AJAN
拡張コマンド
リファレンス
Ｖｏｌ．２

目　次

第1章 はじめに

本ドキュメントは、AJANの拡張コマンドの説明を記載しています。
拡張コマンド以外のコマンド(標準コマンド、IO制御コマンドなど)は、別マニュアルを用意しています。

本ドキュメントでは、説明で表現している表記として下記のように定義します。

・コマンドの書式の説明において、[ ]内の引数は省略できます。
・文字の大小について
  コマンドは大文字 / 小文字のどちらでも動作します。
  変数名は大文字 / 小文字も同じものとして扱われます。
  ファイルパス / ファイル名は大文字/小文字で区別されます。

本ドキュメント記載の、AJANはIoT用プログラミング言語です。
Interface Linux System上でのみ動作可能です。

第2章 オブジェクト変数

!	オブジェクト変数は、専門的な用途で使用するための特殊な変数です。

2.1 オブジェクト変数の基本

オブジェクト変数は、オブジェクト指向言語のクラスの概念に近い変数で、変数名の最後に「@」を付けることにより区別します。
いくつかの専門的な用途で使用するための特殊な変数です。

2.2 オブジェクト変数の宣言

オブジェクト変数を使用するには、まず宣言が必要となります。

```
書式＞
  OBJECT 〈①変数名〉 AS 〈②オブジェクト型名〉
```

オブジェクト変数には、以下の型が定義されています。

オブジェクト型名	解説	ページ
POINTER	メモリのアドレス値を管理します。	P.122
MEMORY	ヒープメモリを確保し、メモリ操作を行えます。	P.123
JSON	JSON型のオブジェクトを管理します。	P.142

2.3 オブジェクト変数の関数・命令

オブジェクト変数には、それぞれの型に応じた関数・命令が使用できます。
例えば、POINTER 型を例にとると、以下の関数・命令が使用できます。

関数・命令	解説
〈POINTER〉.FROMLNG	POINTER型オブジェクト変数に、アドレス値をセットします。
〈POINTER〉.TOLNG	POINTER型オブジェクト変数から、アドレス値を取得します。

例）POINTER型のオブジェクト変数を宣言して、アドレス値 &H12345 を代入します。

```
OBJECT PTR@ AS POINTER
PTR@.FROMLNG(&H12345)
```

より詳しい説明や、その他の型に関しては、以降の章や「第5章 リファレンス」を参照してください。

第3章 機能説明

3.1 C言語連携の基本

C言語連携機能は、AJANから共有ライブラリの任意の関数を呼び出したり、C言語で書かれたソースコードを埋め込んで、ソースコード内の関数を呼び出す事ができます。

> 共有ライブラリとは
> Linuxの/libや/usr/libのディレクトリに格納されており、実行ファイルを実行する際に動的にロードされるライブラリです。
> 拡張子は.soで、代表的な共有ライブラリとして、C言語の標準関数を提供するlibc.soがあります。
> C言語連携を使用する事で、AJANコマンドで提供されていない関数を利用する事ができますので、プログラムの自由度が大きく向上します。

例えば、C言語標準ライブラリの、文字列の長さを測るstrlen関数を呼び出したい時、以下のように記述できます。

```
' libc.soに含まれるstrlen関数を定義します
CDECLARE "libc.so.6", "int strlen(const char* s);"

' C言語で記述する例「int  ret = strlen("hello AJAN");」
RET = 0
ERRNO = CFUNCALL(RET, "strlen", "hello AJAN")    ' strlen を呼び出します
PRINT "errno="; ERRNO
PRINT "戻り値="; RET
```

CDECLARE命令で、どの共有ライブラリの、どの関数を呼び出すか、関数の仕様はどうか。というのを定義します。ここで共有ライブラリの指定は、使用したい共有ライブラリのファイル名を指定し、関数の指定は、C言語のプロトタイプ宣言風に記述できるようになっています。

定義したら、CFUNCALL関数 および CSUBCALL関数にて、CDECLARE命令 で定義した関数を呼び出します。
CSUBCALL関数 は、戻り値の型定義がvoid型で値を返さない。
CFUNCALL関数 は、戻り値の型定義がvoid型以外の値を返すものに対して呼び出します。
引数は、CDECLARE命令 の定義に従って、数値や文字列を渡します。

!	C言語連携はC++には対応しておりません。

3.2 C言語連携：C言語ソースコードを埋め込む

C言語連携では、C言語で書かれたソースコードを、実行プログラムに埋め込んで利用する事が可能です。

「test.c」というC言語ソースコードファイルを、埋め込んで利用するには、以下のように記述できます。

```
' C言語ソースコードファイルを、自身に埋め込みます
CIMPORT "test.c"

' 上のC言語ソースコードファイルで定義された関数を、呼び出しできるように定義します
CDECLARE "", "int test_add(int a, int b);"

' C言語で記述する例「int ret = test_add(1, 2);」
RET = 0
ERRNO = CFUNCALL(RET, "test_add", 1, 2)    ' test_add を呼び出します
PRINT "errno="; ERRNO
PRINT "戻り値="; RET
```

CIMPORT命令で、「test.c」というC言語ソースコードを、自身に埋め込むように指示します。
「test.c」は、以下のようなコードだとします。

```
#include <stdio.h>

int test_add(int a, int b)
{
    printf("test_add(%d, %d) = %d¥n", a, b, a + b);
    return a + b;
}
```

ここでは、「test_add」という関数が定義されています。
埋め込んだ関数を呼び出すには、CDECLARE命令で呼び出し方法を定義する必要があります。
共有ライブラリのファイル名を指定する所で、空文字列を指定すると、(埋め込んだC言語ソースコードがある)自分自身を指し示します。

埋め込む際に、C言語で記述されたソースコードのファイル名や関数名には、幾つか制限があります。これは、AJANの内部で使用している関数シンボルと衝突しないようにする為の制約です。

C言語で記述されたソースコードは、正しくコンパイルできる状態のものを指定してください。
構文的に不正なものを指定すると、コンパイルする際、エラーが発生します。

3.3 C言語連携：ポインタ、配列、const修飾

C言語連携では、C言語特有の型の概念を理解しつつ、AJANから安全に呼び出せるように、CDECLARE命令で定義する必要があります。

特に注意を要するのは、ポインタと文字列です。
C言語でポインタは、呼び出し元に値を返すか、配列状の値を渡すか返す為に用いられます。

例えば、C言語標準ライブラリで、現在時刻を得るtime関数は、凡そ以下のように定義されています。

```
#include <time.h>

time_t time(time_t* t);
```

上記の定義で、引数「t」は、「time_t」型のポインタで宣言されています。
time関数の説明では、1970年1月1日の0時からの経過秒数を返すとあり、「t」引数が NULL値で無ければ、戻り値と同じ値をセットして返すとあります。

C言語で、time関数の呼び出し方を例示すると、以下のように書けます。

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main()
{
    time_t rt, it;
    rt = time(&it);
    printf("%lld = time(%lld)¥n", rt, it);
    printf("time_tの文字列化(戻り値)=%s", ctime(&rt));
    printf("time_tの文字列化(引数)=%s", ctime(&it));

    return 0;
}
```

参考までに実行すると、以下のような出力が得られます。

```
$ ./a.out
1600231119 = time(1600231119)
time_tの文字列化(戻り値)=Wed Sep 16 13:38:39 2020
time_tの文字列化(引数)=Wed Sep 16 13:38:39 2020
```

次に、AJANで先の呼び出し方を書こうとするには、以下のように書けます。

```
CDECLARE "libc.so.6", "time_t time(time_t* t);"
CDECLARE "libc.so.6", "char* ctime(const time_t* t);"

IT& = 0
RT& = 0
ERRNO = CFUNCALL(RT&, "time", IT&)
? RT&; " = time("; IT&; ")"

S$ = ""
ERRNO = CFUNCALL(S$, "ctime", RT&)
? "time_tの文字列化(戻り値)="; S$;
ERRNO = CFUNCALL(S$, "ctime", IT&)
? "time_tの文字列化(引数)="; S$;
```

C言語の「time_t」型は、OS環境によって 32ビット整数か64ビット整数か異なりますが、AJANが動作する Interface Linux Systemでは、64ビット整数(8バイト長)が採用されます。
この為、AJANで「time_t」型の値を授受するには、同じ64ビット整数(8バイト長)である、倍精度整数型(変数名に「&」を修飾する)で受け取ります。

C言語のプログラム例とAJANのプログラム例を比較して、ポインタ引数で定義すると、呼び出し側に値が渡ってくる事を確認してください。

次に別のポインタを扱った事例を示します。
例えば、以下のようなC言語で書かれたプログラムがあるとします。

```
#include <stdio.h>
#include <string.h>

int test_sum(int* ary, int n)
{
    int total = 0;
    int i;
    for(i = 0; i < n; i++){
        total += ary[i];
    }
    return total;
}

int main()
{
    int ary[] = { 1, 2, 3, 4, 5 };
    int total = test_sum(ary, 5);
    printf("合計=%d¥n", total);

    return 0;
}
```

参考までに実行すると、以下のような出力が得られます。

```
$ ./a.out
合計=15
```

上記のC言語プログラムでは、呼び出し元で、「1,2,3,4,5」と列挙されたint型の配列を、「test_sum」関数に渡しています。
「test_sum」関数では、「int」型のポインタで渡ってきた「ary」変数を先頭に、「n」個分、値の加算を繰り返して、「total」変数にセットし、値を返却しています。

「time」関数の事例では、「time_t」型のポインタは、一つの値を返却していましたが、
この「test_sum」関数の事例では、「int」型のポインタは、複数の値を配列のように参照している事に注意ください。
同じポインタ型でも、一つだけ扱ったり、配列のように扱ったりと、挙動が異なります。

次に、AJANで先の呼び出し方を書こうとするには、以下のように書きます。

```
CIMPORT "test.c"
CDECLARE "", "int test_sum(int* ary, int n);"

LIST ARY%
ARY% = [ 1; 2; 3; 4; 5 ]
RET = 0
ERRNO = CFUNCALL(RET, "test_sum", ARY%, 5)
? "合計="; RET
```

CIMPORT命令で埋め込むC言語プログラムは、以下とします。(ファイル名は「test.c」)
先に示した、C言語プログラムの「test_sum」関数部分のみを抜粋したものです。

```
#include <stdio.h>
#include <string.h>

int test_sum(int* ary, int n)
{
    int total = 0;
    int i;
    for(i = 0; i < n; i++){
        total += ary[i];
    }
    return total;
}
```

C言語プログラムで、「test_sum」関数の「ary」引数は、int型のポインタで配列を渡す必要がありました。
AJANでも、「test_sum」関数を呼び出す際、「ARY%」配列変数にして、これを引数にして渡します。
このように、C言語の関数が要求する仕様に合わせてAJAN側も単一の変数を渡したり、配列を渡したりする必要があります。

ポインタを扱う際、配列のように渡すには、呼び出し元が配列を用意します。
仮に、配列の大きさが明示的であれば、CDECLARE命令 で定義する際に、配列の大きさを指示する事で、明示的に記述可能です。

以下のAJANプログラム例を見てください。（ファイル名は「test_ary.ajn」）

```
CIMPORT "test.c"
CDECLARE "", "void test_ary(int ary[10]);"

LIST ARY%
? "受け取り前="; ARY%
ERRNO = CSUBCALL("test_ary", ARY%)
? "受け取り後="; ARY%
```

CIMPORT命令で埋め込むC言語プログラムは、以下とします。（ファイル名は「test.c」）

```
#include <stdio.h>
#include <string.h>

void test_ary(int ary[10])
{
    int   i;
    for(i = 0; i < 10; i++){
        ary[i] = i+1;
    }
}
```

プログラムを実行すると、以下の結果が得られます。

```
$ ./test_ary
受け取り前=[ 0 ]
受け取り後=[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

前述のプログラム例では、「int」型のポインタを「ary」変数の引数としていました。
ここでは、10の要素数の「int」型配列を明示して定義しています。
こうする事で、C言語連携時に10の要素数の配列変数に拡幅して、値をセットして戻ります。

このように、配列の要素数が固定長ならば、配列として明示的に宣言する事で、呼び出しが簡素になります。
ポインタとして定義していると、要素数が不明ですので、呼び出し側が明示的に配列を用意しなければなりません。

次に、文字列を扱ってみましょう。
C言語で文字列は、「char」型のポインタで渡します。文字列のサイズは、ヌルコード(AJANで言えば、CHRB$(0)に相当)に到達するまでが一般的です。

AJANの文字列は、C言語の文字列と異なり、ヌルコード(CHRB$(0))を内包できます。

例えば、C言語標準ライブラリで、文字列の長さを測る strlen関数は、凡そ以下のように定義されています。

```
#include <string.h>

size_t strlen(const char* s);
```

C言語で、strlen関数の呼び出し方を例示すると、以下のように書けます。

```
#include <stdio.h>
#include <string.h>

int main()
{
    printf("長さ=%d", strlen("Hello AJAN"));
    return 0;
}
```

これをAJANでプログラムすると、以下のように書きます。（ファイル名は「test.ajn」）

```
CDECLARE "libc.so.6", "size_t strlen(const char* s);"

RET = 0
S$ = "Hello AJAN"
ERRNO = CFUNCALL(RET, "strlen", S$)
? "長さ="; RET
```

プログラムを実行すると、以下の結果が得られます。

```
$ ./test
長さ=10
```

「char」型のポインタを定義して、AJANで文字列変数を指定すると、このように文字列の内容を、C言語の関数に渡して処理します。

尚、「strlen」関数の定義では、「const char* s」の様に「const」が修飾されています。
「const」は、C言語で定数性、すなわち値を変更しない。という意味を表します。
AJANは、「const」が付加されていると「値を渡す専用」だと認識します。すなわち、呼び出した後、変数の内容が書き換えられないことを明示します。
こうする事で、処理の高速化が図れます。

「const」修飾を付けないと、変数の値を書き換えると判断します。
例えば、C言語標準ライブラリの文字列をコピーするstrcpy関数を使ってみましょう。

C言語で、strcpy関数の呼び出し方を例示すると、以下のように書けます。

```
#include <stdio.h>
#include <string.h>

int main()
{
    const char* src = "Hello AJAN";
    char dest[100];
    char* r = "";
    printf("呼び出し前: %s %s %s¥n", r, dest, src);
    r = strcpy(dest, src);
    printf("呼び出し後: %s %s %s¥n", r, dest, src);

    return 0;
}
```

これをAJANでプログラムすると、以下のように書きます。（ファイル名は「test.ajn」）

```
CDECLARE "libc.so.6", "char* strcpy(char* dest, const char* src);"

RET = 0
S$ = "Hello AJAN"
RS$ = ""
IS$ = ""
? "呼び出し前:", RS$, IS$, S$
ERRNO = CFUNCALL(RS$, "strcpy", IS$, S$)
? "呼び出し後:", RS$, IS$, S$
```

プログラムを実行すると、以下の結果が得られます。

```
$ ./test
呼び出し前:           Hello AJAN
呼び出し後:   Hello AJAN     Hello AJAN     Hello AJAN
```

参考までに、CDECLARE命令で、以下のように定義を変更すると、「IS$」変数には、値が戻ってきません。

```
CDECLARE "libc.so.6", "char* strcpy(char* dest, const char* src);"
  ↓
CDECLARE "libc.so.6", "char* strcpy(const char* dest, const char* src);"
```

3.4 C言語連携：コールバック

C言語では、関数のポインタを渡して、中からコールバックする事が可能です。
以下に、C言語での記述例を示します。

```
#include <stdio.h>
#include <string.h>

void test_cb(int user)
{
    printf("test_cbが呼ばれた:%d¥n", user);
}

void test(void (*cb_func)(int user), int user)
{
    cb_func(user);
}

int main()
{
    test(test_cb, 123);
    test(test_cb, 456);
    return 0;
}
```

プログラムを実行すると、以下の結果が得られます。

```
$ ./a.out
test_cbが呼ばれた:123
test_cbが呼ばれた:456
```

先のプログラムでは、「test」関数に、「test_cb」関数の関数ポインタを渡します。
「test」関数は、関数ポインタを使ってコールバックし、「test_cb」関数を呼び出しています。

AJANでは、関数ポインタをPOINTER型オブジェクト変数に格納する事で、これを実現する事ができます。

```
CIMPORT "test.c"
CDECLARE "", "void test(void (*cb_test)(int user), int user);"
CDECLARE "", "void test_cb(int user);"

OBJECT FNC@ AS POINTER
FNC@ = CGETADRS@("test_cb")

ERRNO = CSUBCALL("test", FNC@, 123)
ERRNO = CSUBCALL("test", FNC@, 456)
```

CIMPORT命令で埋め込むC言語プログラムは、以下とします。(ファイル名は「test.c」)
先に示した、C言語プログラムの「tset_cb」と「test」関数部分のみを抜粋したものです。

```
#include <stdio.h>
#include <string.h>

void test_cb(int user)
{
    printf("test_cbが呼ばれた:%d¥n", user);
}

void test(void (*cb_func)(int user), int user)
{
    cb_func(user);
}
```

AJANで「test_cb」関数の関数ポインタを得るには、CGETADRS@関数を用います。
CGETADRS@関数は、POINTER型オブジェクトを返します。この値を受け取るために、OBJECT命令で、オブジェクト変数(ここでは「FNC@」)を宣言し、代入で受け取ります。

「test」関数で、関数ポインタを渡す所で、POINTER型オブジェクト変数を渡す事で、関数コールバック呼び出しが実現できます。

!	AJANの関数の関数ポインタを取り出して扱う事はできません。 C言語関数のみ取り出して扱えます。

3.5 C言語連携：メモリを扱う

C言語ではポインタを多用しますが、その中でも、任意の大きさのメモリを確保し、これをバッファ的に扱うケースがあります。

例えば、C言語標準ライブラリの、メモリ領域をコピーする memcpy関数は、凡そ以下のように定義されています。

```
#include <string.h>

void* memcpy(void* dest, const void* src, size_t n);
```

C言語で、memcpy関数の呼び出し方を例示すると、以下のように書けます。

```
#include <stdio.h>
#include <string.h>

int main()
{
    const char*     src = "Hello AJAN";
    char  dest[100];
    char* r = NULL;
    r = memcpy(dest, src, strlen(src)+1);
    printf("呼び出し後: %s %s %s¥n", r, dest, src);

    return 0;
}
```

これをAJANでプログラムすると、以下のように書きます。（ファイル名は「test.ajn」）

```
' 呼び出しを楽に記述する為に、戻り値とsrcを、void* -> char* に定義している
CDECLARE "libc.so.6", "char* memcpy(void* dest, const char* src, size_t n);"

S$ = "Hello AJAN"
OBJECT DEST@ AS MEMORY
DEST@ = CALLOC@(100)

ERRNO = CFUNCALL(R$, "memcpy", DEST@, S$, 11)
? "呼び出し後:", R$, DEST@.PEEKSTR$(0, 10), S$
```

