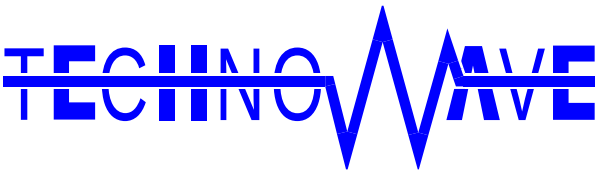
X-A0800
ユーザーファーム開発マニュアル



テクノウェーブ株式会社

目次

1. はじめに

このたびは弊社製品をご購入頂き、まことにありがとうございます。このマニュアルでは弊社製品『USBX-A0800』/『LANX-A0800』/『LANX-A0800-L1』のファームウェア開発方法を解説しています。それぞれの製品には、製品を安全にご利用いただくための注意事項、ハードウェア仕様などを記載したユーザーズマニュアルを別紙にて用意しておりますので、製品のご利用を開始される前にご一読いただけますようお願いいたします。

□ マニュアル内の表記について

本マニュアル内では対応製品を、単に「製品」または「デバイス」と表記します。また、USB インタフェースの製品全般 を「USB デバイス」、LAN インタフェースの製品全般を「LAN デバイス」と表記します。

ハードウェアの各電気的状態について下記のように表記いたします。

表 1　電気的状態の表記方法

表記	状態
"ON"	電流が流れている状態、スイッチが閉じている状態、オープンコレクタ(オープンドレイン)出力がシンク出力している状態。
"OFF"	電流が流れていない状態、スイッチが開いている状態、オープンコレクタ(オープンドレイン)出力がハイインピーダンスの状態。
"Hi"	電圧がロジックレベルのハイレベルに相当する状態。
"Lo"	電圧がロジックレベルのローレベルに相当する状態。

数値について「0x」、「&H」、「H'」はいずれもそれに続く数値が 16 進数であることを表します。"0x10"、"&H1F"、"H'20"などはいずれも 16 進数です。

2. ユーザーファームの概要

□ ユーザーファームとは

製品には制御用として H8/3069 マイコン(ルネサスエレクトロニクス)が搭載されています。マイコンにはホストパソコンからの命令を実行するための基本的なプログラムが組み込まれており、このプログラムのことを***システムファーム***と呼びます（パソコン上で動作するプログラムやソフトウェアと区別するために、マイコン用のプログラムのことをファームウェア、または単にファームと呼びます)。

ホストパソコン上で開発されたアプリケーションソフトから製品を制御する場合、専用のライブラリを通じてシステムファームにコマンドを送り、システムファームが受け取ったコマンドに応じた動作を行います（図 1)。

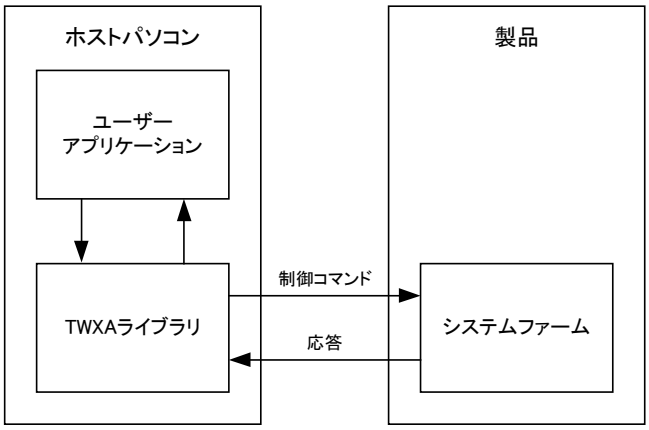


図 1　ホストパソコンからの制御

製品では、上記のように予め用意された機能を利用してホストパソコンから制御を行う他に、搭載マイコン用のプログラムをユーザーが開発する仕組みもサポートされています。これにより、マイコン上のプログラムでなければ実現が困難なリアルタイム性が要求される処理や、基本機能では提供されない複雑な処理にも対応可能です。このマイコン上で動作する追加プログラムのことを***ユーザーファーム***と呼びます。

ユーザーファームはマイコンのフラッシュメモリ、または、RAM 上にシステムファームと共存する形でダウンロードすることができ、さらにシステムファームが提供するユーティリティ関数（***サービス関数***と呼びます）を利用することができます。

□ ユーザーコマンドの追加

ユーザーファームの 1 つめの使用法はシステムファームではサポートされない新しいコマンドを追加することです。パソコンから一つずつ命令を送っていては時間がかかってしまう一連の処理や、細かなタイミング制御を要求されるリアルタイム性の高い処理を予めマイコン用のプログラムとして作成しておき、必要なときにコマンドを送って呼び出すことができます（図 2)。この追加したコマンドを***ユーザーコマンド***と呼びます。またユーザーコマン

ドを処理するための関数は*コマンドハンドラ*と呼びます。

システムファームでサポートされるコマンドはそのまま利用できますので、ユーザーは独自の処理だけをプログラムすれば良く、効率的な開発が可能です。

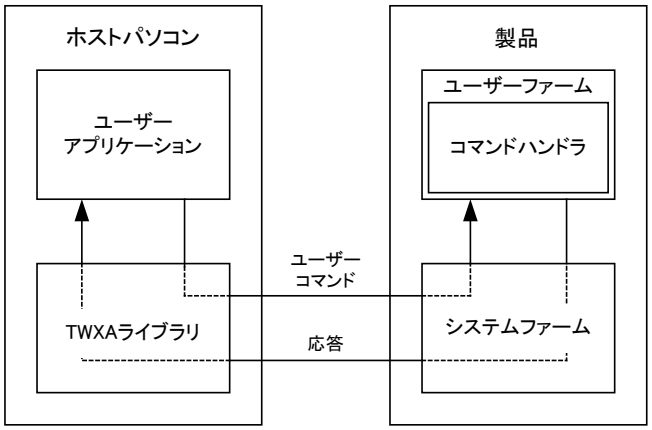


図 2　新しいコマンドの追加

□　自律動作

２つめの使用法はホストパソコンと無関係に自律的な動作をさせることです。開発するアプリケーションによっては、単にホストパソコンからのコマンドに応答するだけでは無く、マイコンが独自のルーチンで自律的に動作する必要があります。そのような場合、追加したプログラムをシステムファームに登録すれば、コマンドの到着を監視している*コマンドループ*の中から定期的に呼び出しを受けることができます（図 3)。呼び出しを受ける関数のことを*メイン関数*と呼びます。

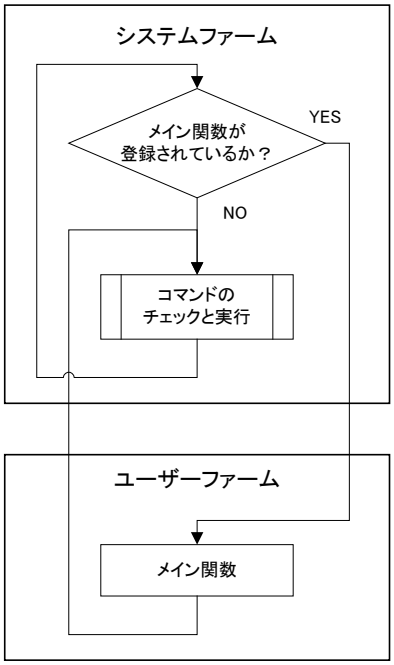


図 3　自律動作

- USB インタフェースの製品は、ホストパソコンから電源の供給を受けるバスパワー動作を行いますので、ホストパソコンの電源が入っていない状態で動作させることはできません。

□ ユーザーファームの配置

ユーザーファームはフラッシュメモリに書き込むことも、RAM 上に一時的に配置されるように作成することもできます。ユーザーファーム用の領域としてフラッシュメモリ上に 256K バイトの領域が予約されています。また、RAM 上にはユーザーが自由に使用できる***ユーザーメモリ***が 10K バイト用意されていますので、その位置にユーザーファームをダウンロードすることができます（図 4）。

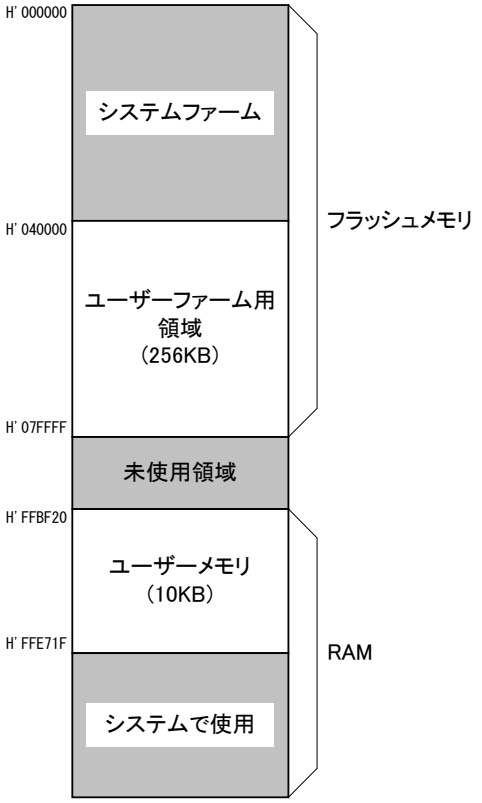


図 4　メモリ空間

フラッシュ版ユーザーファーム

本マニュアルでは便宜上、フラッシュメモリに書き込むユーザーファームをフラッシュ版ユーザーファームと呼ぶことにします。フラッシュ版ユーザーファームはデバイスの電源を切っても消えることはありません。そのため、自律動作のためのユーザーファームを用意すれば、パソコンとの接続は必要なく、デバイス単体で動作することができます。さらに、システムファームの機能を利用すれば、必要なときにはパソコンと接続して通信を行うようなシステムとすることもできます。

アタッチメントファーム

RAM 上にユーザーファームをダウンロードするには、必ずパソコンとの接続が必要になります。ユーザーファームは通常、パソコンに保存され、必要なときにデバイス上のマイコンにダウンロードします。RAM 上にダウンロードするように作成されたユーザーファームのことを特に***アタッチメントファーム***と呼びます。また、コンパイラから出力されたプログラムを、アタッチメントファームとしてダウンロード可能な形式に変換したファイルを ***ATF ファイル***（拡張子は「.atf」）と呼びます。

アタッチメントファームを使用する利点の１つは、必要な機能を必要なときだけ追加できるということです。必要が無くなれば、あるアタッチメントファームを削除し、別の機能のものをダウンロードすることができます（図 5)。また、ATF ファイル自体はパソコン用のファイルなので配布や更新が容易なのもメリットの１つです。欠点としてはデバイスの電源を切ると、消えてしまうということです。利用状況に合わせてフラッシュメモリに配置するのか、RAM に配置するのかを使い分けてください。

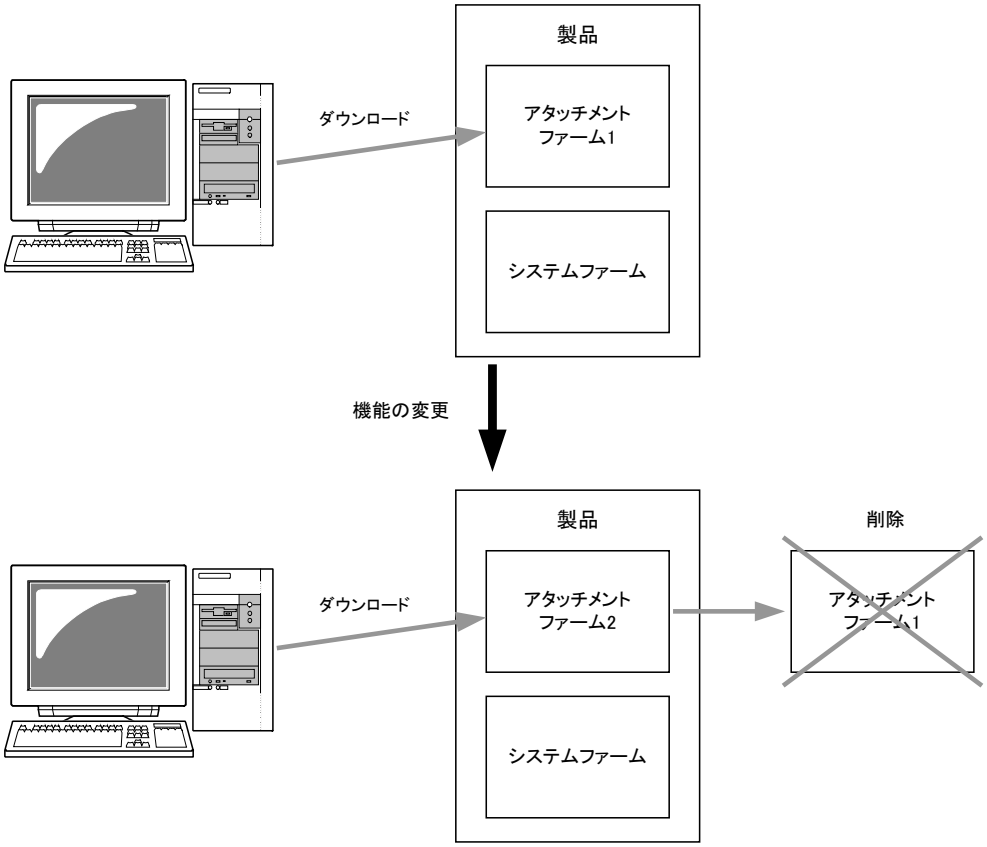


図 5　アタッチメントファームを使用した機能変更

□ ユーザーファームの互換性

一定のルールに従って作成されたユーザーファームは、USB インタフェースと LAN インタフェースのどちらの製品でも利用することができます。システムファームがホストパソコンとのインタフェースの違いを吸収しますので、ユーザーファームのバイナリをどちらのデバイスにダウンロードしても同様に使用可能となります。

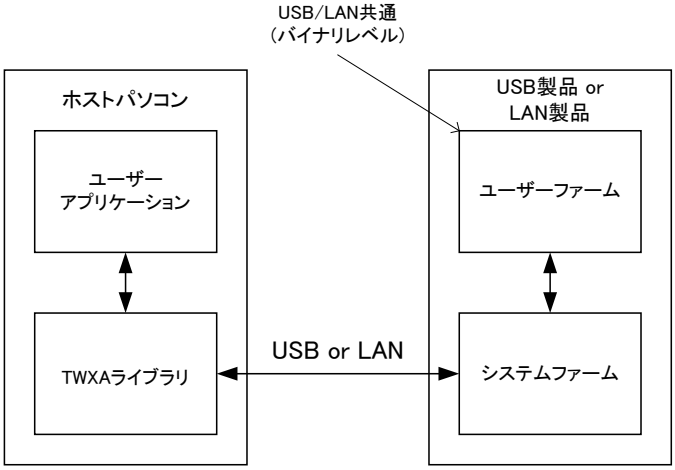


図 6　USB または LAN 経由での制御

□ ネットワークアプリケーションの開発

LAN インタフェースの製品は、USB インタフェースの製品と高い互換性を持っていますが、ネットワーク製品ならではの独自のアプリケーション開発も可能です。LAN インタフェース製品に搭載されるシステムファームでは、ソケットライクなサービス関数を提供し、DHCP、DNS、SNTP を標準でサポート、GUI によるネットワーク設定ツールも付属するなど、最小限のプログラミングでネットワークアプリケーションの作成が可能となっています。

□ ユーザーファームの開発環境

ユーザーファームの開発言語には C 言語を使用します。開発環境はエル・アンド・エフ社の『YellowIDE(YCH8)』、『イエロースコープ(YSH8)』をサポートします。以下に開発に必要な条件を示します（表 2)。

表 2　開発に必要なもの

項目	条件
OS	Windows XP、Vista、7 のいずれか
パソコン	上記 OS がインストールされた PC-AT 互換機で、シリアルポート（RS-232C）が使用可能なもの
製品	対応製品
コンパイラ	YellowIDE（YCH8）7.00 以降
デバッガ	イエロースコープ(YSH8)
デバッグ用通信ケーブル	SER1 とパソコンのシリアルポートを接続するためのケーブル。配線は製品のユーザーズマニュアルを参照してください[1]。

[1] 別売りの評価用ワイヤキット(WMB-X0404-KIT1)には、SER1 コネクタと D-sub コネクタの変換ケーブルが含まれています。

3. 開発の準備

□ コンパイラ、デバッガの準備

ユーザーファームの開発には『YellowIDE(YCH8)』、および、『イエロースコープ(YSH8)』を使用します。既にお持ちの場合には、そのままご利用いただけます。セットアップ方法については、それぞれの製品マニュアル等を参照してください。

□ ユーザーファーム開発用ツールのインストール

ユーザーファームの開発にはコンパイラ、デバッガの他に以下のツールが必要になります。

表 3　ユーザーファームの開発ツール

ツール名	機能	標準のインストールフォルダ
M3069FlashWriter	プログラムをフラッシュメモリに書き込みます。	C:¥Program Files¥Technowave¥USBMTools (C:¥Program Files (x86)¥Technowave¥USBMTools)
ATF Maker	プログラムを ATF ファイルに変換します。	
M3069IniWriter	ユーザーファームの動作パラメータを設定する場合に使用します。	

上記のツールは対応製品の設定ツールに含まれます。設定ツールについては、製品のユーザーズマニュアルを参照してください。

『YellowIDE』への登録

上記のツール類を『YellowIDE』の外部ツールとして登録することで、より便利にご利用いただけます。設定ツールのインストーラは『YellowIDE』がインストールされていることを発見すると、図 7 のダイアログを表示します。[はい]を選択すると自動的にツール類が『YellowIDE』に登録され、図 8 のように[ツールバー]と[ツール]メニューが変更されます。これらを使って簡単に開発ツールを呼び出すことが可能になります。

既に設定ツールをインストール済みの場合、再度セットアッププログラムを実行すれば同様に自動登録することができます。

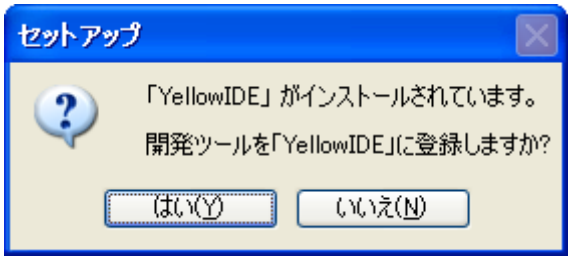


図 7　『YellowIDE』への登録確認

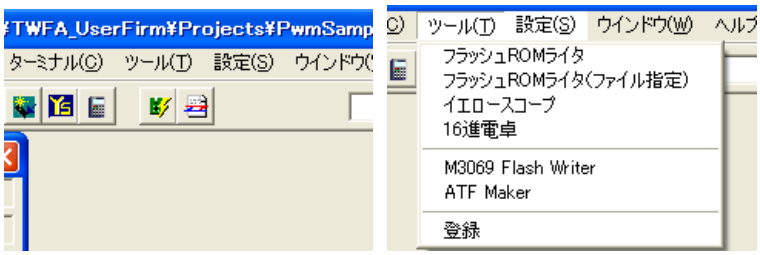


図 8　登録後の[ツールバー]と[ツール]メニュー

何らかの理由で自動登録が正しく実行されない場合には、手動で登録することもできます。『YellowIDE』の[ツール]メニューから[登録]を選択し、登録を行ってください。それぞれのツールの登録画面を図 9 と図 10 に示します。

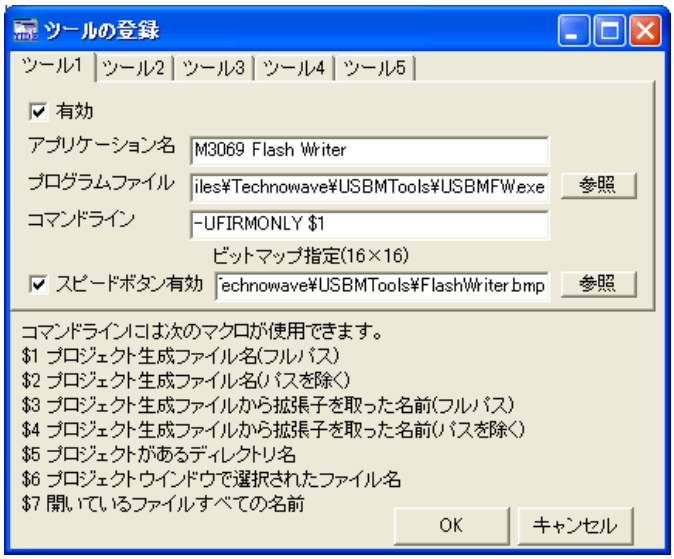


図 9　「M3069FlashWriter」の登録画面

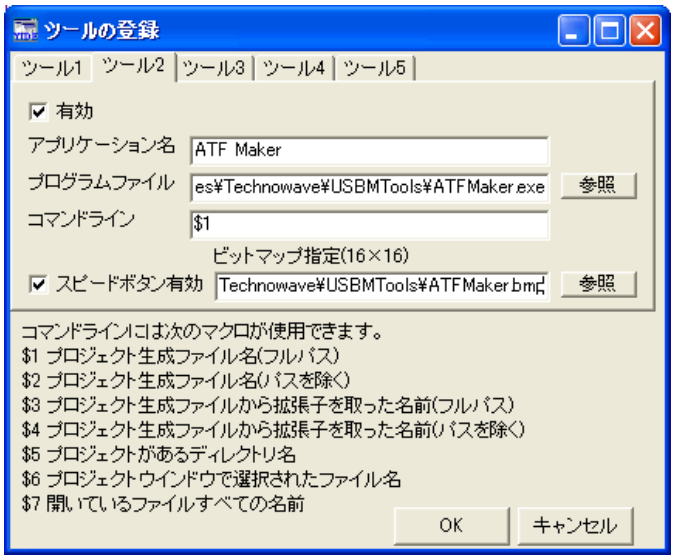


図 10　「ATF Maker」の登録画面

□ 開発用ファイル、サンプルプログラムの準備

ユーザーファーム開発用のスタートアップルーチン、インクルードファイル、サンプルプログラムを準備します。これらのファイルは、弊社のホームページ「https://www.techw.co.jp/SupportFrm.html?pid=X-A0x0x」の「ユーザーファーム開発用ファイル、サンプルプログラム」からダウンロードいただけます。

ダウンロードファイルを解凍したフォルダから、「TWFA_UserFirm」以下のファイルをローカルドライブの適当なフォルダにコピーします。ただし、フォルダパスにスペースが含まれると、『YellowIDE』でプロジェクトを開けなくなりますので、ご注意ください。

「TWFA_UserFirm」以下には表 4 の各フォルダが含まれます。以降、本マニュアルでこれらのフォルダを示す場合、「TWFA_UserFirm」以前のパスを省略して「¥Include」のように記述します。

表 4　ユーザーファームの開発用ファイル

フォルダ名	説明
Include	ユーザーファーム開発用のインクルードファイル(.h)ファイルが含まれます。
Lib	ユーザーファーム開発用のライブラリファイル(.lib)ファイルが含まれます。
Startup	ユーザーファーム開発用のスタートアップルーチンが含まれます。
A0x0xProjects	サンプルプロジェクトと、専用のデバッグモニタのプロジェクトが含まれます。

表中のスタートアップルーチンとは、起動時に最初に実行されるプログラムで、グローバル変数やヒープ領域の初期化、スタックポインタの設定などを行い、主にプログラムの実行環境を整える役目を果たします。ユーザーファームの開発には、専用のスタートアップルーチンを使用します。デバッグモニタについては以降で説明します。

□ インクルードパスの設定

『YellowIDE』のインクルードパスに前記の「¥Include」フォルダを登録します。[設定]メニューから[インクルードパス]を選択してください(図 11)。図 12 のような画面が表示されますので、[参照]ボタンを押して「¥Include」フォルダを選択し、[追加]ボタンでインクルードパスに追加してください。

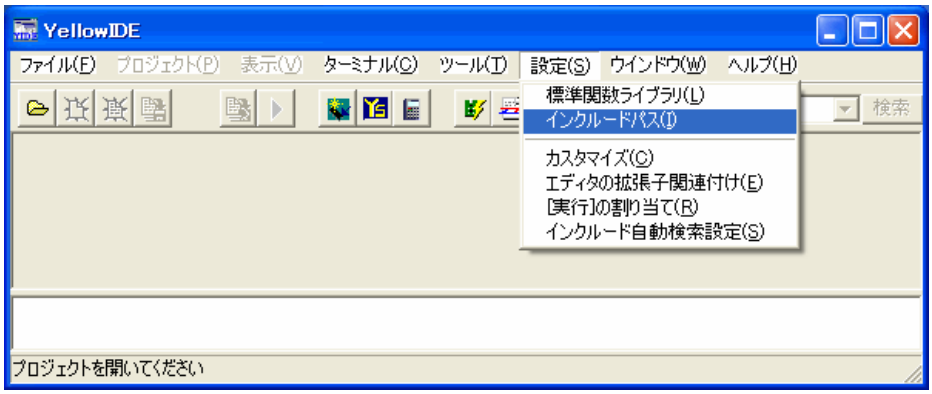


図 11　インクルードパスの設定

図 12　インクルードパスの設定画面

□ デバッグモニタの書き込み

『イエロースコープ』を使用してデバッグを行うために、マイコンにデバッグモニタを書き込みます。デバッグモニタはマイコン上で動作するプログラムで、シリアルポートを通じてデバッガから送られてくるコマンドに応答し、デバッグ中のマイコンの動作を制御したり、レジスタやメモリ情報を読み出したりといった操作を可能にします。

- ユーザーファームの開発で使用するデバッグモニタは専用のものです。『YellowIDE』に付属する通常の H8/3069 マイコンのものは使用できません。

デバッグモニタ書き込み手順

1. 製品の電源を切った状態でディップスイッチの 2 番を"ON"にし、フラッシュ書換えモードに設定します。
2. 製品の電源を入れ、USB ケーブル、または、LAN ケーブルを接続し、パソコンと通信可能な状態にします。
3. 「M3069FlashWriter」を起動します。[スタート]メニュー→[すべてのプログラム]（または、[プログラム]）→[テクノウェーブ]から、「USBX-A0x0xTools」または「LANX-A0x0xTools」を起動します。
4. メニュー画面が表示されますので「M3069FlashWriter」を選択してください。
5. [参照]ボタンを押し[ダウンロードファイル]に「¥A0x0xProjects¥_TWMON¥REM_MON.S」を選択します。

   [書込み]ボタンを押してデバッグモニタを書き込みます。終了したら電源を切ってディップスイッチの 2 番を"OFF"に戻してください。接続に失敗する場合は「M3069FlashWriter」のオンラインヘルプを参照してください。

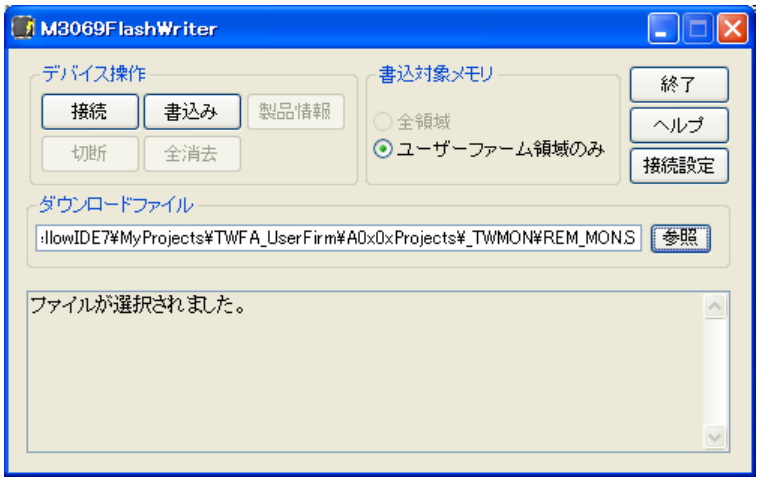


図 13　デバッグモニタの書き込み

デバッグモニタの動作確認

1. 『YellowIDE』を起動し、[ターミナル]メニューから[設定]を選択します。シリアルポートの選択画面が表示されますので、デバッグに使用するポートを選びます。

