INTERFACE CO., LTD.

ITF 共通ライブラリ

初版　　2004 年　7 月　20 日
第 11 版　2009 年　11 月　24 日

User's Manual

インターフェイス株式会社

ご注意

1．本製品及び本書はインターフェイス株式会社の著作物です。
したがって、本製品及び本書の一部または全部を無断で複製、複写、転載、改変することは法律で禁じられています。

2．本製品及び本書の内容については、改良のために予告なく変更することがございます。

3．本製品を運用した結果の他への影響については、責任を負いかねますのでご了承ください。

4．本製品は、外国為替及び外国貿易法の規定により戦略物資等輸出規制品に該当する場合があります。
国外に持ち出す場合には、日本国政府の輸出許可申請などの手続きが必要となる場合があります。

5．本製品は、医療機器、航空宇宙機器など人命にかかわる設備や機器での使用は考慮しておりません。
これらの設備や機器に本製品を使用され、本製品の故障により、人身事故、火災などが発生しても、弊社では責任を負いかねます。

6．本製品は国内仕様です。本製品を国外で使用された場合の不具合などについては対応できませんのであらかじめご了承ください。

【目次】

1 はじめに

このたびは ITF ミドルウェアシリーズをお買い上げいただきまことにありがとうございます。
ITF ミドルウェアシリーズでは、ミドルウェアの共通部をひとつのライブラリとして提供しています。この共通部のライブラリを ITF 共通ライブラリと呼びます。
ITF 共通ライブラリでは、OS のサービスコールの抽象化、ITF ミドルウェアへのデバッグ機能の提供等を行っています。

ご使用になる前に、本マニュアルの内容をよくご理解いただき有効にご利用下さい。

2 本書について

本マニュアルは、本ライブラリの概要および取り扱いについて述べたものです。
本書は以下の章から構成されています。

なお，本書の内容は予告なしに変更することがあります。本書の内容について，ご不明な点がありましたら当社へお問合せください。

3 製品内容

3.1 概要

ITF ミドルウェアシリーズは、共通項目をひとつのライブラリとして提供しています。それは、ＯＳ依存部、プロトタイプの定義等が含まれます。

3.2 ディレクトリ構成

本ライブラリのディレクトリ構成は次の通りとなります。

【DOC】	各種マニュアル
【SRC】	ソース　ルート
+--【ITFLib】	ITF 共通ライブラリ　ルート
+--【Include】	共通インクルード
+--【Library】	ライブラリ
+--【Object】	オブジェクト
+--【Sample】	サンプル
+--【Application】	アプリケーション
+--【Driver】	ドライバ
+--【Source】	ソース
+--【Debug】	デバッグ部
+--【Depend】	依存部
+--【Io】	API
+--【Os】	OS 依存部

※　本ライブラリを使用する場合は、使用する側でインクルードパスに【ITFLib¥Include】を設定する必要があります。

3.3 ファイル構成

次は本ライブラリのファイル一覧です。

（表　ITF 共通ライブラリ　ファイル一覧）

ファイル名	説明
SRC¥ITFLib	
ITFLib_MS7727.hwp	HEW プロジェクトファイル
ITFLib_MS7751.hwp	HEW プロジェクトファイル
ITFLib_MS7760.hwp	HEW プロジェクトファイル
ITFLib_MS7751_CAB60842.hwp	HEW プロジェクトファイル
ITFLib_MS7751_M3A0040.hwp	HEW プロジェクトファイル
ITFLib_ARM7TDMIS.ewp	EWARM プロジェクトファイル
ITFLib_ARM7TDMIS.ewd	EWARM プロジェクトファイル
ITFLib_ARM7TDMIS.dep	EWARM プロジェクトファイル
SRC¥ITFLib¥Source¥Debug	
itfdebug.c	デバッグ関数
itferror.c	エラー関数
SRC¥ITFLib¥Source¥Os	
itfos.c	OS サービスコール抽象化関数
SRC¥ITFLib¥Source¥Io	
itflibapi.c	API
SRC¥ITFLib¥Include	
itfconfig.h	コンフィグレーションファイル
itfsh7727config.h	SH7727 固有のコンフィグレーションファイル
itfsh7760config.h	SH7760 固有のコンフィグレーションファイル
itfml60842config.h	ML60842 固有のコンフィグレーションファイル
itflpc24xxconfig.h	LPC2478 固有のコンフィグレーションファイル
itfdebug.h	デバッグ宣言
itferror.h	エラーコード宣言
itflib.h	ITF 共通ライブラリ宣言
itfos.h	OS コール宣言
itftypes.h	プロトタイプ等宣言
itflib_ver.h	バージョン定義
SRC¥ITFLib¥Sample¥Driver	
pci.h	PCI 定義
sh7751r.h	SH7751/SH7751R レジスタアドレス定義
ml60842.h	ML60842 レジスタアドレス定義
dma.h	DMA ドライバ定義
sh7751r_pci.c	SH7751R 用 PCI サンプルドライバ

m1543c.c	M1543C 用ドライバ
sh7727_usb.c	SH7727 用 USB ドライバ
sh7760_usb.c	SH7760 用 USB ドライバ
ml60842_usb.c	ML60842 用 USB ドライバ
lpc24xx_usb.c	LPC2478 用 USB ドライバ
sh_dma.c	SH 用 DMA ドライバ

※　本ライブラリを使用する場合は、使用する側で itflib.h をインクルードする必要があります。

4 コンフィグレーション

本ライブラリでは、システムのコンフィグレーションを定義しています。システムの環境に合わせてコンフィグレーションすることにより、本ライブラリ、ITF ミドルウェアは適切に構築されます。
コンフィグレーションファイルは次のファイルとなります。

itfconfig.h

ここで決定されたコンフィグレーションは、各 ITF ミドルウェアに反映されます。（本ライブラリを使用しない ITF ミドルウェアには反映されません。）
また、コンフィグレーションを変更した場合には本ライブラリと ITF ミドルウェアを再構築（コンパイル）する必要があります。

4.1 CPU

使用する CPU を決定します。CPU に依存する処理があるミドルウェアを使用する場合は必ず定義してください。
使用する CPU により次のどれかを定義して下さい。

（表　CPU のコンフィグレーション）

CPU	定義名
SH7727	__CPU_SH7727__
SH7751/SH7751R	__CPU_SH7751__
SH7760	__CPU_SH7760__
他の CPU	__CPU_OTHER__

4.2 エンディアン

使用する CPU のエンディアンを決定します。エンディアンに依存する処理があるミドルウェアを使用する場合は必ず定義してください。
使用する CPU のエンディアンにより次のどちらかを定義して下さい。

（表　エンディアンのコンフィグレーション）

エンディアン	定義名
リトルエンディアン	__LITTLE_ENDIAN__
ビッグエンディアン	__BIG_ENDIAN__

4.3 オペレーティングシステム

使用する OS を決定します。ＯＳに依存する処理があるミドルウェアを使用する場合は必ず定義してください。
使用するＯＳにより次のどれかを定義して下さい。

（表　OS のコンフィグレーション）

OS 名	定義名
NORTiV4	__KERNEL_NORTI_V4__
Toppers/jsp	__KERNEL_TOPPERS__
μC3/Compact	__KERNEL_UC3_COMPACT__
OS なし	定義なし

4.4 OS のタイマー周期

ＯＳを使用する場合は、ＯＳのタイマー周期を決定します。ＯＳを使用する場合は必ず定義してください。単位はm秒となります。
定義名は次の通りです。

ITF_MSEC　　10

※OS のコンフィグレーションに uC3/Compact を選択した場合は、OS のチック時間に関わらず「1」を指定して下さい。

4.5 パラメータチェック

ITF ミドルウェアでパラメータをチェックするか否か決定します。
定義名は次の通りです。

__ITFPRMNOCHK__

上記を定義した場合は ITF ミドルウェアでパラメータのチェックを行いません。デフォルトでは定義されていません（パラメータのチェックを行います）。
パラメータをチェックしないことでコードを省き、プログラムサイズを小さくすることが出来ます。ただし、ミドルウェアが対応していない場合はこの限りではありません。

