



SpaceWire/RMAP Library

Version : 2008/11/14

目次

はじめに	1
更新履歴と連絡先	1
更新履歴	1
連絡先	1
SpaceWireとRMAP	2
全体像	2
SpaceWire	4
RMAP	8
SpaceWire/RMAPをもちいた データ取得システム	11
SpaceCube Architecture	11
T-Kernel	12
SpaceCube1用プログラム開発の例	13
ターゲットノード	14
SpaceCube1とSpaceWire I/Fボードを用いたデータ取得系	15
SpaceWire/RMAP Library	16
はじめに	16
このライブラリの背景	16
SpaceWire/RMAP Libraryの機能・コンセプト	17
ドキュメンテーションについて	23
インストールと コンパイルのしかた	24
プログラムの実行	25
フォルダ構造について	25
ユーザプログラムの追加のしかた	27

ライブラリの使用例	28
例題0：main_example00.cc シンプルなRMAPアクセス	28
例題1：main_example01.cc マルチスレッドのプログラム	31
例題2：main_example02.cc マルチプルRMAPトランザクションの例	33
例題3：main_example03.cc SpaceWireパケットの送受信 (RMAPレイヤを使わない場合)	36
例題4：main_example04.cc ルータの使用	37
各クラスの使い方の詳細	39
問題点やコメント	43
現状の問題点	43
ライブラリの速度に関して	43
おまけソフトの使い方	45
rmaphongo	45
speedtest	50
参考文献	51

はじめに

このドキュメントでは、SpaceWireとRMAPのしくみを概説し、それを検出器のデータ取得や制御に利用するための方法を紹介します。その後で、POSIXやSpaceCubeのうえでSpaceWire通信やRMAP通信を簡単に行うためのライブラリである「SpaceWire/RMAP Library」の機能や、実際の使い方を説明します。

[SpaceCubeの初期設定、SpaceCube用クロスコンパイル環境の構築](#)に関しては別のドキュメント「SpaceWire/SpaceCube Tutorial」[14]も参照してください。

このドキュメントの、とくにSpaceWireとRMAPの解説の部分は、[1]と[4]を参考にして記述されています。

このドキュメントやライブラリ本体には、間違があるかもしれません。間違を見つけた場合は制作者などに連絡していただけだと、他の人たちの助けにもなります。

更新履歴と連絡先

更新履歴

このドキュメントは以下のように改訂されてきました。改訂した方は追記してください。

2008-05-10 作成開始(湯浅)

2008-05-11 第0版、全体をある程度執筆して公開

2008-05-19 ルータの導入に関して、 RMAPDestinationの説明や例題を追加。rmaphongoに、RMAPDestinationをコマンドラインから設定する機能を追加。おまけソフトの使い方の例題を追加

2008-07-25 マイナーな誤植などを修正

2008-09-18 Shimafuji SpaceWire IPが大学側標準IPコアになったので、それ用に記述を修正

2008-11-14 SpaceCube Cubeの開発の中で改善された点を反映して公開。

連絡先

このドキュメントは下記の人たちによって作成されました。改訂作業などをした場合は連絡先を追記してください。

湯浅孝行 (yuasa at amalthea.phys.s.u-tokyo.ac.jp)

また、問題の報告や解決策、議論は、SpaceWire MLで行われています。簡単な質問でもかまいませんので、なるべく多くの人たちで問題を共有して、解決できればより良いので、困ったときやすばらしい開発成果ができたときは、ぜひMLに連絡してください。MLメンバーに登録するには、国分さん(JAXA/ISAS)経由で渡辺さん(同)に連絡してください(2008年07月現在)。

SpaceWireとRMAP

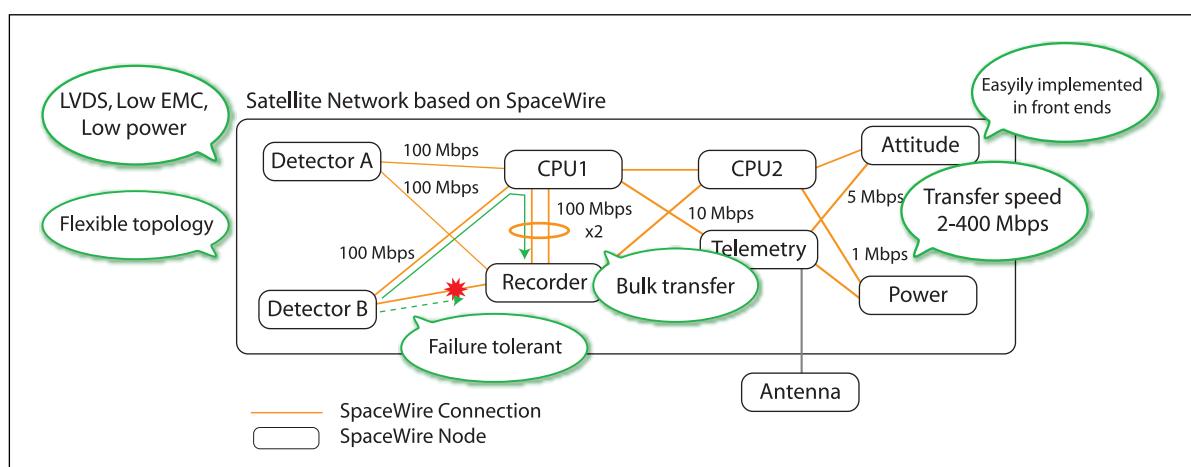
全体像

SpaceWire[1,2]は、宇宙機(衛星)内で、搭載コンポーネント間のデータ通信を行うための通信I/Fおよび通信プロトコルの仕様です。これまでの単品生産、独自仕様のI/Fが少なくなかった衛星製作において、I/Fの共通化をはかることで、コスト削減と製作期間の短縮、技術の蓄積、信頼性の向上などを目的として、ESA、JAXA、NASA、Roskosmosなどがワーキンググループを設立して、標準化作業が行われています。日本からは、JAXA/ISAS 高橋教授、大阪大学 能町教授などがsteering committeeのメンバーとして深く関わっています。また、日本国内では、JAXA、大学、メーカーが協同して「日本SpaceWireユーザ会」という組織を構成して、SpaceWireの標準化、機器開発、プロモーションなどを行ってきてています[5,6,7]。

SpaceWireはすでに世界の多くの衛星計画で機内ネットワークI/Fとして採用され、Swift衛星やMars Express探査機などで宇宙での利用も行われています。日本も、SDS-I衛星のSWIMをはじめとして、BepiColombo/MMO、NeXT衛星などで採用されることが決まっています。

SpaceWireのおもな特長は、以下のようにになります。

- LVDSとDSリンクによるpeer-to-peerの高速シリアルリンク
- 幅広い転送速度(2~400 Mbps)
- 自由なパケットサイズ
- 簡単なプロトコル(実装時の省リソース、省電力)
- ルータを用いたネットワーク自由なネットワークトポロジー
- 自由なトポロジーによって可能になる、豊富な冗長系構成



SpaceWireは、コネクタ形状、信号線本数、信号レベルなどの物理層と、ノード間でパケットを送受信する際のエンコーディング方式、パケッティング方式、エラー処理手順などを定めていますが、パケットの中身(カーゴ,Cargo)の解釈までは規定していません。したがって、実際にSpaceWireを用いて何らかのデータ転送を行うためには、SpaceWireの上位レイヤに相当するプロトコルを用いる必要があります。

SpaceWireワーキンググループでは、そのような上位プロトコルのひとつとして、既にRemote Memory Access Protocol ([4]; RMAP)の標準化作業を進めています(2008年5月現在、Draft F)。RMAPは、「コマンドパケット Command Packet」とそれに対する「リプライパケット Reply Packet」の組を、SpaceWireネットワークを通じて送受信することで実行される「トランザクション Transaction」を単位として動作します。より具体的には、Read/Writeしたいアドレスや長さの要求をコマンドとしてホスト(RMAPのことばで言うと、Source)側から、ターゲット側(同じく、Destination)に送り、ターゲット側でその内容を解釈・実行します。実行結果は、リプライパケットとしてホスト側に返され、Readの場合はこの中に読み出しデータも含まれます。

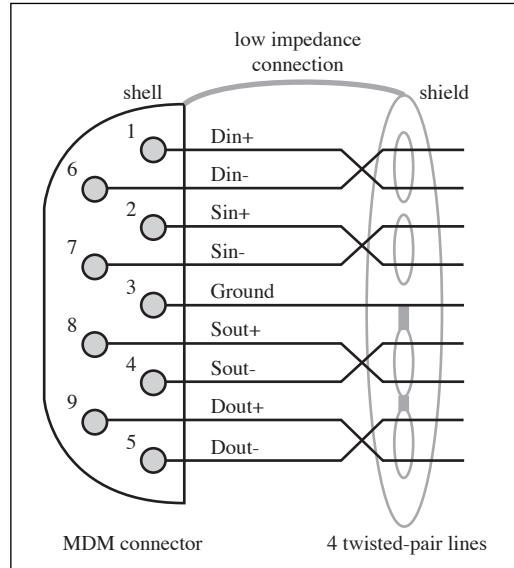
RMAPもSpaceWire同様にプロトコル体系がシンプルなので、FPGAなどにハードウェアロジックとしてプロトコルスタックを構築可能で、CPUを必要としないため、省リソースで実装が行えます。また、各ノードにRMAPを実装することで、SpaceWireネットワークでつながった機器間に、リモート機器のローカルなメモリ空間(たとえば検出器のデータメモリや、パラメタレジスタ、温度計モジュールの温度レジスタなど)にアクセスする手段が提供されることになります。CPUを搭載したインテリジェントなノードに、RMAPをソフトウェア的に実装すれば、その上で動作するソフトウェアからみると、各ターゲットノード(CPUなし)のメモリやレジスタも、あたかも自分のCPUのメモリであるかのように読み書きすることができるようになります。

以下、SpaceWireとRMAPの概説です。

SpaceWire

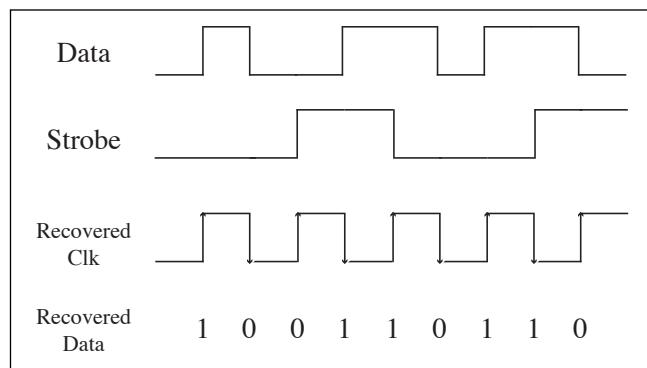
コネクタ

SpaceWireでは、9ピンのマイクロD-Subコネクタ micro D-subminiature connector(MDM)を使用します。片方向の信号を表現するのに、DSリンクでデータ Dataとストローブ Strobeの2組を必要とします。各信号は、LVDS(Low Voltage Differential Signaling)で伝送されるので、1信号につきpositiveとnegativeの2本の伝送線が使用されます。結果、両方向で $2(D,S) \times 2(\text{differential}) \times 2(\text{duplex}) = 8$ 本の信号が必要になり、9ピンコネクタに以下のように接続されます。この配線は、RS-232Cの通信などで使用するD-Sub 9ピンのシリアルケーブルなどとは配線が異なるので注意が必要です。MDMは高価(一個数千円～数万円)で、抜き差し回数の制限もあるので、(日本の?)大学などの実験室レベルでの開発では、D-Sub 9ピンのコネクタを使用しています。



DSエンコーディング

送りたい情報(1や0といった論理的なビット列)を電気信号でどのように表現するかの方法はいろいろあります。SpaceWireではDSエンコーディング(もしくはDS Link)という方法を採用しています。DSエンコーディングは、SpaceWireの前身であるIEEE 1355という規格や、PCのI/Fとして利用されているFireWire (IEEE 1394, iLink)にも採用されている方式で、DataとStrobeの二つの信号線で1つの信号を表し、データ列



自身と、それをデコード(復元)する際に必要なクロック情報を同時に送信します。具体的には、Dataは、送りたい情報(1や0)そのまま表し、Strobeは、11や00などのように、Dataが変化しないときに変化させます。受信側では、DataとStrobeのXOR(exclusive OR, 排他的論理和)を計算することで、クロック(の1/2のクロック)が復元でき、それをもとにDataをラッチすることで、もとのデータ列を取得できます。

Characterレベル

SpaceWireはパケットを送受信することでデータ通信を行いますが、そのパケットの中身('a'とか'b'とか'6'といった文字)や、パケットの終端(区切り目)をどうやって表現するかも決まっています。SpaceWireのCharacterレベルにはCharacterとコード codesが規定されていて、以下のような内容になっています。

- Data Character

普通の文字。パリティビット(1bit)、データフラグ('0' fix)、データ8bitの計10bitで1文字を表現。

- Control Character

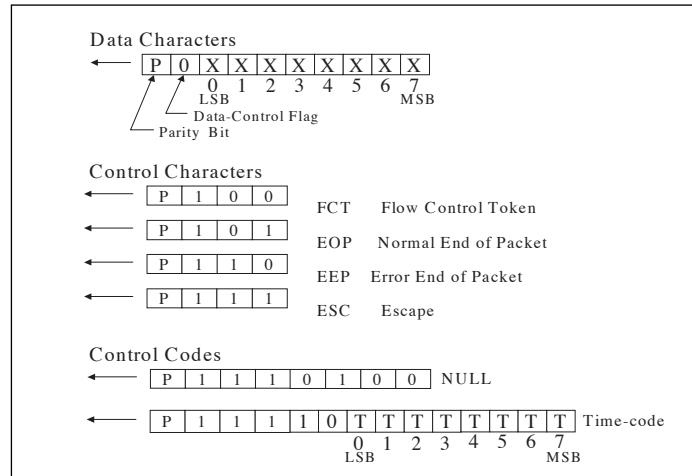
制御文字。フロー制御トークン(Flow Control Token; FCT)、パケット終端(End Of Packet; EOP)、パケットエラー終端(Error End of Packet; EEP)、エスケープ記号(Escape; ESC)。パリティビット(1bit)、制御フラグ('1' fix)、制御文字タイプ(2bit)の計4bitで1文字を表現。

- Control Codes

ナル文字 NULL Characterと時刻コード Time Codes。NULLはESC+FCTとして表現。Time Codeはパリティ(1bit)、4bitの1、1bitの0に続く、8bitの時刻データの計14bitで表現。

SpaceWireリンクでは、データ溢れを防ぐために、フロー制御の仕組みが用いられます。各SpaceWireノードは、自分の受信バッファにあと8バイトのData Characterを受信する余裕があるときに、FCTを相手に送信します。相手ノードでは、FCTの数と、自分が送ったデータ文字数を常に比較して、相手のバッファがあふれないように、送出量を調節する必要があります。SpaceWireの仕様では、各ノードは、相手から届いたFCTを最大で7個まで記録できるように実装することが求められています(7個と決まっているのは、3ビットのレジスタで計数できる範囲、ということ)。

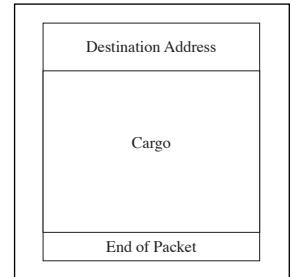
また、送信するデータがない(または準備できていない)ときには、NULLを送信することが定められていて、「何も送っていない状態」はSpaceWireには存在しません。データ通信を行っていないSpaceWireリンクをプローブで測定すると、NULLを送受信しあっているのがみえます。



(図は[1]より抜粋)

