お客様各位

カタログ等資料中の旧社名の扱いについて

2010 年 4 月 1 日を以って NEC エレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010 年 4 月 1 日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

■本ドキュメントに記載されているURLは、以下のとおり読み替えをお願いいたします。
http://www.necel.com/
http://www2.renesas.com/
↓
開発環境トップページ　http://japan.renesas.com/tools
ダウンロードポータル　http://japan.renesas.com/tool_download

■技術問合せについては、以下のページをご覧ください。
http://japan.renesas.com/tech_inquiry

■ツールユーザ登録については、以下のページをご覧ください。
http://japan.renesas.com/myrenesas

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

ユーザーズ・マニュアル

SM+

システム・シミュレータ

ユーザ・オープン・インタフェース編

対象ツール

SM+ for V850 Ver.2.00
SM+ for 78K0 Ver.1.01
SM+ for 78K0S Ver1.01

資料番号　U17247JJ2V0UM00（第2版）
発行年月　November 2004　CP(K)

〔メ　モ〕

目次要約

Windows**および**WindowsNT**は米国**Microsoft Corporation**の米国およびその他の国における登録商標または商標です。**

- 本資料に記載されている内容は2004年11月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品の使用に関連し発生した第三者の特許権，著作権その他の知的財産権の侵害等に関し，一切その責を負いません。当社は，本資料に基づき当社または第三者の特許権，著作権その他の知的財産権を何ら許諾するものではありません。
- 本資料に記載された回路，ソフトウエアおよびこれらに関する情報は，半導体製品の動作例，応用例を説明するものです。お客様の機器の設計において，回路，ソフトウエアおよびこれらに関する情報を使用する場合には，お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し，当社は，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。当社製品の不具合により生じた生命，身体および財産に対する損害の危険を最小限度にするために，冗長設計，延焼対策設計，誤動作防止設計等安全設計を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「特定水準」に分類しております。また，各品質水準は，以下に示す用途に製品が使われることを意図しておりますので，当社製品の品質水準をご確認ください。

  標準水準：コンピュータ，OA機器，通信機器，計測機器，AV機器，家電，工作機械，パーソナル機器，産業用ロボット

  特別水準：輸送機器（自動車，電車，船舶等），交通用信号機器，防災・防犯装置，各種安全装置，生命維持を目的として設計されていない医療機器

  特定水準：航空機器，航空宇宙機器，海底中継機器，原子力制御システム，生命維持のための医療機器，生命維持のための装置またはシステム等

  当社製品のデータ・シート，データ・ブック等の資料で特に品質水準の表示がない場合は，標準水準製品であることを表します。意図されていない用途で当社製品の使用をお客様が希望する場合には，事前に当社販売窓口までお問い合わせください。

（注）

（1）本事項において使用されている「当社」とは，NECエレクトロニクス株式会社およびNECエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいう。

（2）本事項において使用されている「当社製品」とは，（1）において定義された当社の開発，製造製品をいう。

M8E　02.11

はじめに

対 象 者　このマニュアルに記載している内容は，Windows® 98，Windows Me，Windows NT™ 4.0，Windows 2000，Windows XPの32ビット・アプリケーション・プログラム形式であるため，同OSの32 ビット・アプリケーション・プログラム作成経験者を対象とした説明となっています。

目　　的　このマニュアルは，システム・シミュレータSM+が用意しているシミュレーション環境構築手段のうち，ユーザ・オープン・インタフェースを用いたターゲット・システムのシミュレーション環境構築手段について解説したものです。

構　　成　このマニュアルは，次の内容で構成されています。

- 概要
- ユーザ・モデルの作成方法
- ユーザ・モデルの組み込み方法
- 関数レファレンス
- サンプル・プログラム

読 み 方　このマニュアルの読者には，マイクロコンピュータ，C言語に関する一般知識とWindows 98，Windows Me，Windows NT4.0，Windows 2000，Windows XPの32 ビット・アプリケーション・プログラムの作成に関する基礎知識を必要とします。

ユーザ・オープン・インタフェースとしてシステムが用意している関数（提供関数），およびユーザが作成する関数（ユーザ定義関数）について知りたいとき

→第4章　関数レファレンスをお読みください。

一通りSM+のユーザ・オープン・インタフェースについて理解しようとするとき

→目次に従って読んでください。

関連資料　**このマニュアルを使用する場合は，次の資料もあわせてご覧ください。**

関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

78K**シリーズ開発ツールに関する資料（ユーザーズ・マニュアル）**

資　料　名		資料番号	
		和文	英文
CC78K0 Ver.3.70　C**コンパイラ**	**操作編**	U17201J	U17201E
	言語編	U17200J	U17200E
CC78K0S　C**コンパイラ**	**操作編**	－	－
	言語編	－	－
RA78K0 Ver.3.80　**アセンブラ・パッケージ**	**操作編**	U17199J	U17199E
	アセンブリ言語編	U17198J	U17198E
	構造化アセンブリ言語編	U17197J	U17197E
RA78K0S　**アセンブラ・パッケージ**	**操作編**	－	－
	アセンブリ言語編	－	－
	構造化アセンブリ言語編	－	－
SM+　**システム・シミュレータ**	**操作編**	U17246J	U17246E
	ユーザ・オープン・インタフェース編	**本マニュアル**	U17247E
SM78K Ver.2.52　**システム・シミュレータ**	**操作編**	U16768J	U16768E
ID78K0-NS Ver.2.52**以上**　**統合ディバッガ**	**操作編**	U16488J	U16488E
ID78K0S-NS Ver.2.52**以上**　**統合ディバッガ**	**操作編**	U16584J	U16584E
ID78K0-QB Ver.2.81　**統合ディバッガ**	**操作編**	U16996J	U16996E
ID78K0S-QB Ver.2.81　**統合ディバッガ**	**操作編**	U17287J	U17287E
PM+ Ver.6.00　**プロジェクト・マネージャ**		U17178J	U17178E

V850シリーズ開発ツールに関する資料（ユーザーズ・マニュアル）

資料名		資料番号	
		和文	英文
CA850 Ver.3.00 Cコンパイラ・パッケージ	操作編	U17293J	U17293E
	C言語編	U17291J	U17291E
	アセンブリ言語編	U17292J	U17292E
	リンク・ディレクティブ編	U17294J	U17294E
ID850 Ver.2.50 統合ディバッガ	操作編	U16217J	U16217E
ID850NW Ver.2.51 統合ディバッガ	操作編	U16454J	U16454E
ID850NWC Ver.2.51 統合ディバッガ	操作編	U16525J	U16525E
ID850QB Ver.2.80 統合ディバッガ	操作編	U16973J	U16973E
SM+ システム・シミュレータ	操作編	U17246J	U17246E
	ユーザ・オープン・インタフェース編	本マニュアル	U17247E
SM850 Ver.2.50 システム・シミュレータ	操作編	U16218J	U16218E
SM850 Ver.2.00以上 システム・シミュレータ	外部部品ユーザ・オープン・インタフェース仕様編	U14873J	U14873E
RX850 Ver.3.13以上 リアルタイムOS	基礎編	U13430J	U13430E
	インストレーション編	U13410J	U13410E
	テクニカル編	U13431J	U13431E
RX850 Pro Ver.3.15 リアルタイムOS	基礎編	U13773J	U13773E
	インストレーション編	U13774J	U13774E
	テクニカル編	U13772J	U13772E
RX-NET ネットワーク・ライブラリ（TCP/IP）		U15083J	−
RX-NET ネットワーク・ライブラリ（PPP）		U15303J	−
RX-NET ネットワーク・ライブラリ（DNS）		U15304J	−
RX-NET ネットワーク・ライブラリ（DHCP）		U15382J	−
RX-NET ネットワーク・ライブラリ（SMTP）		U15505J	−
RX-NET ネットワーク・ライブラリ（POP）		U15539J	−
RX-NET Ver.1.00 ネットワーク・ライブラリ（telnet）		U16085J	−
RD850 Ver.3.01 タスク・ディバッガ		U13737J	U13737E
RD850 Pro Ver.3.01 タスク・ディバッガ		U13916J	U13916E
AZ850 Ver.3.20 システム・パフォーマンス・アナライザ		U14410J	U14410E
PG-FP4 フラッシュ・メモリ・プログラマ		U15260J	U15260E
TW850 Ver.2.00 性能解析チューニング・ツール		U17241J	U17241E
PM+ Ver.6.00 プロジェクト・マネージャ		U17178J	U17178E

目次

図の目次

図番号　　タイトル,ページ

表の目次

表番号　　タイトル,ページ

第 1 章　概要

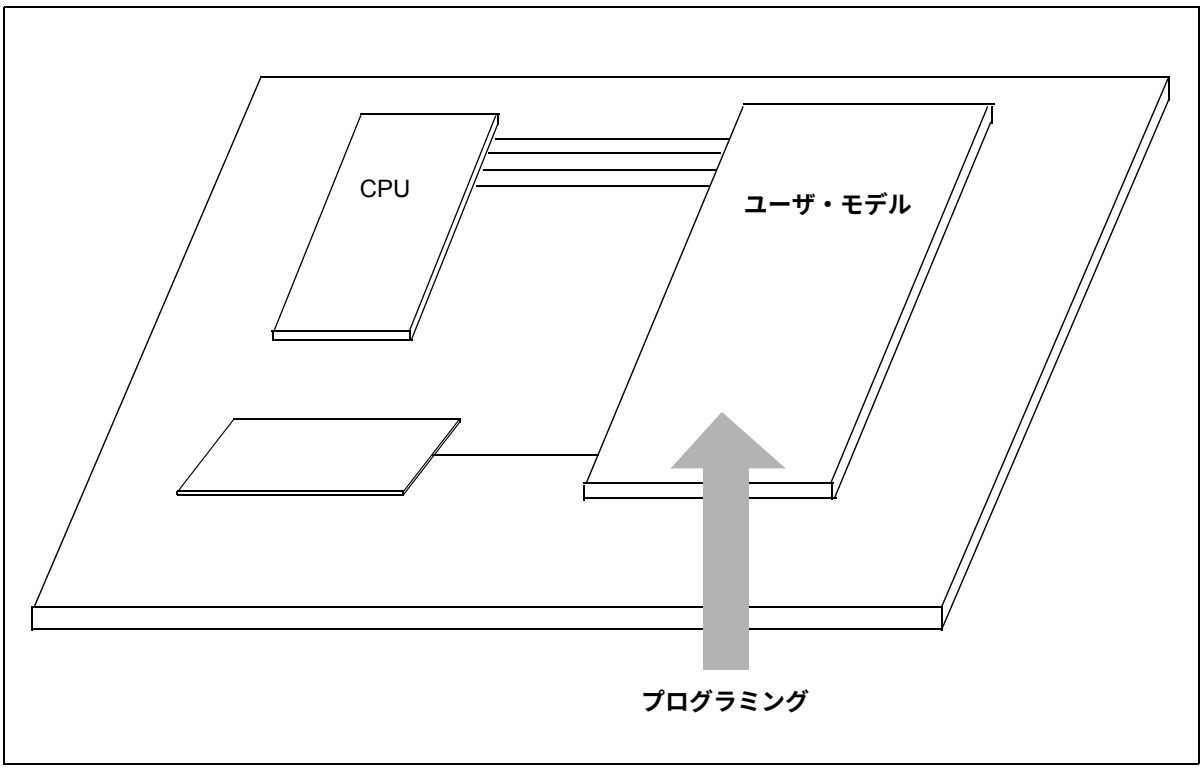
SM+ では，CPU（CPU コア + 内蔵周辺）のシミュレーションに加え，次に挙げる 2 つのターゲット・システムのシミュレーション環境構築手段を用意しています。

1 つは，" 入出力パネル・ウィンドウ[注] " です。これは，標準的な接続部品とその操作環境を提供することにより，GUI 操作による簡単なシミュレーション環境の構築を行うものです。

もう 1 つが，本マニュアルで解説するユーザ・オープン・インタフェースを用いたターゲット・システムのシミュレーション環境の構築です。" ユーザが外部のユーザ・モデルをプログラミングする " ことにより，" 入出力パネル・ウィンドウ " では実現できなかった機能までを実装可能にしています。