プログラムを実行すると、以下の結果が得られます。

```
$ ./test
呼び出し後:   Hello AJAN     Hello AJAN     Hello AJAN
```

「memcpy」関数の「dest」変数で受け取る所は、「void」型のポインタです。
ここで、任意のメモリバッファを指定できるようにする為に、OBJECT命令で、MEMORY型オブジェクト変数(ここでは「DEST@」)を宣言し、代入で受け取ります。

MEMORY型オブジェクト変数は、C言語の「malloc」関数のように、任意のメモリバッファを確保する事ができます。それが、CALLOC@関数です。
CALLOC@関数で、指定したバイト長のメモリバッファを格納し、「DEST@」オブジェクト変数に格納し、「memcpy」関数に渡して、値を受け取る事ができます。

3.6 JSON型の基本

AJANのJSON型は、JSON(JavaScript Object Notation)というデータ交換用の書式で書かれた文字列を任意に扱うためのオブジェクト変数です。

JSONは、Webブラウザ上で動作する JavaScript と サーバとの間でやり取りする、データ(情報)交換の記述書式に JSON 形式が多く使われており、最近では他のプログラミング言語やデータベースの格納書式にも採用が拡がっています。
AJANのJSON型は、このJSON形式のデータを読み取ったり、JSON形式の文字列を作るために用います。

JSONの詳しい書式・仕様は、書籍やインターネットなどの資料を参考にしてください。

JSONは、構造化されたテキストで表現された文字列であり、例えば以下のような記述で表現されます。

```
{
  "name" : "AJAN",
  "year": 2021
}
```

これは、「name」というキーに「AJAN」という文字列が、「year」というキーに「2021」という数値が内包された、JSON形式の文字列です。

この文字列を、AJANのJSON型に取り込むには、JSON_PARSE@ 関数を使用します。

```
S$ = '''{
  ""name"" : ""AJAN"",
  ""year"": 2021
}'''

OBJECT  PT@ AS JSON
PT@ = JSON_PARSE@(S$)     ' 文字列をJSON型に変換し、PT@ 変数に格納します
```

逆に、AJANのJSON型を、元のJSON形式の文字列に変換するには、JSON_DUMP$ 関数を使用します。

```
S$ = '''{
  ""name"" : ""AJAN"",
  ""year"": 2021
}'''

OBJECT  PT@ AS JSON
PT@ = JSON_PARSE@(S$)     ' 文字列をJSON型に変換し、PT@ 変数に格納します

PRINT JSON_DUMP$(PT@)  ' JSON型のPT@ 変数を文字列に変換して出力します
```

次に、JSONのデータ構造の中身の扱いですが、JSON自体は数値や文字列など幾つかの型を、内部構造として表現する決まりとなっています。
JSONでは、以下に挙げる型を持ちます。

JSONの型	説明
オブジェクト	AJANで言う、連想配列の事です。 「{ "name":"AJAN", "year": 2021 }」のように表現します。
配列	AJANで言う、配列の事です。 「[12, 34, 56]」のように表現します。
文字列	「"(ダブルクォーテーション)」で囲んだ文字列です。
数値	「123」や「12.34」のような、整数・浮動小数点数です。
真偽値	「true」や「false」のような、真偽値です。
null値	「null」と表現します。

JSON型オブジェクトの中を、探索するには、まずJSON型の内部の型が、どうなっているか調べる必要があります。
内部の型が何か調べるには、<JSON>.TYPEOF メソッドを使用します。

```
S$ = '''{
  ""name"" : ""AJAN"",
  ""year"": 2021
}'''

OBJECT  PT@ AS JSON
PT@ = JSON_PARSE@(S$)     ' 文字列をJSON型に変換し、PT@ 変数に格納します
PRINT PT@.TYPEOF()        ' PT@変数の内部の型を調べます。オブジェクトなので 0が表示されます
```

内部の型が判ったら、更に中身の値を取り出す事ができます。
JSONの内部の型に応じて、以下に挙げるメソッドを使って値を取り出せます。

JSONの内部の型	対応メソッド
オブジェクト	<JSON>.OBJECT_GET@ で、キーを指定して対応する値を取り出せます。
配列	<JSON>.ARRAY_GET@ で、添字を指定して対応する値を取り出せます。
文字列	<JSON>.GET_CSTR$ で、文字列を取り出せます。
数値	整数の場合、<JSON>.GET_CLNG で、整数値を取り出せます。 実数の場合、<JSON>.GET_CDBL で、実数値を取り出せます。
真偽値	<JSON>.GET_CBOOL で、真偽値を取り出せます。
null値	null値が、明示的なので、ありません。

JSON型オブジェクトに対して、値を置き換えたり、追加・挿入・削除するには、JSON_NEW@ 関数や JSON_PARSE@ 関数で、単一の値を作って、以下に挙げるメソッドを使って、操作を行います。

値を作る場合：

JSONの型	値の作り方
オブジェクト	JSON_PARSE@("{}") のように書きます。
配列	JSON_PARSE@("[]") のように書きます。
文字列	JSON_NEW@("Hello AJAN") のように書きます。
数値	整数の場合、JSON_NEW@(123%) のように書きます。 実数の場合、JSON_NEW@(12.34#) のように書きます。
真偽値	JSON_NEW@(TRUE) のように書きます。
null値	JSON_PARSE@("null") のように書きます。

値を設定・追加・挿入・削除する場合：

JSONの型	値の操作方法
オブジェクト	設定：<JSON>.OBJECT_SET 追加：<JSON>.OBJECT_SET 削除：<JSON>.OBJECT_DEL_KEY 全て削除：<JSON>.OBJECT_CLEAR
配列	設定：<JSON>.ARRAY_SET 追加・挿入：<JSON>.ARRAY_INSERT 削除：<JSON>.ARRAY_REMOVE 全て削除：<JSON>.ARRAY_CLEAR

参考として、オブジェクト(連想配列)に新たな値を設定する方法と、配列に新たな値を追加する方法を例示します。

```
OBJECT  PT@ AS JSON, TT@ AS JSON
PT@ = JSON_PARSE@("{}")             ' 空のオブジェクト(連想配列)を作ります
TT@ = JSON_NEW@("Hello AJAN")       ' 文字列を作ります
PT@.OBJECT_SET("test", TT@)         ' 「test」キーに、文字列を設定します
TT@ = JSON_NEW@(123)                ' 数値を作ります
PT@.OBJECT_SET("hoge", TT@)         ' 「hoge」キーに、数値を設定します
PT@.OBJECT_SET("hoge", TRUE)        ' 「hoge」キーの値を置き換えます

PT@ = JSON_PARSE@(" [] ")           ' 空の配列を作ります
TT@ = JSON_NEW@("Interface")        ' 文字列を作ります
PT@.ARRAY_INSERT(-1, TT@)           ' 配列の末尾に、文字列を追加します
TT@ = JSON_NEW@(123.45)             ' 数値を作ります
PT@.ARRAY_INSERT(0, TT@)            ' 配列の先頭に、数値を挿入します
PT@.ARRAY_SET(0, 54321)             ' 配列の先頭の値を置き換えます

PRINT JSON_DUMP$(PT@)               ' 構築したJSON型を文字列化します
```

このように、JSON型を扱う関数とメソッドを使って、自由にJSON形式のデータを扱う事が可能です。

3.7 MEMORY型やJSON型の変数をサブルーチンに渡す際の注意

MEMORY型やJSON型の変数を、別の変数に代入するとき、基本的に値は複製されます。
これは、SUB ～ END SUB で定義するサブルーチンや、FUNCTION ～ END FUNCTION で定義するユーザー定義関数の引数に渡す際も同じです。

CMEMMAP@ で得られた MEMORY型変数の値のみ、値は複製されず、同じメモリアドレスを参照します。

```
OBJECT  PT@ AS MEMORY
PT@ = CALLOC@(100)

CALL TEST(PT@)

SUB TEST(V@ AS MEMORY)
    PRINT "PT@ のサイズ="; PT@.SIZEOF(); " アドレス="; PT@.TOLNG()
    PRINT "V@ のサイズ="; V@.SIZEOF(); " アドレス="; V@.TOLNG()
END SUB
```

以下に実行例を示します。

```
PT@ のサイズ=100 アドレス=94105733185744
V@ のサイズ=100 アドレス=94105733186112
```

コード例では PT@ 変数が、TEST サブルーチンの V@ 変数に積み込まれて呼び出されています。
値が複製されているため、メモリのサイズは同じですが、メモリアドレスは別となります。

第4章 事例説明

本章では、弊社インタフェースモジュールまたは、Classembly Devicesに搭載のI/O部を制御する、各カテゴリのドライバ(GPG-xxxx)を、C言語連携機能を使って制御する事例を、幾つか紹介いたします。

!	本章では、CoolIOsシリーズのI/O制御および、CAN通信およびUART通信、システム監視のI/O制御方法は 扱いません。 それぞれ、専用のAJANコマンドおよびヘルプがあるので、そちらを参照ください。

4.1 GPG製品のI/O制御の共通事項

弊社製品のI/O機能を制御するドライバおよびライブラリは、製品ごとに幾つか用意されています。ここでは、GPG-xxxx で表現される型式の製品で、C言語連携機能を用いて制御を行う際の幾つか勘所を説明します。

GPG-xxxx の各型式のreadme や ヘルプ、サンプルプログラムなどは、「/usr/src/interface」フォルダ下に型式をフォルダ名にしたものが出来て、そこに格納されています。

```
$ ls -l /usr/src/interface/
合計 116
drwxr-xr-x 4 root root 4096  1月 18 13:16 common
drwxr-xr-x 3 root root 4096  1月 14 18:20 gpg2000
drwxr-xr-x 3 root root 4096  1月  7 14:27 gpg3100
drwxr-xr-x 3 root root 4096  1月 13 19:50 gpg3300
drwxr-xr-x 3 root root 4096  1月 15 13:40 gpg4116
drwxr-xr-x 3 root root 4096  1月 18 13:16 gpg4301
drwxr-xr-x 3 root root 4096  1月 15 15:42 gpg6204
drwxr-xr-x 3 root root 4096  1月 15 17:25 gpg6320
drwxr-xr-x 3 root root 4096 11月 18 00:56 interfacelinuxsystem
<略>
```

まずは、各型式の readme.txt と ヘルプファイルを参照し、どのような機能やAPIが提供されているのか、または製品ごとの制限事項などを把握してください。

C言語連携機能を用いる際に必ず把握が必要なのは、以下の2つです。この2つの情報を、CDECLARE 命令で指定する事で、AJANはI/O制御用のライブラリを呼び出す準備を整えます。

No	項目	解説
1	共有ライブラリ	C言語でアプリケーションを作る際に必須となる、共有ライブラリです。 ファイルの拡張子は、「.so」です。 アプリケーションは、このライブラリに実装されている API(関数)を呼び出す事で、I/Oを制御できます。 C言語連携時、AJANは、共有ライブラリを動的に読み込んで、使用できるようにします。
2	ヘッダファイル	アプリケーションが、上記ライブラリを呼び出す為に必要な APIの宣言および定義をまとめたファイルです。ファイルの拡張子は「.h」です。 C言語連携時、AJANは、このヘッダファイルで定義されたAPIのプロトタイプ宣言を読み込んで、どのように呼び出せば良いか解釈します。

上記ファイルは、readme.txt か ヘルプファイルに記載されています。
例えばGPG-3100の場合、readme.txt の「■ファイル一覧」にファイル名が記載されています。
(以下、一部抜粋しています)

```
<略>
 ●ファイル一覧

   インストール後のディレクトリおよびファイル構成は次の通りです。

   /usr/
      bin/
         dpg0101        デバイス番号設定ユーティリティ
         caddiag        自己診断プログラム
         cdouble        倍速モードAD調整プログラム
         cadadjust      AD調整プログラム
         xadwave        AD波形入力ユーティリティ

      include/
         fbiad.h        GPG-3100定義ファイル
         dpg0100.h      共通モジュール定義ファイル

      lib/ ※ 64bit版には含まれません。
         libgpg3100.so.x.xx.xx       ライブラリ
         libgpg3100t.so.x.xx.xx      テストドライバ用ライブラリ
         libgpgconf.so.x.xx.xx       ライブラリ
         ※ x.xx.xxは、バージョンが入ります。

      lib64/ ※ 32bit版には含まれません。
         libgpg3100.so.x.xx.xx       ライブラリ
         libgpg3100t.so.x.xx.xx      テストドライバ用ライブラリ
         libgpgconf.so.x.xx.xx       ライブラリ
         ※ x.xx.xxは、バージョンが入ります。
<略>
```

GPG-3100の場合、CDECLARE 命令 は、以下のように呼び出す事で、GPG-3100を呼び出す基本準備が整います。

```
' GPG-3100のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbiad.h") ' GPG-3100要インストール
' C言語連携を行う為に、共有ライブラリの指定と関数プロトタイプ宣言、構造体を解釈します
CDECLARE "libgpg3100.so", s$
```

以降では、その他の勘所について説明します。

4.1.1 ドライバのサンプルを動かす所から始める。

いきなりC言語連携のプログラムを作る事から始めず、ドライバ(GPG-xxxx)に添付されているサンプルを動かしてみる事から始めましょう。

これには、ドライバのセットアップを完了せずに、いきなりI/O制御を動かそうとして、問題につまずくケースを防ぐ効果があります。

4.1.2 C言語とAJANの型の大きさ違いに注意する

C言語で扱える型とAJANで扱える型は、同じ数値型でも違います。
また、各型は値を表現する為に必要なサイズが異なります。特にC言語の場合、この各型が必要とするサイズ長を常に意識する必要があります。

C言語の型		対応するAJANの型	
型名	サイズ(byte単位)	型名	サイズ(byte単位)
char, int8_t	1	なし	ー
short, int16_t	2	なし	ー
int, int32_t	4	単精度整数(%)	4
long, int64_t	8	倍精度整数(&)	8
float	4	単精度実数(!)	4
double	8	倍精度実数(#)	8
void*, char* など(ポインタ型)	8	なし	ー
char* など(文字列)	文字数による	文字列型($)	可変長

※ long は、Linuxの64bit環境で、8バイトサイズですが、他の環境では異なる事があります。確実に 8バイトサイズを保証したい場合、int64_t を使用します。

※ void* などのポインタは、Linuxの64bit環境で8バイトサイズですが、他の環境では異なる事があります。sizeof(void*)でサイズが確認できます。

上の表で、int8_t や int16_t は、対応するAJANの型がありません。単に数値を受け渡したい場合は、よりサイズの大きい単精度整数を使用してください。
また、void* や char* などのポインタ型も、対応するAJANの型がありません。このポインタが文字列を意図したものであれば、文字列型を渡し、それ以外の用途を意図したものであれば、『4.1.5 C言語のポインタは取り扱い要注意』を参照ください。

C言語連携を使用すると、C言語の型と異なるAJANの型の変数や数値を指定した時、AJANは内部でC言語の型に変換して対応しようとします。

例えば、int8_t を受け取るC言語の関数があり、それに123456の値をAJANから与えようとしたとします。

```
' sub.c の中身は、↓以下と仮定します。
'  #include <stdio.h>
'  #include <stdint.h>
'
'  int c_hoge(int8_t v)
'  {
'     return printf("c_hoge %d %#x¥n", v, v);
'  }
CIMPORT "sub.c"
CDECLARE "", "int c_hoge(int8_t v);"

? "AJAN=", 123456, HEX$(123456)
errno = CFUNCALL(ret, "c_hoge", 123456)
? errno, ret
```

AJANは、C言語の関数を呼び出す手前で、123456の数値を、int8_t 型に一旦キャストしてから呼び出します。
その結果、上のコードを実行すると、以下のような結果が得られます。

```
AJAN    123456  1E240
c_hoge  64       0x40
0       15
```

C言語の関数に渡った数値が、元のAJANの数値と異なる点に注意してください。
(下位8ビット分の値のみ、C言語側に渡っています)
このように、C言語では、型の大きさに注意する必要がある為、呼び出すAJAN側も、C言語では何の型を使用しているかを、常に認識する必要があります。

4.1.3 C言語に文字列型は存在しない

C言語で「文字列」は扱われますが「文字列型」という専用の型はありません。
一般には、char型のポインタやchar型の配列に、文字列のポインタを渡したり格納する事で、文字列を操作します。

例えば、C言語で文字列の長さを測る「strlen」関数は、引数に char型のポインタを指定します。

```
CDECLARE "libc.so.6", "size_t strlen(const char* s);"
S$ = "Hello AJAN World"
errno = CFUNCALL(ret, "strlen", S$)
? errno, ret
```

AJANは、文字列型変数の S$ を指定した時、文字列型変数の内部バッファの先頭を、C言語の関数のchar型のポインタに渡してから呼び出します。
このポインタの値が保障されるのは、C言語側から処理が戻るまでです。
文字列の実体はAJAN側にありますが、ガベージコレクション等で文字列の格納場所は常に変化する可能性があります。C言語側で前回のポインタを保持し続けることが無いように注意して下さい。

また、C言語で文字列は、"¥0"(AJANでは、CHRB$(0)に相当) で終端します。
AJANは文字列型という専用の型を持ち、文字列のサイズを保持している為、中途に CHRB$(0) を入れても、LENB関数で 文字列型のバイトサイズを認識できます。

例えば、AJAN側から文字列の途中に CHRB$(0) を挿入した文字列型変数を、C言語の「strlen」関数に渡すと、CHRB$(0) で終端した所までが文字数として返りますが、
AJANは、CHRB$(0) を内包できるので、バッファサイズを正しく返します。

```
CDECLARE "libc.so.6", "size_t strlen(const char* s);"
S$ = "Hello"+CHRB$(0)+"AJAN World"
errno = CFUNCALL(ret, "strlen", S$)
? "C言語が認識する文字列サイズ="; ret       ' 5 が出力される
? "AJANが認識する文字列サイズ="; LENB(S$)   ' 16 が出力される
```

C言語側で文字列を処理する場合は、AJAN側で中途にCHRB$(0)が含まれないようにする等の配慮が必要になります。

4.1.4 C言語は大文字小文字を区別する

AJANで変数名やコマンド名は、大文字で書いても小文字で書いても、同じ名前として認識します。
しかし、C言語では、1文字異なると別の名前として認識します。

例えば、AJANで以下の呼び出しは、同じ PRINT文として認識します。

```
PRINT "大文字で呼び出し"
print "小文字で呼び出し"          ' 小文字でも同じPRINT文
```

しかし、C言語では、2行目の Printf 文は、何か別の関数名として認識します。

```
printf("大文字で呼び出し");
Printf("小文字で呼び出し");       ' 先頭1文字が違うだけで、全然別の関数名
```

この事は、CFUNCALLや CSUBCALLによる、C言語のAPI呼び出しや、<MEMORY>.GETMEMVALや<MEMORY>.SETMEMVALなどの、C言語の構造体のメンバ名アクセスに影響します。
必ず、大文字・小文字の区別に注意を払って呼び出す必要があります。

