AN205234

FM MCU における µC/OS-II 移植とアプリケーション

関連製品ファミリ: FM3 Family と FM4 Family

本書では Keil µVision または IAR EWARM IDE 上で µC/OS-II を FM MCU に移植する手順を説明します。また FSSDC-9B506-EK 簡易 Kit に基づき、FM3 MCU 用の OS 環境におけるアプリケーションプログラムを開発する手順も説明します。

Contents

1 はじめに

1.1 µC/OS-II について

µC/OS-II は主に C 言語で書かれた、プリエンプティブな優先度ベースのマイクロプロセッサ用リアルタイムマルチタスクオペレーティングシステムカーネルで、主に組込みシステムでの使用を想定しています。

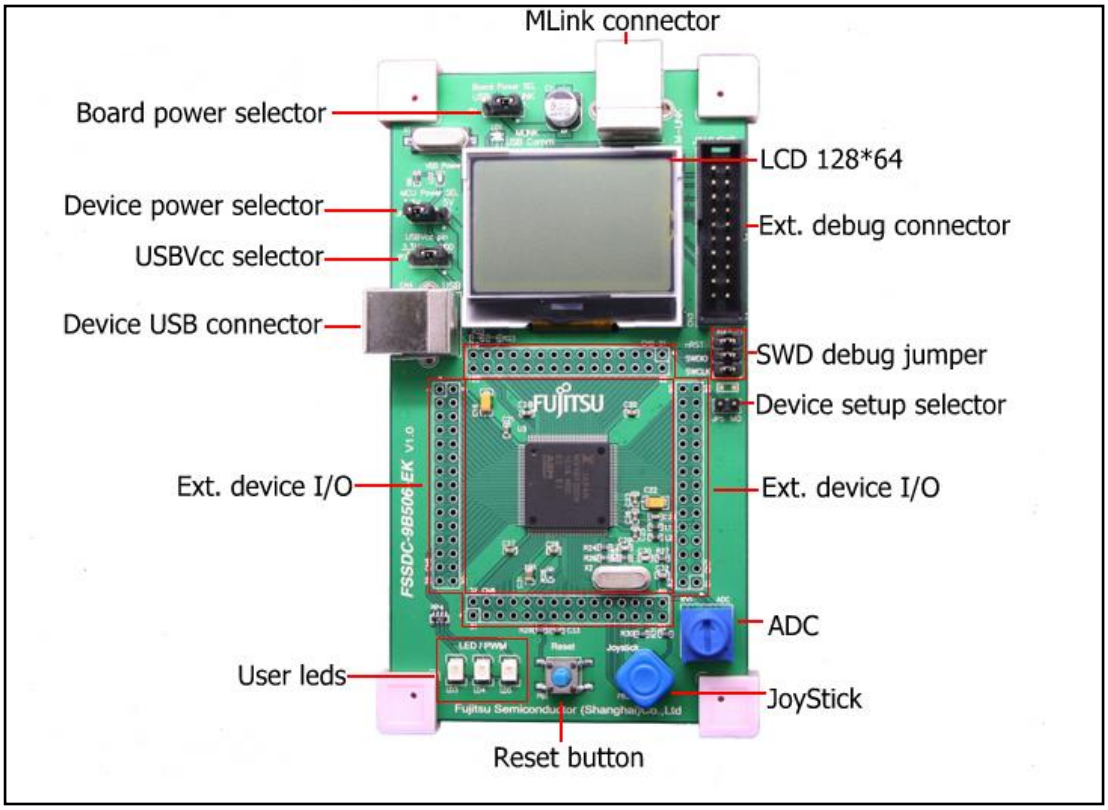
µC/OS-II は RTOS から期待するすべてのサービス (タスク管理，時間やタイマ管理，セマフォ，ミューテックス, メッセージメールボックスや待ち行列，イベントフラグなど) を提供します。

1.2 FSSDC-9B506 簡易キットについて

FSSDC-9B506-EK 簡易キットは以下の特徴を持った MB9B500 シリーズの MCU の使い方を素早く簡単に学ぶことができます。

- ARM Cortex-M3®コアを搭載した、FM3 ファミリの 32 ビット産業向け汎用 MCU
- Cypress の高い信頼性と高速セキュア組込みフラッシュテクノロジ
- モータ制御タイマ，ADC，広域の通信インタフェースなどの周辺機能
- 顧客のアプリケーションに簡単に実装できる、オンボード M-Link
- 図 1 で示す MCU 周辺機能をデモするための単純な外部リソース (LCD, ジョイスティック, LED, ポテンショメータ)

図 1. FSSDC-9B506-EK 簡易キット概要



1.3 本書について

本書では Keil µVision または IAR EWARM IDE 上で µC/OS-II を FM MCU に移植する手順を説明します。また FSSDC-9B506-EK 簡易 Kit に基づき、FM3 MCU 用の OS 環境におけるアプリケーションプログラムを開発する手順も説明します。

2 ハードウェアの設定

FSSDC-9B506-EK 簡易キットは複数のデバッグ方法を提供します。

IAR EWARM IDE を使用する場合、図 2 や図 3 に示すようにデバッグ用にオンボード M-Link や外部 J-link を採用できます。Keil µVision IDE を使用する場合、図 4 に示すように U-link を採用できます。

