衛星搭載SpaceWire通信を用いた複数機器の同時読み出し

広島大学理学部物理科学科
B095962
井上 翔太
高エネルギー宇宙・可視赤外線天文学グループ
主査：高橋 弘充　副査：杉立 徹

2013年2月26日

概 要

ブラックホールや中性子星、銀河団の高温プラズマといった高エネルギー天体からは、X 線やガンマ線といった短波長領域での電磁放射が観測されている。これらの天体を研究することは、宇宙の進化を探る上で非常に重要であり、その放射機構には未だ多くの謎があるので、さらなる詳細な観測、研究が必要である。

X 線、ガンマ線の観測は、地球大気で吸収されてしまうため、人工衛星や気球を飛ばして、大気圏外での観測を行わなければならない。より精度よく観測を行うためには、検出器の改良が欠かせないのであるが、正しく信号を読み取るという意味でデータ収集システムの整備も非常に重要である。

従来、機体に搭載される様々な機器間の通信インターフェイスは、それぞれ独立に作られてた。これにより、機器間の通信規格の変換に伴うハードウェアやソフトウェアを考慮しなければならず、開発期間の長期化、それらのスペースの確保、そして異なるインターフェイス間の信号どうしの干渉などという問題があった。それを解消するべく、SpaceWire という通信インターフェイスの世界統一規格が提唱されている。現在、SpaceWire は、ESA を始め、JAXA、NASA などの様々な研究機関、大学、企業が参加している。2015 年打ち上げ予定の日本の次期 X 線天文衛星 ASTRO-H や気球搭載用硬 X 線偏光計 PoGOLite も通信規格として SpaceWire が採用される。

打ち上げ後も開発時と同じように通信できるよう、これら衛星（または、気球）に搭載する機器の開発環境においても、機器間の通信は SpaceWire により行われる。SpaceWire は専用の SpaceWire ケーブルを用いて通信が行われるのであるが、実験装置からのデータは PC で読み取るため、PC のイーサネットケーブルと SpaceWire ケーブルの変換器として SpaceWire-to-GigabitEther(以下、SpW-to-GbE) が必要となる。この SpW-to-GbE は、１台の実験装置に対して、１台必要であったが、近年 1 本のイーサネットケーブルに対し 4 本の SpaceWire ケーブルを接続できるルーター機能付きの SpW-to-GbE が登場した。これを用いれば、1 台の SpW-to-GbE で、4 台の実験装置を動作させることが可能となる。

また、SpaceWire 通信では、RMAP というプロトコルが使える。RMAP は、SpaceWire の上位層にあたる通信プロトコルであり、日本の SpaceWire ユーザーの間で一般的に採用されている。これを使うと他の機器のメモリをあたかも自身のメモリのように制御通信することが可能となる。この時、操られる機器に CPU が実装されている必要はないので、データの変換の手間や、そのためのハードウェアの配置スペースを省略できる。この RMAP で制御通信を行うためには、そのためのソフトウェアが必要なのであるが、その雛形は宇宙研の湯浅氏を始めとする人々により完成されており、ユーザーは自身の作りたいモジュールのみを作成すればよい。この雛形は、SpaceWire RMAPLibrary（以下、RMAPLibrary）と呼ばれており、web 上で配布されている。この RMAPLibrary は、昨年から ver.2 がリリースされ、SpaceWire や RMAP 機能の柔軟性・可制御性を向上させるための様々な新しい機能が追加された。このような背景から、ASTRO-H プロジェクトや PoGOLite 計画に参加している我々の研究室としても、ver.2 への移行することが望まれる。

そこで私は、衛星搭載用検出器の開発環境の向上を目的とし、ルーター機能付きの SpW-to-GbE を当研究室に導入することにより、必要なハードウェアの省略と実験室内の省スペース化に努めた。加えて、RMAPLibrary ver.2 をインストールし、ver.1 で扱っていたソフトウェアを ver.2 で使えるようプログラムの書き換えを行うことで、当研究室への最新 SpaceWire RMAP プロトコル環境の構築を目指した。

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

宇宙にある様々な天体は、様々な波長域で電磁波を放出しており、ブラックホール、超新星残骸、銀河団などの高温高密度の領域からは、X 線やガンマ線の放射が観測されている。これらの天体を研究することは、宇宙の進化を探る上で非常に重要であり、その放射機構についてまだまだ謎が多いので、さらなる詳しい観測、研究が求められる。

宇宙からの X 線、ガンマ線は、地球の大気に吸収されてしまうため、観測は大気圏外で行う必要があり、人工衛星により X 線を検出しなくてはならない。現在、JAXA(日本) の「すざく」、NASA(アメリカ) の「Chandra」、ESA(ヨーロッパ) の「XMM-Newton」などの X 線天文衛星が宇宙空間の各軌道上で観測を続けている。天体からのデータは、これらの衛星に搭載された検出器で検出され、データ処理装置や記憶装置を介し、最終的に地上との通信装置に送られる。そのため、衛星に搭載された各々の装置には、他の検出器との通信インターフェイスが必要である。従来の衛星では、このような通信インターフェイスは衛星ごと、機器ごとに独立して作られていた。これにより、開発者は検出器のみならず、通信インターフェイスも念頭に入れて開発を行わなければならず、開発期間の長期化といった問題も生じていた。また、開発期間の長期化や送られてくる信号の信頼性の欠落、技術が継承されにくいなどといったことが問題となっていた。このような問題を解決するべく、現在この通信インターフェイスの統一規格として SpaceWire が進められている。SpaceWire は ESA を始め、JAXA や NASA など、世界中の主要な宇宙関連機関、大学、企業が参加しており、2015 年打ち上げ予定である日本の次期 X 線天文衛星 ASTRO-H でも通信インターフェイス規格として SpaceWire が採用されている。

ASTRO-H 搭載予定の検出器のうち、我々の研究室は、半導体多層コンプトンカメラ Soft Gamma-ray detector(SGD) と硬 X 線領域での撮像器 Hard X-ray Imager(HXI) の開発に参加しており、実験室にて開発中である。実験室での検出器開発において、機器間の通信は SpaceWire により行われる。こうすることにより、衛星打ち上げ後も開発時と同じように検出器とのデータ通信ができるのである。SpaceWire は、専用の SpaceWire ケーブルで通信が行われる。実験装置からのデータは PC で読み取るため、PC のイーサネットケーブルと SpaceWire ケーブルの変換器として SpaceWire-to-GigabitEther(以下、SpW-to-GbE) を使用する。基本的には、１つの実験装置に対して、１つの SpW-to-GbE が必要であったが、近年１つのイーサネットケーブルに対し４つの SpaceWire ケーブルを接続できるルーター機能付きの SpW-to-GbE が登場した。これにより、１つの SpW-to-GbE で、４つの実験装置を動作させることが可能となる。我々の研究室においても、実験機器セットアップの省スペース化のため、このルーター機能付き SpW-to-GbE を導入する必要がある。

また、SpaceWire 通信では、RMAP という通信プロトコルが使えるのも大きな特徴である。詳しくは後述するが、RMAP は日本の SpaceWire ユーザーの間で一般的に採用されている SpaceWire の上位層の通信プロトコルであり、これを使うことにより、他の機器のメモリをあたかも自身のメモリのように制御通信ができる。その際、操る他の機器に CPU が実装されていなくてもアクセスが可能となる。RMAP を使うこ

とにより、データ変換の手間や、そのためのハードウェアの配置スペースを省略できるのである。この便利な RMAP を使い制御通信を行うためには、そのためのソフトウェアが必要なのであるが、その雛形は宇宙研の湯浅氏を始めとする人々により完成されており、ユーザーは自身の作りたいモジュールのみを作成すればよい。この雛形は、SpaceWire RMAP Library(以下、RMAPLibrary) と呼ばれ、web 上で公開されていて、誰でもダウンロード可能なオープンソースである。RMAPLibrary は、2006 年に配布が開始され、日本の多くの SpaceWire ユーザーの間で利用されたが、当時は RMAP の起草時期であり、RMAPLibrary ver.1 での RMAP 実装は試験的であったため、現在では RMAPLibrary ver.1 の多くの機能が使えない状態となっている。そのような中で、RMAPLibrary は、湯浅氏たちにより全体的に書き換えられ、RMAPLibrary ver.2 がリリースされた。ver.2 には、SpaceWire や RMAP 機能の柔軟性・可制御性を向上させる様々な新しい機能が追加されている。RMAPLibrary ver.2 は、2012 年にリリースされ、我々の研究室でも、ver.2 への移行が迫られている。

図 1.1: 日本の次期 X 線天文衛星 ASTRO-H[1]

1.2 目的

前述の通り、通信インターフェイス規格を SpW に統一することにより、開発期間の短縮、データの信頼性の確保、技術の継承がしやすいなどといった便益が生じる。SpaceWire は、開発者が衛星搭載用機器の開発に専念するために重要な役割を果たす。この SpaceWire 通信で検出器や実験装置のデータを PC で読み取る際には、SpW-to-GbE が非常に便利である。従来では１つの実験装置に対して１つの SpW-to-GbE が必要という状況であった。

また、RMAP プロトコルの柔軟性や可制御性を向上させるため、RMAPLibrary ver.2 がリリースされた。SpaceWire 通信は、2015 年打ち上げ予定の ASTRO-H でも採用されることが決まっており、ASTRO-H プロジェクトにおいて、SGD と HXI の開発に参加している我々の研究室としても、RMAPLibrary ver.2 への移行が必要不可欠である。

そこで私は、衛星搭載用検出器の開発環境の向上を目的とし、ルーター機能付きの SpW-to-GbE を当研究室に導入することにより、必要なハードウェアの省略と実験室内の省スペース化に努めた。また、同じ目的により、RMAPLibrary ver.2 をインストールし、ver.1 で扱っていたソフトウェアを ver.2 で使えるようプログラムの書き換えを行うことで、当研究室への最新 SpaceWire RMAP プロトコル環境の構築を目指した。

第2章　科学衛星用統一通信規格 SpaceWire

2.1　SpaceWire

宇宙へ打ち上げる科学衛星は、観測装置、各種センサ、データ処理装置、記憶装置、通信装置など多種多様な機器が搭載されている。これらの間で、データが受け渡しがなされて、衛星システムが機能しており、その通信システムの構築は科学衛星を運用する上で非常に重要である。従来の衛星では、これらの様々な装置は、それぞれ独立に開発され、通信規格も異なっていた。これにより、開発期間の長期化、データの信頼性欠落、技術が継承されにくいといった様々な問題が生じ、このことが科学衛星搭載機器の開発者たちにとって大きな負担となっていた。このような問題を解決すべく、世界で衛星搭載機器間の通信インターフェイス規格を統合しようという動きが拡がっており、現在 SpaceWire という衛星搭載用統一通信規格が提唱されている。SpaceWire は、IEEE1355 をベースとして定められた通信規格で、欧州宇宙機関 ESA を初め、JAXA/ISAS、NASA など世界の主要な研究機関、大学、企業で採用されている。SpaceWire への統一により、製作期間の短縮、コストの削減、信頼性の向上、技術の蓄積、必要なハードウェアの省略といった様々な便益がもたらされるため、さらに使いやすいように改良、曖昧さの明確化に向けて研究が行われている。

また、SpaceWire 通信規格を採用すると RMAP と呼ばれるプロトコルが使える。これは、SpaceWire の上位層のプロトコルにあたり、RMAP Packet を送受信することによって、他の機器のメモリをあたかも自分の CPU のメモリのようにを操作できる。

このように SpaceWire という統一的な規格にすることで、データの信頼性確保、開発期間の短縮、機器間の通信プロトコルにおける変換機器の不要といった様々なメリットが生まれる。開発を全く別々に行っていても通信規格を SpaceWire に統一しておけば簡単に接続でき、プロセッサが設置されていなくても SpaceWire と接続されていればその機器は RMAP によってその装置のレジスタをさながら自身のメモリを読み取るかのように知ることができる。このような特徴から、車載機器や通信機器業界からの関心も高く、民生機器への応用も期待されている。

2.1.1　SpaceWire の特徴

SpaceWire は、１ラインあたり 2Mbps〜400Mbps という高速なデータ転送レートをサポートする。この可変なデータ転送レートにより、様々な機器に柔軟に対応できる。伝送には、Low Voltage Differential Signaling (LVDS) 方式を採用している。LVDS は低電圧での差動伝送方式であり、これにより高速通信でも低消費電力、低ノイズでの伝送を可能とする。また、SpaceWire では DS-LINK を採用し、デジタル回路に必要な Clock ラインを持たないが、Data 信号と同時に Strobe 信号を伝送し、図 2.1 のように XOR をとることで Clock 信号を再現している。これにより、Clock 信号そのものを伝送する場合に比べて skew への耐性を高くしている。