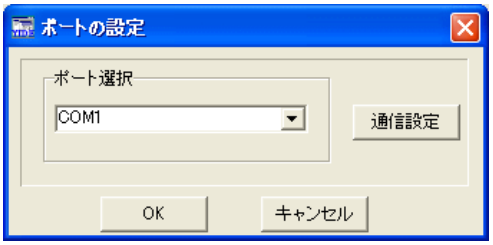


図 14　シリアルポートの選択画面

2. さらに、[通信設定]のボタンを押し、図 15 を参考に同様の設定を行ってください。終了したら[OK]ボタンを押してダイアログを閉じます。

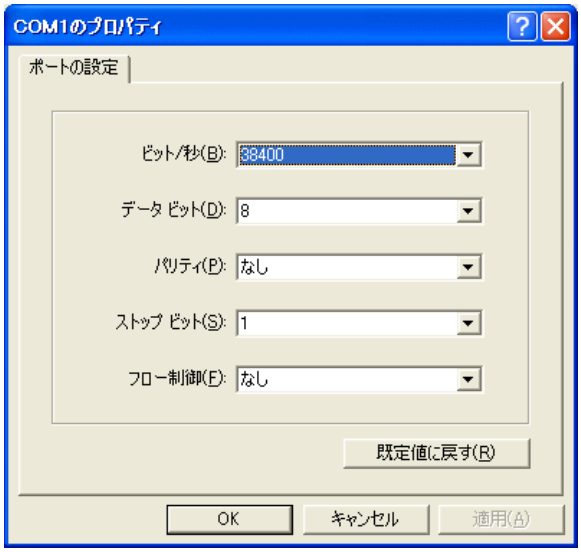


図 15　通信設定の画面

3. ［ターミナル]メニューから[表示]を選択し、ターミナル画面を表示します（図 16)。

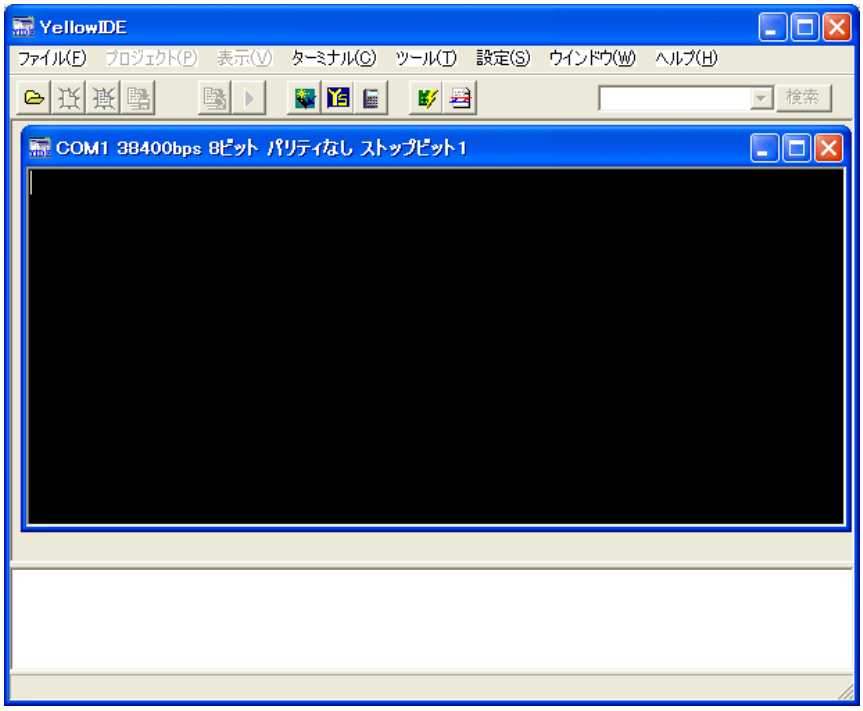


図 16　ターミナル画面

4. 製品の電源を切った状態で、「デバッグ用通信ケーブル」を用い 1. で選択したパソコンのシリアルポートと、製品の「SER1」と表示されたコネクタを接続します。

5. デバッグモニタが正常に動作している場合、製品の電源を入れるとターミナル画面に"A ｿﾞ"と表示されます。パソコンの[Enter]キーを押すと、図 17 のように"ﾁ?"が表示されます。

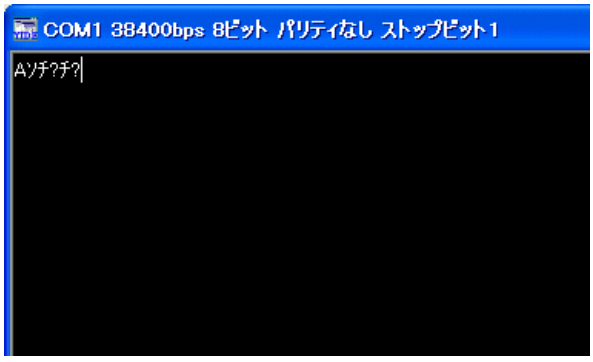


図 17　デバッグモニタの動作確認画面

ここまでで開発の準備は完了です。デバッグモニタの正常動作が確認できない場合、デバッグモニタの書き込み、シリアルポートの設定、ディップスイッチの設定、デバッグ用シリアルケーブルの接続などをもう一度見直してみてください。

□ デバッグモニタの消去

デバッグモニタを消去し、製品を出荷時の状態に戻すにはデバッグモニタの書き込みと同様の手順で「¥A0x0xProjects¥_A0x0xInit¥A0x0xInit.S」ファイルを書き込んでください。このファイルは製品固有の初期化処理を行うためのファームウェアです。

4. YellowIDE／イエロースコープによる開発作業

この章では簡単なサンプルプログラムを通して開発ツールの基本的な使用方法を説明します。『YellowIDE』や『イエロースコープ』についてのより詳しい説明はオンラインヘルプや製品付属のマニュアルを参照してください。

□ YellowIDE の起動

まず、サンプルプロジェクトを開きます。『YellowIDE』を起動し、[ファイル]メニュー→[プロジェクトを開く]をクリックし、表示されたファイル選択画面から「¥A0x0xProjects¥Sample01¥Sample01.yip」を選んで開いてください。以下は『YellowIDE』の画面の説明です。

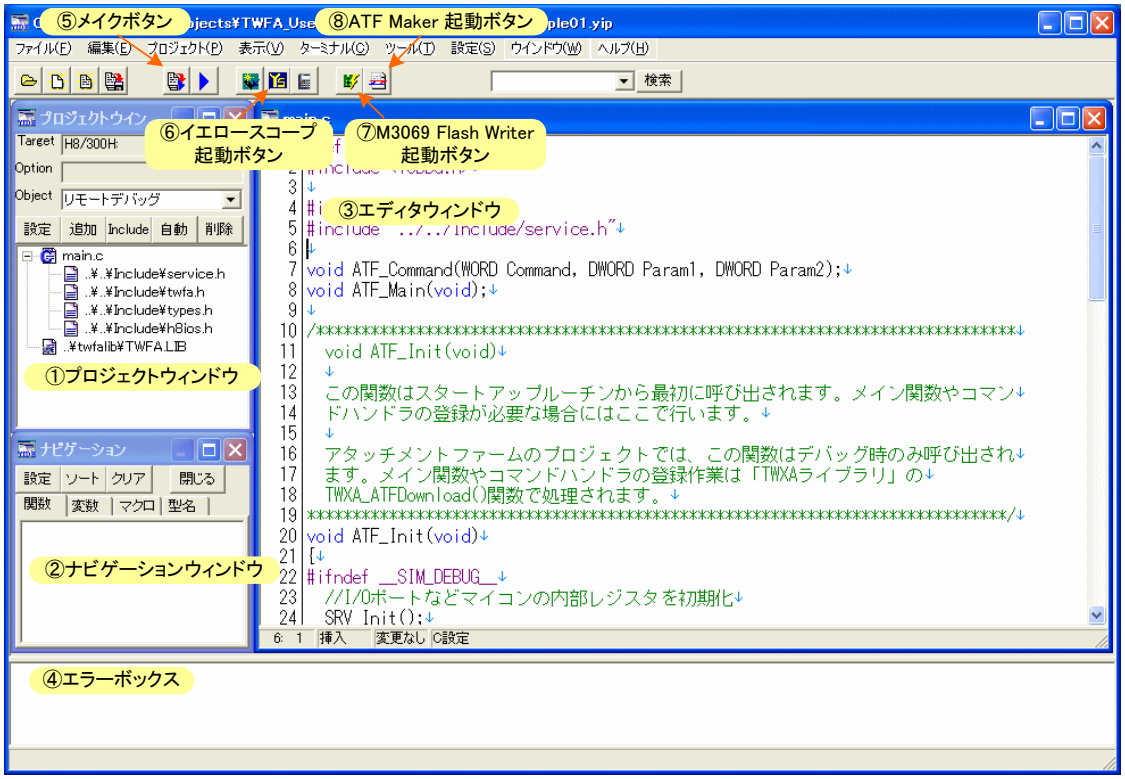


図 18　YellowIDE の画面

① **プロジェクトウィンドウ** - プロジェクトを管理します。[追加]を押してプロジェクトにソースファイル(.c ファイル)を追加することができます。また、ソースファイルを選択した状態で[Include]ボタンを押すと、そのソースファイルに関連のあるヘッダーファイル(.h)を追加することができます。[設定]を押すと[プロジェクトの設定ウィンドウ]が開きます。

- プロジェクトウィンドウでヘッダーファイルを追加しても、ソースファイル中に自動でインクルードされるわけではありません。#include による記述は必要です。

② **ナビゲーションウィンドウ** - メイクを実行すると関数や変数が表示され、素早く探すことができるようになります。表示されていない場合は[プロジェクト]メニュー→[ナビゲーションを開く]で表示することができます。

③ **エディタウィンドウ** - ソースファイルやヘッダーファイルを編集するウィンドウです。

④ **エラーボックス** - メイクを実行したときの結果を表示します。コンパイルエラーが表示されたときは、そのエラー表示をダブルクリックすると、ソースコードの該当箇所がエディタウィンドウに表示されます。

⑤ **メイクボタン** - プロジェクトをメイクします。エラーがある場合はエラーボックスに表示されます。

⑥ **イエロースコープ起動ボタン** - プロジェクト設定がデバッグのときに押すと『イエロースコープ』が起動しデバッグ画面となります。

⑦ **M3069FlashWriter 起動ボタン** - メイクして作成された出力ファイルをデバイスのフラッシュメモリに書き込む際に使用します。表示されない場合は 13 ページを参照し追加してください。

⑧ **ATF Maker 起動ボタン** - メイクして作成された出力ファイルを ATF ファイルに変換する場合に使用します。表示されない場合は 13 ページを参照し追加してください。

□ サンプルプロジェクトのメイク

作成したプログラムを実行するためには、まずメイクを実行します。

サンプルプロジェクトは既にメイク可能な状態となっています。[プロジェクトウィンドウ]の[Object]欄に"リモートデバッグ"と表示されていることを確認し、[メイク]ボタンを押してメイクを行ってください。図のようにメイク終了のメッセージが表示されるはずです。

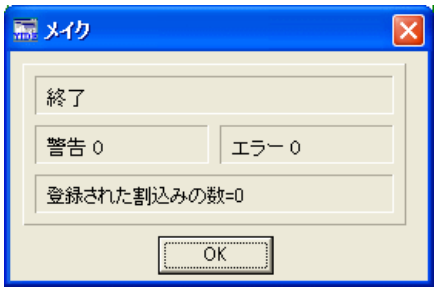


図 19　メイク終了のメッセージ

- メイクでエラーが発生する場合には、15 ページのインクルードパスの設定が正しく行われているかチェックしてください。

□ イエロースコープの起動

では、実際にプログラムを動かしてみます。『イエロースコープ』を起動する前に以下の準備を行います。

イエロースコープの起動準備

1. 16 ページのデバッグモニタの書き込み作業を完了してください。
2. 『YellowIDE』のターミナル画面が開いている場合は閉じてください。
3. デバッグ用通信ケーブルでデバイスとパソコンを接続します(デバッグモニタの動作確認時と同様の接続にします)。
4. ディップスイッチの 1 番が"ON"、2 番が"OFF"となっていることを確認します。ディップスイッチを変更した場合、デバイスの電源を入れなおします。

以上が完了したら『YellowIDE』画面の［イエロースコープ起動]ボタンを押します。図 20 は『イエロースコープ』の画面です。

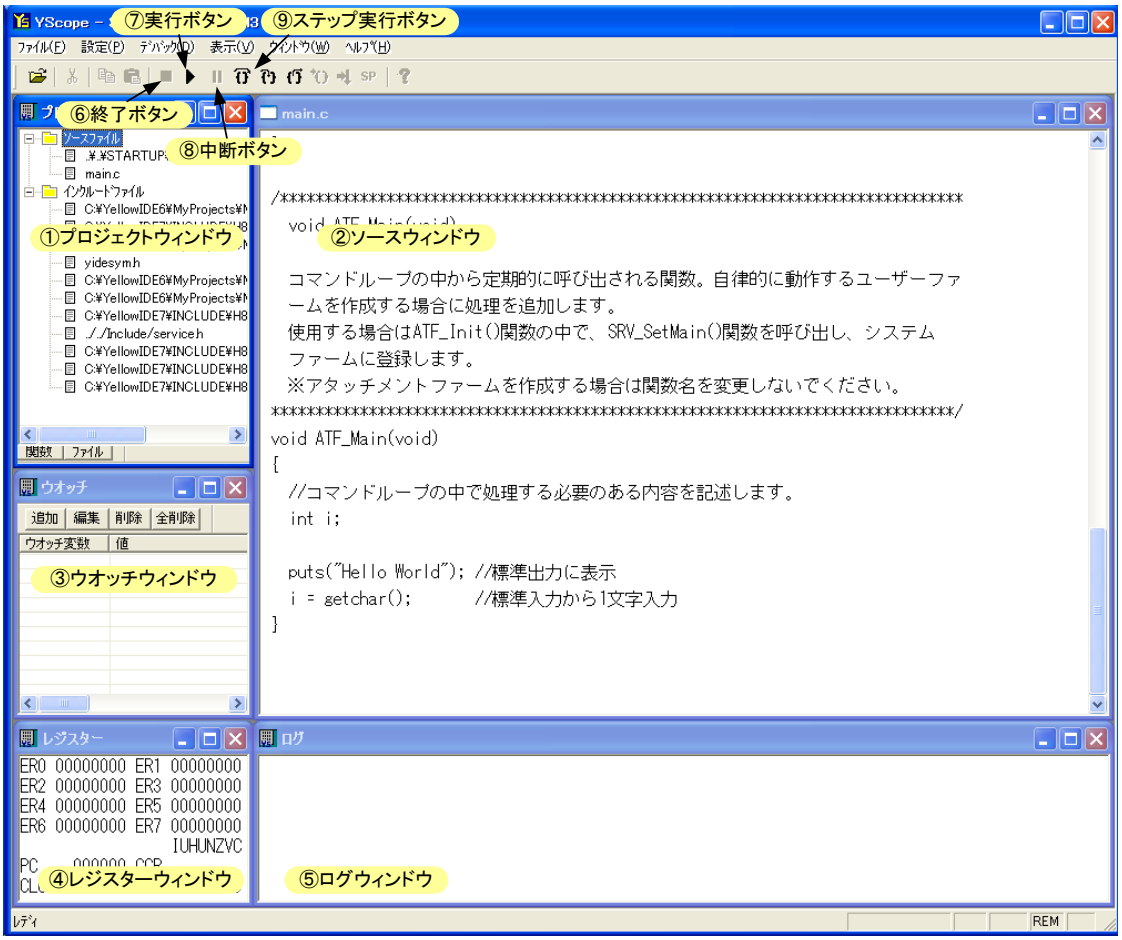


図 20　『イエロースコープ』の画面

① **プロジェクトウィンドウ** - プロジェクトのファイルを表示します。ファイル名をダブルクリックするとソースウィンドウが開きます。

② **ソースウィンドウ** - ソースファイルを表示します。

③ **ウオッチウィンドウ** - 変数の内容を確認したり、編集したりする場合に使用します。

④ **レジスターウィンドウ** - マイコンのシステムレジスタ、汎用レジスタの内容を表示します。

⑤ **ログウィンドウ** - デバッグ中のプログラムの標準出力やデバッグ出力を表示します。

⑥ **終了ボタン** - プログラムの実行を終了します。

⑦ **実行ボタン** - プログラムを実行します。キーボードの[F5]キーでも同様の操作ができます。

⑧ **中断ボタン** - 実行中のプログラムを一時中断します。

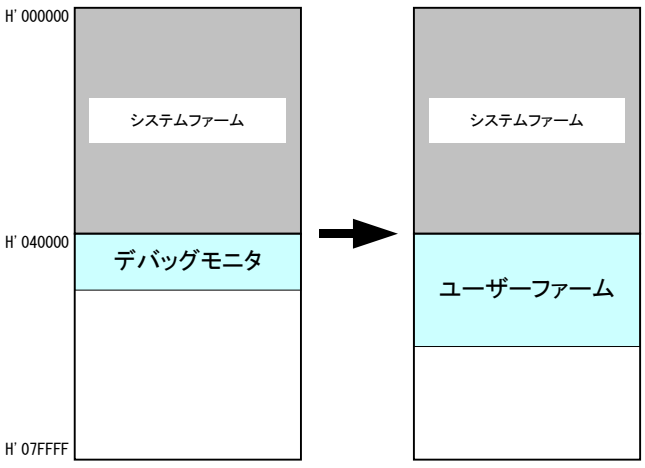
⑨ **ステップ実行ボタン** - プログラムを 1 行ずつ実行する場合に使用します。

- 図 20 の各ウィンドウが表示されない場合は、[表示]メニューから必要なウィンドウを表示することができます。

デバッグモニタの配置

デバッグモニタのプログラムもユーザーファームと同じメカニズムで動作しています。言い換えればデバッグモニタもユーザーファームの一種です。

製品にダウンロードできるユーザーファームは 1 つだけです。そのため、開発したユーザーファームを製品に書き込むと、デバッグモニタが書き換えられてしまいデバッガは使用できなくなります。再度デバッガを使用するためには、デバッグモニタを再びダウンロードする必要があります。



『YellowIDE』バージョン 7.10 以降の設定

『YellowIDE』のバージョンが 7.10 以降(『イエロースコープ』のバージョンが 3.200 以降)をご利用の場合、『イエロースコープ』の[設定]メニューから[システム設定]を選択します。「システム設定」ウィンドウが表示されますので、[システム]タブを選択し、[ダウンロード前にリセットコマンド送信]のチェックを外してください(図 21)。

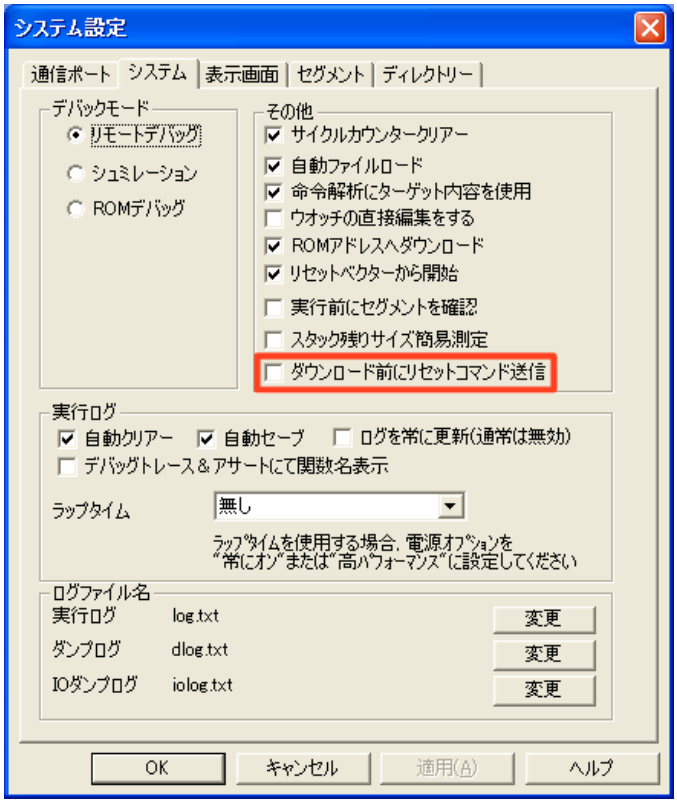


図 21　『イエロースコープ』のシステム設定

□ プログラムの実行

プログラムを実行するには[実行]ボタンを押すか、キーボードの[F5]キーを押します。図 22 のように[ログウィンドウ]に"Hello World"の文字が表示され、[ターゲットからの入力要求]ウィンドウが開けば成功です。

[デバッグ中断]ボタンを押してプログラムを中断します。ツールバーの[終了ボタン]か、キーボードから[Alt] +[F5]キーを入力しプログラムを一度終了してください。

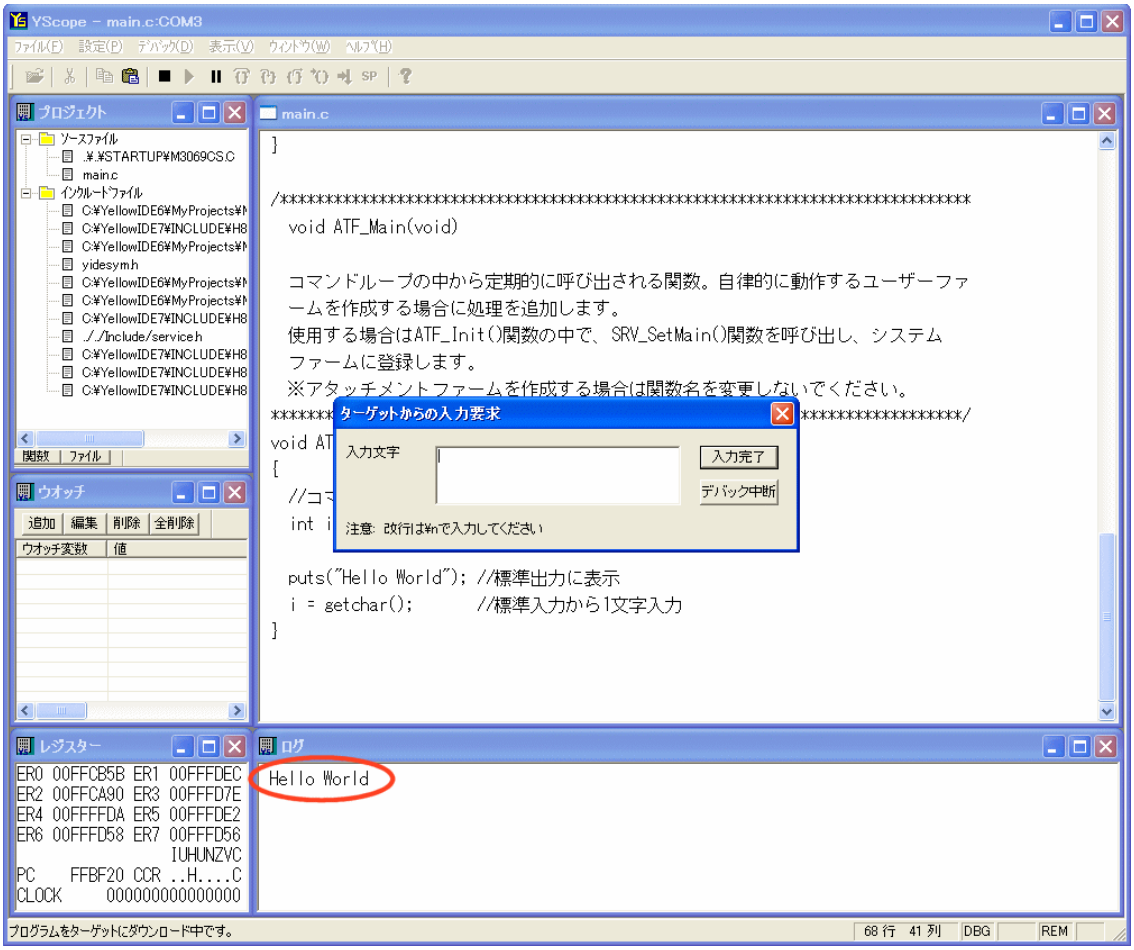


図 22　サンプルプログラムの実行結果

□ ブレークポイントの追加

ソースコード(main.c)が表示されていない場合は[プロジェクトウィンドウ]の「main.c」ファイルをダブルクリックして表示させます。デバッグを停止した状態でソースウィンドウの *puts("Hello World")* と書かれた行にカーソルを置き、キーボードの[F9]キーを押してください。

図 23 のように行が赤く表示され、ブレークポイントが設定されたことを示します。プログラムが中断しているか終了している間であれば、ソースプログラムの任意の位置にブレークポイントを追加することができます。

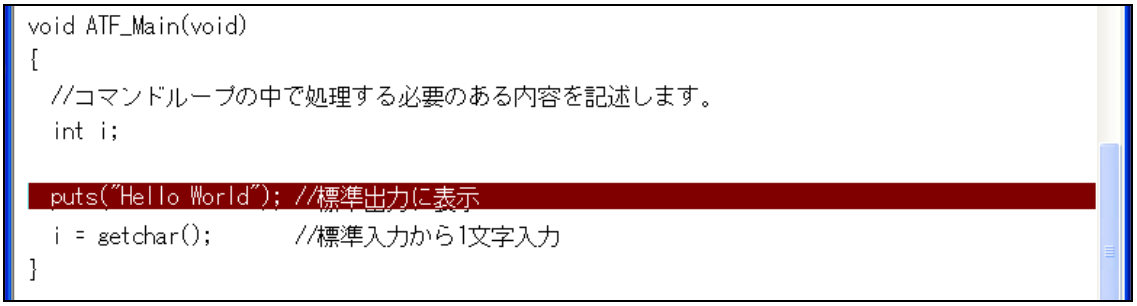


図 23　ブレークポイントの追加

□　ステップ実行

実際にプログラムがブレークポイント位置で停止するか確認します。[実行]ボタンを押してもう一度プログラムを実行します。今度はログウィンドウに何も表示されず図 24 のようにブレークポイントを追加した行が黄色く表示されたはずです。黄色の行は現在プログラムがその位置で停止していることを示します。

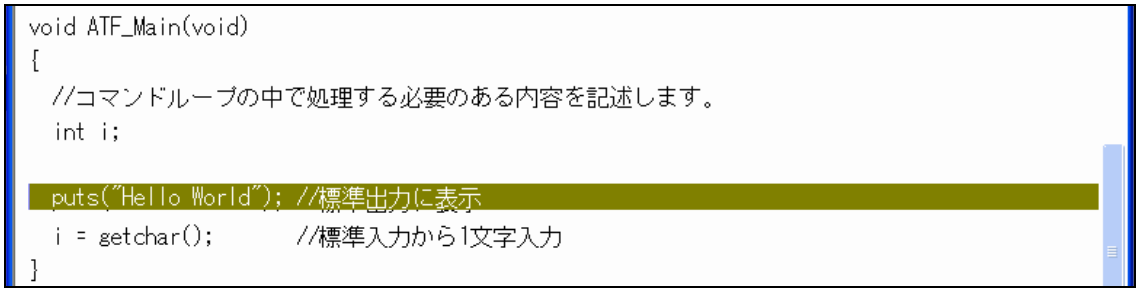


図 24　ステップ実行

プログラムをステップ実行するには、[ステップ実行]ボタンを押すか、キーボードの[F10]を入力します。黄色の行が移動しプログラムが 1 行実行されたことを示します。また、[ログウィンドウ]には実行結果として"Hello World"の文字が表示されたはずです。

- プロジェクト内にソースのある関数には、[F11]キー(トレース)で関数内にステップインすることができます。

□ ウオッチ変数の追加

ウオッチ変数を登録すると、変数の内容を希望のフォーマットで表示することができます。プログラムは中断させたままで、[ウオッチウィンドウ]の[追加]ボタンを押してください。図 25 のようなウィンドウが表示されますので"%ci"と入力し[OK]ボタンを押します。ここで入力した"%c"はウオッチ変数の表示方法を指定するもので *printf()* のフォーマット指定子と同様のものが使用できます。この例では変数 *i* をキャラクタコードと解釈して文字で表示することを指定しています。

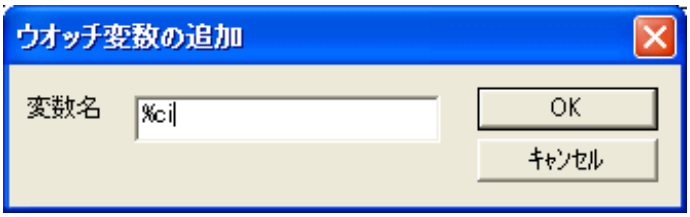


図 25　ウオッチ変数の追加

ウオッチ変数を追加したら、[F10]キーを入力するか[ステップ実行]ボタンを押します。ソース中の *getchar()* の呼び出しにより、図 26 のようなウィンドウが表示されますので、入力文字としてアルファベット 1 文字と"¥n"をタイプし[入力完了]ボタンを押してください。

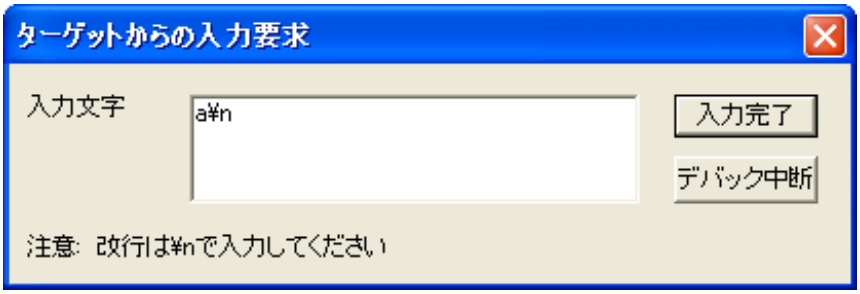


図 26　入力要求

結果として、[ウオッチウィンドウ]にタイプしたアルファベットが表示されるはずです(図 27)。

ウオッチ変数は値の位置をダブルクリックするか、[編集]ボタンを押して内容を書き換えることも可能です。

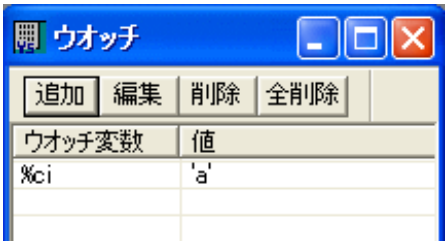


図 27　ウオッチ変数の表示

また、ソースファイル中の変数にマウスカーソルを合わせると、現在値が 10 進数で表示されます(図 28)。

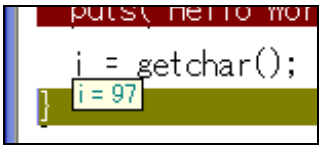


図 28　変数の値を確認

- デバッグ中は標準出力がログウィンドウに表示され、標準入力は図 26 のような入力画面から行いますが、リリースコンパイル(ROM 化)を行った際は、標準入出力はシリアル 1 と接続されます。デバッグ時と同じ方法でシリアル 1 とパソコンを接続し、パソコンでターミナルソフトを実行することで、標準出力をターミナル画面に表示し、標準入力へ入力を行うことができます。
- 上記の例でアルファベットに続いてタイプした"¥n"は標準入力における[Enter]キーの代わりをしています。デバッグ中の標準入力関数は最後に"¥n"を入力しないと正しく動作しません。
- *printf()* 関数はコンパイル後のコードサイズが大きく、通常のデバッグ環境ではコンパイルできません。後述する RLL を利用するか、デバッグトレース機能を利用してください。