パケット

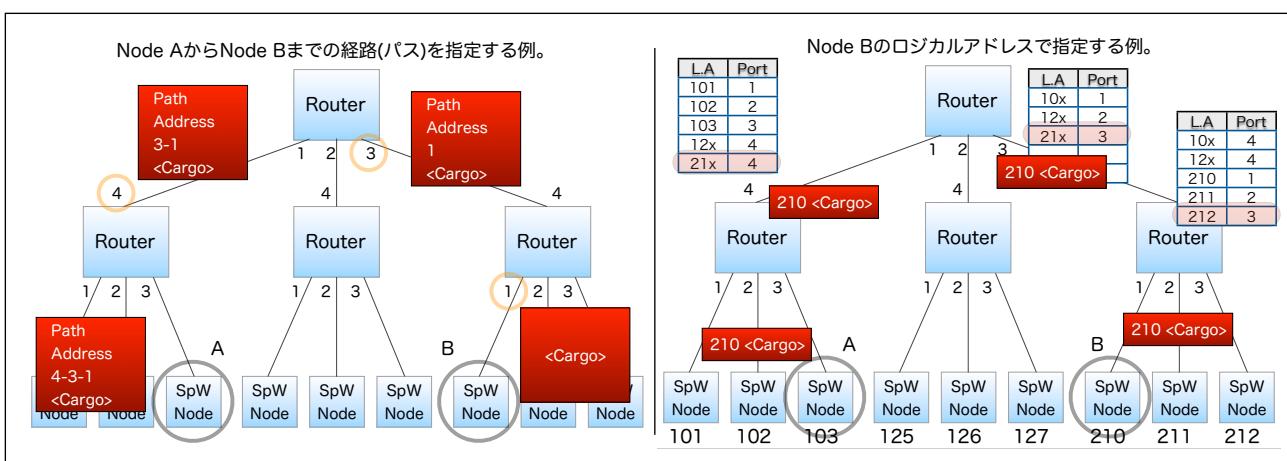
SpaceWireのパケットは、「宛先(SpaceWireのことばで言うと、Destination)」とパケットの中身(カーゴ Cargo)、パケット終端 EOPから構成されます。Destinationの部分には、少なくとも宛先のロジカルアドレス(後述)が含まれる必要があります、場合によっては、バスアドレス(同)がその前に付加されます。RMAPなど上位レイヤのパケットは、Cargoの部分に格納されて転送されます。



アドレッシングとルーティング

SpaceWireは単に1対1の通信を行うための規格ではなく、複数のノードがルータを介して接続されるネットワークを前提としています。各ノードは、それを区別するためにアドレスを持ちます。SpaceWireネットワークに置けるアドレスの付け方は2種類あり、バスアドレッシング Path Addressingと、ロジカルアドレッシング Logical Addressingと呼ばれます。

Path Addressingは、あるノードからあるノードまでの経路を、パケットが通るべきルータのポート番号の列で表現する仕組みで、図では、ノードAからノードBまでの経路をたどるパケットの例を示しています。Path Addressは、ルータを通るたびに、「その先頭の1バイトが解釈され、適当な出口ポートが選択され、そのルータを出るときにはその1バイトは消去された状態で次のルータへ到達し、さらに解釈される…」という作業を繰り返して目的地へ到達します。Path Addressは0から31までの1バイトの数字が使用され、0はルータ内部のバーチャルなコンフィグレーションポートに、1-31はルータの実際のポート番号に対応します。



Logical Addressingは、Internet Protocolで使われるIPアドレスのようなもので、各ノードがSpaceWireネットワーク(もしくはサブドメイン)において、固有のアドレスを割り当てる方式です。Logical Addressには、Path Addressで使われる0-31を除いた、32から254までの数字が使用可能です(上限が254なのは、8bitで表現できる0-255のうち、255がreservedなため)。Logical Addressingでは、ルータは、各Logical Addressに対応するポート番号を表にした情報を持っていないと、パケットを正しくルーティングできません。この表は、ルーティングテーブル Routing Table、もしくはルーティングマトリックス Routing Matrixと呼ばれ、Logical Addressingによる通信を開始するまえに、ユーザがルータに書き込む必要があります。この作業には、普通はPath Addressingを用います。Logical Address 254はDefault Logical Addressとし

て規定されており、ひとつのSpaceWireネットワークの中で複数のノードが254というLogical Addressをもつ可能性があるので、普通はそれ以外の値を割り当てるようになります[3]。また、あるノードがLogical Addressをもたないときや、相手のノードのLogical Addressを知らないときには、254が使用されるので、「自分は254以外のLogical Addressを持つ場合でも、届いたパケットの宛先がLogical Address 254の場合は、自分宛てであると思って解釈する」もしくは「Logical Address 254のパケットが届いても読まずに捨てる」というどちらかの選択をして実装を行う必要であるとされています[3]。

上記二つのアドレッシングモードは、一つのSpaceWireネットワークの中で混在させることが可能で、ルータはパケットを先頭から解釈していき、0-31の数字がある場合はPath Addressingで、また32-254の場合はLogical Addressingでパケットのルーティングを行います(注：上にも書きましたが、[3]によれば、254は普通のLogical Addressとして使用すべきでなく、したがってルーティングも行うべきでないとされています)。

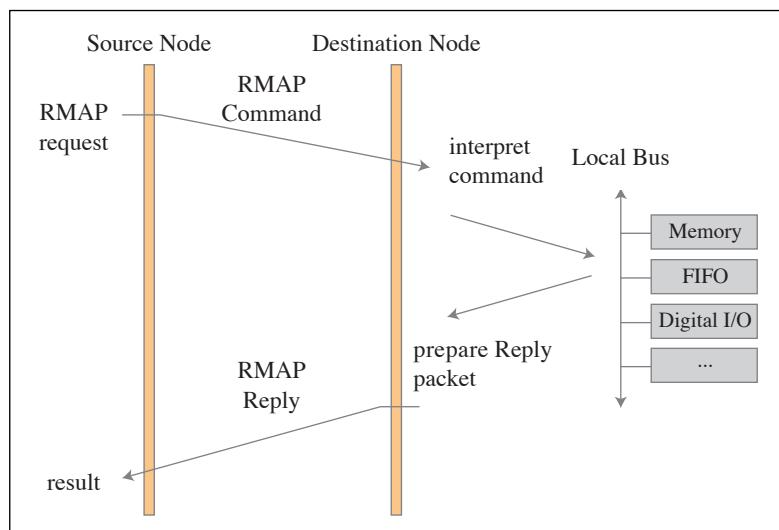
リンクの初期化とエラー処理

SpaceWireリンクの電源が投入された状態から、リンクを確立し、ノード間で相互にパケットの送受信ができるようになる状態までの、初期化手順は仕様書で規定されています。通信断絶や、parityエラーなど、SpaceWireリンクにエラーが生じた際の処理手順(切断手順、復帰手順)も規定されています。それぞれの手順には、状態遷移図が用意されていて、SpaceWireのプロトコルスタックを開発するときには、その遷移図をハードウェア記述言語のステートマシンなどとして実装することになります。初期化の実際の手順を概説すると以下のようになります。エラー処理はクリティカルで、かなり難しい問題なので、仕様書を参考にしてください。

ノードAとBが初期化処理を行う場合の例を示します。まず、双方のノードがNULLを送出しつづけます。ノードAが相手からのNULLを受信すると、自分がノードBからのNULLを受信し、初期化手順を開始したことと、自分の受信バッファがパケットを受信可能であることを示すため、ノードBに向けてFCTを少なくとも1個送出します。FCTを受け取ったノードBは、ノードB→ノードAの方向の初期化手順が完了したとわかるので、その方向のパケット送信を開始します。逆方向も同様にして初期化されると、全二重 full duplexでのパケット送受信が可能になります。

RMAP

RMAPは、もともとはSpaceWireネットワークをコンフィグレーションするための目的で作られたものですが、シンプルなプロトコルと、さまざまな機能のおかげで、SpaceWireノード間のデータ転送やパラメタ設定などにも利用できます。日本SpaceWireユーザ会が開発してきた、SpaceWireによるデータ取得系も、データ転送にはRMAPを用いています。前述のように、RMAPでは、コマンド RMAP Commandとそれに対するリプライ RMAP Replyで通信が行われます。それぞれ、規定されたパケット形式がありますが、以下では、そのパケット構造を理解するために必要なキーワードのみ説明します。各情報がどういう順番でならんでいるか、など具体的なパケット構造は仕様書を参照してください。



RMAPに登場するキーワード

- Destination/Source
RMAPでアクセスする目的地ノードがDestination。RMAPアクセスを開始するノードがSource。
- Header/Data
RMAPパケットは、Header partとData partという二つの部分で構成される。Write Command PacketやRead Reply Packetには、それぞれ、書き込むデータや読み出したデータがData partとして付随するが、Write Reply PacketやRead Command Packetなど、Data partが必要ないパケットでは、Headerだけの場合もある。
- Destination Path Address
SpaceWireレイヤの定義と同じ。Sourceノードからみたときの、DestinationノードまでのPath Address。
- Destination Logical Address
SpaceWireレイヤの定義と同じ。DestinationノードのLogical Address。
- Source Path Address
Destinationノードからみたときの、SourceノードまでのPath Address。Reply PacketをPath Addressingで返送するときは、Sourceノードがこの項目をコマンドパケットに書いて、Destinationに知らせる必要がある。

- Source Logical Address
SourceノードのLogical Address。
- Protocol Identifier
SpaceWireの上位プロトコルは、固有のProtocol IDがわりあてられている。SpaceWireパケットのCargo部の先頭にProtocol IDを記述することで、そのパケットが何プロトコルのパケットか判別できるようになっている。
- Packet Type, Command, Source Path Address Length
このRMAP Packetが、CommandなのかReplyなのか、ReadなのかWriteなのか、Source Path Addressはあるかないか、あるとしたら長さはどれだけか、を表すための1バイト。記法は仕様書参照。
- Status
RMAP Reply Packetに含まれる、Commandの成功/失敗を表す1バイト。失敗の場合は、エラーの種類が格納される。
- Destination Key
Destination側がRMAP Command Packetを受信したときに、そのCommand PacketのCommandを実行してもよいか判断するために使われる認証キー。Source側は、Destinationが受理するDestination Keyを知らないとRMAP Commandを実行できない。Destination Keyが受理されないと、エラーを示すRMAP Reply PacketがSourceに返される。
- Transaction Identifier
RMAP通信の番号を表すID。Source側でRMAP Command Packetに付けた番号が、RMAP Reply Packetでも使用されるので、Source側でこのIDをデータベースにしておくことで、複数トランザクションを同時に実行し、返ってきたRMAP Reply PacketのTransaction IDを調べて対応するトランザクションだけ完了させ、他のものは待つ、ということができる。
- Extended Address
拡張アドレス。普段は使用しない。
- Memory Address
RMAPアクセスの対象となる、Destinationノードのローカルなメモリ空間のアドレス。32bit指定できるので、それぞれのノードに仮想的に4GBの空間があることになる。
- Data Length
上記Memory Addressから何バイトアクセス(Read/Write)するか、のバイト数。
- Header CRC/Data CRC
Header partやData partの最後に付けられる、Cyclic Redundancy Check code。各partのバイト列から、ある数式をもとに計算した1バイトで、Source側で計算して付記する。Destination側で、受信パケットに対して再計算し、Source側が書いた値と比較することで、伝送途中でパケットの中身が化けていなかを判定できる。

RMAP Packet

RMAP Command/Reply Packetの例を、[1]から抜粋して掲載しておきます。

RMAP Command Packet			
<i>First Byte Transmitted</i>			
Destination Logical Address	Protocol Identifier	Packet Type, Command, Source Path Addr Len	Destination Key
Source Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended Write Address
Write Address (MS)	Write Address	Write Address	Write Address (LS)
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC	EOP	
<i>Last Byte Transmitted</i>			
RMAP Reply Packet			
<i>First Byte Transmitted</i>			
Source Logical Address	Protocol Identifier	Packet Type, Command, Source Path Addr Len	Status
Destination Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Reserved = 0
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC	EOP	
<i>Last Byte Transmitted</i>			

RMAPのコマンド

RMAPには、Read/Writeと、それらを組み合わせたRead-modify-Writeというコマンドがあります。ユーザが各コマンドのRMAPアクセスを実行するときには、それに対応したRMAP Command Packetを作り、Destinationに送信し、返ってきたRMAP Reply Packetの中身を解釈するという作業が必要になります。しかし、各ユーザがそれらの作業を自分で記述していくは効率がわるいので、SpaceWire/RMAP Libraryでは、パケット生成/解釈とパケットの送受信をライブラリ側で行ってしまい、ユーザはRMAPアクセスを隠蔽するクラスのread()/write()メソッドを呼べばよい、という仕組みになっています。

SpaceWire/RMAPをもちいた データ取得システム

日本SpaceWireユーザ会では、SpaceWireベースの衛星搭載検出器開発のプラットフォームとして、大きく分けて二つのカテゴリのSpaceWireデバイスを開発しています。一つ目が、CPUを搭載し、SpaceWireネットワークのなかでホスト(マスター)としてデータ取得や検出器制御を行う「インテリジェントなノード」であり、もう一つがCPUを搭載せず、インテリジェントノードからアクセスされる立場の「ターゲットノード」です。

SpaceCube Architecture

CPU上でリアルタイムOSが動作し、SpaceWire I/Fを通じてSpaceWireネットワークに参加するコンピュータの枠組みを、SpaceCube Architectureとして定義し、複数の種類のSpaceCube型コンピュータが開発されています。その中でも、地上試験でこれまで主に使用してきたのが、SpaceCube1(シマフジ電機/JAXA)です。

SpaceCube by Shimafuji & JAXA

CPU : VR5701(64bit MIPS) 200MHz

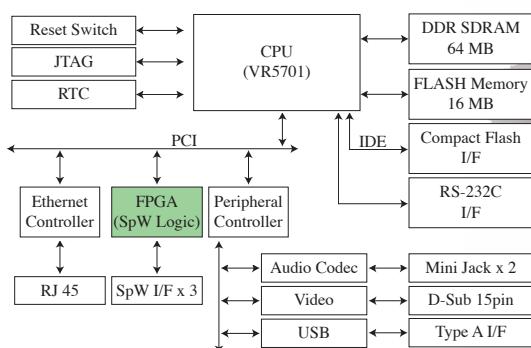
RAM : 64MB

OS : T-Kernel(TRON) or Linux

I/F : SpaceWire x 3, Fast Ethernet,
Serial, USB, VGA, CF

Power : DC +5V

Size : 52x52x55 mm³



宇宙用のSpaceCube型コンピュータとしては、JAXA認定CPU HR5000搭載の宇宙仕様 SpaceCube2(NTSpace/JAXA)や、SpaceCard(MHI)があります。下の写真は、SpaceCube2の外観です。SpaceCube2はSDS-1/SWIMの処理用コンピュータとして実際に利用されています。



CPU	HR5000 64bit 320MIPS
RAM	8+256MB
FlashMemory	8+256MB
I/F	SpaceWire x 3, RS422
Power	DC 5V
Size	71x220.5x175.5 mm ³

T-Kernel

T-Kernelは、日本で策定されたリアルタイムOSの一種で、東京大学 坂村教授が中心となって行ってきたTRONの流れをうけつぐものです。TRONは、ハードウェアリソースがあまりリッチではなかったり、リアルタイム性がもとめられたりする、「組み込みシステム」のフィールドで広くつかわれていていま。これは、OSの仕様がオープンで各社がその仕様に基づいて自由に実装可能というのが大きな要因で、例えば携帯電話や家電製品の制御システムとして、TRON全体としては、世界的にも有数のシェアを獲得しています。衛星内システムでは、これまでもいろいろな種類のリアルタイムOSが採用されてきましたが、SpaceCube Architectureでは、T-KernelをOSとして利用します。

T-Kernelは、ハードウェアに近いレイヤから、高度に抽象化されたレイヤまで、複数のソフトウェアレイヤから構成されており、使用方法によって、それらの中から必要なレイヤだけを使う、といったことも可能になっています。主なレイヤとしては、「T-Kernel」と「T-Kernel/Extension」があり、それじれ、タスクベース、プロセスベースのソフトウェア実行環境を提供します。「タスク」とは、UNIXのことばで言うと「スレッド」に、「プロセス」は同じく「プロセス」に相当します。SpaceCube1では、T-Kernel/Extensionレイヤまで利用できるOSを搭載しているため、プロセスベースで、UNIXに近いイメージで作業が行えます。一方、ハードウェアリソースなどの問題から、T-Kernelレイヤまでしか利用できないOSの場合は、プロセスは一つしか動作せず、そのプロセスの中で複数のタスク(スレッド)が、同時並行で処理を行う、ということになります。