注　「SM+ ユーザーズ・マニュアル 操作編」を参照してください。

図 1 － 1　ユーザ・モデルのプログラミング・イメージ



1.1 インタフェース関数の種類

SM+ のユーザ・オープン・インタフェースでは，次表に示す種類のインタフェース関数を用意しています（詳細は「第 4 章 関数レファレンス」を参照）。

表 1 － 1　ユーザ・オープン・インタフェース提供関数の種類

種類	概要説明，主な機能
基本インタフェース関数	シミュレーションの基本機能 - 初期化通知 - リセット通知など
時間インタフェース関数	モデルの時系列処理を行うための周期タイマ機能 - タイマの設定 - タイマの解除 - タイマの時間通知など
端子インタフェース関数	端子の入出力機能 - 端子への信号出力 - 端子への信号入力通知など
外部バス・インタフェース関数	外部バスのスレーブ機能 - 外部バス・リード・アクセス通知 - 外部バス・ライト・アクセス通知など
シリアル・インタフェース関数	シリアルの送受信機能 - シリアル・データの送信 - シリアル・データの受信通知など
信号出力器インタフェース関数	信号データ・ファイルに従った信号出力機能 - 信号データ・ファイルに従った信号出力など

1.2 インタフェースの方式

SM+ ユーザ・オープン・インタフェースでは，次に示すインタフェース方式を取っています。

1.2.1 C 言語インタフェース

SM+ ユーザ・オープン・インタフェースは，C 言語の API 関数[注]セットで構成されています。

このため，ユーザ・モデルのプログラミングは C 言語で行い，ターゲット・システムのシミュレーション環境を構築してください。

注 Application Program Interface 関数の略

1.2.2 コールバック関数方式

コールバック関数方式とは，ユーザ・プログラム側で作成した関数へのポインタをあらかじめシステム側に通知しておき，システム側はその関数へのポインタを使用してユーザ・プログラム側で作成した関数をコールする方式です。

SM+ ユーザ・オープン・インタフェースでは，" システム側からユーザ・プログラム側を呼ぶ " 手段としてこのコールバック関数方式を採用しています。

コールバック関数は，用意されている API 関数が " ユーザ・プログラム側からシステム側を呼ぶ " 関数となっているのに対し，" システム側からユーザ・プログラム側を呼ぶ " 場合，例えば端子への信号入力の際などに使用します。

1.2.3 イベント・ドリブン方式

SM+ ユーザ・オープン・インタフェースでは，事象（イベント）に従い処理を記述するといったイベント・ドリブン方式を採用しています。

例えば，SM+ 本体側で " シミュレータ初期化／ CPU リセット／端子への信号出力／外部バス・アクセス " の事象が発生した時点でユーザ・モデル側で用意したコールバック関数がコールされます。また，ユーザ・モデルにおいて時系列処理を行うために設けている " 時間インタフェース "（= タイマ機能）でも，設定した時間に到達した時点でユーザ・モデル側で用意したコールバック関数がコールされます。

1.3 開発環境

SM+ ユーザ・オープン・インタフェースでプログラミングを行い，DLL ファイルを作成する際には次の開発ツールを使用してください。

- Microsoft Visual C++ V6.00 以上

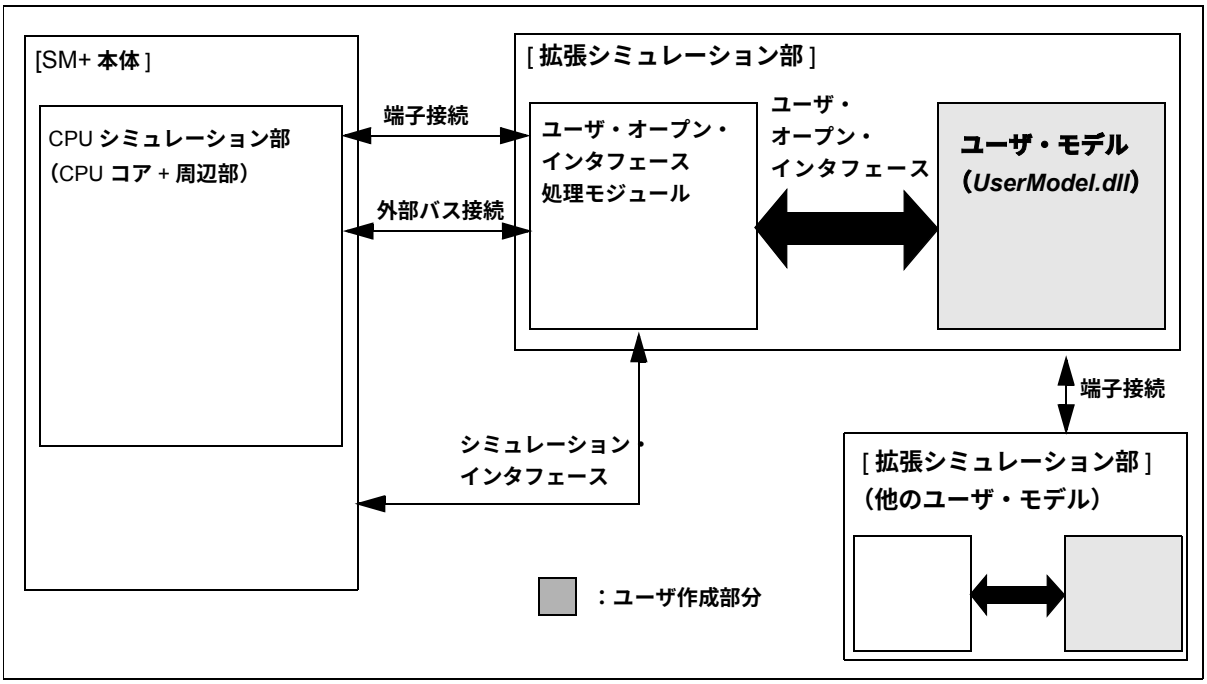
第 2 章　ユーザ・モデルの作成方法

本章では，ユーザ・モデルの作成方法について解説します。

2.1　プログラム構成

次図に SM+ ユーザ・オープン・インタフェースを使用し，システムを拡張する際のプログラム構成を示します。

図 2 － 1　プログラム構成図



システムを拡張するためにはまず，ユーザ・モデルを作成する必要があります。

ユーザ・モデルはシミュレーション・システムと連携動作するため，ユーザ・オープン・インタフェース処理モジュールとインタフェースをとります。このインタフェースが，ユーザ・オープン・インタフェースになります。

ユーザ・モデルは，ユーザ・オープン・インタフェースを介することでコンフィギュレーション（SM+ 起動時に行うシミュレータ構築処理）時に端子や外部バス・スレーブなどの資源を生成します。これにより生成された端子や外部バス・スレーブを，CPU シミュレーション部の端子や外部バス・マスタに接続することにより，CPU シミュレーション部の端子の入出力や CPU シミュレーション部からの外部バス・アクセスの処理が可能になります。

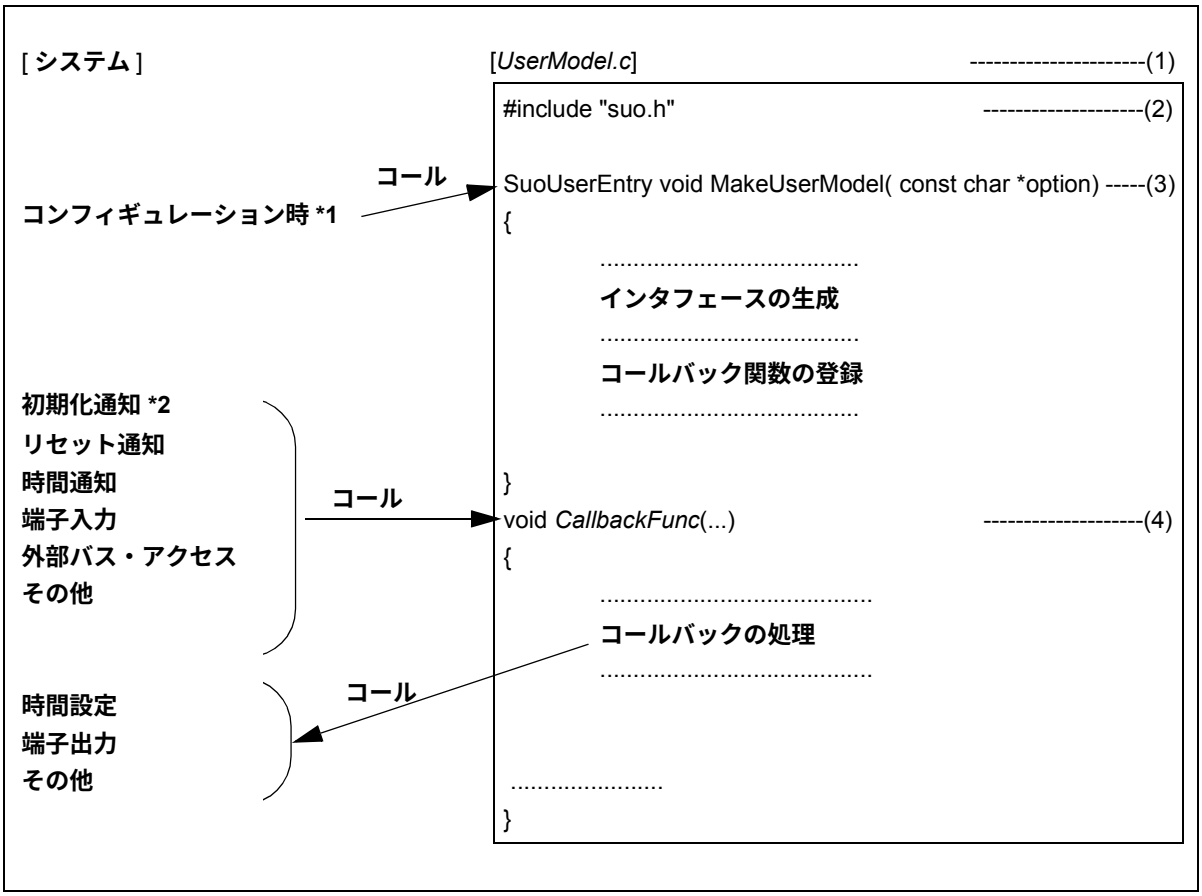
なお，生成された端子や外部バス・スレーブは，CPU シミュレーション部だけではなく拡張シミュレーション部（他のユーザ・モデル）とも接続可能です。

2.2 プログラミング概要

ユーザ・モデルは WIN32 のダイナミック・リンク・ライブラリ（DLL）形式でプログラミングします。

次図にプログラム・ファイルのテンプレートを示します。

図 2－2 プログラム・ファイルのテンプレート



*1 コンフィギュレーション時とは，SM+ 起動時に行われるシミュレータ構築処理のタイミングを指します。

*2 初期化通知は，シミュレータ構築処理が完了し SM+ が起動した直後の 1 回だけ通知されます。

2.3 プログラミング詳細

図 2 － 2 に示す (1) ～ (4) の詳細について示します。

2.3.1 ファイル名

(1) は，ファイル名です。

C 言語ファイルのためサフィックスは "*.c" です。ファイル名は自由に決めることができます。

2.3.2 インクルード・ファイル

(2) は，インクルード・ファイルです。

ユーザ・オープン・インタフェースを使用するには，システム・ヘッダ・ファイル "suo.h" をインクルードする必要があります。

2.3.3 MakeUserModel 関数

(3) は，MakeUserModel 関数で SM+ のコンフィギュレーション時にシステム側からコールされる関数です。

この関数の名前は，"MakeUserModel" でなければなりません（「4.2 ユーザ定義関数一覧」参照）。

指定形式

```
SuoUserEntry void MakeUserModel( const char *option);
```

この関数の中では次の 2 つの処理を記述します。

(1) インタフェースの生成

SM+ では SM+ 起動時のコンフィギュレーション処理において端子やバス接続を行うため，コンフィギュレーションのタイミングで接続する端子やバスなどの資源を生成しておく必要があります。