4.6 デバッグ

ITF ミドルウェアでデバッグの有効、または無効を決定します。
定義名は次の通りです。

__ITFDEBUG__

上記を定義することで本ライブラリは ITF ミドルウェアにデバッグ環境を提供します。本定義はデフォルトでは定義されていません。
本ライブラリが、ITF ミドルウェアに提供されるデバッグ機能については後述します。

4.7 SH7727 固有のコンフィグレーション

4.7.1 SH7727 固有のコンフィグレーションについて

CPU のコンフィグレーションに SH7727 を選択した場合に、SH7727 固有のコンフィグレーションが存在します。ターゲットの環境に合わせてコンフィグレーションを変更する必要があります。
コンフィグレーションファイルは、次のファイルとなります。

itfsh7727config.h

4.7.2 USB ファンクション

SH7727 内蔵 USB ファンクションコントローラを使用するか決定します。
定義名は次の通りです。

__SH7727_USB_FUNCTION__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。

4.7.3 USB ホスト

SH7727 内蔵 USB ホストコントローラを使用するか決定します。
定義名は次の通りです。

__SH7727_USB_HOST__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。

4.7.4 クロックソース選択

USB ファンクション、または USB ホストを使用する場合に、USB コントローラが使用するクロックの入力元を決定します。

（表　クロックソース選択のコンフィグレーション）

定義名	値	説明
__SH7727_USB_CLK__	0	周辺クロック
	1	内部クロック
	2	バスクロック
	3	外部クロック

ソースクロック選択に関する詳細は、「SH7727 ハードウェアマニュアル」を参照して下さい。

4.7.5 ドライバ選択

USB ファンクション、または USB ホストを使用する場合に、USB クロック生成のための分周比を決定します。

（表　ドライバ選択のコンフィグレーション）

定義名	値	説明
__SH7727_USB_DIV__	0	分周比　1／1
	1	分周比　1／2
	2	分周比　1／3

ドライバ選択に関する詳細は、「SH7727 ハードウェアマニュアル」を参照して下さい。
48MHz の USB クロックを生成するために、クロックソース選択とドライバ選択のコンフィグレーションを決定する必要があります。USB クロックの詳細は、「SH7727 ハードウェアマニュアル」を参照して下さい。

4.8 SH7760 固有のコンフィグレーション

4.8.1 SH7760 固有のコンフィグレーションについて

CPU のコンフィグレーションに SH7760 を選択した場合に、SH7760 固有のコンフィグレーショ

ンが存在します。ターゲットの環境に合わせてコンフィグレーションを変更する必要があります。
コンフィグレーションファイルは、次のファイルとなります。

itfsh7760config.h

4.8.2 USB ホスト

SH7760 内蔵 USB ホストコントローラを使用するか決定します。
定義名は次の通りです。

__SH7760_USB_HOST__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。

4.8.3 USB のアライメント方法

USB ホストを使用する場合に、データのアライメント方法を決定します。

（表　アライメント方法のコンフィグレーション）

定義名	値	説明
__SH7760_USB_ALIGNMENT__	0	アライメントなし
	1	バイト境界モード
	2	ロングワード／ワード境界モード

アライメント方法に関する詳細は、「SH7760 ハードウェアマニュアル」の「ダイレクトメモリコントローラ」の項を参照して下さい。
本ライブラリでは、データのアライメントの機能を使用することを前提としています。DMA を使用しない場合でもシステムに沿った正しい設定をして下さい。

4.8.4 USB の DMA

USB ホストを使用する場合に、SH7760 内蔵ダイレクトメモリコントローラを使用するか決定します。
定義名は次の通りです。

__SH7760_USB_DMA__

4.8.5 USB の DMA 割り込み

SH7760 内蔵ダイレクトメモリコントローラの割り込みを使用するか決定します。
定義名は次の通りです。

__SH7760_USB_DMA_IRQ__

4.9 ML60842 固有のコンフィグレーション

4.9.1 ML60842 固有のコンフィグレーションについて

ミドルウェアで ML60842(USB OTG コントローラ)を使用する場合に、ML60842 固有のコンフィグレーションが存在します。ターゲットの環境に合わせてコンフィグレーションを変更する必要があります。
コンフィグレーションファイルは、次のファイルとなります。

itfml60842config.h

4.9.2 USB ホスト

ML60842 内蔵 USB ホストコントローラを使用するか決定します。
定義名は次の通りです。

__ML60842_USB_HOST__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。

4.9.3 ML60842 のベースアドレス

ML60842 のベースアドレスを指定します。
定義名は次の通りです。

__ML60842_BASE_ADDRESS__ (0xA8000000)

4.9.4 ML60842 の割り込み番号

ML60842 の割り込み番号を指定します。
定義名は次の通りです。

__ML60842_IRQ__ (0x3A0/32)

4.9.5 USB ホストの DMA

USB ホストを使用する場合に、USB ホストの DMA を使用するか決定します。
定義名は次の通りです。

__ML60842_HOST_DMA__

4.9.6 USB ホストの DMA チャンネル

USB ホストの DMA を使用する場合に、使用する DMA のチャンネル番号を指定します。
定義名は次の通りです。

__ML60842_HOST_DMA_CH__ (0)

4.10 LPC24xx 固有のコンフィグレーション

4.10.1 LPC24xx 固有のコンフィグレーションについて

CPU のコンフィグレーションに他の CPU を選択し、LPC24xx を使用する場合に、LPC24xx 固有のコンフィグレーションが存在します。ターゲットの環境に合わせてコンフィグレーションを変更する必要があります。
コンフィグレーションファイルは、次のファイルとなります。

itflpc24xxconfig.h

4.10.2 USB ファンクション

LPC24xx 内蔵 USB ファンクションコントローラを使用するか決定します。
定義名は次の通りです。

__LPC24XX_USB_FUNCTION__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。

4.10.3 USB ホスト

LPC24xx 内蔵 USB ホストコントローラを使用するか決定します。
定義名は次の通りです。

__LPC24XX_USB_HOST__

上記を定義することで本ライブラリは後述する USB ドライバの機能を提供します。
USB ホストと USB ファンクションを同時に使用する場合、LPC24xx の USB RAM 領域を重複しないように割り当てる必要があります。USB RAM 領域の指定方法については、各ミドルウェアのマニュアルを参照して下さい。

4.10.4 入力クロック

USB ファンクション、または USB ホストを使用する場合に、USB クロックの入力となる PLL クロック周波数を決定します。単位は Hz となります。
定義名は次の通りです。

__LPC24XX_PLLCLK__ 288000000

4.10.5 割り込み優先レベル

USB ファンクション、または USB ホストを使用する場合に、USB 割り込みの割り込み優先レベルを決定します。
定義名は次の通りです。

__LPC24XX_USB_INT_PRIORITY__　　8

5 プロトタイプ

本ライブラリでは、ITF ミドルウェアシリーズで使用する共通のプロトタイプを定義しています。ITF ミドルウェアシリーズでは、次に定義されているプロトタイプを使用しています。

（表　プロトタイプ一覧）

名称	型	説明
int8_t	signed char	符号付汎用８ビットデータ型
int16_t	signed short	符号付汎用１６ビットデータ型
int32_t	signed long	符号付汎用３２ビットデータ型
uint8_t	unsigned char	符号無し汎用８ビットデータ型
uint16_t	unsigned short	符号無し汎用１６ビットデータ型
uint32_t	unsigned long	符号無し汎用３２ビットデータ型
ITER	long	エラーステータス
ISTATUS	int32_t	ステータス
istatus	ISTATUS	ステータス
IHANDLE	int32_t	ハンドル
ihandle	IHANDLE	ハンドル
IBOOL	int32_t	真偽値
ibool	IBOOL	真偽値
ISIZE_T	int32_t	サイズ
isize_t	ISIZE_T	サイズ
ITINT	int	int
UITINT	unsigned int	unsigned int
ITBOOL	long	真偽値
ITVP	void*	ポインタ
ITVP_INT	ITVP	ポインタ
ITVB	char	char
(*ITFP)()	void	ファイルポインタ
ITPRI	ITINT	優先度値
ITATR	UITINT	Attribute 値
ITID	ITINT	オブジェクト ID
ITER_ID	ITID	クリエイト時、正なら ID、負ならエラーコード
ITINHNO	UITINT	割込み番号
ITSIZE	unsigned long	サイズ
ITDLYTIME	ITW	休止時間
ITRELTIM	ITDLYTIME	relative time
LOCAL	static	ローカル関数／変数
PUBLIC		パブリック関数／変数