図 2. デバッグ用オンボード M-Link の使用

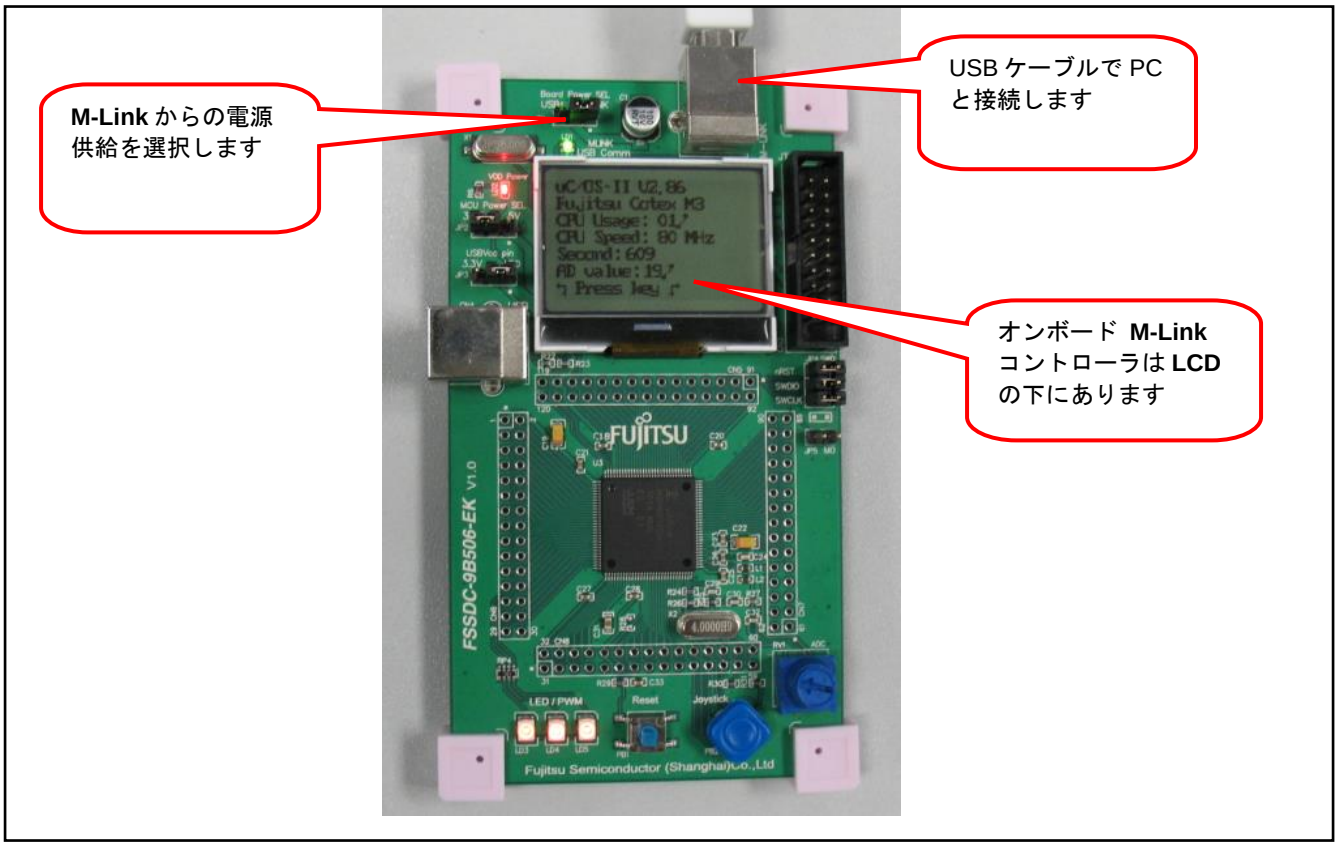


図 3. デバッグ用 J-Link の使用

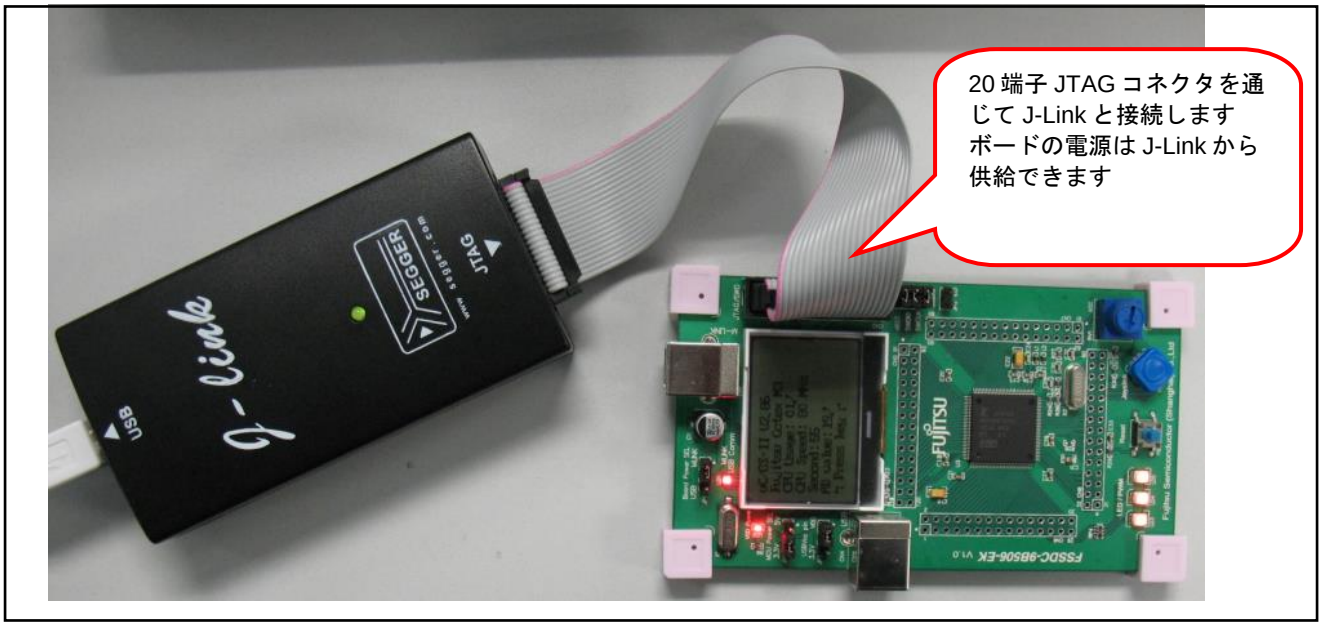
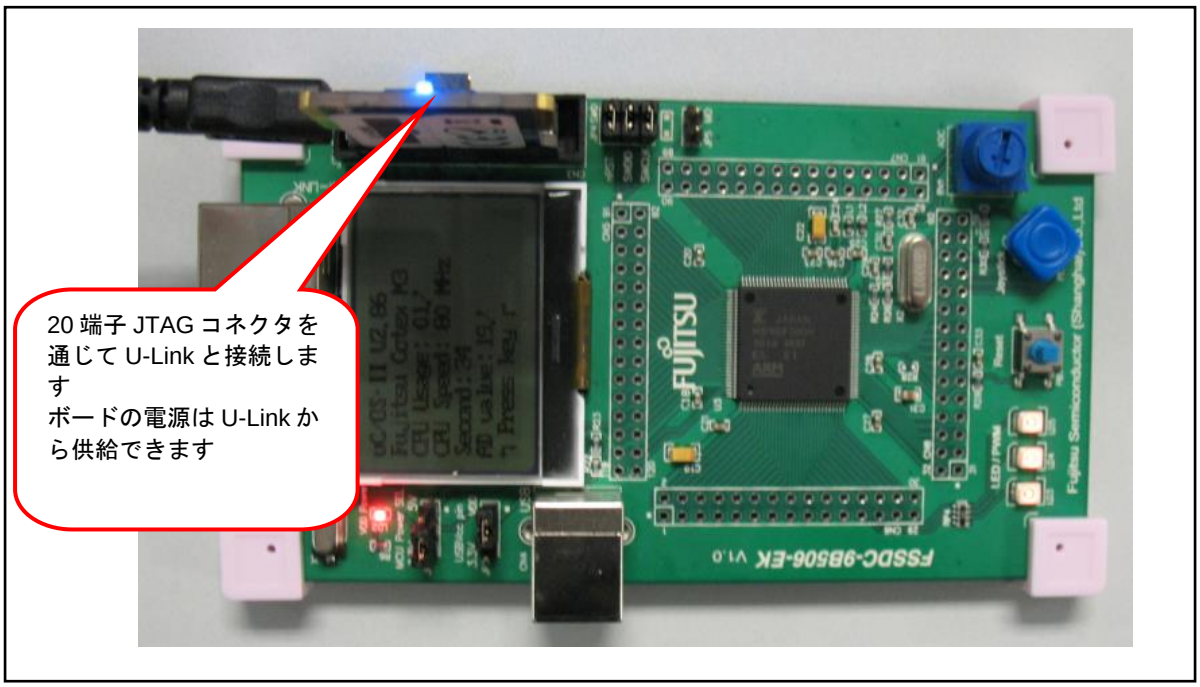


図 4. デバッグ用 U-Link の使用



3 µC/OS-II の FM3 MCU への移植

本節では µC/OS-II を FM3 MCU に移植する方法について説明します。ほとんどの µC/OS-II は移植性のために C で書かれていますが、プロセッサ特有のコードは C やアセンブラ言語で書く必要があります。特に µC/OS-II はアセンブラ言語でだけ操作可能なプロセッサレジスタを持っています。

図 5. µC/OS-II ハードウェア/ソフトウェア アーキテクチャ

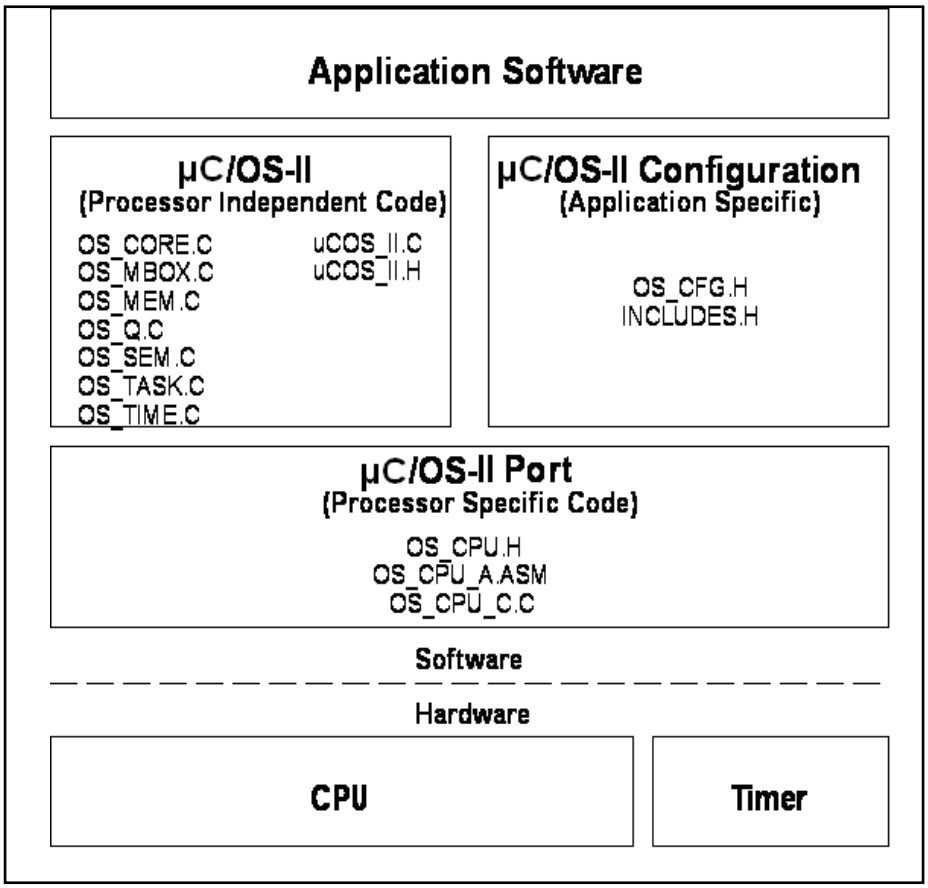


図 5 は µC/OS-II のアーキテクチャとハードウェアとの関係を示しています。µC/OS-II の移植とは、実は 3 つのファイル (OS_CPU.H, OS_CPU_A.ASM, OS_CPU_C.C) を含むプロセス特有のコードと関連付けるだけです。µC/OS-II は IARCC (IAR の C コンパイラ) と ARMCC (ARM の C コンパイラ) の両方へ移植できます。他のファイルのデフォルトの設定を含む、移植の基本的な手順は以下のとおりです。