これを行うために，MakeUserModel 関数の中でインタフェースの生成関数を呼びインタフェースの生成を行います。これに伴い必要な資源の生成も行われます。

(2) コールバック関数の登録

必要に応じてコールバック関数の登録も可能です。

注意 初期化のコールバック関数を記述する場合は必ずこのタイミングで登録してください。
このタイミングで登録しないとコールバックが機能しません。
これは，初期化通知が MakeUserModel 関数コールの次のタイミングであるためです。

2.3.4 コールバック関数

(4) はコールバック関数です。

システムからコールしてもらう関数をコールバック関数と呼びます。

ここでは，初期化通知，リセット通知，時間通知，端子入力，外部バス・アクセス等の複数のコールバック関数が作成可能です（「4.1 提供関数一覧」参照）。

作成したコールバック関数は，システムからコールできるように事前に登録する必要があります。

コールバック関数の名前は自由に決めることができ，その関数の形式はコールバックの種類によって異なります。コールバック関数の中ではコールバック内容に応じた処理を記述します。

2.4 プログラム・ファイル例

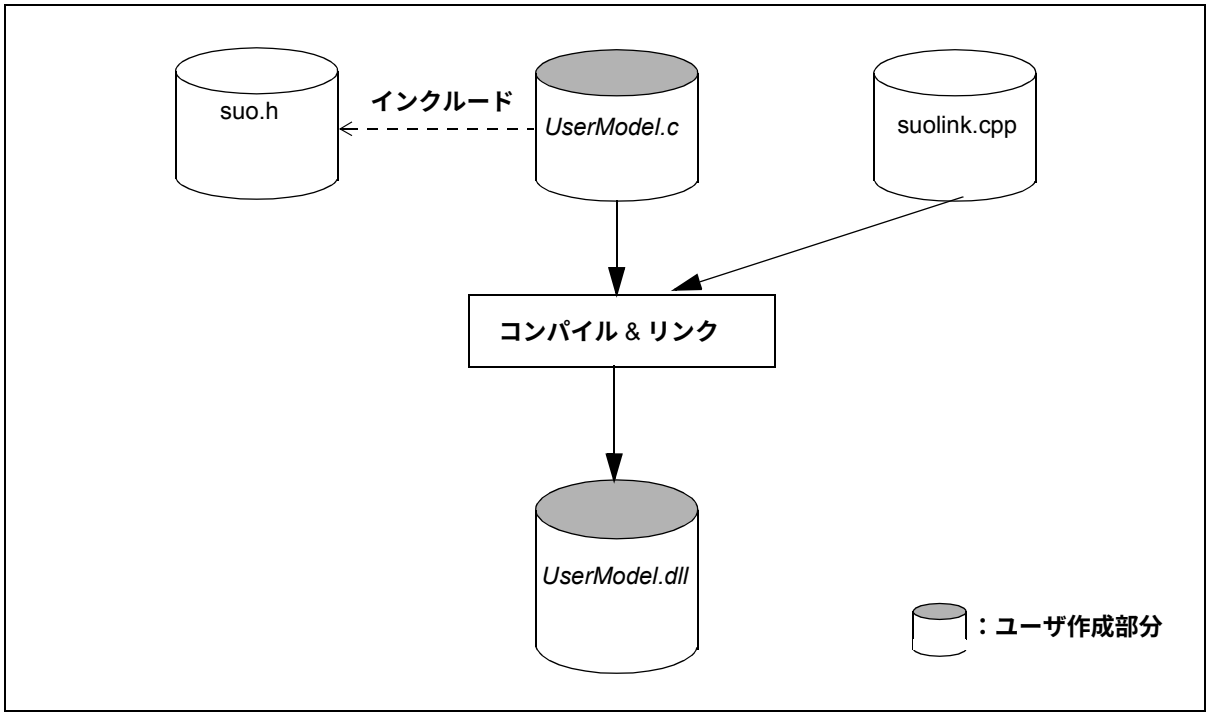
図 2 － 3　プログラム・ファイル例

```
#include "suo.h"
#include <memory.h>

void Init(void);
void InputP00(SuoHandle handle, int pinValue);
void ReadBUS1(SuoHandle handle, unsigned long addr, int accessSize, unsigned char data[]);
void WriteBUS1(SuoHandle handle, unsigned long addr, int accessSize, const unsigned char data[]);

SuoHandle p00;
SuoHandle p01;
SuoHandle bus1;
unsigned char mem[0x100];

/* MakeUserModel */
SuoUserEntry void MakeUserModel( const char *option)
{
	SuoCreatePin("P00", &p00);
	SuoCreatePin("P01", &p01);
	SuoCreateExtbus("BUS1", 0x200000, 0x100, &bus1);

	SuoSetInitCallback(Init);
	SuoSetInputDigitalPinCallback(p00, InputP00);
	SuoSetReadExtbusCallback(bus1, ReadBUS1);
	SuoSetWriteExtbusCallback(bus1, WriteBUS1);
}

/* callbacks */
void Init(void)
{
	memset(mem, 0, 0x100);
}

void InputP00(SuoHandle handle, int pinValue)
{
	SuoOutputDigitalPin(p01, pinValue);
}

void ReadBUS1(SuoHandle handle, unsigned long addr, int accessSize, unsigned char data[])
{
	memcpy(data, &mem[addr-0x200000], accessSize);
}

void WriteBUS1(SuoHandle handle, unsigned long addr, int accessSize, const unsigned char data[])
{
	memcpy(&mem[addr-0x200000], data, accessSize);
}
```

2.5 コンパイルとリンク

次図にコンパイルとリンクのフローを示します。

図 2 － 4　コンパイルとリンクのフロー



コンパイルとリンクを行うことにより，*UserModel.c* と suolink.cpp から *UserModel.dll* を作成します。

(1)　suo.h

suo.h は，ユーザ・オープン・インタフェース用のシステム・ヘッダ・ファイルです。ユーザ・プログラム（*UserModel.c*）がインクルードするだけでコンパイル対象ではありません。

(2)　suolink.cpp

suolink.cpp は，システムのユーザ・オープン・インタフェース処理モジュールとのダイナミック・リンク処理を行うファイルです。

(3)　*UserModel.c*

UserModel.c は,作成すべきユーザ･モデルのソース･ファイルです。ファイル名は自由に決めることができます。

(4)　*UserModel.dll*

UserModel.dll は，ユーザ・モデルのバイナリ・ファイル（DLL ファイル）です。ファイル名は自由に決めることができます。

注意　Microsoft Visual C++ がインストールされていない環境で DLL ファイルを動作させるには，DLL ファイルをリリース版で作成する必要があります。

第 3 章　ユーザ・モデルの組み込み方法

本章では，作成したユーザ・モデル（*UserModel.dll*）を SM+ に組み込む方法について説明します。

SM+ への組み込みには，コンフィギュレーション・ファイルを使用します。

3.1　コンフィギュレーション・ファイル

コンフィギュレーション・ファイルとは，SM+ に対してユーザ・カスタマイズ（ユーザ・モデルの追加）を行うためのファイルです。

コンフィギュレーション・ファイルは C:\NECTools32\bin\smplus.cfg [注] です。

注　　SM+ のインストール・フォルダが C:\NECTools32 の場合

3.2 コンフィギュレーション・ファイルの記述内容

コンフィギュレーション・ファイルには，ユーザ・モデルの生成処理と端子や外部バスの接続処理等を記述します。

3.2.1 生成

```
UserModel1 = Device("USEROPEN", "UserModel1.dll UserOption1");
```

ユーザ・モデルの生成は Device 関数を使って行います。

"USEROPEN " は，ユーザ・オープン・インタフェース処理モジュール（システム・モジュール）を示します。

UserModel1.dll は，作成すべきユーザ・モデルのバイナリ・ファイル（DLL 形式）です。名前は自由に決めることができます。

ファイルのパスは，コンフィギュレーション・ファイルが存在するフォルダからの相対パス，または絶対パスを指定してください。

UserOption1 は，*UserModel1.dll* に対するオプション文字列です。この文字列は MakeUserModel 関数のパラメータ "option" にそのまま渡されます。

UserModel1 は，生成したユーザ・モデルを表す変数です。名前はユーザが自由に決めることができます。.

3.2.2 端子接続

```
wire1 = Wire(1);                                    --- (1)
wire1 += cpu.Port("PinName1");                      --- (2)
wire1 += UserModel1.Port("UserPinName1");           --- (3)
```

端子接続は，次の順に行います。

(1) Wire 関数を使ってワイヤ（= 端子同士を接続する線）を生成します。

Wire 関数の引数には，必ず 1 を指定してください。

wire1 は，生成したワイヤを表す変数です。名前は自由に決めることができます。

(2) ワイヤの一方を CPU の端子に接続します。

"*PinName1*" に接続したい CPU の外部端子を指定します。"" で囲んで指定してください。

(3) ワイヤのもう一方をユーザ・モデルの端子に接続します。

UserModel1 は生成したユーザ・モデルを表す変数です。

"*UserPinName1*" には，接続したいユーザ・モデルの端子名（MakeUserModel 関数の中で生成した端子名）を指定します。"" で囲んで指定してください。

なお，ユーザ・モデルの複数の端子を同じワイヤに接続する場合はこの行を追加してください。

3.2.3 外部バス接続

```
extbus1 = BUS(n);                                                    --- (1)
extbus1 += cpu.BusMasterIF("EXTBUS");                                --- (2)
extbus1 += UserModel1.BusSlaveIF("UserExtbusName1");                 --- (3)
```

外部バス接続は次の順に行います。

(1) BUS 関数を用いバスを生成します。

バス関数の引数 *n* は，データ・バス・ビット幅です。8/16/32 のいずれかを指定できます。

extbus1 は，生成したバスを表す変数です。名前は自由に決めることができます。

(2) バスの一方を CPU の外部バス・マスタに接続します。

外部バス・マスタ "EXTBUS" を指定してください。

(3) バスのもう一方をユーザ・モデルの外部バスに接続します。

UserModel1 は，生成したユーザ・モデルを表す変数です。

"*UserExtbusName1*" には，接続したいユーザ・モデルの外部バス名（MakeUserModel 関数の中で生成した外部バス名）を指定します。"" で囲んで指定してください。

なお，ユーザ・モデルの複数の外部バスを接続する場合はこの行を追加してください。

3.2.4 その他

上記以外に，ユーザ・オープン・インタフェースを動作させるためには定型的な接続が必要です。

具体的な接続方法に関しては，製品添付のサンプル・プログラムを参照してください。

3.3 コンフィギュレーション・ファイル記述例

図 3 － 1 にコンフィギュレーション・ファイルの記述例を示します。

この例では，次表に示す接続処理を行っています。

表 3 － 1 コンフィギュレーション・ファイルの記述例の接続

接続の種類	CPU		ユーザ・モデル（SampleModel.DLL）	
端子	"P00/INTP0"	（P00 端子）	"P00"	（P00 の操作端子）
	"P30/TXD1"	（シリアルの出力端子）	"RXD"	（シリアルの入力端子）
	"P31/RXD1"	（シリアルの入力端子）	"TXD"	（シリアルの出力端子）
外部バス	"EXTBUS"	（外部バス・マスタ）	"EXTBUS1"	（外部バス・スレーブ 1）
	"EXTBUS"	（外部バス・マスタ）	"EXTBUS2"	（外部バス・スレーブ 2）

図 3 － 1 コンフィギュレーション・ファイルの記述例

```
cpu = CPU('a');
# ----------------------------------------------------------------
#   SampleModel description
# ----------------------------------------------------------------

# Generate SampleModel.dll
model = Device("USEROPEN", "SampleModel.dll -a -b");

# Connect PIN (CPU.P00-MODEL.P00)
wire_P00 = Wire(1);
wire_P00 += cpu.Port("P00/INTP0");
wire_P00 += model.Port("P00");

# Connect PIN (CPU.TXD1-MODEL.RXD)
wire_RXD = Wire(1);
wire_RXD += cpu.Port("P30/TXD1");
wire_RXD += model.Port("RXD");

# Connect PIN (CPU.RXD1-MODEL.TXD)
wire_TXD = Wire(1);
wire_TXD += cpu.Port("P31/RXD1");
wire_TXD += model.Port("TXD");

# Connect BUS (CPU.EXTBUS-MODEL.EXTBUS1)
extbus = BUS(32);
extbus += cpu.BusMasterIF("EXTBUS");
extbus += model.BusSlaveIF("EXTBUS1");
extbus += model.BusSlaveIF("EXTBUS2");
```