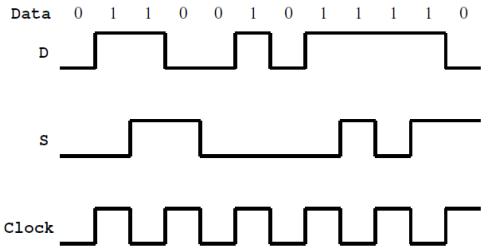

図 2.1: clock 信号の再現

SpaceWire は全二重方式を採用しており、入力と出力を同時に行うことができる。ケーブルは Data と Strobe が入出力あわせて 2 組で構成されている。LVDS 信号を伝送するために 2 本の導線をよりあわせたツイストペアの構造になっており、Data 信号と Strobe 信号の送受信のそれぞれ 2 本、合計 8 本のケーブルにより信号を伝送する。それぞれのツイストペアはシールドされ、さらにケーブルの外周もシールドされているため、クロストークや外部からのノイズの影響が少なく、ケーブルからのノイズの放射も抑えられている。そのため、ケーブルを 10m 以上伸ばすことができ、柔軟な機器配置が可能となる。また、コネクタは宇宙用途で多用されている 9 ピンの D-sub コネクタ、あるいは 9 ピンの micro D-Sub コネクタが指定されている。図 2.2 および図 2.3 にケーブルの構造、コネクタのピンアサインを示す。また、図 2.4 にケーブルの結線図を示す。

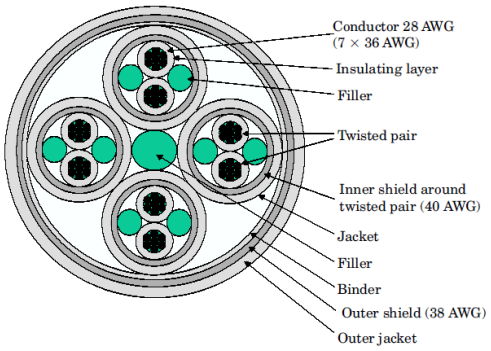

図 2.2: SpaceWire ケーブルの構造 [2]

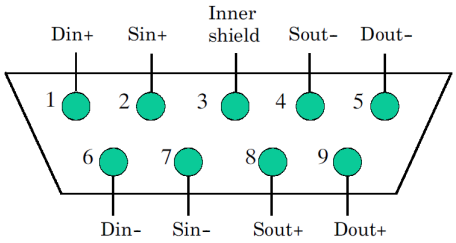

図 2.3: コネクタのピンアサイン [2]

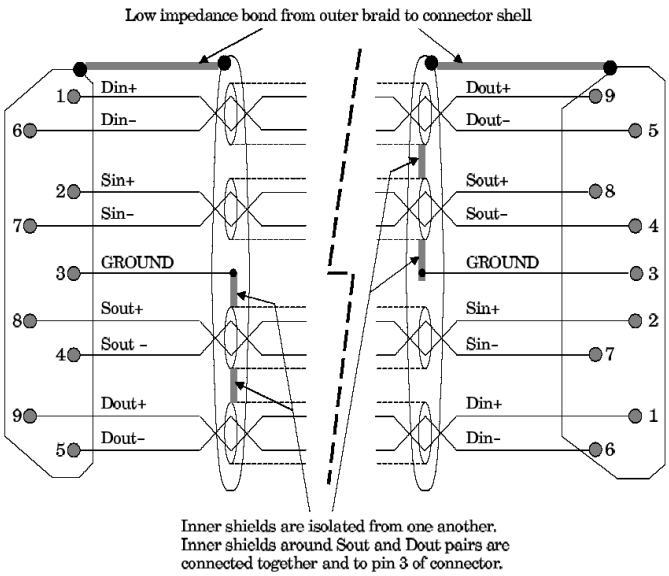

図 2.4: SpaceWire ケーブルの結線図 [2]

SpaceWire 通信におけるパケット（SpaceWire パケット）は、宛先（Destination ）とパケットの中身（Cargo）、パケット終端（EOP）から構成される。（図 2.5）Destination の部分では、アクセスしたいノードのロジカルアドレスやパスアドレス（後述）を指定する。また、Cargo には RMAP など上位層のパケットが格納されている。

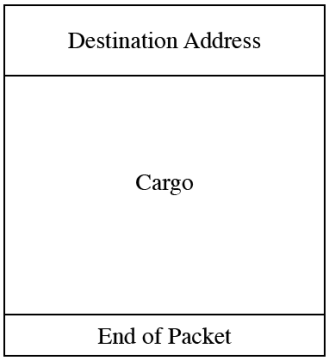

図 2.5: SpaceWire パケット [3]

SpaceWire は、１対１の通信ではなく、ルータを介して複数のノードが接続される通信を前提としている。SpaceWire 通信における Destination の指定には、Path Addressing と Logical addressing の２種類の方法がある。

Path Addressing は、ノードへ行き着く間にあるルータの出口、つまりパケットが出ていくべきポートの番号の列を指定する仕組みである。図 2.6 にその例を示す。Path Address は、0〜31 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、1〜31 はルータの実際のポート番号に対応している。Path Addressing はルータを到達するたびに、先頭の文字が解釈され、ポートを選択した後、その 1 バイトの文字が消去され、次のルータへ向かうという作業を繰り返すことにより指定したノードへ到達する。

Logical Addressing は、SpaceWire ネットワークにおける固有のアドレスを割り当てる方式で、Logical Address は Internet Protocol で使われる IP アドレスのような役割を果たす。Logical Addressing では、ユーザが通信を行う前に、各 Logical Address がどのポートに対応しているのかを示した表（Routing Table または、Routing Matrix）をルータに書き込んでおき、その表にしたがって指定したノードへアクセスする。図 2.7 にその例を示す。Logical Address には、Path Address で使用される 0〜31 の数字を除いた、32〜254 の数字で指定される。

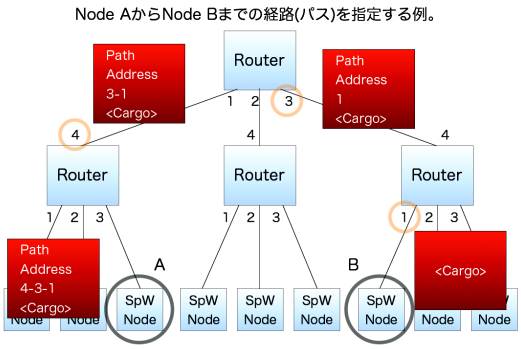

図 2.6: Path Addressing の例 [3]

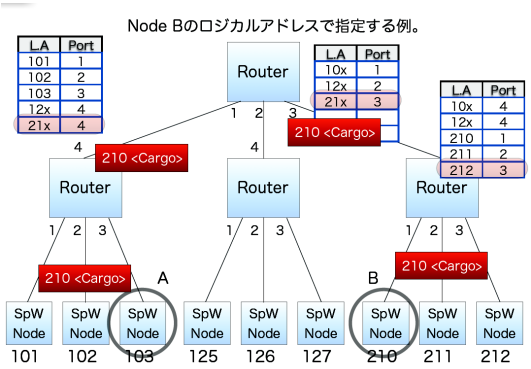

図 2.7: Logical Addressing の例 [3]

以上で述べた仕組みにより通信が行われる SpaceWire 通信規格では、スター型やリング型など自由なトポロジを構成することができる。そのため、SpaceWire 通信を用いれば、あるケーブルで不具合が生じ、通信が断絶したとしても、他の経路によりデータ通信を行えるよう機器を配置することが可能である。

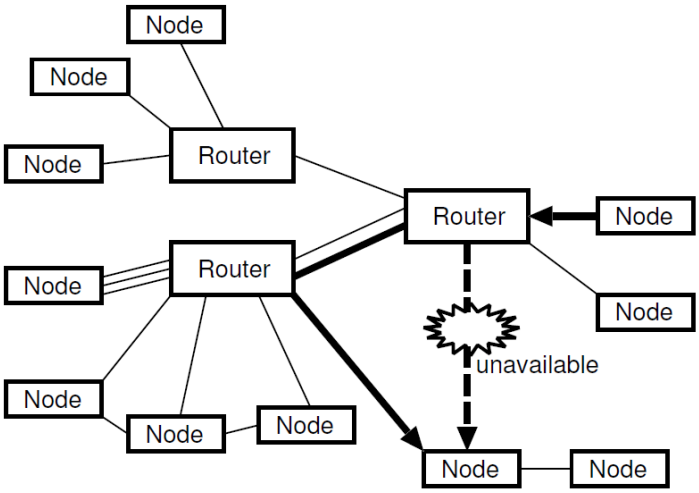

図 2.8: SpaceWire 通信における損傷時の迂回パス例 [4]

2.1.2 SpaceWire I/F 搭載 Board

地上の実験室では、様々な入出力機能を持つ SpaceWire I/F 搭載 Board を用いて実験が行われている。Field Programable Gate Array（FPGA）というユーザーが書き換え可能なプロトコルチップを搭載している。我々の研究室で利用している主なボードとしては、Digital I/O Board(図 2.9)、FADC Board(図 2.10) がある。ボード上には 2 つの FPGA が搭載されている。1 つは SpaceWire 通信を司る SpaceWire FPGA、もう 1 つはユーザーが任意の回路を製作するための User FPGA である。2 つの FPGA は External Bus と呼ばれるバスで接続されており SpaceWire FPGA を通して User FPGA 内のメモリやレジスタにアクセスが可能である。図 2.11 は SpaceWire FPGA におけるメモリマップの例である。先頭の 16MB の空間には SDRAM がマッピングされている。SpaceWire FPGA との双方向通信を利用することでアクセスでき、データの Read/Write が行える。ボードには CMOS ディジタルポートや LVDS ディジタルポートなどが実装されており、FPGA を書き換えて任意の回路を制作することにより、あらゆる実験へ対応することができる。また、0x01010000-0x0101FFFF までは External Bus となっており、User FPGA 内のメモリやレジスタをマッピングすることができる。

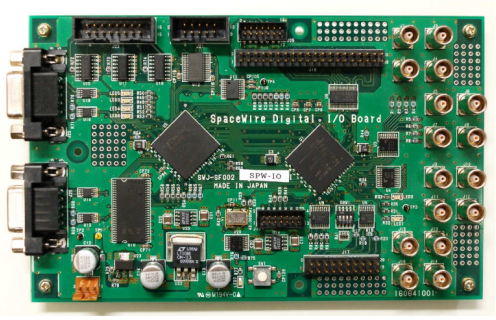

図 2.9: DIO Board

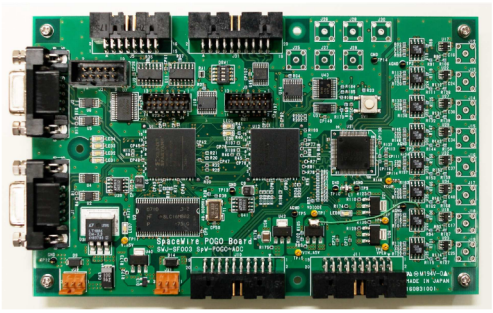

図 2.10: FADC Board

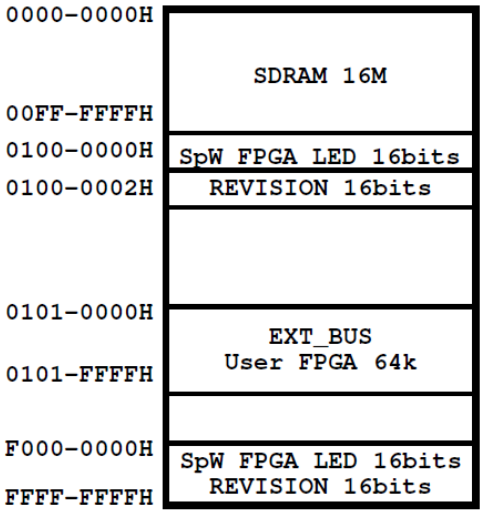

図 2.11: SpaceWire FPGA のメモリマップ例 [6]

ボード上には SpaceWire ケーブルを繋ぐ 2 つのポートがあり、それぞれのポートにはロジカルアドレスが割り振られている。ロジカルアドレスは、32〜254 までの番号により指定され、SpaceWire 通信の際、これを用いることにより任意のノードへアクセスできる。

ここで、FPGA についてもう少し詳しく説明しておく。FPGA は汎用的な論理ブロック (Logic Elements や Slice などと呼ばれる) を多数並べ、それぞれを接続する配線を繋ぎ変えることで瞬時に回路を変更することができるプロトコルチップである。従来のチップでは、開発から製造までに時間がかかり、一度つくってしまうと回路の変更は不可能とされていたが、FPGA ではこの問題が解消される。FPGA の回路の書き換えには、HDL(Hardware Description Language) を使用する。これを用いると回路動作をコンピュータのプログラミング言語のように記述できる。HDL の使用は、回路図を作る手間の省略、設計者以外の者が回路の意図を読みやすい、誤作動に対処しやすいといったメリットがある。現在、HDL として VHDL と Verilog HDL が主流になっており、我々のコミュニティでは主に VHDL が用いられている。