□　メモリ内容の表示／編集

『イエロースコープ』の[表示]メニューから[メモリ]を選択すると、[メモリー編集]画面が表示され、マイコンのメモリ内容を表示したり編集したりが可能になります。

図 29　メモリー編集画面

[アドレス(HEX)]にアドレスを入力し、[Enter]キーを入力するとそのアドレスを表示します。[型]を選択して数値の表示方法を、[配置]を選択してバイトオーダーを変更して表示することも可能です。編集する場合には画面上の数値を直接書き換えます。

- シリアルのレシーブデータレジスタ(RDR)など、リードを行うことで状態が変わってしまうレジスタを表示すると、マイコンの動作に影響を与えてしまいますので表示するアドレスにはご注意ください。
- 上記と同じ理由で H'E00000～H'FEDFFF、H'FEE100～H'FFBF1F、H'FFFFEA～H'FFFFFF の範囲のアドレス空間(内蔵 RAM や I/O がマップされていない領域)を表示しないでください。

□ RLL を利用したデバッグ

RLL(Rom Link Library)は『Yellow IDE』で提供される機能で、デバッグ中のプログラムの一部を予めフラッシュメモリにダウンロードすることを可能にします。

デバッグ中のプログラムはマイコンの内蔵 RAM 上で実行されますが、ユーザーが利用可能な内蔵 RAM はユーザーメモリの 10K バイトの領域しかありません。多くの場合、この 10K バイトの領域だけではユーザーファームの開発に十分ではありません。

RLL 機能を利用すると、標準ライブラリなどのデバッグの必要がないプログラム部分を予めフラッシュメモリ上にダウンロードしておくことができます。デバッグ対象となるプログラム部分は RAM にダウンロードされ、必要なときはフラッシュメモリ上の関数を呼び出して利用します。RAM に配置する必要があるのはプログラムの一部だけですので、小さな RAM 容量でも開発を進めることが可能になります(図 30)。

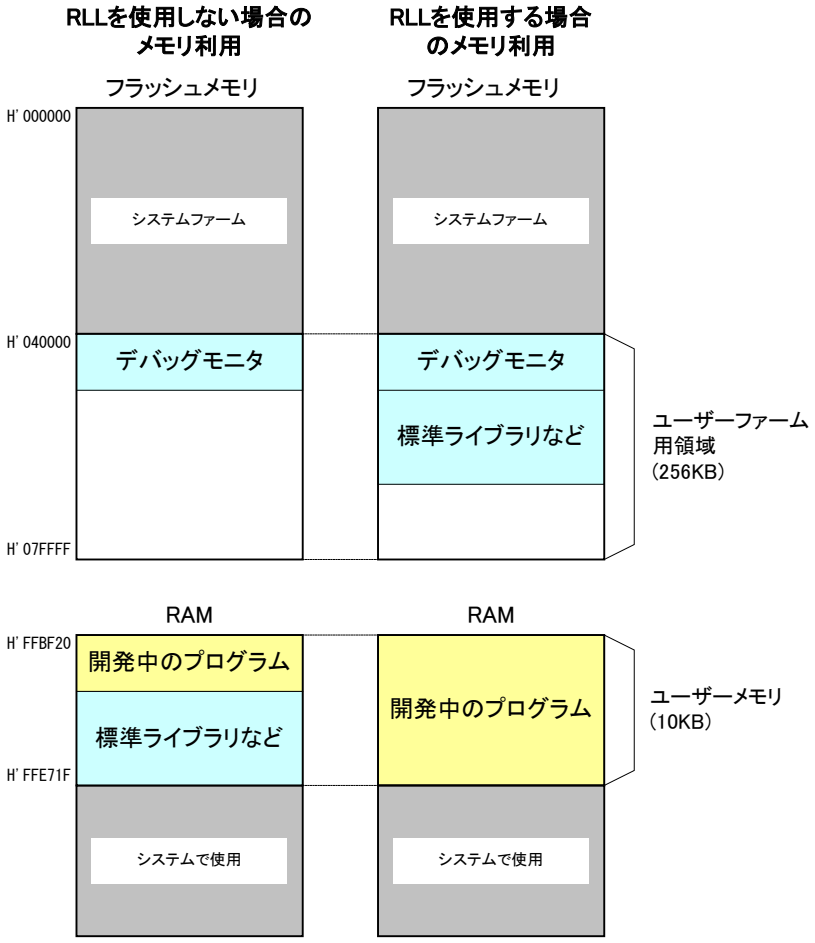


図 30　RLL を使用した場合のメモリ利用

RLL の効果を確認するため、サンプルプログラムの容量を予め確認しておきます。『イエロースコープ』が起動している場合は終了し、『YellowIDE』の画面を表示してください(「Sample01.yip」が開かれていない場合は、改めて開いてください)。

[表示]メニューから[マップファイル(グリッド)]を選択します。[マップファイル]という

ウィンドウが開きますので、ウィンドウ上部の[メモリ使用量表示]というボタンを押します。図 31 のような画面が表示され、メモリの使用量を調べることができます。画面には「ROM 使用量」と「RAM 使用量」という名称で表示されますが、デバッグ時にはどちらも RAM に配置されます。この例では合計 5,888 バイト [2]が RAM 上に配置されることを示しています。

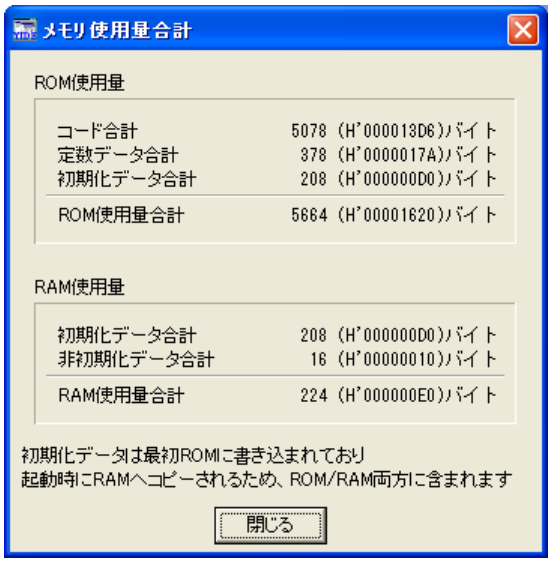


図 31　RLL を使用しない場合のメモリ使用量

[2] コンパイラバージョンの違いなどにより変化する場合があります。

RLL の利用手順

1. [プロジェクトウィンドウ]の[設定]ボタンを押します。
2. [プロジェクトの設定]ウィンドウが開きますので[RLL]タブをクリックします。
3. 図 32 のような画面となりますので[ROM リンクライブラリを使用する]にチェックを入れます。念のため他の項目も画面のように設定されているか確認し、[OK]ボタンを押します。

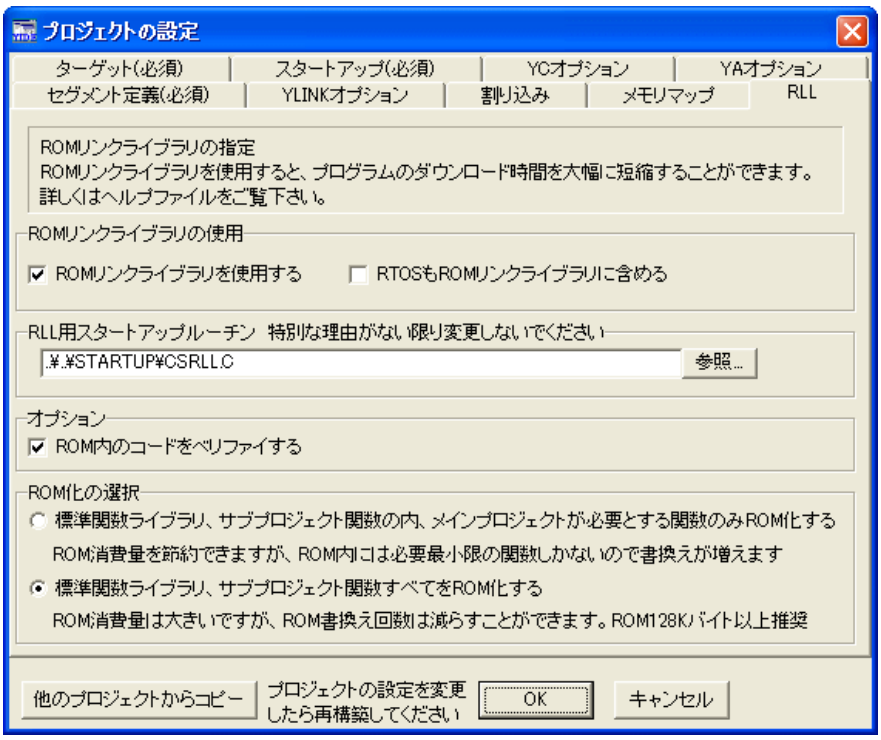


図 32　RLL の設定画面

4. [ファイル]メニューから[サブプロジェクトを開く]を選択してください。ファイルの選択画面が表示されますので「¥A0x0xProjects¥_TWMON¥REM_MON.YIP」を開きます。
5. サブプロジェクトウィンドウ(図 33)が表示されますので[Object]欄が"ROM 化(S)"になっていることを確認して[メイク]ボタンを押します。警告[3]が 2 つ表示されますが無視して構いません。ここまでの作業で「¥A0x0xProjects¥_TWMON」フォルダに「REM_MON.S」というファイルが作成されます。ファイル名は**デバッグモニタの書き込み**(16 ページ)で書き込んだものと同じですが、新しく作成したファイルにはデバッグモニタの機能に加えて、標準の C ライブラリが組み込まれています。

[3] *_Heapbase* と *_struct_ret* に関する警告が表示されます。サブプロジェクトの設定でヒープ領域と、関数の戻り値に構造体を使用する場合のメモリ領域を確保していないことが原因の警告ですが、サブプロジェクト内でこれらの領域を確保するとメインのプロジェクトでサイズの調整ができなくなるため、却って不都合が生じます。

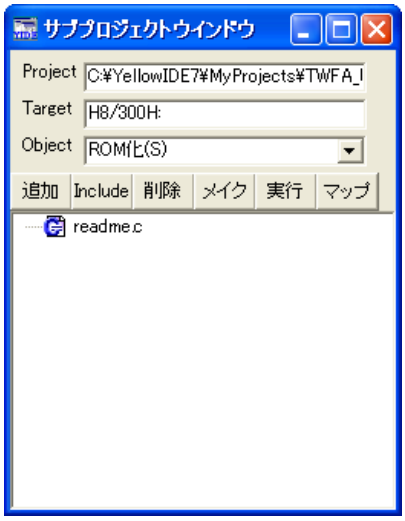


図 33　サブプロジェクトウィンドウ

6. [M3069FlashWriter 起動]ボタンを押して「M3069FlashWriter」を起動します。ダウンロードファイル名が正しくありませんので[参照]ボタンを押し「¥A0x0xProjects¥_TWMON¥REM_MON.S」を選択してください。

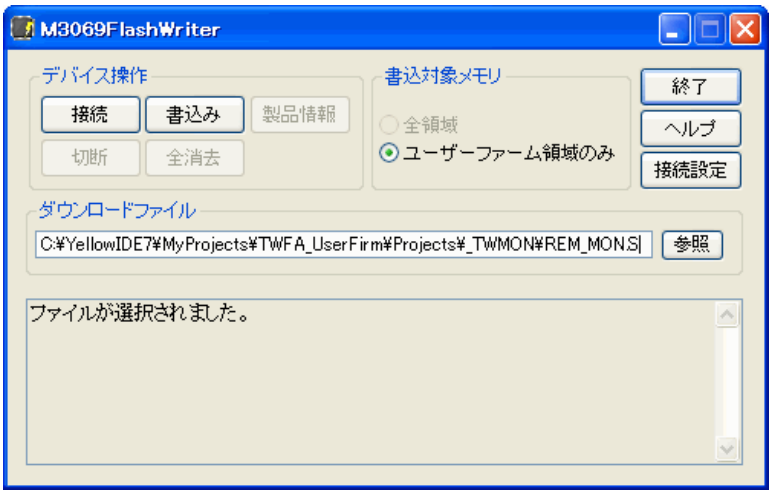


図 34　「REM_MON.S」の書き込み

7. デバイスをフラッシュ書き換えモード(ディップスイッチ 2 番を"ON")で再起動し、[書込み]ボタンを押してファイルを書き込みます。
8. 終了したらディップスイッチ 2 番を"OFF"の状態に戻し、デバイスを再起動します。
9. 次にメインプロジェクトの[プロジェクトウィンドウ]で[Object]がリモートデバッグになっていることを確認し、ツールバー上の[メイク]ボタンか、キーボードの[F9]を押してください。メインプロジェクトのプログラムがコンパイルされます。

以上で RLL を利用したデバッグ環境の作成は終了です。前の例と同様に『イエロースコープ』を使用してデバッグが可能になっているはずです。

RLL を利用したことでメインプロジェクトのプログラムサイズがどの程度になったかを確認します。[表示]メニューから[マップファイル(グリッド)]を選択します。表示されたマッ

プファイル画面から[RLL 除外メモリ使用量]ボタンを押してください。図 35 のような画面が表示されます。標準関数を全てフラッシュメモリに格納する設定としているため、RAM 使用量として表示される部分は増えていますが、コード部分の使用量は大幅に減り、トータルでも 5,136 バイトと 30 ページの例より少なくなっていることがわかります。

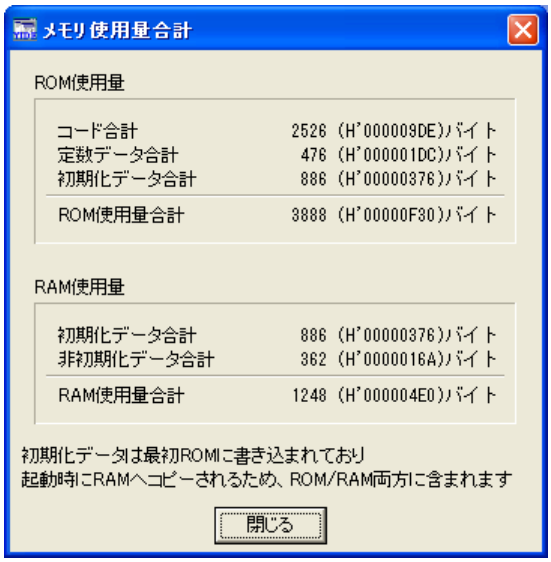


図 35　RLL を利用した場合のメモリ使用量

既にデバッグの終了した「.lib」ファイルや「.c」ファイルがあれば、サブプロジェクトに追加することでＣの標準ライブラリ同様フラッシュメモリに予めダウンロードしておくことが可能になります。

メインプロジェクトからサブプロジェクトにファイルを移動するには、移動したいファイルをメインプロジェクトの[プロジェクトウィンドウ]からドラッグし、サブプロジェクトの[プロジェクトウィンドウ]にドロップします。

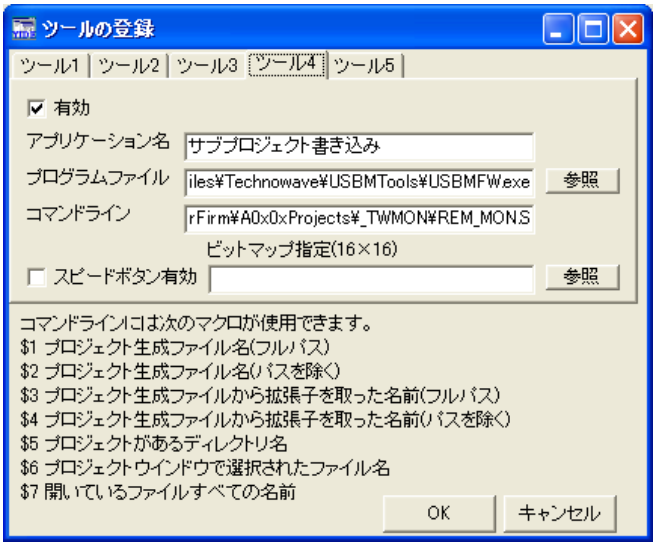
サブプロジェクトの書き込み

サブプロジェクト書き込み用のコマンドを『YellowIDE』にツール登録しておくと、書き込みが必要なときに簡単に呼び出せて便利です。

下図はサブプロジェクト書き込みコマンドの登録例です。登録は[ツール]メニューの登録から行うことができます。

[プログラムファイル]には「M3069FlashWriter」へのパスを入力します。既に登録されている「M3069 FlashWriter」の[プログラムファイル]からコピーすれば簡単です。

[コマンドライン]には「REM_MON.S」ファイルのパスを入力しておきます。



5. ユーザーファームの作成

□ ユーザーファームの構成

図 36 はユーザーファームを実行した場合の処理の流れです。ユーザーファームは大きく分けてスタートアップルーチン、*ATF_Init()* 、*ATF_Main()* 、*ATF_Command()* 、割り込みハンドラで構成されます。

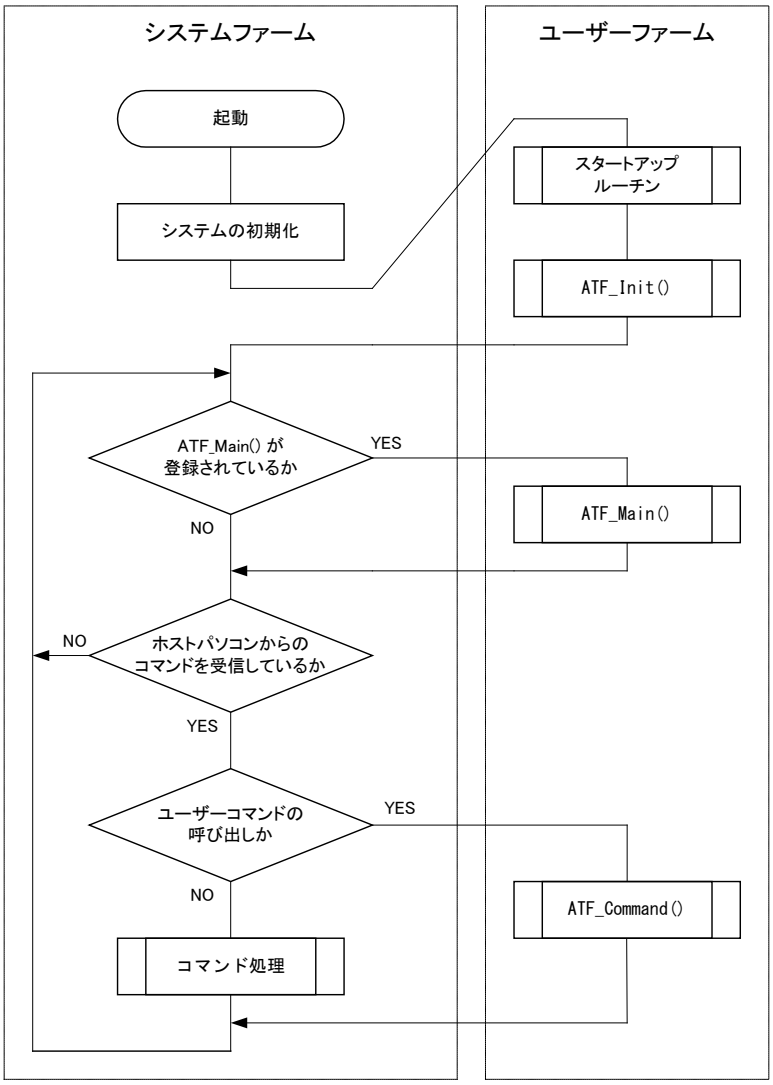


図 36　ユーザーファーム実行時の処理の流れ

スタートアップルーチン

グローバル変数の初期化やヒープ領域の初期化などプログラム実行に必要な準備を行います。ユーザーファーム開発では予め用意されたファイルを使用しますので新たにプログラムする必要はありません。

ATF_Init()

スタートアップルーチンから呼び出される初期化用の関数です。この関数内で入出力端子などのハードウェアの初期化、ネットワーク機能の初期化、*ATF_Main()* 関数の登録、

ATF_Command() 関数の登録、割り込みハンドラの登録などを行います。

ATF_Main()
システムファームから定期的に呼び出される関数です。常に実行する処理がある場合はこの中に記述します。常時実行するような処理が無い場合には無くても構いません。

ATF_Command()
ホストパソコンから *TWXA_ATFUserCommand()* で送信されたユーザーコマンドを処理します。ユーザーコマンドが不要な場合には無くても構いません。

割り込みハンドラ
図 36 のフロー図にはありませんが、ユーザーファームの構成要素の 1 つです。割り込み要因が発生したときの処理を記述します。主な割り込み要因はタイマ割り込みと外部割込みです。割り込みを処理しない場合は必要ありません。

- アタッチメントファームを作成する場合、スタートアップルーチンや *ATF_Init()* 関数が使用されるのはデバッグ時だけです。実際に ATF ファイルをダウンロードして使用する場合には、主な初期化処理は終了し、既にシステムが起動しているためです。アタッチメントファームに必要なグローバル変数などの初期化、*ATF_Main()* や *ATF_Command()* の登録は ATF ファイルのダウンロードルーチンによって自動的に処理されます。割り込みハンドラの登録など特別な初期化作業が必要な場合は、初期化を行うためのコマンドを別途用意してください(49 ページ参照)。

図 36 からもわかるようにシステムファームとユーザーファームは 1 つのタスクの中で動作しています。途中で処理を止めてしまうとシステム全体が停止しますのでご注意ください。通常の C 言語のプログラムでは *main()* 関数の中でループし、プログラム終了まで戻らないように記述しますが、*ATF_Main()* 関数をこのように記述すると、システムファームに処理が渡されずホストパソコンからのコマンドに応答できなくなり、LAN デバイスのネットワークに関する処理も停止します。

同じ理由からユーザーファームのデバッグ中にプログラムが中断状態になっていると、ホストパソコンからの接続処理が失敗してしまいます。ホストパソコンから接続する必要があるときは、まずデバッグ中のプログラムを実行状態にし、その後パソコン上のプログラムから接続処理を行ってください。

□ ユーザーファームのサンプルプログラムの実行

まず、サンプルプログラムを通して、ユーザーファームの構造と基本的なプログラミングを確認します。『YellowIDE』の[ファイル]メニュー→[プロジェクトを開く]をクリックします。ファイル選択画面が表示されますので、「¥A0x0xProjects¥Sample02¥Sample02.yip」を選択して開いてください。

[プロジェクトウィンドウ]の[Object]欄に"リモートデバッグ"と表示されていることを確認し、[メイク]ボタンを押してメイクを行ってください。前章と同じ手順で、パソコンとデバイスをデバッグ用通信ケーブルで接続後、『イエロースコープ』を起動しプログラムを実行します。成功するとログウィンドウにプログラム開始からの経過秒数が表示されます(図 37)。

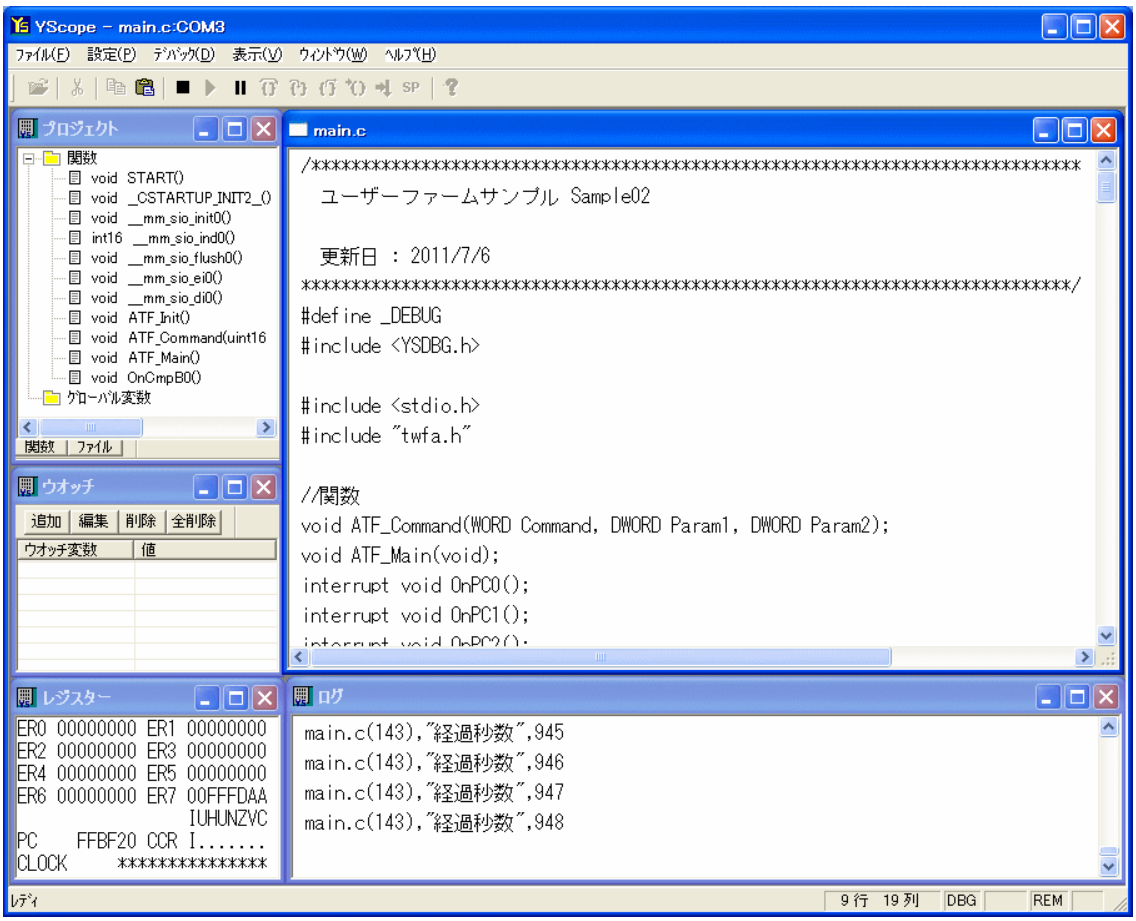


図 37 Sample02 実行画面

次に実行中のユーザーファームに対して、ユーザーコマンドを送信してみます。一旦『YellowIDE』の画面に戻り、[ATF Maker 起動]ボタン、または、[ツール]メニューから[ATF Maker]を選択します。

「ATF Maker」が起動したら画面上部から[テスト]タブを選択します(図 38)。

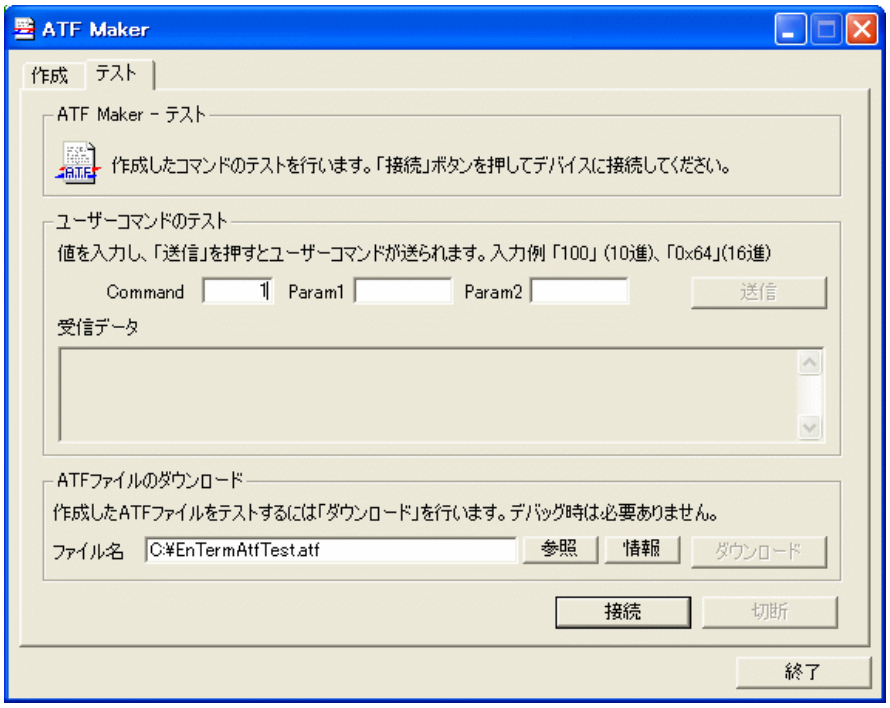


図 38　「ATF Maker」によるユーザーコマンドのテスト

[接続]ボタンを押しデバイスとの接続を行います[4]。接続に成功したら[Command]欄に"1"と入力し、[送信]ボタンを押してください。成功すると『イエロースコープ』のログウィンドウに送信されたコマンド内容が表示され、経過秒数の表示が停止します(図 39)。