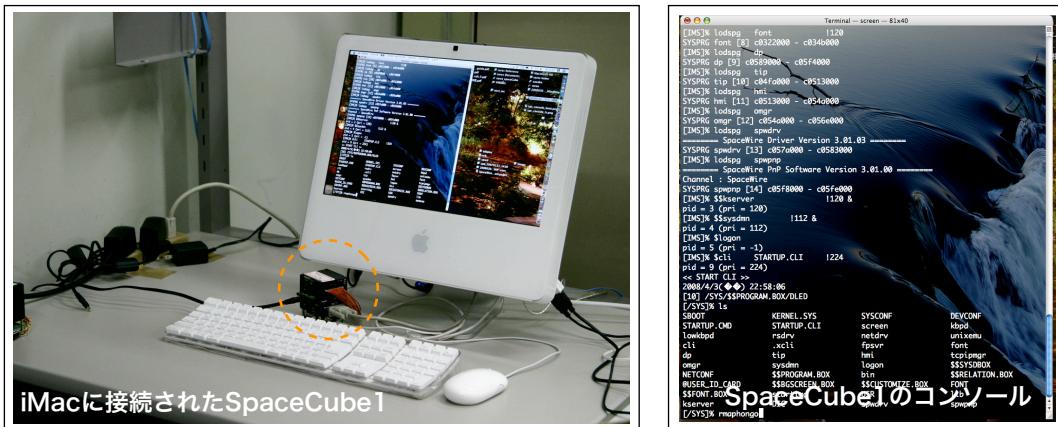
T-Kernel用のプログラム開発は、gccをもちいて、C/C++言語で行えます。ただし、UNIXのプログラムで頻繁に利用される「POSIX」のシステムコールは、T-Kernelでは利用できません。C/C++標準の関数だけをつかったコードであればすぐに移植可能ですが、OSやハードウェアの機能を利用するプログラムは、システムコールの名称がT-KernelとUNIXの世界では全く違うので、ソースコードをそのままコンパイルしようとしてもうまくいきません。

また、厳密なタスクスケジューリングなど、リアルタイムOSの機能をちゃんと使いたい場合は、T-Kernelのコンセプトと実装を、ある程度把握しておく必要があります。詳細は[a]などを参考にするとよいと思います。

SpaceCube1用プログラム開発の例

環境のたちあげ方法、チュートリアル、サンプルプログラムの開発は「SpaceWire/SpaceCube Tutorial」[14]に記述されています。以下、簡単に説明しておきます。

SpaceCube1は、それ自身でもCRTやキーボードを接続すれば、普通のPCのように利用できますが、普段はシリアルケーブル経由で外部コンソール(UNIXのコマンドラインのようなもの)を利用します。現状では、SpaceCube1上のセルフコンパイル環境が整っていないので、普段はUNIXマシンやMacintosh上のクロスコンパイル環境で開発を行います。UNIXやMacでコンパイルしたバイナリを、シリアルケーブル、もしくはFTP(後述)でSpaceCube1に転送してから実行します。



FTPの利用方法

シリアルケーブルでの実行形式の転送は、かなりの時間がかかるため、バグの修正と実機での実行を繰り返すデバッグの段階では、相当な時間ロスになりかねません。SpaceCube1のおまけコマンドとして提供されている、fgetというプログラムを使うと、FTP経由でLAN内のサーバからファイルをダウンロードすることができ、こちらの方がシリアル通信よりも高速なので、利用をおすすめします。とくに、UNIXやMacintoshで開発を行っている場合は、そのマシン単体で簡単にFTPサーバデーモンを実行できます(例えばMacなら、「システム環境設定」→「共有」→「FTPサービス」のチェックボックスをチェックするだけ)。FTPサーバを起動したあとで、SpaceCube1のコマンドラインから、

[/SYS]% fget (サーバ名):(ファイルパス)

(例) fget himawari.phys.s.u-tokyo.ac.jp: /usr/local/te/TestProgram/vr5701.spc/main

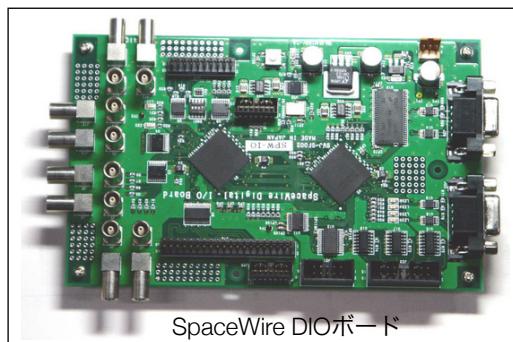
などと入力し、ユーザ名とパスワードを入れれば転送が行われます。

注：FTPでは、ユーザ名やパスワードが平文でネットワークを流れるので、各機関のネットワークポリシーによっては、別の対応(例えばSpaceCubeと開発コンピュータだけで、他から切り離されたネットワークにする、など)が必要になかもしれません。

ターゲットノード

ターゲットノードは、非同期で発生しうる検出器データを同期化(バッファリング)たり、アナログや独自のデジタル信号といったデータ形式を変換してメモリに保存したりすることで、検出器をSpaceWireネットワークからアクセス可能にする役割を担います。これらのノードにはFPGAやASICなどによって実装されたSpaceWire/RMAPプロトコルスタックが搭載されていて、内部のメモリに蓄積されたデータは、SpaceCubeがRMAPによって読み出します。

地上試験用には、シマフジ電機とJAXAなどが協力して開発した、複数の種類のSpaceWire I/Fボードが利用されています。多く使われているボードは、SpaceWire DIOボード(Digital In/Out)、SpaceWire AD/DAボード、SpaceWire FADCボードなどです。これらのボードには、SpaceWire/RMAPプロトコルスタックを搭載したFPGA(SpaceWire FPGA)以外に、ユーザが自由に書き換えられるFPGA(UserFPGA)を別に搭載しており、各ユーザは、自分の検出器用の処理ロジック(データ変換や検出器制御)をこのFPGAに書き込んで利用します。

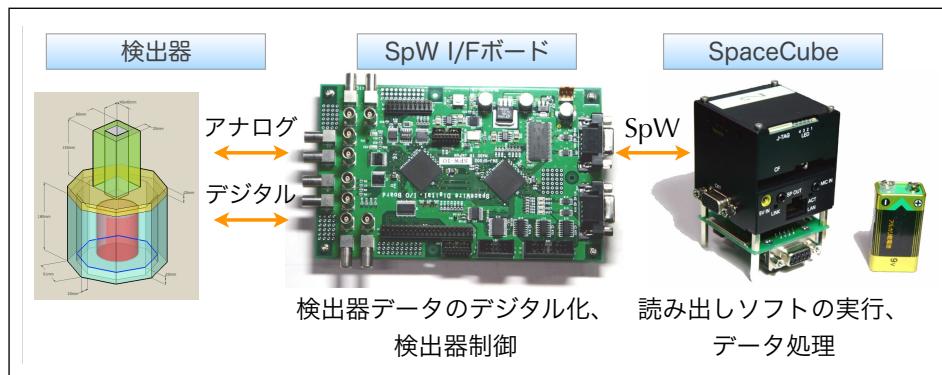


SpaceCube1とSpaceWire I/Fボードを用いたデータ取得系

SpaceCube1とSpaceWire DIOボードなどを用いて、検出器を制御したりデータ取得を行うための枠組みの開発は、2005～2006年頃から行われています。全体像としては、

- ・ 検出器とSpaceWire I/Fボードがアナログ信号やデジタル信号でつながっており、検出器データはボード上のFPGAやメモリに蓄積される
- ・ ボード上に蓄積されたデータを読み出したり、ボードを介して検出器を制御するための、データ取得ソフトウェアがSpaceCube1で動作している
- ・ そのデータ取得ソフトウェアは、SpaceWire経由で取得したデータをSpaceCube1に接続されているコンパクトフラッシュCFに保存する
- ・ CFの書き込み速度が遅くて問題となる場合は、SpaceWire経由で取得したデータ列をそのままEthernetに投げて、TCP/IP経由で別サーバ上のレコーダーソフトに送信し、サーバのHDDに保存

という枠組みでした。



各ユーザが共通に必要になるであろう機能、たとえば、「SpaceCube1上で動作するデータ取得プログラムのための、RMAPアクセスのカプセル化用ライブラリ」や、「SpaceWire I/Fボード上のFPGA内でデータを処理し、SpaceWire経由で読み出し可能な形に整えるための、HDL(ハードウェア記述言語)テンプレート」は、SpaceWireユーザ会で開発し、SpaceCube1、SpaceWire I/Fボード用にそれぞれ、「SpaceWireRMAPHongo Library」、「UserFPGA Template」として公開しました。それらの使用方法は「SpaceWireのつなげかた」[A]にまとめられています。

このデータ取得の枠組みや、SpaceCube1用ライブラリ、UserFPGAテンプレートを用いて、実際に複数の実験で検出器データの読み出しが行われ、成果が報告されています[8,9,10,11,12]。

SpaceWire/RMAP Library

はじめに

まず、今回のライブラリが必要になった背景を説明し、その後でライブラリの機能を説明します。さらに、実装時のコンセプトを説明し、ファイル構成、クラス構成を概観してから、インストール方法と使い方を説明します。「コンセプトなんて知らないくてよい」という場合は、インストールの項まで読み飛ばしてしまってかまいません。

このライブラリには間違いや、設計のまずいところがまだ多く残っていると思います(既知の問題点は後述します)。それを発見・修正された方は、ぜひメーリングリストなどで報告・議論していただければと思います。

このライブラリの背景

前述のように、各ユーザがRMAPパケットを自分で編集してRMAP通信を行うのでは開発効率が悪いので、われわれは、SpaceWireやRMAP通信の処理を内部的に行ってくれるライブラリを開発して公開していました(SpaceWireRMAPHongoライブラリ)。このライブラリを使えば、read()やwrite()といったメソッドを呼ぶだけで、RMAPパケットの中身を意識することなく、SpaceWire/RMAP経由でデータ通信が行えるようになっていました。

しかし、これまでのライブラリは、開発初期のさまざまな事情から、基本的にSpaceCube1専用で、しかもNEC Softさん製のSpaceWire IPコア(リファレンス実装という形で、SpaceWireユーザ会に配布していただいたもの)にのみ対応していました。そのため、SDS-I/SWIM用の機上ソフトウェアの開発では、SpaceCube1からSpaceCube2への移行の際に、中身をほとんど全て書き換える必要はなくなりました。また、別のメーカーさんからもSpaceWire I/Fが提供されるようになり、ことなるコンピュータの上で、複数のSpaceWire I/Fの実装を統一的に利用できるような枠組みが必要になってきました。

また、SpaceWireルータが複数のメーカーさんから利用可能になり、SpaceCube1と一枚のSpaceWire I/Fがつながっていた、これまでのセットアップよりも複雑なセットアップを想定した開発が必要になってきました。また、その中で、複数トランザクションの同時実行(以下、単にマルチトランザクションと呼びます)や、RMAPアクセスを用いた擬似的な割り込み処理も行えるような環境があると、行える実験の幅も広がると考えられます。

そこで、これまでのライブラリの設計を見直して、新たにSpaceWire/RMAP Libraryを開発しました。

SpaceWire/RMAP Libraryの機能・コンセプト

今回のライブラリは、以下の機能やコンセプトを実装しています。

- SpaceWireレイヤとRMAPレイヤの切り分け

これまでのライブラリでは一体となっていたSpaceWireレイヤとRMAPレイヤを、仕様書にもとづいて明確に区別。SpaceWireレイヤはSpaceWire経由でのパケットの送受信のみ担当。パケットの中身の生成、解釈はRMAPレイヤが行う。SpaceWire、RMAPレイヤのクラスはそれぞれ、SpaceWire、RMAPという語を頭につけて区別。

- SpaceWire I/Fのカプセル化 (SpaceWireIF class)

APIやドライバ仕様の異なるSpaceWire I/Fでも、同じようにパケット送受信が可能。利用するSpaceWire I/Fが変更になっても、上位のレイヤは影響をうけない。

- RMAPのラッパーを改良 (RMAPSocke class)

以前のライブラリで、ネットワーク(ルータ)対応時に問題となった部分を改良。あるRMAP DestinationにひもづけられたRMAPSockeクラスをオープンし、そのRMAPSockeに対してread()/write()を行うことで、RMAP通信が実行可能。ルーティングやアドレスの解釈は自動的に行われる。RMAPSockeクラスが、これまでのライブラリのSpaceWireRMAPHongo classに対応。

- RMAPのマルチトランザクションに対応 (RMAPEngine class)

複数のターゲットに対し、RMAP Commandを同時に実行可能。RMAP Replyが返ってきた段階で処理が再開されるので、「タイムアウトがない」というRMAPアクセスの特長をいかして、pull型のアクセスだけで、擬似的な割り込みを実装することも可能。

- Threadのカプセル化 (Thread, Mutex, and Condition classes)

マルチトランザクションは、本質的にマルチスレッドを必要とするので、UNIXのpthread、T-Kernelの「タスク」を、C++のクラスとしてカプセル化。OSの違いを意識することなく、JavaのThreadクラスのように簡単にマルチスレッド(T-Kernelのことばで言うと、マルチタスク)プログラムを記述可能。また、スレッドの同期 synchronizationやイベント通知 signallingに必要なクラスも実装。

- 複数OSでの移植性

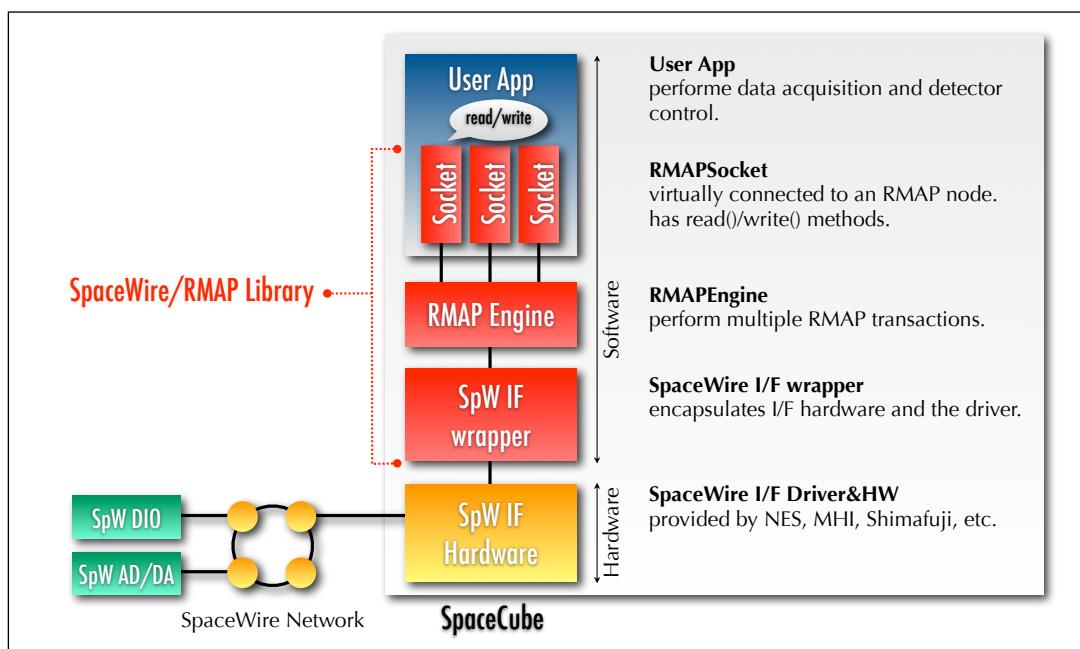
なるべく標準C++ライブラリだけを用いて記述しているので、ハードウェア依存の部分以外は、T-KernelでもUNIXでも同様にコンパイル・動作可能。実際、開発段階の試験の多くはUNIX上で実行。将来的にSpaceWire I/FがPCから利用可能になったり、SpaceWire Tunnelのように、SpaceCube1のSpaceWire I/FがTCP/IP経由でPCから利用可能になったりすれば、UNIX上だけでデータ取得プログラムを書けばよい、ということにもできる。

SpaceWire IFのカプセル化

このライブラリでは、SpaceWire I/F(もしくはIPコア)を、実際のハードウェアやドライバ関数をラップした、SpaceWireIFクラスとして実装することで、プログラムの移植性を高めています。これにより、ハードウェアがどのようなSpaceWire I/Fでも、パケットの送受信はSpaceWireIF::send()やreceive()といったメソッドで統一的に行えます。