4.1.5 C言語のポインタは取り扱い要注意

C言語のポインタは、メモリや変数の参照先、あるいは関数のアドレスを指したりと、非常に使い勝手が良い機能ですが、使い方によっては危険な機能です。

ポインタの難しい点は、構文だけでは それが読み出し専用なのか、書き込みを付随するのか、配列なのか、どの程度の大きさが必要なのか、コールバック関数に使いたいのか、はっきりしない点にあります。

例えば、弊社 GPG-4301(GPIBライブラリ)の、データ受信用の関数 PciGpibExRecvData の書式を見てみましょう。

```
int PciGpibExRecvData(
  int               nBoardNo,       /* デバイス番号 */
  int*              npAdrsTbl,      /* 機器のアドレステーブルのポインタ */
  unsigned long*    ulpLength,      /* 受信データ長を格納する変数へのポインタ */
  void*             pvBuffer,       /* 受信データ領域へのポインタ */
  void*             pvFunc          /* コールバック関数へのポインタ */
);
```

この関数は、4つのポインタを引数に持ちますが、それぞれ単一変数の戻り値に使ったり、配列を渡したり、受信バッファとして渡したり、コールバック用に渡したりと構文からでは判り難いです。

以下に、PciGpibExRecvData の(C言語の)呼び出し事例を示します。

```
// 記述例
int ret;
int npAdrsTbl[2] = { 2, -1 };       // 第2引数に渡す、機器のアドレステーブル
unsigned long len;                  // 第3引数に渡す、受信データ長の変数
char RecvBuf[100];                  // 第4引数に渡す、受信バッファの配列

void cb_recv_cmpl(void)             // 第5引数に渡す、コールバック関数の飛び先
{
  puts("受信完了!!");
}

len = sizeof(RecvBuf);
ret = PciGpibExRecvData(0, npAdrsTbl, &len, RecvBuf, cb_recv_cmpl);
```

各引数は、以下のように役割が異なります。

- 第2引数は、配列のポインタを渡しています。これは読み出し専用です。
- 第3引数は、単一の変数のポインタを渡しています。これは書き込みを期待します。
- 第4引数は、配列のポインタを渡しています。これは書き込みを期待しています。
- 第5引数は、関数のポインタを渡しています。これはコールバックを期待しています。

int* や void* など、型は違っていても同じポインタです。しかし、配列を期待したり、書き込

まれるのを期待したり、コールバック関数として呼ばれるのを期待しています。

AJANのC言語連携では、変数を渡すと内部バッファをポインタとして渡します。
この時、十分なバッファないしはメモリを確保した状態で渡さないと、呼び出したC言語の関数が、AJAN側の内部メモリを破壊する事は十分考えられます。
結果、アプリケーションの動作異常や、Segmentation fault など、コンピュータが致命的異常を検知して動作を停止する事も十分ありえます。

例えば、配列を期待している引数に対しては、AJANの配列か CALLOC@ 関数か、CSTRUCT@ 関数で確保した、MEMORY型のオブジェクトを渡します。
あるいは、コールバック関数のアドレスを期待している引数に対しては、「4.1.7 C言語のコールバックを扱うには CIMPORT を利用する」で示すようなやり方で、POINTER型のオブジェクトを渡します。

引数によっては、C言語で NULL 値を渡せる場合があります。AJANでは、0 を指定すると、NULL値の代わりとなりますので、状況に応じて、GPG-xxxxのオンラインヘルプの説明を読んで指定してください。

4.1.6 C言語の構造体を扱うには CSTRUCT@ を利用する

AJANでC言語の構造体を扱うには、CSTRUCT@関数で、目的とする構造体名を指定して、MEMORY型オブジェクトの作成を行い、<MEMORY>.GETMEMVALメソッドや<MEMORY>.GETMEMSTR$メソッド、<MEMORY>.SETMEMVALメソッドを使って、C言語の構造体の各メンバーに対して読み書きを行うように記述できます。

!	AJANの構造体は、C言語の構造体のメモリ管理方法と異なるため、 C言語の関数に対して、AJANの構造体を直接渡す事はできません。

C言語の構造体は、各メンバの型が要求する型サイズの順番に、直列にメモリにマッピングされるのでなく、各メンバの前後の型によって、CPUがアクセスしやすいように メモリの配置位置が調整されます。(技術的には、データ構造アライメントとも呼ばれます)

AJANは、CSTRUCT@関数 で C言語の構造体定義名を指定した時、データ構造アライメントの方式に従ってアクセスできるように、調整を行います。
これにより、以下のメソッドが使用できるようになります。

メソッド名	機能
<MEMORY>.GETMEMOFF	C言語の構造体に対して、指定したメンバ名に相当するオフセット位置を得ます。
<MEMORY>.GETMEMVAL	C言語の構造体に対して、指定したメンバ名に相当するメンバ位置の、メモリの値を取得します。
<MEMORY>.GETMEMSTR$	C言語の構造体に対して、指定したメンバ名に相当するメンバ位置の、メモリの文字列を取得します。
<MEMORY>.SETMEMVAL	C言語の構造体に対して、指定したメンバ名に相当するメンバ位置の、メモリに対して値を書き込みます。

尚、CALLOC@関数でC言語の構造体定義を行わずにメモリを確保した場合、上記メソッドは使えません。自身でC言語の構造体のメンバーにアクセスするオフセット位置を計算して、<MEMORY>.PEEKINTメソッドや <MEMORY>.POKEINTメソッドなどの、メモリ読み書きメソッドを使って操作します。

又、CSTRUCT@関数で確保したMEMORY型オブジェクトは、<MEMORY>.TOJSON$メソッドを使って、C言語の構造体のメンバ名と値が、どのような状態になっているかを、JSON文字列形式で得る事が可能です。これにより、構造体を使ったデバッグがやりやすくなります。
以下にコードと事例を示します。

コード例：

```
s$ = '''
struct INFO {
  int     id;
  char    msg[20];
};
'''
CDECLARE "", s$
OBJECT   mem@ AS MEMORY
mem@ = CSTRUCT@("INFO")

mem@.SETMEMVAL("id", 123)
mem@.SETMEMVAL("msg", "Hello AJAN")

PRINT mem@.TOJSON$()
```

実行例：

```
{
    "id": 123,
    "msg": "Hello AJAN¥u0000¥u0000¥u0000¥u0000¥u0000¥u0000¥u0000¥u0000¥u0000¥u0000"
}
```

4.1.7 C言語のコールバックを扱うには CIMPORT を利用する

AJANでC言語のコールバック処理を扱うには、C言語で書かれたサブルーチンをCIMPORT命令で組み込んで、CGETADRS@関数で、サブルーチンの関数アドレスをPOINTER型オブジェクトで取得して、これを利用してください。

以下では、C言語で書かれたサブルーチンを、AJANから呼び出せるように、アドレス取得して呼び出す事例です。

AJANコード例：

```
CIMPORT "test_sub.c"
CDECLARE "", '''
int c_add(int a, int b);
int c_sub(int a, int b);
int c_func(int (*func)(int a, int b), int a, int b);
'''

OBJECT fnc_add@ AS POINTER
OBJECT fnc_sub@ AS POINTER
fnc_add@ = CGETADRS@("c_add")
fnc_sub@ = CGETADRS@("c_sub")

errno = CFUNCALL(ret, "c_func", fnc_add@, 2, 4)
PRINT "use c_add=", ret
errno = CFUNCALL(ret, "c_func", fnc_sub@, 2, 4)
PRINT "use c_sub=", ret
```

C言語コード例（test_sub.c）：

```
#include <stdio.h>

int c_add(int a, int b)
{
  printf("c_add(%d, %d) がコールされた¥n", a, b);
  return a + b;
}

int c_sub(int a, int b)
{
  printf("c_sub(%d, %d) がコールされた¥n", a, b);
  return a - b;
}

int c_func(int (*func)(int a, int b), int a, int b)
{
  return func(a, b);
}
```

上記をコンパイルして実行すると、以下のような結果が得られます。

```
c_add(2, 4) がコールされた
use c_add=6
c_sub(2, 4) がコールされた
use c_sub=-2
```

C言語で書かれたコードが、AJANアプリケーションに組み込まれ、その関数のアドレスを CGETADRS@関数 で取得して利用できるようにしています。
上の例では、C言語で書かれた「c_add」関数、「c_sub」関数のアドレスを取得し、「c_func」関数のコールバック関数に引数として渡し、コールバック処理を行っています。

	AJANのサブルーチンは、C言語のサブルーチンと構造が異なるため、 C言語のコールバック関数用の引数に、AJANのサブルーチンを直接渡す事はできません。
	CIMPORT命令で組み込むC言語のルーチンを記述するとき、AJANからアクセスできる関数は、「c_」や「test」などのプレフィクスを付けた関数名にする必要があります。

4.2 GPG-2000 デジタル入出力ドライバの制御事例

4.2.1 基本手順

GPG-2000は、弊社デジタル入出力機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg2000.so
ヘッダファイル	fbidio.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-2000のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbidio.h")        ' GPG-2000 要インストール
CDECLARE "libgpg2000.so", s$
```

GPG-2000のヘルプファイルでは、『制御手順』の章で、各機能と使用する関数について、幾つか列挙されています。
この中で、よく使われると思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	DioOpen	デバイスのオープン
出力	DioOutputByte DioOutputWord DioOutputDword DioOutputPoint	接点 8 点分(8 ビット)の出力 接点 16 点分(16 ビット)の出力 接点 32 点分(32 ビット)の出力 任意の点数の出力
入力	DioInputByte DioInputWord DioInputDword DioInputPoint	接点 8 点分(8 ビット)の入力 接点 16 点分(16 ビット)の入力 接点 32 点分(32 ビット)の入力 任意の点数の入力
終了処理	DioClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンのDioOpen と、デバイスクローズのDioCloseです。
関数の引数と戻り値は、int 型もしくは、unsigned long型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// デバイスオープン
nRet = DioOpen(nDevice, 0);
	nDevice = 1
	' デバイスオープン
	errno = CFUNCALL(nRet%, "DioOpen", nDevice, 0)

int nRet;
int nDevice;
// デバイスクローズ
nRet = DioClose(nDevice);
	' デバイスクローズ
	errno = CFUNCALL(nRet%, "DioClose", nDevice)

次に、出力のDioOutputByteと入力のDioInputByteです。

C言語の例	AJANの例

int nRet;
unsigned char bValue = 0x12;
int nDevice;
// 接点８点の出力
nRet = DioOutputByte(nDevice, FBIDIO_OUT1_8, bValue);
	bValue% = &H12
	’ 接点８点の出力
	errno = CFUNCALL(nRet%, ″DioOutputByte″, nDevice, &H100/* FBIDIO_OUT1_8 */, bValue%)

int nRet;
unsigned char bValue;
int nDevice;
// 接点８点の入力
nRet = DioInputByte(nDevice, FBIDIO_IN1_8, &bValue);
	bValue% = 0
	’ 接点８点の入力
	errno = CFUNCALL(nRet%, ″DioInputByte″, nDevice, &H1/* FBIDIO_IN1_8 */, bValue%)

C言語の例で、定数名「FBIDIO_OUT1_8」または「FBIDIO_IN1_8」があります。
これは、ヘッダファイル(fbidio.h)にて、以下のように定義されていますので、即値を指定します。
(0x00000100 は、C言語で16進数の定数値なので、AJANでは &H00000100 というように記述します)

```
... <略>
#define FBIDIO_IN1_8        0x00000001
#define FBIDIO_IN9_16       0x00000002
... <略>

#define FBIDIO_OUT1_8       0x00000100
#define FBIDIO_OUT9_16      0x00000200
... <略>
```

また、C言語の例で変数名「bValue」を、AJANでは「bValue%」と記述しています。
「bValue%」は単精度整数変数を明示しています。実際には、単に「bValue」としても動作しますが、「bValue」は暗黙的に倍精度実数変数として扱われる為、内部で整数←→実数の変換作業が発生します。
無駄な変換作業が発生しないように「bValue%」を指定しています。

また、C言語の例で「DioInputByte」では、「&bValue」というように変数のポインタを渡しています。AJANではポインタの修飾は不要で、C言語のプロトタイプ関数宣言を読み取って、内部でポインタ処理を行います。

次に、出力の DioOutputPoint と 入力の DioInputPoint です。

C言語の例	AJANの例
int nRet; int Buffer[8];	DIM Buffer%(7) Buffer%(0) = 1

int nDevice;
 	
Buffer[0] = 1;
Buffer[1] = 0;
... <略>
Buffer[7] = 0;
// 任意の点数の出力
nRet = DioOutputPoint(nDevice, Buffer, 16, 8);
	Buffer%(1) = 0
	... <略>
	Buffer%(7) = 0
	' 任意の点数の出力
	errno = CFUNCALL(nRet%, "DioOutputPoint", nDevice, Buffer%, 16, 8)

int nRet;
int Buffer[8];
int nDevice;
// 任意の点数の入力
nRet = DioInputPoint(nDevice, Buffer, 16, 8);
	DIM Buffer%(7)
	' 任意の点数の入力
	errno = CFUNCALL(nRet%, "DioInputPoint", nDevice, Buffer%, 16, 8)

C言語の例では、接点の情報を受け渡しするのに、int型の配列 Buffer変数を指定しています。
8点分の情報を渡すのに、「int Buffer[8]」と記述しています。
C言語では、配列の要素数に8を指定すると、実際には 0 から 7 まで添字に指定できます。
AJANでは、配列は最大添字を指定するので、7を指定する事になります。

尚、配列を渡すのに、上のAJANの例では、「DIM Buffer%(7)」として単精度整数の配列変数を宣言しています。
他に、OBJECT 型の変数を宣言し、CALLOC@ 関数で必要なメモリ量を確保して渡す事も可能です。
この例では、int型の配列を渡すので、同じ単精度整数の配列を宣言して渡す事例としました。

4.2.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-2000は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-2000のオンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』が該当します。

参考に、PCI-2768CをPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg2000/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insdio.sh
④ドライバモジュール(cp2000)が組み込まれた事を確認
# lsmod | grep cp
cp2000                974848  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

**************************************************
 Setup Utility
--------------------------------------------------
 Version: 1.60-10
--------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
**************************************************

Enter the model number of the product: GPG/GPH-2000  <= ②「2000」を入力
============================================================
 Ref.ID | Model                          | RSW1 | Device No.
------------------------------------------------------------
      1 | PCI-2768C                      |    0 |     1
============================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
**************************************
```

```
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、DioOpen関数の呼び出しで、指定するデバイス番号です。
上の例では、PCI-2768Cのデバイス番号は1です。

4.2.3 ポイント＞入出力の各接点とビットの関係を把握する

GPG-2000の入力および出力の命令は、指定した接点の範囲を入力/出力する命令(DioInputByte, DioOutputByteなど)と、接点単位に配列で入力/出力する命令(DioInputPoint, DioOutputPoint)があります。

機能	関数名	説明
出力	DioOutputByte DioOutputWord DioOutputDword DioOutputPoint	接点 8 点分(8 ビット)の出力 接点 16 点分(16 ビット)の出力 接点 32 点分(32 ビット)の出力 任意の点数の出力
入力	DioInputByte DioInputWord DioInputDword DioInputPoint	接点 8 点分(8 ビット)の入力 接点 16 点分(16 ビット)の入力 接点 32 点分(32 ビット)の入力 任意の点数の入力

接点の範囲を入力/出力する命令では、接点の範囲を識別子という数値で指定します。
例えば、DioInputByteでは8点単位に入力するようになっており、識別子に FBIDIO_IN9_16 を指定した場合、IN9～IN16を一度に読み取る事ができます。

数値の各ビットが各接点に対応しており、最下位ビットが接点番号の始まりと解釈できます。

ビット位置	7	6	5	4	3	2	1	0
IN 番号	16	15	14	13	12	11	10	9

例えば、DioInputByte関数で、AJANで各接点の値を取り出そうとするならば、以下のように書く事ができます。

```
bValue% = 0
errno = CFUNCALL(nRet%, "DioInputByte", nDevice, &H2/* FBIDIO_IN9_16 */, bValue%)
ASSERT nRet% = 0, "error DioInputByte"

' IN9からIN16までの接点のON / OFF を TRUE / FALSE で表示する
BOOL IS_ON
FOR I=0 TO 7                                ' bit.0から7までを FORで回す
    IS_ON = nRet% AND (1 << I)              ' 接点毎のON / OFF を判定
    PRINT "IN番号"; 9 + I; " = "; IS_ON
NEXT I
```

仮に、DioInputWord関数だと、16点単位なので、FORループのTOに指定する値は 15となります。
このように、入出力の各接点とビット位置の関係を把握するだけで、簡単に まとめて入出力ができるようになります。

4.2.4 ポイント＞割り込みを使用する

GPG-2000では、デジタル入出力の割り込み要因を満たした時、割り込み発生時に登録しておいたコールバック関数を呼び出す機能があります。

GPG-2000で、コールバック関数の登録は、DioRegistIsr関数です。
C言語で呼び出し部分を抜き出すと、以下のようになります。

```
int   ret, nDevice;

void CallBackProc (unsigned long lUserData, unsigned char bEvent, unsigned short
nDeviceNum)
{
    ＜略＞
    return;
}

ret = DioRegistIsr (dnum, 0x00000000, CallBackProc );
```

上の例では、割り込み発生時に、DioRegistIsr関数で指定した CallBackProc 関数が、ドライバからコールバックされます。

AJANで、コールバック関数を使った呼び出しは、コールバック関数の処理部分をC言語で記述し、CIMPORT 命令で アプリケーションに組み込むように指示し、コールバック関数のポインタアドレスを取得する為に CGETADRS@ 関数で取得するようにします。

C言語で記述する部分を抜き出すと、以下のようになります。
前述のC言語のコールバック関数と、ほぼ変わらない事が判ります。

```
// test_dio.c というファイル名と仮定する
#include <stdio.h>
#include "fbidio.h"

void c_CallBackProc (unsigned long lUserData, unsigned char bEvent, unsigned short
nDeviceNum)
{
    ＜略＞
    return;
}
```

これを CIMPORT 命令で組み込んで、DioRegistIsr で指定する部分を例示します。

```
CIMPORT "test_dio.c", "-lgpg2000"
OBJECT  CallBackProc@ AS POINTER
CallBackProc@ = CGETADRS@("c_CallBackProc")
errno = CFUNCALL(ret%, "DioRegistIsr", dnum, &H00000000, CallBackProc@)
```

これにより、C言語で書いたコールバック関数を、ドライバのライブラリに渡して、呼び出し処理を行う事ができます。

4.3 GPG-2X72Cバスマスタ方式デジタル入出力ドライバの制御事例

4.3.1 基本手順

GPG-2X72Cは、弊社バスマスタ方式デジタル入出力機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg2x72c.so
ヘッダファイル	fbidiobm.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-2X72Cのヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbidiobm.h")      ' GPG-2X72C 要インストール
s$ = ICONV$(s$, "UTF-8", "EUCJP")                  ' EUC形式文字列→UTF-8文字列に変更
CDECLARE "libgpg2x72c.so", s$
```