第 4 章　関数レファレンス

本章では，ユーザ・オープン・インタフェースとしてシステムが用意している関数（「表 4 － 1　提供関数一覧」参照），およびユーザが作成する関数（「表 4 － 3　ユーザ定義関数一覧」参照）を示します。

4.1　提供関数一覧

次表に提供関数一覧を示します。

表 4 － 1　提供関数一覧

関数名	機能概要	備考
基本インタフェース関数		
SuoSetInitCallback	初期化のコールバック登録	注 2
SuoSetResetCallback	リセットのコールバック登録	-
SuoGetMainClock	シミュレーションのメインクロック周期の取得	注 3
時間インタフェース関数		
SuoCreateTimer	タイマの生成	注 1
SuoGetTimerHandle	タイマのハンドルの取得	-
SuoSetTimer	タイマの時間設定	注 3
SuoKillTimer	タイマの時間解除	注 3
SuoSetNotifyTimerCallback	タイマの時間通知のコールバック登録	-
端子インタフェース関数		
SuoCreatePin	端子の生成	注 1
SuoGetPinHandle	端子のハンドルの取得	-
SuoOutputDigitalPin	端子のディジタル値出力	注 3
SuoOutputAnalogPin	端子のアナログ値出力	注 3
SuoOutputHighImpedance	端子のハイ・インピーダンス出力	注 3
SuoSetInputDigitalPinCallback	端子のディジタル値入力のコールバック登録	-
SuoSetInputAnalogPinCallback	端子のアナログ値入力のコールバック登録	-
SuoSetInputHighImpedanceCallback	端子のハイ・インピーダンス状態通知のコールバック登録	-
外部バス・インタフェース関数		
SuoCreateExtbus	外部バスの生成	注 1
SuoGetExtbusHandle	外部バスのハンドルの取得	-
SuoSetReadExtbusCallback	外部バスのリード・アクセスのコールバック登録	-
SuoSetWriteExtbusCallback	外部バスのライト・アクセスのコールバック登録	-

表 4 － 1　提供関数一覧

関数名	機能概要	備考
シリアル・インタフェース関数		
SuoCreateSerialUART	シリアルの生成（UART タイプ）	注 1
SuoCreateSerialCSI	シリアルの生成（CSI タイプ）	注 1
SuoGetSerialHandle	シリアルのハンドルの取得	-
SuoSetSerialParameterUART	シリアルのパラメータ設定（UART タイプ）	注 3
SuoSetSerialParameterCSI	シリアルのパラメータ設定（CSI タイプ）	注 3
SuoGetSerialParameterUART	シリアルのパラメータ取得（UART タイプ）	注 3
SuoGetSerialParameterCSI	シリアルのパラメータ取得（CSI タイプ）	注 3
SuoSendSerialData	シリアルの送信（1 データ）	注 3
SuoSendSerialDataList	シリアルの送信（複数データ）	注 3
SuoSendSerialFile	シリアルの送信（シリアル・ファイル）	注 3
SuoSetNotifySentSerialCallback	シリアルの送信完了通知のコールバック登録	-
SuoSetReceiveSerialCallback	シリアルの受信のコールバック登録	-
信号出力器インタフェース関数		
SuoCreateWave	信号出力器の生成	注 1
SuoGetWaveHandle	信号出力器のハンドルの取得	-
SuoSendWaveFile	信号出力器による送信（信号データ・ファイル）	注 3
SuoSetNotifySentWaveCallback	信号出力器の送信完了通知のコールバック登録	-

注 1　MakeUserModel 関数の中でのみ呼び出し可能です。コールバック関数の中では呼び出し不可です。

注 2　MakeUserModel 関数のタイミングで呼ばない場合，コールバック関数は機能しません。

注 3　MakeUserModel 関数の中では呼び出し不可です。コールバック関数の中でのみ呼び出し可能です。

4.1.1　提供関数詳細

次に提供関数レファレンスを示します。

SuoSetInitCallback

初期化のコールバック登録

```
void  SuoSetInitCallback(SuoInitCallback func);
```

パラメータ

func	初期化処理を行うユーザ定義関数へのポインタを指定します（「InitFunc」参照）。

戻り値

なし

解説

初期化処理を行うユーザ定義関数を登録します。

ここで登録された関数は，SM+ 起動時の一回だけ呼び出されます。

func に NULL を指定した場合は登録を解除します。

例

```
void InitFunc(void);

/* MakeUserModel */
SuoUserEntry void MakeUserModel(const char *option)
{
        .........
        SuoSetInitCallback(InitFunc);          /* Set initialize function */
}

/* Initialize function */
void InitFunc(void){
        .........
}
```

SuoSetResetCallback

リセットのコールバック登録

```
void  SuoSetResetCallback(SuoResetCallback func);
```

パラメータ

func	**リセット処理を行うユーザ定義関数へのポインタを指定します（「ResetFunc」参照）。**

戻り値

なし

解説

リセット処理を行うユーザ定義関数を登録します。

ここで登録された関数は，CPU **リセット時に呼び出されます。**

func **に** NULL **を指定した場合は登録を解除します。**

例

```
void ResetFunc(void);

func1()
{
        .........
        SuoSetResetCallback(ResetFunc);          /* Set reset function */
}

/* Reset function */
void ResetFunc(void){
        .........
}
```

SuoGetMainClock

シミュレーションのメインクロック周期の取得

```
int  SuoGetMainClock(unsigned long* time);
```

パラメータ

time	メインクロックの周期（単位 : ピコ秒）の格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

現在動作中のシミュレーション環境のメインクロックの周期を取得します。

例

```
unsigned long time;

func1()
{
        .........
        SuoGetMainClock(&time);          /* Get main clock */
}
```

SuoCreateTimer

タイマの生成

```
int  SuoCreateTimer(const char* timerName, SuoHandle* handle);
```

パラメータ

timerName	タイマ・インタフェースの名前を指定します。
handle	タイマ・インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

タイマ・インタフェースを生成します。

生成したタイマ・インタフェースは，*timerName* で指定した名前と関連付けられます。

この関数が成功すると，生成したタイマ・インタフェースのハンドルを取得できます。

後のタイマ・インタフェースに対する制御は，ここで取得したハンドルを指定して行います。

ハンドルは SuoGetTimerHandle 関数を使っても取得できます。

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hTim1;

SuoUserEntry void MakeUserModel(const char *option)
{
        .....
        SuoCreateTimer("TIM1", &hTim1);          /* Create "TIM1" */
}
```

SuoGetTimerHandle

タイマのハンドルの取得

```
SuoHandle  SuoGetTimerHandle(const char* timerName);
```

パラメータ

timerName	タイマ・インタフェースの名前を指定します。

戻り値

この関数が成功すると，指定したタイマ・インタフェースのハンドルが返ります。

この関数が失敗すると，NULL が返ります。

解説

指定したタイマ・インタフェースのハンドルを取得します。

この関数が成功すると，指定したタイマ・インタフェースのハンドルが返ります。

timerName には，SuoCreateTimer 関数で指定した名前を指定してください。

異なる名前を指定した場合は NULL が返ります。

例

```
SuoHandle hTim1;

func1()
{
        .........
        hTim1 = SuoGetTimerHandle("TIM1");          /* Get handle of "TIM1" */
}
```

SuoSetTimer

タイマの時間設定

```
int  SuoSetTimer(SuoHandle handle, int timeUnit, unsigned long timeValue);
```

パラメータ

handle

タイマ・インタフェースのハンドルを指定します。

timeUnit

時間単位を指定します（次のいずれかを指定）。

値	意味
SUO_MAINCLK	メイン・クロックの周期の単位
SUO_USEC	マイクロ秒単位

timeValue

タイマ周期時間を指定します。
単位 は *timeUnit* と同じです。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したタイマ・インタフェースに対して周期タイマの設定を行います。

周期時間は，*timeUnit* の単位で *timeValue* の値で指定します。*timeValue* に 0 は指定できません。

この関数の呼び出し直後からタイマは動作を開始します。

SuoSetNotifyTimerCallback 関数でタイマ通知関数が登録されていれば，1 周期毎にタイマ通知関数が呼ばれます。

SuoKillTimer 関数でタイマ動作を停止するまでタイマは動作し続けます。

また，現在動作しているタイマに対してこの関数を呼んだ場合は，タイマはリセットされ新たに指定した周期時間で動作を開始します。

例

```
SuoHandle hTim1;

func1()
{
        .........
        SuoSetTimer(hTim1, SUO_USEC, 20);        /* Invoke 20us cyclic timer */
}
```

SuoKillTimer

タイマの時間解除

```
int  SuoKillTimer(SuoHandle handle);
```

パラメータ

handle	タイマ・インタフェースのハンドルを指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したタイマ・インタフェースに対して周期タイマの停止を行います。

タイマが動作している時はタイマ動作を停止し，タイマが停止している時は何もしません（この場合，エラーにはなりません）。

例

```
SuoHandle hTim1;

func1()
{
        .........
        SuoKilllTimer(hTim1);           /* Stop timer */
}
```

SuoSetNotifyTimerCallback

タイマの時間通知のコールバック登録

```
int  SuoSetNotifyTimerCallback(SuoHandle handle, SuoNotifyTimerCallback func);
```

パラメータ

handle	タイマ・インタフェースのハンドルを指定します。
func	タイマの時間通知を行うユーザ定義関数へのポインタを指定します（「NotifyTimerFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

タイマの時間通知時の処理を行うユーザ定義関数を登録します。

ここで登録された関数は，指定したタイマ・インタフェースのタイマ周期毎に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void NotifyTimerFunc(SuoHandle handle);
SuoHandle hTim1;

func1()
{
        .........
        SuoSetNotifyTimerCallback(hTim1, NotifyTimerFunc);          /* Set notify-timer function */
}

/* Notify-timer function */
void NotifyTimerFunc(SuoHandle handle)
{
        .........
}
```

SuoCreatePin

端子の生成

```
int  SuoCreatePin(const char* pinName, SuoHandle* handle);
```

パラメータ

pinName	端子の名前を指定します。
handle	端子インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

端子インタフェースを生成します。

生成した端子インタフェースは，*pinName* で指定した名前と関連付けられます。

また，*pinName* で指定した端子が生成されます。

この関数が成功すると，生成した端子インタフェースのハンドルを取得できます。

後の端子インタフェースに対する制御は，ここで取得したハンドルを指定して行います。ハンドルは，SuoGetPinHandle 関数を使っても取得できます。

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hPinP00;
SuoHandle hPinABC;

SuoUserEntry void MakeUserModel(const char *option)
{
        .........
        SuoCreatePin("P00", &hPinP00);          /* Create "P00" */
        SuoCreatePin("ABC", &hPinABC);          /* Create "ABC" */
}
```

SuoGetPinHandle

端子のハンドルの取得

```
SuoHandle  SuoGetPinHandle(const char* pinName);
```

パラメータ

pinName	端子の名前を指定します。

戻り値

この関数が成功すると，指定した端子インタフェースのハンドルが返ります。

この関数が失敗すると，NULL が返ります。

解説

指定した端子インタフェースのハンドルを取得します。

この関数が成功すると，指定した端子インタフェースのハンドルが返ります。

pinName には SuoCreatePin 関数で指定した名前を指定してください。

異なる名前を指定した場合は NULL が返ります。

例

```
SuoHandle hPinP00;

func1()
{
        .........
        hPinP00 = SuoGetPinHandle("P00");          /* Get handle of "P00" */
}
```

SuoOutputDigitalPin

端子のディジタル値出力