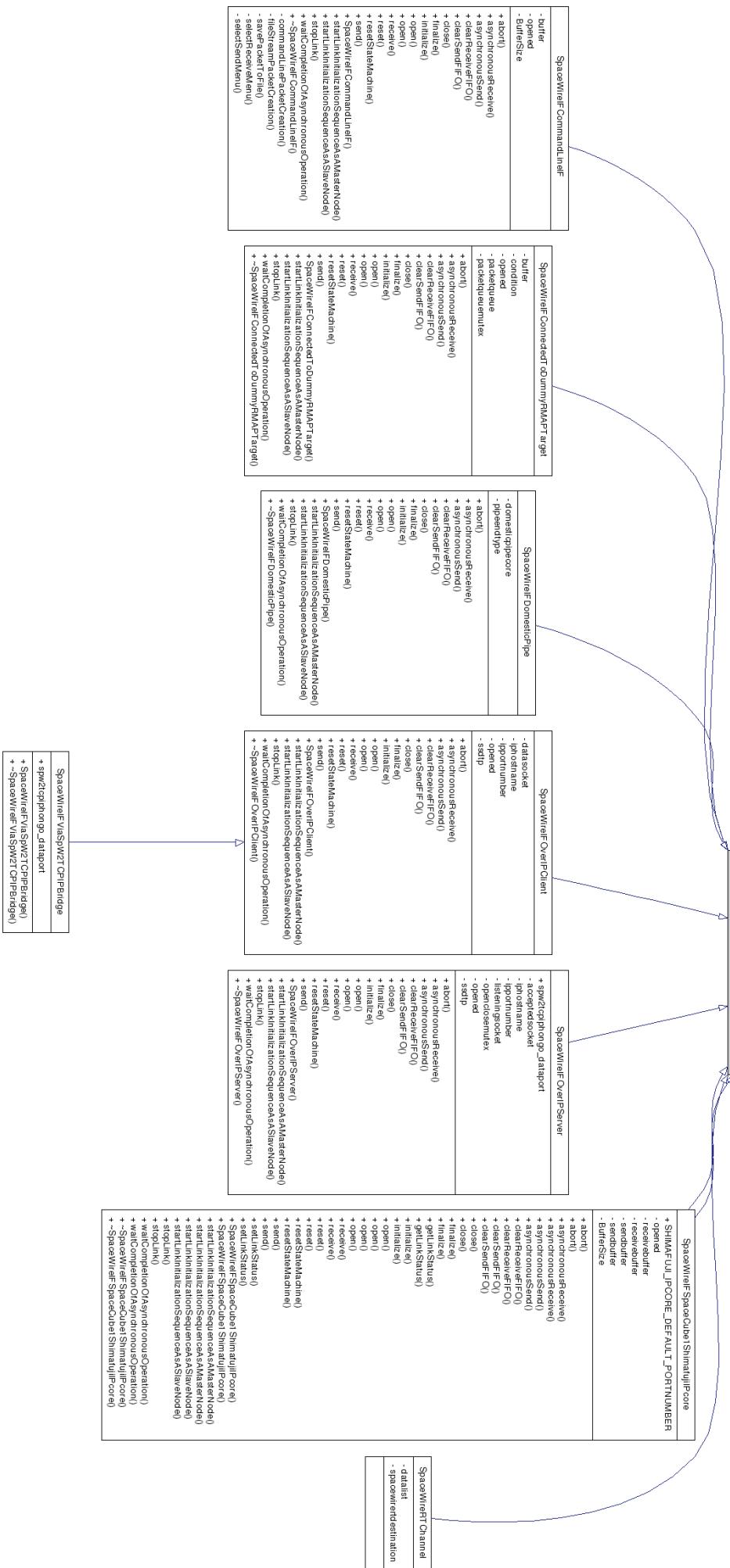
実際には、各SpaceWire I/F独自の関数をSpaceWireIFクラスという抽象クラス abstract classを継承したchild classとしてカプセル化 encapsulateし、コードの中でSpaceWireIFのインスタンスを作成する際に、使用するchild classを指定することで、I/Fを選択できるようになっています。これにより、使用するSpaceCubeコンピュータが、それまでと別のSpaceWire I/Fを搭載していても、インスタンス化するSpaceWireIFクラスの実装を変更するだけでよく、その他の部分は基本的に書き換えなくてもRMAPによるデータ転送などが行えるようになっています。

現状のソースコードで提供されているのは、NEC Soft製およびシマフジ電機製SpaceCube1用SpaceWire IPコアとドライバに対する実装で、それぞれSpaceWireIFSpaceCube1NECSoftIPcore.ccと、SpaceWireIFSpaceCube1ShimafujiiPcore.ccというクラス名になっています。また、デバッグ用として、SpaceWireIFConnectedToDummyRMAPTarget.ccやSpaceWireIFCommandLineIF.cc、SpaceWireIFOVERIPというクラスを提供します。これらはそれぞれ、「SpaceWireIFの先に、ダミーのRMAPターゲットがつながっている状況を模擬したSpaceWireIFクラス」、「SpaceWireIFでのパケット送受信をフックして、コマンドライン制御でユーザ自身がパケットを画面ダンプしたり、手でパケットを入力してプログラム側にReplyを返せる」、「SpaceWireのパケット転送を、TCP/IP経由でパイプする」ものです。SpaceCube1用の実装クラスは、ハードウェアやドライバ関数を直接扱う必要があるので、UNIXでは動作しませんが、その他のものはUNIXでも動作するため、SpaceCubeでの試験を行う前段階のデバッグ時などに利用できるのではないかと思います。



SpaceWireIF.ccのUML図

現段階でのchild classを並べておく。

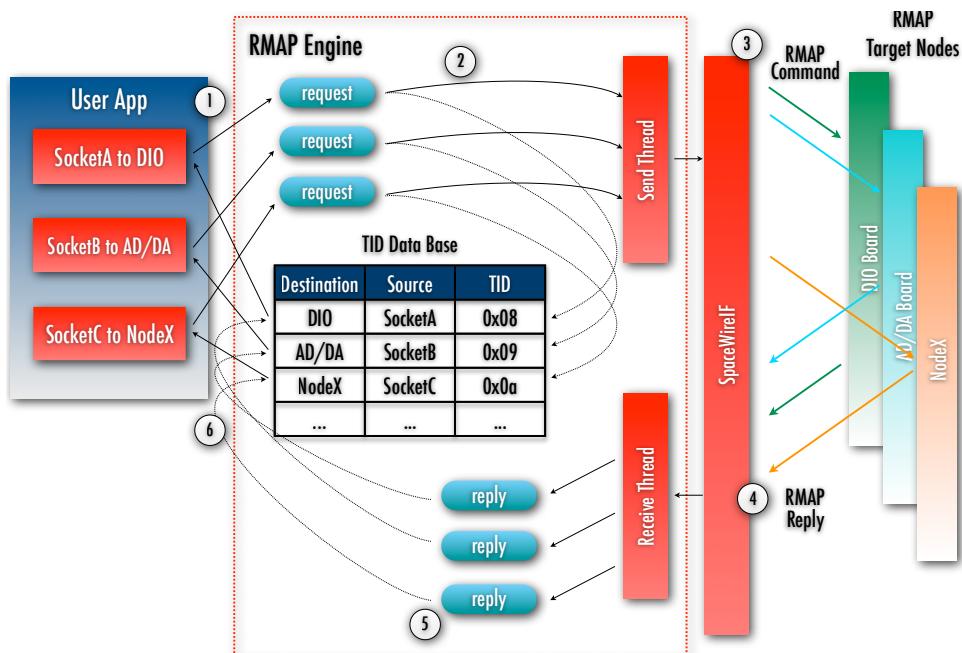


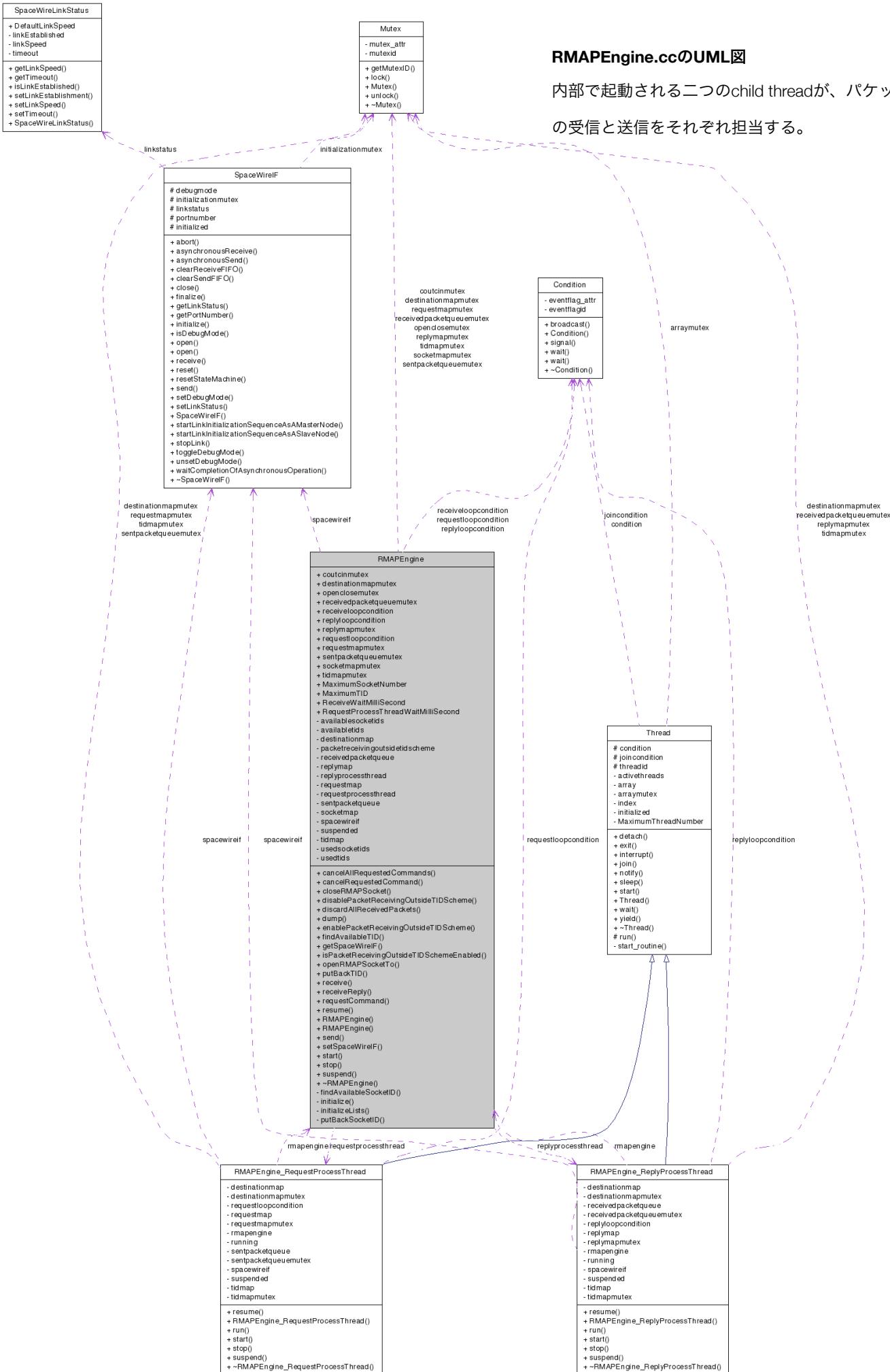
RMAPを実現する仕組み

RMAPアクセスを行う場合の手順は次のようにになります。ユーザは、SpaceWireデバイスをオープン(SpaceWireIF::open())し、RMAPの解釈系であるRMAPEngine classにそのSpaceWireIFを渡します。アクセス先のノードの情報(Logical/Path AddressやDestination Key)を設定したRMAPDestinationを作成し、RMAPEngineのopenSocketTo(RMAPDestination)メソッドを用いて、そのRMAPDestinationに対応するRMAPSocketを生成します。あとはそのRMAPSocketに対して、アクセスしたいアドレスや長さを与えてread()やwrite()を実行するだけで、ノードに対してデータの読み書きができます。

複数のスレッド thread (後述) から、複数のRMAPSocketをオープンすることもできます。その場合、複数のRMAPアクセスが同時に発生する可能性がありますが(図中の[1])、RMAPEngineが適当なキューイング queuingを行います([2])、順番に RMAP Command Packetの送信を行います([3])。そのときに、RMAPEngineは、送り出すRMAP Command Packetの Transaction IDを適当に設定し、そのTransaction IDと送り元のRMAPSocketのIDを内部のデータベース(C++ STLのmapクラス)に登録しておきます。RMAP Reply Packetの到着は、Commandの送信順とは無関係でかまいません ([4])。届いた RMAP Reply Packetから順に、Transaction IDに基づいて、そのデータベースを逆引きして、適切なRMAPSocketに届けます([5-6])。RMAPSocket側では、RMAPPacket classの機能を使って、そのReplyを解釈し(RMAPPacket::interpretPacket())、アクセスが成功したか調べます。成功していれば、Writeの場合はそのままユーザ処理に戻り、Readの場合は読み出したデータをユーザ処理に返します。失敗していれば例外をthrowします。RMAPEngineは内部で二つのスレッドを生成して上記の作業を並行動作させています。

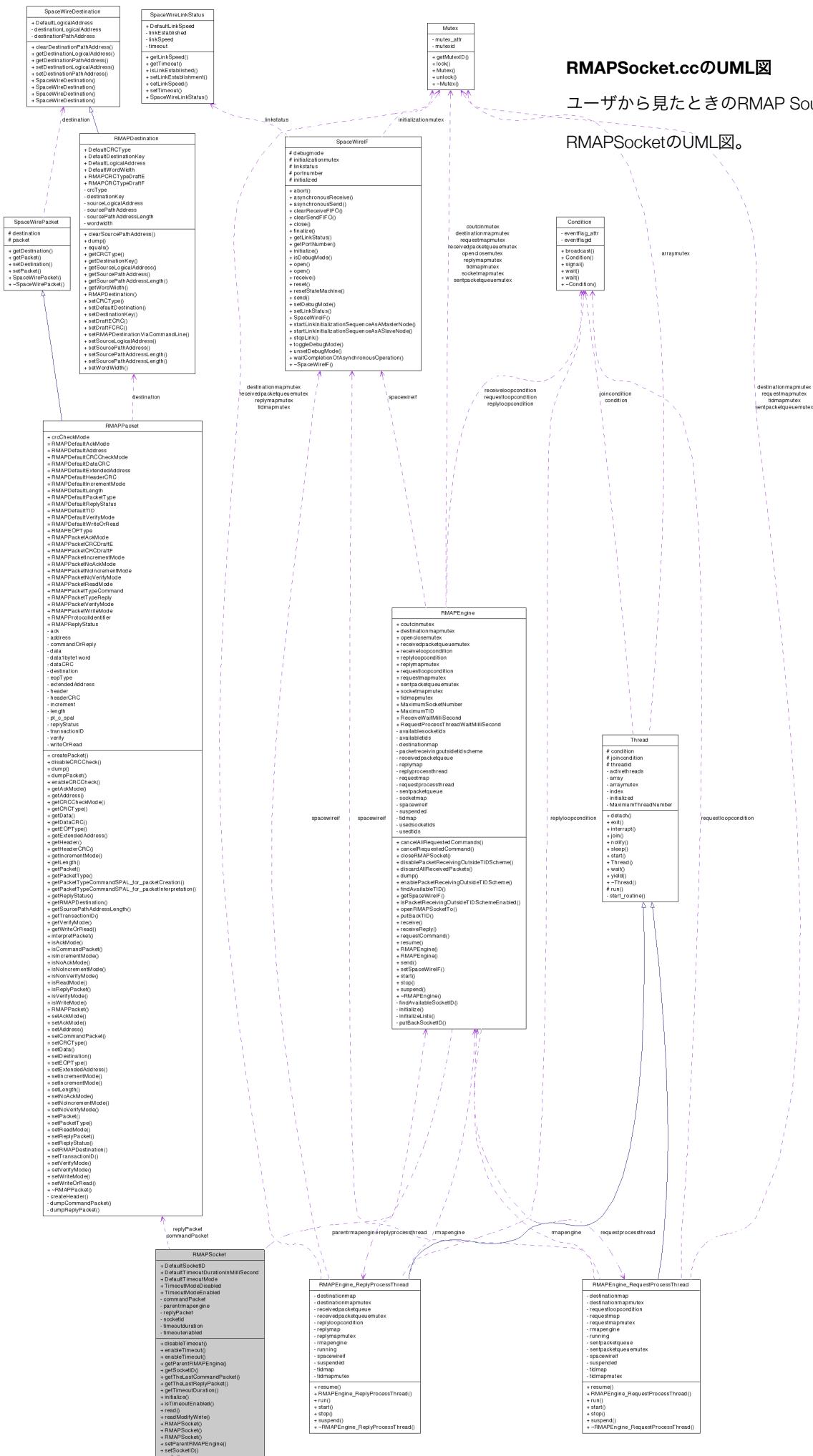
readやwriteの際に、データはC++標準ライブラリのvectorクラスに入れて受け渡しを行うので、C言語の「配列」のように、サイズやバッファオーバーランを気にする必要はありません(vectorは、データ量に応じてサイズが自動的に伸び縮みします)。また、(標準的なJavaの設計のように)あるメソッドに対する入力(in)は「引数」として、出力(out)は「返り値」として表しているので、C言語の関数のように、「どの引数がinなのかoutなのかわかりにくい」ということはありません。