!	2021/10時点で、fbidiobm.h の文字エンコードは EUCJP 形式の為、「ICONV$」関数を使って、AJANが認識可能な UTF-8形式に変換が必要です。
!	GPG-2X72Cのオンラインヘルプの説明では、32bit環境と、64bit環境およびPAE環境 に分けて使用方法が書かれています。 AJANの動作する Interface Linux System は 64bit環境ですので、64bit環境の説明を参考にしてください。

GPG-2X72Cのヘルプファイルでは、『制御手順』の章で、初期化、バスマスタ転送などの事例などが紹介されています。
この中でよく使うと思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	DioBmOpen	デバイスのオープン
終了処理	DioBmClose	デバイスのクローズ
各種設定	DioBmGetDeviceConfigEx DioBmGetDeviceConfig DioBmSetDeviceConfigEx DioBmSetDeviceConfig	モード、周波数、制御信号などの機能設定の取得 〃 モード、周波数、制御信号などの機能設定 〃
条件設定	DioBmSetTriggerConfig	スタート、ストップ、エンド条件の設定
転送バッファ	DioBmSetBufferConfig DioBmGetBufferConfig	バスマスタ転送バッファの設定 バスマスタ転送バッファの設定情報の取得
割り込み設定	DioBmSetEvent DioBmKillEvent	割り込みコールバックの設定 割り込みコールバックの設定解除
バスマスタ転送	DioBmStart	バスマスタ転送を開始
ステータス	DioBmGetStatus	ステータスの取得

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの 「DioBmOpen」 と デバイスクローズの 「DioBmClose」 です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// デバイスオープン
nRet = DioBmOpen(nDevice);
if(!nRet){
　... <略>
　// デバイスクローズ
　nRet = DioBmClose(nDevice);
}
	nDevice = 1
	' デバイスオープン
	errno=CFUNCALL(nRet, "DioBmOpen", nDevice)
	IF nRet = 0 THEN
	　... <略>
	　' デバイスクローズ
	　errno=CFUNCALL(nRet, "DioBmClose", nDevice)
	END IF

次にモード、転送単位、サンプリング周波数、制御信号の機能設定を行う「DioBmSetDeviceConfig」と 「DioBmSetDeviceConfigEx」、機能設定の取得を行う 「DioBmGetDeviceConfig」 と「DioBmGetDeviceConfigEx」 です。

C言語の例	AJANの例
int nRet; DIOBMCONF BmConf; BmConf.dwMode = DIOBM_MODE_SAMPLING; BmConf.dwRedirectWidth = DIOBM_WIDTH_DWORD; BmConf.dwEQCOnf = 0; BmConf.dwClock = DIOBM_CLK_100K; BmConf.dwSmplNum = 1024; BmConf.dwTimingOption = 0; BmConf.dwSTBConf = 0; BmConf.dwACKConf = 0; BmConf.dwREQConf = 0; BmConf.dwOREConf = 0; BmConf.dwUREConf = 0; nRet=DioBmSetDeviceConfig(nDevice, &BmConf);	OBJECT BmConf@ AS MEMORY BmConf@ = CSTRUCT@("DIOBMCONF") BmConf@.SETMEMVAL("dwMode", &h02/*DIOBM_MODE_SAMPLING*/) BmConf@.SETMEMVAL("dwRedirectWidth", &H4/*DIOBM_WIDTH_DWORD*/) BmConf@.SETMEMVAL("dwEQCOnf", 0) BmConf@.SETMEMVAL("dwClock", &h00010000/*DIOBM_CLK_100K*/) BmConf@.SETMEMVAL("dwSmplNum", 1024) BmConf@.SETMEMVAL("dwTimingOption", 0) BmConf@.SETMEMVAL("dwSTBConf", 0) BmConf@.SETMEMVAL("dwACKConf", 0) BmConf@.SETMEMVAL("dwREQConf", 0) BmConf@.SETMEMVAL("dwOREConf", 0) BmConf@.SETMEMVAL("dwUREConf", 0) ' 各種設定 errno=CFUNCALL(nRet%,"DioBmSetDeviceConfig", nDevice, BmConf@)
int nRet; DIOBMCONFEX BmConfEx; BmConfEx.dwMode = DIOBM_MODE_SAMPLING; BmConfEx.dwRedirectWidth = DIOBM_WIDTH_DWORD; BmConfEx.dwEQConf = 0; BmConfEx.dwClock = DIOBM_CLK_100K; BmConfEx.dwSmplNum = 1024; BmConfEx.dwTimingOption = 0;	OBJECT BmConfEx@ AS MEMORY BmConfEx@ = CSTRUCT@("DIOBMCONFEX") BmConfEx@.SETMEMVAL("dwMode", &H02/*DIOBM_MODE_SAMPLING*/) BmConfEx@.SETMEMVAL("dwRedirectWidth", &H4/* DIOBM_WIDTH_DWORD */) BmConfEx@.SETMEMVAL("dwEQConf", 0)

... BmConfEx.dwAssignExt[0] = 0; BmConfEx.dwAssignExt[1] = 0; BmConfEx.dwAssignExt[2] = 0; BmConfEx.dwAssignExt[3] = 0; ... nRet = DioBmSetDeviceConfigEx(nDevice, &BmConfEx);	BmConfEx@.SETMEMVAL("dwClock", &h00010000/*DIOBM_CLK_100K*/) BmConfEx@.SETMEMVAL("dwSmplNum", 1024) BmConfEx@.SETMEMVAL("dwTimingOption", 0) ... BmConfEx@.SETMEMVAL("dwAssignExt[0]", 0) BmConfEx@.SETMEMVAL("dwAssignExt[1]", 0) BmConfEx@.SETMEMVAL("dwAssignExt[2]", 0) BmConfEx@.SETMEMVAL("dwAssignExt[3]", 0) ... errno=CFUNCALL(nRet%,"DioBmSetDeviceConfigEx", nDevice, BmConfEx@)
int nRet; DIOBMCONF BmConf; nRet=DioBmGetDeviceConfig(nDevice, &BmConf);	OBJECT BmConf@ AS MEMORY BmConf@ = CSTRUCT@("DIOBMCONF") errno=CFUNCALL(nRet%,"DioBmGetDeviceConfig", nDevice, BmConf@)
int nRet; DIOBMCONFEX BmConfEx; nRet=DioBmGetDeviceConfigEx(nDevice,&BmConfEx);	OBJECT BmConfEx@ AS MEMORY BmConfEx@ = CSTRUCT@("DIOBMCONFEX") errno=CFUNCALL(nRet%,"DioBmGetDeviceConfigEx", nDevice, BmConfEx@)

C言語の例で、定数名「DIOBM_MODE_SAMPLING」または「DIOBM_WIDTH_DWORD」があります。
これは、ヘッダファイル(fbidiobm.h)にて、以下のように定義されていますので、即値を指定します。
(0x04 はC言語で16進数の定数値なので、AJANでは &H04 と記述します。)

```
...〈略〉
//モード設定
#define DIOBM_MODE_NON                      0x01    // モードなし（各端子オープン状態。電源立ち上げ時のデフォルト）
#define DIOBM_MODE_SAMPLING                 0x02    // サンプリングモード
#define DIOBM_MODE_SAMPLING_GATE            0x04    // ゲート付きサンプリングモード
#define DIOBM_MODE_PATTERN_OUT              0x08    // パターン出力モード
#define DIOBM_MODE_PATTERN_OUT_GATE         0x10    // ゲート付きパターン出力モード
...〈略〉

//バスマスタ転送幅
#define DIOBM_WIDTH_DWORD    0x04    //DWORD
#define DIOBM_WIDTH_WORD     0x02    //WORD
#define DIOBM_WIDTH_BYTE     0x01    //BYTE
...〈略〉
```

また、C言語の例で DIOBMCONFEX 構造体を使った「BmConfEx」変数を、AJANでは OBJECT 型の変数を宣言し、CSTRUCT@ 関数で "DIOBMCONFEX" で構造体に必要なメモリ量を確保しています。

CSTRUCT@ 関数 で確保した OBJECT 型の変数 では、SETMEMVAL や GETMEMVAL などのメソッドを使って、C言語で構造体のメンバを操作するのと同じように、アクセス操作が可能です。

4.3.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-2X72Cは初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-2X72Cのオンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PEX-291144をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg2x72c/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insdiobm.sh
④ドライバモジュール(cp2x72c)が組み込まれた事を確認
# lsmod | grep cp
cp2x72c               475136  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

**************************************************
 Setup Utility
--------------------------------------------------
 Version: 1.60-10
--------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
**************************************************

Enter the model number of the product: GPG/GPH-2x72c  <= ②「2x72c」を入力
================================================================
 Ref.ID | Model                        | RSW1 |  Type | Device No.
----------------------------------------------------------------
      1 | PEX-291144                   |    1 |    DI |    19
----------------------------------------------------------------
      2 | PEX-291144                   |    1 |    DO |    20
================================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
**************************************
```

```
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、DioBmOpen関数の呼び出しで、指定するデバイス番号です。
上の例では、PEX-291144 のデバイス番号は デジタル入力が 19 で、デジタル出力が 20 です。

4.3.3 ポイント＞バスマスタ転送バッファとアクセス方法

GPG-2X72Cでは、デジタル入出力のデータは、バスマスタ転送機能を使って入出力を行います。
バスマスタ転送バッファの設定を行う「DioBmSetBufferConfig」でバスマスタ転送用のメモリブロックを確保し、「DioBmGetBufferConfig」でメモリブロックのアドレスとサイズを取得します。
デジタル入力およびデジタル出力のデータは、このメモリブロックに配置されるので、取得したアドレスを介してデータの読み書きを行います。

以下に「DioBmSetBufferConfig」と「DioBmGetBufferConfig」の呼び出し事例を示します。

C言語の例	AJANの例

int nRet;
void* ptr;
unsigned long dwBufferSize;
 	
nRet=DioBmSetBufferConfig(nDevice, (void*)-1, 128 * sizeof(unsigned long));
 	
nRet = DioBmGetBufferConfig(nDevice, &ptr, &dwBufferSize);
	errno=CFUNCALL(nRet%,"DioBmSetBufferConfig", nDevice, -1&, 128* 8/*sizeof(unsigned long)*/)
	
	errno=CFUNCALL(nRet%,"DioBmGetBufferConfig", nDevice, ptr&, dwBufferSize&)
	
	' ptr& で受け取ったメモリアドレスを直接操作する為の定義
	OBJECT memptr@ AS MEMORY
	memptr@ = CMEMMAP@(ptr&, dwBufferSize&)

C言語の例で「DioBmGetBufferConfig」では、「&ptr」や「&dwBufferSize」というように変数のポインタを渡しています。AJANではポインタの修飾は不要で、C言語のプロトタイプ関数宣言を読み取って、内部でポインタ処理を行います。

C言語の例の「dwBufferSize」変数は unsigned long 型で、AJANでは倍精度整数に対応します。この為、AJANの例では、倍精度整数の変数を渡しています。

C言語の例の「ptr」変数は void* 型で、AJANに対応する型はありません。しかし、「ptr」変数には「DioBmSetBufferConfig」呼び出し時に確保したメモリブロックのアドレス値が得られます。このアドレス値を取得するのが目的なので、AJANの例では void* 型と同じサイズ長の倍精度整数の変数を渡して、アドレス値を受け取っています。
その後、受け取ったアドレスに対して直接メモリの読み書きができるように、OBJECT型の変数を宣言し、「CMEMMAP@」関数で指定したアドレスから任意のサイズ領域をアクセスできるように設定しています。

尚、メモリブロックに配置されるデジタル入出力の格納形式は、GPG-2X72Cのオンラインヘルプの

『データフォーマット』を参照ください。
例えば、32点単位でデジタル入力情報が格納されていると仮定した場合の、アクセス事例を以下に示します。

```
int           i;
void*         ptr;
unsigned long dwBufferSize;
unsigned int* pBuffer
...
nRet = DioBmGetBufferConfig(nDevice, &ptr, &dwBufferSize);

pBuffer = (unsigned int*)ptr;
for(i = 0; i < 1024; i++){
    printf("%d: %#x¥n", i, *(pBuffer + i));
}
```

```
errno=CFUNCALL(nRet%,"DioBmGetBufferConfig", nDevice, ptr&, dwBufferSize&)

' ptr& で受け取ったメモリアドレスを直接操作する為の定義
OBJECT memptr@ AS MEMORY
memptr@ = CMEMMAP@(ptr&, dwBufferSize&)

FOR I=0 TO 1024-1
    PRINT I;":";HEX$(memptr@.PEEKINT(I * 4))
NEXT I
```

AJANは、C言語のようにアドレス値を受け取ったポインタを使って直接読み書きは出来ないので、「CMEMMAP@」関数 を使って OBJECT型の変数に メモリアクセス用の領域を設定し、メソッドを使ってメモリの読み書きを行います。

4.3.4 ポイント＞割り込みを使用する

GPG-2X72Cでは、デジタル入出力の割り込み要因を満たした時、登録しておいたコールバック関数を呼び出す機能があります。

GPG-2X72Cで、コールバック関数を登録するには、「DioBmSetEvent」関数を使用します。
C言語で呼び出し部分を抜き出すと、以下のようになります。

```
int   ret;
unsigned long dwEventData;

void CallBackProc (unsigned long dwEvent, unsigned long dwUser)
{
    ＜略＞
    return;
}

ret = DioBmSetEvent (nDevice, DIOBM_EV_ENDTRG, CallBackProc, 0, &dwEventData);
```

上の例では、割り込み発生時に「DioBmSetEvent」関数で指定した CallBackProc 関数が、ドライバからコールバックされます。

AJANでコールバック関数を使った呼び出しは、コールバック関数の処理部分をC言語で記述し、CIMPORT 命令で アプリケーションに組み込むように指示し、コールバック関数のポインタアドレスをCGETADRS@ 関数で取得します。

C言語で記述する部分を抜き出すと、以下のようになります。
前述のC言語のコールバック関数と、ほぼ変わらない事が判ります。

```
// test_dio.c というファイル名と仮定する
#include <stdio.h>
#include "fbidio.h"

void c_CallBackProc (unsigned long dwEvent, unsigned long dwUser)
{
  ＜略＞
  return;
}
```

これを CIMPORT 命令で組み込んで、DioBmSetEvent で指定する部分を例示します。

```
CIMPORT "test_dio.c", ""
OBJECT  CallBackProc@ AS POINTER
CallBackProc@ = CGETADRS@("c_CallBackProc")
OBJECT  dwEventData@ AS MEMORY
dwEventData@ = CALLOC@(4)
errno = CFUNCALL(ret%, "DioBmSetEvent", nDevice, DIOBM_EV_ENDTRG, CallBackProc@, 0,
dwEventData@)
```

これにより、C言語で書いたコールバック関数を、ドライバのライブラリに渡して、呼び出し処理を行う事ができます。

上の例で、DioBmSetEvent の 5番目の引数に dwEventData@ 変数を渡しています。
ここでは、CALLOC@ で用意したメモリを渡すのが肝要です。
割り込みが発生すると、ここで指定したメモリアドレスに対して、ドライバが値を書き込みます。
メモリアドレスは DioBmSetEvent 呼び出し時に固定しておく必要があります。
このため、CALLOC@ でメモリ位置を固定した変数値を渡します。
倍精度整数の配列などを引数として渡すと、メモリ位置の固定が保証されないため、想定外の動作を起こす可能性があります。

4.4 GPG-3100 アナログ入力ドライバの制御事例

4.4.1 基本手順

GPG-3100は、弊社アナログ入力機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg3100.so
ヘッダファイル	fbiad.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-3100のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbiad.h") ' GPG-3100 要インストール
CDECLARE "libgpg3100.so", s$
```

GPG-3100のヘルプファイルでは、『制御手順』の章で、サンプリングの方法と1件のアナログ入力の事例などが紹介されています。
この中で、よく使われると思われる関数を以下に抜粋します。

機能	関数名	説明
初期化	AdOpen	デバイスのオープン
デバイス情報	AdGetDeviceInfo	デバイス仕様の取得
サンプリング条件	AdGetSamplingConfig AdSetSamplingConfig	現在のサンプリング条件を取得 サンプリング条件を設定
サンプリング開始	AdStartSampling	サンプリング開始
データ取得	AdGetSamplingData	サンプリングデータの取得
1件アナログ入力	AdInputAD	1 件のアナログ入力
割り込み	AdSetBoardConfig	コールバックの設定
動作状態取得	AdGetStatus	サンプリング動作状態の取得
終了処理	AdClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの AdOpen と デバイスクローズの AdClose です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
nDevice = 1
// デバイスオープン
nRet = AdOpen(nDevice);
if(!nRet){
  ... <略>
  // デバイスクローズ
	nDevice = 1
	' デバイスオープン
	errno = CFUNCALL(nRet, "AdOpen", nDevice)
	IF nRet = 0 THEN
	  ... <略>
	  ' デバイスクローズ
	  errno = CFUNCALL(nRet, "AdClose", nDevice)

nRet = AdClose(nDevice); }	END IF

次に、1件4ch分のアナログ入力(AdInputAD) です。

C言語の例	AJANの例
int nRet; unsigned short usData[4]; ADSMPLCHREQ SmplChReq[4]; SmplChReq[0].ulChNo = 1; SmplChRet[0].ulRange = AD_5V; SmplChReq[1].ulChNo = 3; SmplChReq[1].ulRange = AD_5V; ... <略> // 1 件 4ch 分のアナログ入力 nRet = AdInputAD(nDevice, 4, AD_INPUT_SINGLE, &SmplChReq[0], &usData);	OBJECT usData@ AS MEMORY usData@ = CALLOC@(2 * 4) OBJECT SmplChReq@ AS MEMORY SmplChReq@ = CSTRUCT@("ADSMPLCHREQ", 4) SmplChReq@.SETMEMVAL(0, "ulChNo", 1) SmplChReq@.SETMEMVAL(0, "ulRange", &H40000/* AD_5V */) SmplChReq@.SETMEMVAL(1, "ulChNo", 3) SmplChReq@.SETMEMVAL(1, "ulRange", &H40000/* AD_5V */) ... <略> ' 1 件 4ch 分のアナログ入力 errno = CFUNCALL(nRet, "AdInputAD", nDevice, 4, 1/* AD_INPUT_SINGLE */, SmplChReq@, usData@)

C言語の例で、定数名「AD_INPUT_SINGLE」または「AD_5V」があります。
これは、ヘッダファイル(fbiad.h)にて、以下のように定義されていますので、即値を指定します。
(0x00040000 はC言語で16進数の定数値なので、AJANでは &H00040000 と記述します)