```
int  SuoOutputDigitalPin(SuoHandle handle, int pinValue);
```

パラメータ

handle　端子インタフェースのハンドルを指定します。

pinValue　端子への出力値を指定します（次のいずれかを指定）。

値	意味
SUO_HIGH (=1)	HIGH 値
SUO_LOW (=0)	LOW 値

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定した端子インタフェースに対して信号（ディジタル値）を出力します。

信号（アナログ値）出力は SuoOutputAnalogPin 関数を使用してください。

例

```
SuoHandle hPinP00;

func1()
{
        .........
        SuoOutputDigitalPin(hPinP00, SUO_HIGH);          /* Output HIGH */
}
```

SuoOutputAnalogPin

端子のアナログ値出力

```
int  SuoOutputAnalogPin(SuoHandle handle, double pinValue);
```

パラメータ

handle	端子インタフェースのハンドルを指定します。
pinValue	端子への出力値（アナログ値）を指定します（単位 : V（ボルト））。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定した端子インタフェースに対して信号（アナログ値）を出力します。

アナログ値は単位 V（ボルト）で，浮動小数点データで指定してください。

信号（ディジタル値）出力は SuoOutputDigitalPin 関数を使用してください。

例

```
SuoHandle hPinP00;

func1()
{
        .....
        SuoOutputAnalogPin(hPinP00, 3.5);          /* Output 3.5V */
}
```

SuoOutputHighImpedance

端子のハイ・インピーダンス出力

```
int  SuoOutputHighImpedance(SuoHandle handle);
```

パラメータ

handle	端子インタフェースのハンドルを指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したディジタル／アナログ端子インタフェースに対してハイ・インピーダンスを出力します。

例

```
SuoHandle hPinP00;

func1()
{
        .........
        SuoOutputHighImpedance(hPinP00);        /* Output High Impedance */
}
```

SuoSetInputDigitalPinCallback

端子のディジタル値入力のコールバック登録

```
int  SuoSetInputDigitalPinCallback(SuoHandle handle, SuoInputDigitalPinCallback func);
```

パラメータ

handle	端子インタフェースのハンドルを指定します。
func	端子のディジタル入力処理を行うユーザ定義関数へのポインタを指定します（「InputDigitalPinFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

端子のディジタル入力処理を行うユーザ定義関数を登録します。

ここで登録された関数は，指定した端子に信号が入力された時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void InputDigitalPinFunc(SuoHandle handle, int pinValue);
SuoHandle hPinP00;

func1()
{
        .........
        SuoSetInputDigitalPinCallback(hPinP00, InputDigitalPinFunc);  /* Set input-digital-pin function */
}

/* Input-digital-pin function */
void InputDigitalPinFunc(SuoHandle handle, int pinValue)
{
        .........
}
```

SuoSetInputAnalogPinCallback

端子のアナログ値入力のコールバック登録

```
int  SuoSetInputAnalogPinCallback(SuoHandle handle, SuoInputAnalogPinCallback func);
```

パラメータ

handle	端子インタフェースのハンドルを指定します。
func	端子のアナログ入力処理を行うユーザ定義関数へのポインタを指定します（「InputAnalogPinFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

端子のアナログ入力処理を行うユーザ定義関数を登録します。

ここで登録された関数は，指定した端子に信号が入力された時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void InputAnalogPinFunc(SuoHandle handle, double pinValue);
SuoHandle hPinP00;

func1()
{
        .........
        SuoSetInputAnalogPinCallback(hPinP00, InputAnalogPinFunc);      /* Set input-analog-pin function */
}

/* Input-analog-pin function */
void InputAnalogPinFunc(SuoHandle handle, double pinValue)
{
        .........
}
```

SuoSetInputHighImpedanceCallback

端子のハイ・インピーダンス状態通知のコールバック登録

```
int  SuoSetInputHighImpedanceCallback(SuoHandle handle, SuoInputHighImpedanceCallback func);
```

パラメータ

handle	端子インタフェースのハンドルを指定します。
func	接続された全ての端子がハイ・インピーダンス状態になった際の処理を行うユーザ定義関数へのポインタを指定します（「InputHighImpedanceFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

ディジタル／アナログ端子に接続された全ての端子がハイ・インピーダンス状態になった際の処理を行なうユーザ定義関数を登録します。

func に NULL を指定した場合は登録を解除します。

例

```
void InputHighImpedanceFunc(SuoHandle handle);
SuoHandle hPinP00;

func1()
{
        .........
        SuoSetInputHighImpedanceCallback(hPinP00,InputHighImpedanceFunc);
                                                        /* Set input-high-impedance function */
}

/* Input-high-impedance function */
void InputHighImpedanceFunc(SuoHandle handle)
{
        .........
}
```

SuoCreateExtbus

外部バスの生成

```
int  SuoCreateExtbus(const char* extbusName, unsigned long addr, unsigned long size, SuoHandle* handle);
```

パラメータ

extbusName	外部バスの名前を指定します。
addr	外部メモリ領域の先頭アドレスを指定します。
size	外部メモリ領域のサイズを指定します。
handle	外部バス・インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

外部バス・インタフェースを生成します。

生成した外部バス・インタフェースは，*extbusName* で指定した名前と関連付けられます。

この関数が成功すると，生成した外部バス・インタフェースのハンドルを取得できます。

後の外部バス・インタフェースに対する制御はここで取得したハンドルを指定して行います。ハンドルは SuoGetExtbusHandle 関数を使っても取得できます。

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hExtbus1;

SuoUserEntry void MakeUserModel(const char *option)
{
        .....
        SuoCreateExtbus("EXTBUS1", 0x200000, 0x1000, &hExtbus1);        /* Create "EXTBUS1" */
}
```

SuoGetExtbusHandle

外部バスのハンドルの取得

```
SuoHandle  SuoGetExtbusHandle(const char* extbusName);
```

パラメータ

extbusName	外部バスの名前を指定します。

戻り値

この関数が成功すると，指定した外部バス・インタフェースのハンドルが返ります。

この関数が失敗すると，NULL が返ります。

解説

指定した外部バス・インタフェースのハンドルを取得します。

この関数が成功すると，指定した外部バス・インタフェースのハンドルが返ります。

extbusName には SuoCreateExtbus 関数で指定した名前を指定してください。

異なる名前を指定した場合は NULL が返ります。

例

```
SuoHandle hExtbus1;

func1()
{
        .........
        hExtbus1 = SuoGetExtbusHandle("EXTBUS1");        /* Get handle of "EXTBUS1" */
}
```

SuoSetReadExtbusCallback

外部バスのリード・アクセスのコールバック登録

```
int  SuoSetReadExtbusCallback(SuoHandle handle, SuoReadExtbusCallback func);
```

パラメータ

handle	外部バス・インタフェースのハンドルを指定します。
func	外部バスのリード・アクセス処理を行うユーザ定義関数へのポインタを指定します（「 ReadExtbusFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

外部バスのリード・アクセス処理を行うユーザ定義関数を登録します。

ここで登録された関数は，指定した外部バス（登録したアドレス範囲）に対してリード要求が発生した時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void ReadExtbusFunc(SuoHandle handle, unsigned long addr, int accessSize, unsigned char data[]);
SuoHandle hExtbus1;

func1()
{
        .........
        SuoSetReadExtbusCallback(hExtbus1, ReadExtbusFunc);       /* Set read-external-bus function */
}

/* Read-external-bus function */
void ReadExtbusFunc(SuoHandle handle, unsigned long addr, int accessSize, unsigned char data[])
{
        .........
}
```

SuoSetWriteExtbusCallback

外部バスのライト・アクセスのコールバック登録

```
int  SuoSetWriteExtbusCallback(SuoHandle handle, SuoWriteExtbusCallback func);
```

パラメータ

handle	外部バス・インタフェースのハンドルを指定します。
func	外部バスのライト・アクセスの処理を行うユーザ定義関数へのポインタを指定します（「WriteExtbusFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

外部バスのライト・アクセスの処理を行うユーザ定義関数を登録します。

ここで登録された関数は，指定した外部バス（登録したアドレス範囲）に対してライト要求が発生した時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void WriteExtbusFunc(SuoHandle handle, unsigned long addr, int accessSize, const unsigned char data[]);
SuoHandle hExtbus1;

func1()
{
        .........
        SuoSetWriteExtbusCallback(hExtbus1, WriteExtbusFunc);       /* Set write-external-bus function */
}

/* Write-external-bus function */
void WriteExtbusFunc(SuoHandle handle, unsigned long addr, int accessSize, const unsigned char data[])
{
        .........
}
```

SuoCreateSerialUART

シリアルの生成（UART タイプ）

```
int  SuoCreateSerialUART(const char* serialName, const char* pinNameTXD, const char* pinNameRXD,
SuoHandle* handle);
```

パラメータ

serialName	シリアルの名前を指定します。
pinNameTXD	シリアルで使用する送信データ用端子の名前を指定します。
pinNameRXD	シリアルで使用する受信データ用端子の名前を指定します。
handle	シリアル・インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

シリアル・インタフェース（UART タイプ）を生成します。

生成したシリアル・インタフェースは *serialName* で指定した名前と関連付けられます。

また，*pinNameTXD* と *pinNameRXD* で指定した端子が生成されます。

この関数が成功すると，生成したシリアル・インタフェースのハンドルを取得できます。

後のシリアル・インタフェースに対する制御は，ここで取得したハンドルを指定して行います。ハンドルは SuoGetSerialHandle 関数を使っても取得できます。

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hUart1;

SuoUserEntry void MakeUserModel(const char *option)
{
        .....
        SuoCreateSerialUART("UART1", "TXD1", "RXD1", &hUart1);          /* Create "UART1" */
}
```

SuoCreateSerialCSI

シリアルの生成（CSI タイプ）

```
int  SuoCreateSerialCSI(const char* serialName, const char* pinNameSO, const char* pinNameSI, const char*
pinNameSCK, SuoHandle* handle);
```

パラメータ

serialName	シリアルの名前を指定します。
pinNameSO	シリアルで使用する送信データ用端子の名前を指定します。
pinNameSI	シリアルで使用する受信データ用端子の名前を指定します。
pinNameSCK	シリアルで使用するクロック用端子の名前を指定します。
handle	シリアル・インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

シリアル・インタフェース（CSI タイプ）を生成します。

生成したシリアル・インタフェースは *serialName* で指定した名前と関連付けられます。

また，*pinNameSO* と *pinNameSI* と *pinNameSCK* で指定した端子が生成されます。

この関数が成功すると，生成したシリアル・インタフェースのハンドルを取得できます。

後のシリアル・インタフェースに対する制御はここで取得したハンドルを指定して行います。ハンドルは SuoGetSerialHandle 関数を使っても取得できます。

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hCsi1;

SuoUserEntry void MakeUserModel(const char *option)
{
        .....
        SuoCreateSerialCSI("CSI1", "SO1", "SI1", "SCK1", &hCsi1);     /* Create "CSI1" */
}
```

SuoGetSerialHandle

シリアルのハンドルの取得

```
SuoHandle  SuoGetSerialHandle(const char* serialName);
```

パラメータ

serialName	シリアルの名前を指定します。

戻り値

この関数が成功すると，指定したシリアル・インタフェースのハンドルが返ります。

この関数が失敗すると，NULL が返ります。

解説

指定したシリアル・インタフェースのハンドルを取得します。

この関数が成功すると，指定したシリアル・インタフェースのハンドルが返ります。

serialName には，SuoCreateSerialUART，または SuoCreateSerialCSI 関数で指定した名前を指定してください。

異なる名前を指定した場合は NULL が返ります。

例

```
SuoHandle hSerial1;

func1()
{
        .........
        hSerial1 = SuoGetSerialHandle("SERIAL1");          /* Get handle of "SERIAL1" */
}
```