- #define 値に 1 を設定し、10 のデータ型と 3 つの#define マクロを宣言 (OS_CPU.H)
- 5 つのアセンブラ言語関数を記述 (OS_CPU_A.ASM)
- 2 つの単純な関数を C 言語で記述 (OS_CPU_C.C)

3.1 OS_CPU_C.C の修正

3.1.1 #define 値に 1 を設定

図 6. #define 値に 1 を設定

```
/* Stack grows from HIGH to LOW memory on ARM */
#define    OS_STK_GROWTH              1
```

3.1.2 10 のデータ型を宣言

図 7. 10 のデータ型を宣言

```
typedef unsigned      char     BOOLEAN;
typedef unsigned      char     INT8U;          /* Unsigned      8 bit quantity */
typedef signed        char     INT8S;          /* Signed        8 bit quantity */
typedef unsigned      short    INT16U;         /* Unsigned     16 bit quantity */
typedef signed        short    INT16S;         /* Signed       16 bit quantity */
typedef unsigned      int      INT32U;         /* Unsigned     32 bit quantity */
typedef signed        int      INT32S;         /* Signed       32 bit quantity */   (1)
typedef float                  FP32;           /* Single precision floating point */
typedef double                 FP64;           /* Double precision floating point */

typedef unsigned      int      OS_STK;         /* Each stack entry is 32-bit wide */   (2)
typedef unsigned      int      OS_CPU_SR;      /* Define size of CPU status register (PSR = 32 bits)*/   (3)
```

(1) ARM Cotex-M3 は 32-bit プロセッサなので、unsigned short と signed short は INT16S と INT16U を定義し、unsigned int と signed int は INT32U と INT32S を定義します。

(2) プロセッサのスタック要素は 32-bit なので、OS_STK 用に unsigned int を定義します。

(3) CPU ステータスレジスタは 32-bit なので、OS_CPU_SR 用に unsigned int を定義します。

3.1.3 3 つの#define マクロを宣言

図 8. 3 つの#define マクロを宣言

```
#define    OS_CRITICAL_METHOD   3

#if        OS_CRITICAL_METHOD == 3
#define    OS_ENTER_CRITICAL()        {cpu_sr = OS_CPU_SR_Save();}
#define    OS_EXIT_CRITICAL()         {OS_CPU_SR_Restore(cpu_sr);}          (1)
#endif

#define    OS_TASK_SW()               OSCtxSw()                             (2)
```

(1) コンパイラメーカに選択された実装方法を隠すために、µC/OS-II は割込みを無効/有効にする 2 つのマクロ (OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()) を定義します。METHOD 3 は割込み状態を補完することにより割込みを無効/有効にすることを意味します。一般的にはローカル変数 "cpu_sr" で割込み無効フラグの状態を補完し、その後、割込みを無効にします。CPU ステータスレジスタに 'cpu_sr' をコピーし、割込み無効状態を復元します。

(2) µC/OS-II がレディタスクを実行するためにタスクスタックからすべてのプロセッサレジスタを復元し、割込みから return を実行します。コンテキストスイッチのために、割込みをシミュレートするには OS_TASK_SW() を実装します。OS_TASK_SW() はソフトウェア割込み (ARM Cotext-M3 の場合は PendSV) で呼ばれる、アセンブラ言語で書かれた OSCtxSw に定義されています。

3.2 OS_CPU_A.ASM の修正

µC/OS-II では 5 つの単純なアセンブラ言語の関数を記述する必要があります。

- OSStartHighRdy()
- OSCtxSw()
- OSIntCtxSw()
- OS_CPU_SR_Save () と OS_CPU_SR_Restore()
- OS_CPU_PendSVHandler

3.2.1 OSStartHighRdy() の記述

図 9. OSStartHightRdy 関数

```
OSStartHighRdy
        LDR     R0, =NVIC_SYSPRI14      ; Set the PendSV exception priority     (1)
        LDR     R1, =NVIC_PENDSV_PRI
        STRB    R1, [R0]

        MOVS    R0, #0                  ; Set the PSP to 0 for initial context switch call  (2)
        MSR     PSP, R0

        LDR     R0, =OSRunning          ; OSRunning = TRUE                      (3)
        MOVS    R1, #1
        STRB    R1, [R0]

        LDR     R0, =NVIC_INT_CTRL      ; Trigger the PendSV exception (causes context switch)
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]

        CPSIE   I                       ; Enable interrupts at processor level  (4)

OSStartHang
        B       OSStartHang             ; Should never get here
```

(1) この関数は最も高い優先度のタスク ready-to-run を開始するために OSStart() に呼ばれます。ユーザ割込みでブレークしない保証をするために最低の優先度に PendSV 例外を設定します。

(2) 最初の実行でコンテキストスイッチするために PSP を 0 に初期化します。

(3) マルチタスクスケジュールを起動するために OSRunning を TRUE に設定します。

(4) PendSV 例外は有効後にだけ起動できます。

3.2.2 OSCtxSw() の記述

図 10. OSCtxSw 関数

```
OSCtxSw
        LDR     R0, =NVIC_INT_CTRL          ; Trigger the PendSV exception (causes context switch)   (1)
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]
        BX      LR
```

(1) OS がタスクコンテキストスイッチするとき OSCtxSw() が呼ばれます。この関数は実動作の PendSV 例外を起動します。

3.2.3 OSIntCtxSw()の記述

図 11. OSIntCtxSw 関数

```
OSIntCtxSw
        LDR     R0, =NVIC_INT_CTRL          ; Trigger the PendSV exception (causes context switch)   (1)
        LDR     R1, =NVIC_PENDSVSET
        STR     R1, [R0]
        BX      LR
```

(1) 割込み結果によってコンテキストスイッチが必要な場合に、OSIntCtxSw() は OSIntExit() から呼ばれます。この関数はアクティブまたは有効な割込みがないときに扱われる、PendSV 例外を単純に起動します。

3.2.4 OS_CPU_SR_Save() と OS_CPU_SR_Restore() の記述

図 12. OS_CPU_SR_Save と OS_CPU_SR_Restore 関数

```
OS_CPU_SR_Save
        MRS     R0, PRIMASK                 ; Set prio int mask to mask all (except faults)   (1)
        CPSID   I
        BX      LR
OS_CPU_SR_Restore
        MSR     PRIMASK, R0                 ; (2)
        BX      LR
```

(1) すべての割込みをマスクし、PRIMASK の値を保管します。

(2) PRIMASK の値を復元します。

3.2.5 OS_CPU_PendSVHandler() の記述

PendSV 例外は、コンテキストスイッチを引き起こすために用いられます。これは Cortex-M3 でコンテキストスイッチを実行する推奨方法です。これは Cortex-M3 がどんな例外でもプロセッサコンテキストの半分を自動で保存し、例外から復帰と同時に、プロセッサコンテキストを元に戻すためです。つまり R4-R11 を保存する必要あり、スタック・ポインタを割り当てます。PendSV 例外を使うこの方法は、コンテキストの保存や削減がスレッド，割込み，例外によって発生するかどうかと同じであることを意味します。PendSV 例外ルーチンのコードは、図 13 のように表示されます。

図 13. OS_CPU_PendSVHandler 関数

```
OS_CPU_PendSVHandler
        CPSID   I                       ; Prevent interruption during context switch
        MRS     R0, PSP                 ; PSP is process stack pointer                  (1)
        CBZ     R0, OS_CPU_PendSVHandler_nosave     ; Skip register save the first time

        SUBS    R0, R0, #0x20           ; Save remaining regs r4-11 on process stack
        STM     R0, {R4-R11}            ;                                               (2)

        LDR     R1, =OSTCBCur           ; OSTCBCur->OSTCBStkPtr = SP;                   (3)
        LDR     R1, [R1]
        STR     R0, [R1]                ; R0 is SP of process being switched out
                                        ; At this point, entire context of process has been saved
OS_CPU_PendSVHandler_nosave
        PUSH    {R14}                   ; Save LR exc_return value
        LDR     R0, =OSTaskSwHook       ; OSTaskSwHook();                               (4)
        BLX     R0
        POP     {R14}

        LDR     R0, =OSPrioCur          ; OSPrioCur = OSPrioHighRdy;                    (5)
        LDR     R1, =OSPrioHighRdy
        LDRB    R2, [R1]
        STRB    R2, [R0]

        LDR     R0, =OSTCBCur           ; OSTCBCur  = OSTCBHighRdy;                     (6)
        LDR     R1, =OSTCBHighRdy
        LDR     R2, [R1]
        STR     R2, [R0]

        LDR     R0, [R2]                ; R0 is new process SP; SP = OSTCBHighRdy->OSTCBStkPtr; (7)
        LDM     R0, {R4-R11}            ; Restore r4-11 from new process stack          (8)
        ADDS    R0, R0, #0x20
        MSR     PSP, R0                 ; Load PSP with new process SP
        ORR     LR, LR, #0x04           ; Ensure exception return uses process stack
        CPSIE   I
        BX      LR                      ; Exception return will restore remaining context   (9)
```