ITUINT	unsigned long	unsigned long
ITFN	ITINT	
ITRDVNO	ITINT	
ITRDVPTN	ITUINT	
ITMODE	ITUINT	
ITSTAT	ITUINT	
ITER_UINT	ITINT	
ITTEXPTN	ITUINT	
ITFLGPTN	ITUINT	
ITINTNO	ITUINT	
ITDLYTIME	ITOVRTIM	
ITIXXXX	ITUINT	
ITEXCNO	ITUINT	

6 OS サービスコール

本ライブラリでは、ITF ミドルウェアで使用する OS のサービスコールを用意しています。

通常、本ライブラリではサービスコールが使用された場合にμITRON4.0 準拠のシステムコールを呼び出しています。また、サービスコールのインターフェイスもμITRON4.0 準拠のシステムコールと同等のものを用意しています。

μITRON4.0 準拠のＯＳを使用しない場合は、このサービスコールの処理を変更する必要があります。また、ＯＳがμITRON4.0 準拠の場合でもＯＳにより動作が異なる場合がありますので、各ＯＳのサービスコールの動作を確認する必要があります。

次は本ライブラリが提供するＯＳサービスコールとμITRON4.0 準拠のシステムコールの対応です。

（表　OS サービスコール一覧）

サービスコール名	μITRON4.0 準拠システムコール	機能
タスク管理機能		
OSAPI_Cre_Tsk	cre_tsk	タスク生成.
OSAPI_Acre_Tsk	acre_tsk	タスク生成.
OSAPI_Del_Tsk	del_tsk	タスク削除.
OSAPI_Act_Tsk	act_tsk	タスク起動.
OSAPI_Iact_Tsk	iact_tsk	タスク起動.
OSAPI_Can_Act	can_act	タスク起動要求のキャンセル.
OSAPI_Sta_Tsk	sta_tsk	タスク起動.
OSAPI_Ext_Tsk	ext_tsk	自タスク終了.
OSAPI_Exd_Tsk	exd_tsk	自タスクの終了と削除.
OSAPI_Ter_Tsk	ter_tsk	他タスク強制終了.
OSAPI_Chg_Pri	chg_pri	タスクベース優先度変更.
OSAPI_Get_Pri	get_pri	タスク現在優先度参照.
OSAPI_Ref_Tsk	ref_tsk	タスク状態参照.
OSAPI_Ref_Tst	ref_tst	タスク状態参照.
タスク付属同期機能		
OSAPI_Slp_Tsk	slp_tsk	自タスクを起床待ち状態へ移行.
OSAPI_Tslp_Tsk	tslp_tsk	自タスクを起床待ち状態へ移行(タイムアウト有). ※1
OSAPI_Wup_Tsk	wup_tsk	他タスクの起床.
OSAPI_Iwup_Tsk	iwup_tsk	他タスクの起床.
OSAPI_Can_Wup	can_wup	タスクの起動要求を無効化.
OSAPI_Rel_Wai	rel_wai	待ち状態の強制解除.
OSAPI_Irel_Wai	irel_wai	待ち状態の強制解除.
OSAPI_Sus_Tsk	sus_tsk	タスクを強制待ち状態へ移行.
OSAPI_Rsm_Tsk	rsm_tsk	強制待ち状態のタスクを再開.

OSAPI_Frsm_Tsk	frsm_tsk	強制待ち状態のタスクを強制再開.
OSAPI_Dly_Tsk	dly_tsk	自タスク遅延. ※1
タスク例外処理機能		
OSAPI_Def_Tex	def_tex	タスク例外処理ルーチンの定義.
OSAPI_Ras_Tex	ras_tex	タスク例外処理要求.
OSAPI_Iras_Tex	iras_tex	タスク例外処理要求.
OSAPI_Dis_Tex	dis_tex	タスク例外処理禁止.
OSAPI_Ena_Tex	ena_tex	タスク例外処理許可.
OSAPI_Sns_Tex	sus_tex	自タスクのタスク例外処理禁止状態の参照.
OSAPI_Ref_Tex	ref_tex	タスク例外処理状態参照.
同期・通信機能　セマフォ		
OSAPI_Cre_Sem	cre_sem	セマフォ生成.
OSAPI_Acre_Sem	acre_sem	セマフォ生成（ID 自動割り当て）.
OSAPI_Del_Sem	del_sem	セマフォ削除.
OSAPI_Sig_Sem	sig_sem	セマフォ資源返却.
OSAPI_Isig_Sem	isig_sem	セマフォ資源返却.
OSAPI_Wai_Sem	wai_sem	セマフォ資源獲得.
OSAPI_Pol_Sem	pol_sem	セマフォ資源獲得（ポーリング）.
OSAPI_Twai_Sem	twai_sem	セマフォ資源獲得（タイムアウト有）. ※1
OSAPI_Ref_Sem	ref_sem	セマフォ状態参照.
同期・通信機能　イベントフラグ		
OSAPI_Cre_Flg	cre_flg	イベントフラグ生成.
OSAPI_Acre_Flg	acre_flg	イベントフラグ生成（ID 自動割り当て）.
OSAPI_Del_Flg	del_flg	イベントフラグ削除.
OSAPI_Set_Flg	set_flg	イベントフラグのセット.
OSAPI_Iset_Flg	iset_flg	イベントフラグのセット.
OSAPI_Clr_Flg	clr_flg	イベントフラグのクリア.
OSAPI_Wai_Flg	wai_flg	イベントフラグ待ち.
OSAPI_Pol_Flg	pol_flg	イベントフラグ待ち（ポーリング）.
OSAPI_Twai_Flg	twai_flg	イベントフラグ待ち（タイムアウト有）. ※1
OSAPI_Ref_Flg	ref_flg	イベントフラグ状態参照.
同期・通信機能　データキュー		
OSAPI_Cre_Dtq	cre_dtq	データキュー生成.
OSAPI_Acre_Dtq	acre_dtq	データキュー生成（ID 自動割り当て）.
OSAPI_Del_Dtq	del_dtq	データキュー削除.
OSAPI_Snd_Dtq	snd_dtq	データ送信.
OSAPI_Psnd_Dtq	psnd_dtq	データ送信（ ポーリング).
OSAPI_Ipsnd_Dtq	ipsnd_dtq	データ送信（ ポーリング).
OSAPI_Tsnd_Dtq	tsnd_dtq	データ送信.
OSAPI_Fsnd_Dtq	fsnd_dtq	強制データ送信.
OSAPI_Ifsnd_Dtq	ifsnd_dtq	強制データ送信.
OSAPI_Rcv_Dtq	rcv_dtq	データキューからの受信.
OSAPI_Prcv_Dtq	prcv_dtq	データキューからの受信（ポーリング）.