```
... <略>
#define AD_INPUT_SINGLE          1
#define AD_INPUT_DIFF            2
... <略>
#define AD_2P5V                  0x00020000
#define AD_5V                    0x00040000
#define AD_10V                   0x00080000
... <略>
```

また、C言語の例で ADSMPLCHREQ 構造体を使った「SmplChReq」配列変数を、AJANでは OBJECT 型の変数を宣言し、CSTRUCT@ 関数の引数に "ADSMPLCHREQ" を指定して構造体に必要なメモリ量と要素数 4を指定して、全体で必要なメモリを確保しています。

CSTRUCT@ 関数 で確保した OBJECT 型の変数 では、SETMEMVAL や GETMEMVAL などのメソッドを使って、C言語で構造体のメンバを操作するのと同じように、アクセス操作が可能です。

また、C言語の例で アナログデータを格納する為に「usData」配列変数を宣言しているのを、AJANでは OBJECT 型の変数を宣言し、CALLOC@ 関数で アナログデータを格納する為のメモリを確保しています。
これは、「usData」配列変数が unsigned short型(2バイト長)の整数であり、AJANには同じサイズ長の整数型が無い為、このようにしています。(単精度整数型は4バイト長)

次に、サンプリング条件を扱う AdGetSamplingConfig と AdSetSamplingConfig です。

C言語の例	AJANの例

ADSMPLREQ SmplConfig;
int nRet;
 	
// サンプリング条件の取得
nRet=AdGetSamplingConfig(nDevice,&SmplConfig);
// 4チャンネル指定
SmplConfig.ulChCount = 4;
SmplConfig.SmplChReq[0].ulChNo = 1;
SmplConfig.SmplChReq[0].ulRange = AD_5V;
SmplConfig.SmplChReq[1].ulChNo = 3;
SmplConfig.SmplChReq[1].ulRange = AD_5V;
... <略>
 	
// サンプリング条件の設定
nRet=AdSetSamplingConfig(nDevice,&SmplConfig);
	OBJECT SmplConfig@ AS MEMORY
	SmplConfig@ = CSTRUCT@("ADSMPLREQ")
	
	' サンプリング条件の取得
	errno = CFUNCALL(nRet, "AdGetSamplingConfig", nDevice, SmplConfig@)
	' 4チャンネル指定
	SmplConfig@.SETMEMVAL("ulChCount", 4)
	SmplConfig@.SETMEMVAL("SmplChReq[0].ulChNo", 1)
	SmplConfig@.SETMEMVAL("SmplChReq[0].ulRange", &H40000/* AD_5V */)
	SmplConfig@.SETMEMVAL("SmplChReq[1].ulChNo", 3)
	SmplConfig@.SETMEMVAL("SmplChReq[1].ulRange", &H40000/* AD_5V */)
	... <略>
	
	' サンプリング条件の設定
	errno = CFUNCALL(nRet, "AdSetSamplingConfig", nDevice, SmplConfig@)

AdInputADの事例と同じように、C言語の例で ADSMPLREQ 構造体を使った「SmplConfig」変数を、AJANでは OBJECT 型の変数を宣言し、CSTRUCT@ 関数の引数に "ADSMPLREQ" を指定して構造体に必要なメモリ量を確保しています。
構造体のメンバのアクセス方法は前述のとおりです。

次に、サンプリングスタートを行う AdStartSampling と、サンプリングデータを取得する AdGetSamplingData です。

C言語の例	AJANの例

int nRet;
unsigned long Len;
unsigned short AdData[100 * 4];
 	
// サンプリングスタート
nRet = AdStartSampling(nDevice, FLAG_SYNC);
if(!nRet){
  Len = 100;
  // サンプリングデータの取得
  nRet=AdGetSamplingData(nDevice,&AdData[0], &Len);
}
	OBJECT AdData@ AS MEMORY
	AdData@ = CALLOC@(2 * 100 * 4)
	
	' サンプリングスタート
	errno = CFUNCALL(nRet, "AdStartSampling", nDevice, 1/* FLAG_SYNC */)
	IF nRet = 0 THEN
	  Len = 100
	  ' サンプリングデータの取得
	  errno = CFUNCALL(nRet, "AdGetSamplingData", nDevice, AdData@, Len)
	END IF

AdInputADの事例と同じように、C言語の例でアナログデータを格納する為に「AdData」配列変数

を宣言しているのを、AJANでは OBJECT 型の変数を宣言し、CALLOC@ 関数で アナログデータを格納する為のメモリ(100件 × 4ch分 × 2(1件のサイズ))を確保しています。

次に、割り込みを扱う AdSetBoardConfig です。

C言語の例	AJANの例

int nRet;
 	
int event_proc(int dummy)
{
  printf("The sampling is successfully completed.¥n");
}
 	
// 割り込みコールバックの設定
nRet=AdSetBoardConfig(nDevice, 0, event_proc, 0);
if(!nRet){
  nRet=AdStartSampling(nDevice, FLAG_ASYNC);
}
	' sub.c の中身が、↓これと仮定します
	' int c_event_proc(int dummy)
	' {
	' printf("The sampling is successfully completed.¥n");
	' }
	CIMPORT "sub.c"
	CDECLARE "", "int c_event_proc(int dummy);"
	
	OBJECT event_proc@ AS POINTER
	event_proc@ = CGETADRS@("c_event_proc")
	
	' 割り込みコールバックの設定
	errno = CFUNCALL(nRet, "AdSetBoardConfig", nDevice, 0, event_proc@, 0)
	IF nRet = 0 THEN
	  errno = CFUNCALL(nRet, "AdStartSampling", nDevice, 2/* FLAG_ASYNC */)
	END IF

C言語の例で AdSetBoardConfig は、サンプリング停止時に コールバックする event_proc 関数を定義して渡しています。

AJANの場合、以下のように定義します。

1. CIMPORT 命令で、C言語で書かれたコールバックしたいサブルーチンのファイルを指定します。
   これにより、アプリケーション内にC言語で書かれたサブルーチンのコードが組み込まれます。
   尚、AJANから呼び出すサブルーチン名は、「c_」または「test」をプレフィクスとする必要があります。(static 修飾していない公開可能なサブルーチンが該当します)
2. CDECLARE 命令で、ライブラリ名を空文字列として、呼び出し関数の定義に 1.項で定義したサブルーチンのプロトタイプ関数宣言を指定します。
   これにより、AJANからサブルーチンを呼び出せるようになります。
3. CGETADRS@ 関数で、サブルーチンの関数アドレスを取得します。
4. CFUNCALL 関数で、AdSetBoardConfig 関数を呼び出すようにして、コールバック関数を指定する所に、3.項で得た サブルーチンの関数アドレスを指定します。

4.4.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-3100は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-3100のオンラインヘルプでは『実行手順』の章の『デバイスを動か

すまで』項が該当します。

参考に、PCI-3521をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg3100/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insad.sh
④ドライバモジュール(cp3100)が組み込まれた事を確認
# lsmod | grep cp
cp3100                352256  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

*************************************************
 Setup Utility
-------------------------------------------------
 Version: 1.60-10
-------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
*************************************************

Enter the model number of the product: GPG/GPH-3100   <= ②「3100」を入力
=============================================================
 Ref.ID | Model                        | RSW1 |  ADDA | Device No.
-------------------------------------------------------------
      1 | PCI-3521                     |    1 |    ad |     1
=============================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
**************************************
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、AdOpen関数の呼び出しで指定するデバイス番号です。
上の例では、PCI-3521のデバイス番号は 1 です。

4.4.3 ポイント＞ADは1件のアナログ入力とサンプリングに大別できる

GPG-3100でアナログ入力を行う時、1件のアナログ入力を行う関数群と、サンプリングを行う関数群に大別する事ができます。
任意のタイミングでアナログ値をスナップショットのように読み取りたい場合は 1件のアナログ入力を、アナログ値の時系列変化を観測したい場合はサンプリングを用います。

機能	関数名	説明
サンプリング条件	AdGetSamplingConfig AdSetSamplingConfig	現在のサンプリング条件を取得 サンプリング条件を設定
サンプリング開始	AdStartSampling	サンプリング開始
データ取得	AdGetSamplingData	サンプリングデータの取得
1件アナログ入力	AdInputAD	1 件のアナログ入力
動作状態取得	AdGetStatus	サンプリング動作状態の取得

1件のアナログ入力を行う場合は、例えば AdInputAD 関数を呼び出すだけで、呼び出し時点のアナログ値を読み取れます。

```
OBJECT usData@ AS MEMORY
usData@ = CALLOC@(2 * 4)
OBJECT SmplChReq@ AS MEMORY
SmplChReq@ = CSTRUCT@("ADSMPLCHREQ", 4)

SmplChReq@.SETMEMVAL(0, "ulChNo", 1)
SmplChReq@.SETMEMVAL(0, "ulRange", &H40000/* AD_5V */)
SmplChReq@.SETMEMVAL(1, "ulChNo", 3)
SmplChReq@.SETMEMVAL(1, "ulRange", &H40000/* AD_5V */)
... <略>
' 1 件 4ch 分のアナログ入力
errno = CFUNCALL(nRet, "AdInputAD", nDevice, 4, 1/* AD_INPUT_SINGLE */, SmplChReq@,
usData@)
```

サンプリングを行う場合は、サンプリング条件の設定(AdSetSamplingConfig)、サンプリング開始(AdStartSampling)、サンプリングデータの取得(AdGetSamplingData) などの組み合わせが必要です。

```
OBJECT SmplConfig@ AS MEMORY
SmplConfig@ = CSTRUCT@("ADSMPLREQ")

errno = CFUNCALL(nRet, "AdGetSamplingConfig", nDevice, SmplConfig@)

SmplConfig@.SETMEMVAL("ulChCount", 4)
SmplConfig@.SETMEMVAL("SmplChReq[0].ulChNo", 1)
SmplConfig@.SETMEMVAL("SmplChReq[0].ulRange", &H40000/* AD_5V */)
SmplConfig@.SETMEMVAL("SmplChReq[1].ulChNo", 2)
SmplConfig@.SETMEMVAL("SmplChReq[1].ulRange", &H40000/* AD_5V */)
... <略>
' サンプリング条件の設定
errno = CFUNCALL("AdSetSamplingConfig", nDevice, SmplConfig@)
```

```
OBJECT AdData@ AS MEMORY
AdData@ = CALLOC@(2 * 100 * 4)

' サンプリング開始
errno = CFUNCALL(nRet, "AdStartSampling", nDevice, 1/* FLAG_SYNC */)
IF nRet = 0 THEN
  Len = 100
  ' サンプリングデータの取得
  errno = CFUNCALL(nRet, "AdGetSamplingData", nDevice, AdData@, Len)
END IF
```

尚、サンプリング開始時、サンプリング入力処理の引数で同期処理(FLAG_SYNC、値は1)でなく、非同期処理(FLAG_ASYNC、値は2)を指定して呼び出した場合、サンプリングの動作状態を取得する関数(AdGetStatus)などを定期的に呼び出すか、別の手段でもってサンプリングが完了するのを待機する必要があります。

```
OBJECT AdData@ AS MEMORY
AdData@ = CALLOC@(2 * 100 * 4)

' サンプリング開始
errno = CFUNCALL(nRet, "AdStartSampling", nDevice, 2/* FLAG_ASYNC */)
ASSERT nRet = 0, "fail start"

' サンプリングが停止するまでループして動作監視し続ける
DO WHILE TRUE
  ' サンプリング動作状態を取得
  errno = CFUNCALL(nRet, "AdGetStatus", nDevice, stat&, smpCnt&, availCnt&)
  ASSERT nRet = 0, "fail getStatus"

  IF stat& = 1 THEN EXIT DO      ' サンプリング停止したらループを抜ける
  SLEEP 0.1
LOOP

' サンプリングデータの取得
Len = 100
errno = CFUNCALL(nRet, "AdGetSamplingData", nDevice, AdData@, Len)
```

4.4.4 ポイント＞アナログ入力のデジタル値から電圧値を求める

GPG-3100で取得したアナログ入力はデジタル値であり、電圧値や電流値として読み取るには、変換式により値を求める必要があります。

変換方法は、GPG-3100オンラインヘルプの、『機能説明』章の『使用上の注意事項』項に書いてあります。

ここで、デジタル値から電圧を求める変換式は、以下のように記載されています。

```
電圧 =（ レンジ上限 －　レンジ下限 ）＊ デジタル値 / 分解能 + レンジ下限
```

レンジ上限と下限は、使用する入力レンジで決まります。
例えば以下のような組み合わせとなります。

入力レンジ	レンジ上限	レンジ下限
ユニポーラ 0～5V	5	0
ユニポーラ 0～10V	10	0
バイポーラ ±5V	5	-5
バイポーラ ±10V	10	-10

分解能は、デジタル値で表現できる値の範囲を示します。
12ビット分解能は 0～4095までの値で表現し、16ビット分解能は 0～65535までの値で表現するという意味になります。
上記式に当て嵌める値は、以下のような組み合わせとなります。

分解能	指定値
8	256
12	4096
16	65536

例えば、入力レンジ：バイポーラ ±5Vで、分解能：12ビット、デジタル値が 3000 の時、変換式に当て嵌めると、以下のようになります。

```
電圧 =（ レンジ上限 －　レンジ下限 ）＊ デジタル値 / 分解能 + レンジ下限
     ↓
PRINT （ 5 －（-5)）＊ 3000 / 4096 +（-5)
2.32421875
```

尚、使用上の注意事項に「変換データのデジタル値の最大値に相当する電圧は、レンジの上限になりません」とあります。
これはどういう事かと言いますと、例えば 分解能：12ビットの時、デジタル値の取る範囲は、0 から 4095 です。これを式に当て嵌めて計算してみると判ります。

```
’デジタル値に最小値:0 を当て嵌める
PRINT （ 5 －（-5)）＊ 0 / 4096 +（-5)
-5
```

```
’デジタル値に最大値:4095 を当て嵌める
PRINT ( 5 - (-5)) * 4095 / 4096 + (-5)
4.99755859375
```

上記の通り、最大値 4095 を当て嵌めた時、得られる電圧値は、レンジの上限値でない事に留意が必要です。

ここまで、デジタル値から電圧値への変換方法について説明しましたが、この変換作業を簡単にする為に、AJANでは CONVPHY 関数が用意されています。
CONVPHY関数は、デジタル値と分解能、レンジ上限、レンジ下限を渡す事で変換式と同様の計算を行います。
先ほどの値をCONVPHY関数で当て嵌めると、以下の結果が得られます。

```
PRINT CONVPHY(3000, 4096, 5, -5)
[ 2.32421875 ]
```

変換式を使って求めた時と、同じ結果が得られる事が判ります。

4.4.5 ポイント＞割り込みを使用する

GPG-3100では、サンプリング停止時に、登録しておいたコールバック関数を呼び出す機能があります。
GPG-3100でコールバック関数を登録するには、AdSetBoardConfig関数を使用します。
C言語で呼び出し部分を抜き出すと、以下のようになります。

```
void event_proc ( int dummy )
{
  ＜略＞
  return;
}

ret = AdSetBoardConfig (nDevice, 0, event_proc, 0 );
```

上の例では、サンプリング停止時にAdSetBoardConfig 関数で指定した event_proc 関数が、ドライバからコールバックされます。

AJANで、コールバック関数を使った呼び出しは、コールバック関数の処理部分をC言語で記述し、CIMPORT 命令で アプリケーションに組み込むように指示し、コールバック関数のポインタアドレスを取得する為に CGETADRS@ 関数で取得するようにします。

C言語で記述する部分を抜き出すと、以下のようになります。
前述のC言語のコールバック関数と、ほぼ変わらない事が判ります。

```
// test_ad.c というファイル名と仮定する
#include <stdio.h>

void c_event_proc (int dummy)
{
  ＜略＞
  return;
}
```

これを CIMPORT 命令で組み込んで、AdSetBoardConfig で指定する部分を例示します。

```
CIMPORT "test_ad.c", "-lgpg3100"
OBJECT  event_proc@ AS POINTER
event_proc@ = CGETADRS@("c_event_proc")
errno = CFUNCALL(ret%, "AdSetBoardConfig", nDevice, 0, event_proc@, 0)
```

これにより、C言語で書いたコールバック関数を、ドライバのライブラリに渡して、呼び出し処理を行う事ができます。

4.5 GPG-3300 アナログ出力ドライバの制御事例

4.5.1 基本手順

GPG-3300は、弊社アナログ出力機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg3300.so
ヘッダファイル	fbida.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-3300のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbida.h") ' GPG-3300 要インストール
s$ = ICONV$(s$, "UTF-8", "EUCJP")          ' EUC形式文字列→UTF-8文字列に変更
's$ = REPLACE$(s$, "int DaOpenEx", "//int DaOpenEX")    ' 2021/1/14時点の未実装関数です
CDECLARE "libgpg3300.so", s$
```

!	2021/1時点で、fbida.h の文字エンコードは EUCJP 形式の為、ICONV$関数を使って、AJANが理解可能な UTF-8形式に変換が必要です。
!	2021/1時点で、libgpg3300.so に fbida.h で宣言されている一部関数が存在しない為、CDECLARE命令呼び出し時、宣言した関数が存在しない警告が表示されます。 警告が気になる場合は、上のREPLACE$のコメント部分を外してください。 C言語のコメント文を付加する事で、プロトタイプ宣言対象から外します。

GPG-3300のヘルプファイルでは、『制御手順』の章で、連続アナログ出力の方法と1件のアナログ出力の事例などが紹介されています。
この中で、よく使われると思われる関数を以下に抜粋します。

機能	関数名	説明
初期化	DaOpen	デバイスのオープン
デバイス情報	DaGetDeviceInfo	デバイス仕様の取得
バッファサイズと割り込み	DaSetBoardConfig	バッファサイズとコールバックの設定
連続アナログ出力条件	DaGetSamplingConfig DaSetSamplingConfig	現在の連続アナログ出力条件を取得 連続アナログ出力条件を設定
データ設定	DaSetSamplingData	連続アナログ出力データの設定
連続アナログ出力開始	DaStartSampling	連続アナログ出力開始
1件アナログ出力	DaOutputDA	1 件のアナログ出力
終了処理	DaClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの DaOpen と デバイスクローズの DaClose です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// デバイスオープン
nRet = DaOpen(nDevice);
if(!nRet){
  ... <略>
  // デバイスクローズ
  nRet = DaClose(nDevice);
}
	nDevice = 1
	' デバイスオープン
	errno = CFUNCALL(nRet, "DaOpen", nDevice)
	IF nRet = 0 THEN
	  ... <略>
	  ' デバイスクローズ
	  errno = CFUNCALL(nRet, "DaClose", nDevice)
	END IF

次に、1件4ch分のアナログ出力の DaOutputDA です。

C言語の例	AJANの例
int nRet; DASMPLCHREQ SmplChReq[4]; unsigned short Data[4]; SmplChReq[0].ulChNo = 1; SmplChReq[1].ulChNo = 3; ... <略> Data[0] = 0xffff; Data[1] = 0x0000; Data[2] = 0xffff; Data[3] = 0x0000; ... <略> // 1 件 4ch 分のアナログ出力 nRet = DaOutputDA(nDevice, 4, &SmplChReq[0], &Data);	OBJECT Data@ AS MEMORY Data@ = CALLOC@(2 * 4) OBJECT SmplChReq@ AS MEMORY SmplChReq@ = CSTRUCT@("DASMPLCHREQ", 4) SmplChReq@.SETMEMVAL(0, "ulChNo", 1) SmplChReq@.SETMEMVAL(1, "ulChNo", 3) ... <略> Data@.POKEHALF(0, &Hffff) Data@.POKEHALF(2, 0) Data@.POKEHALF(4, &Hffff) Data@.POKEHALF(6, 0) ... <略> ' 1 件 4ch 分のアナログ出力 errno = CFUNCALL(nRet, "DaOutputDA", nDevice, 4, SmplChReq@, Data@)