(1) 保存部分 (最初のコンテキストスイッチ) をスキップ (d へ移行) する場合、プロセス SP を取得してください。

(2) プロセス・スタック上で残りの規則 r4-r11 を保存してください。

(3) TCB(OSTCBCur-OSTCBStkPtr = SP) で、プロセス SP を保存してください。

(4) OSTaskSwHook() を呼んでください。

(5) 現在の高い優先度 (OSPrioCur = OSPrioHighRdy) を取得してください。

(6) 現在準備中のスレッド (TCB, OSTCBCur = OSTCBHighRdy) を取得してください。

(7) TCB(SP = OSTCBHighRdy-OSTCBStkPtr) から、新しいプロセス SP を取得してください。

(8) 新しいプロセス・スタックから R4-R11 を復元してください。

(9) 残りのコンテキストを復元する例外復帰を実行してください。

3.3 OS_CPU_C.C の修正

µC/OS-II では 3 つの単純な C 言語の関数を記述する必要があります。

- OSTaskStkInit
- OS_CPU_SysTickInit
- OS_CPU_SysTickHandler

3.3.1 OSTaskStkInit の記述

この機能は、生成中のタスクのスタックフレームを初期化するために、OSTaskCreate() か OSTaskCreateExt() によって呼ばれます。この関数は高度なプロセッサ特有のものです。

図 14. OSTaskStkInit 関数

```
OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;
    (void)opt;                          /* 'opt' is not used, prevent warning */
    stk = ptos;                         /* Load stack pointer */

                                        /* Registers stacked as if auto-saved on exception */
                                        /* (1) */
    *(stk) = (INT32U)0x01000000L;       /* xPSR */
    *(--stk) = (INT32U)task;            /* Entry Point */
    *(--stk) = (INT32U)0xFFFFFFFEL;     /* R14 (LR) (init value will cause fault if ever used)*/
    *(--stk) = (INT32U)0x12121212L;     /* R12 */
    *(--stk) = (INT32U)0x03030303L;     /* R3 */
    *(--stk) = (INT32U)0x02020202L;     /* R2 */
    *(--stk) = (INT32U)0x01010101L;     /* R1 */
    *(--stk) = (INT32U)p_arg;           /* R0 : argument */

                                        /* Remaining registers saved on process stack */
    *(--stk) = (INT32U)0x11111111L;     /* R11 */
    *(--stk) = (INT32U)0x10101010L;     /* R10 */
    *(--stk) = (INT32U)0x09090909L;     /* R9 */
    *(--stk) = (INT32U)0x08080808L;     /* R8 */
    *(--stk) = (INT32U)0x07070707L;     /* R7 */
    *(--stk) = (INT32U)0x06060606L;     /* R6 */
    *(--stk) = (INT32U)0x05050505L;     /* R5 */
    *(--stk) = (INT32U)0x04040404L;     /* R4 */

    return (stk);
}
```

(1) 各タスクのスタックで R0-R12, LR, PC, xPSR の値を初期化してください。これらのレジスタの初期値は重要ではありません。初期値はデバッグ時に各レジスタを簡単に見分けられる値を設定してください。

3.3.2 OS_CPU_SysTickInit の記述

OS_CPU_SysTickInit は System tick タイマを初期化します。この関数は OSStart() が呼ばれプロセッサが初期化された後に、呼ばれる必要があります。

図 15. OS_CPU_SysTickInit 関数

```
void  OS_CPU_SysTickInit (void)
{
        INT32U  cnts;

        cnts = 80000000 / OS_TICKS_PER_SEC;                     (1)

        OS_CPU_CM3_NVIC_ST_RELOAD = (cnts - 1);             /* Enable timer. */
        OS_CPU_CM3_NVIC_ST_CTRL |=OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC | OS_CPU_CM3_NVIC_ST_CTRL_ENABLE;
                                                                (2)          /* Enable timer interrupt. */
        OS_CPU_CM3_NVIC_ST_CTRL  |= OS_CPU_CM3_NVIC_ST_CTRL_INTEN;                  (3)
}
```

(1) システムタイマカウントを設定します。80000000 は MCU コアのクロックです。

(2) システムタイマやタイマを有効にするためのソースクロックとして CPU クロックを選択します。

(3) システムタイマ割込みを有効にします。

3.3.3 OS_CPU_SysTickHandler の記述

OS_CPU_SysTickHandler は µC/OS-II tick 割込みを生成するために使用される system tick (SysTick) 割込みをハンドルします。この関数は Cortex-M3 ベクタテーブルのエントリ 15 に置く必要があります。

図 16. OS_CPU_SysTickHandler 関数

```
void  OS_CPU_SysTickHandler (void)
{
        OS_CPU_SR  cpu_sr;

        OS_ENTER_CRITICAL();        /* Tell uC/OS-II that we are starting an ISR */
        OSIntNesting++;             /* (1) */
        OS_EXIT_CRITICAL();

        OSTimeTick();               /* Call uC/OS-II's OSTimeTick()  (2) */

        OSIntExit();                /* Tell uC/OS-II that we are leaving the ISR */
}
```

(1) 割込みネスト変数を増やしてください。

(2) システム時間を同調するために µC/OS-II の OSTimeTick() を呼んでください。

4 FM3 MCU における µC/OS-II アプリケーション

移植完了後、µC/OS-II プラットフォームにおいてアプリケーションプログラムを作成できます。本アプリケーションノートではアプリケーションプログラムを作成する方法を説明し、FSSDC-9B506-EK 簡易キットにおける簡単な実例で OS の並列実現を実演します。