RMAPEngine.ccのUML図

内部で起動される二つのchild threadが、パケットの受信と送信をそれぞれ担当する。



Thread(タスク)、Mutex、Conditionのカプセル化

マルチトランザクションのためには、本質的にマルチスレッドの並行動作が必要になります¹。ユーザプログラムがシングルスレッドであっても、少なくともパケットの送信と受信を行う部分は、並行して動いていいといけません。今回のライブラリでは、T-Kernelのタスク関連のシステムコールをあらわに記述せず、C++のクラスとしてカプセル化したThreadを用いているため、移植性の低い部分が非常に狭い範囲に限定できています。UNIX環境向けにも、標準的なスレッドシステムコールであるpthreadを、同様のインターフェースでカプセル化している²ので、T-Kernel用のマルチスレッドプログラムが、UNIXでも全く同じソースコードのまま、再コンパイルするだけで動作します。

マルチスレッドプログラミングでは、複数スレッドの処理がどの段階でプリエンプト/ディスパッチされるか予想できないので、同時処理が出来ない部分(static変数や、共有オブジェクトの操作)は、「クリティカルセクション」として、同時処理を禁止するための指示が必要になります。Javaには、synchronizeという便利な指示詞があり、メソッド名にこの指示詞を付けるだけで、複数スレッドが同時にアクセスしないクリティカルセクションを明示することができます。しかし、C/C++にはそれがないので、Mutex(mutual condition)やSemaphore、Condition(Event Flag)という機構を通じて、複数スレッド間での処理の調停を行います(これはT-KernelでもUNIXのpthreadでも共通の話です)。今回は、Threadだけでなく、同時に必要になるMutex、Conditionもクラスとしてカプセル化してあります(SemaphoreはMutexと本質的には同じものなので省略しましたが、必要があれば簡単に作成できます)。

とくに、T-Kernelにおけるマルチタスク処理と同期化について詳しく知りたい場合は、[a]を参考にしてください。

ドキュメンテーションについて

今回のライブラリでは、ソースコード内にDoxygenというドキュメントジェネレータに対応した形式のコメントを配置し、ライブラリに含まれるクラスやメソッドの一覧を、Doxygenで自動生成できるようになっています。生成されたHTML形式のドキュメントは、documents/html/index.htmlから参照することができます。ライブラリにの中に、どんなクラスがあって、どのようなメソッドを持つかは、このリファレンスを参照するとわかります。新しいクラスやメソッドを追加した場合は、その機能や使い方を示すコメントのある文法で記述してもらうと、次に利用する人にもわかりやすく、効率的に開発が行えると思います。

Doxygenの機能や、コメントの文法、ソフトウェアのダウンロードは、[B]を参考してください。

¹ 正確には、マルチスレッドでなくとも、複数トランザクションの同時実行は可能ですが、機能的により制限されたものになります。そのような限定的なマルチトランザクションは、これまでの(シングルスレッドで実装していた)ライブラリでも実現可能でした。

² Q：なぜ既存のC++用ポータブルスレッドライブラリ(ACEやboost)を使わないのか？ A：できることなら、ぜひそれらを使いたかったです。しかし、「T-Kernelのシステムコールに対応しているもののがなく、どれを使うにしても自分でゼロから中身を書き換えないといけなかったこと」と、「スケルトン自体が他のライブラリクラスに依存していて、スレッドに関係ないクラスまでポートしなければならず、作業量が膨大になってしまふこと」から、SynchronousThread(C++)やJavaなど、既存のライブラリを参考しながら、シンプルな実装を改めて作成しました。

インストールと コンパイルのしかた

(1) アーカイブの入手と展開

SpaceWire WikiやMLに流れるメールの添付ファイルとして入手できる、ライブラリ全体のアーカイブファイル(SpaceWireRMAPLibrary_200xMMDD_HHMM.zip)を展開します。SpaceWireRMAPLibraryフォルダが生成されるので、適当な場所(ふつうは、T-Kernelの開発を行っている/usr/local/teの中か、Eclipseのworkspaceディレクトリの中)にコピーしてください。

(2) 標準IPコア以外のSpaceWire IPコアを利用する場合

ユーザ会のJAXA/大学側標準SpaceWire IPコアである、SpaceCube1用のShimafuji製SpaceWire IPコアを利用する場合は、追加でインストールするものはありません。必要なドライバやライブラリは、driver/以下に入っています。

今回のSpaceWire/RMAP Libraryは、移植性を高めているので、Shimafuji SpaceWire IPコア以外にも、NEC Soft製IPコアでも動作します(実際に動作実績があります)。また、おそらくMHI製IPコアでも、ラッパークラスを適切に記述すれば、動作すると考えられます。

しかし、NEC Soft製やMHI製のSpaceWire IPを利用するためのドライバやライブラリは、ライセンス等の関係から集録されていません。基本的にはShimafuji製IPコアだけで、データ取得系のプログラムは開発できるはずですが、何らかの理由でNEC Soft製やMHI製のIPコアを利用する必要がある場合は、とりまとめ窓口となっているJAXAのスタッフ(国分さん、渡辺さんなど)に相談して入手してください。

それらの、標準以外のNEC Soft製IPコアや、MHI製IPコアを利用するときは、driver/以下に必要なドライバやライブラリを配置する必要がありますが、ディテールなのでここには記述しません。別ドキュメントで方法が説明されるかもしれません。

(3) 環境変数の設定

build/Makefile中で設定されているSPACEWIRERMAPLIBRARY_PATHという環境変数に、SpaceWire/RMAP Libraryのフォルダへの絶対パスを設定してください。ここは、開発者ごとにことなるはずなので、標準の記述では利用できず、必ず適切に設定する必要があります。

また、T-Kernel用の環境変数BDを/usr/local/te以外に設定している場合は、build/t-kernel/Makefileの中の「BD=/usr/local/te」の部分を、自分の環境の絶対パスに変更する必要があります。

(4) ビルドの実行

build/フォルダに移動して、make(gmake)を実行してください。makeが正しく実行できると、UNIX用とT-Kernel用のコンパイルが両方行われ、エラーがなければ、libraries/以下にライブラリのアーカイブが、executables/フォルダにサンプルとおまけソフトの実行形式が生成されるはずです。

もしも、POSIX環境用だけ、またはT-Kernel環境用だけのビルドを行いたい場合は、build/Makefile中の、ビルド対象となる環境の設定箇所を以下のように変更してください。

(変更前) all : t-kernel posix

(変更後 POSIX用だけの場合) all : posix

(変更後 T-Kernel用だけの場合) all : t-kernel

プログラムの実行

SpaceWire通信を利用するサンプルプログラムを実行するときは、前もってドライバなどのロードが必要になります。シマフジ電機製SpaceWire IPコアの場合は、drivers/spc1_shimafuji/spc_driver以下にある、spwというファイルがドライバの実行形式なので、それをSpaceCubeに転送し、コマンドラインで「lodspg spw」としてロードしてください。もう少し詳しい方法は、SpaceWire/SpaceCube Tutorial (参考文献[14])で解説されるか、他のドキュメントが用意されるかもしれません。

フォルダ構造について

このライブラリは、SpaceWireRMAPLibraryという名前のフォルダに入って配布されています。その中の各フォルダ、ファイルの構造は以下のようになっています。各フォルダ内のposix/とt-kernel/というフォルダはそれぞれ、UNIX用、T-Kernel用のソースや実行形式を入れるためのフォルダです。

- build
実行形式を生成するためのMakefileが置かれ、makeの実行中には中間ファイルなどが置かれるところ。T-Kernel用のMakefileは、SpaceCube1の開発環境にあわせて、いくつか外部設定ファイルを呼び出している。
- documents
ライブラリの更新履歴や、APIリファレンスが置かれている。
- driver
SpaceWire I/F用のドライバや、ライブラリを置くためのフォルダ。公開版では、シマフジ電機製IPコア用のフォルダが作ってある。「インストールとコンパイル」の項でも述べたように、他社製IPコアはライセンス上、配布物に含められないで、それらを利用する場合は、適切なフォルダ構造でここに配置する(別ドキュメントで説明される(?))。
- executables
UNIX、T-Kernel用の実行形式が生成される場所。
- libraries
各環境向けのライブラリのアーカイブが置かれる場所。
- source_Executables
実行形式用のmain関数を含むファイルをおく場所。このディレクトリの中に、exampleやおまけソフトのソースコードが入っている。main_test_xxx.ccというファイル群は、SpaceWireRMAPLibraryの開

発段階で、各要素を試験するために記述したテスト用ファイル。何かの参考になるかもしれないの
で、一応消さずに配布に含めてある。

- source_SpaceWireRMAPLibrary

このライブラリの構成クラス群を記述したファイルを置く場所。SpaceWire、RMAPという接頭辞がつ
いているものは、それぞれ、SpaceWireレイヤ、RMAPレイヤに属する機能を実現するためのクラスで
あるということ。

- source_ThreadLibrary

Thread、Mutex、Conditionをカプセル化したライブラリのソースコード置き場。UNIX用とT-Kernel用が
あり、コンパイルの際に、各Makefileから自動的に必要な方のファイルが呼び出されてコンパイルされ
る。

- userprograms

ユーザが、このSpaceWire/RMAP Libraryを使って、自分のプログラムを作るときのフォルダ構造や
Makefileのひな形。このフォルダを好きな場所にコピーして利用する(後述)。

ユーザプログラムの作成のしかた

SpaceWire/RMAP Libraryの機能を使った、検出器読み出しシステムのプログラムを自分で新たに作成するときは、serprogramsフォルダにあるuserprogram_templateというフォルダを適当な場所にコピーして、そのフォルダをEclipseのプロジェクトパッケージとしてImportし(「SpaceWire/SpaceCube Tutorial」に方法が書いてあります)、その中で開発を行ってください。ユーザが開発するフォルダは、どこに配置されても構いません(ライブラリのuserprogramsフォルダの中でなくとも構いません)。

サンプルとしてsourceフォルダの中に、main_UserProgram_Template.ccというファイルが置いてあります。この例の中では、Mainクラスのrun()メソッドにコードを追加していけば、その処理が実行されます。

新たに作成したプログラムを、コンパイル(make)の対象とするためには、build posixやbuild/t-kernel フォルダの中のMakefileを開いて、下記のように「実行形式のソース」のリストに、新しいファイルの名前(例ではmain_UserProgram_Template.cc)を追加します。これでmakeを実行すると、追加したファイルも自動的にコンパイルされ、executablesフォルダに実行形式が生成されます。以下の例は、main_Detector_Configuration.ccという実行形式を追加するときの例です(Makefileの文法のため、行と行の連結にはバックスラッシュ(もしくは円マーク)を用いています)。

```
(build/t-kernel/Makefile または、 build posix/Makefile)

#####
# User Program Sources
#####
USERPROGRAM_SOURCE = \
main_UserProgram_Template.cc \
main_Detector_Configuration.cc
```

ライブラリの使用例

ここでは、ライブラリの使用方法を、サンプルプログラムで示します。

例題0： main_example00.cc シンプルなRMAPアクセス

以下のソースは、SpaceCube1で、RMAPEngine/RMAPSocketを使って、シマフジ電機製SpaceWire DIOボードのSDRAM の0x0000_0000番地から16バイト分の領域に、(1)データを書いて、(2)その領域を読み出す、という試験プログラムです。SDRAMは、RAMという名前のとおり、書き込まれたデータを覚えているので、正しく動作すれば、書き込んだのと同じ値が読み出されるはずです。

注意：ここでは、ボードのロジカルアドレスが0xFEに、RMAPのCRC(チェックサムみたいなもの)のバージョンがDraft F バージョンに、Destination Keyが0x02に設定されているDIOボード(最新のIPコアではそうなっている、はず)を対象に想定しています。なっていない場合は、ソースコードのRMAPDestinationの情報設定部分のパラメータを、使おうとしているボードのパラメータに合わせて修正するか、ボードに書き込まれているロジックを焼き変える必要があります。ロジックの焼き変えに関しては、スタッフやSpaceWire MLなどに相談してください。

```
00001 #include "SpaceWire.hh"
00002 #include "RMAP.hh"
00003
00004 #include <iostream>
00005 using namespace std;
00006
00007 class Main {
00008 public:
00009     Main() {
00010         //Initialization
00011     }
00012 public:
00013     void run();                                SpaceWireIFのインスタンス化
00014     SpaceWireIF* spacewireif=
00015         SpaceWireIFImplementations::selectInstanceFromCommandLineMenu();
00016     spacewireif->initialize();
00017     spacewireif->open();
00018
00019     //create an instance of RMAPEngine          RMAPEngineのインスタンス化
00020     RMAPEngine* rmapengine=new RMAPEngine(spacewireif);
00021     rmapengine->start();                        RMAPDestinationのインスタンス化
00022
00023     //create an instance of RMAPDestination
00024     //set properties for Shimafuji SpaceWire DIO Board
00025     RMAPDestination rmapdestination;
00026     rmapdestination.setDefaultDestination();
00027     rmapdestination.setDestinationLogicalAddress(0xFE);
00028     rmapdestination.setSourceLogicalAddress(0xFE); // J106 Connector
00029     rmapdestination.setDestinationKey(0x02);
00030     rmapdestination.setDraftFCRC();
00031     vector<unsigned char> path;
00032     path.clear();
00033     rmapdestination.setDestinationPathAddress(path);
00034     rmapdestination.setSourcePathAddress(path);
00035     rmapdestination.setWordWidth(1);
```

```

00035
00036 //open RMAPSocket to "rmapdestination"
00037     RMAPSocket* rmapsocket=rmapengine->openRMAPSocketTo(rmapdestination);
00038
00039 //prepare data vector for write access
00040     unsigned int writelength=16;
00041     vector<unsigned char> writedata;
00042     for(unsigned int i=0;i<writelength;i++){           Writeするデータの準備
00043         writedata.push_back((unsigned char)i);
00044     }
00045
00046 //execute RMAP Write to Shimafuji DIO Board's SDRAM
00047     unsigned int writeaddress=0x00000000; //SDRAM Address
00048     try{
00049         cout << "Writing...";                                (1) RMAP Writeの実行
00050         rmapsocket->write(writeaddress,&writedata);
00051         cout << "Done" << endl;
00052     }catch(RMAPException e){
00053         //if an exception occurs
00054         cout << "Exception in Write Access" << endl;
00055         //dump the exception
00056         e.dump();
00057     }
00058
00059 //execute RMAP Read to Shimafuji DIO Board's SDRAM
00060     unsigned int readaddress=0x00000000;                  (2) RMAP Readの実行
00061     unsigned int readlength=16;                           読み出し結果はvector*として返ってくる
00062     vector<unsigned char>* readdata;
00063     try{
00064         cout << "Reading...";
00065         readdata=rmapsocket->read(readaddress,readlength);
00066         cout << "Done" << endl;
00067     }catch(RMAPException e){
00068         //if an exception occurs
00069         cout << "Exception in Read Access" << endl;
00070         //dump the exception
00071         e.dump();
00072     }
00073
00074 //dump the result
00075     cout << "Written Data : " << endl;
00076     SpaceWireUtilities::dumpPacket(&writedata);
00077     cout << "Read Data : " << endl;
00078     SpaceWireUtilities::dumpPacket(readdata);
00079
00080 //finalization
00081     rmapengine->closeRMAPSocket(rmapsocket);
00082     rmapengine->stop();
00083     spacewireif->close();
00084     delete rmapengine;
00085     delete spacewireif;
00086 }
00087 };
00088
00089 int main(int argc,char* argv[]){
00090     Main main;
00091     main.run();
00092     return 0;
00093 }
```

SpaceCubeに転送して実行してみると以下のようになります。起動直後に、使用しているIPコアを選択するよう求められます。Writing...とReading...の間のところで、実際のSpaceWire通信が行われています。通信が完了すると、書き込んだはずのデータと、RMAP経由で読み込んだデータを表示します。