C言語の例で DASMPLCHREQ 構造体を使った「SmplChReq」配列変数を、AJANでは OBJECT 型の変数を宣言し、CSTRUCT@ 関数の引数に "DASMPLCHREQ" を指定して構造体に必要なメモリ量と、要素数 4を指定して、全体で必要なメモリを確保しています。

CSTRUCT@ 関数 で確保した OBJECT 型の変数 では、SETMEMVAL や GETMEMVAL などのメソッドを使って、C言語で構造体のメンバを操作するのと同じように、アクセス操作が可能です。

また、C言語の例でアナログ出力データを格納する為に「Data」配列変数を宣言しているのを、AJANでは OBJECT 型の変数を宣言し、CALLOC@ 関数で アナログ出力データを格納する為のメモリを確保しています。
これは、「Data」配列変数が unsigned short型(2バイト長)の整数であり、AJANには同じサイズ長の整数型が無い為、このようにしています。(単精度整数型は4バイト長)
AJANには CALLOC@ 関数で取得したメモリ型オブジェクトに、2バイト単位で読み書きするメソッド(PEEKHALF, POKEHALF)があるので、これを使ってC言語の事例と同じように記述する事が可能です。

次に、連続アナログ出力条件を扱う DaGetSamplingConfig と DaSetSamplingConfig です。

C言語の例	AJANの例

DASMPLREQ SmplConfig;
int nRet;
 	
// 連続アナログ出力条件の取得
nRet=DaGetSamplingConfig(nDevice,&SmplConfig);
 	
SmplConfig.ulChCount = 2;
SmplConfig.SmplChReq[0].ulChNo = 1;
SmplConfig.SmplChReq[1].ulChNo = 2;
... <略>
 	
// 連続アナログ出力条件の設定
nRet=DaSetSamplingConfig(nDevice,&SmplConfig);
 	
	OBJECT SmplConfig@ AS MEMORY
	SmplConfig@ = CSTRUCT@("DASMPLREQ")
	
	' 連続アナログ出力条件の取得
	errno = CFUNCALL(nRet, "DaGetSamplingConfig", nDevice, SmplConfig@)
	
	SmplConfig@.SETMEMVAL("ulChCount", 2)
	SmplConfig@.SETMEMVAL("SmplChReq[0].ulChNo", 1)
	SmplConfig@.SETMEMVAL("SmplChReq[1].ulChNo", 2)
	... <略>
	
	' 連続アナログ出力条件の設定
	errno=CFUNCALL("DaSetSamplingConfig",nDevice, SmplConfig@)

DaOutputDAの事例と同じように、C言語の例で DASMPLREQ 構造体を使った「SmplConfig」変数を、AJANでは OBJECT 型の変数を宣言し、CSTRUCT@ 関数の引数に "DASMPLREQ" を指定して構造体に必要なメモリ量を確保しています。
構造体メンバのアクセス方法は前述のとおりです。

次に、連続アナログ出力データを設定する DaSetSamplingData と、連続アナログ出力スタートを行う DaStartSampling です。

C言語の例	AJANの例

int i, nRet;
unsigned short SmplData[4096][2];
 	
nRet = DaClearSamplingData(nDevice)
 	
// アナログ出力データの作成
for(i = 0; i < 4096; i++){
  SmplData[i][0] = i
  SmplData[i][1] = 4095 - i;
}
 	
// アナログ出力データの設定
nRet=DaSetSamplingData(nDevice,
&SmplData[0][0], 4096);
if(!nRet){
  // 連続アナログ出力スタート
  nRet = DaStartSampling(nDevice, FLAG_SYNC);
}
 	
	OBJECT SmplData@ AS MEMORY
	SmplData@ = CALLOC@(2 * 2 * 4096)
	
	errno = CFUNCALL(nRet, "DaClearSamplingData", nDevice)
	
	' アナログ出力データの作成
	IDX = 0
	FOR I=0 TO 4096 - 1
	  SmplData@.POKEHALF(IDX, I)
	  IDX = IDX + 2
	  SmplData@.POKEHALF(IDX, 4095 - I)
	  IDX = IDX + 2
	NEXT I
	
	'アナログ出力データの設定
	errno = CFUNCALL(nRet, "DaSetSamplingData", nDevice, SmplData@)
	IF nRet = 0 THEN
	  ' 連続アナログ出力スタート

	errno = CFUNCALL(nRet, "DaStartSampling", nDevice, 1/* FLAG_SYNC */) END IF

DaOutputDAの事例と同じように、C言語の例でアナログ出力データを格納する為に「SmplData」配列変数を宣言しているのを、AJANでは OBJECT 型の変数を宣言し、CALLOC@ 関数で アナログデータを格納する為のメモリを確保しています。

次に、バッファサイズの設定と割り込みを扱う DaSetBoardConfig です。

C言語の例	AJANの例

int nRet;
unsigned long ulSmplBufferSize;
 	
int event_proc(int dummy)
{
printf("completed.¥n");
}
 	
ulSmplBufferSize = 2048;
// バッファサイズと割り込みの設定
nRet=DaSetBoardConfig(nDevice,
ulSmplBufferSize, 0, event_proc, 0);
	' sub.c の中身が、↓これと仮定します
	' int c_event_proc(int dummy)
	' {
	' printf("completed.¥n");
	' }
	CIMPORT "sub.c"
	CDECLARE "", "int c_event_proc(int dummy);"
	
	OBJECT event_proc@ AS POINTER
	event_proc@ = CGETADRS@("c_event_proc")
	
	ulSmplBufferSize = 2048
	' バッファサイズと割り込みの設定
	errno=CFUNCALL(nRet,"DaSetBoardConfig",
	nDevice, ulSmplBufferSize, 0, event_proc@, 0)

C言語の例でDaSetBoardConfig は、連続アナログ出力停止時に コールバックする event_proc 関数を定義して渡しています。

AJANの場合、以下のように定義する事が必要です。

1. CIMPORT 命令で、C言語で書かれたコールバックしたい サブルーチンのファイルを指定します。これにより、アプリケーション内に、C言語で書かれたサブルーチンのコードが組み込まれます。
   尚、AJANから呼び出すサブルーチン名は、「c_」または「test」をプレフィクスとする必要があります。(static 修飾していない公開可能なサブルーチンが該当します)
2. CDECLARE 命令で、ライブラリ名を空文字列として、呼び出し関数の定義に1.項で定義したサブルーチンのプロトタイプ関数宣言を指定します。
   これにより、AJANから、サブルーチンを呼び出せるようになります。
3. CGETADRS@ 関数で、サブルーチンの関数アドレスを取得します。
4. CFUNCALL 関数で、DaSetBoardConfig 関数を呼び出すようにして、コールバック関数を指定する所に、3.項で得た サブルーチンの関数アドレスを指定します。

4.5.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-3300は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-3300オンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PCI-3521をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg3300/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insda.sh
④ドライバモジュール(cp3300)が組み込まれた事を確認
# lsmod | grep cp
cp3300                200704  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

*************************************************
 Setup Utility
-------------------------------------------------
 Version: 1.60-10
-------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
*************************************************

Enter the model number of the product: GPG/GPH-3300  <= ②「3300」を入力
=================================================================
 Ref.ID | Model                          | RSW1 |  ADDA | Device No.
-----------------------------------------------------------------
      1 | PCI-3521                       |    1 |    da |      1
=================================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
**************************************
```

```
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、DaOpen関数の呼び出しで、指定するデバイス番号です。
上の例では、PCI-3521のデバイス番号は 1 です。

4.5.3 ポイント＞DAは1件のアナログ出力と連続出力に大別できる

GPG-3300でアナログ出力を行う時、1件のアナログ出力を行う関数群と、指定周期で連続出力を行う関数群に大別する事ができます。
任意のタイミングでアナログ値を出力したい場合は 1件のアナログ出力を、アナログ値を一定周期で出力したい場合は連続出力を用います。

機能	関数名	説明
バッファサイズと割り込み	DaSetBoardConfig	バッファサイズとコールバックの設定
連続出力条件	DaGetSamplingConfig DaSetSamplingConfig	現在の連続出力条件を取得 連続出力条件を設定
データ設定	DaSetSamplingData	連続出力データの設定
連続出力開始	DaStartSampling	連続出力開始
1件アナログ出力	DaOutputDA	1 件のアナログ出力

1件のアナログ出力を行う場合は、DaOutputDA関数を呼び出すだけで、アナログ出力を変更できます。

```
OBJECT Data@ AS MEMORY
Data@ = CALLOC@(2 * 4)
OBJECT SmplChReq@ AS MEMORY
SmplChReq@ = CSTRUCT@("DASMPLCHREQ", 4)

SmplChReq@.SETMEMVAL(0, "ulChNo", 1)
SmplChReq@.SETMEMVAL(1, "ulChNo", 3)
... <略>
Data@.POKEHALF(0, &Hffff)
Data@.POKEHALF(2, 0)
Data@.POKEHALF(4, &Hffff)
Data@.POKEHALF(6, 0)
... <略>
' 1 件のアナログ出力
errno = CFUNCALL(nRet, "DaOutputDA", nDevice, 4, SmplChReq@, Data@)
```

連続出力を行う場合は、連続出力条件の設定(DaSetSamplingConfig)、連続出力データのセット(DaSetSamplingData)，連続出力開始(DaStartSampling) などの組み合わせが必要です。

```
OBJECT SmplConfig@ AS MEMORY
SmplConfig@ = CSTRUCT@("DASMPLREQ")

errno = CFUNCALL(nRet, "DaGetSamplingConfig", nDevice, SmplConfig@)

SmplConfig@.SETMEMVAL("ulChCount", 2)
SmplConfig@.SETMEMVAL("SmplChReq[0].ulChNo", 1)
SmplConfig@.SETMEMVAL("SmplChReq[1].ulChNo", 2)
... <略>
' 連続出力条件の設定
```

```
errno = CFUNCALL("DaSetSamplingConfig", nDevice, SmplConfig@)

OBJECT SmplData@ AS MEMORY
SmplData@ = CALLOC@(2 * 2 * 4096)

errno = CFUNCALL(nRet, "DaClearSamplingData", nDevice)

' アナログ出力データの作成
IDX = 0
FOR I=0 TO 4096 - 1
  SmplData@.POKEHALF(IDX, I)
  IDX = IDX + 2
  SmplData@.POKEHALF(IDX, 4095 - I)
  IDX = IDX + 2
NEXT I

' 連続出力データのセット
errno = CFUNCALL(nRet, "DaSetSamplingData", nDevice, SmplData@)
IF nRet = 0 THEN
  ' 連続出力開始
  errno = CFUNCALL(nRet, "DaStartSampling", nDevice, 1/* FLAG_SYNC */)
END IF
```

尚、連続出力開始時、連続出力処理の引数で 同期処理(FLAG_SYNC、値は1)でなく、非同期処理(FLAG_ASYNC、値は2)を指定して呼び出した場合、連続出力の動作状態を取得する関数(DaGetStatus)などを定期的に呼び出すか、別の手段でもって連続出力が完了するのを待機する必要があります。

```
' 連続出力開始
errno = CFUNCALL(nRet, "DaStartSampling", nDevice, 2/* FLAG_ASYNC */)
ASSERT nRet = 0, "fail start"

' 連続出力が停止するまでループして動作監視し続ける
DO WHILE TRUE
  ' 連続出力動作状態を取得
  errno=CFUNCALL(nRet,"DaGetStatus",nDevice,stat&,smpCnt&,availCnt&,availRep&)
  ASSERT nRet = 0, "fail getStatus"

  IF stat& = 1 THEN EXIT DO      ' 連続出力が停止したらループを抜ける
  SLEEP 0.1
LOOP
```

4.5.4 ポイント＞電圧値からアナログ出力のデジタル値を求める

GPG-3300でセットするアナログ出力はデジタル値にしなければなりません。電圧値や電流値からデジタル値を得るには、変換式により値を求める必要があります。

変換方法は、GPG-3300のreadmeのFAQ項に書いてあります。

ここで、電圧値からデジタル値を求める変換式は、以下のように記載されています。

```
デジタル値 =（ 分解能 *（電圧 -　レンジ下限））/（レンジ上限 -　レンジ下限）
```

レンジ上限と下限は、使用する出力レンジで決まります。
例えば以下のような組み合わせとなります。

出力レンジ	レンジ上限	レンジ下限
ユニポーラ 0～5V	5	0
ユニポーラ 0～10V	10	0
バイポーラ ±5V	5	-5
バイポーラ ±10V	10	-10

分解能は、デジタル値で表現できる値の範囲を示します。
12ビット分解能は 0～4095までの値で表現し、16ビット分解能は 0～65535までの値で表現するという意味になります。
上記式に当て嵌める値は、以下のような組み合わせとなります。

分解能	指定値
8	256
12	4096
16	65536

例えば、出力レンジ：バイポーラ ±5Vで、分解能：12ビット、2.5Vの電圧値を出力したい時、変換式に当て嵌めると、以下のようになります。

```
デジタル値 =（ 分解能 *（電圧 -　レンジ下限））/（レンジ上限 -　レンジ下限）
     ↓
PRINT （ 4096 *（ 2.5 - (-5))) / (5 - (-5))
3072
```

尚、注意事項に「変換データデジタル値の最大値に相当する電圧は、レンジの上限になりません」とあります。
これはどういう事かと言いますと、例えば 分解能：12ビットの時、デジタル値の取る範囲は、0 から 4095 です。式にレンジの上限と下限を当て嵌めて計算してみると判ります。

```
’電圧値にレンジ下限:-5 を当て嵌める
PRINT （ 4096 *((-5) - (-5))) / (5 - (-5))
0

’電圧値にレンジ上限:5 を当て嵌める
PRINT （ 4096 *(5 - (-5))) / (5 - (-5))
```

```
4096    '<== デジタル値の範囲 0 から 4095 を超えている
```

上記の通り、レンジ上限を当て嵌めて式を計算すると、とりうる範囲(0から4095)を超えてしまう事がわかります。即ち、範囲最大値に対応する電圧値は、レンジ上限より少し低いという事です。

ここまで、デジタル値から電圧値への変換方法について説明しましたが、この変換作業を簡単にする為に、AJANでは CONVBIN 関数が用意されています。
CONVBIN関数は、電圧値(=アナログ値)と分解能、レンジ上限、レンジ下限を渡す事で変換式と同様の計算を行います。
先ほどの値を、CONVBIN関数で当て嵌めると、以下の結果が得られます。

```
PRINT CONVBIN(2.5, 4096, 5, -5)
[ 3072 ]
```

変換式を使って求めた時と、同じ結果が得られる事が判ります。

4.5.5 ポイント＞割り込みを使用する

GPG-3300では、連続出力停止時に登録しておいたコールバック関数を呼び出す機能があります。
GPG-3300で、コールバック関数を登録するにはDaSetBoardConfig関数を使用します。
C言語で呼び出し部分を抜き出すと、以下のようになります。

```
void event_proc ( int dummy )
{
  ＜略＞
  return;
}

int   smpBufferSize = 2048;
ret = DaSetBoardConfig (nDevice, smpBufferSize, 0, event_proc, 0 );
```

上の例では、連続出力停止時にDaSetBoardConfig 関数で指定した event_proc 関数が、ドライバからコールバックされます。

AJANで、コールバック関数を使った呼び出しは、コールバック関数の処理部分をC言語で記述し、CIMPORT 命令で アプリケーションに組み込むように指示して、コールバック関数のポインタアドレスを取得する為に CGETADRS@ 関数で取得するようにします。

C言語で記述する部分を抜き出すと、以下のようになります。
前述のC言語のコールバック関数と、ほぼ変わらない事が判ります。

```
// test_da.c というファイル名と仮定する
#include <stdio.h>

void c_event_proc (int dummy)
{
  ＜略＞
  return;
}
```

これを CIMPORT 命令で組み込んで、DaSetBoardConfig で指定する部分を例示します。

```
CIMPORT "test_da.c", "-lgpg3300"
OBJECT  event_proc@ AS POINTER
event_proc@ = CGETADRS@("c_event_proc")
smpBufferSize = 2048
errno = CFUNCALL(ret%, "DaSetBoardConfig", nDevice, smpBufferSize, 0, event_proc@, 0)
```