4.1 タスク一覧

本アプリケーションノートにおける例は表 1 で示す通り全部で 6 つのタスクを生成します。

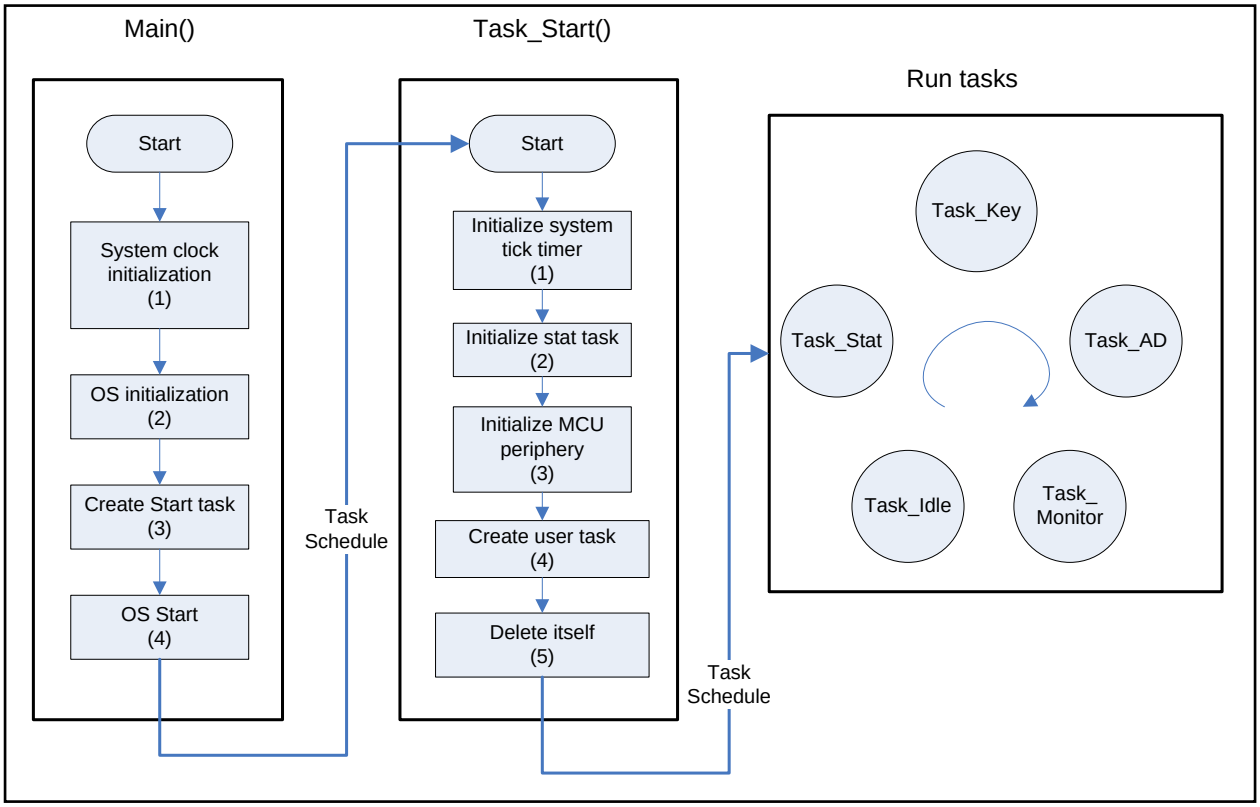
表 1. OSTaskStkInit 関数

タスク名	優先度	説明
Task_Start "Start"	2	このタスクは MCU ペリフェラル, system tick タイマ, システムタスクを初期化するため、またはユーザタスクを生成するために使用されます。従ってこのタスクは OS をスケジュール開始後最初に実装してください。
Task_AD "AD"	13	このタスクは AD channel 0 によって AD 値入力をサンプリングします。
Task_Key "Key"	14	このタスクはキー押下状態をスキャンします。
Task_Monitor "Monitor"	249	このタスクはシステムパラメータを入手し、LCD でそれらを表示します。
Task_Stat "uC/OS-II Stat"	251	これは CPU 使用量を計算するシステムタスクです。
Task_Idle "uC/OS-II Idle"	252	これは他のすべてのタスクがサスペンドしたときに実行するシステムタスクです。

4.2 アプリケーション実装のフロー

OS システムにおけるプログラムデザインは 従来のものとは非常に異なっています。ユーザが独自のアプリケーションを作る場合、信頼できる startup プロシジャがコンパイルされる必要があります。図 17 は µC/OS-II システムを起動するための方法を示しています。

図 17. アプリケーションコードフロー



main 関数の手順は以下のとおりです。

(1) システムクロックを初期化します。デフォルト設定: HCLK= 80MHz, PCLK0= PCLK1= PCLK2=40MHz.

(2) OS を初期化するために OSInit() を呼びます。主に TCB を実装し、イベントを初期化し、idle, stat, timer task を含むシステムタスクを生成します。stat と timer task を使用するかどうかは、図 18 で示す OS_CFG.H 内の設定に依存します。timer task はこの例では使用されません。

図 18. システムタスク設定

```
#define OS_TASK_STAT_EN          1      /* Enable (1) or Disable(0) the statistics task */
#define OS_TMR_EN                0      /* Enable (1) or Disable (0) code generation for TIMERS */
```

(3) start タスクを生成し、最初に実行できることを保証するために最高の優先度を設定します。

(4) タスクスケジュールを開始するために OSStart() を呼びます。

スタートタスクの手順は、以下のとおりです。

(1) スタートが最も高い優先度として、タスクスケジュールを起動した後に初めて実行します。スタートタスクの起動時に system tick タイマを初期化します。タイマの周波数は、Define によって OS_CFG.H (デフォルト: 50Hz) で設定できます。タイマのより高い周波数, 多くのローダーCPU が含まれます。

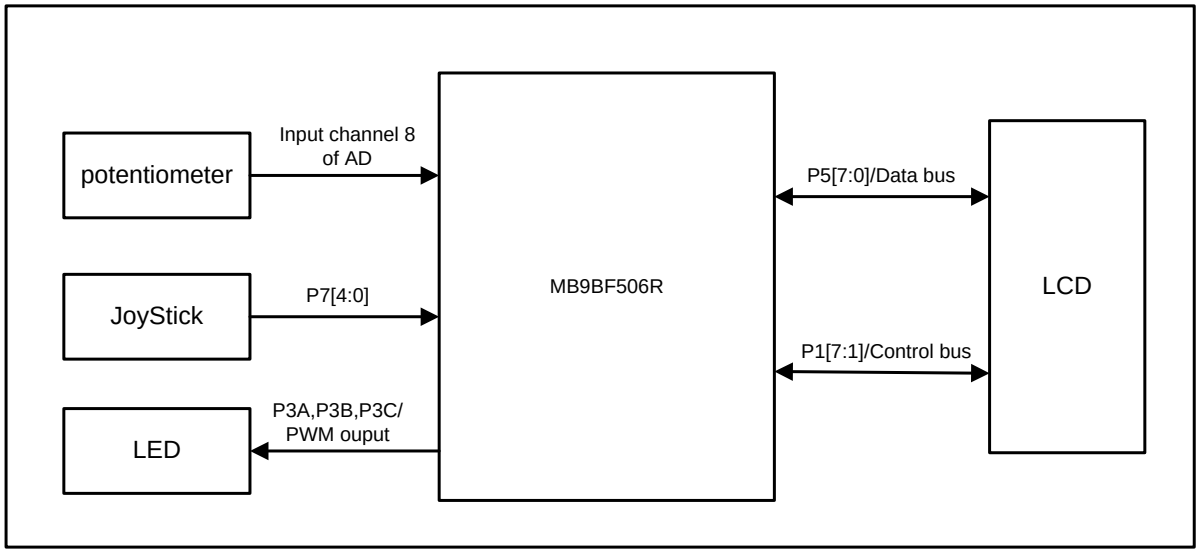
図 19. System Tick タイマ周波数設定

```
#define OS_TICKS_PER_SEC        20      /* Set the number of ticks in one second */
```

(2) スタートタスクを初期化します。これはアイドルタスク以外で実行中のタスクがない場合、アイドルタスクをエントリするために最大カウンタを取得するために使用します。このカウントは CPU 使用量を計算するために使用します。

(3) キー, AD, ベースタイマ, LCD などを含む MCU ペリフェラルを初期化します。

図 20. MCU IO 割当て



(4) OSTaskCreateExt または OSTaskCreate 関数によりユーザタスクを生成します。ユーザタスクはキータスク, AD タスク, モニタタスクから成ります。

