしょう。

```
#define CONSOLE(...) do { System_printf(__VA_ARGS__); System_flush(); } while(0)
#define SLEEP(X) Task_sleep((X)*1000/Clock_tickPeriod)
```

CONSOLE() は、System_printf() と System_flush() をまとめたもので、文字列表示用の内部バッファに書き出し、それをただちに CCS 上に表示するマクロです。なお、System_printf() の解説にもあるとおり、内部バッファのサイズは 128 バイトなので、このマクロで一度に表示できる文字列も 128 バイトまでです。

SLEEP() は遅延時間をミリ秒で指定するマクロで、仮に 1 ティックの長さを変えることになったとしても遅延の実時間は変わらずに済みます。

なお、これらの処理は開発環境や OS、その他ソフトウェアライブラリへの依存性の強いものですが、マクロにしてしまうことで、他の環境への移植が必要な際にも修正作業量を減らすことができます。

LED の点滅

コーディング main.c 06

プログラムの基本的な流れを理解するためには、新規作成した Empty プロジェクトから empty.c ファイルを削除し、新たなソースコードファイルを作成して自分なりの方法でソースコードをゼロから書いてみるのも良いでしょう。

ファイルの削除は、プロジェクトエクスプローラ上で削除したいファイルを右クリックし、「Delete」を選択することで行うことができます。またソースコードファイルの新規作成は、プロジェクトエクスプローラ上でプロジェクトフォルダを右クリックして「New」→「Source File」を選択し、作りたいソースファイル名（例えば「main.c」）を指定することで行うことができます。また、既存のファイルの名前を変更したい場合は、プロジェクトエクスプローラ上で名前を変更したいファイルを右クリックし、「Rename」を選択して名前を変更します。

以下は、新規作成した「main.c」のコーディング例です。

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
10 /* ドライバヘッダファイル */
11 #include <ti/drivers/GPIO.h>
12
13 /* ボードヘッダファイル */
14 #include "Board.h"
15
16 /* ユーザー定義マクロ */
17 #define CONSOLE(...) do { System_printf(__VA_ARGS__); System_flush(); } while(0)
18 #define SLEEP(X) Task_sleep((X)*1000/Clock_tickPeriod)
19
20 /*
21  * ユーザータスクの優先度
22  */
23 #define TASK_LED_PRIO 1
24
25 /*
26  * ユーザースタックのサイズ
27  */
28 #define TASK_LED_STACK 512
29
```

```
30 /*
31  * ユーザータスクの構造体
32  */
33 Task_Struct taskLEDStruct;
34
35 /*
36  * ユーザータスクのスタック
37  */
38 Char taskLEDStack[TASK_LED_STACK];
39
40 /*
41  * ユーザータスク
42  */
43
44 /*
45  * LED 点滅タスク
46  */
47 Void task_LED(UArg arg0, UArg arg1)
48 {
49     CONSOLE("LED 点滅タスクの開始 \n");
50
51     // LED1 を点灯
52     GPIO_write(Board_LED0, Board_LED_ON);
53
54     while (1)
55     {
56         // LED を 0.5 秒おきにトグル
57         SLEEP(500);
58         GPIO_toggle(Board_LED0);
59     }
60 }

61     次 STEP 以降の新しいタスクはここに追加していくと良いでしょう

62 /*
63  * メイン
64  */
65
66 int main(void)
67 {
68     Task_Params taskParams;
69
70     // ボードの初期設定
71     CONSOLE(" ボードの初期設定 \n");
72     Board_initGeneral();
73     Board_initGPIO();
74
```

LED の点滅

コーディング main.c 06

```
75      // ユーザータスクの作成
76      CONSOLE(" ユーザータスクの作成 \n");
77      Task_Params_init(&taskParams);
78      taskParams.stackSize = TASK_LED_STACK;
79      taskParams.stack = &taskLEDStack;
80      taskParams.priority = TASK_LED_PRIO;
81      Task_construct(&taskLEDStruct, (Task_FuncPtr) task_LED, &taskParams, NULL);
82
83      // OS の起動
84      CONSOLE("OS の起動 \n");
85      BIOS_start();
86
87      return (0);
88  }
89
```

LED を点滅させることはできましたか？　本コースの以降の STEP では、このソースコードに書き足していくことにします。

コードの整形　【CCS】

コードを選択し キーボードで Ctrl ＋ Shift ＋ F　を同時に押す、もしくは、コードを右クリック ＞ Source ＞ Format を選択すると、インデントを揃えて見易く整形されます。

整形 前

```
Void task_LED(UArg arg0, UArg arg1){
CONSOLE("LED 点滅タスクの開始 \n");

// LED1 を点灯
GPIO_write(Board_LED0, Board_LED_ON);

while (1){
// LED を 0.5 秒おきにトグル
SLEEP(500);
GPIO_toggle(Board_LED0);
}
}
```

整形 後

```
Void task_LED(UArg arg0, UArg arg1)
{
    CONSOLE("LED 点滅タスクの開始 \n");

// LED1 を点灯
    GPIO_write(Board_LED0, Board_LED_ON);

    while (1)
    {
// LED を 0.5 秒おきにトグル
        SLEEP(500);
        GPIO_toggle(Board_LED0);
    }
}
```