これにより、C言語で書いたコールバック関数を、ドライバのライブラリに渡して、呼び出し処理を行う事ができます。

4.6 GPG-4116 HDLC通信ドライバの制御事例

4.6.1 基本手順

GPG-4116は、弊社HDLC通信機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg4116.so
ヘッダファイル	pcihdlc.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-4116のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/pcihdlc.h")       ' GPG-4116 要インストール
's$ = REPLACE$(s$, "long HdlcSetPriority", "//long HdlcSetPriority")    ' 2021/1/14時点の、未実装関数です
's$ = REPLACE$(s$, "long HdlcGetPriority", "//long HdlcGetPriority")    ' 2021/1/14時点の、未実装関数です
CDECLARE "libgpg4116.so", s$
```

!	2021/1時点で、libgpg4116.so に pcihdlc.h で宣言されている一部関数が存在しない為、CDECLARE命令呼び出し時、宣言した関数が存在しない警告が表示されます。 警告が気になる場合は、上のREPLACE$のコメント部分を外してください。 C言語のコメント文を付加する事で、プロトタイプ宣言対象から外します。

GPG-4116のヘルプファイルでは、『制御手順(HDLC通信部)』の章で、初期化、送信、受信の事例などが紹介されています。
この中で、よく使われると思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	HdlcOpen	デバイスのオープン
送信	HdlcSendFrame	HDLC フレームの送信
受信	HdlcGetFrameCount HdlcGetFrameLength HdlcReceiveFrame	HDLC 受信フレーム数の取得 HDLC 受信フレーム長の取得 HDLC 受信フレームの取得
ステータス確認	HdlcGetStatus	ステータス確認
終了処理	HdlcClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの HdlcOpen と デバイスクローズの HdlcClose です。
HdlcOpen関数は、引数に構造体のポインタを必要とします。構造体の各メンバで各種設定を行い、これを引数として渡します。以下のように呼び出し可能です。

C言語の例	AJANの例

long lRet;
int nPort = HDLC_PORTNO_4116_0;
HDLCPORTINITDATA Port;
Port.ulBaudRate = 250000;
...<略>
// デバイスオープン
lRet = HdlcOpen(nPort, &Port);
	nPort = 1/* HDLC_PORTNO_4116_0 */
	OBJECT Port@ AS MEMORY
	Port@ = CSTRUCT("HDLCPORTINITDATA")
	Port@.SETMEMVAL("ulBaudRate", 250000)
	...<略>
	' デバイスオープン
	errno = CFUNCALL(lRet, "HdlcOpen", nPort, Port@)

int nPort = HDLC_PORTNO_4116_0;
long lRet;
// デバイスクローズ
lRet = HdlcClose(nPort);
	nPort = 1/* HDLC_PORTNO_4116_0 */
	' デバイスクローズ
	errno = CFUNCALL(lRet, "HdlcClose", nPort)

次に、HDLCフレームの送信を行う HdlcSendFrame と、受信フレーム数を得る HdlcGetFrameCount、受信を行う HdlcReceiveFrame です。

C言語の例	AJANの例

int nPort = HDLC_PORTNO_4116_0;
long lRet;
unsigned char data[5];
data[0] = 1;
data[1] = 2;
data[2] = 3;
data[3] = 4;
data[4] = 5;
// HDLC フレームの送信
lRet = HdlcSendFrame(nPort, data, 5);
	nPort = 1/* HDLC_PORTNO_4116_0 */
	data$ = CHRB$(1)
	data$ = data$ + CHRB$(2)
	data$ = data$ + CHRB$(3)
	data$ = data$ + CHRB$(4)
	data$ = data$ + CHRB$(5)
	errno = CFUNCALL(lRet, "HdlcSendFrame", nPort, data$, LENB(data$))

int nPort = HDLC_PORTNO_4116_0;
long lRet;
long frmCnt;
unsigned char data[16384];
unsigned long length;
 	
// HDLC 受信フレーム数の取得
lRet = HdlcGetFrameCount(nPort, &frmCnt);
for(n = 0; n < frmCnt; n++){
// HDLC 受信フレームの取得
lRet = HdlcReceiveFrame(nPort, data, &length);
}
	nPort = 1/* HDLC_PORTNO_4116_0 */
	
	' HDLC 受信フレーム数の取得
	errno = CFUNCALL(lRet, "HdlcGetFrameCount", nPort, frmCnt)
	
	FOR N=0 TO frmCnt - 1
	data$ = STR_REPEAT$(CHRB$(0), 16384)
	' HDLC 受信フレームの取得
	errno = CFUNCALL(lRet, "HdlcReceiveFrame", nPort, data$, length&)
	NEXT N

C言語で、フレーム送受信に使う unsigned char型の配列は、AJANでは文字列型として置き換える事ができます。
フレーム受信で、例では 16384要素の配列を準備して受け取るようにしています。AJANでは空のバッファとして STR_REPEAT$ 関数を使って 16384バイト長の文字列を作り、これを渡しています。

4.6.2 ポイント＞HdlcOpen呼び出しは、初回スーパーユーザモード(root権限)で実行が必要

GPG-4116のオンラインヘルプ、HdlcOpen関数の備考欄に注意書きがあります。
以下、抜粋します。

> ＜GPG-4116 オンラインヘルプより抜粋＞
> 本関数で初めてポートをオープンする時にデバイスノードが作成されます。そのため最初の実行時にはスーパーユーザ権限が必要になります。2 回目以降に同じポートをオープンする場合には、すでにデバイスノードが作成されているため、スーパーユーザである必要はありません。

デバイスノードが作られてないと、HdlcOpenの呼び出しはエラーとなります。

尚、HdlcOpenに指定するポート番号は、GPG-4116のオンラインヘルプで『デバイスを動かすまで』の章の、『通信ポートの確認』項を参考にしてください。

以下は、PCI-4172 をPCIスロットに挿して、通信ポートを確認した事例です。

```
$ cat /proc/driver/hdlc/cp4116
cp4116 info:1.1
49: PCI-4172(bid=0h) CH1  [9600bps, FD] tx:0 rx:0
50: PCI-4172(bid=0h) CH2  [9600bps, FD] tx:0 rx:0
```

ここでは、指定できる通信ポートは、49 と 50 になります。

4.7 GPG-4141 調歩同期シリアル通信ドライバの制御事例

4.7.1 基本手順

GPG-4141は、弊社調歩同期シリアル通信機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	標準 C ライブラリを使用します(libc.so.6)
ヘッダファイル	主に標準 C ライブラリ向けのヘッダを使用します。 この為、AJAN 向けに GPG4141_EX.AJN を用意しました。

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-4141の呼び出し準備を行います(GPG-4141 要インストール)
INCLUDE "GPG4141_EX.AJN"
' GPG-4141 の呼び出しで必要なプロトタイプ宣言
CDECLARE "libc.so.6", GPG4141_PROTOTYPE$
```

調歩同期シリアル通信制御は、標準Cライブラリが提供する関数を使用する為、AJAN向けに「GPG4141_EX.AJN」を用意しています。
これを INCLUDE すると、通信制御に用いる関数プロトタイプ宣言を「GPG4141_PROTOTYPE$」変数に、通信制御に用いる #define の定数設定を AJANの CONST 文で代用する処理を行うようにしています。

GPG-4141のヘルプファイルでは、『制御手順』の章で、調歩同期通信の事例などが紹介されています。
この中で、よく使われると思われる関数を以下に抜粋します。

機能	関数名	説明
初期化	open	ポートのオープン
通信設定	tcgetattr tcsetattr tcflush	通信設定の取得 通信設定の設定 通信設定のフラッシュ
送信	write	データの送信
受信	read	データの受信
その他	ioctl	詳細情報の取得、半二重切り替えなど
終了	close	ポートのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、ポートオープンの open と、 ポートクローズの close です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int fd;
// ポートオープン
fd = open("/dev/ttyS64", O_RDWR);
... <略>
// ポートクローズ
close(fd);
	fd = 0
	' ポートオープン
	errno = CFUNCALL(fd, "open", "/dev/ttyS64", O_RDWR)
	... <略>
	' ポートクローズ
	errno = CFUNCALL(ret, "close", fd)

	シリアル通信は、デバイスノードの権限により、管理者権限(スーパーユーザ)で使用する必要があります。 openに失敗した時、errno が 13(EACCES) の場合、権限不足によりオープンに失敗した事が判ります。
	シリアル通信のデバッグ中に、open 出来なくなったりした場合、ドライバのサンプルを動かして正常通りに動いているか確認する事をお奨めします。 例えば、ドライバモジュールを組み込み忘れているかも知れません。あるいは、認識しているデバイス名が想定と異なっているかも知れません。

通信パラメータの設定(例えば通信ボーレートなど)は、tcsetattr を使用します。
現在の情報取得は、tcgetattr で得られます。

C言語の例	AJANの例

struct termios cfg;
// 現在の設定値を取得
tcgetattr(fd, &cfg);
// 以下の条件で通信設定を変更
// データ長 8bit、1 ストップビット
// フロー制御なし、ローカルエコーなし
cfg.c_cflag &= ~(CRTSCTS | CSIZE);
cfg.c_cflag |= (CS8 | CLOCAL | CREAD);
cfg.c_iflag &= ~(IXON | IXOFF | ICRNL | ISTRIP);
cfg.c_oflag = 0;
cfg.c_lflag &= ~(ICANON | ECHO);
tcsetattr(fd, TCSANOW, &cfg);
	OBJECT cfg@ AS MEMORY
	cfg@ = CSTRUCT@("termios")
	// 現在の設定値を取得
	errno = CFUNCALL(ret%, "tcgetattr", fd, cfg@)
	// 以下の条件で通信設定を変更
	// データ長 8bit、1 ストップビット
	// フロー制御なし、ローカルエコーなし
	v% = cfg@.GETMEMVAL("c_cflag")
	v% = v% and not(CRTSCTS& or CSIZE)
	v% = v% or (CS8 or CLOCAL or CREAD)
	cfg@.SETMEMVAL("c_cflag", v%)
	v% = cfg@.GETMEMVAL("c_iflag")
	v% = v% and not(IXON or IXOFF or ICRNL or ISTRIP)
	cfg@.SETMEMVAL("c_iflag", v%)
	cfg@.SETMEMVAL("c_oflag", 0)
	v% = cfg@.GETMEMVAL("c_lflag")
	v% = v% and not(ICANON or ECHO)
	cfg@.SETMEMVAL("c_lflag", v%)
	errno = CFUNCALL(ret%, "tcsetattr", fd, cfg@)

データを送信するには、write を使用します。

C言語の例	AJANの例

char buff[256];
size_t buff_size;
// buff に、送信データを格納
strcpy(buff, "The quick brown fox jumps over the lazy dog.");
// データ長の取得
buff_size = strlen(buff);
// データ送信
write(fd, buff, buff_size);
	' buff$ に、送信データを格納
	buff$ = "The quick brown fox jumps over the lazy dog."
	' データ長の取得
	buff_size = LENB(buff$)
	' データ送信
	errno = CFUNCALL(ret%, "write", fd, buff$, buff_size)

データを受信するには、read を使用します。

C言語の例	AJANの例

char buff[256];
// 受信を行います
ret = read(fd, buff, 256);
// 受信したデータを表示します
if(ret > 0){
buff[ret] = '¥0';
printf("receive:%s¥n", buff);
}
	buff$ = STR_REPEAT$(" ", 256)
	' 受信を行います
	errno = CFUNCALL(ret%, "read", fd, buff$, LENB(buff$))
	' 受信したデータを表示
	IF ret% > 0 THEN
	buff$ = LEFTB$(buff$, ret%)
	PRINT "receive:"; buff$
	END IF

4.7.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-4141は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-4141オンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PEX-466102をPCIExpressスロットに挿して、ドライバモジュールを組み込む作業を例示します。

組み込むドライバモジュール名は、使用する製品により異なります。
オンラインヘルプの『ドライバモジュールの組み込み』の一覧表を参考ください。

```
①スーパーユーザーになる
$ su
②ドライバモジュールの組み込み
# modprobe cp4161
③ドライバモジュール(cp4851)が組み込まれた事を確認
# lsmod | grep cp
cp4161                188416  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、/proc ファイルシステムを参照する事で、ボードとデバイス名を確認します。

```
$ cat /proc/tty/driver/cp4161
cp4161 info:5.41.38.00
ttyG0: PEX-466102(bid=0h) CH1  [9600bps, FD] tx:0 rx:0
ttyG1: PEX-466102(bid=0h) CH2  [9600bps, FD] tx:0 rx:0
ttyG2: PEX-466102(bid=0h) CH1  [9600bps, FD] tx:0 rx:0
ttyG3: PEX-466102(bid=0h) CH2  [9600bps, FD] tx:0 rx:0
```

上の例では、PEX-466102 の CH1 が ttyG0 というデバイス名である事が判ります。

4.7.3 ポイント＞拡張機能の呼び出し

通信機能の細かい制御を、ioctl関数を用いて行う事ができます。

「CP4141_GET_BAUDRATE」を用いると、現在の通信ボーレートを取得できます。

AJANの例

```
' ボーレートの取得
errno = CFUNCALL(ret%, "ioctl", fd%, CP4141_GET_BAUDRATE, v&)
? "CP4141_GET_BAUDRATE=", errno, ret%
? "ボーレート="; v&
' ↑115200 などが表示される
```

「CP4141_GET_PORTINFO」を用いると、ポートの詳細情報を取得できます。

AJANの例

```
' ボーレートの取得
OBJECT info@ AS MEMORY
info@ = CSTRUCT@("CP4141_PORTINFO")
errno = CFUNCALL(ret%, "ioctl", fd%, CP4141_GET_PORTINFO, info@)
? "CP4141_GET_PORTINFO=", errno, ret%
? info@.tojson$()
```

「CP4141_GET_HSFUNCTION」を用いると、拡張通信設定の情報を取得できます。

AJANの例

```
' ボーレートの取得
OBJECT info@ AS MEMORY
info@ = CSTRUCT@("CP4141_PORTINFO")
errno = CFUNCALL(ret%, "ioctl", fd%, CP4141_GET_PORTINFO, info@)
? "CP4141_GET_PORTINFO=", errno, ret%
? info@.tojson$()
```

「TIOCMGET」を用いると、制御信号の状態を取得できます。

AJANの例

```
' 制御信号の取得
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMGET, st%)
? "TIOCMGET=", errno, ret%
? BIN$(st%, 16)
```

幾つかの制御信号は任意に操作可能です。「TIOCMBIS」で任意の信号をONにし、「TIOCMBIC」で任意の信号をOFFにできます。
以下に、RTS信号をON → OFF にします。

AJANの例

```
' 制御信号の取得
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMGET, st%)
? "TIOCMGET=", errno, ret%
? BIN$(st%, 16)

' RTS 信号を ON に
st% = TIOCM_RTS
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMBIS, st%)
? "TIOCMBIS=", errno, ret%

' 制御信号の取得
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMGET, st%)
? "TIOCMGET=", errno, ret%
? BIN$(st%, 16)

' RTS 信号を OFF に
st% = TIOCM_RTS
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMBIC, st%)
? "TIOCMBIS=", errno, ret%

' 制御信号の取得
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMGET, st%)
? "TIOCMGET=", errno, ret%
? BIN$(st%, 16)
```

4.7.4 ポイント＞RS-485特有の通信制御

RS-485通信は、RS-232C通信と比べて、様々な通信方式を使用する事ができます。
通信方式の設定は、ioctl 関数を用いて、特別な IOCTL コントロールを呼び出す事で操作可能です。

RS-485通信は、全二重通信、2線式半二重通信、4線式半二重通信の通信方式を選択可能です。
2線式半二重通信(CP4141_HALD_DUPLEX_2W)に設定した後、正しく設定されたか通信方式を再取得してみます。

!	2線式半二重通信は、T信号のみを接続して半二重通信を行います。 4線式半二重通信は、TとR信号を接続して半二重通信を行います。

AJANの例

```
' 2 線式半二重に通信方式を設定
v& = CP4141_HALF_DUPLEX_2W
errno = CFUNCALL(ret%, "ioctl", fd%, CP4141_SET_DUPLEX_MODE, v&)
? "set_duplex_mode=", errno, ret%

' 通信方式の取得
errno = CFUNCALL(ret%, "ioctl", fd%, CP4141_GET_DUPLEX_MODE, v&)
? "GET_DUPLEX_MODE=", errno, ret%
? "通信方式="; v&
```

RS-485通信で、通信方式に半二重通信を選択したとします。送信時に、送信有効・受信無効に切り替え→データ送信→送信有効・受信有効に戻す。といった手順を踏む必要があります。

AJANの例

```
' 送信有効・受信無効に切り替え
st% = TIOCM_RTS
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMBIS, st%)
' データ送信
msg$ = "Hello AJAN world"
errno = CFUNCALL(ret%, "write", fd%, msg$, LENB(msg$))
' 送信データが回線に送出されるまで待つ
do while TRUE
    errno = CFUNCALL(ret%, "ioctl", fd%, TIOCSERGETLSR, st%)
    if st% and TIOCSER_TEMT then
        exit do
    end if
loop
' 送信無効・受信有効に切り替え
st% = TIOCM_RTS
errno = CFUNCALL(ret%, "ioctl", fd%, TIOCMBIC, st%)
```

4.8 GPG-4301 GP-IB通信ドライバの制御事例

4.8.1 基本手順

GPG-4301は、弊社GP-IB通信機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg4301.so
ヘッダファイル	pcigpib.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-4301のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/pcigpib.h")      ' GPG-4301 要インストール
CDECLARE "libgpg4301.so", s$
```

GPG-4301のヘルプファイルでは『制御手順』の章で、初期化、データ送信、データ受信の事例などが紹介されています。
この中で、よく使われると思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	PciGpibExInitBoard	デバイスのオープン
信号送出	PciGpibExSetIfc PciGpibExSetRen	IFC 信号送出 REN 信号送出
データ送信	PciGpibExSendData	データ送信
データ受信	PciGpibExRecvData	データ受信
終了処理	PciGpibExFinishBoard	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの PciGpibExInitBoard と、デバイスクローズの PciGpibExFinishBoard です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int ret;
int nDevice = 0;
// デバイスオープン
ret = PciGpibExInitBoard(nDevice, 0);
... 〈略〉
// デバイスクローズ
PciGpibExFinishBoard(nDevice);
	nDevice = 0
	' デバイスオープン
	errno = CFUNCALL(ret, "PciGpibExInitBoard", nDevice, 0)
	... 〈略〉
	' デバイスクローズ
	errno = CFUNCALL(ret, "PciGpibExFinishBoard", nDevice)

次に、IFC信号を送出する PciGpibExSetIfc と REN信号を送出する PciGpibExSetRen です。

C言語の例	AJANの例

int ret;
// IFC 信号の送出
ret = PciGpibExSetIfc(nDevice, 1);
	' IFC 信号の送出
	errno = CFUNCALL(ret, "PciGpibExSetIfc", nDevice, 1)

int ret;
// REN 信号の送出
ret = PciGpibExSetRen(nDevice);
	' REN 信号の送出
	errno = CFUNCALL(ret, "PciGpibExSetRen", nDevice)

次に、データを送信する PciGpibExSendData と データを受信する PciGpibExRecvData です。

C言語の例	AJANの例

int ret;
int npAdrsTbl[2] = { 2, -1 };
char* msg = "0123456789";
// データの送信
ret = PciGpibExSendData(nDevice, npAdrsTbl, strlen(msg), msg, 0);
	DIM npAdrsTbl(1)
	npAdrsTbl = [2; -1]
	msg$ = "0123456789"
	' データの送信
	errno = CFUNCALL(ret, "PciGpibExSendData", nDevice, npAdrsTbl, LENB(msg$), msg$, 0)

int ret;
int npAdrsTbl[2] = { 2, -1 };
unsigned long length;
char recvBuf[100];
// データの受信
ret = PciGpibExRecvData(nDevice, npAdrsTbl, &length, recvBuf, 0);
	DIM npAdrsTbl(1)
	npAdrsTbl = [2; -1]
	length = 0
	recvBuf$ = STR_REPEAT$(CHRB$(0), 100)
	' データの受信
	errno = CFUNCALL(ret, "PciGpibExRecvData", nDevice, npAdrsTbl, length, recvBuf$, 0)

4.8.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-4301は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-4301のオンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PCI-4301をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg4301/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insgpibhg.sh
④ドライバモジュール(cp4301)が組み込まれた事を確認
# lsmod | grep cp
cp4301                  53248  0
ifcpmgr                 81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

 ***************************************************
 Setup Utility
 -------------------------------------------------
 Version: 1.60-10
 -------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
 ***************************************************

Enter the model number of the product: GPG/GPH-4301   <= ②「4301」を入力
 ============================================================
 Ref.ID | Model                          | RSW1 | Device No.
 ------------------------------------------------------------
      1 | PCI-4301                       |    0 |     0
 ============================================================

 ************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
 **************************************
```

```
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、PciGpibInitBoard関数の呼び出しで、指定するデバイス番号です。
上の例では、PCI-4301のデバイス番号は 0 です。