(5) ここに至るまでにスタートタスクの仕事は終わります。またユーザタスクは OSTaskDel 関数で削除された後、スケジュールを開始します。

4.3 画面表示

コードが実行された後、図 21 のとおり画面はジョイスティックにより切り替えられます。

図 21. 画面表示

最初の画面ではシステム基本情報を表示しています。AD 値はポテンショメータ調整中に変更可能です。

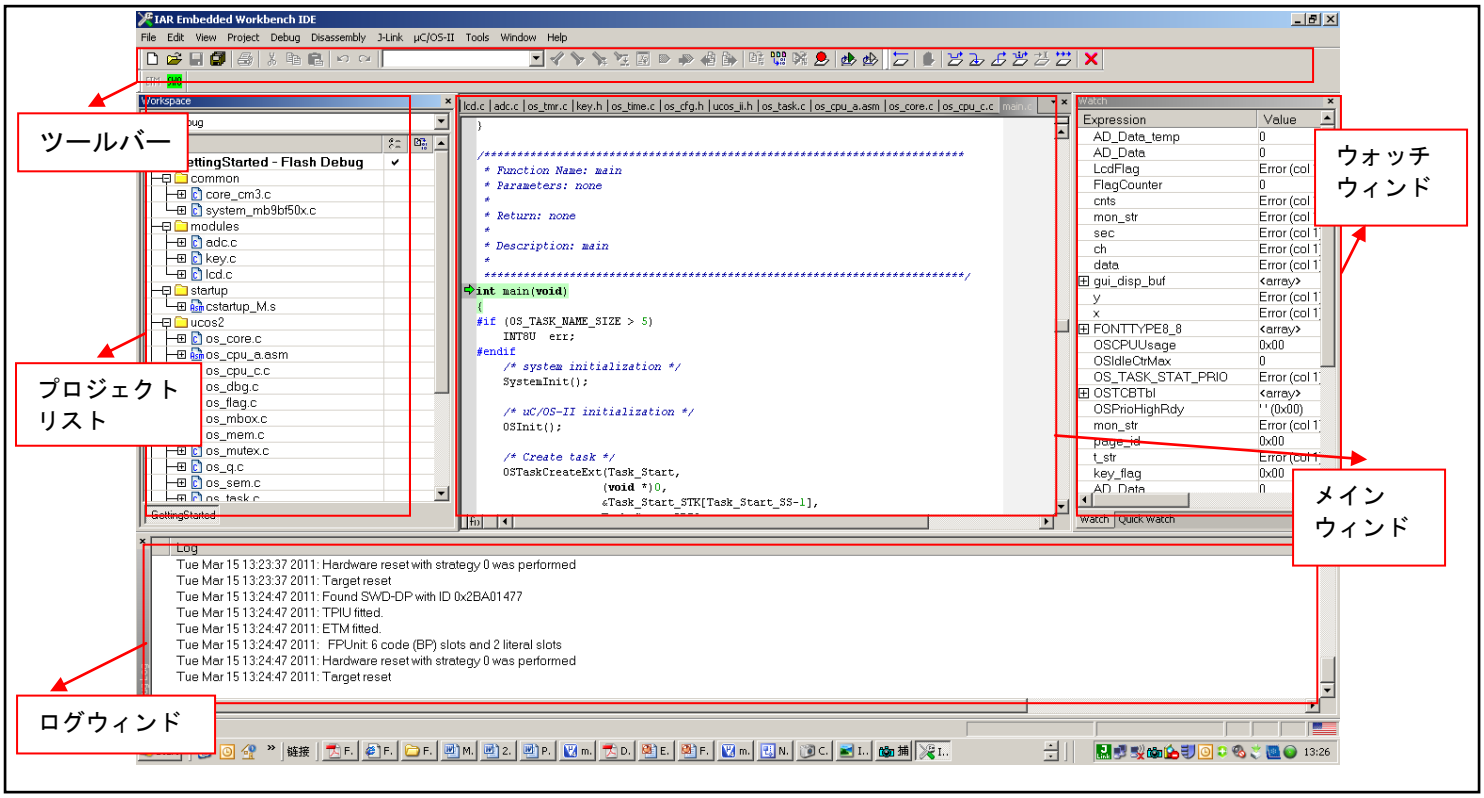
次の画面ではコード実行中にタスク名や優先度を表示しています。

3 番目は各タスクのスタック使用量を表示しています。

4.4 IAR EWARM Workbench でのデバッグ

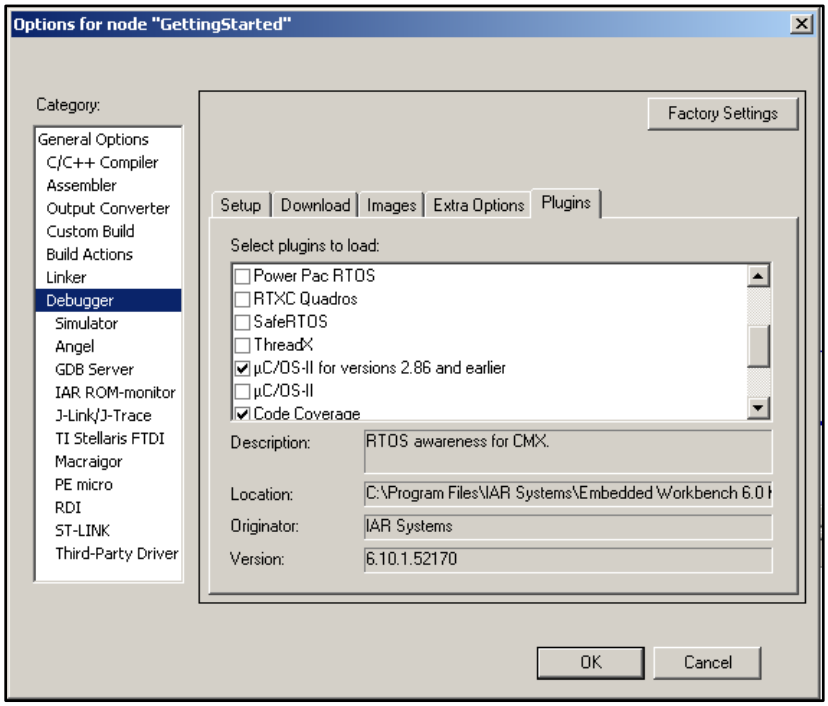
demo アプリケーションコードはオンボードの M-Link または J-Link と接続した IAR EWARM Workbench でデバッグできます。図 22 は基本的なデバッグウィンドを示します。

図 22. IAR EWARM Workbench デバッグウィンド



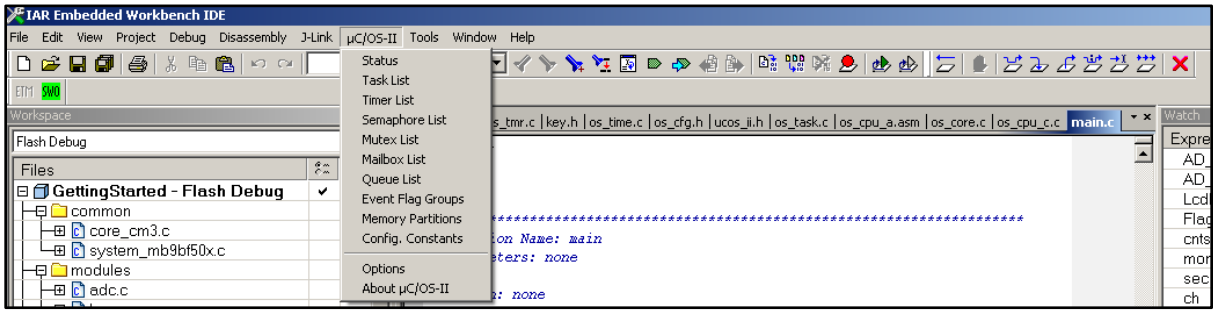
IAR C-Spy デバッガにおいて、OS タスクリストや実行状態を監視できる µC/OS-II kernel plug-in があります。この plug-in は図 23 の通り、project オプション設定の debugger 内で有効にできます。

図 23. µC/OS-II Plug-in



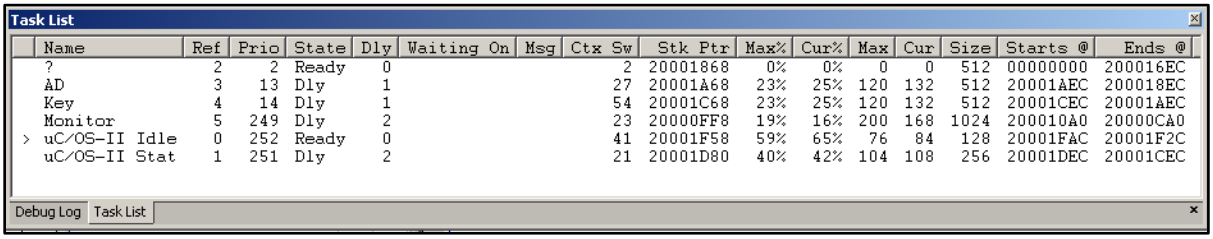
この plug-in が有効になった後、µC/OS-II メニューはデバッグウィンドに追加され、タスクリスト, タイマリスト, セマフォリストなど多くの情報をこのウィンドで監視できます。