OSAPI_Trcv_Dtq	trcv_dtq	データキュー待ち（タイムアウト有）．※1
OSAPI_Ref_Dtq	ref_dtq	データキュー状態参照.
同期・通信機能　メールボックス		
OSAPI_Cre_Mbx	cre_mbx	メールボックス生成.
OSAPI_Acre_Mbx	acre_mbx	メールボックス生成.
OSAPI_Del_Mbx	del_mbx	メールボックス削除.
OSAPI_Snd_Mbx	snd_mbx	メールボックスへ送信.
OSAPI_Rcv_Mbx	rcv_mbx	メールボックスから受信.
OSAPI_Prcv_Mbx	prcv_mbx	メールボックスから受信（ポーリング）.
OSAPI_Trcv_Mbx	trcv_mbx	メールボックスから受信（タイムアウトあり）．※1
OSAPI_Ref_Mbx	ref_mbx	メールボックス状態参照.
拡張同期・通信機能　ミューテックス		
OSAPI_Cre_Mtx	cre_mtx	ミューテックス生成.
OSAPI_Acre_Mtx	acre_mtx	ミューテックス生成（ID 自動割り当て）.
OSAPI_Del_Mtx	del_mtx	ミューテックス削除.
OSAPI_Loc_Mtx	loc_mtx	ミューテックス資源獲得.
OSAPI_Ploc_Mtx	ploc_mtx	ミューテックス資源獲得（ポーリング）.
OSAPI_Tloc_Mtx	tloc_mtx	ミューテックス資源獲得（タイムアウト有）．※1
OSAPI_Unl_Mtx	unl_mtx	ミューテックスロック解除.
OSAPI_Ref_Mtx	ref_mtx	ミューテックス状態参照.
拡張同期・通信機能　メッセージバッファ		
OSAPI_Cre_Mbf	cre_mbx	メッセージバッファ生成.
OSAPI_Acre_Mbf	acre_mbx	メッセージバッファ生成（ID 自動割り当て）.
OSAPI_Del_Mbf	del_mbx	メッセージバッファ削除.
OSAPI_Snd_Mbf	snd_mbx	メッセージバッファへ送信.
OSAPI_Psnd_Mbf	rcv_mbx	メッセージバッファへ送信（ポーリング）.
OSAPI_Tsnd_Mbf	prcv_mbx	メッセージバッファへ送信（タイムアウト有）．※1
OSAPI_Rcv_Mbf	trcv_mbx	メッセージバッファから受信.
OSAPI_Prcv_Mbf	ref_mbx	メッセージバッファから受信（ポーリング）.
OSAPI_Trcv_Mbf	cre_mbx	メッセージバッファから受信（タイムアウト有）．※1
OSAPI_Ref_Mbf	acre_mbx	メッセージバッファ状態参照.
拡張同期・通信機能　ランデブ		
OSAPI_Cre_Por	cre_por	ランデブ用ポート生成.
OSAPI_Acre_Por	acre_por	ランデブ用のポート生成（ID 自動割り当て）.
OSAPI_Del_Por	del_por	ランデブ用のポート削除.
OSAPI_Cal_Por	cal_por	ポートに対するランデブの呼出.
OSAPI_Tcal_Por	tcal_por	ポートに対するランデブの呼出（タイムアウト有）．※1
OSAPI_Acp_Por	acp_por	ポートに対するランデブ受付.
OSAPI_Pacp_Por	pacp_por	ポートに対するランデブ受付（ポーリング）.
OSAPI_Tacp_Por	tacp_por	ポートに対するランデブ受付（タイムアウト有）．※1
OSAPI_Fwd_Por	fwd_por	ポートに対するランデブ回送.
OSAPI_Rpl_Rdv	rpl_rdv	ランデブ返答.

OSAPI_Ref_Por	ref_por	ポート状態参照.
メモリプール管理機能　固定長メモリプール		
OSAPI_Cre_Mpf	cre_mpf	固定長メモリプール生成.
OSAPI_Acre_Mpf	acre_mpf	固定長メモリプール生成.
OSAPI_Del_Mpf	del_mpf	固定長メモリプール削除.
OSAPI_Get_Mpf	get_mpf	固定長メモリブロック獲得.
OSAPI_Pget_Mpf	pget_mpf	固定長メモリブロック獲得（ポーリング）.
OSAPI_Tget_Mpf ※1	tget_mpf	固定長メモリブロック獲得（タイムアウト有）. ※1
OSAPI_Rel_Mpf	rel_mpf	固定長メモリブロックの返却.
OSAPI_Ref_Mpf	ref_mpf	固定長メモリプール状態参照.
メモリプール管理機能　可変長メモリプール		
OSAPI_Cre_Mpl	cre_mpl	可変長メモリプール生成.
OSAPI_Acre_Mpl	acre_mpl	可変長メモリプール生成（ID 自動割り当て）.
OSAPI_Del_Mpl	del_mpl	可変長メモリプール削除.
OSAPI_Get_Mpl	get_mpl	可変長メモリブロック獲得.
OSAPI_Pget_Mpl	pget_mpl	可変長メモリブロック獲得（ポーリング）.
OSAPI_Tget_Mpl	tget_mpl	可変長メモリブロック獲得（タイムアウト有）. ※1
OSAPI_Rel_Mpl	rel_mpl	可変長メモリブロック返却.
OSAPI_Ref_Mpl	ref_mpl	可変長メモリプール状態参照.
時間管理機能　システム時刻管理		
OSAPI_Set_Tim	set_tim	システム時刻設定.
OSAPI_Get_Tim	get_tim	システム時刻参照.
OSAPI_Isig_Tim	isig_tim	チックタイムの経過通知.
時間管理機能　周期ハンドラ		
OSAPI_Cre_Cyc	cre_cyc	周期ハンドラ生成.
OSAPI_Acre_Cyc	acre_cyc	周期ハンドラ生成（番号自動割り当て）.
OSAPI_Del_Cyc	del_cyc	周期ハンドラ削除.
OSAPI_Sta_Cyc	sta_cyc	周期ハンドラ動作開始.
OSAPI_Stp_Cyc	stp_cyc	周期ハンドラ動作停止.
OSAPI_Ref_Cyc	ref_cyc	周期ハンドラ状態参照.
時間管理機能　アラームハンドラ		
OSAPI_Cre_Alm	cre_alm	アラームハンドラ生成.
OSAPI_Acre_Alm	acre_alm	アラームハンドラ生成（番号自動割り当て）.
OSAPI_Del_Alm	del_alm	アラームハンドラの削除.
OSAPI_Sta_Alm	sta_alm	アラームハンドラ動作開始.
OSAPI_Stp_Alm	stp_alm	アラームハンドラ動作停止.
OSAPI_Ref_Alm	ref_alm	アラームハンドラ状態参照.
時間管理機能　オーバーランハンドラ		
OSAPI_Def_Ovr	def_ovr	オーバーランハンドラの定義.
OSAPI_Sta_Ovr	sta_ovr	オーバーランハンドラの動作開始. ※1

OSAPI_Stp_Ovr	stp_ovr	オーバーランハンドラの動作停止.
OSAPI_Ref_Ovr	ref_ovr	オーバーランハンドラ状態参照.
システム状態管理機能		
OSAPI_Rot_Rdq	rot_rdq	タスクのレディキュー回転.
OSAPI_Irot_Rdq	irot_rdq	タスクのレディキュー回転.
OSAPI_Get_Tid	get_tid	実行状態のタスクＩＤの参照.
OSAPI_Iget_Tid	iget_tid	実行状態のタスクＩＤの参照.
OSAPI_Loc_Cpu	loc_cpu	CPU ロック状態への移行(割込みとディスパッチの禁止).
OSAPI_Iloc_Cpu	iloc_cpu	CPU ロック状態への移行(割込みとディスパッチの禁止).
OSAPI_Unl_Cpu	unl_cpu	CPU ロック状態の解除.
OSAPI_Iunl_Cpu	iunl_cpu	CPU ロック状態の解除.
OSAPI_Dis_Dsp	dis_dsp	ディスパッチ禁止.
OSAPI_Ena_Dsp	ena_dsp	ディスパッチ許可.
OSAPI_Sns_Ctx	sns_ctx	コンテキスト参照.
OSAPI_Sns_Loc	sns_loc	CPU ロック状態参照.
OSAPI_Sns_Dsp	sns_dsp	ディスパッチ禁止状態参照.
OSAPI_Sns_Dpn	sns_dpn	ディスパッチ保留状態参照.
OSAPI_Ref_Sys	ref_sys	システム状態参照.
割り込み管理機能		
OSAPI_Def_Inh	def_inh	割込みハンドラ定義.
OSAPI_Cre_Isr	cre_isr	割込みサービスルーチンの生成.
OSAPI_Acre_Isr	acre_isr	割込みサービスルーチンの生成（ID 自動割り当て）.
OSAPI_Del_Isr	del_isr	割込みサービスルーチンの削除.
OSAPI_Ref_Isr	ref_isr	割込みサービスルーチンの状態参照.
OSAPI_Dis_Int	dis_int	割り込みの禁止.
OSAPI_Ena_Int	ena_int	割り込みの許可.
OSAPI_Chg_Ixx	chg_ixx	割り込みマスクの変更.
OSAPI_Get_Ixx	get_ixx	割り込みマスクの参照.
サービスコール管理機能		
OSAPI_Def_Svc	def_svc	拡張サービスコールの定義.
OSAPI_Cal_Svc	cal_svc	サービスコールの呼出.
システム構成管理機能		
OSAPI_Def_Exc	def_exc	CPU 例外ハンドラの定義.
OSAPI_Ref_Cfg	ref_cfg	コンフィグレーション情報参照.
OSAPI_Ref_Ver	ref_ver	バージョン参照.