通信が終わっても、spwAbortFnと表示されたままでプログラムが終了していないように見える(コマンドラインに戻れない)のは、バグではなくて現状の仕様です。電源を切って再起動してください。これは、SpaceWire IPコアのドライバ、SpaceWireIF、およびRMAPEngineといったクラスの終了処理をちゃんと記述していないことが原因です。詳細は「問題とコメント」のセクションや、SpaceWire MLでの議論を参照してください(原因も対策もわかっていますが、湯浅は時間の関係で作業できていません。自前で修正した方がおられましたらフィードバックしてください)。

```
[/SYS]% fget himawari:/Users/yuasa/Desktop/SpaceWireRMALibrary/build/t-kernel/main_example00

[/SYS]% main_example00
Which SpaceWireIF implementation:
  Shimafuji IP Core      [2]
  Shimafuji Router IP Core [3]
select > 2
SpaceWireIFSpaceCube1ShimafujiIPcore : initialize
spw0 opened
fpga ver. = 0x1000
SpaceWireIFSpaceCube1ShimafujiIPcore : open (0)
connecting ...
link0 on (0x0000003f)
status = 0x0000003f
connected.
SpaceWireIFSpaceCube1ShimafujiIPcore : connect (0)
Writing...Done
Reading...Done
Written Data :
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
Read Data :
0x00 0x00 0x01 0x02 0x03 0x04 0x05 0x06
0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e
SpaceWireIFSpaceCube1ShimafujiIPcore : close (0)
spwAbortFn
```

例題1： main_example01.cc マルチスレッドのプログラム

シンプルなマルチスレッドプログラムの例を示します。まずは、簡単のために、SpaceWire/RMAP通信は行わず、二つのスレッドで交互にメッセージを表示する、という例をみてみます。「cout」を使って画面に文字を出力している間は、他のスレッドが同時に画面に出力して文字が化けてしまわないように、Mutexクラスを用いて、排他制御を行っています(排他制御を行わないと、実際には文字化けではなく、文字が互い違いに表示される場合が生じます)。

```
00001 #include "Thread.hh"
00002 #include "Mutex.hh"
00003 #include "Condition.hh"
00004
00005 #include <iostream>
00006 using namespace std;
00007
00008 class ChildClassOfThread : public Thread { Thread classを継承したchild classを宣言
00009 private:
00010     int id;
00011     Condition* condition;
00012     Mutex* mutex;
00013 public:
00014     ChildClassOfThread(int newid,Condition* newcondition,Mutex* newmutex)
00015         : Thread(),id(newid),condition(newcondition),mutex(newmutex){}
00016 public:
00017     virtual void run(){ 1秒ごとにメッセージを表示(5回)
00018         for(unsigned int i=0;i<5;i++){
00019             mutex->lock();
00020             cout << "ChildClassOfThread : " << id << endl;
00021             mutex->unlock();
00022             this->sleep(1000);
00023         }
00024
00025         //wait until "signalled"
00026         mutex->lock();
00027         cout << "ChildClassOfThread : " << id << " waiting for signal" << endl;
00028         mutex->unlock(); メインスレッドからsignalされるまでwait
00029         condition->wait();
00030
00031         //show completion message
00032         mutex->lock();
00033         cout << "ChildClassOfThread : " << id << " signalled" << endl;
00034         mutex->unlock(); signal()受信したら、スレッド終了
00035
00036         this->exit();
00037     }
00038 };
00039
00040 class Main {
00041 private:
00042     Condition condition;
00043     Mutex mutex;
00044 public:
00045     Main(){
00046         //Initialization
00047     }
00048     void run(){
00049         //Instantiate ChildClassOfThread class 各スレッドクラスのインスタンス化
00050         ChildClassOfThread thread1(1,&condition,&mutex);
00051         ChildClassOfThread thread2(2,&condition,&mutex);
00052
00053         //start threads
00054         mutex.lock(); スレッドを開始
00055         int status1=thread1.start();
00056         int status2=thread2.start();
```

```

00057         cout << "thread1 started : " << status1 << endl;
00058         cout << "thread2 started : " << status2 << endl;
00059         mutex.unlock();
00060
00061         //wait 10sec                                10秒間、単なる待ちに入って、、、
00062         condition.wait(10000);
00063
00064         //signalling                               10秒間経ったら、wait()で待っている他のスレッドに
00065         condition.broadcast();                      一斉にsignal()を送る
00066
00067         //wait 1sec more                            さらに1秒待って、、、
00068         condition.wait(1000);
00069
00070         //then, quit
00071         mutex.lock();
00072         cout << "Main completed" << endl;
00073         mutex.unlock();
00074     }
00075 }
00076
00077 int main(int argc,char* argv[]){
00078     Main main;
00079     main.run();
00080     return 0;
00081 }
```

このサンプルは、SpaceWireのハードウェアを利用していませんので、UNIXでもT-Kernelでも動作します(UNIX版の実行形式は、executables posix フォルダに入っています)。例えばSpaceCube1で実行したときの様子を掲載しておきます。

```
(example00と同様に、FTPなどでSpaceCube1に転送し、コマンドラインから実行)
[/SYS]% main_example01
thread1 started : 66    <== スレッド番号(実際には、T-Kernelから与えられたタスクID)
thread2 started : 67    <== 同上
ChildClassOfThread : 1  <== 1秒ごとに、各スレッドインスタンスがメッセージを表示
ChildClassOfThread : 2
ChildClassOfThread : 1
ChildClassOfThread : 2
ChildClassOfThread : 1 waiting for signal <== 5秒くらいでこここの状態になり、、、
ChildClassOfThread : 2 waiting for signal
ChildClassOfThread : 1 signalled           <== さらに5秒くらいの間隔の後、この表示が出て、
ChildClassOfThread : 2 signalled
Main completed          <== さらに1秒くらいの間隔の後、この表示が出て実行終了
[/SYS]%
```

例題2：main_example02.cc マルチプルRMAPトランザクションの例

この例では、二つのSpaceCubeを向かい合わせにSpaceWireケーブルで接続して、両方で実行します。なので、SpaceCubeが2セット、シリアルケーブルが2セット必要になります。

実行時メニューに表示されるように、2台のうち、一方のSpaceCubeはこの試験における”RMAP Source”として動作し、もう一方は”RMAP Destination”を模擬します。Source SpaceCubeは、複数のスレッドで複数のRMAP Socketをオープンし、それぞれのスレッドから、Destination SpaceCubeへ向けて同時に複数の(この試験では8個)のRMAP Command Packetを送信します。Destination側では、8個のRMAP Commandを受信すると、届いたパケットのTransaction IDを表示し、ユーザに「どのTransaction IDに対して、RMAP Replyを送るか」の入力を求めるモードに画面が切り替わります。ユーザは、適当な順番でTransaction IDを選択し、入力ていきます。すると、その順番で順次Replyが送出され、Source 側に届きます。Source側では、返ってきたTIDの順に処理が行われ、Replyを受信したスレッドから順に、Replyの中身を表示して終了していき、8個のトランザクション全てが終了すると、プログラムは終了します。

ソースコードはかなり長いので、ここには掲載しませんが、source_Executablesに入っているのでご覧ください。基本的には、example00の、RMAPSocketのオープンを複数のスレッドで行う、というソースです。ここでは、2台のSpaceCubeそれぞれでの実行画面を示します。

(Source側のSpaceCube)

```
L:\>javaw -Djava.library.path=.\lib -jar SpaceWireIFSpaceCube1NECSpaceCore.jar  
SpaceWireIFSpaceCube1NECSpaceCore : initialize  
SpaceWireIFSpaceCube1NECSpaceCore : port 0  
initialize (successful)  
SpaceWireIFSpaceCube1NECSpaceCore : receive  
port registering...done
```

Multiple RMAP Transaction Test Program

MT Test (Source Mode) [1]
MT Test (Destination Mode) [2]
Quit [9]

Multiple Transaction Test (Source Mode)

Thread id:0 Read (address=00000000 length=01) activethreads=1 8個のスレッドぞ
Thread id:1 Read (address=00000001 length=02) activethreads=2
Thread id:2 Read (address=00000002 length=03) activethreads=3
Thread id:3 Read (address=00000003 length=04) activethreads=4
Thread id:4 Read (address=00000004 length=05) activethreads=5
Thread id:5 Read (address=00000005 length=06) activethreads=6
Thread id:6 Read (address=00000006 length=07) activethreads=7
Thread id:7 Read (address=00000007 length=08) activethreads=8

The number of active SingleTransaction instances : 8 8個のスレッドぞ
The number of active SingleTransaction instances : 8 8個のスレッドぞ
The number of active SingleTransaction instances : 8 Reply行
The number of active SingleTransaction instances : 8
The number of active SingleTransaction instances : 8
The number of active SingleTransaction instances : 8
The number of active SingleTransaction instances : 7
The number of active SingleTransaction instances : 7
The number of active SingleTransaction instances : 7
The number of active SingleTransaction instances : 6
The number of active SingleTransaction instances : 5
The number of active SingleTransaction instances : 5

(Destination側SpaceCubeの画面)

```
SpaceWireIFSpaceCube1NECSoftIPcore : initialize  
SpaceWireIFSpaceCube1NECSoftIPCore : port 0  
initialize (successful)  
SpaceWireIFSpaceCube1NECSoftIPcore : receive  
port registering...done
```

MT Test (Source Mode) [1]
MT Test (Destination Mode) [2]
Quit [9]

	Multiple Transaction Test (Destination Mode)
Waiting a packet (0/8received)...received (TID=0)	8個のRMAP Commandを受信
Waiting a packet (1/8received)...received (TID=1)	
Waiting a packet (2/8received)...received (TID=2)	
Waiting a packet (3/8received)...received (TID=3)	
Waiting a packet (4/8received)...received (TID=4)	
Waiting a packet (5/8received)...received (TID=5)	
Waiting a packet (6/8received)...received (TID=6)	
Waiting a packet (7/8received)...received (TID=7)	
Start Replying	
Remaining TIDs : 0 1 2 3 4 5 6 7	Transaction ID 4のRMAP Replyを送ると選択
select tid to be replied> 4	
replying...done	
Remaining TIDs : 0 1 2 3 5 6 7	Transaction ID 2のRMAP Replyを送ると選択
select tid to be replied> 2	
replying...done	
Remaining TIDs : 0 1 3 5 6 7	
select tid to be replied> 3	
replying...done	
Remaining TIDs : 0 1 5 6 7	
select tid to be replied> 0	
	残りのTransaction IDについて繰り返し

```

The number of active SingleTransaction instances : 5
The number of active SingleTransaction instances : 5
The number of active SingleTransaction instances : 5
Thread id:0 Completed (replied data length=01)
received data : 0x00
The number of active SingleTransaction instances : 4
Thread id:1 Completed (replied data length=02)
received data : 0x00 0x01
Thread id:5 Completed (replied data length=06)
received data : 0x00 0x01 0x02 0x03 0x04 0x05
Thread id:6 Completed (replied data length=07)
received data : 0x00 0x01 0x02 0x03 0x04 0x05 0x06
The number of active SingleTransaction instances : 1
Thread id:7 Completed (replied data length=08)
received data : 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
The number of active SingleTransaction instances : 0
Completed

```

8つのトランザクションがすべてに
完了したので、プログラム終了

```

spw_ReadLink ABORT
SpaceWireIFSpaceCube1NECSoftIPcore : port 0 close
[/SYS]%

```

replying...done

```

Remaining TIDs : 1 5 6 7
select tid to be replied> 1
replying...done

```

Remaining TIDs : 5 6 7

```

select tid to be replied> 5
replying...done

```

Remaining TIDs : 6 7

```

select tid to be replied> 6
replying...done

```

Remaining TIDs : 7

```

select tid to be replied> 7
replying...done

```

Remaining TIDs : 7

```

select tid to be replied> 7
replying...done

```

8つのトランザクションすべてに
RMAP Replys送信完了

```

Reply Completed
spw_ReadLink ABORT
SpaceWireIFSpaceCube1NECSoftIPcore : port 0 close
[/SYS]%

```

例題3：main_example03.cc SpaceWireパケットの送受信 (RMAPレイヤを使わない場合)

このライブラリでは、RMAPパケットの生成・解釈はRMAPPacket classだけで可能です。従って、RMAPPacket classで生成したパケット列を、そのままSpaceWireIF classで送信すれば、RMAPEngine/RMAPSocketを使わなくても、RMAP通信を行うことができます。これは、お勧めの方法ではありませんが、後述のように、RMAPEngineによるオーバーヘッドが減少するため、転送速度が若干上昇します。また、試験などのときに、わざと間違ったRMAPパケットを送信したい場合などにも使えるかもしれません。このサンプルのソースコードは、example00ととても近いので、とくに掲載はしません。

RMAPDestinationの設定、RMAPPacketの生成(createPacket())、パケット送信(send())、パケット受信(receive())、受信パケット解釈、成功失敗判定などを、RMAPEngine/RMAPSocketまかせではなく、全部手動で行うことになります。

例題4 : main_example04.cc ルータの使用

この例題では、NEC製のSpaceWire Routerの内部レジスタにアクセスしてみます。バスアドレスやロジカルアドレスを用いた例題を作成してくれる人がいたら、フィードバックをお願いします(2008年5月現在、シマフジ電機製ボードがロジカルアドレッシングのみ対応で、しかもボードのロジカルアドレスが0x01に設定されているということから、わかりやすい例題作成が簡単ではありません...)。NEC製SpaceWire Routerの内部レジスタは4バイト1ワードで管理されており、アクセスの単位は4バイトごとになっています。ここで、注意が必要なのは、「RMAPでは、データを表現するときのバイトオーダはビッグエンディアンなので、複数バイトを1ワードとしてアクセスするさいに、データの並び順を意識する必要がある」、ということです。しかし、今回のSpaceWire/RMAP Libraryの中で、RMAPパケットを表現するRMAPPacket classは、バイトオーダの問題を自動的に処理してくれるので、ユーザは難しいことを気にする必要がありません。具体的には、Routerボードの内部レジスタ(という行き先)を表現するRMAPDestinationの、1ワードの幅(バイト数)プロパティに「4バイト」を設定して、RMAPPacket classにそのRMAPDestinationを渡します。この情報を用いて、RMAPPacket classは、データの並び順を適切に変更して実際のパケット生成、送受信、解釈を行います。このおかげで、ユーザがRMAPPacketのsetData()/getData()でやり取りするときのvector<unsigned char>の中身は、「1バイト1ワード」という形式に統一されています。

例題では、SpaceWire Routerボードの内部レジスタとして格納されている、「Routerボード自身のLogical Address」レジスタと、「各ポート(port 1~14)が受信したパケットの数」レジスタを読み出します。SpaceWire Routerボードの内部レジスタのアドレスや、レジスタ構成などについては、[13]の資料を参照してください。この資料の入手方法に関しては、JAXA/大学のSpaceWire開発メンバーは、とりまとめをしてくださっている、JAXA/ISAS 国分さんにコンタクトをお願いします。ソースコードはmain_example00.ccとほとんど同じなので、重要な部分だけ抜粋して掲載します。実行結果は以下のようになります。

```
00022 //create an instance of RMAPDestination for SpaceWire Router Board by NEC
00023 RMAPDestination router_configurationregister_rmapdestination;
00024     router_configurationregister_rmapdestination.
00025         setDestinationLogicalAddress(0xFE);
00026     router_configurationregister_rmapdestination.setDestinationKey(0x00);           ワード幅を指定。SpaceWire Routerでは4バイト1ワード
00027     router_configurationregister_rmapdestination.setWordWidth(4); //4bytes-1word
00028
00029     vector<unsigned char> destinationpathaddress;
00030     destinationpathaddress.push_back(0x00);
00031         //router's internal "configuration register" is connected to port 0
00032     router_configurationregister_rmapdestination.
00033         setDestinationPathAddress(destinationpathaddress);
configuration registerは仮想的にルータ内の0番ポートにつながっているように見えるので、アクセス先のバスアドレスとして0x00を指定。この場合は、Source Path Addressは指定しなくてよい、というのがRMAPでの仕様。
```

```
[/SYS]% main_example04
SpaceWireIFSpaceCube1NECSoftIPcore : initialize <== 初期化手続き
SpaceWireIFSpaceCube1NECSoftIPcore : port 0 initialize (successful) <== ...
SpaceWireIFSpaceCube1NECSoftIPcore : receive port registering...done <== ...
Reading Logical Address register...Done <== 初期化手続きここまで

Logical Address Register : 0x000000fe <== Logical Addressは0xFEに設定されて
Reading Receive Packet Counter register (port 1)...Done いますよ、 ということ。
Received Packet Counter Register (port 1) : 0x00000002 <== Port 1は2個のパケットを受信しましたよ、
Reading Receive Packet Counter register (port 2)...Done ということ。
...中略...
Received Packet Counter Register (port 13) : 0x00000000 <== Port 13までread完了
spw_ReadLink ABORT
SpaceWireIFSpaceCube1NECSoftIPcore : port 0 close <== 終了手続き
[/SYS]%
```