図 24. µC/OS-II メニュー



demo アプリケーションコードにおけるこのタスクリストは図 25 で表示されます。

図 25. タスクリスト



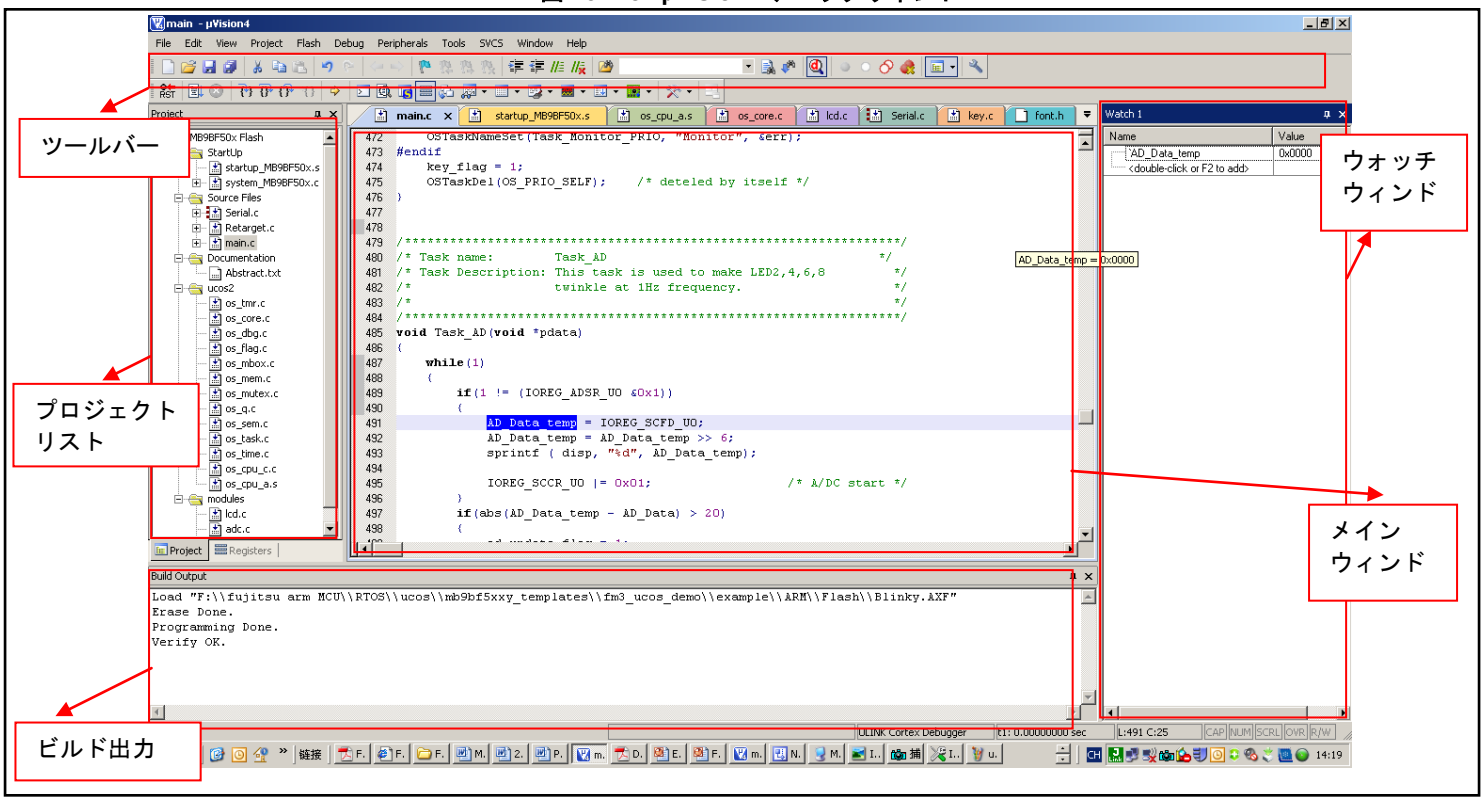
＜注意事項＞

- 本 IAR アプリケーションコードは IAR EWARM Workbench V6.21 で開発されています。もし他のバージョンを使用する場合、いくつかのプロジェクト設定は無効になり、ユーザは MCU タイプや include ファイルなどの設定をリセットする必要があります。
- µC/OS-II plug-in が使用された場合、OS_DBG.C ファイルは本プロジェクトでインクルードされるべきです。また OS_CFG.H 内の定義 “OS_DEBUG_EN” は有効にする必要があります。
- µC/OS-II V2.86 は本アプリケーションコード内で使用されます。

4.5 Keil µVision4 でのデバッグ

本 demo アプリケーションコードは U-Link と接続した Keil µVision4 でもデバッグできます。図 26 は基本的なデバッグウィンドを示します。

図 26. Keil µVision4 デバッグウィンド



＜注意事項＞

- 本 Keil 用アプリケーションコードは Kei MDK V4.21 で開発されています。もし他のバージョンを使用する場合、いくつかのプロジェクト設定は無効になり、ユーザは MCU タイプや include ファイルなどの設定をリセットする必要があります。

5 FM4 MCU における µC/OS-II 移植

S6E2GM シリーズは、ハイパフォーマンスかつ競争的な価格という特徴を持った、組込み用 32-bit マイクロコントローラです。

本シリーズはオンチップ型フラッシュメモリや SRAM を搭載した ARM Cortex-M4F processor に基づいており、モニタ制御タイマや A/D コンバータ，通信インタフェース (USB, CAN, UART, CSIO (SPI), I2C, LIN) のようなペリフェラルを持っています。

本デバイスにおける µC/OS-II 移植手順は以下の項目を除き、chapter 3 で紹介した FM3 の場合と同じです。

MCU のコア周波数は OS_CPU_C.C ファイル内のこのデバイスに従って設定される必要があります。

図 27. OS_CPU_SysTickInit 関数

```
void  OS_CPU_SysTickInit (void)
{
            INT32U  cnts;

            cnts = 180000000 / OS_TICKS_PER_SEC; (1)

            OS_CPU_CM3_NVIC_ST_RELOAD = (cnts - 1);       /* Enable timer. */
            OS_CPU_CM3_NVIC_ST_CTRL                       |=              OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC              |
OS_CPU_CM3_NVIC_ST_CTRL_ENABLE;             (2)         /* Enable timer interrupt. */
            OS_CPU_CM3_NVIC_ST_CTRL  |= OS_CPU_CM3_NVIC_ST_CTRL_INTEN;          (3)
}
```

6 FM4 MCU における µC/OS-II アプリケーション

µC/OS-II 移植用 demo コードは提供されません。以下は 3 LED タスクを作る、単純なアプリケーションのサンプルコードです。各タスクは OS がスケジューリングを開始した後、異なる周波数で LED を点滅します。

メイン関数: スタートタスクを生成します

図 28. FM4 µC/OS-II サンプルコードのメイン関数

```
int main(void)
{
#if (OS_TASK_NAME_SIZE > 5)
        INT8U  err;
#endif

        /* uC/OS-II initialization */
        OSInit();

        /* Create task */
        OSTaskCreateExt(Task_Start,
                 (void *)0,
                &Task_Start_STK[Task_Start_SS-1],
                Task_Start_PRIO,
                Task_Start_PRIO,
                &Task_Start_STK[0],
                Task_Start_SS, (void *)0,
                OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR);

#if (OS_TASK_NAME_SIZE > 5)
        OSTaskNameSet(Task_Start_PRIO, "Start", &err);
#endif

        /* uC/OS-II starts to schdule*/
        OSStart();

        /*Main Loop*/
        while(1);
}
```

スタートタスク: 3 LED タスクを生成します

図 29. FM4 µC/OS-II サンプルコードのスタートタスク

```
void Task_Start(void *pdata)
{
#if (OS_TASK_NAME_SIZE > 5)
        INT8U  err;
#endif
        OS_CPU_SysTickInit();                   /* initial systemtick timer */
        OSStatInit();                           /* start stat task  */
        /* Init LEDs*/
        InitLEDs();

        OSTaskCreateExt(Task_LED1, (void *)0,
                &Task_LED1_STK[Task_LED1_SS-1],
                Task_LED1_PRIO,
                Task_LED1_PRIO,
                &Task_LED1_STK[0],
                Task_LED1_SS,
                (void *)0,
                OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR);

#if (OS_TASK_NAME_SIZE > 5)
        OSTaskNameSet(Task_LED1_PRIO, "LED1", &err);
#endif

        OSTaskCreateExt(Task_LED2, (void *)0,
                &Task_LED2_STK[Task_LED2_SS-1],
                Task_LED2_PRIO,
                Task_LED2_PRIO,
                &Task_LED2_STK[0],
                Task_LED2_SS,
                (void *)0,
                OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR);
#if (OS_TASK_NAME_SIZE > 5)
        OSTaskNameSet(Task_LED2_PRIO, "LED2", &err);
#endif

        OSTaskCreateExt(Task_LED3, (void *)0,
                &Task_LED3_STK[Task_LED3_SS-1],
                Task_LED3_PRIO,
                Task_LED3_PRIO,
                &Task_LED3_STK[0],
                Task_LED3_SS,
                 (void *)0,
                OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR);
#if (OS_TASK_NAME_SIZE > 5)
        OSTaskNameSet(Task_LED3_PRIO, "LED3", &err);
#endif

        OSTaskDel(OS_PRIO_SELF);    /* deteled by itself */
}
```

LED1 タスク: 500ms ごとに LED1 をポーリングします

図 30. FM4 µC/OS-II サンプルコードの LED1 タスク

```
void Task_LED1(void *pdata)
{
        while(1)
        {
                LED1_ON();
                OSTimeDly(500);      /* Task delay 500ms */
                LED1_OFF();
                OSTimeDly(500);      /* Task delay 500ms */
        }
}
```

LED2 タスク: 1000ms ごとに LED2 をポーリングします

図 31. FM4 µC/OS-II サンプルコードの LED2 タスク

```
void Task_LED2(void *pdata)
{
        while(1)
        {
                LED2_ON();
                OSTimeDly(1000);     /* Task delay 1000ms */
                LED2_OFF();
                OSTimeDly(1000);     /* Task delay 1000ms */
        }
}
```

LED3 タスク: 1500ms ごとに LED3 をポーリングします

図 32. FM4 µC/OS-II サンプルコードの LED3 タスク

```
void Task_LED3(void *pdata)
{
        while(1)
        {
                LED3_ON();
                OSTimeDly(1500);     /* Task delay 1500ms */
                LED3_OFF();
                OSTimeDly(1500);     /* Task delay 1500ms */
        }
}
```