SuoSetSerialParameterUART

シリアルのパラメータ設定（UART タイプ）

```
int  SuoSetSerialParameterUART(SuoHandle handle, const SuoSerialParameterUART* param);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
param	シリアルのパラメータ（UART タイプ）の格納場所を指定します。 SuoSerialParameterUART 構造体へのポインタで指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したシリアル・インタフェースに対してシリアル動作に関するパラメータ（UART タイプ）を設定します。

ディフォールトの設定値は次の通りです。

ボー・レート	:	9600bps
転送方向	:	LSB ファースト
データ・ビット長	:	7 ビット
ストップ・ビット長	:	1 ビット
パリティ	:	なし

例

```
SuoHandle hUart1;

func1()
{
        SuoSerialParameterUART param;
        .....
        param.baudrate = 19200;                         /* 19200bps */
        param.direction = SUO_LSBFIRST;                 /* LSB First */
        param.dataLength = 8;                           /* databit 8bit */
        param.stopLength = 1;                           /* stopbit 1bit */
        param.parity = SUO_EVENPARITY;                  /* even parity */
        SuoSetSerialParameterUART(hUart1, &param);      /* Set parameter of UART1 */
}
```

構造体

SuoSerialParameterUART **構造体**

typedef struct {

unsigned long baudrate; **ボー・レート値（単位** : bps**）**

int direction; **転送方向（次のいずれかを指定）**

値	意味
SUO_MSBFIRST	MSB **ファースト**
SUO_LSBFIRST	LSB **ファースト**

int dataLength; **データ・ビット長（**1 - 32 **を指定）**

int stopLength; **ストップ・ビット長（**1 - 2 **を指定）**

int parity; **パリティ（次のいずれか）**

値	意味
SUO_NONEPARITY	**パリティなし**
SUO_ZEROPARITY	0 **パリティ** **（送信時はパリティ** 0**，受信時はパリティ・チェックなし）**
SUO_ODDPARITY	**奇数パリティ**
SUO_EVENPARITY	**偶数パリティ**

} SuoSerialParameterUART;

SuoSetSerialParameterCSI

シリアルのパラメータ設定（CSI タイプ）

```
int  SuoSetSerialParameterCSI(SuoHandle handle, const SuoSerialParameterCSI* param);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
param	シリアルのパラメータ（CSI タイプ）の格納場所を指定します。 SuoSerialParameterCSI 構造体へのポインタで指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したシリアル・インタフェースに対してシリアル動作に関するパラメータ（CSI タイプ）を設定します。

ディフォールトの設定値は次の通りです。

モード	：	スレーブ
転送クロック	：	0
位相	：	通常の位相
転送方向	：	MSB ファースト
データ・ビット長	：	8 ビット

例

```
SuoHandle hCsi1;

func1()
{
        SuoSerialParameterCSI param;
        .........
        param.mode = SUO_SLAVE;                 /* slave */
        param.frequency = 1000000;              /* 1MHz */
        param.phase = 0;                        /* normal */
        param.direction = SUO_LSBFIRST;         /* LSB First */
        param.dataLength = 8;                   /* databit 8bit */
        SuoSetSerialParameterCSI(hCsi1, &param);    /* Set parameter of CSI1 */
}
```

構造体

SuoSerialParameterCSI **構造体**

typedef struct {

int mode;

動作モード（次のいずれかを指定）

値	意味
SUO_MASTER	**マスタ動作**
SUO_SLAVE	**スレーブ動作**

unsigned long frequency;

転送クロックの周波数（単位 : Hz**）**
マスタ動作の場合は 0 **を指定できません。**

int phase;

位相（次のいずれかを指定）
詳細は表 4 － 2 **を参照してください。**

値	意味
0	**通常の位相**
SUO_PRECEDEDATA	**データ先行出力**
SUO_REVERSECLOCK	**クロック反転**
SUO_PRECEDEDATA | SUO_REVERSECLOCK	**データ先行出力，クロック反転の両方を指定**

int direction;

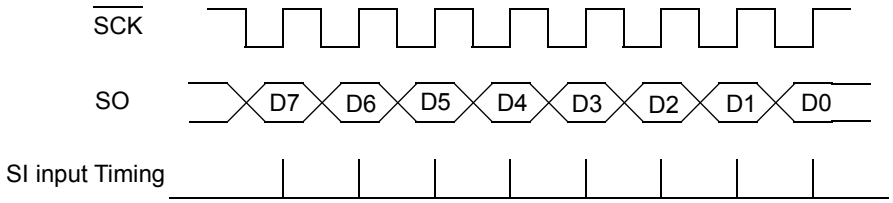
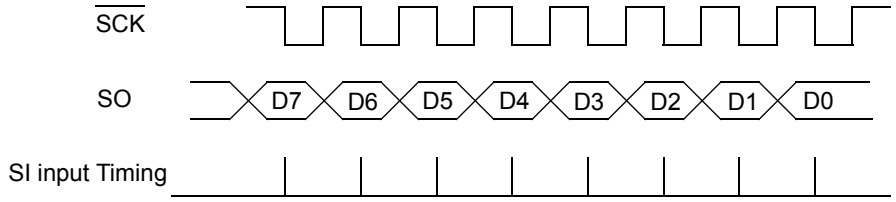
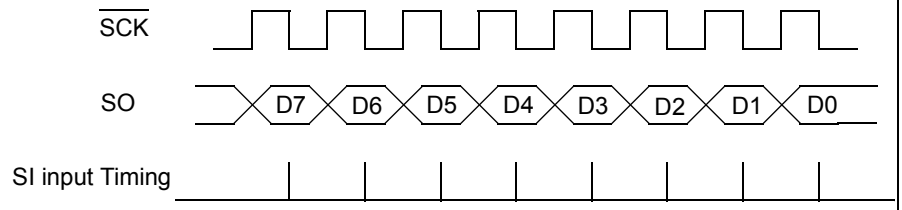
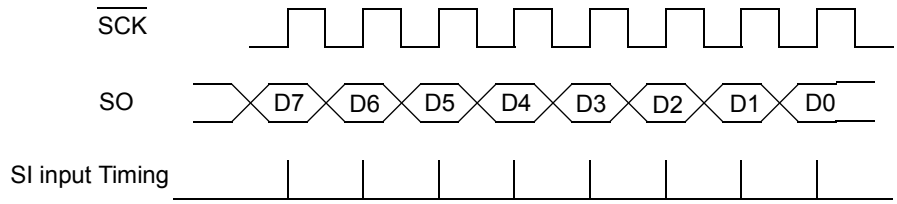
転送方向（次のいずれかを指定）

値	意味
SUO_MSBFIRST	MSB **ファースト**
SUO_LSBFIRST	LSB **ファースト**

int datalength; **データ・ビット長**（1 - 32）

} SuoSerialParameterCSI;

表 4 － 2　CSI 位相タイプ（SuoSetSerialParameterCSI **関数**）

phase **の値**	**位相**
0	
SUO_PRECEDEDATA	
SUO_REVERSECLOCK	
SUO_PRECEDEDATA| SUO_REVERSECLOCK	

SuoGetSerialParameterUART

シリアルのパラメータ取得（UART タイプ）

```
int  SuoGetSerialParameterUART(SuoHandle handle, SuoSerialParameterUART* param);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
param	シリアルのパラメータ（UART タイプ）の格納場所を指定します。 SuoSerialParameterUART 構造体へのポインタで指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したシリアル・インタフェースに対してシリアル動作に関するパラメータ（UART タイプ）を取得します。

例

```
SuoHandle hUart1;

func1()
{
        SuoSerialParameterUART param;
        .........
        SuoGetSerialParameterUART(hUart1, &param);        /* Get parameter of UART1 */
        .........
}
```

SuoGetSerialParameterCSI

シリアルのパラメータ取得（CSI タイプ）

```
int  SuoGetSerialParameterCSI(SuoHandle handle, SuoSerialParameterCSI* param);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
param	シリアルのパラメータ（CSI タイプ）の格納場所を指定します。 SuoSerialParameterCSI 構造体へのポインタで指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

指定したシリアル・インタフェースに対してシリアル動作に関するパラメータ（CSI タイプ）を取得します。

例

```
SuoHandle hCsi1;

func1()
{
        SuoSerialParameterCSI param;
        .........
        SuoGetSerialParameterCSI(hCsi1, &param);        /* Get parameter of CSI1 */
        .........
}
```

SuoSendSerialData

シリアルの送信（1 データ）

```
int  SuoSendSerialData(SuoHandle handle, unsigned long data);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
data	送信データ（1 データ）を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

1 つのシリアル・データを送信開始します。

シリアル・データを送信完了するまで時間がかかります。送信完了のタイミングを通知してもらうためには，SuoSetNotifySentSerialCallback 関数で送信完了通知関数を設定してください。

現在送信中のシリアル・インタフェースに対して，この関数を呼んだ場合はエラーになります。

例

```
SuoHandle hSerial1;

func1()
{
        .........
        SuoSendSerialData(hSerial1, 0x80);          /* Send 0x80 */
}
```

SuoSendSerialDataList

シリアルの送信（複数データ）

```
int  SuoSendSerialDataList(SuoHandle handle, long count, unsigned long dataList[]);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
count	送信データの個数を指定します（1 - 32767）。
dataList[]	送信データ（複数データ）を指定します。配列でデータ個数分指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

複数個のシリアル・データを送信開始します。

シリアル・データを送信完了するまで時間がかかります。送信完了のタイミングを通知してもらうためには，SuoSetNotifySentSerialCallback 関数で送信完了通知関数を設定してください。

現在送信中のシリアル・インタフェースに対して，この関数を呼んだ場合はエラーになります。

例

```
SuoHandle hSerial1;

func1()
{
        unsigned long dataList[6] = {0x73, 0x65, 0x72, 0x69, 0x61, 0x6c};
        .........
        SuoSendSerialDataList(hSerial1, 6, dataList);          /* Send dataList */
}
```

SuoSendSerialFile

シリアルの送信（シリアル・ファイル）

```
int  SuoSendSerialFile(SuoHandle handle, const char* serialFile);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
serialFile	シリアル・ファイルを指定します。 シリアル・ファイルとは SM+ のシリアル・ウィンドウで編集して保存したファイルのことです。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

シリアル・ファイルに記載されたシリアル・データを送信開始します。

serialFile を相対パスで指定した場合は，ユーザ・モデル（*UserModel.dll*）のパスからの相対として扱われます。

シリアル・データを送信完了するまで時間がかかります。送信完了のタイミングを通知してもらうためには，SuoSetNotifySentSerialCallback 関数で送信完了通知関数を設定してください。

現在送信中のシリアル・インタフェースに対して，この関数を呼んだ場合はエラーになります。

例

```
SuoHandle hSerial1;

func1()
{
        .........
        SuoSendSerialFile(hSerial1, "foo.ser");        /* Send serial data on "foo.ser" */
}
```

SuoSetNotifySentSerialCallback

シリアルの送信完了通知のコールバック登録

```
int  SuoSetNotifySentSerialCallback(SuoHandle handle, SuoNotifySentSerialCallback func);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
func	シリアルの送信完了時の処理を行うユーザ定義関数へのポインタを指定します（「NotifySentSerialFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

シリアルの送信完了時の処理を行うユーザ定義関数を登録します。

ここで登録された関数は,送信指定した 1 つまたは複数のシリアル･データが全て送信完了した時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void NotifySentSerialFunc(SuoHandle handle);
SuoHandle hSerial1;

func1()
{
        .........
        SuoSetNotifySentSerialCallback(hSerial1, NotifySentSerialFunc);    /* Set notify-sent-serial function */
}

/* Notify-sent-serial function */
void NotifySentSerialFunc(SuoHandle handle)
{
        .........
}
```

SuoSetReceiveSerialCallback

シリアルの受信のコールバック登録

```
int  SuoSetReceiveSerialCallback(SuoHandle handle, SuoReceiveSerialCallback func);
```

パラメータ

handle	シリアル・インタフェースのハンドルを指定します。
func	シリアルの受信時の処理を行うユーザ定義関数へのポインタを指定します（「ReceiveSerialFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

シリアルの受信時の処理を行うユーザ定義関数を登録します。

ここで登録された関数は，シリアル・データを 1 つ受信した時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void ReceiveSerialFunc(SuoHandle handle, unsigned long data, int status);
SuoHandle hSerial1;

func1()
{
        .........
        SuoSetReceiveSerialCallback(hSerial1, ReceiveSerialFunc);    /* Set receive-serial function */
}

/* Receive-serial function */
void ReceiveSerialFunc(SuoHandle handle, unsigned long data, int status)
{
        .........
}
```

SuoCreateWave

信号出力器の生成

```
int  SuoCreateWave(const char* waveName, int count, const char* pinNameList[], SuoHandle* handle);
```

パラメータ

waveName	信号出力器の名前を指定します。
count	信号出力器で使用する端子の個数を指定します。
pinNameList	信号出力器で使用する端子の名前（複数個）を指定します。配列で端子数分指定します。
handle	信号出力器インタフェースのハンドルの格納場所を指定します。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

信号出力器インタフェースを生成します。

生成した信号出力器インタフェースは，*waveName* で指定した名前と関連付けられます。

また，*count*/*pinNameList* で指定した端子が生成されます。

この関数が成功すると，生成した信号出力器インタフェースのハンドルを取得できます。

後の信号出力器インタフェースに対する制御は，ここで取得したハンドルを指定して行います。ハンドルは SuoGetWaveHandle 関数を使っても取得できます。.