※ 1　サービスコールで使用するタイムアウト時間の単位はm秒で扱っています。本ライブラリのＯＳサービスコールの中で、タイムアウト時間の単位をＯＳの時間単位に変換する必要があります。

7 エラーコード

本ライブラリで使用するエラーコードは、次のファイルに定義されています。

itferror.h

8 標準関数

本ライブラリは、ITF ミドルウェアが標準関数を使用するためのインターフェイスを提供しています。

本ライブラリは標準関数をサポートする標準ライブラリを使用することを前提としています。標準関数は標準ライブラリによって提供されます。アプリケーションを構築する時に、標準ライブラリを組み込む必要があります。ライブラリの組み込みについては、OS 等のマニュアルを参照して下さい。

本ライブラリでは、定義ファイルをインクルードしています。

（表　インクルードしている定義ファイル）

定義ファイル名
stdio.h
stdlib.h
string.h

これらの定義ファイルは、”itflib.h”でインクルードしています。標準関数の実装をユーザ独自に行う場合は、”itflib.h”でインクルードされている定義ファイルを削除し、必要な定義を追加して下さい。

9 API 仕様

ITF 共通ライブラリの API について解説します。

9.1 情報取得関数

ITFLib_Version

機　能　製品バージョン番号読み出し

形　式　ISTATUS ITFLib_Version(int16_t *major,
int16_t *minor
);
major　：メジャー番号を格納する領域
minor　：マイナー番号を格納する領域

解　説　ITF 共通ライブラリの製品バージョン番号を返します。
例えば「Ver 1.02」の場合は、major = 1、minor ＝2 を返します。

戻　値　0　　：正常終了
!0　　：異常終了

例　　ITF 共通ライブラリの製品バージョン番号を読み出します。

```
void task( void )
{
    ISTATUS r;
    int16_t major;
    int16_t minor;
    :
    r = ITFLib_Version(&major, &minor);
    if (r == 0) {
        /* 読み出し成功 */
    }
    :
}
```

10 PCI ドライバ

10.1 概要

ITF ミドルウェアには、PCI デバイスをサポートする製品があります。PCI の処理はハードウェアに依存します。ユーザは、ハードウェアに関わらず同等の PCI アクセス用のインターフェイス（PCI ドライバ）を ITF ミドルウェアに用意する必要があります。

ここでは、この PCI ドライバの仕様を解説します。ユーザは環境に合わせて ITF ミドルウェア用の PCI ドライバを作成する必要があります。

PCI ドライバの機能を使用する場合、次の定義ファイルをインクルードする必要があります。

pci.h

10.2 関数

PCI ドライバが用意する関数を次の通りです。ただし、ITF ミドルウェアが使用していない関数については実装する必要はありません。

（表　PCI ドライバの関数一覧）

関数	説明
pci_Init	PCIC(PCI コントローラ)の初期化。
pci_read_config_byte	PCI のコンフィグ空間バイトリードアクセス。
pci_read_config_word	PCI のコンフィグ空間ワードリードアクセス。
pci_read_config_dword	PCI のコンフィグ空間ダブルワードリードアクセス。
pci_write_config_byte	PCI のコンフィグ空間バイトライトアクセス。
pci_write_config_word	PCI のコンフィグ空間ワードライトアクセス。
pci_write_config_dword	PCI のコンフィグ空間ダブルワードライトアクセス。
pci_FindDevice	PCI デバイスサーチ。
pci_FindDevices	複数の PCI デバイスサーチ。
pci_AllocMemIo	PCI デバイスメモリアロケーション。
pci_FreeMemIo	PCI デバイスメモリ開放。
pci_sys_malloc	PCI ターゲット空間の確保。
pci_sys_free	メモリの開放。
pci_handler_register	割込み登録。
pci_handler_unregist	割込み解除。
pci_handler	PCI 割込みハンドラ本体。

pci_Init

機　能　PCIC の初期化

形　式　**void pci_Init(void);**

解　説　PCI コントローラ、内部変数の初期化を行います。

PCI ドライバで割り込みハンドラの登録が必要な場合は、割り込みハンドラの登録を行います。

本関数は、ITF ミドルウェアの初期化より以前に呼び出す必要があります。

戻　値　なし

例　初期化を行います。

```
#include “pci.h”

void task( void )
{
    pci_Init();
      :
}
```

pci_read_config_byte

機　能　PCI のコンフィグ空間バイトリードアクセス

形　式　**void pci_read_config_byte(PCI_DEV *pci,**
unsigned long regs,
unsigned char *data);
***pci　　: pci オブジェクト**
regs　　: レジスタ
***data　　: リードデータ格納ポインタ**

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスから、data が示す領域へデータを読み出します。

戻　値　なし

例　コンフィグレーションアドレス 0x400 からデータをリードします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned char cnfdata;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_read_config_byte(&pci, cnfadd, &cnfdata);
      :
}
```

pci_read_config_word

機　能　PCI のコンフィグ空間ワードリードアクセス

形　式　**void pci_read_config_word (PCI_DEV *pci,**
unsigned long regs,
unsigned short *data);
***pci　　: pci オブジェクト**
regs　　: レジスタ
***data　　: リードデータ格納ポインタ**

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスから、data が示す領域へデータを読み出します。

戻　値　なし

例　コンフィグレーションアドレス 0x400 からデータをリードします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned short cnfdata;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_read_config_word(&pci, cnfadd, &cnfdata);
      :
}
```

pci_read_config_dword

機　能　PCI のコンフィグ空間ダブルワードリードアクセス

形　式　**void pci_read_config_dword (PCI_DEV *pci,**
unsigned long regs,
unsigned long *data);
***pci　　: pci オブジェクト**
regs　　: レジスタ
***data　　: リードデータ格納ポインタ**

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスから、data が示す領域へデータを読み出します。

戻　値　なし

例　コンフィグレーションアドレス 0x400 からデータをリードします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned long cnfdata;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_read_config_dword(&pci, cnfadd, &cnfdata);
      :
}
```

pci_write_config_byte

機　能　PCI のコンフィグ空間バイトライトアクセス

形　式　**void pci_write_config_byte(PCI_DEV *pci,**
unsigned long regs,
unsigned char value);
***pci　　: pci オブジェクト**
regs　　: レジスタ
value　: ライトデータ

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスに、value のデータをライトします。

戻　値　なし

例　　コンフィグレーションアドレス 0x400 に、0xa5 をライトします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned char value = 0xa5;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_write_config_byte(&pci, cnfadd, value);
      :
}
```

pci_write_config_word

機　能　PCI のコンフィグ空間ワードライトアクセス

形　式　**void pci_write_config_word(PCI_DEV *pci,**
unsigned long regs,
unsigned short value);