2.1.3 SpaceWire-to-GigabitEther

2.1.2 の SpaceWire I/F Board との通信は、もちろん SpaceWire ネットワークに接続している。地上の実験室などでは、この実験ボードで取得したデータを自身のパーソナルコンピュータで解析しなければならないので、PC からのイーサケーブルを SpaceWire ネットワークに接続しなければならない。この TCP/IP プロトコルと SpaceWire 通信規格の変換器との役目を、SpaceWire-to-GigabitEther（以下、SpW-to-GbE）が担っており、この変換器によりギガビットイーサネットから SpaceWire ネットワークへの接続が可能となる。ユーザーは一般的なコンピューターでプログラムを実行し、SpaceWire Packet を送受信できる。このプログラムとして、C++プログラミング言語で書かれたクラスライブラリも使われる。ライブラリを使うことにより、ユーザーは、SpaceWire ネットワークを通してに SpW-to-GbE に繋がれた RMAP ターゲットノードへ RMAP 通信を行うことができる。SpW-to-GbE は、科学実験における SpaceWire/RMAP を基礎とするデータ取得システム、SpaceWire インターフェースを採用したフライトモデルの地上試験などを意識して作られており、実際に打ち上げる衛星には搭載されない。

SpW-to-GbE は、ZestET1 ボードを使用している。SpaceWire Open IP や Host I/F が、そのボードの User FPGA 上に実装されている。Host I/F は、搭載された TCP/IP スタックを制御でき、プログラムを実行するコンピュータからの接続を受け付ける TCP サーバとして操作する。接続が解除されると待機状態となり、新しい接続を待つ。

SpaceWire I/F は、10.4MHz をデフォルトとして動作する。リンク速度は可変であり、Tx クロックが最大で、デバイスに実装された FPGA の本来のクロックと等しい 125MHz まで変化する。SpW-toGbE は、Error End of Packet(EEP)/End of Packet(EOP) を用いることによりどんなサイズのパケットでも送受信できる。また、タイムコードも送受信できる。

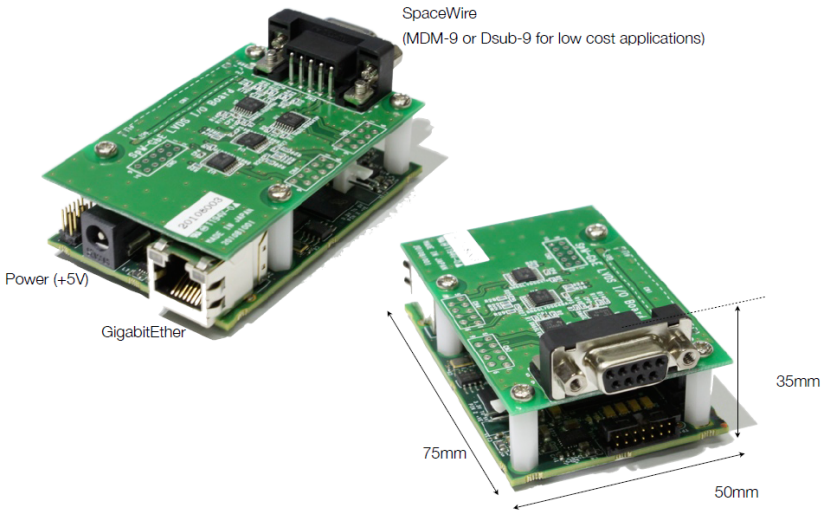

図 2.12: SpaceWire-to-GigabitEther[7]

従来、SpW-to-GbE は、１本のイーサケーブルに対して１本の SpaceWire ケーブルしか変換できない仕様であったが、４ポートの SpaceWire ケーブル用ポートが実装されたルータ機能付き SpW-to-GbE が登場した。これが、フジシマ電機製スタンドアロン版 SpaceWire-to-GigabitEther（以下、４端子 SpW-to-GbE）である。ブリッジ機能は FPGA に実装されており、PC からドライバレスで 800Mbps 超のデータ転送を実現する。コネクタ部分は micro D-Sub コネクタとなっている。

図 2.13: フジシマ電機製スタンドアロン版 SpaceWire-to-GigabitEther

2.1.4 Remote Memory Access Protocol(RMAP)

SpaceWire 通信規格の上位層のプロトコルとして Remoto Memory Accsess Protocol（RMAP）があり、現在、この標準化作業が進められている。RMAP では、RMAP Command Packet とそれに対する RMAP Reply Command Packet を、SpaceWire ネットワークにより送受信することで実行される「トランザクション（Transaction）」を単位として動作する。RMAP では、他の機器に実装されたメモリをあたかも自分の CPU のメモリであるかのように操作することを可能とするプロトコルである。以下でその原理について簡単に述べる。

通常、PC 上のプログラムで何かのデータを参照したいというときは、CPU がデータの格納されたメモリからデータを取り出してこれを実行する。この時 CPU は、メモリのアドレス空間を参照してデータを取り出す。同じように外部機器との I/O 機能を参照するためにメモリアドレスとは別に I/O アドレスというものがあるが、この I/O アドレスはメモリアドレスよりも空間範囲が狭いため、メモリアドレス上に配置して擬似的に I/O アドレスのようにすることができる。これはメモリマップド I/O という機能である。

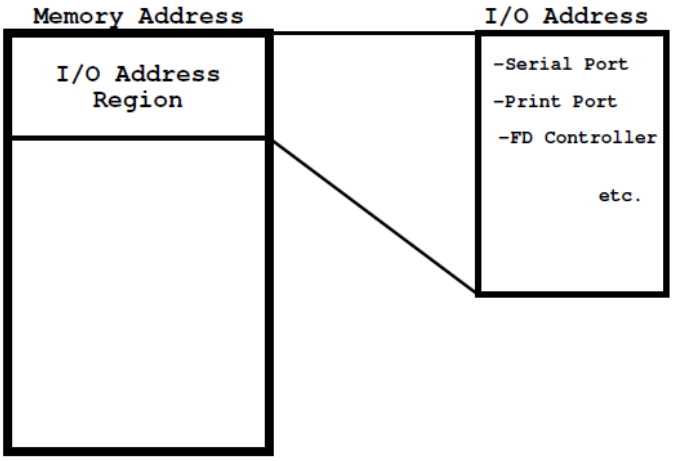

図 2.14: メモリマップド I/O 機能 [4]

SpaceWire に接続された機器はこのメモリマップド I/O 機能が使える。そのため SpW-to-GbE 側から見れば自身のメモリにアクセスするかのごとく SpaceWire の先の機器にアクセスすることが可能となる。これにより SpaceWire に繋がった機器に CPU が実装されていなくても制御通信ができる。これが RMAP の仕組みである。RMAP を用いれば CPU を持たない基盤のメモリ情報を簡単にネットワークに組み込むことができる。そのため機器間での複雑なデータ変換のためのハードウェア、ソフトウェアを省略することができ、機器のハードウェアの省スペース、干渉によるデータの損失といった心配が減ることになる。

RMAP では、RMAP Command Packet と RMPA Reply Packet を送受信し、ノードとのデータ通信を行う。RMAP Command と RMAP Reply のそれぞれの RMAP Packet における形式を Read Command を例にして図 2.15、図 2.16 に示す。

First byte transmitted

	Target SpW Address	....	Target SpW Address
Target Logical Address	Protocol Identifier	Instruction	Key
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Initiator Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended Address
Address (MS)	Address	Address	Address (LS)
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
EOP			

Last byte transmitted

図 2.15: RMAP Command Packet における形式の例 [9]

First byte transmitted

	Reply SpW Address	....	Reply SpW Address
Initiator Logical Address	Protocol Identifier	Instruction	Status
Target 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 CRC	EOP	

Last byte transmitted

図 2.16: RMAP Reply Packet における形式の例 [9]

RMAP によりノードとの制御通信を行うためには、そのためのプログラムを用いてパケットの送受信を行わければならない。このプログラムをユーザが各々で作りあげて実験を行うのでは非効率なので、宇宙研の湯浅氏をはじめとする人たちによって、RMAP 通信を行うためのソフトウェアの雛形が作られた。これが SpaceWire RMAP Library（以下、RMAPLibrary）である。RMAPLibrary は、web 上で配布されており、ユーザは自身の開発環境にこれをインストールし、作りたいモジュール部のみを作って、RMAP 通信を実行できるようになっている。RMAPLibrary は 2006 年に ver.1 の配布が開始されたが、当時は RMAP の起草時期であり、RMAPLibrary ver.1 での RMAP 実装は試験的であったため、現在では RMAPLibrary ver.1 の多くの機能が使えない状態となっている。このことから、再び湯浅氏たちによりプログラム実装が行われ、2012 年 1 月に RMAPLibrary ver.2 がリリースされた。RMAPLibrary ver.2 では、そこで定義されているクラスの全体的な再構築や、XML ファイルによる Destination 設定機能の追加などにより、従来に比べ、柔軟性・可制御性が向上した RMAP 通信を行えるようになっている。

2.2 APD Processing Module Unit (APMU)

日本の次期 X 線天文衛星 ASTRO-H は 2015 年打ち上げを予定しており、現在様々な研究機関、大学で装置開発が行われている。ASTRO-H に搭載される 4 つの検出器の中に、5-80keV のエネルギー帯域で撮像観測を行う Hard X-ray Imager(HXI) と半導体多層コンプトンカメラで 10-600keV の帯域の観測を行う Soft Gamma-ray Detector がある。この 2 つの検出器にはバックグラウンドを除去するため BGO アクティブシールドが搭載される。このシールドはそれ自体が検出器となっており、主検出器と組み合わせることによりバックグラウンドを低減させる。図 2.17 は SGD の模式図であり、中央に搭載されるコンプトンカメラとその周りに BGO アクティブシールドの BGO シンチレータが囲む構造が表わされている。BGO シンチレータは、ゲルマニウム酸ビスマス ($Bi_4Ge_3O_{12}$) という無機物質の結晶で、他のシンチレータより原子番号や比重が比較的大きく、阻止能が高い。BGO シンチレータには、複数のアバランシェフォトダイオード (以下、APD) が接続されており、これがバックグラウンドがシンチレータに落としたエネルギーを読み取る。この複数の APD から出力される信号の読み出しを APD Processing Module Unit が行う。SGD の場合、 2 枚の APMU で 14 個の APD の信号を処理し、主検出器へ制御信号が送られる。

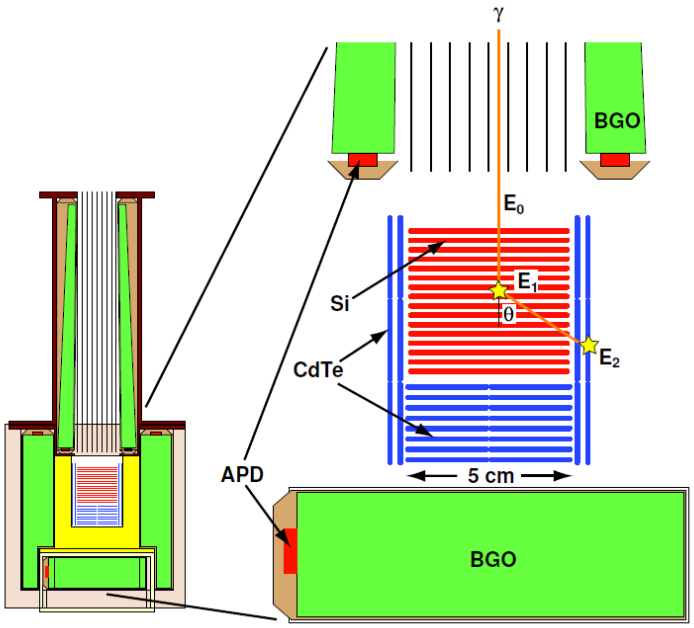

図 2.17: SGD の構造 [4]

第3章　衛星搭載検出器の開発環境の改善

私は、衛星搭載検出器の開発環境の向上を目的として、４端子 SpW-to-GbE を当研究室に設置し、SpaceWire RMAP Library を ver.2 へのアップデートを行った。ここでは、それらの導入手順について述べ、それらを実際に動作させて取得したデータを考察し、正しく動作されたかどうか動作を検証にしていく。以下で著す検証項目としては、3.1〜3.3 で、４端子 SpW-to-GbE の設置、正しく動作するかの検証、APMU を用いて行った複数ボードの動作試験を記述し、3.4 で、本研究室への SpaceWire RMAP Library ver.2 導入と、RMAPLibrary ver.2 で動作するよう作成したプログラムの動作検証について記していく。