この関数は MakeUserModel 関数内でしか呼ぶことができません。他のタイミングで呼んだ場合はエラーになります。

例

```
SuoHandle hWave1;

SuoUserEntry void MakeUserModel(const char *option)
{
        .....
        char* pinNameList[4] = {"P00", "P01", "P02", "P03"};
        SuoCreateWave("WAVE1", 4, pinNameList, &hWave1);        /* Create "WAVE1" */
}
```

SuoGetWaveHandle

信号出力器のハンドルの取得

```
SuoHandle  SuoGetWaveHandle(const char* waveName);
```

パラメータ

waveName	信号出力器の名前を指定します。

戻り値

この関数が成功すると，指定した信号出力器インタフェースのハンドルが返ります。

この関数が失敗すると，NULL が返ります。

解説

指定した信号出力器インタフェースのハンドルを取得します。

この関数が成功すると，指定した信号出力器インタフェースのハンドルが返ります。

waveName には SuoCreateWave 関数で指定した名前を指定してください。

異なる名前を指定した場合は NULL が返ります。

例

```
SuoHandle hWave1;

func1()
{
        .........
        hWave1 = SuoGetWaveHandle("WAVE1");          /* Get handle of "WAVE1" */
}
```

SuoSendWaveFile

信号出力器による送信（信号データ・ファイル）

```
int  SuoSendWaveFile(SuoHandle handle, const char* waveFile);
```

パラメータ

handle	信号出力器インタフェースのハンドルを指定します。
waveFile	信号データ・ファイルを指定します。信号データ・ファイルとは，SM+ の信号データ・エディタ・ウィンドウで編集して保存したファイルのことです。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

信号データ・ファイルにタイミング記述された信号値を送信開始します。

waveFile を相対パスで指定した場合は，ユーザ・モデル（*UserModel.dll*）のパスからの相対として扱われます。

信号データ・ファイルを送信完了するまで時間がかかります。送信完了のタイミングを通知してもらうためには，SuoSetNotifySentWaveCallback 関数で送信完了通知関数を設定してください。

現在送信中の信号出力器インタフェースに対してこの関数を呼んだ場合は，送信中のデータはキャンセルされ新たに指定したデータが送信されます。

例

```
SuoHandle hWave1;

func1()
{
        .........
        SuoSendWaveFile(hSerial1, "foo.wvi");        /* Send pin data on "foo.wvi" */
}
```

SuoSetNotifySentWaveCallback

信号出力器の送信完了通知のコールバック登録

```
int  SuoSetNotifySentWaveCallback(SuoHandle handle, SuoNotifySentWaveCallback func);
```

パラメータ

handle	信号出力器インタフェースのハンドルを指定します。
func	信号出力器の送信完了時の処理を行うユーザ定義関数へのポインタを指定します（「NotifySentWaveFunc」参照）。

戻り値

この関数が成功すると，SUO_NOERROR が返ります。

この関数が失敗すると，エラー番号が返ります（「表 4 － 4　エラー番号一覧」参照）。

解説

信号出力器の送信完了時の処理を行うユーザ関数を登録します。

ここで登録された関数は，送信指定した信号データが全て送信完了した時に呼ばれます。

func に NULL を指定した場合は登録を解除します。

例

```
void NotifySentWaveFunc(SuoHandle handle);
SuoHandle hWave1;

func1()
{
        .........
        SuoSetNotifySentWaveCallback(hWave1, NotifySentWaveFunc);     /* Set notify-sent-wave function */
}

/* Notify-sent-wave function */
void NotifySentWaveFunc(SuoHandle handle)
{
        .........
}
```

4.2 ユーザ定義関数一覧

次表にユーザ定義関数（MakeUserMode エントリ関数とコールバック関数）一覧を示します。

表 4 － 3 ユーザ定義関数一覧

関数名	機能概要
MakeUserModel	MakeUserModel 関数
InitFunc	初期化のコールバック関数
ResetFunc	リセットのコールバック関数
NotifyTimerFunc	タイマの時間通知のコールバック関数
InputDigitalPinFunc	端子のディジタル値入力のコールバック関数
InputAnalogPinFunc	端子のアナログ値入力のコールバック関数
InputHighImpedanceFunc	端子のハイ・インピーダンス状態通知のコールバック関数
ReadExtbusFunc	外部バスのリード・アクセスのコールバック関数
WriteExtbusFunc	外部バスのライト・アクセスのコールバック関数
NotifySentSerialFunc	シリアルの送信完了通知のコールバック関数
ReceiveSerialFunc	シリアルの受信のコールバック関数
NotifySentWaveFunc	信号出力器の送信完了通知のコールバック関数

4.2.1 ユーザ定義関数詳細

次にユーザ定義関数レファレンスを示します。

MakeUserModel

MakeUserModel 関数 [ユーザ定義関数]

```
SuoUserEntry void  MakeUserModel(const char *option);
```

注意 MakeUserModel はユーザ・モデルの静的エントリ関数であるため，この関数名を使用しなければなりません。

パラメータ

option	コンフィギュレーション・ファイルで指定したオプション文字列です。 コンフィギュレーション・ファイルでオプションを指定しなかった場合は NULL 文字（" "）が入ります。

戻り値

なし

解説

この関数では，ユーザ・モデルで使用する資源の生成を行う必要があります。

この関数以外の関数では資源の生成を行うことができません。

また，この関数では必要に応じて，コールバック関数の登録をする必要があります。

特に初期化のコールバック関数の登録は，この関数で登録する必要があります（この関数以外で登録しても初期化タイミングを過ぎてしまい意味を持たないため）。

例

```
SuoHandle hTim1;
SuoHandle hPinP00;
SuoHandle hExtbus1;

void InitFunc(void);
void ResetFunc(void);

SuoUserEntry void MakeUserModel(const char *option)
{
        /* Create source */
        SuoCreateTimer("TIM1", &hTim1);                                  /* Create "TIM1" */
        SuoCreatePin("P00", &hPinP00);                                   /* Create "P00" */
        SuoCreateExtbus("EXTBUS1", 0x200000, 0x1000, &hExtbus1);         /* Create "EXTBUS1" */

        /* Set callbacks */
        SuoSetInitCallback(InitFunc);                                    /* Set initialize function */
        SuoSetResetCallback(ResetFunc);                                  /* Set reset function */
}
```

InitFunc

初期化のコールバック関数 [ユーザ定義関数]

```
void  InitFunc (void);
```

注意　InitFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

なし

戻り値

なし

解説

InitFunc では初期化処理を記述します。

InitFunc をコールバック関数として登録するためには，SuoSetInitCallback 関数を使用します。

ResetFunc

リセットのコールバック関数 [ユーザ定義関数]

```
void  ResetFunc (void);
```

注意　ResetFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

なし

戻り値

なし

解説

ResetFunc ではリセット処理を記述します。

ResetFunc をコールバック関数として登録するためには，SuoSetResetCallback 関数を使用します。

NotifyTimerFunc

タイマの時間通知のコールバック関数 [ユーザ定義関数]

```
void  NotifyTimerFunc (SuoHandle handle);
```

注意 NotifyTimerFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	タイマ・インタフェースのハンドルです。

戻り値

なし

解説

NotifyTimerFunc では，タイマの時間通知時の処理を記述します。

NotifyTimerFunc をコールバック関数として登録するためには，SuoSetNotifyTimerCallback 関数を使用します。

InputDigitalPinFunc

端子のディジタル値入力のコールバック関数 [ユーザ定義関数]

```
void  InputDigitalPinFunc (SuoHandle handle, int pinValue);
```

注意　InputDigitalPinFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle　端子インタフェースのハンドルです。

pinValue　端子への入力値（ディジタル値）です（次のいずれかを指定）。

値	意味
SUO_HIGH (=1)	HIGH 値
SUO_LOW (=0)	LOW 値

戻り値

なし

解説

InputDigitalPinFunc では，端子のディジタル入力処理を記述します。

InputDigitalPinFunc をコールバック関数として登録するためには，SuoSetInputDigitalPinCallback 関数を使用します。

InputAnalogPinFunc

端子のアナログ値入力のコールバック関数 [ユーザ定義関数]

```
void  InputAnalogPinFunc (SuoHandle handle, double pinValue);
```

注意 InputAnalogPinFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	端子インタフェースのハンドルです。
pinValue	端子への入力値（アナログ値）です（単位 :V（ボルト））。

戻り値

なし

解説

InputAnalogPinFunc では端子のアナログ入力処理を記述します。

InputAnalogPinFunc をコールバック関数として登録するためには，SuoSetInputAnalogPinCallback 関数を使用します。

InputHighImpedanceFunc

端子のハイ・インピーダンス状態通知のコールバック関数 [ユーザ定義関数]

```
void  InputHighImpedanceFunc (SuoHandle handle);
```

注意 InputHighImpedanceFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	端子インタフェースのハンドルです。

戻り値

なし

解説

InputHighImpedanceFunc では，ディジタル／アナログ端子に接続された全ての端子がハイ・インピーダンス状態になった際の処理を記述します。

InputHighImpedanceFunc をコールバック関数として登録するためには，SuoSetInputHighImpedanceCallback 関数を使用します。

ReadExtbusFunc

外部バスのリード・アクセスのコールバック関数 [ユーザ定義関数]

```
void  ReadExtbusFunc (SuoHandle handle, unsigned long addr, int accessSize, unsigned char data[]);
```

注意 ReadExtbusFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	外部バス・インタフェースのハンドルです。
addr	アドレスです。
accessSize	アクセス・サイズです。
data[]	データ格納領域です。アクセス・サイズ分データを格納する必要があります。

戻り値

なし

解説

ReadExtbusFunc では外部バスのリード・アクセスの処理を記述します。

data[] へのデータ格納処理を行う必要があります。

ReadExtbusFunc をコールバック関数として登録するためには，SuoSetReadExtbusCallback 関数を使用します。

WriteExtbusFunc

外部バスのライト・アクセスのコールバック関数 [ユーザ定義関数]

```
void  WriteExtbusFunc (SuoHandle handle, unsigned long addr, int accessSize, const unsigned char data[]);
```

注意 WriteExtbusFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	外部バス・インタフェースのハンドルです。
addr	アドレスです。
accessSize	アクセス・サイズです。
data[]	データ格納領域です。アクセス・サイズ分データを格納する必要があります。

戻り値

なし

解説

WriteExtbusFunc では外部バスのライト・アクセスの処理を記述します。

WriteExtbusFuncをコールバック関数として登録するためには，SuoSetWriteExtbusCallback関数を使用します。

NotifySentSerialFunc

シリアルの送信完了通知のコールバック関数 [ユーザ定義関数]

```
void  NotifySentSerialFunc (SuoHandle handle);
```

注意　NotifySentSerialFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	シリアル・インタフェースのハンドルです。

戻り値

なし

解説

NotifySentSerialFunc ではシリアルの送信完了時の処理を記述します。

NotifySentSerialFunc をコールバック関数として登録するためには，SuoSetNotifySentSerialCallback 関数を使用します。

ReceiveSerialFunc

シリアルの受信のコールバック関数 [ユーザ定義関数]

```
void  ReceiveSerialFunc (SuoHandle handle, unsigned long data, int status);
```

注意 ReceiveSerialFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle シリアル・インタフェースのハンドルです。

data 受信したシリアル・データです。

status 受信ステータスです（次のいずれかを指定）。

値	意味
0	正常受信
SUO_PARITYERR	パリティ・エラー （パリティ・ビットの不一致の場合）
SUO_FRAMINGERR	フレーミング・エラー （ストップ・ビットが検出されない場合）

戻り値

なし

解説

ReceiveSerialFunc ではシリアルの受信時の処理を記述します。

ReceiveSerialFunc をコールバック関数として登録するためには，SuoSetReceiveSerialCallback 関数を使用します。

NotifySentWaveFunc

信号出力器の送信完了通知のコールバック関数 [ユーザ定義関数]

```
void  NotifySentWaveFunc (SuoHandle handle);
```

注意　NotifySentWaveFunc はユーザ定義の関数名のプレース・ホルダであるため，この関数名を使う必要はありません。

パラメータ

handle	信号出力器インタフェースのハンドルです。

戻り値

なし

解説

NotifySentWaveFunc では信号出力器の送信完了時の処理を記述します。

NotifySentWaveFunc をコールバック関数として登録するためには，SuoSetNotifySentWaveCallback 関数を使用します。

4.3 エラー番号一覧

提供関数からの戻り値の多くはエラー番号として返されます。なお，エラー番号は，提供ヘッダファイル (suo.h) で定義されているマクロ名称で示されます。

次表にエラー番号一覧を示します。.