***pci**　：pci オブジェクト
regs　：レジスタ
value　：ライトデータ

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスに、value のデータをライトします。

戻　値　なし

例　コンフィグレーションアドレス 0x400 に、0xa5a5 をライトします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned short value = 0xa5a5;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_write_config_word(&pci, cnfadd, value);
      :
}
```

pci_write_config_dword

機　能　PCI のコンフィグ空間ダブルワードライトアクセス

形　式　**void pci_write_config_dword(PCI_DEV *pci,**
unsigned long regs,
unsigned long value);

***pci　　: pci オブジェクト**
regs　　: レジスタ
value　: ライトデータ

解　説　コンフィグレーションサイクルを発生させます。
pci が示す PCI デバイスの regs が指定するコンフィグレーションアドレスに、value のデータをライトします。

戻　値　なし

例　　コンフィグレーションアドレス 0x400 に、0xa5a5a5a5 をライトします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long cnfadd = 0x400;
    unsigned long value = 0xa5a5a5a5;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    pci_write_config_dword(&pci, cnfadd, value);
      :
}
```

pci_FindDevice

機　能　PCI デバイスサーチ

形　式　unsigned long pci_FindDevice(PCI_DEV *pPci,
unsigned short wDeviceID,
unsigned short wVenderID,
int32_t index);

*pPci	: 見つかったデバイスの Bus/Dev の返却
wDeviceID	: サーチする DeviceID
wVenderID	: サーチする VenderID
index	: 最大検索 PCI ファンクション数

解　説　PCI コンフィグレーションレジスタをアクセスし指定 ID を持つデバイスを検索します。

戻　値

0xffffffff	: 発見できず
!0xffffffff	: 見つかったファンクション数

例　デバイス ID が 0x1234、ベンダーID が 0x5678 のデバイスをサーチします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long dev_id = 0x1234;
    unsigned long ven_id = 0x5678;
      :
    pci_FindDevice(&pci, dev_id, ven_id, 1);
      :
}
```

pci_FindDevices

機　能　PCI デバイスサーチ

形　式　unsigned long pci_FindDevices(PCI_DEV *pPci,
unsigned short wDeviceID,
unsigned short wVenderID,
int32_t index);

*pPci	: 見つかったデバイスの Bus/Dev の返却
wDeviceID	: サーチする DeviceID
wVenderID	: サーチする VenderID
index	: １デバイスの最大検索 PCI ファンクション数
max	: 最大検索 PCI ファンクション数

解　説　PCI コンフィグレーションレジスタをアクセスし指定 ID を持つデバイスを検索します。

戻　値

0xffffffff	: 発見できず
!0xffffffff	: 見つかったファンクション数

例　デバイス ID が 0x1234、ベンダーID が 0x5678 のデバイスをサーチします。

```
#include “pci.h”

void task( void )
{
    PCI_DEV pci;
    unsigned long dev_id = 0x1234;
    unsigned long ven_id = 0x5678;
      :
    pci_FindDevices(&pci, dev_id, ven_id, 1);
      :
}
```

pci_AllocMemIo

機　能　PCI デバイスメモリアロケーション

形　式　**int32_t pci_AllocMemIo(PCI_DEV *pPci);**
***pPci　：資源割り当てを依頼する PCI デバイス構造体ポインタ**

解　説　PCI コンフィグレーションレジスタをアクセスし PCI 空間上にデバイスをマッピングします。想定としてデバイス割り当ての上位側は普遍と考えた場合のマッピングです。サイズの規模が大きくなる場合はコンフィグレーション経由でのアクセスが必須となります。

戻　値　**-1　：空間マップに失敗**
-2　：Configration ライトエラー
-3　：Command ライトエラー
0　：割り当て成功

例　PCI デバイスメモリを確保します。

```
#include “pci.h”

void task( void )
{
    int32_t r;
    PCI_DEV pci;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    r = pci_AllocMemIo(&pci);
    if (r < 0) {
        /* error */
    }
      :
}
```

pci_FreeMemIo

機　能　PCI デバイスメモリ開放

形　式　**int32_t pci_FreeMemIo(PCI_DEV *pPci);**
***pPci　：資源開放を依頼する PCI デバイス構造体ポインタ**

解　説　pci_AllocMemIo でアロケーションした PCI デバイスメモリを開放します。

戻　値　**-1　　：資源開放失敗**
0　　：資源開放成功

例　PCI デバイスメモリを確保します。

```
#include “pci.h”

void task( void )
{
    int32_t r;
    PCI_DEV pci;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    r = pci_AllocMemIo(&pci);
    if (r < 0) {
        /* error */
    }
      :
    r = pci_FreeMemIo(&pci);
    if (r < 0) {
        /* error */
    }
      :
}
```

pci_sys_malloc

機　能　PCI ターゲット空間の確保

形　式　**void* pci_sys_malloc(unsigned long size);**
size　　：確保したいバイト数

解　説　PCI ターゲットメモリを割り当てます。
確保サイズをアライメントサイズで割ることで確保するインデックスの数を求めインデックステーブルにマーキングを行います。

戻　値　**0　　：確保失敗**
!0　　：割り当て PCI アドレス

例　PCI ターゲットメモリを 100 バイト確保します。

```
#include “pci.h”

void task( void )
{
    void *buf;
      :
    buf = pci_sys_malloc(100);
    if (!buf) {
        /* error */
    }
      :
}
```

pci_sys_free

機　能　メモリの開放

形　式　**void pci_sys_free(void* buffer);**
***buffer　: pci_sys_malloc で確保したバッファポインタ**

解　説　pci_sys_malloc で確保したメモリの開放します。ブロック管理テーブルのマーキングクリア処理を行います。

戻　値　なし

例　確保した PCI ターゲットメモリを開放します。

```
#include “pci.h”

void task( void )
{
    void *buf;
      :
    buf = pci_sys_malloc(100);
    if (buf) {
        /* error */
    }
      :
    pci_sys_free(buf);
      :
}
```

pci_handler_register

機　能　割込み登録

形　式　int32_t pci_handler_register(PCI_DEV *pDev,
　　void (*func)(PCI_DEV *));
*pDev　：pci オブジェクト
*func　：コールバック関数ポインタ

解　説　PCI 割込みハンドラから登録割込みをコールバックするためのデバイスハンドラの登録を行います。

pci オブジェクトのメンバのコンテキスト context は、登録元が自由に使用して良いメンバ変数となります。割り込みハンドラから登録したルーチンが呼び出されるときに、pci オブジェクトが渡されますので、そこから context を使用することが可能となります。

戻　値　1　：正常登録
0　：登録失敗

例　コールバックルーチンを登録します。

```
#include “pci.h”

void callback(PCI_DEV *pci)
{
      :
}

void task( void )
{
    int32_t r;
    PCI_DEV pci;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    r = pci_handler_register(&pci, callback);
    if (r == 0) {
        /* error */
    }
      :
}
```

pci_handler_unregist

機　能　割込み解除

形　式　int32_t pci_handler_unregist(PCI_DEV *pDev);
*pDev　：pci オブジェクト

解　説　登録した割り込みハンドラを呼び出されないようにするための削除処理を行います。

戻　値　1　　：正常解放
0　　：未登録です

例　pci_handler_register で登録したコールバックルーチンの登録を解除します。

```
#include “pci.h”

void callback(PCI_DEV *pci)
{
      :
}

void task( void )
{
    int32_t r;
    PCI_DEV pci;
      :
    pci_FindDevice(&pci, 0, 0, 1);
      :
    r = pci_handler_register(&pci, callback);
    if (r == 0) {
        /* error */
    }
      :
    r = pci_handler_unregist(&pci);
    if (r == 0) {
        /* error */
    }
}
```

pci_handler

機　能　PCI 割込みハンドラ処理

形　式　**void pci_handler(void);**

解　説　PCI 割り込みが発生した時に行う処理です。
本関数を PCI の割り込みハンドラとするか、PCI の割り込みハンドラから本関数を呼び出す必要があります。

本関数内では、pci_handler_register で登録されたコールバックルーチンを呼び出します。

PCI ドライバで割り込みを発生させたデバイスの特定ができない場合、登録されている全てのコールバックルーチンを呼び出します。よって、コールバックルーチンを登録した各ドライバは、その割り込みが自分自身かまず判定しなければなりません。自分自身の割り込みでない場合、即時に処理を終了する必要があります。

戻　値　なし

例　pci_handler_register で登録されて割り込みハンドラを呼び出します。