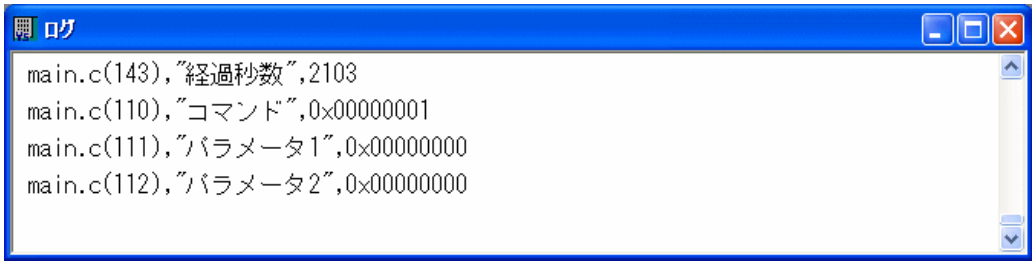


図 39　コマンドによるログ表示

- 「ATF Maker」のユーザーコマンドのテスト機能は *TWXA_ATFUserCommand()* 関数によるコマンド送信をシミュレートしています。パソコン上のアプリケーションプログラムを開発する場合は、プログラム内で *TWXA_ATFUserCommand()* 関数を呼び出すことでテストと同じ結果を得ることができます。
- 「¥A0x0xProjects¥HostSample」フォルダの「HostSample.sln」には、ユーザーコマンド送信のサンプルプログラム「HostSample01_MFC」が収められています。

[4] デバイスはユーザーズマニュアルに従って、ドライバやライブラリのインストールが終了し、サンプルプログラム等から接続可能な状態になっていることが必要です。

次に再び「ATF Maker」の画面に戻り、[Command]欄に"3"と入力し[送信]ボタンを押してください。[受信データ]データ欄にデバイスからの応答データが表示されます(図 40)。

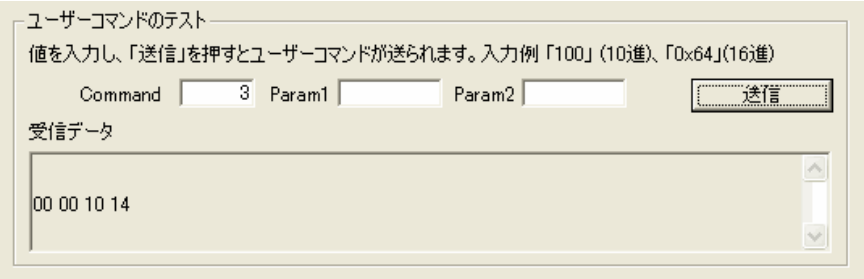


図 40　ユーザーコマンドに対する応答表示

- 応答データの内容は経過秒数を 16 進表記したものですが、デバイス内のマイコンがビッグエンディアンとなっているため、一般のパソコンとはバイトオーダーが逆になっています。デバイスとの間でデータを送受信する場合、*short* や *long* などの複数バイトからなるデータの順序に注意してください。
- TWXA ライブラリで提供される関数は、(メモリ読み出しなどの)生のデータを送受信する場合を除き、ライブラリ内部でデータ順を変換しているため、バイトオーダーを気にする必要はありません。

[Command]欄に"2"と入力し「送信」ボタンを押すと、デバイスは経過秒数の表示を再開します。

バイトオーダー(エンディアン)

複数バイトからなる数値データを扱う場合、システムにより下位バイトから並べるか、上位バイトから並べるかのルールが異なっています。このルールのことをバイトオーダーといい、下位バイトから並べるものをリトルエンディアン、上位バイトから並べるものをビッグエンディアンと呼びます。下の図は同じ"0x12345678"という数値を 0 番地のメモリに格納した場合のバイトオーダーによる違いを示しています。

リトルエンディアンの場合

アドレス	0	1	2	3
データ	0x78	0x56	0x34	0x12

ビッグエンディアンの場合

アドレス	0	1	2	3
データ	0x12	0x34	0x56	0x78

一般的に使用されているパソコンはリトルエンディアンを採用していますが、製品に搭載されているマイコンや、ネットワークプロトコルではビッグエンディアンを採用しているため、数値の取り扱いには注意が必要です。

パソコン用のプログラムではバイトオーダーの変換関数はネットワーク用のライブラリとして提供されています。Winsock では *htons()* や *htonl()* といった関数を使用することが可能です。また、.NET では *IPAddress* クラスに *NetworkToHostOrder()* というメソッドが用意されています。

□ ユーザーファームサンプル(Sample02)のソースコード

リスト 1 はユーザーファームの初期化を行う *ATF_Init()* 関数の内容です。

リスト 1　ATF_Init() 関数

```
void ATF_Init(void)
{
  DWORD dwFreq;

#ifndef __SIM_DEBUG__
  //I/O ポートなどマイコンの内部レジスタを初期化...①
  TWFA_Initialize(TWFA_INIT_ALL);

  //X-A0x0x デバイス固有機能の初期化...②
  //TWXA ライブラリ関数(TWXA_ADStartFastSampling(),TWXA_ADStartAutoSampling())を使用する場合は
  //必ず呼び出します
  //TWFA_A0x0xInit(0);

  //LAN デバイスのみネットワークが初期化される...③
  SRV_LanmInit(LANMM_ENABLE_CONTROL | LANMM_ENABLE_LIST);

  //メイン関数を登録...④
  SRV_SetMain(ATF_Main); //必要な場合はここでメイン関数を登録します

  //コマンドハンドラを登録...⑤
  SRV_SetCommand(ATF_Command); //必要な場合はここでコマンドハンドラを登録します

  //割り込みハンドラの設定...⑥
  SRV_EnableInt(SRV_INT_DISABLE); //割り込み禁止
  //g_TimerIntA[0] = SRV_SetVect(VECT_TIMER0_A, OnCmpA0);
  //g_TimerIntA[1] = SRV_SetVect(VECT_TIMER1_A, OnCmpA1);
  //g_TimerIntA[2] = SRV_SetVect(VECT_TIMER2_A, OnCmpA2);

  //チャンネル 0 のコンペアマッチ B にハンドラ登録
  g_TimerIntB[0] = SRV_SetVect(VECT_TIMER0_B, OnCmpB0);

  //g_TimerIntB[1] = SRV_SetVect(VECT_TIMER1_B, OnCmpB1);
  //g_TimerIntB[2] = SRV_SetVect(VECT_TIMER2_B, OnCmpB2);
  //g_TimerIntOvf[0] = SRV_SetVect(VECT_TIMER0_OVF, OnOvf0);
  //g_TimerIntOvf[1] = SRV_SetVect(VECT_TIMER1_OVF, OnOvf1);
  //g_TimerIntOvf[2] = SRV_SetVect(VECT_TIMER2_OVF, OnOvf2);
  SRV_EnableInt(SRV_INT_ENABLE); //割り込み許可

  //割り込みの許可(選択されたチャンネルは許可され、
  //選択されないチャンネルは禁止されます)...⑦
  //TWFA_TimerEnableIntA(TWFA_TIMER_BIT0 | TWFA_TIMER_BIT1 | TWFA_TIMER_BIT2);

  //タイマチャンネル 0 のコンペアマッチ B の割り込みを許可
  TWFA_TimerEnableIntB(TWFA_TIMER_BIT0 /*| TWFA_TIMER_BIT1 | TWFA_TIMER_BIT2*/);

  //TWFA_TimerEnableIntOvf(TWFA_TIMER_BIT0 | TWFA_TIMER_BIT1 | TWFA_TIMER_BIT2);

  //タイマの初期化...⑧
  TCR16(0) = 0x40; //コンペアマッチ B でクリア
  dwFreq = 100;
  TWFA_TimerSetPwmQ16(0, &dwFreq, NULL, NULL);
  TWFA_TimerStart(TWFA_TIMER_BIT0);
#endif
}
```

① マイコンの初期設定を行うために *TWFA_Initialize()* を呼び出しています。デバイスの初期化については 52 ページを参照してください。

② TWXA ライブラリ関数(TWXA_ADStartFastSampling()、TWXA_ADStartAutoSampling())を使用する場合は、TWFA_A0x0xInit() を必ず呼び出してください。デバイスの初期化については 52 ページを参照してください。

③ LAN デバイスの初期設定を行っています。LAN デバイスが使用できる通信チャンネル数はシステムが使用するものも含めて 4 チャンネルまでです。用途により不足する場合には、初期化オプションを変更し、システムが使用するチャンネルを制限できます。この関数呼び出しは USB デバイスでは無視されますので削除する必要はありません。

④ *ATF_Main()* 関数を登録しています。この登録作業を行うことで *ATF_Main()* 関数が定期的に呼び出されるようになります。

⑤ *ATF_Command()* 関数を登録しています。この登録作業を行うことで、ユーザーコマンドの通知を受けることができます。

⑥ 独自の割り込み処理を行う場合には、割り込みベクタにハンドラとなる関数を登録する必要があります。この例では 16 ビットタイマ 0 チャンネルのコンペアマッチ B という割り込みに関数を登録しています。割り込みについての詳細は後述します。

⑦ 必要な割り込みに許可を与えています。ベクタに関数を登録しただけでは割り込みは発生しません。ここでは⑤で登録を行った 16 ビットタイマ 0 チャンネルのコンペアマッチ B 割り込みを許可しています。

⑧ 登録した割り込みが希望の周期で発生するようにタイマの設定と、動作開始を行っています。ここでは 100Hz の周波数で割り込みが発生するように設定しています。このようにタイマを使って一定周期の割り込みを発生させたい場合はコンペアマッチ B に割り込みを登録し、*TWFA_TimerSetPwm()* 関数や *TWFA_TimerSetPwmQ16()* 関数で周期設定を行うことができます。

リスト 2 は *ATF_Main()* 関数と割り込みハンドラ関数です。*ATF_Main()* 関数内では経過秒数の表示を行います。

OnCmpB0() 関数は、10msec 周期毎に発生する 16 ビットタイマ 0 チャンネルのコンペアマッチ B による割り込みで呼び出され、1 秒ごとにグローバル変数をインクリメントします。

リスト 2　ATF_Main() 関数とタイマ割り込みのハンドラ関数

```
void ATF_Main(void)
{
  static int dwPreSec;

  if(!g_flgStop){
    if(dwPreSec != g_dwSec){
      dwPreSec = g_dwSec;
      //デバッガに経過秒数を表示...①
      DEBUG_TRACE0_MSG("経過秒数", g_dwSec);
    }
  }
}

interrupt void OnCmpB0()
{
  static int cnt = 0;

  //割り込みフラグのクリア(必須)...②
  TISRB &= ~TWFA_TIMER_BIT0;

  cnt++;
  if(cnt >= 100){
    cnt = 0;
    g_dwSec++; //1 秒経過毎にインクリメント
  }
}
```

① 経過秒数の表示には『イエロースコープ』のデバッグトレースの機能を利用して表示しています。デバッグトレースは *printf()* よりも軽量ですので複雑な書式設定が必要ない場合はこちらを推奨します。詳しくは『イエロースコープ』のオンラインマニュアルで「デバッグ支援機能」の章を参照してください。

② 割り込み関数では必ず対応する割り込みフラグをクリアします。フラグをそのままにしておくと、割り込み関数から戻ったときに、残ったフラグにより再び同じ割り込みが発生してしまいます。予め用意された割り込み関数のスケルトンコードには、対応する割り込みフラグのクリア処理が書かれていますのでこれを消さないようにしてください。

リスト 3 は *ATF_Command()* 関数の処理です。ここではホストパソコンから受け取ったコマンドをデバッガに表示し、経過秒数の表示開始および停止、経過秒数の送信の各コマンドに対応した処理を行っています。

リスト 3　ATF_Command() 関数

```
void ATF_Command(WORD Command, DWORD Param1, DWORD Param2)
{
  //ユーザーコマンドに対応する処理を記述します。

  //コマンドをデバッガに表示
  DEBUG_TRACE0_MSG_HEX("コマンド", Command);
  DEBUG_TRACE0_MSG_HEX("パラメータ 1", Param1);
  DEBUG_TRACE0_MSG_HEX("パラメータ 2", Param2);

  //コマンド処理
  switch(Command){
  case 1:
    g_flgStop = TRUE;
    break;
  case 2:
    g_flgStop = FALSE;
    break;
  case 3:
    SRV_Transmit(&g_dwSec, 4, 1); //応答の送信...①
    break;
  }
}
```

① *SRV_Transmit()* 関数によって応答データを送信しています。ホストパソコン側はここで送信された全てのデータを確実に取り出す必要があります。受信バッファ内に不要なデータを残しておくと、次にデバイスに対して操作を行ったときに誤動作の原因となります。

□ ユーザーファームの書き込み

次にユーザーファームをフラッシュメモリに書き込む手順について説明します。サンプルプロジェクトは「Sample02.yip」を使用します。『YellowIDE』を表示し、サンプルが開いていない場合は［ファイル］メニューの［プロジェクトを開く］を選択し、「¥A0x0xProjects¥Sample02¥Sample02.yip」を開いてください。

- フラッシュメモリにユーザーファームを書き込むと、デバッグモニタが消去され『イエロースコープ』でのデバッグができなくなります。再度、デバッガを使用する場合には「デバッグモニタ」の書き込みが必要になります。
- 搭載マイコンのフラッシュメモリの書き換え保証回数は 100 回です。通常のご使用では、デバッグ作業が完了した段階での書き込みを推奨します。

1. ［プロジェクトウィンドウ］の［Object］を"ROM 化(S)"に変更します。

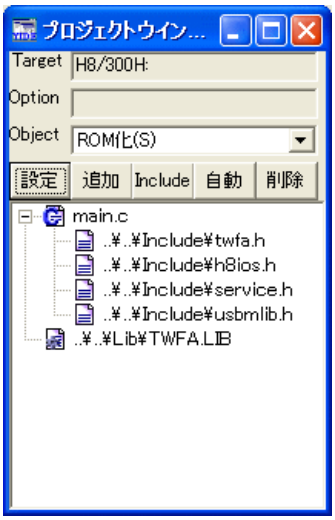


図 41　［Object］を"ROM 化(S)"に変更

2. ［メイク］ボタンを押してプログラムをコンパイルします。
3. デバイスのディップスイッチの 2 番を"ON"にし再起動し、"フラッシュ書き換えモード"に設定します。
4. ［M3069FlashWriter 起動］ボタンを押して「M3069FlashWriter」を起動します。

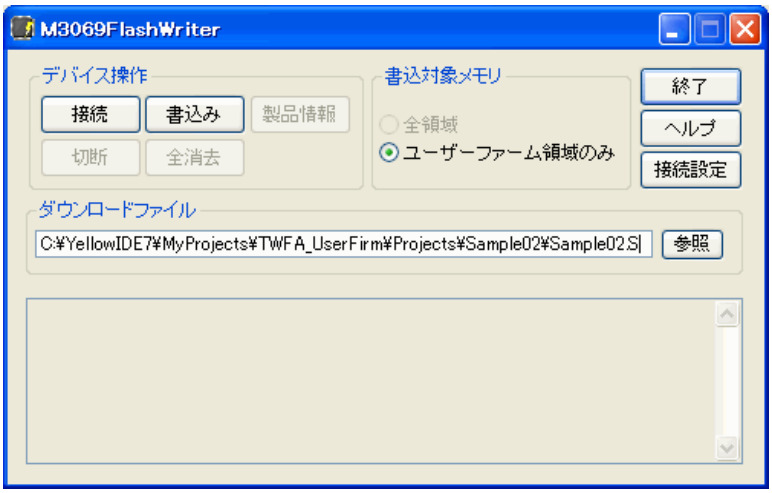


図 42　ユーザーファームの書き込み

5. ［書込み］ボタンを押してユーザーファームを書き込みます。
6. デバイスのディップスイッチ 2 番を"OFF"とし再起動します。

以上で書き込み作業は終了です。ディップスイッチを設定して起動すると直ちにユーザーファームが実行されます。デバッグ用ではありませんのでデバッグトレースによる表示は行われませんが、37 ページと同様にユーザーコマンドの"3"を送信することで応答が返るはずです。

□ アタッチメントファームの作成と実行

次にアタッチメントファームの作成方法を説明します。アタッチメントファームとして利用するには、作成したユーザーファームから、拡張子が「.atf」の ATF ファイルに変換する必要があります。

ATF ファイルにはユーザーファームの実行コードに加えて、プログラムを RAM 上のどの位置に配置すれば良いかといったダウンロードに関する情報も含まれています。ホストパソコン上のプログラムでは *TWXA_ATFDownload()* の引数に ATF ファイルのパスを渡すだけで、デバイス上の適切な位置にプログラムがダウンロードされ実行が開始されます。

1. 『YellowIDE』から、サンプルプロジェクトとして「A0x0xProjects¥Sample03¥Sample03.yip」を開きます。
2. [プロジェクトウィンドウ]の[Object]欄が"RAM へダウンロード(S)"となっていることを確認し、[メイク]ボタンを押してください。

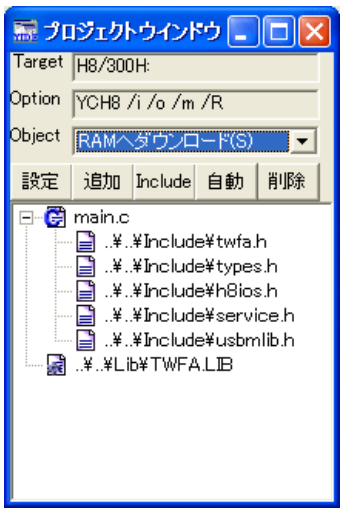


図 43　ATF ファイル作成時のターゲット選択

- RLL は使用しない設定になっている必要があります。[設定]ボタン→[RLL]タブを押し、[ROM リンクライブラリを使用する]のチェックを外してください。

3. [ATF Maker 起動]ボタンを押して、「ATF Maker」を起動します。
4. 必要があれば[管理情報]の各項目を入力します。[要求するファームウェアバージョン]以外の項目は、情報としてファイルに埋め込まれますが動作には影響しません。[要求するファームウェアバージョン]に必要なシステムファームのバージョンを入力しておくと、ATF ファイルをダウンロードする際に実際のシステムファームのバージョンがチェックされます。システムファームが指定よりも古いバージョンの場合、*TWXA_ATFDownload()* 関数はエラーを返し、ダウンロードは失敗します。

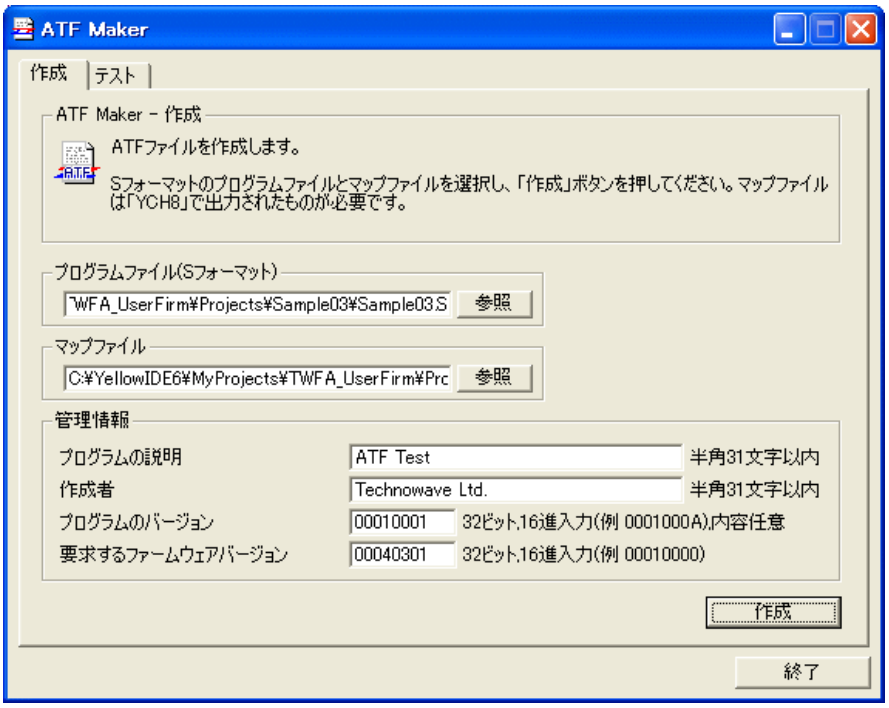


図 44　「ATF Maker」のファイル作成画面

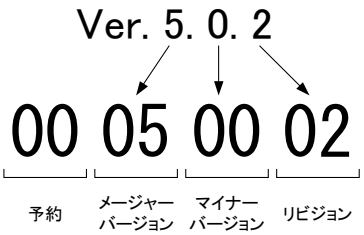


図 45　ファームウェアバージョンの入力方法

5.　[作成]ボタンを押すと、ファイルの保存画面が開きますので ATF ファイルの名前を入力し、[保存]ボタンを押します。ここでは"Sample03.atf"という名前を付けて保存します。保存が成功すれば、ATF ファイルの作成は終了です。

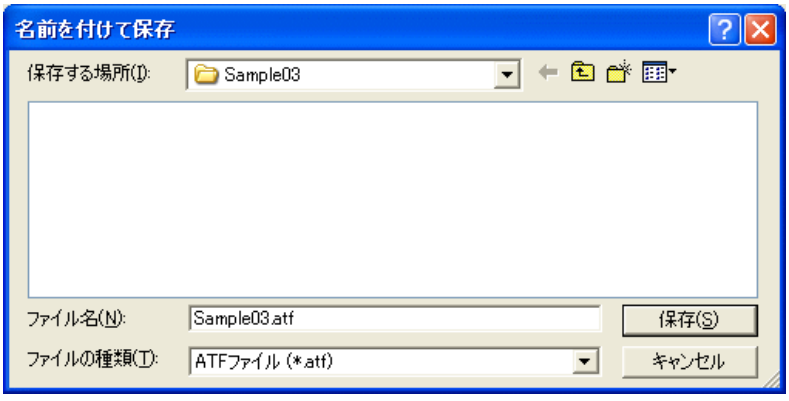


図 46　ATF ファイルの保存

6. 次に作成した ATF ファイルを実際にダウンロードしてテストを行います。「ATF Maker」の画面から[テスト]タブをクリックします。[ファイル名]には先ほど作成した"Sample03.atf"のパスが表示されているはずです。フォルダやファイル名を変更した場合は正しいファイル名を指定してください。

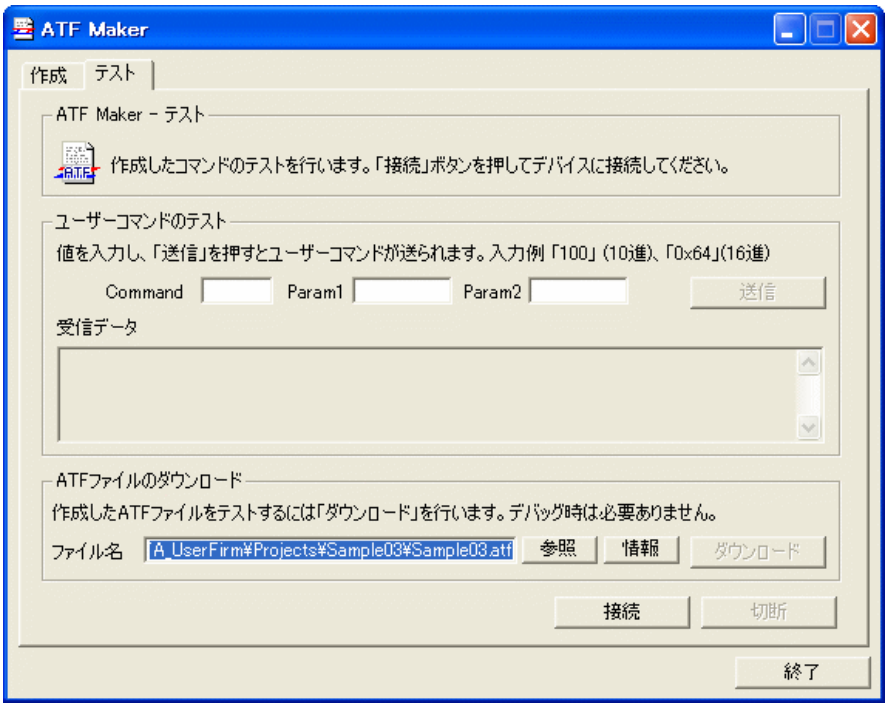


図 47 アタッチメントファームのテスト画面

7. デバイスの**ディップスイッチを 1 番**、2 番とも"OFF"として再起動します。USB や LAN ケーブルを接続し、パソコンと通信可能な状態にしてください。

- ここではデバッグモニタの動作を止めて通常の動作に変更する必要があります。通常デバッグモニタを止めるには 19 ページに従ってデバッグモニタを消去する必要がありますが、簡易的な方法としてディップスイッチの 1 番を"OFF"に設定してください。この状態では RS-485 や AD 変換の機能が正しく動作しませんが、サンプルプログラムの動作確認は可能です。

8. [接続]ボタンを押してデバイスに接続し、[ダウンロード]ボタンで ATF ファイルをデバイスにダウンロードします。
9. サンプルのユーザーファームは標準出力(シリアル 1)に出力を行います。実行状態を確認するためにパソコンのシリアルポートとデバイスのシリアル 1 を接続します。接続方法はデバッグを行う場合と同様です。『イエロースコープ』が開いている場合には閉じてください。『YellowIDE』の[ターミナル]メニューから[表示]を選択し、ターミナル画面を開きます。
10. 再び「ATF Maker」の画面に戻って、[Command]欄に"4"と入力し、[送信]ボタンを押してください。正しく動作している場合、『YellowIDE』のターミナル画面に"tick"という文字が 1 秒毎に表示されます(図 48)。

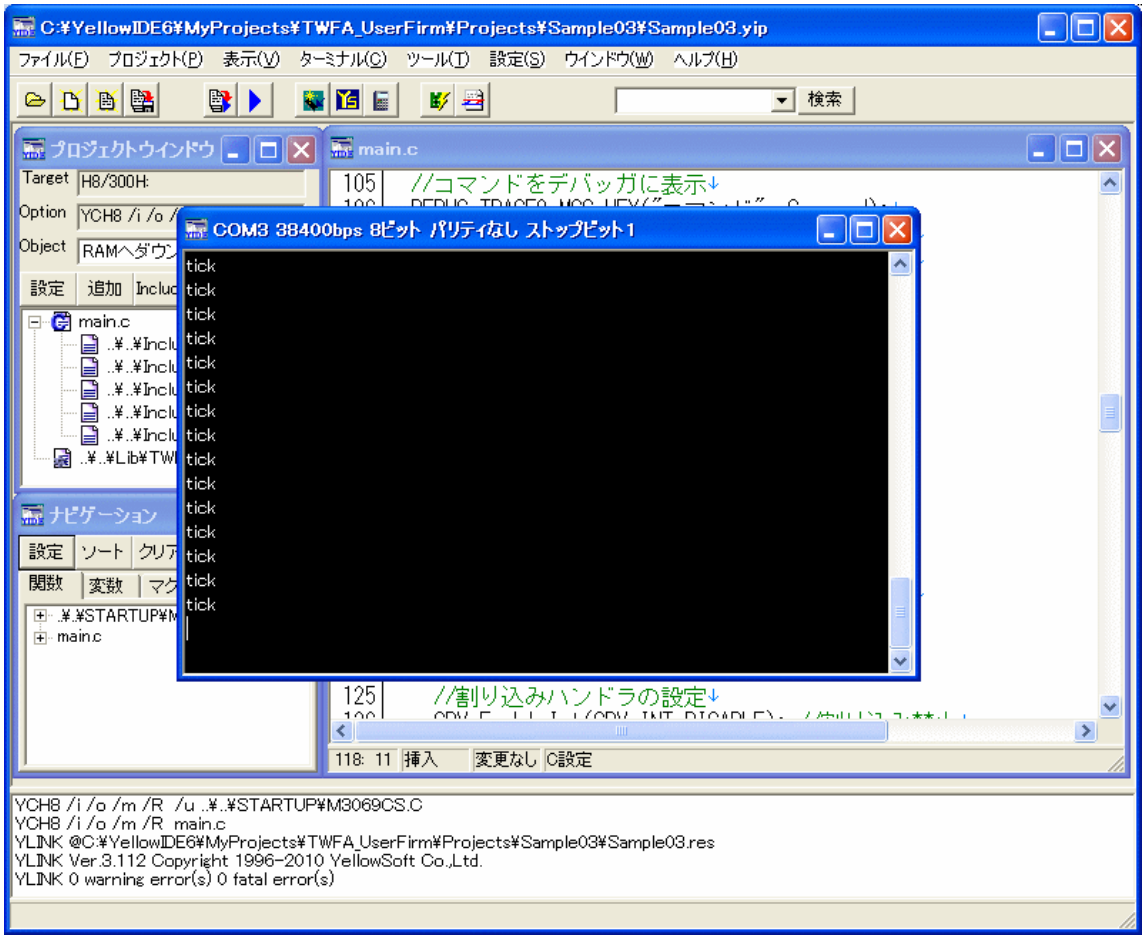


図 48　Sample03 の実行結果

11. 終了するには「ATF Maker」の[Command]欄に"5"と入力して[送信]ボタンを押し、[切断]ボタンを押します。
12. デバッグモニタを再び動作させるためにはディップスイッチの 1 番を"ON"にして、製品を再起動します。