3.1　SpaceWire-to-GigabitEther の動作試験

初めに４端子 SpW-to-GbE が動作するかどうか、パケットの送受信を行えるかどうかを確認した。動作確認として、最初は４端子 SpW-to-GbE に付属されていた「SpaceWire RMAP TEST」というサンプルプログラムを用いて、動作確認を行った。このプログラムは WindowsPC 上で動作する GUI で、簡単に SpaceWire 上にある RMAP プロトコルの Read/Write を試験できるソフトウェアである。

動作させる前に、４端子 SpW-to-GbE の IP アドレスを変更した。４端子 SpW-to-GbE の IP アドレスは web ブラウザにより変更できるようになっている。まずは、4 端子 SpW-to-GbE と外部パソコンをイーサケーブルで繋ぐ。この際外部コンピュータは、OS が windows のものを用意し、web ブラウザとして Windows Internet Explorer を使用した。４端子 SpW-to-GbE の IP アドレスはデフォルト 192.168.1.100 となっており、Windows Internet Explorer のアドレスバーにこの IP アドレスを入力すると図 3.1 が表示される。図 3.1 中央にある「Change IP address to ...」の欄に変更したい IP アドレスを入力して、４端子 SpW-to-GbE の IP アドレスを 192.168.0.141 と変更した。

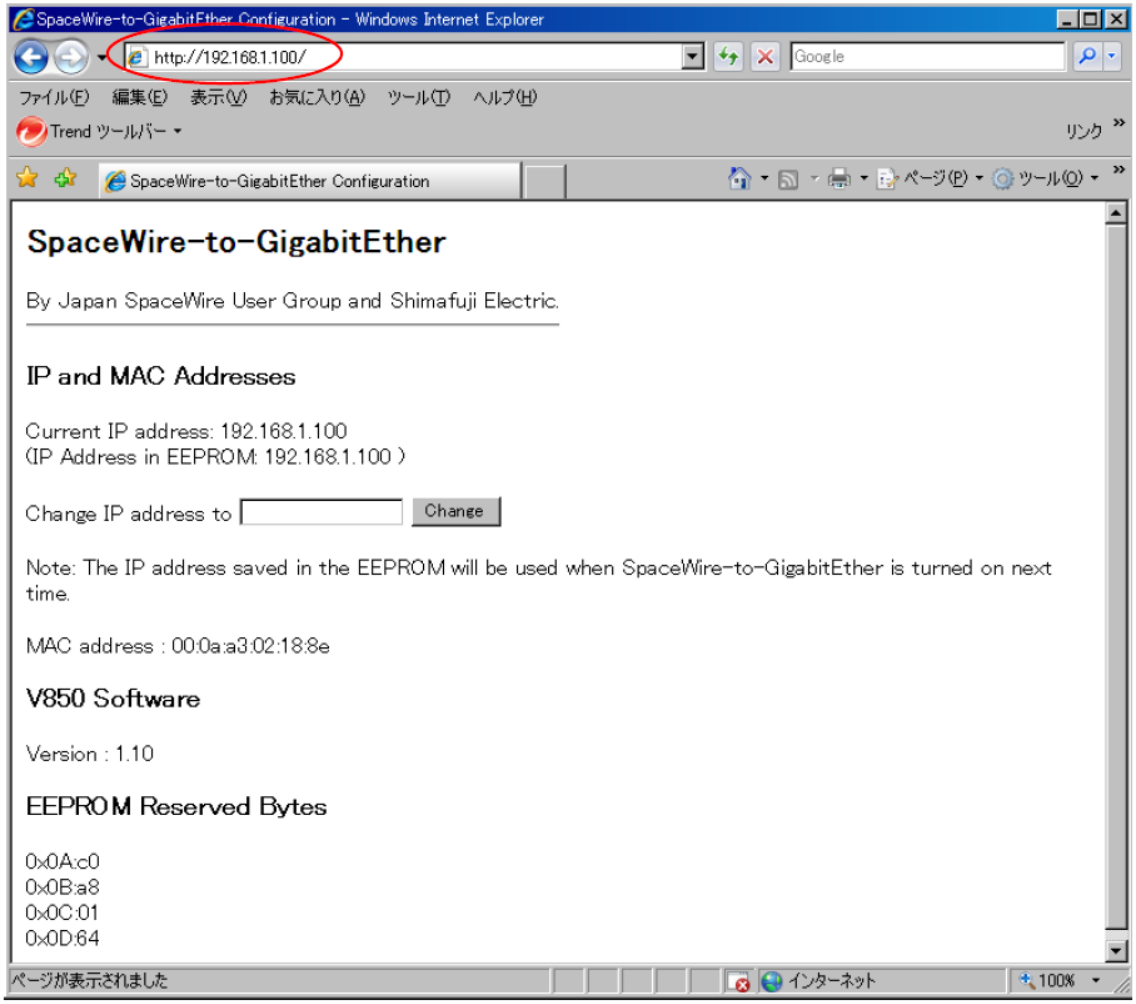

図 3.1: ４端子 SpW-to-GbE の IPaddress の変更 [10]

次に、実際に付属のサンプルプログラムで４端子 SpW-to-GbE を動作させ通信できるかどうかを検証した。図 3.2 がこの動作試験のセットアップである。Windows PC と４端子 SpW-to-GbE をイーサケーブルで接続し、FADC Board と４端子 SpW-to-GbE の SpaceWire ポート 1 を SpaceWire ケーブルで接続した。図 3.3 が、今回使用した SpaceWire ケーブルである。このケーブルは、本研究のために自作したケーブルで、図 2.2 のようにシールドコーティングはしていないが、片側が 9pin の D-Sub コネクタ、もう一方が 9pin の micro D-Sub コネクタのケーブルとなっている。また、４端子 SpW-to-GbE へ 5V、FADC Board へデジタルとアナログのそれぞれ 5V の定電圧を加えることにより起動する。

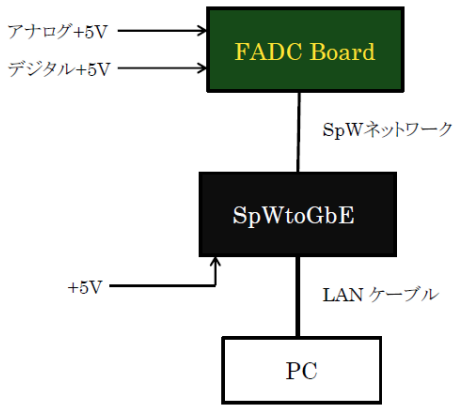

図 3.2: 動作検証のセットアップ

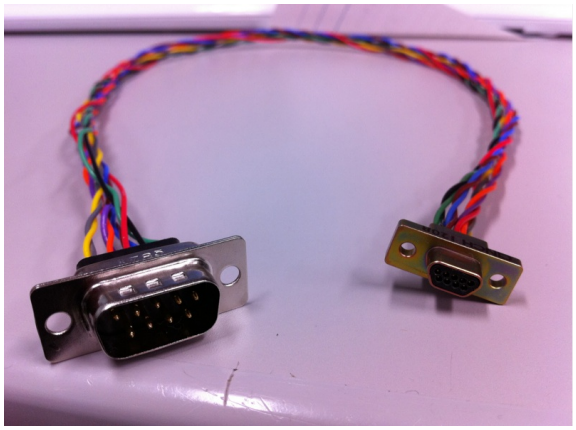

図 3.3: 本研究で使用した SpaceWire ケーブル

サンプルプログラムを起動させ、まず Ether setting を入力して connect ボタンを押し、４端子 SpW-to-GbE への通信ができたことは確認できた。次に RMAP setting を入力し、RMAP Command の Read を選択し送信したが、その応答が返って来なかった。この対処として、４端子 SpW-to-GbE 上に実装されている DIP スイッチの 6 番目を OFF から ON へ切り替え、転送クロックの設定値をデフォルトの 200MHz から 10MHz に変更した。変更後、同じように Read コマンドを送信すると、図 3.4 のような出力が返ってくるようになった。セッティングは、SpaceWire Path Address（Command Packet にのせる Path Address）を 01、Destination Logical Address を 0x30、Destination Key を 0x02、Reply Address(Reply Packet にのせる Path Address) を 0x06、Source Logical Address を 0xFE、読み取るメモリのアドレスを 0x01010040、読み取るメモリの長さを 0x000010 と設定し、Read Command を送信した。これらは 3.2 で述べるように、使用した FADC Board の仕様に合わせた値となっている。図 3.4 の中央右側が、送信した Command に対する応答である。右側の中ほどに「Return length...」とあるように、きちんと指定したノードに RMAP コマンドを送信して、データを受信できたことが確認できた。

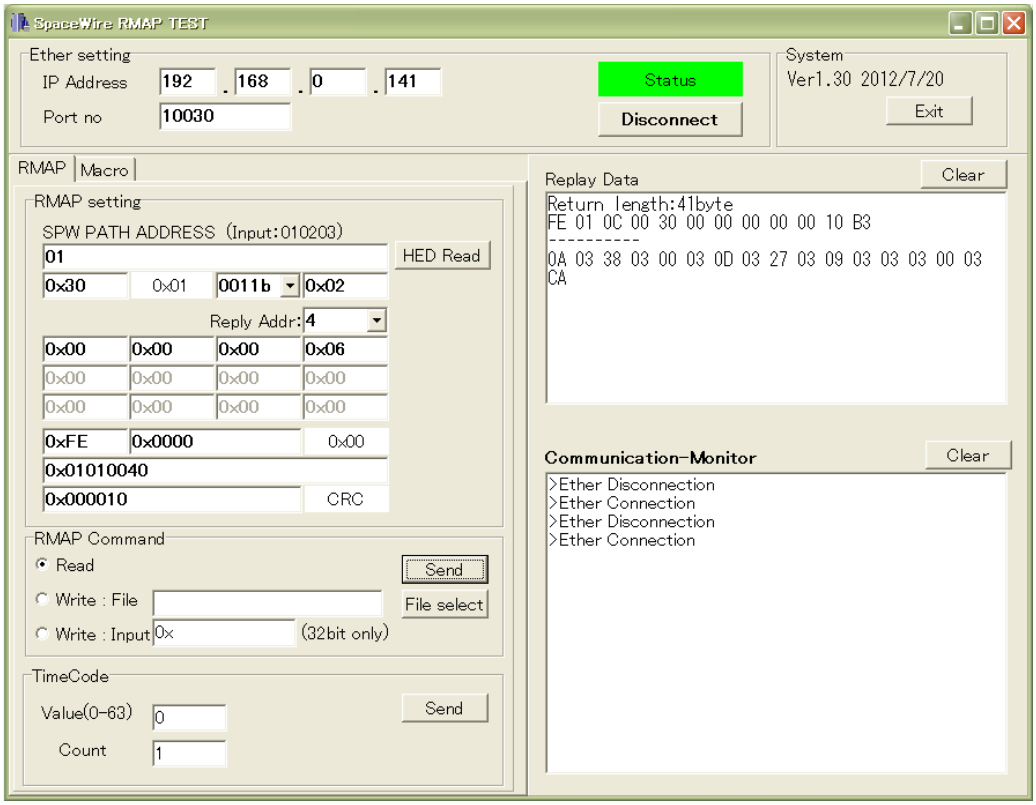

図 3.4: SpaceWire RMAP TEST を用いたメモリ値の出力

3.2 旧 RMAPLibrary による SpW-to-GbE の動作

４端子 SpW-to-GbE が動作することを確認できたので、次にデータとして受信した FADC Board のメモリ値が正しく読み取られたかどうかを詳しく検証した。検証には、RMAPLibrary(ver.1) 環境下で動かすことのできるソフトウェア「checkFADC」を用いて行う。FADC Board では、実装されている FADC によって 8 ポートに入力されたアナログ信号が 8 ポートに入力されたアナログ信号が８つのデジタル信号に変換される。このプログラムはこの８つのデジタル信号を読み取るためのソフトウェアである。

ここで、checkFADC での FADC 値の読み取りの仕組みを詳しく説明しておく。FADC Board には、波形を読み取るためのポートが 0ch から 7ch まで実装されている。それぞれのポートには ADC が繋がっており、入力される波形は Flash 型の A/D 変換がなされる。FADC 信号が一度に伝送できる情報量は 12bit である。それぞれのポートから得られた信号は２進数のデジタル値として FADC Board 上のメモリに保存される。それぞれのポートから送られてくる値が格納されるメモリアドレスは 0x01010040 から 0x0101004F までの 16byte で、図 3.5 のようになっている。16 進数でいうと FADC 値の下二桁はメモリアドレスが偶数番目のメモリに格納されており、奇数番目のメモリには FADC 値の三桁目が 4bit に格納されていて、後の 4bit は 0 で埋められている。このような設定は、VHDL 言語によって FADC Board 上の User FPGA の設定に実装されている。FADC 値は、12bit、つまり 4096 通りの情報を識別できる。入力したアナログ値が負の値も