```
#include “pci.h”

static PCI_DEV *gPCIDevice[MAX_PCI_DEVICE];
void pci_handler(void)
{
    int32_t i;

    for(i=0;i<MAX_PCI_DEVICE;i++){
        if( gPCIDevice[i] ){
            if( gPCIDevice[i]->handler ){
                (*gPCIDevice[i]->handler)(gPCIDevice[i]);
            }
        }
    }
}
```

10.3 構造体

PCI 構造体

```
typedef struct pci_dev_t {
    unsigned long bus_no;          バス番号
    unsigned long dev_no;          デバイス番号
    unsigned long fnc_no;          ファンクション番号
    unsigned long base_io;         I/O ベースアドレス
    unsigned long base_mem;        メモリベースアドレス
    unsigned long base_mem2;       メモリベースアドレス
    unsigned long base_mem3;       メモリベースアドレス
    unsigned long base_rom;        拡張 ROM アドレス
    unsigned long intr;            割り込み番号
    void (*handler)(struct pci_dev_t *);    割り込みハンドラ
    void *context;                 コンテキスト
} PCI_DEV;
```

10.4 アドレス変換

PCI ドライバでは、PCI バス空間メモリアドレスと物理メモリアドレスを変換するマクロを提供します。これは、pci.h に定義されています。

物理メモリアドレスから PCI バス空間メモリアドレスに変換するマクロは次の通りです。

phymem_to_pcimem(mem)
mem　　：変換する物理メモリアドレス
戻り値　：変換された PCI バス空間メモリアドレス

PCI バス空間メモリアドレスから物理メモリアドレスに変換するマクロは次の通りです。

pcimem_to_phymem(mem)
mem　　：変換する PCI バス空間メモリアドレス
戻り値　：変換された物理メモリアドレス

ITF ミドルウェアでは、各アドレスを取得するときに上記のマクロを使用します。PCI ドライバを作成する場合、適切にマクロを記述する必要があります。

10.5 サンプルドライバ

本ライブラリには、SH7751/SH7751R（Solution Engine MS7751RSE01）に内蔵されている PCI コントローラ用のドライバのサンプルソースが付属しています。PCI ドライバを作成する上で参考にして下さい。

サンプルソースファイルは、次の通りです。

sh7751r_pci.c
m1543c.c
sh7751r.h
pci.h

SH7751/SH7751R の PCI ドライバを使用する場合は、上記ファイルを【SRC¥ITFLib¥Sample¥Driver】から、【SRC¥ITFLib¥Source¥Depend】にコピーして使用して下さい。PCI ドライバを使用する場合は、pci.h をコピーしたディレクトリにインクルードパスを設定する必要があります。ここでは、【SRC¥ITFLib¥Source¥Depend】となります。

pci.h には、PCI メモリとして使用する領域が定義されています。この定義は使用している環境に合わせて適切に変更して下さい。

PCI メモリ空間開始アドレスは次の定義となります。

```
#define   PCI_MEMWINDOW        0x0f000000
```

PCI メモリ空間サイズは次の定義となります。

```
#define   PCI_MEMSIZE              0x01000000
```

11 USB ドライバ

11.1 概要

ITF ミドルウェアには、USB コントローラをサポートする製品があります。それらの製品は通常、コントローラのドライバとして提供されています。

本ライブラリは、ターゲットに依存する処理を USB ドライバとして提供します。ターゲットに依存するような処理が存在する場合、USB ドライバを各ミドルウェアに提供する必要があります。

11.2 関数

USB ドライバが用意する関数を次の通りです。ただし、ITF ミドルウェアが使用していない関数については実装する必要はありません。

（表　USB ドライバの関数一覧）

関数	説明
usb_Init	ターゲット依存の USB の初期化。

usb_Init

機　能　ターゲット依存の USB の初期化

形　式　**void usb_Init(void);**

ISTATUS usb_Init(void);

解　説　USB 周辺モジュールに関するターゲット依存部の初期化処理を行います。
コントローラの初期化等は、各ミドルウェアで行います。

本関数は、ITF ミドルウェアの初期化より以前に呼び出す必要があります。

戻　値　なし

0　　：正常終了
0 以外　：異常終了

例　初期化を行います。

```
void task( void )
{
    usb_Init();
      :
}
```

11.3 サンプルドライバ

本ライブラリは USB ドライバのサンプルを提供しています。サンプルドライバでは各コントローラに必要な処理（主に初期化処理）が記述されています。

各サンプルドライバを使用する場合は、サンプルドライバのファイルを【SRC¥ITFLib¥Sample¥Driver】から、【SRC¥ITFLib¥Source¥Depend】にコピーして使用して下さい。また、サンプルドライバにヘッダファイルが存在する場合は、サンプルドライバをコピーしたディレクトリにインクルードパスを設定する必要があります。ここでは、【SRC¥ITFLib¥Source¥Depend】となります。

サンプルドライバを使用する場合、お客様の使用しているハードウェアによっては、サンプルドライバの変更が必要となる場合があります。その場合はハードウェアのマニュアル等を参照し、サンプルドライバを適切に変更して下さい。

11.3.1 SH7727 用ドライバ

本ライブラリには、SH7727 の USB ドライバのソースが付属しています。このソースには、SH7727 内蔵 USB ファンクションコントローラ、及び SH7727 内蔵 USB ホストコントローラの共通の初期化処理が記述されています。

サンプルソースファイルは、次の通りです。

sh7727_usb.c

11.3.2 SH7760 用ドライバ

本ライブラリには、SH7760 の USB ドライバのソースが付属しています。このソースには、SH7760 内蔵 USB ホストコントローラの初期化処理や DMA の処理が記述されています。

サンプルソースファイルは、次の通りです。

sh7760_usb.c

11.3.3 ML60842 用ドライバ

本ライブラリには、ML60842 の USB ドライバのソースが付属しています。このソースには、ML60842 の初期化処理や割り込み処理が記述されています。

サンプルソースファイルは、次の通りです。

ml60842_usb.c
ml60842.h

11.3.4 SH 用 DMA ドライバ

本ライブラリには、SH の DMA ドライバのソースが付属しています。このソースには、SH7751 内蔵 DMA コントローラの処理が記述されています。
サンプルソースファイルは、次の通りです。

sh_dma.c
dma.h

本ドライバを使用する場合は、お客様の環境に合わせて変更する必要があります。
本ドライバが提供する API は次の通りです。

dma_Init

機　能　DMA の初期化

形　式　**void dma_Init(void);**

解　説　DMA の初期化処理を行います。

本関数は、ITF ミドルウェアの初期化より以前に呼び出す必要があります。

戻　値　なし

例　DMA の初期化を行います。

```
void task( void )
{
    dma_Init();
      :
}
```

dma_Start

機　能　ターゲット依存の DMA の開始

形　式　int32_t dma_Start(int32_t ch, void *src, void *dest, uint32_t count, uint32_t flag);

ch	: DMA チャンネル番号
src	: 転送元アドレス
dest	: 転送先アドレス
count	: 転送バイト数
flag	: フラグ

解　説　DMA を開始します。

戻　値

0	: 正常終了
0 以外	: 異常終了

例　DMA を開始します。

```
#define DMA_CH          0

void task( void )
{
    int32_t r;

      :

    r = dma_Start(
                DMA_CH,
                src_buffer,
                dst_buffer,
                2048,
                dma_flag
                );
    if ( r ) {
        /* エラー */
    }
      :
}
```

dma_Wait

機　能　DMA の完了待機

形　式　int32_t dma_Wait(int32_t ch, int32_t timeout);
ch　　：DMA チャンネル番号
timeout　：タイムアウト時間[m 秒]

解　説　dma_Start で開始した DMA の完了を待機します。

戻　値　0　　：正常終了
0 以外　：異常終了

例　DMA の完了を待機します。