□ アタッチメントファームサンプル(Sample03)のソース

Sample03 は前出の Sample02 に修正を加えたもので内容はほぼ同様です。しかし、アタッチメントファームとして実行するために初期化部分が変更されています。Sample03 では割り込みの登録やタイマのスタートを *ATF_Init()* 関数内では無く、*ATF_Command()* に初期化コマンドを追加し、その中で行うようにしています(リスト 4)

リスト 4　Sample03 の ATF_Command() 関数

```
void ATF_Command(WORD Command, DWORD Param1, DWORD Param2)
{
  //ユーザーコマンドに対応する処理を記述します。
  DWORD dwFreq;

  //コマンドをデバッガに表示
  DEBUG_TRACE0_MSG_HEX("コマンド", Command);
  DEBUG_TRACE0_MSG_HEX("パラメータ 1", Param1);
  DEBUG_TRACE0_MSG_HEX("パラメータ 2", Param2);

  //コマンド処理
  switch(Command){
  case 1:
    g_flgStop = TRUE;
    break;

  case 2:
    g_flgStop = FALSE;
    break;

  case 3:
    SRV_Transmit(&g_dwSec, 4, 1); //応答の送信
    break;

  case 4: //割り込み初期化...①
    //割り込みハンドラの設定
    SRV_EnableInt(SRV_INT_DISABLE); //割り込み禁止

    //チャンネル 0 のコンペアマッチ B にハンドラ登録
    g_TimerIntB[0] = SRV_SetVect(VECT_TIMER0_B, OnCmpB0);

    SRV_EnableInt(SRV_INT_ENABLE); //割り込み許可

    //タイマチャンネル 0 のコンペアマッチ B の割り込みを許可
    TWFA_TimerEnableIntB(TWFA_TIMER_BIT0 /*| TWFA_TIMER_BIT1 | TWFA_TIMER_BIT2*/);

    //タイマの初期化
    TCR16(0) = 0x40; //コンペアマッチ B でクリア
    dwFreq = 100;
    TWFA_TimerSetPwmQ16(0, &dwFreq, NULL, NULL);
    TWFA_TimerStart(TWB_TIMER_BIT0);
    break;

  case 5: //割り込みをデフォルトに復帰...②
    TWFA_TimerStop(TWFA_TIMER_BIT0); //割り込み停止
    SRV_EnableInt(SRV_INT_DISABLE); //割り込み禁止
    SRV_InitVect();
    SRV_EnableInt(SRV_INT_ENABLE); //割り込み許可
  }
}
```

① 初期化用コマンドに対する処理です。割り込みハンドラの登録、タイマのスタートなどを行います。

② 終了処理を行います。タイマを停止し割り込みベクタを初期状態に戻しています。割り込みベクタを変更したままにすると、TWFA ライブラリで提供される標準機能が動作しなくなります。

アタッチメントファームとフラッシュ版のユーザーファームの大きな違いの一つは、*ATF_Init()* の使われ方です。フラッシュ版のユーザーファームでは、デバイスの起動時に *ATF_Init()* が必ず呼び出されシステムの初期化を制御することができます。

しかし、アタッチメントファームで *ATF_Init()* が呼び出されるのは『イエロースコープ』を利用したデバッグ中のみです。アタッチメントファームは既に起動しているデバイスに対してダウンロードされることを前提としていますので、通常の実行では *ATF_Init()* 関数の呼び出しは行われません。そのため、割り込みハンドラの登録などが必要な場合 *ATF_Init()* に代わる初期化手段を用意する必要があります。ただし、*ATF_Main()* と *ATF_Command()* の登録は、ホスト側の *TWXA_ATFDownload()* 関数呼び出し時に自動的に行われます。

逆にデバッグ時は未初期化の状態からプログラムが開始されるため、システムの初期化と *ATF_Main()* や *ATF_Command()* の登録のために *ATF_Init()* が呼び出されます。

上記のことから、アタッチメントファーム開発時に *ATF_Init()* に修正を加えるとプログラム開始時の状態がデバッグ時とリリース時で変わってしまい、バグの原因となりますので注意してください。

6. プログラミング

この章では、ユーザーファームでデバイスを制御する方法や、割り込みの使い方などプログラミングに必要な情報を説明しています。

□ 制御用ライブラリ

製品を制御するには、C 言語の標準ライブラリの他に、3 つの専用ライブラリを主に使用します。1 つ目はシステムファームの機能として提供されるもので、システムタイマ、割り込み、ホストパソコンとの通信、ネットワークの制御などに使用します。これらの関数を***サービス関数***と呼び、関数名は *SRV_* で始まります。

2 つ目は、「TWFA.lib」というライブラリファイルで提供される関数です [5]。このライブラリは ***TWFA ライブラリ***と呼び、ポート(デジタル入出力)、アナログ入出力、タイマ、パルスカウンタ、シリアルポートなど製品固有の機能を制御することを目的としています。これらの関数は関数名が *TWFA_* で始まります。

3 つ目は、「A0x0x.lib」というライブラリファイルで提供される関数です。このライブラリは ***A0x0x ライブラリ***と呼び、X-A0x0x に搭載されている 16 ビット AD コンバータを制御することを目的としています。これらの関数は TWFA ライブラリと同様に、関数名が *TWFA_* で始まります。

それぞれのライブラリの個々の関数の使い方については、関数リファレンス(80 ページおよび 95 ページ)で説明しています。

□ 固定小数点の使用

製品搭載のマイコンで計算を行う場合、整数演算と比較して *double* 型や *float* 型などの浮動小数点を用いた演算にはかなり長い時間が必要になります。

そのため、プログラムの実行速度を上げるために固定小数点での演算を利用した方が良い場合もあります。TWFA ライブラリでは固定小数点数の利用も考慮し、一部の関数は浮動小数点用と固定小数点用の両方を用意しています。

TWFA ライブラリで使用する固定小数点数は Q16 フォーマットの 32 ビット値です。浮動小数点数との変換用に表 5 のマクロが用意されています。

表 5　固定小数点数変換マクロ

マクロ名	説明
TO_Q16(d)	浮動小数点数 d を Q16 フォーマットの 32 ビット符号付固定小数点数に変換します。
TO_UQ16(d)	浮動小数点数 d を Q16 フォーマットの 32 ビット符号なし固定小数点数に変換します。
FROM_Q16(L)	Q16 フォーマットの 32 ビット固定小数点数 L を double 型に変換します。

[5] 関数の一部はマクロによるものや、マクロによるサービス関数の呼び出しとなっているものもあります。

□ デバイスの初期化

デバイスを使用する際は、必ず初期化を行う必要があります。

表 6 に初期化に使用する関数をあげます。

表 6　初期化に使用する関数

関数名	説明
TWFA_Initialize()	マイコンの内部レジスタを初期化します。各デバイス専用の初期化処理を行う前に必ず呼び出します。
TWFA_AOxOxInit()	X-AOxOx デバイス固有機能を初期化します。

デバイス固有機能の初期化

TWXA ライブラリの *TWXA_ADStartFastSampling()* 関数、または、*TWXA_ADStartAutoSampling()*関数を使用する場合は、*TWFA_AOxOxInit()* 関数を必ず呼び出してください。また、*TWFA_AOxOxInit()* 関数を呼び出す際は、必ず *TWFA_Initialize()* 関数の後に呼び出してください。

リスト 5　デバイスの初期化の例

```
//I/O ポートなどマイコンの内部レジスタを初期化
TWFA_Initialize(TWFA_INIT_ALL);

//X-AOxOx デバイス固有機能の初期化
TWFA_AOxOxInit(0);
```

□ アナログ入力

製品はアナログ入力として非絶縁 16 ビット AD コンバータを 8 チャンネル搭載しています。全てのチャンネルの AD 変換は同じタイミングで行われます。アナログ入力に使用する端子は AD0～AD7 端子です。

表 7 はアナログ入力を制御するための関数です。

表 7　アナログ入力で使用する関数

関数名	説明
TWFA_ADSetRange()	アナログ入力の入力レンジを設定します。
TWFA_ADGetRange()	入力レンジの現在の設定を取得します。
TWFA_ADSetMode()	16 ビット AD コンバータの動作モードを設定します。
TWFA_ADGetMode()	16 ビット AD コンバータの現在の動作モードを取得します。
TWFA_AD16Read()	アナログ入力から変換結果を読み出します。
TWFA_An16ToVolt()	アナログ入力の取得値を電圧値(ボルト単位)に変換します。
TWFA_An16ToVoltQ16()	アナログ入力の値を電圧値(ボルト単位)に変換し、Q16 フォーマットで返します。

入力レンジの設定

アナログ入力端子の入力レンジを変更するには *TWFA_ADSetRange()* 関数を使用します。

入力レンジの設定を変更する際は、*TWXA_ADStartFastSampling()* 関数、および、*TWXA_ADStartAutoSampling()* 関数による連続サンプリングが停止している状態で行います。

リスト 6　TWFA_ADSetRange() の関数宣言

```
SRV_STATUS TWFA_ADSetRange(int Range)
```

表 8　TWFA_ADSetRange() の Range 引数に指定する値

値	説明
TWFA_AN_10VPP	入力レンジを 10Vpp(-5～+5[V]) に設定します。
TWFA_AN_20VPP	入力レンジを 20Vpp(-10～+10[V]) に設定します。

入力電圧値と読み出される値の関係は表 9 のようになります。

表 9　アナログ入力電圧と変換結果の関係

入力レンジ	入力電圧値([V])	読み出される値
-5～+5[V]	5-LSB (LSB = 10 / 65536)	32767(H'7FFF)
	2.5	16384(H'4000)
	0	0
	-2.5	-16384(H'C000)
	-5	-32768(H'8000)
-10～+10[V]	10-LSB (LSB = 20 / 65536)	32767(H'7FFF)
	5	16384(H'4000)
	0	0
	-5	-16384(H'C000)
	-10	-32768(H'8000)

・表は理論値を示しています。

オーバーサンプリングレートの設定

オーバーサンプリングレートを変更するには *TWFA_ADSetMode()* 関数を使用します。

リスト 7　TWFA_ADSetMode() の関数宣言

```
SRV_STATUS TWFA_ADSetMode(int Mode)
```

表 10　TWFA_ADSetMode() の Mode 引数に指定する値

値	説明
TWFA_AN_OSR_NON	オーバーサンプリング機能を使用しません。
TWFA_AN_OSR2	オーバーサンプリングレートを 2 に設定します。
TWFA_AN_OSR4	オーバーサンプリングレートを 4 に設定します。
TWFA_AN_OSR8	オーバーサンプリングレートを 8 に設定します。
TWFA_AN_OSR16	オーバーサンプリングレートを 16 に設定します。
TWFA_AN_OSR32	オーバーサンプリングレートを 32 に設定します。
TWFA_AN_OSR64	オーバーサンプリングレートを 64 に設定します。

TWFA_ADSetMode() 関数によりオーバーサンプリングレートを設定すると、以降全ての AD 変換はオーバーサンプリングレートに比例した回数のサンプリングが行われ、その平均値が変換結果として読み出されます（図 49）。

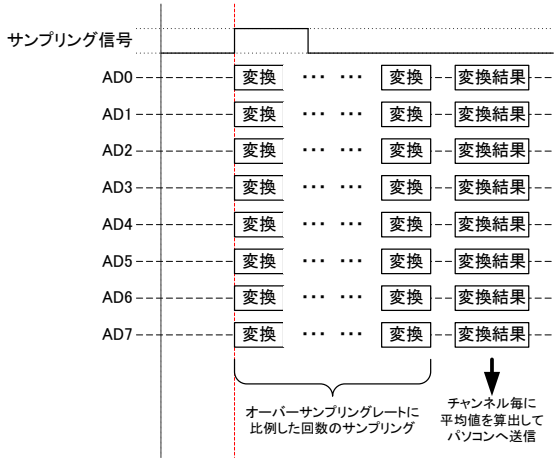


図 49　オーバーサンプリングレートと変換結果の関係

アナログ入力値を読み取る（命令毎に変換）

アナログ入力端子の AD 変換結果を読み出すには *TWFA_AD16Read()* 関数を使用します。

リスト 8　TWFA_AD16Read() の関数宣言

```
SRV_STATUS TWFA_AD16Read(int Ch, short *pData)
```

AD 変換結果は 16 ビット符号付き整数で返されます。1 チャンネルずつ読み出すこともできますが、*Ch* 引数に *TWFA_AD_ALL*（相当の値）を指定すると 0～7 チャンネルまで全ての変換結果を同時に読み出すことができます。その場合は、*pData* 引数として 8 チャンネル分(16 バイト)の領域を確保するようにしてください。

読み出した変換値は *TWFA_An16ToVolt()* 関数や *TWFA_An16ToVoltQ16()* 関数を使用して電圧値に変換することが可能です。*Opt* 引数には *TWFA_ADSetRange()* 関数で設定した入力レンジと同じ値を指定してください。

リスト 9　TWFA_An16ToVolt() の関数宣言

```
double TWFA_An16ToVolt(WORD Data, int Opt)
```

リスト 10　TWFA_An16ToVoltQ16() の関数宣言

```
long TWFA_An16ToVoltQ16(WORD Data, int Opt)
```

表 11　TWFA_An16ToVolt()、および、TWFA_An16ToVoltQ16() の Opt 引数に指定する値

値	説明
TWFA_AN_10VPP	入力レンジが 10Vpp(-5～+5[V]) の場合に指定します。
TWFA_AN_20VPP	入力レンジが 20Vpp(-10～+10[V]) の場合に指定します。

リスト 11　アナログ入力の例

```
short sData[8];
double dVolt;

//入力レンジを -10～+10[V] に設定
TWFA_ADSetRange(TWFA_AN_20VPP);

//オーバーサンプリングレートを 4 に設定
TWFA_ADSetMode(TWFA_AN_OSR4);

//AD0-AD7 の読み出し
TWFA_AD16Read(TWFA_AD_ALL, sData);

//AD0 を電圧値に変換
dVolt = TWFA_An16ToVolt(sData[0], TWFA_AN_20VPP);
```

□ シリアルポート

シリアルポートは最大 2 チャンネル使用可能です。シリアル 0 は RS-485 の半二重通信用です。通常は受信状態となっており、送信用の関数を呼び出した場合のみ自動的に送信状態に切り替わります。

シリアル 1 は RS-232C に準拠した信号レベルでの通信を行います。デフォルトの状態ではユーザーファームのデバッグ用ポート、または、標準入出力ポートとして機能します。ユーザーファームを利用しない場合は、*TWFA_SCISetMode()* をシリアル 1 に対して呼び出すことで、TWFA ライブラリから制御可能な状態となります。

通信方式は調歩同期、通信速度は 300bps～38400bps でフロー制御はありません。受信バッファは 127 バイトでオーバーフローするとステータスレジスタにエラーを記録し、オーバーフローしたデータは捨てられます。

また、受信データを改行コードなどで分割して読み出したい場合には、デリミタコードを設定しておくことができます。デリミタコードを設定しておくと、*TWFA_SCIRead()* 呼び出し時に受信データがチェックされ、デリミタコード(1 バイトまたは 2 バイト)が現れると、シリアルポートからの読み取りを一旦中止し、デリミタコードより後には指定バイトまで 0 をコピーしてデータを返します。

表 12 にシリアルポート制御で使用する関数をあげます。

表 12　シリアルポート制御で使用する関数

関数名	説明
TWFA_SCISetMode()	通信条件の設定を行います。
TWFA_SCIReadStatus()	シリアルポートのエラー、受信バイト数を読み出します。
TWFA_SCIRead()	シリアルポートから指定バイト数のデータを読み出します。
TWFA_SCIWrite()	シリアルポートからデータを送信します。
TWFA_SCISetDelimiter()	デリミタ文字を指定します。

表 13　シリアルポート制御のサンプルプログラム

フォルダ名	説明
SerialSample	受信したデータをそのまま送り返します。

シリアルポートの設定

リスト 12 は *TWFA_SCISetMode()* 関数の宣言です。*Mode* 引数には表 14 に示す値を OR で結合して指定します。その際、データ長、パリティ、ストップビットの設定から 1 つずつオプションを選択して結合するようにしてください。指定がない設定項目はデフォルトと書かれたオプションが選択されます。*Baud* 引数には表 15 の値を指定し、通信速度の設定を行います。

リスト 12　TWFA_SCISetMode() の関数宣言

```
SRV_STATUS TWFA_SCISetMode(int Ch, BYTE Mode, WORD Baud)
```

表 14　TWFA_SCISetMode() の Mode 引数に指定する値

設定項目	値	説明
データ長	TWFA_SCI_DATA8	データ長を 8 ビットにします(デフォルト)。
	TWFA_SCI_DATA7	データ長を 7 ビットにします。
パリティ	TWFA_SCI_NOPARITY	パリティビットを使用しません(デフォルト)。
	TWFA_SCI_EVEN	偶数パリティを使用します。
	TWFA_SCI_ODD	奇数パリティを使用します。
ストップビット	TWFA_SCI_STOP1	ストップビットを 1 ビットとします(デフォルト)。
	TWFA_SCI_STOP2	ストップビットを 2 ビットとします。

表 15　TWFA_SCISetMode() の Baud 引数に指定する値

値	説明
TWFA_SCI_BAUD300	ボーレートを 300bps にします。
TWFA_SCI_BAUD600	ボーレートを 600bps にします。
TWFA_SCI_BAUD1200	ボーレートを 1200bps にします。
TWFA_SCI_BAUD2400	ボーレートを 2400bps にします。
TWFA_SCI_BAUD4800	ボーレートを 4800bps にします。
TWFA_SCI_BAUD9600	ボーレートを 9600bps にします。
TWFA_SCI_BAUD14400	ボーレートを 14400bps にします。
TWFA_SCI_BAUD19200	ボーレートを 19600bps にします。
TWFA_SCI_BAUD38400	ボーレートを 38400bps にします。

シリアルポートの使用手順

1. *TWFA_SCISetMode()* 関数で通信設定を行います。
2. 必要があれば *TWFA_SCISetDelimiter()* 関数でデリミタコードを設定します。
3. データ送信には *TWFA_SCIWrite()* 関数を使用します。
4. 受信データ数やエラーを調べるには *TWFA_SCIReadStatus()* 関数を使用します。
5. データを受信するには *TWFA_SCIRead()* 関数を使用します。

- 受信バッファへの取り込みは割り込みを利用して行われますので、割り込みが禁止になっている間はバッファへの格納が行われません。

□ システムタイマ／カレンダー時計

システムタイマはデバイス起動後の経過時間を記録しています。システムタイマのカウント値は、ウェイトやカレンダー時計などファームウェアで時間を管理する場合に使用されています。

システムタイマのカウンタ変数は 32 ビットで、約 83.9msec 毎にインクリメントされます。ただし、デフォルトでは、このインクリメント動作は自動ではありません。正しくカウントするためには、83.8msec 以下の周期で *SRV_StimeUpdate()* 関数を定期的に呼び出し、タイマカウンタを更新させる必要があります。

システムタイマの更新を自動的に行うには、*SRV_StimeAutoUpdate()* 関数を使用します。自動更新を有効にすると、システムタイマが発生する割り込みを使ってカウンタ変数がインクリメントされます。時間管理が重要なアプリケーションでは、自動更新に設定することを推奨します。

システムファームはシステムタイマを利用して、カレンダー時計を管理する仕組みを持っています。製品はリアルタイムクロックを搭載していないため、電源を投入する度に何らかの方法で時刻合わせを行う必要がありますが、LAN デバイスに関しては SNTP による時刻合わせ機能を搭載していますので、ネットワークを通じて NTP サーバーのカレンダー時計と時刻を同期させることができます。SNTP を利用して時刻合わせを行うには、設定ツールを使用してアクセスする NTP サーバーを指定してください。未設定の場合にもハードコーディングされた NTP サーバーと同期することができますが、アクセス先のサーバーはランダムに変更されますので精度は期待できません。

表 16　システムタイマ／カレンダー時計関連の関数

関数名	説明
SRV_StimeUpdate()	システムタイマのタイマカウンタを更新します。
SRV_StimeGetCnt()	システムタイマのタイマカウンタ値を取得します。
SRV_StimeSetAutoUpdate()	システムタイマの自動更新を許可／禁止します。自動更新は割り込みを利用します。
SRV_StimeGetTime()	システム起動後の時間を msec 単位で返します。
SRV_StimeSleep()	指定時間(msec 単位)経過後戻ります。
SRV_GetTime()	time_t 形式の日時を返します。ANSI の time()関数と同様です。
SRV_SetTime()	time_t 形式で現在日時を設定します。
SRV_SyncTime()	SNTP プロトコルを用いて NTP サーバーと時刻同期を行います。

表 17　システムタイマ／カレンダー時計のサンプルプログラム

フォルダ名	説明
SysTimerSample	システムタイマを自動更新に設定し、日時を標準出力に表示します。ユーザーコマンドで 32 ビットの time_t 形式を送信して時刻合わせすることができます。 また、LAN デバイスの場合には NTP サーバーとの時刻同期を行います。

- システムタイマは搭載マイコンの 8 ビットタイマ(チャンネル 2)を使用しています。

□ ホストインタフェース

比較的小さなデータを扱う場合、ユーザーコマンドを利用してホストパソコンとの通信が可能ですが、ある程度まとまった量のデータを送受信する場合には、直接ホストインタフェースを制御することもできます。

表 18　ホストパソコンとデータを送受信するための関数

関数名	説明
SRV_IsTXE()	送信バッファに空きがあるかを調べます。
SRV_IsRXF()	受信バッファにデータがあるかを調べます。
TWFA_Transmit()	データを送信します。
TWFA_Receive()	データを受信します。
SRV_GetHsIfStatus()	USB デバイスの接続スピードを調べます。
SRV_SetTimeouts()	送受信のタイムアウト時間を設定します。

リスト 13　TWFA_Transmit()、TWFA_Receive()の関数宣言

```
SRV_STATUS TWFA_Transmit(void *pData, WORD n)
SRV_STATUS TWFA_Receive(void *pData, WORD n)
```

リスト 13 はデータの送受信に使用する *TWFA_Transmit()* と *TWFA_Receive()* 関数の宣言です。引数 *pData* は送受信データの格納アドレス、*n* には送受信するバイト数を指定します。

これらの関数は要求されるデータの送受信が完了するまでブロッキングし、一定時間[6]が経過すると *SRVS_TIMEOUT* のステータスを返して終了します。送受信でブロッキングを起こさないようにするためには、あらかじめ *SRV_IsTXE()* 、*SRV_IsRXF()* 関数を呼び出し、送信バッファの空きや、受信バッファ中のデータ数を調べておく必要があります。

LAN デバイスの場合、*SRV_IsTXE()* 、*SRV_IsRXF()* 関数は、それぞれ送信バッファの空きと、受信バッファ中のデータ数をバイト単位で返します。

USB デバイスの場合、*SRV_IsTXE()* 、*SRV_IsRXF()* 関数の戻り値は、ブロッキングなしの送受信が保証されるバイト数です。例えば、*SRV_IsTXE()* 関数の戻り値が 1024 であれば、1024 バイトまでブロッキングなしで送信することができます。戻り値が 1 の場合は、1 バイトの送信でブロッキングしないことは保証されますが、2 バイトの送信ではブロッキングする可能性があります。これらの関数の戻り値はホストパソコンとの接続スピードにより取り得る値がかわります(表 19)。接続スピードを調べるには *SRV_GetHsIfStatus()* 関数を使用します。

表 19　USB の接続スピードと SRV_IsTXE()、SRV_IsRXF()の戻り値の関係

接続スピード	SRV_IsTXE()、SRV_IsRXF()の戻り値
ハイスピード接続の場合	0, 1, 512, 1024
フルスピード接続の場合	0, 1, 64, 128

[6] デフォルトのタイムアウト時間は USB デバイスが約 5 秒、LAN デバイスは約 10 秒です。

表 20　TWFA_Transmit()と TWFA_Receive()を使用したサンプルプログラム

フォルダ名	説明
InterfaceSample	ループバックモードに設定すると、0xff 以外の受信データをホストパソコンにそのまま送り返します。0xff を受信するとループバックモードを抜けて、通常のコマンド処理が可能になります。 「HostSample¥HostSample.sln」中の「InterfaceSample_MFC」というサンプルプログラムで動作確認を行うことができます。

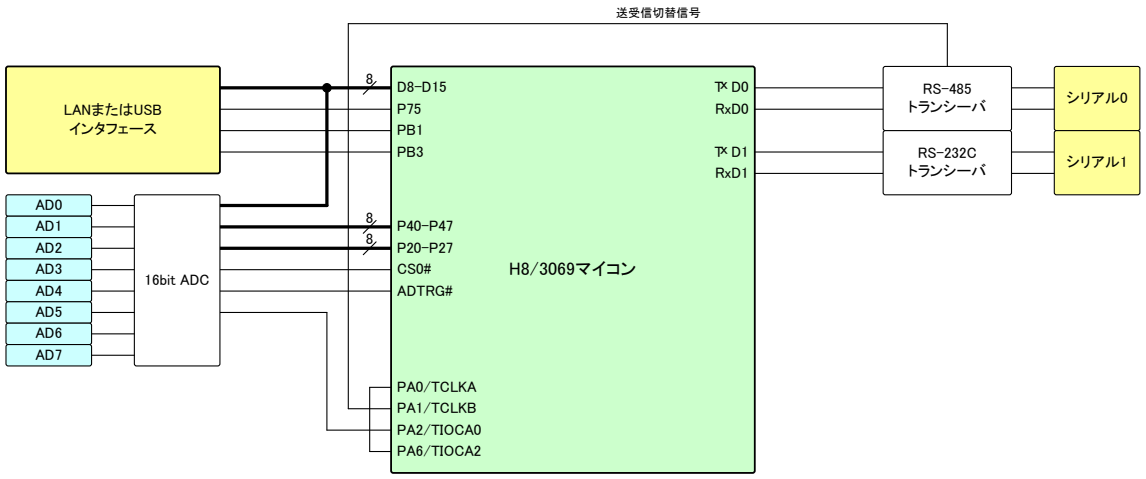
□ レジスタアクセス

ユーザーファームを開発する上で、ライブラリでサポートされないマイコンの周辺機能を利用したい場合にはマイコン内部の制御用レジスタに直接アクセスする必要があります。

「Include¥h8ios.h」ファイル内にはマイコンのレジスタにアクセスするためのマクロが定義されています。

- TWFA ライブラリでサポートされる機能についてはライブラリ経由で制御してください。
- サービス関数や TWFA ライブラリでサポートされていない機能については、弊社のホームページ「https://www.techw.co.jp/SupportFrm.html?pid=X-A0x0x」の「H8/3069R マイコンハードウェアマニュアル」を参照してください。

搭載マイコンと各端子の関係



□ 割り込み

ライブラリでは 16 ビットタイマによる割り込み処理がサポートされています。割り込みルーチンはシステムファームの内部処理や、ライブラリ関数を呼び出している間でも実行されますので、タイムクリティカルな処理を行うのに適しています。ただし、16 ビットタイマのチャンネル 0 とチャンネル 2 は TWXA ライブラリの *TWXA_ADStartFastSampling()* 関数、および、*TWXA_ADStartAutoSampling()* 関数によるアナログ入力の連続変換で使用されているので、アナログ入力の連続変換を行う場合は使用できません。

TWXA ライブラリ関数の呼び出しによるアナログ入力の連続変換を行わない場合、16 ビットタイマを 37 ページのサンプルプログラム(Sample02.yip)のように、ユーザーファーム内部で一定周期の割り込みを発生させたい場合などに利用することができます。

16 ビットタイマの割り込みは 1 チャンネルにつき 3 つの要因があります。1 つはタイマカウンタの値がオーバーフローした場合に発生するオーバーフロー割り込みです。この割り込みはパルスカウンタに設定したチャンネルのカウンタ値がオーバーフローした場合などに発生します。

後の 2 つは、GRA、GRB という 16 ビットレジスタの値とタイマカウンタの値が一致したときに発生するもので、それぞれコンペアマッチ A、コンペアマッチ B と呼ばれます。パルスカウンタに設定したチャンネルの GRA または GRB レジスタに予め値を設定しておけば、カウンタ値がその値と一致したときに割り込みが発生します。

表 21　16 ビットタイマの割り込み要因

割り込み要因	説明
コンペアマッチ A	カウンタ値が GRA レジスタと一致した場合に発生します。
コンペアマッチ B	カウンタ値が GRB レジスタと一致した場合に発生します。
オーバーフロー	カウンタ値がオーバーフローした場合に発生します。