読み取れることが望ましいので、ADC 入力の前に約 600mV のオフセットを与え、FADC 値としては 800 程度オフセットを取るという仕組みになっている。（図 3.6）

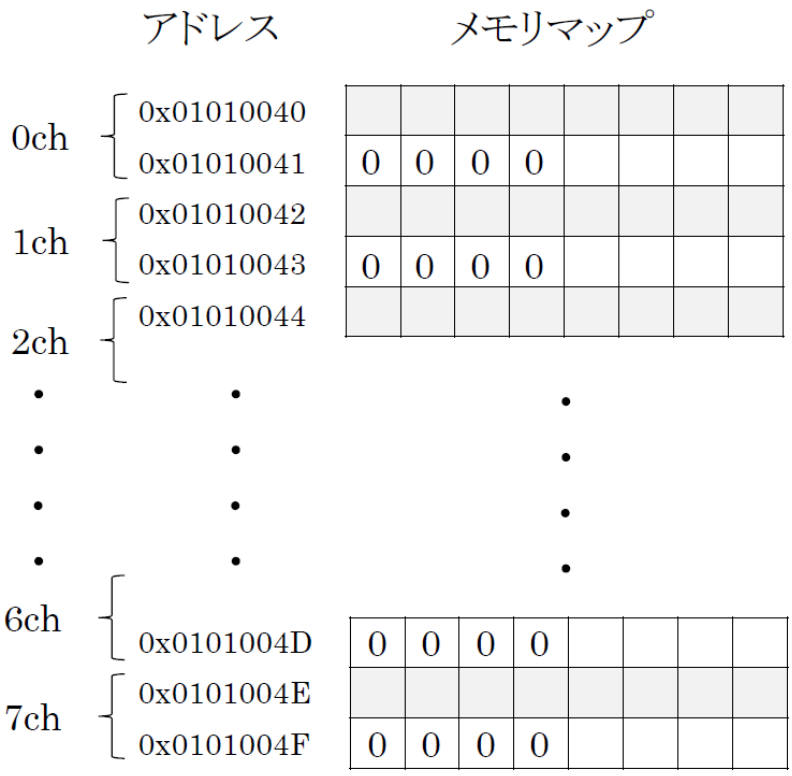

図 3.5: FADC 値が格納されるメモリマップ

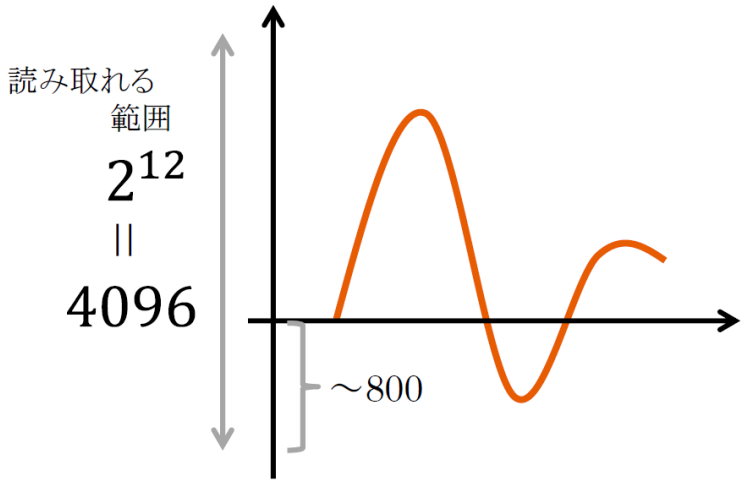

図 3.6: デジタル値に変換される FADC 波形

また、checkFADC では、FADC 値の出力以外に、FADC 値が正しく読み取られていることを保証するためのチェックも行われる。FADC Board では 0x01010030 のメモリに Write Command で 0x04 を書き込むとそれぞれのチャンネルの ADC から 0x0aaa を出力するように User FPGA が設定されている。同様に、0x01010030 のメモリに 0x08 を書き込むとそれぞれのチャンネルの ADC から 0x003f が出力されるように設定されている。このようにして、0x0aaa や 0x003f がすべてのポートから正しく出力されているか確認することで、skew が発生していないかをチェックできる。checFADC では、FADC 値を読み取る前にこれらのチェックを行う。なお、読み取ったアナログ信号の FADC 値をそのまま出力させるには、0x01010030 のメモリに 0x00 を書き込む。

この実験のセットアップは外部パソコンが OS として Linux を使用していることを除くと、前節のもの（図 3.2）と同じであり、今回の試験では FADC Board の 8 ポートの入力には何も接続されていない。checkFADC を実行すると、端末に図 3.7 の出力が表示された。図 3.7 の下部にある”reset and dump” 以下の値がメモリに格納された波形の FADC 値である。それぞれのポートから 0x0300 付近の値が出力されており、これは 10 進数で約 800 ぐらいである。この実験では、どの FADC ポートにも外部機器を接続しておらず、波高は 0 であるので、上記のように読み取られる信号としてはオフセットを加えた 800 程度と予想される。よって、図 3.7 の出力は読み取られるべき値が出力されたと言え、４端子 SpW-to-GbE を介しても正しく SpaceWire 通信が行われていると考えられる。

```
Hostname : 192.168.0.141
TCP/IP Portnumber : 10030
SSDTPModule : SSDTP2 Protocol is used.
Destination
  Logical Address   : 30
  Path Address      : 0x01
  Destination Key   : 02
  CRC Version       : Draft F
  Word Width        : 1bytes-1Word

Source
  Logical Address   : f1
  Path Address      : 0x06
  Path Address Len. : 01

Please set Destination information
checkFADC
checking deskew
ch0 is OK. 0x0aaa
ch1 is OK. 0x0aaa
ch2 is OK. 0x0aaa
ch3 is OK. 0x0aaa
ch4 is OK. 0x0aaa
ch5 is OK. 0x0aaa
ch6 is OK. 0x0aaa
ch7 is OK. 0x0aaa
checking sync
ch0 is OK. 0x003f
ch1 is OK. 0x003f
ch2 is OK. 0x003f
ch3 is OK. 0x003f
ch4 is OK. 0x003f
ch5 is OK. 0x003f
ch6 is OK. 0x003f
ch7 is OK. 0x003f
reset and dump
ch0 : 0x030c
ch1 : 0x0338
ch2 : 0x0301
ch3 : 0x030e
ch4 : 0x0327
ch5 : 0x0309
ch6 : 0x0304
ch7 : 0x0301
```

図 3.7: checkFADC の出力結果

3.3 APMU を用いた複数 Board 動作試験

前節までは、４端子 SpW-to-GbE に対して一つのノードのみの動作であった。ここでは、APMU を用いて複数ボードの制御通信を行う。

動作させるためのソフトウェアは、旧 RMAPLibrary のプログラムである「rmaphongo」を使用した。このプログラムは、コマンドラインで簡単に RMAP の read/write や、生パケットの送受信を行えるプログラムで、SpaceWire I/F 搭載 Board のロジックのデバグや、SpaceWire ネットワークの試験に利用される。

この実験のセットアップを図 3.8 に示す。４端子 SpW-to-GbE の SpaceWire ポート１と２へ、DIO-I Board と DIO-II Board を接続した。ここで、今回の実験における検証の手順を説明する。rmaphongo を実行し、まずは IP アドレスを入力して、４端子 SpW-to-GbE にアクセスする。そして、RMAP Destination を設定し、SpaceWire ポート１につながった DIO Board へ Read コマンドを送信し、指定したメモリに格納された値を読み取る。読み取れたことを確認したら、SpaceWire IF との接続を維持したまま、RMAP Destination の設定を変更し、Read コマンドでポート２に接続された DIO Board 上の指定したメモリ内の値を読み取る。このような方法によって、一つの SpaceWire インターフェイス（SpW-to-GbE）で、複数ボードへの読み出しが可能であることを検証した。

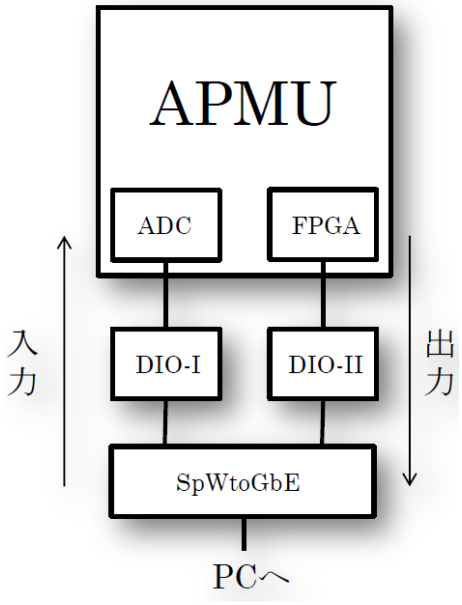

図 3.8: APMU を用いた実験セットアップ

この実験を行った当初、DIO-I Board との RMAP 通信が行えなかった。この原因は、DIO-I Borad の Logical Address が 0x01 であり、４端子 SpW-to-GbE ではサポートされていないためと考えられた。そこで、DIO-I Board に実装された SpaceWire FPGA を書き換えて Logica Address を DIO-II board と同じ 0x30 に変更した。書き換えには、DIO-II に実装されているものと同じ VHDL ファイルを使用した。これにより、DIO-I と DIO-II の違いは、接続された４端子 SpW-to-GbE のポートの違いによる Path address だけとなった。

ポート１に接続された DIO-I へアクセスするため、Path address を 0x01 に設定した（図 3.9）。そして、

Read Command を送信し、エラーなく DIO-I のメモリを読み取れたことを確認した（図 3.10）。次に、SpW-to-GbE との接続を維持した状態で、DIO-II と接続できるよう Path address を 0x02 に設定した。そして、Read Command を送信すると先と同様の出力を得られた。こうして、４端子 SpW-to-GbE へ接続を維持している間に２つのノードと制御通信に成功し、４端子 SpW-to-GbE の導入を完了した。

```
---Destination Part---
 Logical Address (hex) :  Logical Address (hex) : 30
 Path Address : (to terminate, please input a value greater than 0x100)
  [0] (hex) : 01
  [1] (hex) : 100
 Destination Key (hex) :  Destination Key (hex) : 02
 CRC Type :
  Draft E [1]
  Draft F [2]
 1 or 2 : 2
 Word Width (in byte) : 1
---Source Part---
 Logical Address (hex) :  Logical Address (hex) : fe
 Path Address : (to terminate, please input a value greater than 0x100)
  [0] (hex) : 06
  [1] (hex) : 100
 Source Path Address Length :
  Auto   [1]
  Manual [2]
 1 or 2 : 1

--- new RMAPDestination ---
Destination
  Logical Address   : 30
  Path Address      : 0x01
  Destination Key   : 02
  CRC Version       : Draft F
  Word Width        : 1bytes-1Word

Source
  Logical Address   : fe
  Path Address      : 0x06
  Path Address Len. : 01
```

図 3.9: rmaphongo の Destination 設定画面

```
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]
  TimeCode Send            [10]
  TimeCode Get             [11]
  Toggle Debug Mode [Off]   [0]
  Quit                      [9]

select > select > 1
Read (2bytes)
address in hex > 01010040
Reading...done
Read result :
0101-0040 aaaa
```

図 3.10: rmaphongo の出力画面

3.4 SpaceWire RMAP Library ver.2 の導入

ここでは、当研究室への SpaceWire RMAP Library ver.2 の導入と、RMAPLibrary ver.2 で実行可能なプログラムの作成について述べていく。

まずは、RMAPLibrary ver.2 のインストールを行った。RMAPLibrary ver.2 は、オープンソースプログラムで、宇宙研の湯浅氏のホームページからこのソースファイルをダウンロードできる。RMAPLibrary は、zip 形式で圧縮されており、これをダウンロードして、Linux PC 上の/usr/local/te/下のディレクトリに解凍し、ディレクトリ名を SpaceWireRMAPLibrary3/とした。RMAPLibrary ver.2 を扱うには、XML ファイルなどを読み込ませて解析できる機能が付いているので、それを行うための xerces c++というライブラリもインストールしなければならない。これは特に書き換える理由もないので、xerces c++のバイナリファイルをインストールした。そして、RMAPLibrary ver.2 で指定されている「SPACEWIRERMAPLIBRARY_PATH」

と「XERCESDIR」という２つのパス設定を行った。そして、端末で SpaceWireRMAPLibrary3/下の source/というディレクトリへ移動した。ここには、RMAPLibrary にもともと内蔵されているサンプルプログラムやテストプログラムのソースファイルと、それらの Makefile が置いてある。ここで、make を実行し、エラーが発生することなく、これらのソースファイルのコンパイルができたことを確認した。こうして、まず SpaceWire RMAP Library ver.2 の環境設定を行うことができた。