```
#define DMA_CH          0
#define DMA_TIMEOUT 10000

void task( void )
{
    int32_t r;
     :
    r = dma_Start(DMA_CH,src_buffer,dst_buffer,2048,dma_flag);
    if ( r ) {
        /* エラー */
    }

    r = dma_Wait(DMA_CH, DMA_TIMEOUT);
    if ( r ) {
        /* エラー */
    }
     :
}
```

dma_AddressCheck

機　能　DMA のアドレスチェック

形　式　int32_t dma_AddressCheck(void *adr);
adr　　：チェックするアドレス

解　説　DMA 転送に使用できるアドレスエリアかチェックします。

戻　値　1　　：DMA 転送が可能
0　　：DMA 転送が不可能

例　　DMA 転送用バッファをチェックします。

```
#define DMA_BUFFER   0xac000000

void task( void )
{
    int32_t r;

     r = dma_AddressCheck(DMA_BUFFER);
     if ( r ) {
          /* DMA 可能 */
     }

}
```

11.3.1 LPC24xx 用ドライバ

本ライブラリには、LPC24xx の USB ドライバのソースが付属しています。このソースには、LPC24xx 内蔵 USB ファンクションコントローラ、及び LPC24xx 内蔵 USB ホストコントローラの共通の初期化処理が記述されています。

サンプルソースファイルは、次の通りです。

lpc24xx_usb.c

12 割り込み

本ライブラリの PCI や USB のサンプルドライバでは割り込みを使用している場合があります。
それぞれのサンプルドライバでは、割り込みハンドラを次のように pragma を使用して定義しています。

```
#pragma interrupt(InterruptRoutine)
```

割り込みハンドラはコンパイラ等の環境によって定義の方法が異なります。割り込みハンドラの定義方法の詳細については、コンパイラ等のマニュアルを参照し、その内容に従って割り込みハンドラを定義するようにして下さい。
上記内容は各ミドルウェアにも当てはまります。各ミドルウェアで割り込みを使用している場合は上記内容に注意して下さい。

13 デバッグ

コンフィグレーションでデバッグを有効（「__ITFDEBUG__」を定義）にした場合に、本ライブラリは ITF ミドルウェアにデバッグ機能を提供します。
提供される機能は主に次の機能です。

- メッセージ出力機能
- アサートチェック機能
- ダンプ機能

デバッグ機能は、次のファイルに全て定義されています。全てのデバッグ機能はマクロで記述されています。マクロの詳細は省略します。

itfdebug.h

デバッグ機能で必要となる処理は、次のファイルに記述されています。

itfdbug.c

“itfdebug.h”には、どのデバッグ機能を有効にするか宣言されています。

（表　デバッグ機能と宣言名）

機能	宣言名
メッセージ出力	__ITFDEBUG_STRING_PRINT__
アサートチェック	__ITFDEBUG_ASSERT__
ダンプ	__ITFDEBUG_DUMP__

表の宣言名が定義されていることにより、各デバッグ機能は有効になります。

13.1 メッセージ出力機能

13.1.1 機能

メッセージ出力機能は、メッセージを出力する場合に使用します。メッセージ出力機能は全てマクロで提供されています。
次は提供されているマクロの一覧です。

（表　メッセージ出力機能一覧）

マクロ名	機能
IDP	文字列出力
IDP1	フォーマットあり文字列出力（パラメータ１個）
IDP2	フォーマットあり文字列出力（パラメータ２個）
IDP3	フォーマットあり文字列出力（パラメータ３個）
IDP4	フォーマットあり文字列出力（パラメータ４個）
IDP5	フォーマットあり文字列出力（パラメータ５個）
IDP6	フォーマットあり文字列出力（パラメータ６個）
IFLDP	ファイル名とライン数文字列出力
ITRACE	トレース文字列出力
IERR	エラー文字列出力
IFLERR	エラー文字列とファイル名とライン数文字列出力
IFLERR1	エラー文字列とファイル名とライン数文字列出力
IWARN	警告文字列出力
IFLWARN	警告文字列とファイル名とライン数文字列出力
IFLWARN1	警告文字列とファイル名とライン数文字列出力
IDBP	バイナリ出力

13.1.2 出力処理

メッセージ出力機能では、出力処理を環境に合わせて変更する必要があります。変更する関数は次の通りです。

ITFDEBUG_Put_String

機　能　文字列を出力する

形　式　**void ITFDEBUG_Put_String(char *str);**
str,　　：出力する文字列

解　説　str 文字列を出力します。
出力先は、シリアルインターフェイスやメモリの場合が考えられます。

戻　値　なし

ITFDEBUG_Put_Binary

機　能　バイナリを出力する

形　式　**void ITFDEBUG_Put_Binary(unsigned char *bin, long length);**
bin,　　**：出力するバッファ**
length,　　**：出力するバイト数**

解　説　bin が示すバッファを length バイト出力します。
出力先は、シリアルインターフェイスやメモリの場合が考えられます。

戻　値　なし

これらの関数は、”itfdebug.c”に記述されています。

13.2 アサートチェック機能

アサートチェック機能では、渡されたパラメータの真偽を判定します。判定結果が偽の場合は、メッセージ出力機能を使用して警告し、その場で永久ループします。永久ループ後、そのタスクの復帰は不可能です。

アサートチェック機能もメッセージ出力機能と同様にマクロで提供されています。マクロ名は次の通りです。

IASSERT(a)

a には真偽を判定する値を入れます。

13.3 ダンプ機能

ダンプ機能は、指定バッファをダンプした文字列で出力する場合に使用します。文字列の出力はメッセージ出力機能を使用しています。

ダンプ機能もメッセージ出力機能と同様にマクロで提供されています。マクロ名は次の通りです。

（表　ダンプ機能一覧）

マクロ名	機能
IDUMP8	16 進数で１バイト単位のダンプ
IDUMP16	16 進数で２バイト単位のダンプ
IDUMP32	16 進数で４バイト単位のダンプ
IDUMPB	２進数で１バイト単位のダンプ

13.4 注意事項

本デバッグ機能は、ITF ミドルウェア開発時におけるサポート機能です。本機能をユーザアプリケーションに提供しているわけではありません。

通常使用時において、ユーザアプリケーションが本機能を使用することはありません。また、本機能を使用して発生した問題等についてはいかなることも保証致しません。

14 導入

本ライブラリは単独では使用しません。導入方法については、各ミドルウェアのマニュアルを参照して下さい。

15 一覧

15.1 API

関数名	内容
情報取得関数	
ITFLib_Version	製品バージョン番号の読み出し

15.2 エラーコード

シンボル名	エラー番号	内容
IE_OK	0	正常終了
IE_SYS	-5	システムエラー
IE_NOSPT	-9	未サポート機能
IE_RSFN	-10	予約機能コード
IE_RSATR	-11	予約属性
IE_PAR	-17	パラメータエラー
IE_ID	-18	不正 ID 番号
IE_CTX	-25	コンテキストエラー
IE_MACV	-26	メモリアクセス違反
IE_OACV	-27	オブジェクトアクセス違反
IE_ILUSE	-28	サービスコール不正使用
IE_NOMEM	-33	メモリ不足
IE_NOID	-34	ID 番号不足
IE_OBJ	-41	オブジェクト状態エラー
IE_NOEXS	-42	オブジェクト未生成
IE_QOVR	-43	キューイングオーバーフロ
IE_RLWAI	-49	待ち状態の強制解除
IE_TMOUT	-50	ポーリング失敗またはタイムアウト
IE_DLT	-51	待ちオブジェクトの削除
IE_CLS	-52	待ちオブジェクトの状態変化
IE_WBLK	-57	ノンブロッキング受付け
IE_BOVR	-58	バッファオーバーフロー
IE_INOSPT	-97	未サポート

ご注意

■お問い合わせ

インターフェイス株式会社

〒190-0022　東京都立川市錦町２－２－１１

TEL　042-528-8651　　FAX　042-528-8678

E-mail　sales@itf.co.jp