表 22　割り込みの制御に使用する主な関数

関数名	説明
SRV_EnableInt()	割り込みの許可／禁止に使用します。(優先順位の高い)プライオリティ 1 の割り込みだけを許可することもできます。
SRV_SetVect()	割り込みベクタに割り込みルーチンを登録します。
TWFA_TimerEnableIntA()	16 ビットタイマのコンペアマッチ A 割り込みを許可／禁止します。
TWFA_TimerEnableIntB()	16 ビットタイマのコンペアマッチ B 割り込みを許可／禁止します。
TWFA_TimerEnableIntOvf()	16 ビットタイマのオーバーフロー割り込みを許可します。

割り込みハンドラの記述方法

割り込み発生時に実行されるプログラムを割り込みハンドラ、または、割り込みルーチンと呼びます。割り込みハンドラは通常のプログラム実行を中断して実行され、どの行を実行中に発生するかもわかりません。そのため、割り込みハンドラは、汎用レジスタやシステムレジスタなどを割り込み発生前の状態に復帰して戻る必要があります。また、割り込みハンドラの中でグローバル変数などのプログラム全体で共有されるリソースを変更すると、通常実行のプログラムと競合し誤動作を起こす場合がありますので十分な注意が必要です。

上記のことから、ライブラリ関数の多くも割り込みハンドラの中から呼び出すことができません。割り込みハンドラ内から呼び出すことができる関数は、関数リファレンスの説明欄にそのことが明記されています。

割り込みハンドラを記述する場合、割り込み発生時に実行したい関数に *interrupt* キーワードを指定して定義します。*interrupt* キーワードにより、関数終了時に汎用レジスタやシステムレジスタを復帰するための命令が自動的に追加されます。ユーザーファームのスケルトンプログラムには、予め 16 ビットタイマによる割り込みに対応した割り込みハンドラ関数が定義されていますので、それらに処理を追加して利用してください。

割り込みベクタの設定

割り込み要因と、割り込みハンドラ(のアドレス)の関係をテーブル化したものを割り込みベクタと呼びます。割り込み要因が発生したときに、希望の割り込みハンドラを実行させるためには、この割り込みベクタに、割り込みハンドラのアドレスを前もって登録する必要があります。割り込みベクタに関数を登録するには *SRV_SetVect()* 関数を呼び出してください。

ユーザーファームのスケルトンコードには 16 ビットタイマに対応するベクタ登録のコードが埋め込まれていますので、必要な部分のコメントを外すことで登録が実行されます(40 ページ、リスト 1 参照)。

- ユーザーファーム開発では『YellowIDE』の設定画面で割り込みベクタを設定することはできません。

割り込みの許可／禁止

割り込み全体の許可／禁止には *SRV_EnableInt()* 関数(リスト 14)を使用します。*Pri* 引数に *SRV_INT_ENABLE* を指定すると割り込み許可、*SRV_INT_DISABLE* を指定すると割り込みの禁止、*SRV_INT_ENABLE_PRI1* を指定すると優先順位が高いプライオリティレベル 1 の割り込みだけが許可されます。プライオリティレベルについては後述します。

リスト 14　SRV_EnableInt()の関数宣言

```
void SRV_EnableInt(int Pri)
```

割り込みの許可や禁止は全体の操作の他に、個々の割り込み要因に対しても行うことができます。16 ビットタイマの割り込み許可／禁止は、コンペアマッチ A、コンペアマッチ B、オーバーフローの要因毎に *TWFA_TimerEnableIntA()*、*TWFA_TimerEnbaleIntB()*、*TWFA_TimerEnableIntOvf()* 関数を使用します。

これらの割り込み要因はデフォルトでは禁止されていますので、使用時には各関数を呼び出して必要なチャンネルを許可する必要があります。

16 ビットタイマによる割り込みの使用手順

1. 割り込みハンドラを記述し、*SRV_SetVect()* 関数で割り込みベクタに登録します。
2. コンペアマッチ A、コンペマッチ B 割り込みを利用するために GRA、GRB レジスタへの書き込みが必要な場合、リスト 15 のようにマクロを利用して設定することができます。

リスト 15　GRA、GRB レジスタへの書き込み

```
GRA(0) = 10000; //チャンネル 0 の GRA レジスタに値を設定
GRB(0) = 20000; //チャンネル 0 の GRB レジスタに値を設定
```

3. タイマクロックの選択、カウントエッジの選択、クリア条件などは、チャンネル毎に用意された 16TCR レジスタに設定します。16TCR への書き込みはリスト 16 のようにマクロを利用して記述します。レジスタ機能の詳細は H8/3069 マイコンのハードウェアマニュアルを参照してください。

リスト 16　16TCR レジスタへの書き込み

```
TCR16(0) = 0x40; //コンペアマッチ B でカウンタクリア、クロックに 25MHz を選択
```

4. *TWFA_TimerEnableIntA()*、*TWFA_TimerEnableIntB()*、*TWFA_TimerEnableIntOvf()* 関数を使用し必要な割り込みを許可します。
5. *TWFA_TimerStart()* 関数で使用するタイマチャンネルを起動します。

システムファームが使用する割り込み

システムファームでは表 23 の割り込みを使用します。表中のプライオリティレベルは割り込みの優先度を表しています。

製品搭載のマイコンでは、要因別に割り込みの優先順位をプライオリティレベル 1(優先)とプライオリティレベル 2(非優先)の 2 段階に設定可能となっています。割り込みの許可／禁止を設定する場合にはプライオリティレベル 1 だけを許可することができます。また、同時に割り込みが発生した場合は、プライオリティレベルが高い割り込みが先に処理されます。同じプライオリティレベルの割り込みが同時に発生した場合はベクタ番号が小さい割り込みが先に受け付けられます。

デフォルトでは 16 ビットタイマによる割り込みだけがプライオリティレベル 1 に設定されます。

表 23　システムファームが使用する割り込み

割り込み要因	ベクタ番号	プライオリティレベル	説明
16 ビットタイマ 0 コンペアマッチ B	25	1	TWFA_TimerSetNumOfPulse()によるパルス停止に使用
16 ビットタイマ 1 コンペアマッチ B	29	1	TWFA_TimerSetNumOfPulse()によるパルス停止に使用
16 ビットタイマ 2 コンペアマッチ B	33	1	TWFA_TimerSetNumOfPulse()によるパルス停止に使用
8 ビットタイマ 2 オーバーフロー	43	2	SRV_StimeSetAutoUpdate()によりシステムタイマを自動更新に設定した場合に使用
シリアル 0 受信/エラー	53	2	シリアル 0 の受信に使用。割り込み処理中も他の割り込みが許可されます。
シリアル 1 受信/エラー	57	2	シリアル 1 の受信に使用。割り込み処理中も他の割り込みが許可されます。

□ ユーザーファームの動作設定

ユーザーファームの動作設定を製品付属のツールでフラッシュメモリに保存しておくことができます。ユーザーファームではサービス関数を使用して、設定データ中のパラメータを読み出します。

表 24　動作設定の読み出しに使用する関数

関数名	説明
SRV_GetProfileString()	パラメータの値を文字列として読み出します。
SRV_GetProfileInt()	パラメータの値を 32 ビット符号無し整数として読み出します。
SRV_EnumParam()	セクション内のパラメータ名を列挙します。

動作設定ファイルの作成と書き込み

動作設定ファイルの書き込みは「M3069IniWriter」というプログラムを使用します。「M3069IniWriter」は製品付属の設定ツールのメニュー画面から呼び出すことができます(図 50)。

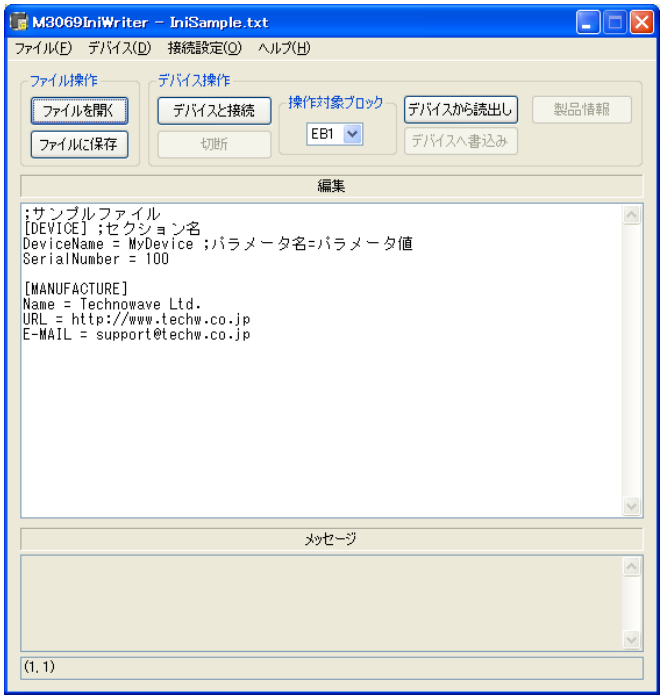


図 50　「M3069IniWriter」の画面

設定ファイルは通常のテキストファイルで、Windows の INI ファイルと同じような形式で作成することができます。リスト 17 は設定ファイルの例です。

リスト 17　設定ファイルの例

```
;サンプルファイル ... ①
[DEVICE] ;セクション名 ... ②
DeviceName = MyDevice ;パラメータ名=パラメータ値 ... ③
SerialNumber = 100

[MANUFACTURE]
Name = Technowave Ltd.
URL = http://www.techw.co.jp
E-MAIL = support@techw.co.jp
```

① ";"の後はコメントとみなされます。

② 文字列を "["、"]" で囲んでセクションを定義できます。セクション名はサービス関数でパラメータを検索するときのキーとなります。セクション内には複数のパラメータを定義することができます。

③ パラメータは "パラメータ名=パラメータ値" の形で定義します。また、"=" 以後を省略して値を持たないパラメータを定義することもできます。ユーザーファームではサービス関数を使用して、パラメータ名を指定して対応するパラメータ値を読み出すことと、セクション内のパラメータ名を列挙することができます。

設定ファイルは「M3069IniWriter」によってファームウェアが検索しやすい形式にエンコードされ、バイナリデータとしてデバイスに書き込まれます。

設定ファイルの記述方法、制限事項、具体的な書き込み手順は「M3069IniWriter」のオンラインヘルプを参照してください。

パラメータの読み出し

SRV_GetProfileString() 関数(リスト 18) はパラメータ値を文字列として読み出し、先頭アドレスを返します。返される文字列は'¥0'で終端された文字列です。何らかの理由でパラメータが見つからない場合は *pDefStr* 引数で渡されたアドレスを返します。

Address 引数は表 25 の値により対象となるフラッシュメモリブロックを指定します。リスト 19 はリスト 17 の *DeviceName* パラメータを読み出す例です。

リスト 18　SRV_GetProfileString() の関数宣言

```
char *SRV_GetProfileString(DWORD Address, char *pSection, char *pParam,
                           int *pnStr, char *pDefStr, int nDefStr)
```

表 25　SRV_GetProfileString() / SRV_GetProfileInt()の Address 引数に指定する値

値	説明
SRV_EB1_ADDRESS	フラッシュメモリブロック 1 のデータを検索します。
SRV_EB2_ADDRESS	フラッシュメモリブロック 2 のデータを検索します。
SRV_EB3_ADDRESS	フラッシュメモリブロック 3 のデータを検索します。

リスト 19　SRV_GetProfileString() 関数の使用例

```
char *p;

//DeviceName の値を読み出して標準出力に表示
p = SRV_GetProfileString(SRV_EB1_ADDRESS, "DEVICE", "DeviceName", NULL, "NoName", 0);
puts(p);
```

SRV_GetProfileInt() 関数(リスト 20)はパラメータ値を 32 ビット符号無し整数に変換して返します。何らかの理由でパラメータが見つからない場合は *Default* 引数で渡された値を返します。

リスト 21 はリスト 17 の *SerialNumber* パラメータの値を読み出す例です。

リスト 20　SRV_GetProfileInt() の関数宣言

```
DWORD SRV_GetProfileInt(DWORD Address, char *pSection, char *pParam, DWORD Default)
```

リスト 21　SRV_GetProfileString() 関数の使用例

```
DWORD dw;

//SerialNumber の値を読み出してデバッグ出力に表示
dw = SRV_GetProfileInt(SRV_EB1_ADDRESS, "DEVICE", "SerialNumber", 0);
DEBUG_TRACE0_MSG("SerialNumber", dw);
```

表 26　動作設定の読み出しのサンプルプログラム

フォルダ名	説明
IniSample	書き込まれた設定情報をデバッグ画面へ出力します。「M3069IniWriter」でプロジェクトフォルダ中の「IniSample.txt」ファイルを EB1 に書き込んで使用します。

□ ウォッチドッグタイマ

ウォッチドッグタイマを有効にすると、ウォッチドッグタイマのカウンタがオーバーフローした場合にリセットがかかり、システムが再起動されます。カウンタをクリアしてからオーバーフローが発生するまでの時間は約 41.9msec です。

表 27　ウォッチドッグタイマの制御に使用する関数

関数名	説明
SRV_WdEnable()	ウォッチドッグタイマの有効／無効を切り替えます。
SRV_StimeUpdate()	ウォッチドッグタイマのタイマカウンタをクリアします。

表 28　ウォッチドッグタイマのサンプルプログラム

フォルダ名	説明
WatchdogSample	ウォッチドッグタイマを起動します。また、ウォッチドッグタイマによるリセットを検出すると標準出力に表示を行います。デバッグ中に[中断]ボタンで停止するとウォッチドッグタイマによるリセットがかかります。

ウォッチドッグタイマの有効／無効を切り替えるには *SRV_WdEnable()* 関数を使用します。カウンタをクリアするためには、*SRV_StimeUpdate()* 関数を呼び出してください。

また、ウォッチドッグタイマによるリセットが発生したかどうかを調べたり、ウォッチドッグタイマによるリセットステータスをクリアしたりするには制御用のレジスタを直接操作します。詳細はサンプルプログラムを参照してください。

- デバッグ時はブレーク(中断)が発生した時点で、リセットがかかってしまいますのでウォッチドッグタイマは使用できません。
- システムタイマの更新を *SRV_StimeSetAutoUpdate()* 関数で自動化してもウォッチドッグタイマのカウンタクリアは自動化されません。

7. ネットワークプログラミング

この章では、LAN デバイスを用いたネットワークに関するプログラミングについて説明します。

□ ネットワークリソース

製品は TCP/IP のプロトコルをハードウェアで処理していますが、ハードウェア上の制約で同時に使用できるネットワークチャンネルは 0～3 までの最大 4 チャンネルとなっています。また、一部のチャンネルは TWXA ライブラリとの通信や、DHCP のためにシステムファームが使用しますので、ユーザーファームで使用することができません(表 29)。

表 29　システムファームが使用するネットワークチャンネル

チャンネル	固定 IP の場合	DHCP を使用する場合
0	TWXA ライブラリとの通信用	TWXA ライブラリとの通信用
1	TWXA ライブラリのリストアップコマンドへの応答／ハードウェアイベントの通知	DHCP 用
2	ユーザーファームで使用可能	TWXA ライブラリのリストアップコマンドへの応答／ハードウェアイベントの通知
3	ユーザーファームで使用可能	ユーザーファームで使用可能

表 29 の「リストアップコマンドへの応答」とは、TWXA ライブラリがネットワーク内の対応デバイスを検索するためのコマンドへの応答を意味しています。

システムファームが利用するチャンネルの一部は、ユーザーファームの初期化中にその動作を禁止して解放させることができます。

リスト 22 はシステムファームにネットワークの初期化を指示するための *SRV_LanmInit()* 関数の宣言です。*Option* 引数には表 30 の値を OR で組み合わせて初期化の指示を行います。許可しない機能があれば、その機能の使用チャンネルが解放されます。

リスト 22　SRV_LanmInit() の関数宣言

```
SRV_STATUS SRV_LanmInit(DWORD Option)
```

表 30　SRV_LanmInit() 関数の Option 引数に指定する値

値	説明	使用チャンネル
LANMM_ENABLE_CONTROL	TWXA ライブラリとの通信用チャンネルを有効にします。指定すると TWXA ライブラリによる制御が可能になります。	0
LANMM_ENABLE_LIST	TWXA ライブラリのリストアップへの応答を有効にします。指定すると TWXA ライブラリがネットワーク内を検索してそのデバイスを発見できるようになります。また、ハードウェアイベントの通知が可能になります。 逆に指定しない場合は、TWXA ライブラリから接続するために IP アドレスを指定する必要があります。	1 または 2

□ TCP によるサーバープログラム

TCP を利用したサーバープログラムに使用する主な関数を表 31 にあげます。

表 31　TCP によるサーバープログラムに使用する主な関数

関数名	説明
SRV_SockOpen()	ネットワークチャンネルを初期化し、使用可能にします。
SRV_ScokClose()	ネットワークチャンネルの使用を終了します。
SRV_SockReadStatus()	ネットワークチャンネルのステータスを取得します。
SRV_SockListen()	ネットワークチャンネルを接続待ちの状態にします。
SRV_SockDisconnectA()	相手とのネットワーク接続を切断します。
SRV_SockSend()	接続先にデータを送信します。
SRV_SockRecv()	受信バッファからデータを取り出します。
SRV_SockPeek()	受信バッファにデータを残したまま読み取りを行います。
SRV_SockPurge()	受信バッファのデータを削除します。
SRV_SockGetRecvSize()	受信バイト数を調べます。
SRV_SockGetFreeSize()	送信バッファの空き容量を調べます。
SRV_SockKeepAlive()	接続が有効か調べるためにキープアライブパケットを送信します。

表 32　TCP によるサーバーのサンプルプログラム

フォルダ名	説明
TcpServerSample	起動すると 50000 ポートを開いてクライアントの接続を待ちます。接続後は受け取ったデータをそのままループバックします。 「HostSample¥HostSample.sln」中の「TcpSample_MFC」というサンプルプログラムで動作確認を行うことができます。

ネットワークチャンネルを使用するには、まず、*SRV_SockOpen()* 関数(リスト 23)でそのチャンネルを初期化する必要があります。TCP で利用するためには *Protocol* 引数に *SOCK_STREAM* を指定してください。*Port* 引数には接続待ちのためにオープンするローカルポートの番号を指定します。

小さなデータを頻繁に送受信する場合は、*Option* 引数に *SOCKOPT_NDACK* を指定します。このオプションを指定すると、相手からデータを受信すると直ちに ACK パケットを送信し、正しく受け取ったことを知らせます。逆にこのオプションを指定しない場合、一定量の受信データに対してまとめて ACK を返すような動作となり、トラフィックは減りますが応答の遅延時間が長くなります。

リスト 23　SRV_SockOpen() の関数宣言

```
SRV_STATUS SRV_SockOpen(int CH, int Protocol, WORD Port, int Option)
```

接続の待ち受けを開始するには *SRV_SockListen()* 関数を呼び出します。一般の socket によるプログラムでは *listen()* の後に *accept()* 関数を呼び出し、実際に送受信を行うためのソケットを作成しますが、ユーザーファームのプログラムでは *SRV_SockListen()* 関数を呼び出したチャンネルが直接クライアントと接続されます。そのため、複数のクライアントと接続を行うには、接続を行うチャンネル全てに対して *SRV_SockListen()* 関数を呼び出します。

指定のチャンネルが実際にクライアントと接続されたかどうかは、SRV_SockReadStatus()関数(リスト 24)が返すチャンネルのステータスでチェックします。表 33 は *SRV_SockReadStatus()* 関数の主な戻り値です。戻り値が *SOCKS_ESTABLISHED* となっていればクライアントから接続されたことを示します。

リスト 24 SRV_SockReadStatus() の関数宣言

```
WORD SRV_SockReadStatus(int CH)
```

表 33 SRV_SockReadStatus() 関数の主な戻り値

戻り値	説明
SOCKS_CLOSED	指定のチャンネルは閉じています。*SRV_SockOpen()*で初期化する必要があります。
SOCKS_ESTABLISHED	TCP により相手と接続されています。データの送受信が可能です。
SOCKS_CLOSE_WAIT	相手からの切断要求を示します。*SRV_SockDisconnectA()*で切断してください。
SOCKS_UDP	UDP 通信用に初期化されたチャンネルです。

相手との接続を切る場合は *SRV_SockDisconnectA()* 関数(リスト 25)を呼び出します。*Option* 引数は 0 とすると切断要求に対する相手からの応答を待って関数がブロックします。*Option* 引数に *SOCK_NB* を指定すると完了を待たずに関数から戻ります。その場合、*SRV_SockReadStatus()* 関数の戻り値が *SOCKS_CLOSED* になった時点で切断が完了となります。

リスト 25 SRV_SockDisconnectA() の関数宣言

```
SRV_STATUS SRV_SockDisconnectA(int CH, int Option)
```

一定時間、無通信状態のときに、相手との接続が有効かどうかを調べたい場合があります。このような場合は、*SRV_SockKeepAlive()* 関数を呼び出してください。キープアライブと呼ばれる空のパケットを送ることで相手との接続が有効かどうか調べることができます。キープアライブに対する相手からの応答が一定時間得られない場合、チャンネルは自動的に閉じられ、*SRV_ScokReadStatus()* 関数の戻り値が *SOCKS_CLOSED* になります。

TCP によるサーバー動作の手順

1. *SRV_SockOpen()* 関数で TCP による通信チャンネルの初期化を行います。*Protocol* 引数には *SOCK_STREAM* を指定します。
2. *SRV_Listen()* 関数を呼び出し、1.で初期化したチャンネルを接続待ち状態にします。
3. *SRV_SockReadStatus()* 関数の戻り値により、クライアントとの接続が完了したかどうかを調べます。
4. 接続が完了していれば、*SRV_SockSend()*、*SRV_SockRecv()* などの関数を呼び出してデータを送受信することができます。これらの関数は、直ちに実行可能な範囲でデータの送受を行いますのでブロッキング状態になることはありません。例えば *SRV_SockSend()* 関数は、送信バッファの空き容量が指定の送信データ数より少ない場合は、バッファに格納可能なバイト数のみ送信処理を行い処理を終えます。送信処理が完了したバイト数は戻り値として返されます。
5. クライアント(接続相手)から切断要求がある場合には、*SRV_SockReadStatus()* 関数の戻り値が *SOCKS_CLOSE_WAIT* となりますので *SRV_SockDisconnectA()* 関数を呼び出して切断処理をします。また、こちらから切断処理を行う場合も *SRV_SockDisconnectA()* 関数を呼び出

します。

6. 適切に切断された通信チャンネルはステータスが *SOCKS_CLOSED* となりますので、再度利用する場合は *SRV_SockOpen()* 関数で初期化を行います。

- 何らかの理由で通信が行えなくなったチャンネルは、切断要求に対する応答を期待できません。このような場合、*SRV_SockDisconnectA()* 関数ではなく、*SRV_SockClose()* 関数で強制的に閉じることで、そのチャンネルをすぐに使用できるようになります。

□ TCP によるクライアントプログラム

TCP を利用したクライアントプログラムでは表 31 の関数に加えて表 34 の関数を主に使用します。

表 34　TCP によるクライアントプログラムに使用する主な関数

関数名	説明
SRV_SockConnectA()	サーバーとの接続を行います。
SRV_SockGetHostByName()	ドメイン名から IP アドレスを取得します。DNS サーバーのアドレスが設定されている必要があります。
SRV_SockGetHostByNameA()	*SRV_SockGetHostByName*()と同様ですがオプションでノンブロッキングに指定できます。

表 35　TCP によるサーバーのサンプルプログラム

フォルダ名	説明
TcpClientSample	起動すると定数で設定されたサーバーアドレスに対して接続を行います。接続後は受け取ったデータをそのままループバックします。 「HostSample¥HostSample.sln」中の「TcpSample_MFC」というサンプルプログラムで動作確認を行うことができます。

TCP によるクライアントプログラムは、接続処理を除いてサーバープログラムの場合と同様に行います。接続処理にはリスト 26 の *SRV_SockConnectA()* 関数を使用します。

リスト 26　SRV_SockConnectA() の関数宣言

```
SRV_STATUS SRV_SockConnectA(int CH, BYTE *pDstAddr, WORD Port, int Option)
```

pDstAddr 引数には、サーバーの IP アドレスの 4 つのフィールドを格納した 4 バイトの配列を渡します。例えば"192.168.0.1"のサーバーアドレスを表現する配列は以下のように初期化します。

リスト 27　IP アドレスの格納方法

```
BYTE server_ip[4] = { 192, 168, 0, 1 };
```

Port 引数は接続先サーバーのポート番号を指定します。*Option* 引数は 0 とすると、接続が完了するか、タイムアウトするまで関数がブロックします。*Option* 引数に *SOCK_NB* を指定すると、関数は処理が完了していなくても終了し *SRV_SOCK_PENDING* を返します。この場合、接続処理の結果は *SRV_SockReadSutatus()* 関数で調べます。接続に成功するとステータスが *SOCKS_ESTABLISHED* となります。接続に失敗した場合はステータスが *SOCKS_CLOSED* に戻ります。接続処理が完了していない場合はそれ以外の値が返ります。

サーバーのアドレスがドメイン名で与えられる場合には、*SRV_SockGetHostByName()* 関数(リスト 28)を使用し、接続前に IP アドレスを調べます。この関数を使用するには、製品のネットワーク設定で DNS サーバーが登録されていることが必要です。

リスト 28　SRV_SockGetHostByName() の関数宣言

```
SRV_STATUS SRV_SockGetHostByName(int CH, char *pName, BYTE *pIP, DWORD Timeout)
```

SRV_SockGetHostByName() 関数は DNS サーバーから応答があるまでブロックします。*Timeout* 引数にはタイムアウトまでの時間を msec 単位で指定してください。また、この関数が使用するネットワークチャンネルは関数が終了した時点で解放されますので、通常は *SRV_SockConnectA()* 関数で使用予定のチャンネルを利用します。

- *SRV_SockGetHostByNameA()* 関数を使用すると、ノンブロッキングを指定して DNS サーバーに問い合わせが可能です。
- *SRV_SyncTime()* 関数が使用するチャンネルも *SRV_SockGetHostByName()* 関数同様に、関数の実行中のみ使用され終了すれば解放されます。このような関数に対しては DHCP 用のチャンネルを一時的に利用することが可能です。DHCP で使用しているチャンネルのステータスを調べ *SOCKS_CLOSED* となっている場合は、そのチャンネルを利用できます。

TCP によるクライアント動作の手順

1. *SRV_SockOpen()* 関数を使用し、TCP による通信チャンネルの初期化を行います。*Protocol* 引数には *SOCK_STREAM* を指定します。
2. *SRV_ConnectA()* 関数を呼び出し、サーバーへの接続処理を行います。
3. *SRV_ConnectA()* 関数の呼び出しで *Option* 引数に *SOCK_NB* を指定した場合は、*SRV_SockReadStatus()* 関数の戻り値により、サーバーとの接続が完了したかを調べます。
4. 接続が完了していれば、*SRV_SockSend()*、*SRV_SockRecv()* などの関数を呼び出してデータを送受信することができます。
5. サーバー(接続相手)から切断要求がある場合には、*SRV_SockReadStatus()* 関数の戻り値が *SOCKS_CLOSE_WAIT* となりますので *SRV_SockDisconnectA()* 関数を呼び出して切断処理をします。また、こちらから切断処理を行う場合も *SRV_SockDisconnectA()* 関数を呼び出します。
6. 適切に切断された通信チャンネルはステータスが *SOCKS_CLOSED* となります。再び接続する場合は *SRV_SockOpen()* 関数で再初期化を行います。

□ UDP による通信

UDP による通信に使用する主な関数を表 36 にあげます。

表 36　UDP による通信に使用する主な関数

関数名	説明
SRV_SockOpen()	ネットワークチャンネルを初期化し、使用可能にします。
SRV_ScokClose()	ネットワークチャンネルの使用を終了します。
SRV_SockSendTo()	指定のアドレスの指定ポートにデータを送信します。
SRV_SockRecvFrom()	UDP チャンネルの受信バッファからデータを取り出します。データはパケット単位で取り出されます。
SRV_SockGetRecvSize()	受信バイト数を調べます。
SRV_SockGetFreeSize()	送信バッファの空き容量を調べます。

表 37　UDP による通信のサンプルプログラム

フォルダ名	説明
UdpSample	起動すると 50000 ポートを開いて待ちます。データを受信すると受け取ったデータをそのままループバックします。 「HostSample¥HostSample.sln」中の「UdpSample_MFC」というサンプルプログラムで動作確認を行うことができます。

UDP では接続処理が必要ありませんので、*SRV_SockOpen()* 関数の *Protocol* 引数に *SOCK_DGRAM* を指定して初期化を行えば送受信を開始できます。

データの送受信には *SRV_SockSendTo()*、*SRV_SockRecvFrom()* 関数を使用します。UDP チャンネルのデータ受信に *SRV_SockRecv()* 関数は使用できません。

また、製品では IP レイヤでのフラグメントをサポートしていませんので、UDP で扱うデータサイズは 1472 バイト以下[7]とする必要があります。

[7] イーサネットの MTU(1500 バイト)を基準とした値です。通信経路の MTU 値が小さくなると、送受信できるサイズも小さくなります。

8. その他

□ プロジェクト設定

ユーザーファーム開発におけるプロジェクト設定は専用の設定を必要とします。新しいプロジェクトを作成する場合には「A0x0xProjects¥_UserFirmBase」フォルダをフォルダごとコピーし、内容を修正することを推奨します。