この段階で実行できるプログラムは、初めから内蔵されているサンプルプログラムやテストプログラムしか実行できず、我々が今までの実験で利用していたモジュールは、RMAPLibrary ver.2 に一つも存在しない。そこで、当研究室の RMAPLibrary ver.1 で実行していたプログラムと同じように動作して、RMAPLibrary ver.2 で実行できるようなアルゴリズムを構成し、そのモジュール部を作成することにした。作成するプログラムとしては、3.2 で扱った「checkFADC」を選んだ。「checkFADC」の動作手順は、3.2 で述べた通りである。

checkFADC のアルゴリズムには他のノードのメモリを読み取る工程があるので、まずは Read/Write Command のサンプルプログラムを実行し、Read が正しく行えるかどうか調べる。セットアップは、3.2 と同じ（図 3.2）で、FADC Borad も 3.2 と同じものを用いる。source/下にある Read/Write のサンプルプログラムである「tutorial_RMAPLayer.cc」を開き、中の Destination の設定と思われるコードを使用した FADC Board のものに書き換え、make した後に「tutorial_RMAPLayer」を実行した。実行すると図 3.11 の出力が返ってきた。ソースコードの RMAP Command 部分を表 3.1 にあらわす。このソースコードによると、Read/Write のコマンド、リプライパケットが正しく送受信できないとこれをエラーとして文字列化して知らせてくれるようになっている。この出力で ”RMAP Read/Write Example1 done” と表示されたということは、表 3.1 の 11 行目および 22 行目の Read/Write メソッドがによる Read/Write の RMAP Command が正しく実行されたということを表している。

```
Opening SpaceWireIF...done
Example 1
ID                      :
Target Logical Address    : 0x30
Target SpaceWire Address  : 0x01
Reply Address             : 0x00  0x00  0x00  0x06
Default Key               : 0x02


RMAP Read/Write Example1 done
```

図 3.11: tutorial_RMAPLayer の出力画面

```
1  try {
2          //case 1-1 : read using C-array as a read buffer
3          uint32_t readLength = 1024;
4          uint8_t* readData = new uint8_t[(size_t) readLength];
5          uint32_t readAddress = 0xFF801100;
6          rmapInitiator->read(rmapTargetNode1, readAddress, readLength, readData, readTimeoutDuration);
7
8          //case 1-2 : read using std::vector<uint8_t> as a read buffer
9          std::vector<uint8_t> readDataVector(readLength);
10         rmapInitiator->
11              read(rmapTargetNode1, readAddress, readLength, &(readDataVector.at(0)), readTimeoutDuration);
12
13         //case 1-3 : write using C-array write data
14         uint32_t writeAddress = 0xFF803800;
15         uint32_t writeLength = 4;
16         uint8_t* writeData = new uint8_t[writeLength];
17         writeData[0] = 0xAB;
18         writeData[1] = 0xCD;
19         writeData[2] = 0x12;
20         writeData[3] = 0x34;
21         rmapInitiator->write(rmapTargetNode1, writeAddress, writeData, writeLength, writeTimeoutDuration);
22
23         delete readData;
24         delete writeData;
25         delete rmapTargetNode1;
26
27         cout << ''RMAP Read/Write Example1 done'' << endl;
28 } catch (RMAPInitiatorException e) {
29         cerr << ''RMAPInitiatorException '' << e.toString() << endl;
30         cerr << ''Continue to next example'' << endl;
31 } catch (RMAPReplyException e) {
32         cerr << ''RMAPReplyException '' << e.toString() << endl;
33         cerr << ''Continue to next example'' << endl;
34 } catch (RMAPEngineException e) {
35         cerr << ''RMAPEngineException '' << e.toString() << endl;
36         cerr << ''Continue to next example'' << endl;
37 } catch (...) {
38         cerr << ''Unkown error'' << endl;
39         exit(-1);
40 }
```

表 3.1: tutorial_RMAPLayer.cc の RMAP Command 部分 [8]

次に、Read Command により読み取った値を出力することを考える。「tutorial_RMAPLayer.cc」のソースコードには、読み取った値を出力するためのコードが記述されておらず、checkFADC と同じアルゴリズムを考えるのであれば、Read メソッドにより読み取られた値を出力しなければならない。なお、このソースコードでは、read バッファとして配列を用いるか (2〜6 行目)、vector を用いるか（8〜11 行目)、２通りの方法で Read Command が記述されているが、今回は配列による方法で Read のデータを出力することとする。ソースコードによると、5 行目で定義された”readData” という配列の中に、read メソッドによって読みとった値が格納されると考えられる。まずは、単純にこの配列の先頭の値を出力させてみる。

```
cout << hex << readData[0] << endl;
```

という配列の出力を行うコードを 7 行目の read メソッドの後に書き込み出力する。すると文字化けした読めない文字が出力された。5 行目、“readData” の定義を詳しく見てみると、配列が動的メモリで確保されているが、配列数を指定する”readLength” の型が

```
[(size_t) readLength]
```

というコードで size_t 型に変更されている。size_t 型は、sizeof という演算子によって得られる値の型である。6 行目にある read メソッドの引数として用いている変数、ポインタは “readTimeoutDuration” 以外の全

てが uint(hoge)_t という型で定義されており、わざわざ型を変更しているところから、char 型や int 型とは異なる型であると考えられる。配列 “readData” は、この uint8_t 型で定義されており、これを出力しようとしているのでうまく機能していないのである。そこで、

```
cout << hex << (size_t) readData[0] << endl;
```

というように出力するコードを挿入し、これを実行すると数字が返ってくるようになった。これから、0x01010040 のメモリに格納された値を 16 進数で出力することができたと思われる。また、14〜22 行目の Write Command に関するコードで 0x01010030 のメモリ値を 08 に書き換えた後、0x01010040 の値を出力すると”3f” が出力された。これは、3.2 で動作させた checFADC の checking sync で出力された値の下二桁であり（図 3.7）、checing sync 機能が正しく実行されたことを確認した。

これらの結果に基づき、checkFADC と同じような機能を持つようアルゴリズムを構成した。表 3.2 が作成した RMAPLibrary ver.2 で動作する新 checkFADC のソースコードである。従来の checkFADC では、実行するアルゴリズムを全てオブジェクトの中で指定し、実行時にそのメンバ関数として呼び出すように記述されていたが、この checkFADC はそれらを全てメイン関数の中で記述している。また、新しい checkFADC は、従来型が持っていないエラー処理の機能も付いている。実行の前に、ユーザは SpW-to-GbE と接続をするための IP アドレスによる SpaceWire I/F 設定（表 3.2：33 行目）と、SpaceWire ネットワーク内のノードにアクセスするための Destination 設定（表 3.2：52〜67 行目）を行う必要がある。このプログラムで使用するクラスのうち、RMAPInitiator は、Read/Write メソッドを持つクラスである RMAPLibrary ver.1 の RMAPSocket に対応しており、RMAPTargetNode はノードの Destination 設定を行うクラスである ver.1 の RMAPDestination に対応しているが、SpaceWireIF や RMAPEngine は従来型と同じように扱われている。この新しい checkFADC を実行した時の出力が図 3.12 である。ch0 から ch7 にかけて、FADC 値は入力されているアナログ値のゆらぎによってゆらぐのが自然である。図 3.7 と図 3.12 を比較すると各チャンネルの FADC 値も同じようにゆらいでいることが見て取れる。加えて、正しく checking deskew 機能と checking sync 機能も動作していることが確認できる。これらのことから、新しく作成した checkFADC は、RMAPLibrary ver.1 で扱っていたそれと同じ機能を達成したことが言える。

```
1  /*
2   *      checkFADC.cc
3   *
4   *  Created on: Feb 2, 2013
5   *      Author: inoue
6   *
7   *  This program is based on
8   *  ``tutorial_RMAPLayer.cc''
9   *     created by Mr.yuasa.
10  */
11
12 #include ``SpaceWire.hh''
13 #include ``RMAP.hh''
14
15 double readTimeoutDuration = 1000;//1000ms
16 double writeTimeoutDuration = 1000;//1000ms
17
18 int main() {
19         using namespace std;
20
21         cout << endl;
22         cout << ``###############################################'' << endl;
23         cout << ``###     SpaceWire program : checkFADC    ###'' << endl;
```

```
24          cout << ``###           ver 1 (20130202)           ###'' << endl;
25          cout << ``###                                      ###'' << endl;
26          cout << ``###   This program is totaly based on    ###'' << endl;
27          cout << ``###       SpaceWire RMAP Library 2.      ###'' << endl;
28          cout << ``##########################################'' << endl << endl;
29
30
31          /* Open the SpaceWire interface */
32          cout << ``Opening SpaceWireIF...'';
33          SpaceWireIF* spwif = new SpaceWireIFOverIPClient(``192.168.0.141'', 10030);
34          try {
35                  spwif->open();
36          } catch (...) {
37                  cerr << ``Connection timed out.'' << endl;
38                  exit(-1);
39          }
40          cout << ``done'' << endl << endl;
41
42          /* Construct and start RMAP Engine */
43          RMAPEngine* rmapEngine = new RMAPEngine(spwif);
44          rmapEngine->start();
45
46          /* Construct an RMAP Initiator instance */
47          RMAPInitiator* rmapInitiator = new RMAPInitiator(rmapEngine);
48          rmapInitiator->setInitiatorLogicalAddress(0xFE);
49
50          ////////////////////////////////////////////////////////////////////////////////
51
52          /* Manually sets RMAPTargetNode information */
53          RMAPTargetNode* rmapTargetNode1 = new RMAPTargetNode();
54          rmapTargetNode1->setTargetLogicalAddress(0x30);
55          rmapTargetNode1->setDefaultKey(0x02);
56          std::vector<uint8_t> targetSpaceWireAddress;
57          targetSpaceWireAddress.push_back(0x01);
58          //targetSpaceWireAddress.push_back(0x0a);
59          //targetSpaceWireAddress.push_back(0x05);
60          rmapTargetNode1->setTargetSpaceWireAddress(targetSpaceWireAddress);
61          std::vector<uint8_t> replyAddress;
62          //replyAddress.push_back(0x00);
63          //replyAddress.push_back(0x00);
64          //replyAddress.push_back(0x00);
65          replyAddress.push_back(0x06);
66          rmapTargetNode1->setReplyAddress(replyAddress);
67          cout << rmapTargetNode1->toString() << endl;
68
69
70          try {
71
72          /*check deskew*/
73                  uint32_t writeAddress = 0x01010030;
74                  uint32_t writeLength = 2;
75                  uint8_t* writeData = new uint8_t[writeLength];
76                  writeData[0] = 0x04;
77                  writeData[1] = 0x00;
78                  rmapInitiator->
79                      write(rmapTargetNode1, writeAddress, writeData, writeLength, writeTimeoutDuration);
80
81                  uint32_t readLength = 16;
82                  uint8_t* readData = new uint8_t[(size_t) readLength];
83                  uint32_t readAddress = 0x01010040;
84                  rmapInitiator->
85                      read(rmapTargetNode1, readAddress, readLength, readData, readTimeoutDuration);
86
87                  cout << ``check deskew'' << endl;
88                  for(int i=0; i<=7; i++){
89                    cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
90                         << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
91                         << (size_t)readData[2*i] << endl;
92                  }
93                  cout << endl;
94
95                  delete readData;
96                  delete writeData;
```

```
97
98
99          /*check sync*/
100                 writeAddress = 0x01010030;
101                 writeLength = 2;
102                 writeData = new uint8_t[writeLength];
103                 writeData[0] = 0x08;
104                 writeData[1] = 0x00;
105                 rmapInitiator->
106                     write(rmapTargetNode1, writeAddress, writeData, writeLength, writeTimeoutDuration);
107
108                 readLength = 16;
109                 readData = new uint8_t[(size_t) readLength];
110                 readAddress = 0x01010040;
111                 rmapInitiator->
112                     read(rmapTargetNode1, readAddress, readLength, readData,readTimeoutDuration);
113
114
115                 cout << ``check sync'' << endl;
116                 for(int i=0; i<=7; i++){
117                   cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
118                        << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
119                        << (size_t)readData[2*i] << endl;
120                 }
121                 cout << endl;
122
123                 delete readData;
124                 delete writeData;
125
126
127         /*reset and dump*/
128                 writeAddress = 0x01010030;
129                 writeLength = 2;
130                 writeData = new uint8_t[writeLength];
131                 writeData[0] = 0x00;
132                 writeData[1] = 0x00;
133                 rmapInitiator->
134                     write(rmapTargetNode1, writeAddress, writeData,  writeLength, writeTimeoutDuration);
135
136                 readLength = 16;
137                 readData = new uint8_t[(size_t) readLength];
138                 readAddress = 0x01010040;
139                 rmapInitiator->
140                     read(rmapTargetNode1, readAddress, readLength, readData, readTimeoutDuration);
141
142                 cout << ``reset and dump'' << endl;
143                 for(int i=0; i<=7; i++){
144                   cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
145                        << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
146                        << (size_t)readData[2*i] << endl;
147                 }
148
149                 delete readData;
150                 delete writeData;
151                 delete rmapTargetNode1;
152
153                 cout << endl;
154
155         } catch (RMAPInitiatorException e) {
156                 cerr << ``RMAPInitiatorException `` << e.toString() << endl;
157         } catch (RMAPReplyException e) {
158                 cerr << ``RMAPReplyException `` << e.toString() << endl;
159         } catch (RMAPEngineException e) {
160                 cerr << ``RMAPEngineException `` << e.toString() << endl;
161         } catch (...) {
162                 cerr << ``Unkown error'' << endl;
163                 exit(-1);
164         }
165
166
167 }
168
169
```

```
170 /** History
171  * 2013-02-02 file create (Shota Inoue)
172  */
```