表 4 － 4　エラー番号一覧

エラー番号（マクロ）	意味（上段 : 概要，下段 : 詳細）
SUO_NOERROR	エラーはありません（正常終了）。
SUO_CANTALLOC	メモリが確保できません。
	メモリが確保できませんでした。
SUO_ILLIFNAME	インタフェース名が正しくありません。
	インタフェース名に NULL，または "" が指定されました。もしくは，ハンドル取得関数において生成していないインタフェース名が指定されました。
SUO_ILLHANDLE	ハンドルが正しくありません。
	生成したインタフェースのハンドル以外のハンドルが指定されました。
SUO_ILLPARAM	パラメータが正しくありません。
	パラメータに指定できる値以外が指定されました。
SUO_CANTCALL	関数コールできません。
	MakeUserModel 関数でしか呼べない関数が，MakeUserModel 関数以外で呼ばれました。もしくは，MakeUserModel 関数以外でしか呼べない関数が，MakeUserModel 関数で呼ばれました。呼び出し規則は表 4 － 1　の備考欄を参照してください。
SUO_CONFLICTRES	生成する資源が競合しています。
	MakeUserModel 関数内で生成したインタフェース名または端子名の中に同じものが複数個存在します。
SUO_ILLFILENAME	ファイル名が正しくありません。
	ファイル名に NULL，または無効文字を含む名前が指定されました。
SUO_CANTOPENFILE	ファイルを開くことができません。
	ファイルが存在しません。またはファイルにリード許可されていません。
SUO_ILLFILEFMT	ファイルの形式が正しくありません。
	異なる種類のファイルを指定しました。
SUO_ILLFILECONT	ファイルの内容が正しくありません。
	ファイルに記載されたデータの内容に矛盾がある，またはデータが 1 つも存在しません。
SUO_ILLPINNAME	端子名が正しくありません。
	端子名に NULL，または "" が指定されました。
SUO_ILLADDRRANGE	アドレス範囲が正しくありません。
	有効なアドレス範囲ではありません。

表 4 － 4　エラー番号一覧

エラー番号（マクロ）	意味（上段 : 概要，下段 : 詳細）
SUO_UNDERSENDING	すでに送信中です。
	すでに送信中のため，新たに送信開始できません。

第 5 章　サンプル・プログラム

本章では，SM+ ユーザ・オープン・インタフェースを使用して作成したユーザ・モデルのサンプル・プログラムについて説明します。

次表にサンプル・プログラム一覧を示します。

表 5 － 1　サンプル・プログラム一覧

No.	サンプル名	概要
1	Timer	タイマ・インタフェースを使用したサンプル

5.1 Timer

5.1.1 概要

Timer モデルは，タイマ・インタフェースを使用したサンプル・プログラムです。

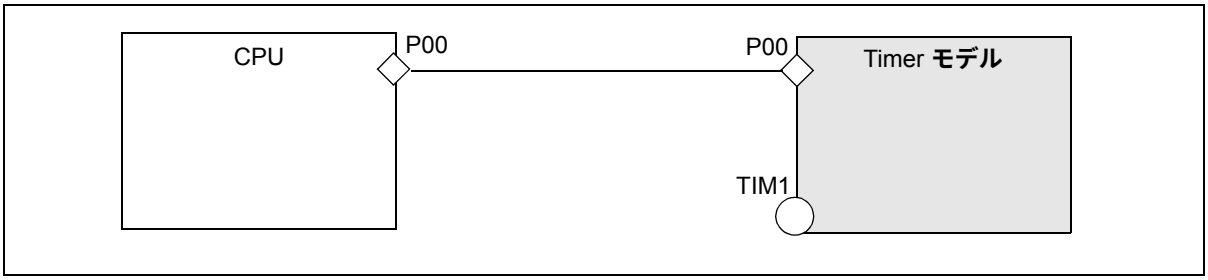
決められた時間間隔で端子に対して値を出力します。

5.1.2 構成

Timer モデルは P00 端子と TIM1 タイマを生成します。

生成した P00 端子は CPU の P00 端子に接続します。

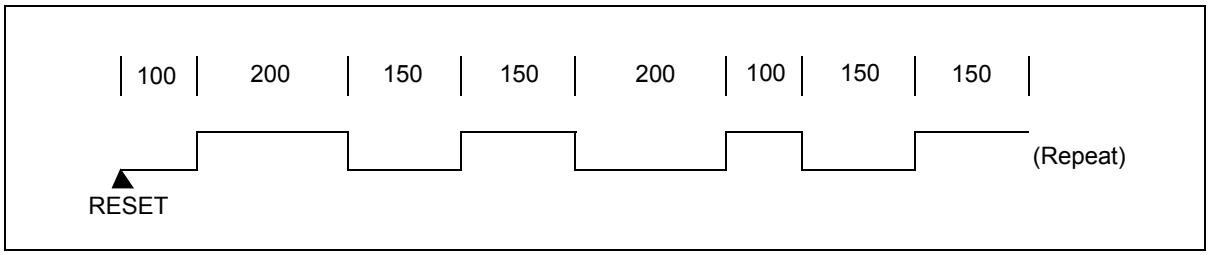
図 5 － 1　Timer モデル構成図



5.1.3 動作説明

タイマ・インタフェースを使用して決められた時間を計測し，端子 P00 に対して LOW/HIGH を交互に出力します。出力する値と時間は以下の通りです。

図 5 － 2　Timer モデル動作説明



5.1.4 プロジェクト・ファイル

次表に Timer モデルの Visual C++ のプロジェクト・ファイルの設定情報を示します。

表 5 － 2　Timer モデルの設定情報

設定情報	内容
プロジェクト・タイプ	Win32 Dynamic-Link Library
構成ソース・ファイル	..\sys\suolink.c, uo_timer.c
インクルード・ファイルのパス	..\sys（suo.h の格納フォルダを指定します）。

5.1.5 ファイル詳細

(1) モデルのソース・ファイル（uo_timer.c）

(1/3)

```
#include <windows.h>
#include "suo.h"

/* Handle */
SuoHandle p00;
SuoHandle tim1;

/* Wave-Table */
#define MAXWAVE 8
struct _WaveTable {
   unsigned long time;              /* Wait Time [MainClk] */
   int pinValue;                    /* Pin Value (SUO_HIGH or SUO_LOW) */
} waveTable[MAXWAVE] = {
   100,   SUO_HIGH,
   200,   SUO_LOW,
   150,   SUO_HIGH,
   150,   SUO_LOW,
   200,   SUO_HIGH,
   100,   SUO_LOW,
   150,   SUO_HIGH,
   150,   SUO_LOW
};
int waveIndex;

/* Declare */
void Reset(void);
void NotifyTimer1(SuoHandle handle);
void puterr(int error);

/* MakeUserModel */
SuoUserEntry void MakeUserModel(void)
{
   int error;

   /* Create interface */
   if((error = SuoCreateTimer("TIM1", &tim1)) != SUO_NOERROR){
      puterr(error);
      return;
   }
   if((error = SuoCreatePin("P00", &p00)) != SUO_NOERROR){
      puterr(error);
      return;
   }
```

(2/3)

```
    /* Set callback */
    SuoSetResetCallback(Reset);
    SuoSetNotifyTimerCallback(tim1, NotifyTimer1);
}

/* Reset callback */
void Reset(void)
{
    int error;

    /* Initialize Wave-Tabel index */
    waveIndex = 0;

    /* Output LOW(initial value) to P00 */
    if((error = SuoOutputDigitalPin(p00, SUO_LOW)) != SUO_NOERROR){
        puterr(error);
        return;
    }

    /* Set wait time */
    if((error = SuoSetTimer(tim1, SUO_MAINCLK, waveTable[waveIndex].time)) != SUO_NOERROR){
        puterr(error);
        return;
    }
}
/* NotifyTimer callback */
void NotifyTimer1(SuoHandle handle)
{
    int error;

    /* Output value to P00 */
    if((error = SuoOutputDigitalPin(p00, waveTable[waveIndex].pinValue)) != SUO_NOERROR){
        puterr(error);
        return;
    }

    /* Set next Wave-Tabel index */
    waveIndex++;
    if(waveIndex >= MAXWAVE){
        waveIndex = 0;
    }
```

(3/3)

```
    /* Set wait time */
    if((error = SuoSetTimer(tim1, SUO_MAINCLK, waveTable[waveIndex].time)) != SUO_NOERROR){
        puterr(error);
        return;
    }
}

/* Report error */
void puterr(int error)
{
    char message[80];
    wsprintf(message, "The user open interface error (0x%04x) occurred.", error);
    MessageBox(NULL, message, "ERROR", MB_OK|MB_ICONERROR);
}
```

(2) コンフィギュレーション・ファイル（smplus.cfg）

(1/1)

```
cpu = CPU('a');

# ------------------------------------------------------------
#   UO_TIMER description (CPU=uPD70F3289Y)
# ------------------------------------------------------------

# Generate uo_timer.dll
uo_timer = Device("USEROPEN", "-f=uo_timer.dll");

# Connect PIN (CPU.P00-UO_TIMER.P00)
wire_P00 = Wire(1);
wire_P00 += cpu.Port("P00/TIP61/TOP61");
wire_P00 += uo_timer.Port("P00");
```

(3) ターゲット・プログラムのソース・ファイル（lm_timer.c）

(1/1)

```
/* Target Program for UO_TIMER */

#pragma ioreg


main()
{
   unsigned char value;

   PM0 = 0xff;              /* set input mode */
   PM1 = 0x00;              /* set output mode */

   while(1){
      value = P0.0;         /* input signal from "P00" */
      P1.0 = value;         /* output signal to "P10" */
   }
}
```

付録 A　総合索引

【発　行】
NECエレクトロニクス株式会社
〒211-8668　神奈川県川崎市中原区下沼部1753
電話（代表）：044(435)5111

お問い合わせ先

【ホームページ】

NECエレクトロニクスの情報がインターネットでご覧になれます。
URL(アドレス)　http://www.necel.co.jp/

【営業関係，技術関係お問い合わせ先】

半導体ホットライン
（電話：午前 9:00～12:00，午後 1:00～5:00）
電　話　：044−435−9494
E-mail　：info@necel.com

【資料請求先】

NECエレクトロニクスのホームページよりダウンロードいただくか，NECエレクトロニクスの販売特約店へお申し付けください。