もし、新規にプロジェクトを作成したい場合には、[プロジェクトの設定]画面から[他のプロジェクトからコピー]ボタンを押し、「A0x0xProjects¥_UserFirmBase¥MyUserFirm.yip」から設定をコピーするようにしてください。

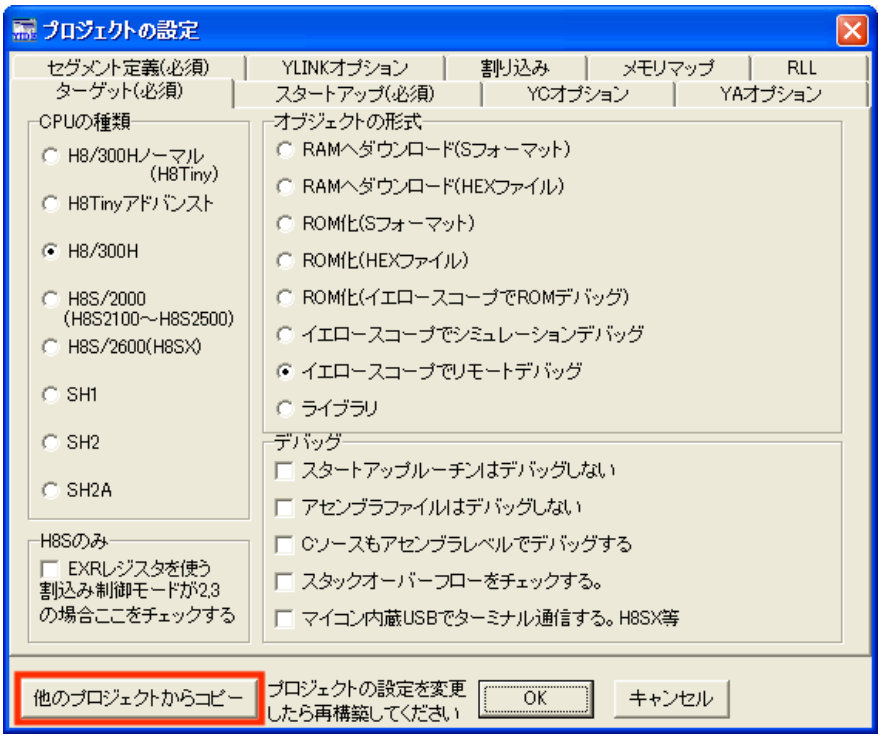


図 51　プロジェクト設定のコピー

- プロジェクトをフォルダごとコピーすると、コピー元プロジェクトで開かれていたソースファイルが、コピー先のプロジェクトでも開かれたままになります。この開かれたままのソースファイルはコピー先フォルダのファイルではなく、コピー元フォルダのファイルを指していますので、誤って修正するとコピー元のファイルが変更されてしまいます。プロジェクトをコピーした場合は、コピー先プロジェクトを初めて開いたときに、表示されているソースファイルを一度閉じてください。

□ スタック

スタックはローカル変数や、関数の引数などに利用される RAM 領域です。ユーザーファームで使用するスタック領域はシステムファームと共通となっており、ローカル変数として大きな配列を定義した場合などには、スタックの容量が不足し誤動作する可能性があります。[プロジェクトの設定]画面の[ターゲット]タブにある、[スタックオーバーフローをチェックする]という項目にチェックを入れると、関数の先頭にスタック容量をチェックするためのプログラムが自動的に挿入され、問題がある場合はデバッグ時に表示されます。

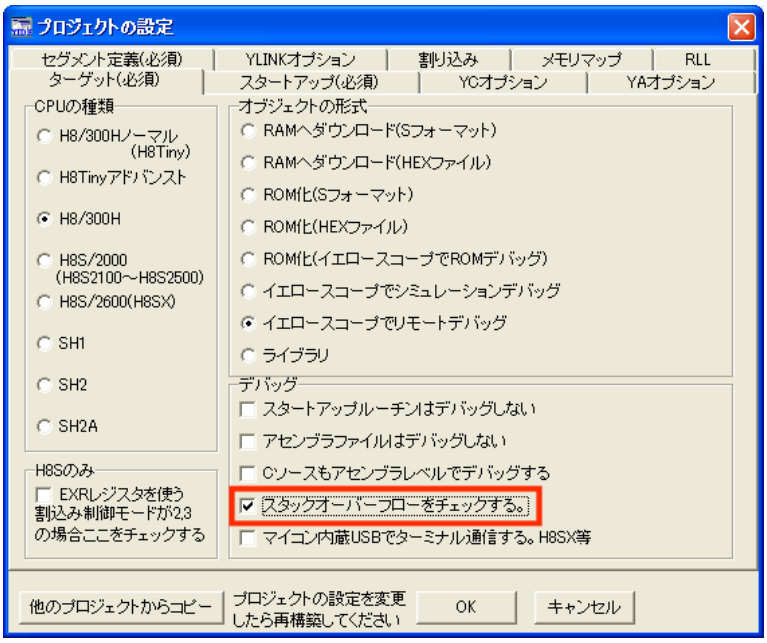


図 52　スタックオーバーフローのチェック設定

プロジェクト設定によるスタックチェックで使用されるスタック容量は仮の値となっており、実際に使用できるスタック容量よりも少ない値となります。真のスタック空き容量は *SRV_GetStackSize()* 関数を使用することで取得することができます。使用できるスタック容量は、デバイスのインタフェース種別やファームウェアのバージョンによって異なります。表 38 はシステムファーム Ver.5.6.1 の初期スタック容量です。

表 38　初期スタック容量

インタフェース種別	スタック容量
LAN インタフェースデバイス	約 3400 バイト
USB インタフェースデバイス	約 4700 バイト

- RLL を利用する場合(特にライブラリ関数全てを ROM 化する設定とした場合)は、ライブラリが要求するメモリ領域を確保するため、デバッグ時の初期スタック容量が小さくなります。
- RLL のうち実際には使用されない関数のために確保されていたメモリ領域は、ユーザーファームを ROM 化したときに解放されるため、スタックはデバッグ時よりも多く使用できます。

□ アタッチメントファームから RLL を利用する方法

29 ページではデバッグ時に RLL(Rom Link Library)を利用する手順を説明しましたが、アタッチメントファームの実行時にも RLL を利用することが可能です。関数の一部をフラッシュメモリに書き込んでおくことで RAM にダウンロードするプログラムサイズを小さくすることができます。

- フラッシュメモリにはアタッチメントファーム作成時に使用したサブプロジェクトの出力が書き込まれていなくてはいけません。ATF ファイルのダウンロード時に RLL との整合性がチェックされます。
- コンパイラのバージョンアップやプロジェクト内容の変更でサブプロジェクトの出力結果が少しでも変わってしまうと、既に書き込まれている RLL のコードと新たに作成した ATF ファイルの整合が取れなくなり、ダウンロードができなくなります。

1. 『YellowIDE』でアタッチメントファームのプロジェクトを開きます。
2. [プロジェクトウィンドウ]の[設定]ボタンを押し[RLL]タブを選びます。[ROM リンクライブラリを使用する]にチェックを入れます。
3. [ROM 化の選択]では[標準関数ライブラリ、サブプロジェクト関数全てを ROM 化する]の方を選択してください。必要な関数だけを ROM 化すると、メインプロジェクトから呼び出される関数が変更されるだけでサブプロジェクトが修正され、アタッチメントファームの修正や変更のたびにフラッシュメモリへの書き込みが必要になります。
4. [プロジェクトウィンドウ]の[Object]は"RAM へダウンロード(S)"を選択します。
5. [ファイル]メニューの[サブプロジェクトを開く]から「A0x0xProjects¥_ATF_RLL¥ATF_RLL.yip」を開きます。
6. [サブプロジェクト]ウィンドウの[メイク]ボタンを押します。
7. 『YellowIDE』の[メイク]ボタンでメインプロジェクトをメイクします。
8. 「ATF Maker」を起動し、ATF ファイルを作成します。
9. 作成した ATF ファイルをダウンロードできるようにデバイスに RLL を書き込みます。デバイスのディップスイッチ 2 番を"ON"にし起動します。ケーブルを接続し、パソコンと通信できる状態としてください。
10. 「M3069FlashWriter」を起動し、「A0x0xProjects¥_ATF_RLL¥ATF_RLL.S」を書き込みます。
11. ディップスイッチ 1 番を"ON"、2 番を"OFF"としデバイスを再起動します。

以上の手順でアタッチメントファームから RLL を利用できるようになります。

- サブプロジェクトにはライブラリが使用するメモリ領域を確保するための処理が含まれています。

9. サービス関数リファレンス

この章はシステムファームが提供するサービス関数のリファレンスです。内容は、「システムファーム Ver.5.6.x」を元に作成されています。

各関数の説明は、C 言語のプロトタイプ、変数、戻り値の説明、動作説明の順になっています。戻り値が *SRV_STATUS* となっている関数は 16 ビットの整数で実行結果を返します（以下参照）。関数がそれ以外の特別な戻り値を返す場合は、各関数の動作説明の欄で内容を示します。

構造体が使用される場合には、各関数分類の先頭で説明しています。

□ 戻り値の意味

以下に戻り値の意味を示します。これらは「service.h」の中で定義されています。

表 39　関数の戻り値

定数	意味
SRVS_OK	正常終了
SRVS_TIMEOUT	処理がタイムアウトした
SRVS_INVALID_ARGS	無効なパラメータ
SRVS_NOT_SUPPORTED	サポートされない機能
SRVS_NO_CONNECTION	ピアホストとの通信が切れている(LAN デバイスのみ)
SRVS_SOCKET_ERROR	ソケット関数のエラー(LAN デバイスのみ)
SRVS_INVALID_MAC	MAC アドレスが正しくない(LAN デバイスのみ)
SRVS_INVALID_IP	IP アドレスが正しくない(LAN デバイスのみ)
SRVS_DNS_FAILURE	DNS の失敗(LAN デバイスのみ)
SRVS_INVALID_SOCKET	ネットワークチャンネルの指定が不正(LAN デバイスのみ)
SRVS_NO_CONFIG	ネットワークのコンフィギュレーション情報が存在しない(LAN デバイスのみ)
SRVS_BUSY	使用中のため命令が実行できない
SRVS_NOT_ENABLED	指定の機能が有効になっていない
SRVS_SOCKET_PENDING	ネットワーク関連処理の完了を待たずに終了した
SRVS_OTHER_ERROR	その他のエラー

□ 汎用関数

SRV_GetVersion

DWORD SRV_GetVersion()

戻り値 : システムファームのバージョン番号

システムファームのバージョン番号を返します。バージョン番号は 32 ビットの数値で最上位 8 ビットが予約。以降 8 ビット毎にメジャーバージョン、マイナーバージョン、リビジョンの順です。

ビット	31-24	23-16	15-8	7-0
意味	予約	メジャーバージョン	マイナーバージョン	リビジョン

ファームウェアのバージョンが 5.4.1 の場合、格納される値は 0x00050401 となります。
※この関数は割り込みハンドラから呼び出すことができます。

SRV_GetStackSize

long SRV_GetStackSize()

戻り値 : スタックのサイズ

スタックの残り容量を計算します。この関数で返される値はシステムファームのバージョンによって異なる可能性があります。
※この関数は割り込みハンドラから呼び出すことができます。

SRV_SetVect

void (*SRV_SetVect(short Number, void (*pNewHandler)()))()

Number : 割り込み番号
pNewHandler : ハンドラへのポインタ

戻り値 : 古いハンドラへのポインタ

指定の割り込み番号にハンドラを登録し、古いハンドラへのポインタを返します。

SRV_SetMain

void (*SRV_SetMain(void (*pMain)(void)))(void)

pMain : 登録するメイン関数へのポインタ

戻り値 : 古いメイン関数へのポインタ

メイン関数のアドレスを登録します。メイン関数はシステムファームのコマンドループの中から定期的に呼び出される関数です。ホストパソコンからのコマンドと無関係に自律的な処理を行う場合に使用します。メイン関数からリターンしないとコマンドループが実行されないため、TWXA ライブラリからのコマンドは受け付けられません。メイン関数のプロトタイプは以下です。

void *FunctionName*(void);

SRV_SetCommand

```
void (*SRV_SetCommand(void (*pCommand)(WORD, DWORD, DWORD)))(WORD, DWORD, DWORD)
```

pCommand : 登録するコマンドハンドラへのポインタ

戻り値　 : 古いコマンドハンドラのアドレス

コマンドハンドラのアドレスを登録します。TWXA ライブラリから TWXA_ATFUserCommand()を呼び出したときに、ユーザーファーム側で呼び出される関数を登録します。コマンドハンドラのプロトタイプは以下です。

```
void FunctionName(WORD Command, DWORD Param1, DWORD Param2);
```

Command : ユーザー定義のコマンド番号
Param1　: ユーザー定義のパラメータ 1
Param2　: ユーザー定義のパラメータ 2

SRV_InitVect

```
void SRV_InitVect()
```

デバッグモニタで使用するベクタ番号 10、11、57 の割り込みを除いて、全ての割り込みベクタを初期状態に戻します。

SRV_EnableInt

```
void SRV_EnableInt(int Pri)
```

Pri : 割り込み許可するプライオリティレベルを指定します
　　SRV_INT_ENABLE　　: 全て許可
　　SRV_INT_DISABLE　 : 全て禁止
　　SRV_INT_ENABLE_PRI1 : プライオリティ 1 のみ許可

割り込みの許可／禁止を設定します。
※この関数は割り込みハンドラから呼び出すことができます。

SRV_WdEnable

```
void SRV_WdEnable(BOOL flgEnable)
```

flgEnable : ウォッチドッグタイマの許可/禁止を指定します
　　TRUE　: ウォッチドッグタイマによるリセットを許可
　　FALSE : ウォッチドッグタイマによるリセットを禁止

ウォッチドッグタイマによるリセットの許可／禁止を設定します。ウォッチドッグタイマによるリセットを許可した場合、41.9msec 以下の間隔で SRV_StimeUpdate()関数を呼び出し、タイマカウンタをクリアしてください。カウンタクリアが正しく行われず、オーバーフローが発生するとリセットが発生し、システムは再起動されます。
デバッグ中はブレーク(中断)が発生した時点でリセットされてしまいますので使用できません。また、printf() などの実行時間の長い関数を使用するとリセットが発生しやすくなりますので注意してください。

SRV_GetProfileString

```
char *SRV_GetProfileString(DWORD Address, char *pSection, char *pParam,
                           int *pnStr, char *pDefStr, int nDefStr)
```

Address : INI データの先頭アドレス
SRV_EB1_ADDRESS : フラッシュメモリブロック 1
SRV_EB2_ADDRESS : フラッシュメモリブロック 2
SRV_EB3_ADDRESS : フラッシュメモリブロック 3
pSection : [入力]セクション名
pParam : [入力]パラメータ名
pnStr : [出力]戻り値文字列のバイト数。不要な場合 NULL とできます。
pDefStr : [入力]パラメータが見つからない場合やエラー時に戻り値とする値
nDefStr : パラメータが見つからない場合やエラー時に pnStr に格納する値

戻り値 : パラメータ値文字列へのポインタ

M3069IniWriter で書き込んだパラメータ値を文字列として読み出します。パラメータが見つからない場合は pDefStr に渡されたポインタを返します。

SRV_GetProfileInt

```
DWORD SRV_GetProfileInt(DWORD Address, char *pSection, char *pParam, DWORD Default)
```

Address : INI データの先頭アドレス
SRV_EB1_ADDRESS : フラッシュメモリブロック 1
SRV_EB2_ADDRESS : フラッシュメモリブロック 2
SRV_EB3_ADDRESS : フラッシュメモリブロック 3
pSection : [入力]セクション名
pParam : [入力]パラメータ名
Default : パラメータが見つからない場合に戻り値とする値

戻り値 : パラメータ値を 32bit 符号無し整数に変換した結果

M3069IniWriter で書き込んだパラメータ値を 32 ビット符号無し整数に変換して読み出します。
パラメータが見つからない場合は Default に渡された値を返します。
パラメータ値が 0x で始まる場合は 16 進数、その他は 10 進数として解釈されます。

SRV_EnumProfileParam

```
char *SRV_EnumProfileParam(DWORD Address, char *pSection, char *pPrevParam)
```

Address : INI データの先頭アドレス
SRV_EB1_ADDRESS : フラッシュメモリブロック 1
SRV_EB2_ADDRESS : フラッシュメモリブロック 2
SRV_EB3_ADDRESS : フラッシュメモリブロック 3
pSection : [入力]セクション名
pPrevParam : [入力]前回の戻り値。初回の場合 NULL を指定する

戻り値 : パラメータ名文字列

M3069IniWriter で書き込んだセクション内のパラメータ名を列挙します。初回は pPrevParam = NULL を渡します。以降は前回の戻り値を pPrevParam に渡すことで順番にパラメータ名を取り出すことができます。セクションの終わりに達した場合は NULL が返ります。

□ システムタイマ関数

SRV_StimeUpdate

DWORD SRV_StimeUpdate(void)

戻り値 : 約 83.9msec 毎にインクリメントされる 32 ビットのカウント値

システムタイマのカウントを更新します。システムタイマは 8 ビットタイマのチャンネル 2 を使って起動からの経過時間を記録します。タイマのオーバーフローを監視し、8192×256×40nsec 毎に 32 ビットのカウンタ値を 1 ずつインクリメントします。
デフォルトの状態ではオーバーフローのチェックが自動ではありませんので、83.9msec 以内にこの関数を呼び出してカウンタの更新を行ってください。

SRV_StimeGetCnt

DWORD SRV_StimeGetCnt(void)

戻り値 : 約 83.9msec 毎にインクリメントされる 32 ビットのカウント値

システムタイマのカウントを取得します。返される値は SRV_StimeUpdate()と同じですが、この関数ではカウンタの更新を行いません。
※この関数は割り込みハンドラから呼び出すことができます。

SRV_StimeSetAutoUpdate

void SRV_StimeSetAutoUpdate(BOOL flgAutoUpdate)

flgAutoUpdate : 自動更新の許可/禁止
TRUE : 許可
FALSE : 禁止

システムタイマの自動更新を設定します。自動更新にすると定期的に割り込み処理が実行されます。

SRV_StimeGetTime

DWORD SRV_StimeGetTime(DWORD *pHigh)

pHigh : [出力]32 ビットより上位の桁の格納先

システム起動後の経過時間を msec 単位で返します。
pHigh には 32 ビットより上位の桁が返されます。不要なときは NULL とすることができます。
戻り値を *Low* とすると経過時間は以下の式で計算できます。

*pHigh * (2 ^ 32) + Low [msec]

経過時間は約 4169 日で 0 に戻ります。

SRV_StimeSleep

void SRV_StimeSleep(DWORD Millisec)

Millisec : スリープ時間を msec 単位で指定します

指定時間スリープします。スリープ中はシステムタイマの更新は内部で行われます。

SRV_GetTime

long SRV_GetTime(long *timeptr)

timeptr : [出力]時刻の格納先

戻り値 : timeptr に返される値と同じ

システムのカレンダー時計の時刻を取得します。時刻は 1970/1/1 から起算した秒数を UTC で返します。ANSI の time()関数と同様の機能です。

SRV_SetTime

SRV_STATUS SRV_SetTime(long Time)

Time : 設定する時刻(正の値のみ)

システムのカレンダー時計の時刻を設定します。時刻は 1970/1/1 から起算した秒数を UTC で渡します(ANSI の time_t 変数と同様の形式)。システム内部の時刻は NTP プロトコルのタイムスタンプと同様、1900 年から起算した秒数で管理されるため、2036 年 2 月 7 日 6 時 28 分 15 秒より後の時刻は設定できません。

□ インタフェース関数

ホストパソコンの TWXA ライブラリと通信するための関数です。

SRV_IsTXE

short SRV_IsTXE()

戻り値 ： 送信バッファに書き込み可能なバイト数

USB デバイスは送信バッファに書き込み可能な最低バイト数を返します。返される値は 1,64,128,512,1024 のいずれかで、少なくとも返されたバイト数まではブロックせずに書き込みが可能です。
LAN デバイスでは送信バッファの空きをバイト単位で返します。

SRV_IsRXF

short SRV_IsRXF()

戻り値 ： 受信バッファから読み出し可能なバイト数

USB デバイスでは受信バッファから取り出し可能な最低バイト数を返します。返される値は 1,64,128,512,1024 のいずれかで、少なくとも返されたバイト数まではブロックせずに読み出しが可能です。
LAN デバイスでは受信バッファのデータ数をバイト単位で返します。

SRV_Transmit

SRV_STATUS SRV_Transmit(void *pSrc, WORD n, short Inc)

pSrc : [入力]送信データ
n : データのバイト数(0 のとき 65536 バイト送信)
inc : インクリメント
1 転送毎に pSrc をインクリメント
0 pSrc は固定
-1 転送毎に pSrc をデクリメント

指定のデータをホストに送信します。ホストパソコンでは送信されたデータを TWXA ライブラリの TWXA_Read()関数で取り出すことができます。**n = 0 のとき 65536 バイトの送信となりますので注意してください。この関数の使用は推奨されません。TWFA_Transmit()を使用してください。**

SRV_Receive

SRV_STATUS SRV_Receive(void *pDst, WORD n, short Inc)

pDst : [出力]データの格納先
n : データのバイト数(0 のとき 65536 バイト受信)
Inc : インクリメント
1 転送毎に pDst をインクリメント
0 pDst は固定
-1 転送毎に pDst をデクリメント

ホストパソコンから TWXA_Write()で送信されたデータ、または、TWXA_ATFUserCommand()の追加データとして送信されたデータを受信する場合に呼び出します。**n = 0 のとき 65536 バイトの受信となりますので注意してください。この関数の使用は推奨されません。TWFA_Receive()を使用してください。**

SRV_SetTimeouts

void SRV_SetTimeouts(WORD TxTimeout, WORD RxTimeout)

TxTimeout : 送信タイムアウト(msec 単位)
RxTimeout : 受信タイムアウト(msec 単位)

送信および受信のおおよそのタイムアウト時間を設定します。

SRV_GetHsIfStatus

WORD SRV_GetHsIfStatus()

戻り値 : USBインタフェースの状態
ビット0 : ハイスピード接続のとき1、フルスピード接続のとき0になります
ビット1 : 常に1
その他 : 常に0

USB インタフェースの状態を返します。返されるのはホストパソコンから最後に接続されたときの状態です。現在、接続されているか、切断されているかを調べることはできません。

SRV_TransmitEvent

SRV_STATUS SRV_TransmitEvent(DWORD Message, DWORD WParam, DWORD LParam)

Message : ホストパソコンのアプリケーションに通知するメッセージ番号
WParam : ホストパソコンのアプリケーションに渡すパラメータ
LParam : ホストパソコンのアプリケーションに渡すパラメータ

Windows 上のアプリケーションにメッセージを送ります。
Windows 上のアプリケーションプログラムでは予め TWXA_SetHwEvent()関数を呼び出す必要があります。
Message を 0 とすると TWXA_SetHwEvent()で指定したメッセージ番号で通知されます。
送信バッファに十分な空きがない場合には SRVS_BUSY が返り、送信は行われません。
LAN インタフェースの場合はリストアップチャンネルを使用している必要があります。

□ LAN デバイス制御関数

LAN デバイスの制御やネットワーク設定に使用する関数です。

LANM_CONFIG 構造体

```
typedef struct {
  BYTE MAC[6];              //デバイスの MAC アドレス
  BYTE IP[4];               //デバイスの IP アドレス
  BYTE Gateway[4];          //デフォルトゲートウェイの IP アドレス
  BYTE Subnetmask[4];       //サブネットマスク
  WORD Port;                //サーバーモードでホストパソコンの接続を待つポート番号
  BYTE rsv1[32];            //予約
  BYTE DnsServer[4];        //DNS サーバー
  short rsv2;               //予約
  char NtpServerName[32]; //NTP サーバー
  WORD Option;              //クライアントモードの場合ビット 2 が 1 となる
  char ServerName[32];      //クライアントモードの接続先 IP アドレス
  WORD ServerPort;          //クライアントモードの接続先ポート番号
} LANM_CONFIG;
```

付属ツールを使用してフラッシュメモリに書き込まれた、ネットワーク設定を読み出す際に使用する構造体です。NTP サーバー名とクライアントモード時の接続先は、ドメイン指定の場合はドメイン名が格納されます。IP 指定の場合、先頭 2 バイトが 0 となり続く 4 バイトに IP アドレスが格納されます。

SRV_LanmInit

SRV_STATUS SRV_LanmInit(DWORD Option)

Option : LAN デバイスの動作設定。以下の値を OR で組み合わせて指定
LANMM_ENABLE_CONTROL : 制御チャンネルを使用する
LANMM_ENABLE_LIST : パソコンからのリストアップ要求に応答する

LAN デバイスに必要な初期化作業全般を行います。フラッシュメモリの保存領域からネットワーク設定を読み取り、W3150A+[8]チップを初期化します。また、DHCP を利用する設定となっている場合には、DHCP 関連の初期化作業も行います。

ネットワーク設定・維持を自動的に行う場合、最初にこの関数を呼び出します。その後、定期的に SRV_LanmCheckState()を呼び出してください。

システムでは W3150A+のチャンネルを最大 3 チャンネル使用します。1 つはホストパソコンの TWXA ライブラリと通信するための制御チャンネル。2 つめはホストパソコンがネットワーク上のデバイスを探すために使用するリストアップ用のチャンネル。3 つめは DHCP を利用するための DHCP 用チャンネルです。

制御チャンネルはチャンネル 0 を使用します。LANMM_ENABLE_CONTROL を指定しない場合、ユーザーが使用できますが、SRV_Transmit()、SRV_Receive()などの関数は使用できません。また、TWXA ライブラリによる制御もできなくなります。

リストアップ用チャンネルは、DHCP を使用しない場合チャンネル 1、DHCP を使用する場合チャンネル 2 を使用します。LANMM_ENABLE_LIST を指定しない場合、該当チャンネルを自由に使用できますが、TWXA ライブラリから検索とハードウェアイベントの通知ができなくなります。この場合、デバイスへの接続には IP アドレスの指定が必須になります。また、ホストパソコンから接続され、ステートが LANMS_CONNECT となっている間はこのチャンネルは使用されませんので、一時的に他の用途で使用することができます。

DHCP 用チャンネルは必要な場合チャンネル 1 が使用されます。

TWXA ライブラリとの通信が必要ない場合は Option を 0 としてください。その場合、チャンネル 0,2,3 が自由に使用できます。チャンネル 1 が使用できるかは DHCP の要否によります。

SRV_LanmInitA

SRV_STATUS SRV_LanmInitA(LANM_CONFIG *pConfig, DWORD Option)

pConfig : ネットワーク設定情報をセットした LANM_CONFIG 構造体
Option : LAN デバイスの動作設定。以下の値を OR で組み合わせて指定
LANMM_ENABLE_CONTROL : 制御チャンネルを使用する
LANMM_ENABLE_LIST : パソコンからのリストアップ要求に応答する

SRV_LanmInit()と同様の機能ですが、LANM_CONFIG 構造体によって IP アドレスなどのネットワーク設定を指定することができます。
ネットワーク設定ツール(LANM_Config.exe)による設定はパスワードを除いて無視されます。
DHCP を利用する場合は LANM_CONFIG の IP を全て 0 とします。

[8] LAN インタフェース製品に内蔵する WIZnet 社のネットワーク用 IC です。

SRV_LanmCheckState

SRV_STATUS SRV_LanmCheckState(WORD *pLanmState)

pLanmState : [出力]LAN デバイスの状態が格納されます。DHCP 使用の場合、以下のいずれかに加えて LANMS_DHCP_FLAG(0x8000)ビットが"1"になります。

LANMS_INIT(0x0001)	: 初期状態。ネットワークは使用できません
LANMS_READY(0x0002)	: ネットワークは使用可能。制御チャンネルは使用不可
LANMS_CONNECT(0x0004)	: 制御チャンネルがホストパソコンと接続された状態
LANMS_DISCONNECT(0x0008)	: ホストパソコンとの接続が切断された状態

LAN デバイスの状態をチェックします。関数内で以下の作業を行います。
・制御チャンネルによるホストパソコンとの接続/切断処理
・KeepAlive による制御チャンネルの状態チェック
・DHCP の更新時刻管理と更新作業
・システムタイマの更新

通常は制御チャンネルや DHCP の管理のために、システムファーム内で自動的に呼び出されています。メイン関数でループし、システムファームに処理を戻さない場合は明示的に呼び出す必要があります。

LANMS_INIT を除く状態ではソケット関数などを使用してネットワークにアクセスすることができますが、DHCP を利用している場合は一度ネットワークが使用可能になった後でも再び LANMS_INIT に戻る可能性があるのでチェックする必要があります。

LANMS_CONNECT の状態では TWFA_Transmit()、TWFA_Receive()を使用して TWXA ライブラリとの通信が可能です。

LANMS_DISCONNECT は一時的な状態です。ホストパソコンから接続が切れたことを示します。

SRV_LanmReadConfig

SRV_STATUS SRV_LanmReadConfig(LANM_CONFIG *pConfig)

pConfig : [出力]LANM_CONFIG 構造体へのポインタ

内蔵フラッシュ内にネットワークのコンフィギュレーション情報があれば読み出します。結果は LANM_CONFIG 構造体で受け取ります。このコンフィギュレーション情報は付属のネットワーク設定ツール(LanmConfig.exe)で設定できます。情報が無い場合、SRVS_NO_CONFIG を返します。