表 3.2: 新しい checkFADC のソースコード

```
##############################################
###     SpaceWire program : checkFADC     ###
###           ver 1 (20130202)            ###
###                                       ###
###   This program is totaly based on     ###
###       SpaceWire RMAP Library 2.       ###
##############################################

Opening SpaceWireIF...done

ID                          :
Target Logical Address      : 0x30
Target SpaceWire Address    : 0x01
Reply Address               : 0x06
Default Key                 : 0x02


check deskew
ch0 : 0x0aaa
ch1 : 0x0aaa
ch2 : 0x0aaa
ch3 : 0x0aaa
ch4 : 0x0aaa
ch5 : 0x0aaa
ch6 : 0x0aaa
ch7 : 0x0aaa

check sync
ch0 : 0x003f
ch1 : 0x003f
ch2 : 0x003f
ch3 : 0x003f
ch4 : 0x003f
ch5 : 0x003f
ch6 : 0x003f
ch7 : 0x003f

reset and dump
ch0 : 0x0309
ch1 : 0x033b
ch2 : 0x0302
ch3 : 0x0312
ch4 : 0x032b
ch5 : 0x030c
ch6 : 0x0309
ch7 : 0x0304
```

図 3.12: 新しい checkFADC の出力画面

3.5 マルチトランザクションによる複数機器同時読み出し

前述の通り、RMAP は Command Packet と Reply Packet の送受信で実行される「トランザクション」を一つの単位として動作する。RMAPLibrary ver.2 では、通信を行うノードの Destination 設定を XML ファイルにより設定することができる。XML ファイルにより Destination 設定することで、複数のノード、または複数のメモリを同時に設定することができる。表 3.3 が XML ファイルでの Destination 設定例である。XML ファイルで Destination の設定を行うと複数のトランザクションを同時に実行することができる。このマルチトランザクションを利用できれば、先で導入した 4 端子 SpW-to-GbE と組み合わせることで複数機器との同時に制御通信が行える。そこで、マルチトランザクションを可能とするプログラムの作成を行った。

```
1
2  <RMAPTargetNode id=''ShimafujiDIO''>
3          <TargetLogicalAddress>0x30</TargetLogicalAddress>
4          <TargetSpaceWireAddress></TargetSpaceWireAddress>
5          <ReplyAddress></ReplyAddress>
6          <Key>0x02</Key>
7
8          <RMAPMemoryObject id=''Command''>
9                  <ExtendedAddress>0x00</ExtendedAddress>
10                 <Address>0xFF803800</Address>
11                 <Length>0x10</Length>
12                 <Key>0x25</Key>
13                 <AccessMode>WriteOnly</AccessMode>
14         </RMAPMemoryObject>
15
16         <RMAPMemoryObject id=''EssentialHK''>
17                 <ExtendedAddress>0x00</ExtendedAddress>
18                 <Address>0xFF801100</Address>
19                 <Length>0x08</Length>
20                 <Key>0x20</Key>
21                 <AccessMode>ReadOnly</AccessMode>
22         </RMAPMemoryObject>
23 </RMAPTargetNode>
24
25 <RMAPTargetNode id=''ShimafujiDIO2''>
26         <TargetLogicalAddress>0x31</TargetLogicalAddress>
27         <TargetSpaceWireAddress></TargetSpaceWireAddress>
28         <ReplyAddress></ReplyAddress>
29         <Key>0x02</Key>
30
31         <RMAPMemoryObject id=''Command''>
32                 <ExtendedAddress>0x00</ExtendedAddress>
33                 <Address>0xFF803800</Address>
34                 <Length>0x10</Length>
35                 <Key>0x25</Key>
36                 <AccessMode>WriteOnly</AccessMode>
37         </RMAPMemoryObject>
38
39         <RMAPMemoryObject id=''EssentialHK''>
40                 <ExtendedAddress>0x00</ExtendedAddress>
41                 <Address>0xFF801100</Address>
42                 <Length>0x08</Length>
43                 <Key>0x20</Key>
44                 <AccessMode>ReadOnly</AccessMode>
45         </RMAPMemoryObject>
46 </RMAPTargetNode>
47
48 </root>
```

表 3.3: XML ファイルでの Destination 設定例 [8]

まずは、3.4 と同様にまずはサンプルプログラムで XML ファイルの読み出し方法を調べた。ここで扱う

サンプルプログラムは、Read/Write コマンドを送信するプログラムで、3.4 のサンプルプログラムとほぼ同じであるが、Target Node の設定部分（表 3.4）と Read/Write メソッドの仮引数の部分（表 3.5）が異なっている。XML ファイルは、argc 変数、argv 変数の設定によりコマンドライン引数で入力され、その情報は rmapTargetNodeDB に渡され、最終的に rmapInitiator に渡される。そして、RMAPInitiator クラスの Read/Write メソッドを呼び出す。この際、仮引数の１番目に RMAPTargetNode id、２番目に RMAPMemoryObject id を入力するという形になっている。メソッドはこれらの id を設定した XML ファイルから探して、同じ id で設定されているノードのメモリを Read/Write し、XML ファイルの中に id を見つけれなければエラーが発生する。XML ファイルを正しく設定し実行すると、3.4 と同様に”RMAP Read/Write Example2 done” の文字が表示され、XML ファイルを扱った Read/Write に成功した。

```
42 std::string filename(argv[1]);
43 cout << ``Constructing RMAPTargetNodes from `` << filename << endl;
44 RMAPTargetNodeDB* rmapTargetNodeDB;
45 try {
46 rmapTargetNodeDB = new RMAPTargetNodeDB(filename);
47 } catch (RMAPTargetNodeDBException e) {
48 cerr << ``An exception thrown while loading the XML file `` << filename
49 << endl;
50 cerr << e.toString() << endl;
51 exit(-1);
52 }
53
54 //check the number of entries
55 if (rmapTargetNodeDB->getSize() == 0) {
56 cerr << ``No RMAPTargetNode instance was constructed...'' << endl;
57 exit(-1);
58 }
59
60 //set the db to RMAPInitiator
61 rmapInitiator->setRMAPTargetNodeDB(rmapTargetNodeDB);
```

表 3.4: サンプルプログラムの XML ファイル読み出し部分 [8]

```
62 uint32_t readLength = 2;
63 uint8_t* readData = new uint8_t[(size_t) readLength];
64 rmapInitiator->read(``SpaceWireDigitalIOBoard'', ``LEDRegister'',
65 readData, readTimeoutDuration);
66
67 //case 1-2 : read using std::vector<uint8_t> as a read buffer
68 std::vector<uint8_t> readDataVector(readLength);
69 rmapInitiator->read(``SpaceWireDigitalIOBoard'', ``LEDRegister'',
70 \&(readDataVector.at(0)), readTimeoutDuration);
71
72 //case 1-3 : write using C-array write data
73 uint32_t writeLength = 2;
74 uint8_t* writeData = new uint8_t[writeLength];
75 writeData[0] = 0xFF;
76 writeData[1] = 0xFF;
77 rmapInitiator->write(``SpaceWireDigitalIOBoard'', ``LEDRegister'',
78 writeData, writeTimeoutDuration);
```

表 3.5: XML ファイルで設定する場合の Read/Write メソッド [8]

次にこれらの結果を踏まえ、マルチトランザクションを実行するプログラムとして、「Multi_checkFADC」というプログラムを開発した。このプログラムは、XML ファイルの読み出しコードと 3.4 で作成した checkFADC

のコードが複合されており、XML ファイルのマルチトランザクションにより複数の FADC Board で FADC 値を同時に読み出しを実行できる。表 3.6 が Multi_checkFADC のソースコードで、表 3.7 がこのプログラム実行にあたって準備する XML ファイルの設定例である。ユーザは一つの FADC Board の数だけ表 3.7 のファイルを用意し、一つの XML ファイルに対して通信する FADC/ Board の Destination 設定を一つだけする。XML ファイルにおいてユーザが行う設定は、<TargetLogicalAddress>、<TargetSpaceWireAddress>、<ReplyAddress>、<Key> の値の設定のみで、それ以外の部分を編集すると Multi_checkFADC の実行はエラーとなる。Multi_checkFADC の実行は、用意した XML ファイルをコマンドライン引数として全て入力して行い、checkFADC の一連のアルゴリズムを XML ファイルごとにループさせる。Destination 設定を行った 2 つの XML ファイルを準備し、

```
[/SYS]% ./tutorial_RMAPLayer  ./sampleXML/SampleRMAPTargetNode_001.xml
        ./sampleXML/SampleRMAPTargetNode_002.xml
```