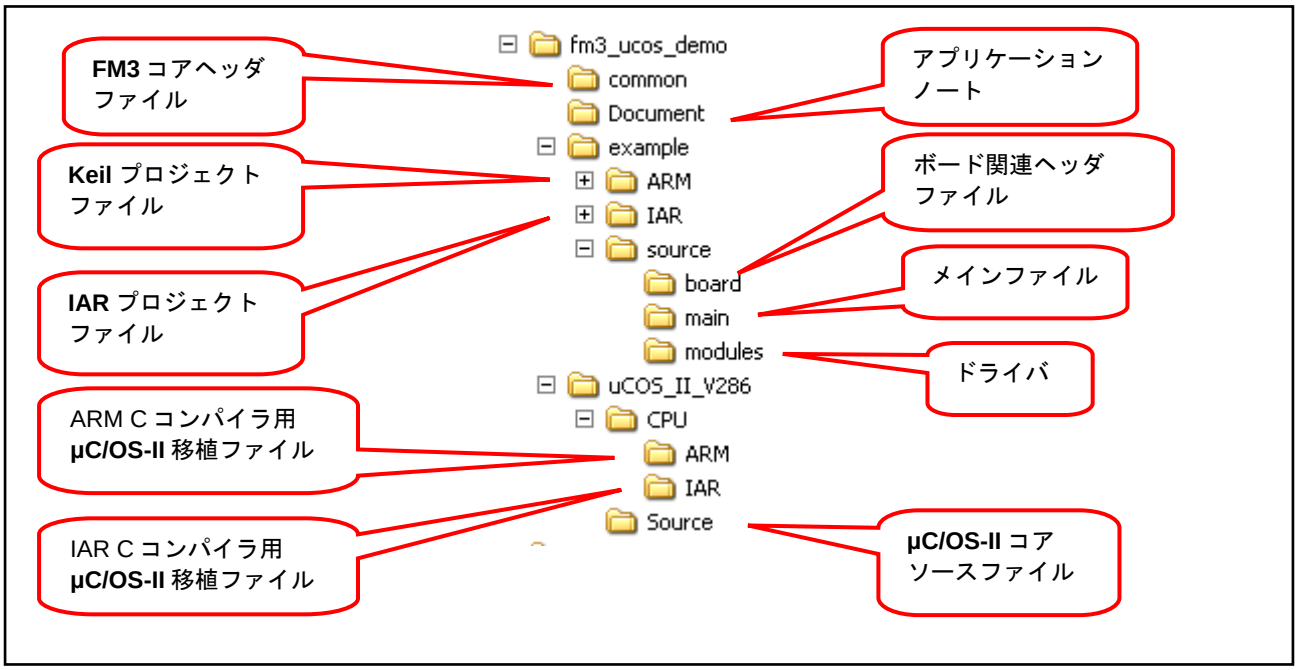
7 デリバリ

FM µC/OS-II デモプロジェクトは以下を含みます。

- FM3 µC/OS-II デモプロジェクト
- FM MCU における µC/OS-II 移植方法を記載したアプリケーションノート

図 33 にプロジェクトツリーリストを表示します。

図 33. µC/OS-II デモコードプロジェクトツリーリスト



8 改訂履歴

文書名: AN205234 - FM MCU における µC/OS-II 移植とアプリケーション

文書番号: 002-05234

版	ECN 番号	変更者	発行日	変更内容
**	-	NNAK	09/11/2015	サイプレスとして Spansion アプリケーションノート MCU-AN-510004-J-12 をドキュメントコード 002-05234 に登録しました。
*A	5646431	NNAK	03/01/2017	最新のテンプレートへ更新しました。

セールス, ソリューションおよび法律情報

ワールドワイドな販売と設計サポート

サイプレスは、事業所、ソリューション センター、メーカー代理店、および販売代理店の世界的なネットワークを保持しています。お客様の最寄りのオフィスについては、サイプレスのロケーション ページをご覧ください。

製品

ARM® Cortex® Microcontrollers	cypress.com/arm
車載用	cypress.com/automotive
クロック&バッファ	cypress.com/clocks
インターフェース	cypress.com/interface
IoT (モノのインターネット)	cypress.com/iot
メモリ	cypress.com/memory
マイクロコントローラ	cypress.com/mcu
PSoC	cypress.com/psoc
電源用 IC	cypress.com/pmic
タッチ センシング	cypress.com/touch
USB コントローラー	cypress.com/usb
ワイヤレス／RF	cypress.com/wireless

PSoC® ソリューション

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

サイプレス開発者コミュニティ

フォーラム | WICED IOT Forums | Projects | ビデオ | ブログ | トレーニング | Components

テクニカルサポート

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2015-2017. 本書面は、Cypress Semiconductor Corporation 及び Spansion LLC を含むその子会社（以下、「Cypress」という。）に帰属する財産である。本書面（本書面に含まれ又は言及されているあらゆるソフトウェア又はファームウェア（以下、「本ソフトウェア」という。）を含む）は、アメリカ合衆国及び世界のその他の国における知的財産法令及び条約に基づき、Cypress が所有する。Cypress はこれらの法令及び条約に基づく全ての権利を留保し、また、本段落で特に記載されているものを除き、Cypress の特許権、著作権、商標権又はその他の知的財産権のライセンスを一切許諾していない。本ソフトウェアにライセンス契約書が伴っておらず、かつ、あなたが Cypress との間で別途本ソフトウェアの使用方法を定める書面による合意をしていない場合、Cypress は、あなたに対して、（1）本ソフトウェアの著作権に基づき、（a）ソースコード形式で提供されている本ソフトウェアについて、Cypress ハードウェア製品と共に用いるためにのみ、組織内部でのみ、本ソフトウェアの修正及び複製を行うこと、並びに（b）Cypress のハードウェア製品ユニットに用いるためにのみ、（直接又は再販売者及び販売代理店を介して間接のいずれかで）エンドユーザーに対して、バイナリーコード形式で本ソフトウェアを外部に配布すること、並びに（2）本ソフトウェア（Cypress により提供され、修正がなされていないもの）に抵触する Cypress の特許権のクレームに基づき、Cypress ハードウェア製品と共に用いるためにのみ、本ソフトウェアの作成、利用、配布及び輸入を行うことについての非独占的で譲渡不能な一身専属的ライセンス（サブライセンスの権利を除く）を付与する。本ソフトウェアのその他の使用、複製、修正、変換又はコンパイルを禁止する。

適用される法律により許される範囲内で、Cypress は、本書面又はいかなる本ソフトウェアに関しても、明示又は黙示をとわず、いかなる保証（商品性及び特定の目的への適合性の黙示の保証を含むがこれらに限られない）も行わない。適用される法律により許される範囲内で、Cypress は、別途通知することなく、本書面を変更する権利を留保する。Cypress は、本書面に記載のあるいかなる製品又は回路の適用又は使用から生じる一切の責任を負わない。本書面で提供されたあらゆる情報（あらゆるサンプルデザイン情報又はプログラムコードを含む）は、参照目的のためのみに提供されたものである。この情報で構成するあらゆるアプリケーション及びその結果としてのあらゆる製品の機能性及び安全性を適切に設計し、プログラムし、かつテストすることは、本書面のユーザーの責任において行われるものとする。Cypress 製品は、兵器、兵器システム、原子力施設、生命維持装置若しくは生命維持システム、蘇生用の設備及び外科的移植を含むその他の医療機器若しくは医療システム、汚染管理若しくは有害物質管理の運用のために設計され若しくは意図されたシステムの重要な構成部分として用いるため、又はシステムの不具合が人身傷害、死亡若しくは物的損害を生じさせることになるその他の使用（以下、「本目的外使用」という。）のためには、設計、意図又は承認されていない。重要な構成部分とは、装置又はシステムのその構成部分の不具合が、その装置若しくはシステムの不具合を生じさせるか又はその安全性若しくは実効性に影響すると合理的に予想できる、機器又はシステムのあらゆる構成部分をいう。Cypress 製品のあらゆる本目的外使用から生じ、若しくは本目的外使用に関連するいかなる請求、損害又はその他の責任についても、Cypress はその全部又は一部をとわず一切の責任を負わず、かつ、あなたは Cypress をそれら一切から免除するものとし、本書により免除する。あなたは、Cypress 製品の本目的外使用から生じ又は本目的外使用に関連するあらゆる請求、費用、損害及びその他の責任（人身傷害又は死亡に基づく請求を含む）から Cypress を免責補償する。

Cypress、Cypress のロゴ、Spansion、Spansion のロゴ及びこれらの組み合わせ、WICED、PSoC、CapsSense、EZ-USB、F-RAM、及び Traveo は、米国及びその他の国における Cypress の商標又は登録商標である。Cypress の商標のより完全なリストは、cypress.com を参照のこと。その他の名称及びブランドは、それぞれの権利者の財産として権利主張がなされている可能性がある。