各クラスの使い方の詳細

SpaceWireIF

SpaceWireIFのハードウェアやドライバを仮想化するクラスです。RMAPを中心に使う場合には、ユーザはこのクラスを直接使用することはありません。ソースコードのなかでは、実際に使用するSpaceWireIFクラスの(child class)インスタンスを、RMAPEngineに渡すだけです。自分でこのクラスを使いたい場合の、初期化、オープン、パケット送受信、クローズの手順は以下のようになります。

```
SpaceWireIF* spacewireif=new SpaceWireIFSpaceCube1NECSoftIPcore(); <== インスタンス化
spacewireif->initialize(); <== 初期化
spacewireif->open(); <== デフォルトポートをオープン(複数のSpW I/Fがあるときは、引数でI/F番号を指定)

try{
    spacewireif->send(vector<unsigned char>* sentpacketdata); <== 1パケット送信
    vector<unsigned char>* receivedpacketdata=spacewireif->receive(); <== 1パケット受信
}catch(SpaceWireException e){
    //例外処理
}

spacewireif->close(); <== クローズ
delete spacewireif; <== インスタンス消去
```

また、ある環境で利用できるSpaceWireIF実装クラスの一覧を表示して、ユーザにどれを使用するか選んでもらうためのおたすけクラスも用意されていて、例題のコードでそうなっているように、 SpaceWireIF Implementations :: selectInstanceFromCommandLineMenu()で利用できます。返り値は、SpaceWireIF*です。

実際のSpaceWire I/Fでは、たとえば相手のノードのリンクがダウンしたときや、SpaceWireケーブルが抜けたときなど、パケットの送受信ができなくなる状況が発生します。そのような場合に、SpaceWireIF::receive()やsend()メソッドが無限の待ちに入ったまま、ユーザの処理が止まってしまうようなことがないように、SpaceWireIF::receive()やsend()は、タイムアウト例外をthrowする場合があります。普通のプログラミングでは、 おもにRMAPEngineを使うのでSpaceWireIFクラスを直接扱うことはないかもしれません、直接それらのメソッドをコールするときは、try～catch文で、それらの例外をハンドルしてあげる必要があります。

RMAPEngine

複数のRMAPアクセスを並列動作させるためのクラスです。このクラスをインスタンス化すると、ユーザのメインレッドとは別に、二つのデーモンスレッドが生成され、それぞれRMAPパケットの送受信を担当します。ユーザは、RMAPEngineのopenRMAPSocketTo(RMAPDestination)メソッドを用いて、あるRMAPノードへのソケットを生成し、そのソケットが持つread()/write()メソッドを使って、実際のRMAP通信を行います。RMAPEngineの起動、終了の例はRMAPSocketのところで一緒に示します。

RMAPSocket

RMAPEngine/SpaceWireIFを経由して、仮想的にリモートのRMAPノードに接続されたようなソケットを表現します。RMAPパケットの生成(実際にはプロパティの設定)や解釈を行い、RMAPレイヤの作業がユーザに見えないようにカプセル化します。RMAPSocketは、RMAPEngineのopenRMAPSocketTo()メソッドからのみ生成され、ユーザが自分でnew RMAPSocket()などとすることは、普通はありません。また、RMAPEngineが生成したRMAPSocketインスタンスのdeleteも、RMAPEngine側で管理しながら行うので、あるソケットを閉じたいとき(RMAP通信が完了したとき)は、RMAPEngineのcloseSocket(RMAPSocket*)メソッドを呼んでください。この内でインスタンスのdeleteまで行ってくれます。以下、RMAPEngineとRMAPSocketの使い方の順序を示します。

(先に、上の例を参考にして、SpaceWireIFのインスタンスを作成し、オープンしておく)

```
RMAPEngine* rmapengine=new RMAPEngine(spacewireif);    <== 使用するSpaceWireIFを与えてインスタンス化  
rmapengine->start();                                     <== 受信/送信用スレッド開始(バックグラウンドで動作)  
  
RMAPSocket* rmapsocket=rmapengine->openRMAPSocketTo(rmapdestination);  
                                         <== あるRMAPDestinationにつながったソケットを生成  
  
//RMAP通信を伴う処理  
unsigned int address=0x0000abcd;  
unsigned int length=16;  
vector<unsigned char> readdata=rmapsocket->read(address,length);  <== 例：RMAP Read実行  
  
rmapengine->closeRMAPSocket(rmapsocket);                <== RMAP通信の処理が完了したら、RMAPSocketクローズ  
rmapengine->stop();                                     <== RMAPEngineを停止(二つのデーモンスレッド終了)  
delete rmapengine;  
  
(SpaceWireIFの終了処理)
```

RMAPPacket

このクラスは、RMAPSocketとRMAPEngineの内部で使われるため、普通のユーザプログラムではユーザが使う必要はありません。しかし、機能的に多くのことが可能なので、SpaceWireIFボードのロジックのデバッグ作業の時などに役立つかもしれません。

このクラスは、以下のようなRMAPアクセスに関連した諸情報(プロパティ)から、RMAPパケットを生成する、あるいは、あるデータ列をRMAPパケットだと思って解釈し、それらの情報を読み取る、といった機能を持っています。これらのプロパティは、アクセサメソッド accessorsを通してset/getができます。パケットの生成はcreatePacket()、パケットの解釈はsetPacket()/interpretPacket()で行います。アクセサメソッド一覧は、HTML形式のAPIリファレンスを参照してください。

また、パケットの解釈時に、CRCのチェックを行うかどうかが、enableCRCCheck()/disableCRCCheck()といったメソッドで変更できます。CRCチェックには、O(N)の処理コストがかかるので、巨大なパケットを送受信し合う場合で、パケット化けの心配があまりないような環境では、disableにした方が転送速度があがります。

RMAPPacketが管理する情報の例：パケットの種類(コマンド/リプライ)、コマンド(Write/Read)、このパケットの目的地(RMAPDestination)、データ、アクセス先のアドレス、アクセスの長さ(バイト数)、RMAP Transaction ID、CRC(バリティ)、…

これらの情報のデフォルト値は、RMAPPacket.hhでstatic constな変数として定義されており、それぞれ以下のようになっています。

```
RMAPPacket.hhの抜粋  
(各種プロパティのデフォルト値)  
static const unsigned char RMAPProtocolIdentifier=0x01;  
static const unsigned char RMAPDefaultPacketType=0x01;  
static const unsigned char RMAPDefaultWriteOrRead=0x01;  
static const unsigned char RMAPDefaultVerifyMode=0x01;  
static const unsigned char RMAPDefaultAckMode=0x01;  
static const unsigned char RMAPDefaultIncrementMode=0x01;  
static const unsigned int RMAPDefaultTID=0x00;  
static const unsigned char RMAPDefaultExtendedAddress=0x00;  
static const unsigned int RMAPDefaultAddress=0x00;  
static const unsigned int RMAPDefaultLength=0x00;  
static const unsigned char RMAPDefaultHeaderCRC=0x00;  
static const unsigned char RMAPDefaultDataCRC=0x00;  
static const unsigned char RMAPDefaultReplyStatus=0x00;  
  
(以下、各種情報を表す数字をハードコードしないための定数変数)  
static const unsigned char RMAPPacketTypeCommand=0x01;  
static const unsigned char RMAPPacketTypeReply=0x00;  
  
static const unsigned char RMAPPacketWriteMode=0x01;  
static const unsigned char RMAPPacketReadMode=0x00;  
  
static const unsigned char RMAPPacketVerifyMode=0x01;  
static const unsigned char RMAPPacketNoVerifyMode=0x00;  
  
static const unsigned char RMAPPacketAckMode=0x01;  
static const unsigned char RMAPPacketNoAckMode=0x00;  
  
static const unsigned char RMAPPacketIncrementMode=0x01;  
static const unsigned char RMAPPacketNoIncrementMode=0x00;  
  
static const unsigned char RMAPPacketCRCDraftE=0x00;  
static const unsigned char RMAPPacketCRCDraftF=0x01;
```

RMAPDestination

SpaceWireネットワークの中の、あるRMAPノードから目的地のRMAPノードへの行き先を表現するためのクラスです。ユーザは以下の情報を設定することで、自分のRMAPアクセスが、どこ宛てなのかをRMAPPacketやRMAPSocket、RMAPEngineに伝えるためのインスタンスを作成できます。RMAPPacket class同様、各情報はアクセサメソッドでset/getできます。

RMAPDestinationが管理する情報：Destinationのアドレス(Path/Logical)、Sourceのアドレス(Path/Logical)、SourceのPath Addressの長さ(これはSource Path Addressとは独立に変化しうる)、CRCのバージョン(Draft E/Draft F)、目的地ノードのワード幅(何バイトを1ワードと思ってアクセスするか)

これらの情報のデフォルト値は、RMAPDestinationクラス中のstatic constな変数で定義されており、以下のようになっています。

```
static const unsigned char DefaultLogicalAddress=0xFE;
static const unsigned char DefaultDestinationKey=0x00;
static const unsigned char RMAPCRCTypeDraftE=0x00;
static const unsigned char RMAPCRCTypeDraftF=0x01;
static const unsigned char DefaultCRCType=RMAPCRCTypeDraftF;
static const unsigned int DefaultWordWidth=1;
```

また、プログラムを記述する際の利便性を考えて、標準的なRMAPDestinationを集めておき、コマンドラインからユーザーが指定して利用することができるよう、RMAPDestinationKnownDestinations::selectInstanceFromCommandLineMenu()というstaticメソッドが用意されています。

```
RMAPDestination* destination=RMAPDestinationKnownDestinations::selectInstanceFromCommandLineMenu();
```

とすることで、利用できます。

SpaceWireIFを変更する方法

SpaceCube1で、NEC Soft製IPコアを使って通信を行っていたプログラムを、シマフジ製IPコアを使用するように変更する場合は、下記のように、SpaceWireIFクラスのインスタンス化の部分だけを書き換えればOKです。

[変更する前]

```
...略...
//NEC Soft製IPコア用SpaceWireIF classのインスタンス化
SpaceWireIF* spacewireif=new SpaceWireIFSpaceCube1NECSoftIPcore();
spacewireif->initialization();
spacewireif->open();
...略...
```

[変更したあと]

```
...略...
//シマフジ製IPコア用SpaceWireIF classのインスタンス化
SpaceWireIF* spacewireif=new SpaceWireIFSpaceCube1ShimafujiIPcore();
spacewireif->initialization();
spacewireif->open();
...略...
```

問題点やコメント

現状の問題点

現状のライブラリは、以下のような問題を抱えている、と思っています。他にもたくさん問題があるはずなので、適宜報告・修正してください。

- データのコピー回数

ライブラリ内でのデータのコピー渡しはなるべく控え、参照渡しを心がけましたが、実装しやすさとの兼ね合いで、まだまだコピー渡しが残っています。Javaではなく、C++なのでしかたない、とも考えられますが、頑張れば若干実行速度があがるかもしれません。

- 例外・エラー処理

各所で例外スローを宣言しているのに、実際にはスローしないメソッドがある(これは将来的な予約なのでOKか)。深刻なのはSpaceWireリンクが切れたときの処理で、いまのところ具体的に対策をとっていません。エラー処理は、ミッション・実験のエラーハンドリングポリシーとも絡るので、各自での対処(実装)になるかもしれません。

- コピー渡しとポインタ渡しの混在

Javaでは、メソッドをまたいだクラス渡しは、プリミティブ型を除いて参照渡しに統一されているのでわかりやすいのですが、C++では一意でない(コピー渡しかポインタ渡しか選べてしまう)ので、ややこしくなっています。このライブラリでも、コピー渡しを減らして速度を稼ぐため、引数や返り値がところどころポインタになっています。わかりづらいとは思いますが、EclipseなどのIDEを使えば、引数や返り値の方はインラインで表示されるので、それらのツールで補助して作業してください。

- SpaceWire Timecodeへの対応

抽象クラスSpaceWireIF.ccでは、まだTimecode関連のメソッドを定義していません。タイムコード送受信や、時刻合わせに関連するメソッドが将来的には必要になると認識しています。

- ドキュメンテーションが不十分

各クラス、メソッドに対するインラインドキュメンテーションが不足しています。コメントリビューションはおおいにencourageされています。

ライブラリの速度に関して

このライブラリを用いることで発生しうるオーバーヘッドについて述べます。具体的な転送速度については、「なまもの」的な要素と、企業秘密的な要素があるので、ここでは述べません(別ドキュメントなどで触れられるはずです)。

「配列」でなくvectorクラスを使うことによるオーバーヘッド

まず、このライブラリでは、データ列を表現するために、C言語的な「配列」ではなく、C++言語の標準テンプレートライブラリ Standard Template Library(STL)のvectorクラスを使っています。vectorクラスは、C++ STLにおいて、C言語の配列を置き換える目的で設計されたクラスで、サイズを自動調整してくれたり、サイズをvectorオブジェクト自身が知っていたり、定義されていない要素にアクセスしても、正しく例外をthrowして、バッファオーバーランしないように処理してくれるなど、C言語の配列にはない機能を豊富に持っています。一方で、vectorの方が、内部処理が増えることから、一般に（コンパイラにもよるかもしれません）配列よりも若干処理速度が低下します。しかしこのライブラリでは、vectorを用いることで得られる利点（配列のサイズを決めうちしなくてよい、サイズをインスタンス自身が知っているので、わざわざint sizeなどサイズ用変数を持ち出さなくてよい、バッファオーバーランしない、コードが読みやすい）と、それによって高められる開発効率を優先して、vectorを使った実装を行っています。

もちろん、異なる考え方もあり得て、速度優先でC言語的に書くのだ、というのもよいと思います。より高速な、より使いやすいライブラリがC言語で作られれば、そちらを使えばよいでしょう。

RMAPEngineを入れることによるオーバーヘッド

RMAPPacketで生成したパケット列を、RMAPSocket/RMAPEngineを通さずに、SpaceWire I/Fに直に流し込んで送受信を行い、RMAPアクセスを行うことも実際には可能です。この場合と、RMAPSocket/RMAPEngineを使用した場合の転送速度を比較することで、RMAPEngineを用いたマルチプルトランザクションを利用することで生じるオーバーヘッドを測定することができます。

「おまけソフト」の項で触れたspeedtestプログラムを用いて、NEC Softさん製のSpaceCube1用SpaceWire I/Fで測定を行ったところ、RMAPEngine/RMAPSocketを用いた場合は、用いない場合の約85%の転送速度になりました。この値はパケットサイズにも緩やかに依存しますが、RMAPEngineの導入で大体15%程度、転送速度が遅くなる、と言えます。

おまけソフトの使い方

このライブラリには、rmaphongoと、speedtestというおまけソフトがついています。

rmaphongo

概要

rmaphongo(main_rmaphongo.cc)は、SpaceCubeのコマンドラインから、RMAPのread/writeや、生パケットの送受信をするためのソフトで、SpaceWire IFボードのロジックのデバッグや、SpaceWireネットワークの試験に利用可能です。使い方は簡単で、表示されるコマンドメニューに表示された番号を選んで、アドレスや長さ、書き込みたいデータを入力します。起動直後に、使用するSpaceWireIF実装と、Read/Writeを行う宛先(RMAPDestination)を設定するよう求められます。

実行例

コンパイルして生成された実行形式(main_rmaphongo)を、rmaphongoという名前で(長いので名前を省略して)SpaceCubeに転送して、実行する例を示します。

```
[/SYS]% fget (ホスト名):/usr/local/te/SpaceWireHongo/build/t-kernel/main_rmaphongo rmaphongo
User: xxx
Password: xxx
[/SYS]% rmaphongo <== 実行
Which SpaceWireIF implementation:
  Shimafuji IP Core      [2]
  Shimafuji Router IP Core [3]
select > 2 <== 使用するSpaceWireIFの実装を選択
SpaceWireIFSpaceCube1ShimafujiIPcore : initialize
spw0 opened
fpga ver. = 0x1000
SpaceWireIFSpaceCube1ShimafujiIPcore : open (0)
connecting ...
link0 on (0x0000003f)
status = 0x0000003f
connected.
SpaceWireIFSpaceCube1ShimafujiIPcore : connect (0)
#####
### SpaceWire sample program : rmaphongo ###
###           ver 2.1 (20080624)           ###
###           This program is totally based on   ###
###           SpaceWire/RMAP 'Hongo' Library.   ###
#####

Please set Destination information
Select :
 1 : Input new RMAP Destination
 2 : Reset to default RMAPDestination
 3 : Dump Current RMAP Destination
 4 : Return to the menu
```

```

### Frequently Used Settings ####
10 : Old Shimafuji SpaceWire IF Boards
    (Logical Address=0x01, Draft E CRC)
11 : Shimafuji SpaceWire IF Boards
    (Logical Address=0xFE, Draft E CRC)
12 : Shimafuji SpaceWire IF Boards
    (Logical Address=0xFE, Draft F CRC)
20 : SpaceWireRouter by NEC
30 : SpaceWire Universal IO Board by MHI
>12                                         <== シマフジ電機SpaceWire DIOボードの設定を選択

RMAPDestination::dump() invoked
Destination
Logical Address   : 01
Path Address     : none
Destination Key  : 02
CRC Version      : Draft E
Word Width        : 1bytes-1Word

Source
Logical Address   : fe
Path Address     : none
Path Address Len. : 00

Menu :
Read(2bytes)          [1]
Write(2bytes)         [2]
Read(general)         [3]
Write(general)         [4]
Receive a raw SpW packet [5]
Send a raw SpW packet [6]
Set Destination Info [8]
Toggle Debug Mode [Off] [0]
Quit                  [9]

select > 2                                         <== RMAP Write
Write (2bytes)
address in hex > 00000000
data for 0100-0000 > abcd
Writing...done

Menu :
(...メニュー表示省略...)

select > 1                                         <== RMAP Read
Read (2bytes)
address in hex > 00000000
Reading...done
Read result :
0000-0000 abcd