というコマンドを端末で実行した時の出力結果が表 3.8 である。2 つのノードからの FADC 値が正しく読めている事が確認でき、マルチトランザクションに成功したと言える。1 台の FADC/ Board で 8 ポート FADC がこの Multi_checkFADC と 3.1〜3.3 の 4 端子 SpaceWire-to-GbE と組み合わせると原理的に 32 ポートからの同時読み出しが可能となったことになる。こうして、我々の研究室で、従来試みたことのないマルチトランザクションが RMAPLibrary ver.2 により簡単に扱えるようになった。この Multi_checkFADC での FADC 値同時読み出しはあくまでマルチトランザクションの簡単な一例であり、id やメソッドを上手く利用することで複数ノードと同時通信を行う様々なプログラムを組むことができる。積極的にマルチトランザクションを利用していく事により、更なる衛星搭載機器開発環境の向上が期待される。

```
79  *    Multi_checkFADC.cc
80  *
81  *  Created on: Feb 7, 2013
82  *      Author: inoue
83  *
84  *  This program is based on
85  *  ``tutorial_RMAPLayer.cc''
86  *    created by Mr.yuasa.
87  */
88
89  #include ``SpaceWire.hh''
90  #include ``RMAP.hh''
91
92  double readTimeoutDuration = 1000;//1000ms
93  double writeTimeoutDuration = 1000;//1000ms
94
95  int main(int argc, char* argv[]) {
96          using namespace std;
97
98          cout << endl;
99          cout << ``##########################################'' << endl;
100         cout << ``###     SpaceWire program : checkFADC    ###'' << endl;
101         cout << ``###           ver 1 (20130207)           ###'' << endl;
102         cout << ``###                                      ###'' << endl;
103         cout << ``###   This program is totaly based on    ###'' << endl;
104         cout << ``###       SpaceWire RMAP Library 2.      ###'' << endl;
105         cout << ``##########################################'' << endl << endl;
106
107         /* Open the SpaceWire interface */
108         cout << ``Opening SpaceWireIF...'';
109         SpaceWireIF* spwif = new SpaceWireIFOverIPClient(``192.168.0.131'', 10030);
110         try {
111                 spwif->open();
112         } catch (...) {
113                 cerr << ``Connection timed out.'' << endl;
```

```
114             exit(-1);
115         }
116         cout << ``done'' << endl;
117 
118         /* Construct and start RMAP Engine */
119         RMAPEngine* rmapEngine = new RMAPEngine(spwif);
120         rmapEngine->start();
121 
122         /* Construct an RMAP Initiator instance */
123         RMAPInitiator* rmapInitiator = new RMAPInitiator(rmapEngine);
124         rmapInitiator->setInitiatorLogicalAddress(0xFE);
125 
126         ////////////////////////////////////////////////////////////////////////////////
127 
128         /* Use RMAPTargetNodes constructed from an XML-like configuration file. */
129         cout << endl;
130 
131         if (argc < 2) {
132             cerr << ``Example2 requires an XML-like configuration file.'' << endl;
133             exit(-1);
134         }
135 
136         for (int j=1; j<argc; j++){
137         //check file existence
138         if (!CxxUtilities::File::exists(argv[j])) {
139             cerr << ``File `` << argv[j] << `` does not exist.'' << endl;
140             if (j < argc-1){
141               cerr << ``Continue to next XML file'' << endl << endl;
142             }
143         }
144 
145         else{
146         //construct RMAPTargetNodes from the XML file
147         std::string filename(argv[j]);
148         cout << ``Constructing RMAPTargetNodes from `` << filename << endl;
149         RMAPTargetNodeDB* rmapTargetNodeDB;
150         try {
151             rmapTargetNodeDB = new RMAPTargetNodeDB(filename);
152         } catch (RMAPTargetNodeDBException e) {
153             cerr << ``An exception thrown while loading the XML file `` << filename << endl;
154             cerr << e.toString() << endl;
155             exit(-1);
156         }
157 
158         //check the number of entries
159         if (rmapTargetNodeDB->getSize() == 0) {
160             cerr << ``No RMAPTargetNode instance was constructed...'' << endl;
161             exit(-1);
162         }
163 
164         //set the db to RMAPInitiator
165         rmapInitiator->setRMAPTargetNodeDB(rmapTargetNodeDB);
166 
167         /* RMAP Read/Write with address/length */
168         try {
169 
170           /*check deskew*/
171             uint32_t writeLength = 2;
172             uint8_t* writeData = new uint8_t[writeLength];
173             writeData[0] = 0x04;
174             writeData[1] = 0x00;
175             rmapInitiator->write(``FADCBoard'', ``WriteMemory'', writeData, writeTimeoutDuration);
176 
177 
178             uint32_t readLength = 16;
179             uint8_t* readData = new uint8_t[(size_t) readLength];
180             rmapInitiator->read(``FADCBoard'', ``ReadMemory'', readData, readTimeoutDuration);
181 
182             cout << ``check deskew'' << endl;
183             for(int i=0; i<=7; i++){
184               cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
185                    << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
186                    << (size_t)readData[2*i] << endl;
```

```
187                 }
188                 cout << endl;
189
190                 delete readData;
191                 delete writeData;
192
193           /*check sync*/
194                 writeLength = 2;
195                 writeData = new uint8_t[writeLength];
196                 writeData[0] = 0x08;
197                 writeData[1] = 0x00;
198                 rmapInitiator->write(``FADCBoard'', ``WriteMemory'', writeData, writeTimeoutDuration);
199
200
201                 readLength = 16;
202                 readData = new uint8_t[(size_t) readLength];
203                 rmapInitiator->read(``FADCBoard'', ``ReadMemory'', readData, readTimeoutDuration);
204
205                 cout << ``check sync'' << endl;
206                 for(int i=0; i<=7; i++){
207                   cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
208                        << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
209                        << (size_t)readData[2*i] << endl;
210                 }
211                 cout << endl;
212
213                 delete readData;
214                 delete writeData;
215
216           /*reset and dump*/
217                 writeLength = 2;
218                 writeData = new uint8_t[writeLength];
219                 writeData[0] = 0x00;
220                 writeData[1] = 0x00;
221                 rmapInitiator->write(``FADCBoard'', ``WriteMemory'', writeData, writeTimeoutDuration);
222
223
224                 readLength = 16;
225                 readData = new uint8_t[(size_t) readLength];
226                 rmapInitiator->read(``FADCBoard'', ``ReadMemory'', readData, readTimeoutDuration);
227
228                 cout << ``reset and dump'' << endl;
229                 for(int i=0; i<=7; i++){
230                   cout << ``ch'' << i << `` : 0x'' << setfill('0') << hex << setw(2)
231                        << (size_t)readData[2*i+1] << setfill('0') << hex << setw(2)
232                        << (size_t)readData[2*i] << endl;
233                 }
234                 cout << endl;
235
236                 delete readData;
237                 delete writeData;
238
239                 cout << ``FADC for `` << filename << `` have read.'' << endl;
240
241         } catch (RMAPInitiatorException e) {
242                 cerr << ``RMAPInitiatorException `` << e.toString() << endl;
243                 if (j < argc-1){
244                 cerr << ``Continue to next XML file'' << endl;
245         }
246         } catch (RMAPReplyException e) {
247                 cerr << ``RMAPReplyException `` << e.toString() << endl;
248                 if (j < argc-1){
249                 cerr << ``Continue to next XML file'' << endl;
250         }
251         } catch (RMAPEngineException e) {
252                 cerr << ``RMAPEngineException `` << e.toString() << endl;
253                 if (j < argc-1){
254                 cerr << ``Continue to next XML file'' << endl;
255         }
256         } catch (...) {
257                 cerr << ``Unkown error'' << endl;
258                 exit(-1);
259         }
```

```
260
261         if (j < argc-1){
262           cout << ``Continue to next XML file'' << endl << endl;
263         }
264         }
265         }
266         ////////////////////////////////////////////////////////////////////////////////
267
268 }
```

表 3.6: Multi_checkFADC のソースコード

```
49
50 <RMAPTargetNode id=''FADCBoard''>
51         <TargetLogicalAddress>0x30</TargetLogicalAddress>
52         <TargetSpaceWireAddress></TargetSpaceWireAddress>
53         <ReplyAddress></ReplyAddress>
54         <Key>0x02</Key>
55
56         <RMAPMemoryObject id=''WriteMemory''>
57                 <ExtendedAddress>0x00</ExtendedAddress>
58                 <Address>0x01010030</Address>
59                 <Length>0x000002</Length>
60         </RMAPMemoryObject>
61
62         <RMAPMemoryObject id=''ReadMemory''>
63                 <ExtendedAddress>0x00</ExtendedAddress>
64                 <Address>0x01010040</Address>
65                 <Length>0x000016</Length>
66         </RMAPMemoryObject>
67
68 </RMAPTargetNode>
69
70 </root>
```

表 3.7: Multi_checkFADC のための XML ファイル

```
1  ##########################################
2  ###     SpaceWire program : checkFADC    ###
3  ###           ver 1 (20130202)           ###
4  ###                                      ###
5  ###   This program is totaly based on    ###
6  ###       SpaceWire RMAP Library 2.      ###
7  ##########################################
8
9  Opening SpaceWireIF...done
10
11 Constructing RMAPTargetNodes from sampleXML/SampleRMAPTargetNode_001.xml
12 check deskew
13 ch0 : 0x0aaa
14 ch1 : 0x0aaa
15 ch2 : 0x0aaa
16 ch3 : 0x0aaa
17 ch4 : 0x0aaa
18 ch5 : 0x0aaa
19 ch6 : 0x0aaa
20 ch7 : 0x0aaa
21
```

```
check sync
ch0 : 0x003f
ch1 : 0x003f
ch2 : 0x003f
ch3 : 0x003f
ch4 : 0x003f
ch5 : 0x003f
ch6 : 0x003f
ch7 : 0x003f

reset and dump
ch0 : 0x030a
ch1 : 0x0338
ch2 : 0x02ff
ch3 : 0x030d
ch4 : 0x0327
ch5 : 0x0308
ch6 : 0x0305
ch7 : 0x0301

FADC for sampleXML/SampleRMAPTargetNode_001.xml have read.
Continue to next XML file

Constructing RMAPTargetNodes from sampleXML/SampleRMAPTargetNode_002.xml
check deskew
ch0 : 0x0aaa
ch1 : 0x0aaa
ch2 : 0x0aaa
ch3 : 0x0aaa
ch4 : 0x0aaa
ch5 : 0x0aaa
ch6 : 0x0aaa
ch7 : 0x0aaa

check sync
ch0 : 0x003f
ch1 : 0x003f
ch2 : 0x003f
ch3 : 0x003f
ch4 : 0x003f
ch5 : 0x003f
ch6 : 0x003f
ch7 : 0x003f

reset and dump
ch0 : 0x030b
ch1 : 0x0338
ch2 : 0x02ff
ch3 : 0x030d
ch4 : 0x0329
ch5 : 0x0308
ch6 : 0x0304
ch7 : 0x0301

FADC for sampleXML/SampleRMAPTargetNode_002.xml have read.
```

表 3.8: Multi_checkFADC の出力例

第4章　まとめと今後

本研究では、衛星搭載機器の開発環境の向上を目的として、以下の２つの項目に取り組んだ。

- 4 端子 SpaceWire-to-GigabitEther の導入

  従来、SpaceWire 開発環境で用いられる SpaceWire I/F 搭載 Board と自身のパーソナルコンピュータでデータ通信を行う際、一つボードに対し、一つの SpW-to-GbE が必要であった。私は、本研究でルータ機能の付いたフジシマ電機製スタンドアロン版 SpaceWire-to-GigabitEther を当研究室へ導入することにより、１つの SpW-to-GbE で４つまでのボードを接続できる開発環境を構築した。動作の確認は、Windows 用ソフトウェア「SpaceWire RMAP TEST」での動作、旧 RMAPLibrary のプログラムでの動作を通して正しく動作することを確認した。また、APMU に繋がった複数のボードを４端子 SpW-to-GbE へ接続し、１つの SpaceWire インターフェイスで複数ボードと制御通信することに成功した。

- SpaceWire RMAP Library 環境のアップデート

  2012 年 1 月に SpaceWire RMAP Library ver.2 がリリースされた。これは、従来のそれにはない様々な機能が追加されており、RMAP 制御通信の柔軟性、可制御性を向上させる。私は本研究でこの SpaceWire RMAP Library ver.2 を導入することにより、当研究室の衛星搭載機器開発環境を最新の RMAP プロトコル環境へアップデートした。そして、従来のライブラリで扱っていた「checkFADC」と同じ機能を持ち RMAPLibrary ver.2 環境下で動作する新しいソフトウェアを作り、そのプログラムが正常に機能するかを検証した。そして、RMAPLibrary ver.2 の新しい機能の一つである XML ファイルによる入力機能を用いて、複数機器のマルチトランザクションを実現する新たなプログラムの作成に成功した。そして、開発したマルチトランザクションプログラムとを先に導入した４端子 SpW-to-GigabitEther を組み合わせることにより、FADC 信号の 32ch 同時読み出しが可能となった。

今後の課題としては、残った「checkFADC」以外のソフトウェアを同じようにコンバージョンしてやる必要がある。ver.1 で使っていたものと同じように機能する新しいプログラムを作成していくことにより、当研究室における RMAP プロトコル環境の完全移行を目指す。また、本研究により開発された新たな環境はまだ構築されたばかりであり、本格的な始動により様々な不具合に見舞われる可能性は十分にある。積極的にこれらの機能を活用して、現れた問題点を随時解決していき、より良い開発環境へ整備していくことが重要である。

謝辞

この卒業論文を執筆するにあたって、多くの方々のお世話となりました。指導教官の高橋先生には、たくさんのアドバイスを頂き、常に進むべき道を照らし出してもらいました。大野先生には、実験室で APMU や様々な機器について教えてもらい、特に SpaceWire RMAP Library ver.2 のインストールの際、深夜でありながらたくさんサポートして頂き大変お世話になりました。大学院生の徳田さんには、SpaceWire ケーブルの作り方を付きっきりで教えてもらい、物を作ることの楽しさとできた時の感動を教えてもらいました。同じく大学院生の後藤さんには、この卒業研究が始まってまだ何もわからない時と、元カノ絡みの件で落ち込んだ際にたくさんのエールを頂きました。そして、高エネルギー宇宙・可視赤外天文学研究グループの教授である深沢先生には、研究に集中できる素晴らしい環境をあたえてもらい、たくさんのことをご教授してもらうとともに、私のたくさんのわがままを聞いてもらい本当にお世話になりました。この他にも、ここでは紹介しきれない多くの人々にお世話していただきました。私はこの研究生活を通じて、論理的思考や生じた問題を解決していくスキル、自分の考えを人に伝える技術など、これから生きていく上で重要な能力が養われたと思います。最後になりましたが、この 1 年間を通じてお世話になった全ての人にこの場を借りてお礼申し上げます。ありがとうございました。

関連図書

[1] JAXA:X 線天文衛星「ASTRO-H」,'http://www.jaxa.jp/projects/sat/astro_h/index_jj.html'

[2] European Cooperation for Space Standardization, Standard ECSS-E-50-12A, “SpaceWire - Links, nodes, routers and networks”January 2003

[3] 湯浅 孝行 「SpaceWire/RMAP Library」, 2008

[4] 松岡 正之 「宇宙軟ガンマ線の気球・衛星観測におけるシンチレータ信号の波形ディジタル処理の実証」, 広島大学大学院, 修士論文, 2010

[5] 田中 琢也 「衛星搭載機器統一通信規格 SpaceWire を用いた宇宙 X 線・ガンマ線観測用データ収集システムの開発」, 広島大学大学院, 修士論文, 2007

[6] シマフジ電機株式会社,SPW Digital I/O ボード（双方向拡張バス、uart 機能付き）取り扱い説明書 ver 0.3 ,2007

[7] T.Yuasa,”SpaceWire-to-GigabitEther User Guide,” September, 2010

[8] T.Yuasa,”SpaceWire RMAP Library v2 User Guide,” January, 2012

[9] European Cooperation for Space Standardization, Standard ECSS-E-50-12A,”Space engineering,” 2010

[10] Shimafuji Electric Inc.,” スタンドアロン版 Space Wire to GigabitEhter ハードウェア仕様書 Rev1.5,”2012