SRV_SyncTime

SRV_STATUS SRV_SyncTime(int CH, BYTE *pServerAddr, DWORD Timeout);

CH	: 使用するチャンネル番号(0～3)
pServerAddr	: [入力]同期する NTP サーバーの IP アドレス(4 バイト)
Timeout	: タイムアウト時間(msec 単位)

SNTP プロトコルを使用して指定の NTP サーバーと、システムのカレンダー時計を同期します。pServerAddr を NULL とすると予め登録された複数のサーバーとランダムに同期しますが、特定のサーバーに対する負荷を避けるため、可能であればローカルサーバーなどを利用するようにしてください。

□ ソケット関数

ネットワーク関連のサービス関数です。ここであげる関数はすべて LAN デバイス専用となります。

SRV_SockOpen

```
SRV_STATUS SRV_SockOpen(int CH, int Protocol, WORD Port, int Option)

  CH       : チャンネル番号(0～3)
  Protocol : プロトコル
             SOCK_STREAM(0x01) : TCP 用に初期化します
             SOCK_DGRAM (0x02) : UDP 用に初期化します
  Port     : ローカルのポート番号
  Option   : オプション
             SOCKOPT_NDACK : "Delayed Ack"を使用しない(TCP 用)
```

チャンネルを指定プロトコル用に初期化します。ローカルのポート番号は必ず指定してください。指定チャンネルが使用中の場合は SRVS_INVALID_SOCKET が返ります。
TCP 用に初期化する場合に SOCKOPT_NDACK を指定すると、受信パケットに対する応答をすぐに返すようになります。小さなデータを頻繁に受信する場合に指定するとアクセス時間が短くなります。

SRV_SockClose

```
void SRV_SockClose(int CH)

  CH : チャンネル番号(0～3)
```

指定チャンネルをクローズします。

SRV_SockConnectA

```
SRV_STATUS SRV_SockConnectA(int CH, BYTE *pDstAddr, WORD Port, int Option)

  CH       : チャンネル番号(0～3)
  pDstAddr : [入力]接続先のアドレス
  Port     : 接続先のポート
  Option   : オプション
             SOCK_NB : 接続完了を待たずに戻ります
```

TCP チャンネルを指定アドレスの指定ポートに接続します。
pDstAddr には IP アドレスを上位から指定します。例えば"192.168.0.1"に接続する場合、以下のようになります。

```
  pDstAddr[0] = 192, pDstAddr[1] = 168, pDstAddr[2] = 0, pDstAddr[3] = 1
```

Option 引数に SOCK_NB を指定すると接続完了を待たずに戻ります。その場合、戻り値は SRVS_SOCK_PENDING が返ります。接続処理の完了は SRV_SockReadStatus()の戻り値で調べてください。戻り値が SOCKS_CLOSED となった場合接続失敗、SOCKS_ESTABLISHED となった場合接続が成功したことを示します。

SRV_SockDisconnectA

```
void SRV_SockDisconnect(int CH, int Option)
```

```
  CH     : チャンネル番号(0～3)
  Option : オプション
           SOCK_NB : 切断完了を待たずに戻ります
```

TCP チャンネルの接続を切断してクローズします。
Option 引数に SOCK_NB を指定すると接続相手との切断完了を待たずに戻ります。その場合、戻り値は SRVS_SOCK_PENDING がかえります。切断の完了時は SRV_SockReadStatus()の戻り値が SOCKS_CLOSED となります。

SRV_SockListen

```
SRV_STATUS SRV_SockListen(int CH)
```

```
  CH : チャンネル番号(0～3)
```

TCP チャンネルを接続待ち状態にします。UDP 用にオープンされたチャンネルや、既に接続されているチャンネルの場合 SRVS_INVALID_SOCKET が返ります。

SRV_SockSendTo

```
WORD SRV_SockSendTo(int CH, void *pBuff, WORD nBuff, BYTE *pDstAddr, WORD Port)
```

```
  CH       : チャンネル番号(0～3)
  pBuff    : [入力]送信データ
  nBuff    : 送信データのバイト数
  pDstAddr : [入力]送信先アドレス
  Port     : 送信先ポート

  戻り値   : 送信したバイト数
```

UDP チャンネルからデータを送信します。送信完了後に関数から戻ります。

SRV_SockRecvFrom

```
WORD SRV_SockRecvFrom(int CH, void *pBuff, WORD nBuff, BYTE *pSrcAddr, WORD *pPort)
```

```
  CH       : チャンネル番号(0～3)
  pBuff    : [出力]受信データの格納先
  nBuff    : pBuff に格納可能なバイト数
  pSrcAddr : [出力]送信元アドレスの格納先(4 バイト)
  pPort    : [出力]送信元ポートの格納先

  戻り値   : 読み出したバイト数
```

UDP チャンネルの受信バッファにパケットがあれば読み出します。nBuff がパケットのデータバイト数より小さい場合、nBuff の長さだけ pBuff にコピーされ、残りのデータは捨てられます。

SRV_SockSend

```
WORD SRV_SockSend(int CH, void *pBuff, WORD nBuff)
```

CH　　: チャンネル番号(0～3)
pBuff　: [入力]送信データの格納先
nBuff　: 送信するバイト数(最大 2048)

戻り値 : 送信したバイト数

TCP チャンネルからデータを送信します。送信バッファに格納可能なバイト数だけ、送信処理を行いすぐに戻ります。1 度に送信できるデータは最大 2048 バイトです。

SRV_SockRecv

```
WORD SRV_SockRecv(int CH, void *pBuff, WORD nBuff)
```

CH　　: チャンネル番号(0～3)
pBuff　: [出力]受信データの格納先
nBuff　: 読み出すデータのバイト数

戻り値 : 読み出したバイト数

TCP チャンネルの受信バッファにデータがあれば読み出します。受信バイト数が nBuff よりも小さい場合は受信したバイト数だけ読み出します。

SRV_SockPeek

```
WORD SRV_SockPeek(int CH, void *pBuff, WORD nBuff)
```

CH　　: チャンネル番号(0～3)
pBuff　: [出力]受信データの格納先
nBuff　: 読み出すデータのバイト数

戻り値 : 読み出したバイト数

TCP チャンネルの受信バッファにデータがあれば読み出します。受信バイト数が nBuff よりも小さい場合は受信したバイト数だけ読み出します。SRV_SockRecv() と違い、読んだデータはそのまま受信バッファに残ります。必要の無くなったデータは SRV_SockPurge()で削除してください。

SRV_SockPurge

```
WORD SRV_SockPurge(int CH, WORD nPurge)
```

CH　　: チャンネル番号(0～3)
nPurge : 削除するバイト数

戻り値 : 削除したバイト数

TCP チャンネルの受信バッファから指定バイト数のデータを削除します。受信バイト数が nPurge よりも小さい場合は受信したバイト数だけ削除します。

SRV_SockReadStatus

WORD SRV_SockReadStatus(int CH)

CH　　: チャンネル番号(0～3)

戻り値 : チャンネルのステータス。以下の値をとります
　　SOCKS_CLOSED　　: ソケットはクローズしている
　　SOCKS_INIT　　: 接続前の TCP ソケット
　　SOCKS_LISTEN　　: ソケットは接続待ち
　　SOCKS_SYNSENT　　: 接続処理中の一時的な状態
　　SOCKS_SYNRECV　　: 接続処理中の一時的な状態
　　SOCKS_ESTABLISHED : 接続中
　　SOCKS_FIN_WAIT1　: 終了処理中の一時的な状態
　　SOCKS_FIN_WAIT2　: 終了処理中の一時的な状態
　　SOCKS_CLOSING　　: 終了処理中の一時的な状態
　　SOCKS_TIME_WAIT　: 終了処理中の一時的な状態
　　SOCKS_CLOSE_WAIT　: 接続相手から切断要求がある状態
　　SOCKS_LAST_ACK　　: 終了処理中の一時的な状態
　　SOCKS_UDP　　: ソケットは UDP 用

チャンネルのステータスを読み出します。

SRV_SockGetHostByName

SRV_STATUS SRV_SockGetHostByName(int CH, char *pName, BYTE *pIP, DWORD Timeout)

CH　　: DNS で使用するチャンネル番号(0～3)
pName　: [入力]ホストの名前
pIP　　: [出力]ホストの IP アドレス
Timeout : タイムアウトまでの時間(msec 単位)

DNS を利用してホスト名に対応する IP アドレスを取得します。DNS サーバーのアドレスが設定されていない場合には SRVS_NO_CONFIG が返ります。ネットワークチャンネルは DNS サーバーとの通信のため、一時的に使用されます。関数から戻った後は自由に使用できます。

SRV_SockGetHostByNameA

SRV_STATUS SRV_SockGetHostByNameA(int CH, char *pName, BYTE *pIP,
　　DWORD Timeout, int Option)

CH　　: DNS で使用するチャンネル番号(0～3)
pName　: [入力]ホストの名前
pIP　　: [出力]ホストの IP アドレス
Timeout : タイムアウトまでの時間(msec 単位)
Option　: オプション
　　SOCK_NB : 応答を待たずに戻ります

DNS を利用してホスト名と対応する IP アドレスを取得します。DNS サーバーのアドレスが設定されていない場合には SRVS_NO_CONFIG が返ります。
Option に SOCK_NB を指定するとサーバーからの応答待ちの間、関数は SRVS_SOCKET_PENDING を返します。その場合 SRVS_OK が返るまで繰り返し関数を呼び出してください。関数はサーバーからの応答を受信すると pIP にアドレスをセットして SRVS_OK を返します。
Timeout に指定した時間が経過すると関数は SRVS_TIMEOUT を返し、リクエストは失敗となります。次の呼び出しでは再びサーバーにリクエストを送信します。

10. TWFA ライブラリ・リファレンス

□ 初期化／デバイス情報取得用関数

TWFA_Initialize

```
SRV_STATUS TWFA_Initialize(long Opt)
```

Opt ： 初期化する機能を指定。以下の値を OR で結合
TWFA_INIT_PORT_DATA ： 出力ポートの値を初期化
TWFA_INIT_TIMER ： 16 ビットタイマを初期化
TWFA_INIT_SCI ： シリアルポートの初期化
TWFA_INIT_PC ： カウンタの初期化
TWFA_INIT_DA ： DA の初期化
TWFA_INIT_ALL ： 上記機能全てを初期化

デバイスの初期化を行います。ATF_Init()内で Opt = TWFA_INIT_ALL として必ず呼び出してください。

TWFA_A0x0xInit

```
SRV_STATUS TWFA_A0x0xInit(long Opt)
```

Opt ： 予約。0 としてください

X-A0x0x デバイス固有の機能を初期化します。本関数の前に TWFA_Initialize() 関数を必ず呼び出してください。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_GetDeviceNumber

```
WORD TWFA_GetDeviceNumber()
```

戻り値 ： デバイスの装置番号

デバイスに付与された装置番号を返します。
※割り込みハンドラから呼び出すことができます。

□ ポート操作関数

TWFA_PortWrite

void TWFA_PortWrite(DWORD Port, BYTE Data, BYTE Mask)

Port ： 書き込み先の指定
TWFA_USER_STATUS ： ユーザー用ステータスレジスタへ書き込み

Data ： 書き込むデータ
Mask ： 操作するビットを指定するマスク(1 と対応するビットのみ影響を受ける)

ユーザー用ステータスレジスタやユーザーメモリなどの領域への書き込みに使用します。Mask 引数は特定のビットだけを操作する場合に使用します。
Mask 引数の 1 となっているビットのみ書き込みの影響を受けます。
※割り込みハンドラから呼び出すことができます。

TWFA_PortRead

BYTE TWFA_PortRead(DWORD Port)

Port ： 読み出し対象を指定
TWFA_USER_STATUS ： ユーザー用ステータスレジスタの読み出し

戻り値 ： 読み出したデータ

ユーザー用ステータスレジスタやユーザーメモリなどの領域の読み出しに使用します。
※割り込みハンドラから呼び出すことができます。

□ アナログ入力／アナログ値変換関数

TWFA_ADSetRange

SRV_STATUS TWFA_ADSetRange(int Range)

Range : アナログ入力端子の入力レンジを指定。以下のいずれか。

TWFA_AN_10VPP : 入力レンジを-5～+5[V]に設定
TWFA_AN_20VPP : 入力レンジを-10～+10[V]に設定

アナログ入力端子の入力レンジを設定します。
連続変換が開始されている場合、エラーが返ります。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_ADGetRange

SRV_STATUS TWFA_ADGetRange(int *pRange)

pRange : [出力]現在の入力レンジ設定値の格納先

現在設定されているアナログ入力端子の入力レンジを取得します。
連続変換が開始されている場合、エラーが返ります。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_ADSetMode

SRV_STATUS TWFA_ADSetMode(int Mode)

Mode : 16 ビット AD コンバータの動作モードを指定。以下のいずれか。

TWFA_AN_OSR_NON : オーバーサンプリング機能を使用しません
TWFA_AN_OSR2 : オーバーサンプリングレートを 2 に設定
TWFA_AN_OSR4 : オーバーサンプリングレートを 4 に設定
TWFA_AN_OSR8 : オーバーサンプリングレートを 8 に設定
TWFA_AN_OSR16 : オーバーサンプリングレートを 16 に設定
TWFA_AN_OSR32 : オーバーサンプリングレートを 32 に設定
TWFA_AN_OSR64 : オーバーサンプリングレートを 64 に設定

16 ビット AD コンバータの動作モードを設定します。
連続変換が開始されている場合、エラーが返ります。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_ADGetMode

SRV_STATUS TWFA_ADGetMode(int *pMode)

pMode : [出力]16 ビット AD コンバータの現在の動作モード設定値の格納先

現在設定されている 16 ビット AD コンバータの動作モードを取得します。
連続変換が開始されている場合、エラーが返ります。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_AD16Read

SRV_STATUS TWFA_AD16Read(int Ch, short *pData)

Ch　　: 入力するチャンネル(0,1,2,3,4,5,6,7 or TWFA_AD_ALL)
pData : [出力]変換結果の格納先

指定チャンネル/全チャンネルの AD 変換した結果を読み出します。
連続変換が開始されている場合、エラーが返ります。
※「A0x0x.lib」ファイルをプロジェクトに追加してください。

TWFA_An16ToVolt

double TWFA_An16ToVolt(WORD Data, int Opt)

Data　: AD 変換で得られた値
Opt　　: 以下のいずれか

TWFA_AN_10VPP : 入力レンジ-5～+5[V]に合わせて変換
TWFA_AN_20VPP : 入力レンジ-10～+10[V]に合わせて変換

戻り値 : AD 変換結果をボルト単位に変換した値

TWFA_AD16Read()で得られた AD 変換の結果をボルト単位に変換して返します。

TWFA_An16ToVoltQ16

long TWFA_An16ToVoltQ16(WORD Data, int Opt)

Data　: AD 変換で得られた値
Opt　　: 以下のいずれか

TWFA_AN_10VPP : 入力レンジ-5～+5[V]に合わせて変換
TWFA_AN_20VPP : 入力レンジ-10～+10[V]に合わせて変換

戻り値 : AD 変換結果をボルト単位に変換した値(Q16 固定小数点数)

TWFA_AD16Read()で得られた AD 変換の結果をボルト単位に変換して Q16 固定小数点数で返します。

□ 16 ビットカウンタ操作関数

TWFA_TimerSetPwm

```
SRV_STATUS TWFA_TimerSetPwm(int Ch, double *pFrequency,
                            double *pDuty, double *pPhase)
```

```
  Ch         : 設定する16ビットタイマのチャンネル(0,1,2)
  pFrequency : [入力]希望の周波数(Hz単位)      [出力]実際に設定できた周波数
  pDuty      : [入力]希望のデューティ(0～1.0) [出力]実際に設定できたデューティ
  pPhase     : [入力]希望の初期位相(0～1.0)   [出力]実際に設定できた初期位相
```

デューティ(pDuty)は 0～1.0 の範囲で ON デューティを指定します。例えば 0.3 を指定した場合、周期の約 30%が ON のパルスが出力されます。
初期位相(pPhase)は該当のタイマチャンネルが停止中のみ変更可能です。0～1.0 の範囲で指定します。1.0 は 360°、0.5 は 180°に相当します。
波形は製品の内部クロック(25MHz)を分周して作られるため、周波数、デューティ、初期位相の各パラメータは設定できる値が離散的になります。そのため、引数に与えた希望値と設定可能な値が異なる場合があります。関数は各引数に実際に設定できた値をセットして戻ります。

TWFA_TimerSetPwmExt

```
SRV_STATUS TWFA_TimerSetPwmExt(int Ch, double dClkFreq, double *pFrequency,
                               double *pDuty, double *pPhase)
```

```
  Ch         : 設定する16ビットタイマのチャンネル(0,1,2)
  dClkFreq   : 外部クロックの周波数(Hz単位)
  pFrequency : [入力]希望の周波数(Hz単位)      [出力]実際に設定できた周波数
  pDuty      : [入力]希望のデューティ(0～1.0) [出力]実際に設定できたデューティ
  pPhase     : [入力]希望の初期位相(0～1.0)   [出力]実際に設定できた初期位相
```

基準となるクロックとして外部入力を使用する点を除いて TWFA_TimerSetPwm()関数と同様です。
内部クロックを用いた場合、出力できる周波数の下限が約 48Hz となりますので、それより低い周波数を出力する場合に使用してください。
外部クロックは CLK1 に入力します。別の機器からの出力信号を用いることもできますが、他のチャンネルの PWM 出力を入力することも可能です。

TWFA_TimerSetPwmQ16

```
SRV_STATUS TWFA_TimerSetPwmQ16(int Ch, DWORD *pFrequency,
                               DWORD *pQ16Duty, DWORD *pQ16Phase)
```

```
  Ch         : 設定する16ビットタイマのチャンネル(0,1,2)
  pFrequency : [入力]希望の周波数(Hz単位)      [出力]実際に設定できた周波数
  pQ16Duty   : [入力]希望のデューティ(0～1.0) [出力]実際に設定できたデューティ
  pQ16Phase  : [入力]希望の初期位相(0～1.0)   [出力]実際に設定できた初期位相
```

TWFA_TimerSetPwm()関数と同様ですが、引数に Q16 固定小数を使用します。周波数は整数、デューティと初期位相は Q16 固定小数点フォーマットで 0～1.0 の範囲を指定します。
周波数のフォーマットが TWFA_TimerSetPwmExtQ16()関数と異なりますので注意してください。

TWFA_TimerSetPwmExtQ16

```
SRV_STATUS TWFA_TimerSetPwmExtQ16(int Ch, DWORD dwClkFreq, DWORD *pQ16Freq,
                                  DWORD *pQ16Duty, DWORD *pQ16Phase)
```

```
  Ch        : 設定する16ビットタイマのチャンネル(0,1,2)
  dwClkFreq : 外部クロックの周波数(Hz単位)
  pQ16Freq  : [入力]希望の周波数(Hz単位)      [出力]実際に設定できた周波数
  pQ16Duty  : [入力]希望のデューティ(0～1.0) [出力]実際に設定できたデューティ
  pQ16Phase : [入力]希望の初期位相(0～1.0)    [出力]実際に設定できた初期位相
```

TWFA_TimerSetPwmExt()関数と同様ですが、引数に Q16 固定小数を使用します。周波数は 30kHz 以下の値を Q16 固定小数点数で指定します。デューティと初期位相は 0～1.0 の範囲を Q16 固定小数点数で指定します。
周波数のフォーマットが TWFA_TimerSetPwmQ16()関数と異なりますので注意してください。

TWFA_TimerStart

```
void TWFA_TimerStart(int ChBits)
```

```
  ChBits : スタートする 16 ビットタイマのチャンネル。以下の値を OR で結合
           TWFA_TIMER_BIT0 : チャンネル 0 をスタート
           TWFA_TIMER_BIT1 : チャンネル 1 をスタート
           TWFA_TIMER_BIT2 : チャンネル 2 をスタート
```

指定のタイマチャンネルの動作を開始します。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerStop

```
void TWFA_TimerStop(int ChBits)
```

```
  ChBits : ストップする 16 ビットタイマのチャンネル。以下の値を OR で結合
           TWFA_TIMER_BIT0 : チャンネル 0 をストップ
           TWFA_TIMER_BIT1 : チャンネル 1 をストップ
           TWFA_TIMER_BIT2 : チャンネル 2 をストップ
```

指定のタイマチャンネルの動作を停止します。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerReadStatus

```
int TWFA_TimerReadStatus()
```

```
  戻り値 : タイマの状態
           ビット 0 : チャンネル 0 が動作状態のとき 1 となります
           ビット 1 : チャンネル 1 が動作状態のとき 1 となります
           ビット 2 : チャンネル 2 が動作状態のとき 1 となります
```

16 ビットタイマの動作状態を返します。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerReadCnt

WORD TWFA_TimerReadCnt(int Ch)

Ch　　: チャンネル(0,1,2)

戻り値 : 指定チャンネルのカウンタ値

16 ビットタイマのカウンタ値を返します。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerSetCnt

void TWFA_TimerSetCnt(int Ch, WORD Cnt)

Ch　: チャンネル(0,1,2)
Cnt : カウント数

16 ビットタイマのカウンタレジスタに値をセットします。
※割り込みハンドラから呼び出すことができます。

□ シリアルポート操作関数

TWFA_SCISetMode

SRV_STATUS TWFA_SCISetMode(int Ch, BYTE Mode, WORD Baud)

Ch : チャンネル(0,1)
Mode : シリアルポートの動作設定。次の値からデータ長、パリティ、ストップビットに関する設定を 1 つずつ選択して OR で結合します。指定しない項目はデフォルトが選択されます。

TWFA_SCI_DATA8 : 8 ビットデータ(デフォルト)
TWFA_SCI_DATA7 : 7 ビットデータ
TWFA_SCI_NOPARITY : パリティなし(デフォルト)
TWFA_SCI_EVEN : 偶数パリティ
TWFA_SCI_ODD : 奇数パリティ
TWFA_SCI_STOP1 : 1 ストップビット(デフォルト)
TWFA_SCI_STOP2 : 2 ストップビット

Baud : ボーレート。以下の定数で指定します。
TWFA_SCI_BAUD300 : 300bps
TWFA_SCI_BAUD600 : 600bps
TWFA_SCI_BAUD1200 : 1200bps
TWFA_SCI_BAUD2400 : 2400bps
TWFA_SCI_BAUD4800 : 4800bps
TWFA_SCI_BAUD9600 : 9600bps
TWFA_SCI_BAUD14400 : 14400bps
TWFA_SCI_BAUD19200 : 19200bps
TWFA_SCI_BAUD38400 : 38400bps

シリアルポートの動作設定と速度設定を行います。チャンネル 1 に対してこの関数を呼び出すと、再起動するまでユーザーファームのデバッグ用ポートや標準出力として使用できなくなります。

TWFA_SCIReadStatus

SRV_STATUS TWFA_SCIReadStatus(int Ch, BYTE *pStatus, int *pnReceive)

Ch : チャンネル(0,1)
pStatus : [出力]ステータスの格納先。各ビットの意味は以下です
0(LSB)ビット～2ビット : 常に0となります
3ビット : パリティエラーが起こった場合に1になります
4ビット : フレーミングエラーが起こった場合に1になります
5ビット : オーバーランエラーが起こった場合に1になります
6ビット～7ビット(MSB) : 常に0となります。

pnReceive : [出力]受信データバイト数の格納先

シリアルポートのステータスと受信バッファ中のデータ数を取得します。

TWFA_SCIRead

```
SRV_STATUS TWFA_SCIRead(int Ch, void *pData, int nData, int *pnRead)
```

```
  Ch     : チャンネル(0,1)
  pData  : [出力]読み出したデータの格納先
  nData  : 受信するバイト数(0～255)
  pnRead : [出力]実際に受信したバイト数の格納先
```

デバイスのシリアルポートからデータを読み出します。指定されたデータ数を受信するまでブロックし、約5秒でタイムアウトします。

TWFA_SCIWrite

```
SRV_STATUS TWFA_SCIWrite(int Ch, void *pData, int nData)
```

```
  Ch    : チャンネル(0,1)
  pData : [入力]送信データ
  nData : 送信するバイト数(0～255)
```

デバイスのシリアルポートからデータを送信します。指定バイト数の送信が終わるまでブロックします。

TWFA_SCISetDelimiter

```
SRV_STATUS TWFA_SCISetDelimiter(int Ch, void *pDelimiter, int nDelimiter)
```

```
  Ch         : チャンネル(0,1)
  pDelimiter : [入力]デリミタコード
  nDelimiter : デリミタコードのバイト数(0～2)
```

シリアルポートにデリミタコードを設定します。デリミタの設定は TWFA_SCIRead()の動作に影響します。
TWFA_SCIRead()関数はデリミタコード(1 バイトまたは 2 バイト)が現れると、シリアルポートからの読み取りを一旦中止し、デリミタコードより後には指定バイトまで 0 をコピーしてデータを返します。

□ インタフェース関数

TWFA_Transmit

```
SRV_STATUS TWFA_Transmit(void *pData, WORD n)
```

```
pData : [入力]送信データ
n     : 送信バイト数(0～65535)
```

指定のデータを接続中のホストパソコンに送信します。送信完了までブロックし一定時間でタイムアウトします。ホストパソコンでは送信したデータを TWXA ライブラリの TWXA_Read()関数で取り出すことができます。

TWFA_Receive

```
SRV_STATUS TWFA_Receive(void *pData, WORD n)
```

```
pData : [出力]受信データ
n     : 受信バイト数(0～65535)
```

指定バイトのデータを接続中のホストパソコンから受信します。指定バイト数の受信が完了するまでブロックし一定時間でタイムアウトします。ホストパソコンからデータを送信する場合は TWXA ライブラリの TWXA_Write()関数を使用します。

□ 割り込み許可／禁止用関数

TWFA_TimerEnableIntA

```
void TWFA_TimerEnableIntA(int ChBits)
```

ChBits : コンペアマッチ A による割り込みを許可するチャンネル。以下の値を OR で結合
TWFA_TIMER_BIT0 : チャンネル 0 のコンペアマッチ A による割り込みを許可
TWFA_TIMER_BIT1 : チャンネル 1 のコンペアマッチ A による割り込みを許可
TWFA_TIMER_BIT2 : チャンネル 2 のコンペアマッチ A による割り込みを許可

16 ビットタイマのコンペアマッチ A による割り込みを許可します。ChBits で指定したチャンネルが許可され、他は禁止されます。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerEnableIntB

```
void TWFA_TimerEnableIntB(int ChBits)
```

ChBits : コンペアマッチ B による割り込みを許可するチャンネル。以下の値を OR で結合
TWFA_TIMER_BIT0 : チャンネル 0 のコンペアマッチ B による割り込みを許可
TWFA_TIMER_BIT1 : チャンネル 1 のコンペアマッチ B による割り込みを許可
TWFA_TIMER_BIT2 : チャンネル 2 のコンペアマッチ B による割り込みを許可

16 ビットタイマのコンペアマッチ B による割り込みを許可します。ChBits で指定したチャンネルが許可され、他は禁止されます。
※割り込みハンドラから呼び出すことができます。

TWFA_TimerEnableIntOvf

```
void TWFA_TimerEnableIntOvf(int ChBits)
```

ChBits : オーバーフローによる割り込みを許可するチャンネル。以下の値を OR で結合
TWFA_TIMER_BIT0 : チャンネル 0 のオーバーフローによる割り込みを許可
TWFA_TIMER_BIT1 : チャンネル 1 のオーバーフローによる割り込みを許可
TWFA_TIMER_BIT2 : チャンネル 2 のオーバーフローによる割り込みを許可

16 ビットタイマのオーバーフロー／アンダーフローによる割り込みを許可します。ChBits で指定したチャンネルが許可され、他は禁止されます。アンダーフローはチャンネル 2 が 2 相カウントモードに設定されている場合のみ発生します。
※割り込みハンドラから呼び出すことができます。

サポート情報

製品に関する情報、最新のファームウェア、ユーティリティなどは弊社ホームページにてご案内しております。また、お問い合わせ、ご質問などは下記までご連絡ください。

テクノウェーブ(株)

URL ： https://www.techw.co.jp

E-mail ： support@techw.co.jp

(1) 本書、および本製品のホームページに掲載されている応用回路、プログラム、使用方法などは、製品の代表的動作・応用例を説明するための参考資料です。これらに起因する第三者の権利（工業所有権を含む）侵害、損害に対し、弊社はいかなる責任も負いません。

(2) 本書の内容の一部または全部を無断転載することをお断りします。

(3) 本書の内容については、将来予告なしに変更することがあります。

(4) 本書の内容については、万全を期して作成いたしましたが、万一ご不審な点や誤り、記載もれなど、お気づきの点がございましたらご連絡ください。

改訂記録

年月	版	改訂内容
2016年3月	初	
2018年4月	2	・X-A0x0x用にライブラリを分離
2020年4月	3	・ADコンバータの追加機能に対応 ・誤記の修正
2021年4月	4	・対応製品を追加 ・ファイルの入手先を変更 ・誤記の修正