```

デスティネーション変更の例

メニューに表示されるプリセット以外の機器に接続する場合や、ルータを経由した通信がしたい場合は、RMAPDestinationを変更する必要があります。メニューの8番(Set Destination Info)を選んで、RMAPDestinationを NEC製SpaceWireRouterボード向け再設定したときの例を示します。

```
(前略)
select > 8                                     <== 「Set Destination Info [8]」を選択
Select :
1 : Input new RMAP Destination
2 : Reset to default RMAPDestination
3 : Dump Current RMAP Destination
4 : Return to the menu
### Frequently Used Settings ####
10 : Old Shimafuji SpaceWire IF Boars
      (Logical Address=0x01, Draft E CRC)
11 : Shimafuji SpaceWire IF Boars
      (Logical Address=0xFE, Draft E CRC)
12 : Shimafuji SpaceWire IF Boars
      (Logical Address=0xFE, Draft F CRC)
20 : SpaceWireRouter by NEC
30 : SpaceWire Universal IO Board by MHI
>1                                         <== 「1 : Input new RMAP Destination」を選択

RMAPDestination::setRMAPDestinationViaCommandLine() invoked
--Destination Part--                         <== Destinationに関するプロパティの設定
Logical Address (hex) : Logical Address (hex) : fe
Path Address : (to terminate, please input a value greater than 0x100)
[0] (hex) : 0                                     <== 内部レジスタを表すパスアドレス「0」を指定
[1] (hex) : 999
Destination Key (hex) : Destination Key (hex) : 00
CRC Type :
Draft E [1]
Draft F [2]
1 or 2 : 1
Word Width (in byte) : 4                         <== 1ワードの幅を4バイトに設定(Router側の仕様)
--Source Part--                                <== Sourceに関するプロパティの設定
Logical Address (hex) : Logical Address (hex) : fe
Path Address : (to terminate, please input a value greater than 0x100)
[0] (hex) : 999                                     <== Source Path Addressなし
Source Path Address Length :
Auto [1]
Manual [2]
1 or 2 : 1

--- new RMAPDestination ---                     <== 設定した新しいRMAPDestinationをダンプ
RMAPDestination::dump() invoked
Destination
Logical Address   : fe
Path Address     : 0x00
Destination Key  : 00
CRC Version      : Draft E
Word Width       : 4bytes-1Word
```

```

Source
Logical Address    : fe
Path Address       : none
Path Address Len.  : 00

Select :
1 : Input new RMAP Destination
2 : Reset to default RMAPDestination
3 : Dump Current RMAP Destination
4 : Return to the menu
### Frequently Used Settings ####
10 : Old Shimafuji SpaceWire IF Boars
      (Logical Address=0x01, Draft E CRC)
11 : Shimafuji SpaceWire IF Boars
      (Logical Address=0xFE, Draft E CRC)
12 : Shimafuji SpaceWire IF Boars
      (Logical Address=0xFE, Draft F CRC)
20 : SpaceWireRouter by NEC
30 : SpaceWire Universal IO Board by MHI
>4                                         <== メインメニューに戻る

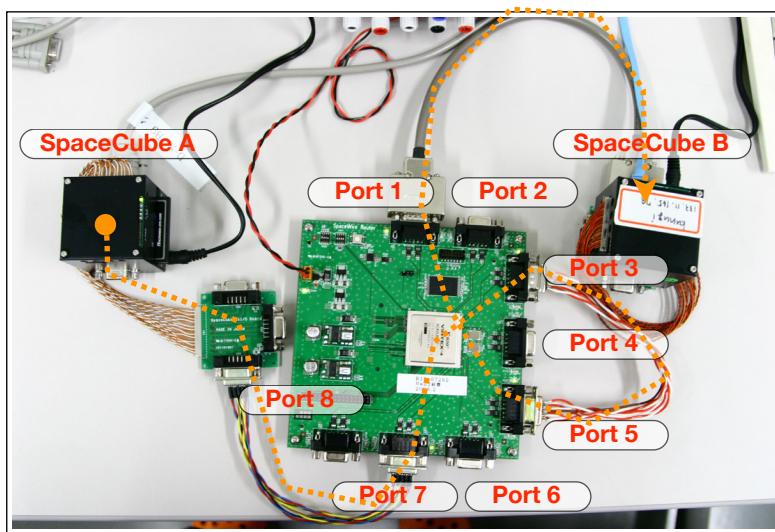
Menu :
Read(2bytes)          [1]
Write(2bytes)         [2]
Read(general)         [3]
Write(general)         [4]
Receive a raw SpW packet [5]
Send a raw SpW packet [6]
Set Destination Info [8]
Toggle Debug Mode [Off] [0]
Quit                  [9]

select > 3                                         <== SpaceWireRouterにreadしてみる
Read (general)
address in hex > 0                                     <== 内部レジスタのアドレス0x0000
length in decimal > 4                                    <== 長さは4バイト(1ワード)
Reading...done
Read result :
Address   Data
0000-0000  0x00fe                                     <== 結果表示
0000-0002  0x0000

```

SpaceWireRouterをまたいだパケット送受信の例

rmaphongoは、メニュー表示にもあるように、2バイト単位のread/write、任意長さのread/write、生SpaceWireパケットの送受信が行えます。生パケット送受信の機能を使って、SpaceWireRouterボードをはさんで二つのSpaceCubeがPath Addressingでパケットを送受信し合う例を示します。二つのSpaceCubeは以下のように、ルータのPort 1とPort 7に接続されています。ルータのPort3とPort 5は、SpaceWireケーブルでつながっています。SpaceCube AからSpaceCube Bへ、オレンジの破線のような経路でパケットを送ってみます。



(SpaceCube Aの画面)

```
...メニューは省略...

select > 6      <== 生パケット送信モードを選択
Send a raw SpW packet
enter packet data to be sent (in hex)...
to finish input, type a number greater
than 0xFF

0000 >3      <== 送信する生パケットを入力
0001 >1      <== 0x03と0x01は
0002 >a      <== ダミーのカーゴ部分
0003 >b
0004 >c
0005 >d
0006 >999     <== 入力終了

Entered Packet values are :
0x03 0x01 0x0a 0x0b 0x0c 0x0d

Select :       <== これでよいか確認される
1 : OK
2 : NG, re-input
3 : Cancel Sending (return to menu)
>1

Sending...done
```

(SpaceCube Bの画面)

```
Menu :
Read(2bytes)      [1]
Write(2bytes)      [2]
Read(general)      [3]
Write(general)      [4]
Receive a raw SpW packet [5]
Send a raw SpW packet [6]
Set Destination Info [8]
Toggle Debug Mode [Off] [0]
Quit                [9]

select > 5 <== 受信状態で待ちにはいる
Receive a raw SpW packet
waiting a packet...received

Raw packet dump
0x0a 0x0b 0x0c 0x0d    <== 到着したパケットの中身

Interpret this packet as an RMAP packet? (y or n) :n
<== RMAP Packetを送った訳ではないので、RMAPの解釈は不要

左で入力したパケット内容のうち、バスアドレスを表す先頭の0x03と
0x01が、ルーティングの際にたやすくはぎ取られて、カーゴ部分だけ
が届いているのがわかる。
```

speedtest

概要

speedtest(main_speedtest.cc)は、SpaceCubeのSpaceWire I/Fの転送速度を測定するためのソフトです。SpaceCubeとシマフジ電機製SpaceWire I/Fボードを接続し、ボードに搭載されているSDRAMに連続的にReadアクセスを実行し、あるサイズのデータをN回転送するのにかかった時間を表示します。RMAPEngineが利用できるSpaceWire I/Fでは、モード選択メニューで、「via SpaceWireIF」と「via RMAPSocket」が選択可能で、SpaceWireIFを生で利用した場合と、RMAPEngine/RMAPSocketを用いた場合のオーバーヘッドの違いが比較できます。

以下、シマフジ電機製のSpaceWire IPコアを搭載したSpaceCubeでの実行例を示します。

```
[/SYS]% speedtest
Which SpaceWireIF implementation: [1] <== 搭載されているSpaceWire IFを選択
    NEC Soft IP Core [1]
    Shimafuji IP Core [2]
select > 2 <== シマフジ電機製SpW IPコア搭載SpCなので、「2」
SpaceWireIFSpaceCube1ShimafujiIPcore : initialize <== SpaceWire I/Fの初期化処理
fpga ver. = 0x0001
SpaceWireIFSpaceCube1ShimafujiIPcore : open (0)
connecting ...
status = 0x003f (err = 0x00000000)
connected.
SpaceWireIFSpaceCube1ShimafujiIPcore : connect (0)

#####
# SpaceWire speed test #
#####

Menu :
    SDRAM access (via SpaceWireIF) [1]
    SDRAM access (via RMAPSocket) [2]
    Quit [9]

select > 1 <== RMAPEngineを使わない転送速度を試験するので、「1」
initial address of read access
address in hex > 00000000 <== SDRAMのアドレスを指定
length of each access
length in decimal > 2000 <== パケットサイズ(この例では、2kBを指定)
number of iteration
iteration number in decimal > 1000 <== 繰り返し回数は1000回に設定
display frequency
show degree of process per N accesses : N > 100 <== 100回ごとにメッセージ表示

Start Read Access
Done (1) <== 転送中...
Done (101)
...
Done (901)
Transferred 2000kBytes in 11.48sec <== 転送完了
Bit-rate 1393.73kbit/s <== 2kB × 1000 = 2MBを11.48secで転送したので、
Completed <== 1.39Mbit/sの転送速度、とわかる
```

参考文献

- [1] S.M. Parkes et al., "SpaceWire: Links, Nodes, Routers and Networks", European Cooperation for Space Standardization, Standard No. ECSS-E50-12A, Issue 1, January 2003
URL : <http://spacewire.esa.int/content/TechPapers/documents/SpaceWire%20Standard%20%20ISWS%202003.pdf>
- [2] Parkes, S. & Rosello, J. 2003, "SpaceWire ECSS-E50-12A", International SpaceWire Seminar, Noordwijk, The Netherlands
URL : <http://spacewire.esa.int/content/TechPapers/documents/SpaceWire%20Standard%20%20ISWS%202003.pdf>
- [3] S.M. Parkes et al., "SpaceWire Protocol ID", European Cooperation for Space Standardization, Standard No. ECSS-E-50-11 Draft B, February 2005
URL : <http://spacewire.esa.int/content/Standard/documents/SpaceWire%20Protocol%20ID%20ECSS-E-50-11%20Draft%20B%202005%20February%202005.pdf>
- [4] S.M. Parkes et al., "RMAP Protocol Specification", European Cooperation for Space Standardization, Standard No. ECSS-E-50-11 Draft F, December 2006
URL : <http://spacewire.esa.int/content/Standard/documents/SpaceWire%20RMAP%20Protocol%20Draft%20F%204th%20Dec%202006.pdf>
- [5] T. Takahashi, "科学衛星データ処理系の将来展望", 宇宙科学シンポジウム, JAXA相模原キャンパス, 2005
URL : http://www.astro.isas.jaxa.jp/~takahashi/DownLoad/ISAS_Sympo_DataProcess2005_3.pdf
- [6] M. Nomachi, "次世代データ収集プラットフォーム開発", 「ストレンジネスで探るクオーク多体系」研究会, 仙台, November, 2007
URL : <http://nexus.kek.jp/Tokutei/workshop/2007/pdf/P2605Nomachi.pdf>
- [7] T. Takahashi et al., "SpaceCube 2 -- An Onboard Computer Based on SpaceCube Architecture", International SpaceWire Conference, Dundee, September, 2007
URL : presentation slide
<http://spacewire.computing.dundee.ac.uk/proceedings/Presentations/Onboard%20Equipment%20and%20Software/takahashi.pdf>
URL : proceeding
<http://spacewire.computing.dundee.ac.uk/proceedings/Papers/Onboard%20Equipment%20and%20Software/takahashi.pdf>

- [8] T. Hagihara, “撮像型X線TESマイクロカロリメーターのデジタル信号処理”, 東京大学理学系研究科修士論文, January, 2007
URL : http://www.astro.isas.jaxa.jp/~mitsuda/lab0/index.php?plugin=attach&refer=Thesis&openfile=MThesis_ THagihara.pdf
- [9] H. Odaka et al., “Development of a SpaceWire-based Data Acquisition System for a Semiconductor Compton Camera”, International SpaceWire Conference, Dundee, September, 2007
URL : presentation slide
<http://spacewire.computing.dundee.ac.uk/proceedings/Presentations/Missions%20and%20Applications%202/odaka.pdf>
URL : proceeding
<http://spacewire.computing.dundee.ac.uk/proceedings/Papers/Missions%20and%20Applications%202/odaka.pdf>
- [10] T. Yuasa et al., “Development of a SpW/RMAP-based Data Acquisition Framework for Scientific Detector Applications”, International SpaceWire Conference, Dundee, September, 2007
URL : presentation slide
<http://spacewire.computing.dundee.ac.uk/proceedings/Presentations/Missions%20and%20Applications%202/yuasa.pdf>
URL : proceeding
<http://spacewire.computing.dundee.ac.uk/proceedings/Papers/Missions%20and%20Applications%202/yuasa.pdf>
- [11] W. Kokuyama, “衛星搭載用超小型重力波検出器の開発”, 「高エネルギー天体现象と重力波」研究会, 東京大学本郷キャンパス, November, 2007
URL : presentation slide
http://www.gw.hep.osaka-cu.ac.jp/HE_ASTRO_GW/viewgraph/he_astro_gw07_1117_kokuyama.pdf
- [12] K. Ishidohiro, “FPGAを用いた小型宇宙重力波検出器(SWIM μ v)のデジタル信号処理系”, 重力波研究交流会, April, 2008
URL : http://tamago.mtk.nao.ac.jp/gw_talks/080404/talk1/gw_talks20080404final.ppt
- [13] NEC Corporation, “SpaceWire Router IP 取り扱い説明書 Rev1.0”, NECST-SPF5PL-06004, March 2007
- [14] T. Yuasa et al., “SpaceWire/SpaceCube Tutorial”, 2008 (最新版はSpaceWire Wikiで入手可能)

オンラインで入手可能なその他の参考文献

- [A] H. Odaka, T. Yuasa et al., “SpaceWireのつなげかた”
URL : http://www.astro.isas.jaxa.jp/~odaka/spacewire/documents/HowToConnectSpaceWire_1.0.pdf
(要パスワード。SpW-MLに問い合わせてください)
- [B] Doxygen URL : <http://www.doxygen.jp>
- [C] SpaceWire Wiki URL : <http://www.astro.isas.jaxa.jp/SpaceWire/wiki/>
- [D] Xilinx ISEのダウンロード、マニュアル URL : http://japan.xilinx.com/ise/logic_design_prod/webpack.htm
- [E] Yuasa, “SpaceWire-based Data Acquisition System”, 理化学研究所 牧島宇宙放射線研究室 地の共有ゼミ, 2008年4月
URL : http://www-utheal.phys.s.u-tokyo.ac.jp/~yuasa/conference/riken_200804/yuasa_spw_20080430_1230.pdf
- [F] JTAGの解説 URL : http://www.ednjapan.com/content/issue/2007/03/content05_02.html
- [G] Eclipse IDE URL : <http://www.eclipse.org/>

[H] SpaceWire/RMAP Library Online API Reference

URL : <http://www-utheal.phys.s.u-tokyo.ac.jp/~yuasa/spc/SpaceWireRMAPLibrary/apireference/html/>

参考書籍

[a] パーソナルメディア株式会社, “T-Kernel 組み込みプログラミング 強化書”, パーソナルメディア,

ISBN978-4-89362-246-4