4.8.3 ポイント＞GP-IB計測器を制御するとき

マルチメータなどのGP-IB計測器を制御したい時、GPG-4301オンラインヘルプの『実行手順』章の『コントローラとしての実行』項を参考にしてください。

ここで重要なのは、『GP-IBインタフェースの初期化』項で示されている、PciGpibExSetIfc 関数で IFC信号の送出と、PciGpibExSetRen 関数で REN信号の送出です。
これにより、GP-IB計測器がリモート制御できる状態になります。

```
' デバイスのオープン
nDevice = 0
errno = CFUNCALL(ret, "PciGpibExInitBoard", nDevice, 0)
PRINT "init=", ret, errno
ASSERT ret = 0, "初期化エラー"

' IFC 信号の送出
errno = CFUNCALL(ret, "PciGpibExSetIfc", nDevice, 1)       ' 100us
PRINT "ifc=", ret, errno
ASSERT ret = 0, "IFC エラー"

' REN 信号の送出
errno = CFUNCALL(ret, "PciGpibExSetRen", nDevice)
PRINT "ren=", ret, errno
ASSERT ret = 0, "REN エラー"
```

次に重要なのは、制御する対象である GP-IB計測器のアドレスです。
これは送受信する際、どのアドレスのGP-IB計測器と通信するか指定する必要があるからです。
送受信時、間違ったアドレスを指定すると通信できません。

!	幾つかのGP-IB計測器では、電源を入れるとパネルに「ADDR <数値>」という具合に、自身のアドレス値が一定時間表示されるものがあります。 GP-IB計測器のマニュアルを参照してください。

ほとんどのGP-IB計測器は、1次アドレス(プライマリアドレスとも呼ばれます)を扱います。
2次アドレス(セカンダリアドレスとも呼ばれます)を扱う事は稀です。
複数のGP-IB計測器を同時に制御する場合、アドレスが被らないように GP-IB計測器のアドレスを設定しておくことが肝要です。

GPG-4301の場合、アドレスを指定する際、整数配列中に アドレスと、終端値に － 1 を指定する形式をとります。
以下は、1次アドレスが 22 のGP-IBデバイスに対して、送信しようとする例です。

```
DIM   adrsTbl(1)
adrsTbl = [ 22; -1 ]    ' アドレスをテーブルに記述

sdata$ = "*IDN?"
' メッセージを送信
```

```
errno = CFUNCALL(ret, "PciGpibExSendData", nDevice, adrsTbl, LENB(sdata$), sdata$, 0)
```

多くのGP-IB計測器は、送受信時に可読可能な文字列でやり取りします。数値も数文字の形式でやり取りするパターンが多いです。
(全てのGP-IB計測器に当てはまる訳ではありません。詳細は制御しようとするGP-IB計測器のマニュアルを参照ください)

サンプリングデータなどの、複数件のデータをやり取りする際、任意ブロック形式(Definite-Length Block Response Data)などと呼ばれるバイナリ値を含んだデータを受け渡しする事があります。
この形式のデータは、フォーマットが少々複雑なので、AJANでは BINBLK2ARRAY 関数やARRAY2BINBLK$ 関数を使って、数値配列と任意ブロック形式のバイナリ文字列を相互に変換できます。

4.9 GPG-4851 CAN通信ドライバの制御事例

4.9.1 基本手順

GPG-4851は、弊社CAN通信機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libifcan.so
ヘッダファイル	ifcan.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-4851のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/ifcan.h") ' GPG-4851をインストール必須です
CDECLARE "libifcan.so", s$
```

GPG-4851のヘルプファイルでは、『実行手順』の章で、CANメッセージの送受信の事例などが紹介されています。
この中で、よく使われると思われる関数を以下に紹介します。

機能	関数名	説明
初期化	CanOpenPort CanActivate	デバイスのオープン CANバスに接続
各種設定の取得	CanGetConfig	各種設定の取得
メッセージ送信	CanSendMessage CanGetCompletionMessage	メッセージの送信 送信完了メッセージの確認
メッセージ周期送信	CanCyclicSendMessage CanStopCyclicSendMessage	データの周期送信 周期送信の停止
メッセージ受信	CanReceiveMessage	メッセージの受信
その他	CanGetStatus	各種ステータスの取得
終了処理	CanDeactivate CanClosePort	CANバスから切り離し デバイスのクローズ

CAN FD版CAN製品を使用の場合、CAN FDメッセージの送受信が可能です。
先に挙げた関数と異なる CAN FD専用の関数を、幾つか以下に抜粋します。

機能	関数名	説明
各種設定の取得	CanGetConfigFD	各種設定の取得
メッセージ送信	CanSendMessageFD CanGetCompletionMessageFD	メッセージの送信 送信完了メッセージの確認
メッセージ周期送信	CanCyclicSendMessageFD	データの周期送信
メッセージ受信	CanReceiveMessageFD	メッセージの受信
その他	CanGetStatusFD	各種ステータスの取得

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの CanOpenPort と、 デバイスクローズの CanClosePort です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int fd;
// デバイスオープン
fd = CanOpenPort("ifcan1");
... 〈略〉
// デバイスクローズ
CanClosePort(fd);
	fd = 0
	' デバイスオープン
	errno = CFUNCALL(fd, "CanOpenPort", "ifcan1")
	... 〈略〉
	' デバイスクローズ
	errno = CFUNCALL(ret, "CanClosePort", fd)

デバイスをオープンした後、CANバスに接続しないとメッセージは送信できません。CANバスへの接続は、CanActivate を使用します。

C言語の例	AJANの例

int fd;
int ret;
// デバイスオープン
fd = CanOpenPort("ifcan1");
... 〈略〉
// CAN バスへの接続
ret = CanActivate(fd);
	fd = 0
	' デバイスオープン
	errno = CFUNCALL(fd, "CanOpenPort", "ifcan1")
	... 〈略〉
	' CAN バスへの接続
	errno = CFUNCALL(ret, "CanActivate", fd)

CANメッセージを送信するには CanSendMessage を、CAN FDメッセージを送信するには CanSendMessageFD を使用します。
送信するメッセージは、CAN_MESSAGE 構造体 または CAN_MESSAGE_FD 構造体で、組み立てる必要があります。

CAN通信の場合＞

C言語の例	AJANの例

CAN_MESSAGE msg;
msg.ulLength = 2; // 送信メッセージ長
msg.ulID = 0x30;
msg.ulFlag = 0;
msg.ulTime = 0;
msg.bData[0] = 0x11;
msg.bData[1] = 0x22;
// 1 件のメッセージを送信
CanSendMessage(fd, &msg, 1);
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE")
	msg@.SETMEMVAL("ulLength", 2) ' メッセージ長
	msg@.SETMEMVAL("ulID", &H30)
	msg@.SETMEMVAL("ulFlag", 0)
	msg@.SETMEMVAL("ulTime", 0)
	msg@.SETMEMVAL("bData[0]", &h11)
	msg@.SETMEMVAL("bData[1]", &h22)
	' 1 件のメッセージを送信
	errno = CFUNCALL(ret%, "CanSendMessage", fd, msg@, 1)

CAN FD通信の場合＞

C言語の例	AJANの例

CAN_MESSAGE_FD msg;
msg.ulLength = 2; // 送信メッセージ長
msg.ulID = 0x13;
msg.ulFlag = 3;
msg.ulTime = 0;
msg.bData[0] = 0x11;
msg.bData[1] = 0x22;
// 1件のメッセージを送信
CanSendMessageFD(fd, &msg, 1);
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE_FD")
	msg@.SETMEMVAL("ulLength", 2) ' メッセージ長
	msg@.SETMEMVAL("ulID", &H13)
	msg@.SETMEMVAL("ulFlag", 3)
	msg@.SETMEMVAL("ulTime", 0)
	msg@.SETMEMVAL("bData[0]", &h11)
	msg@.SETMEMVAL("bData[1]", &h22)
	' 1件のメッセージを送信
	errno = CFUNCALL(ret%, "CanSendMessageFD", fd, msg@, 1)

CANメッセージを送信した後に、CanGetCompletionMessage で、送信完了メッセージを受け取る事で送信完了を確認できます。
CAN FDメッセージの送信の場合は、CanGetCompletionMessageFD を用います。

なお、送信完了メッセージですが、特定条件で送信完了メッセージが受け取れないので注意が必要です。

- C言語版で CanGetConfig または CanGetConfigFD で設定状態を取得可能な、通信モードがFIFO送信モードを選択している。
- CanCyclicSendMessage などの、周期送信を用いている。

CAN通信の場合＞

C言語の例	AJANの例

CAN_MESSAGE msg;
unsigned long cnt=1;
 	
// 送信完了メッセージの取り出し
CanGetCompletionMessage(fd, &msg, &cnt);
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE")
	cnt& = 1
	
	' 送信完了メッセージの取り出し
	errno = CFUNCALL(ret%, "CanGetCompletionMessage", fd, msg@, cnt&)

CAN FD通信の場合＞

C言語の例	AJANの例

CAN_MESSAGE_FD msg;
unsigned long cnt=1;
 	
// 送信完了メッセージの取り出し
CanGetCompletionMessageFD(fd, &msg, &cnt);
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE_FD")
	cnt& = 1
	
	' 送信完了メッセージの取り出し
	errno = CFUNCALL(ret%, "CanGetCompletionMessageFD", fd, msg@, cnt&)

CANメッセージの受信は自動的に行われ、ドライバ内部のバッファに蓄えられるので、CAN通信の場合は CanReceiveMessage、CAN FD通信の場合は CanReceiveMessageFD で取得します。
受信データがバッファに蓄えられたか確認するには、動作状態を取得する関数を用います。CAN通信の場合は CanGetStatus、CAN FD通信の場合は CanGetStatusFD で取得します。

CAN通信の場合＞

C言語の例	AJANの例

CAN_PORT_STATUS stat;
CAN_MESSAGE msg;
unsigned long cnt;
int i;
 	
// 受信データが入るまで待機
while(1){
ret = CanGetStatus(fd, &stat);
if(ret == IFCAN_ERROR_SUCCESS){
if(stat.ulRXBCount){
break; // 受信データあり
}
}
}
 	
// 受信メッセージの取り出し
cnt = stat.ulRXBCount;
ret = CanReceiveMessage(fd, &msg, &cnt);
 	
// 受信メッセージの表示
if(ret == IFCAN_ERROR_SUCCESS){
printf(" メ ッ セ ー ジ 長 :%ld¥n", msg.ulLength);
printf("ID: %lx¥n", msg.ulID);
for(i=0; i < msg.ulLength; i++){
printf("%02x ", msg.bData[i]);
}
printf("¥n");
}
	OBJECT stat@ AS MEMORY
	stat@ = CSTRUCT@("CAN_PORT_STATUS")
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE")
	
	' 受信データが入るまで待機
	DO WHILE TRUE
	errno = CFUNCALL(ret%, "CanGetStatus", fd, stat@)
	if ret% = 0 then
	if stat@.GETMEMVAL("ulRXBCount") then
	EXIT DO ' 受信データあり
	end if
	end if
	LOOP
	
	' 受信メッセージの取り出し
	cnt& = stat@.GETMEMVAL("ulRXBCount")
	errno = CFUNCALL(ret%, "CanReceiveMessage", fd, msg@, cnt&)
	
	' 受信メッセージの表示
	if ret = 0/*IFCAN_ERROR_SUCCESS*/ then
	sz% = msg@.GETMEMVAL("ulLength")
	PRINT "メッセージ長:"; sz%
	PRINT "ID: ";
	hex$(msg@.GETMEMVAL("ulID"))
	FOR I=0 TO sz%-1
	b% =
	msg@.GETMEMVAL("bData["+str$(i)+"]")
	PRINT hex$(b%, 2); " ";
	NEXT I
	PRINT ""
	end if

CAN FD通信の場合＞

C言語の例	AJANの例

CAN_PORT_STATUS_FD stat;
CAN_MESSAGE_FD msg;
unsigned long cnt;
 	
// 受信データが入るまで待機
while(1){
ret = CanGetStatusFD(fd, &stat);
	OBJECT stat@ AS MEMORY
	stat@ = CSTRUCT@("CAN_PORT_STATUS_FD")
	OBJECT msg@ AS MEMORY
	msg@ = CSTRUCT@("CAN_MESSAGE_FD")
	
	' 受信データが入るまで待機
	DO WHILE TRUE

```
    if(ret == IFCAN_ERROR_SUCCESS){
        if(stat.ulRXBCount){
            break;  // 受信データあり
        }
    }
}

// 受信メッセージの取り出し
cnt = stat.ulRXBCount;
CanReceiveMessageFD(fd, &msg, &cnt);

// 受信メッセージの表示
if(ret == IFCAN_ERROR_SUCCESS){
    printf(" メ ッ セ ー ジ 長 :%ld¥n",
msg.ulLength);
    printf("ID: %lx¥n", msg.ulID);
    for(i=0; i < msg.ulLength; i++){
        printf("%02x ", msg.bData[i]);
    }
    printf("¥n");
}
```

```
    errno = CFUNCALL(ret%, "CanGetStatusFD",
fd, stat@)
    if ret% = 0 then
        if stat@.GETMEMVAL("ulRXBCount") then
            EXIT DO  ' 受信データあり
        end if
    end if
LOOP

' 受信メッセージの取り出し
cnt& = stat@.GETMEMVAL("ulRXBCount")
errno = CFUNCALL(ret%, "CanReceiveMessageFD",
fd, msg@, cnt&)

' 受信メッセージの表示
if ret = 0/*IFCAN_ERROR_SUCCESS*/ then
    sz% = msg@.GETMEMVAL("ulLength")
    PRINT "メッセージ長:"; sz%
    PRINT               "ID:               ";
hex$(msg@.GETMEMVAL("ulID"))
    FOR I=0 TO sz%-1
        b%                                  =
msg@.GETMEMVAL("bData["+str$(i)+"]")
        PRINT hex$(b%, 2); " ";
    NEXT I
    PRINT ""
end if
```

通信処理を終えてプログラムを終了する前、CanDeactivate で CANバスへの接続を切り離してから、CanClosePort でクローズします。

CAN通信の場合＞

C言語の例	AJANの例

// CAN バスの接続を切り離し
CanDeactivate(fd);
 	
// デバイスのクローズ
CanClosePort(fd);
	' CAN バスの接続を切り離し
	errno = CFUNCALL(ret%, "CanDeactivate", fd)
	
	' デバイスのクローズ
	errno = CFUNCALL(ret%, "CanClosePort", fd)

4.9.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-4851は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-4851のreadmeでは『ドライバモジュールの組み込み方法』項が該当します。

参考に、PEX-H485220PをPCIExpressスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②ドライバモジュールの組み込み
# modprobe cp4851
③ドライバモジュール(cp4851)が組み込まれた事を確認
# lsmod | grep cp
cp4851                 118784  0
ifcpmgr                 81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、/proc ファイルシステムを参照する事で、ボードとデバイス名を確認します。

```
$ cat /proc/driver/can/cp4851
cp4851 info:1.0
ifcan1: PCI/LPC/PAZ/PEX-(H)485220(P) (bid=5h) CH1 (High-speed)  [125000bps] tx:1 rx:0
ifcan2: PCI/LPC/PAZ/PEX-(H)485220(P) (bid=5h) CH2 (High-speed)  [125000bps] tx:0 rx:4
```

上の例では、PEX-H485220 の CH1 が ifcan1、CH2 が ifcan2 というデバイス名である事が判ります。

4.10 GPG-6204 3モードパルスカウンタドライバの制御事例

4.10.1 基本手順

GPG-6204は、弊社3モードパルスカウンタ機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg6204.so
ヘッダファイル	fbipenc.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-6204のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbipenc.h")      ' GPG-6204 要インストール
CDECLARE "libgpg6204.so", s$
```

GPG-6204のヘルプファイルでは、『制御手順』の章で、初期化、カウンタ値の取得などの事例などが紹介されています。
この中で、よく使われると思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	PencOpen	デバイスのオープン
モード設定	PencSetMode	動作モード、カウンタ方向などを設定
カウンタ値取得	PencGetCounter	カウンタ値の取得
終了処理	PencClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの PencOpen と デバイスクローズの PencClose です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// デバイスオープン
nRet = PencOpen(nDevice, PENC_FLAG_NORMAL);
	' デバイスオープン
	nDevice = 1
	errno = CFUNCALL(nRet, "PencOpen", nDevice, 0/* PENC_FLAG_NORMAL */)

int nRet;
int nDevice = 1;
nRet = PencOpen(nDevice, 0);
if(!nRet){
  ... <略>
  // デバイスクローズ
  nRet = PencClose(nDevice);
}
	nDevice = 1
	errno = CFUNCALL(nRet, "PencOpen", nDevice, 0)
	IF nRet = 0 THEN
	  ... <略>
	  ' デバイスクローズ
	  errno = CFUNCALL(nRet, "PencClose", nDevice)
	END IF

次に動作モードを指定する PencSetMode と、カウンタ値を取得する PencGetCounter です。

C言語の例	AJANの例

int nRet;
int nCh = 1;
// 動作モードの設定
nRet = PencSetMode(nDevice, nCh, 0x04, 0, 1, 0);
	nCh = 1
	’ 動作モードの設定
	errno = CFUNCALL(nRet, ″PencSetMode″, nDevice, nCh, &h04, 0, 1, 0)

int nRet;
int nCh = 1;
unsigned long ulCnt;
// カウンタ値の取得
nRet = PencGetCounter(nDevice, nCh, &ulCnt);
	nCh = 1
	’ カウンタ値の取得
	errno = CFUNCALL(nRet, ″PencGetCounter″, nDevice, nCh, ulCnt&)

指定する引数は整数および整数のポインタが多いので、AJANは引数を連ねるだけでよいです。

4.10.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-6204は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-6204のオンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PCI-6204をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg6204/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash inspenc.sh
④ドライバモジュール(cp6204)が組み込まれた事を確認
# lsmod | grep cp
cp6204                241664  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

*************************************************
 Setup Utility
-------------------------------------------------
 Version: 1.60-10
-------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
*************************************************

Enter the model number of the product: GPG/GPH-6204  <= ②「6204」を入力
=============================================================
 Ref.ID | Model                          | RSW1 | ENCODER | Device No.
-------------------------------------------------------------
      1 | PCI/PAZ/PEX-(H)6204            |    0 |         |     1
-------------------------------------------------------------
      2 | PCI/PAZ-(H)360116              |    0 |  penc   |     2
=============================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
```

```
 99. Exit the program.
**************************************
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、PencOpen関数の呼び出しで指定するデバイス番号です。
上の例では、PCI-6204のデバイス番号は 1 です。

4.11　GPG-6320 万能カウンタドライバの制御事例

4.11.1 基本手順

GPG-6320は、弊社万能カウンタ機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

	万能カウンタ	高速カウンタ
共有ライブラリ	libgpg6320u.so	libgpg6320hs
ヘッダファイル	ifucnt.h	ifhscnt.h

以下のように呼び出す事で、呼び出しの基本準備が整います。

万能カウンタの場合＞

```
' GPG-6320のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/ifucnt.h")        ' GPG-6320 要インストール
CDECLARE "libgpg6320u.so", s$
```

高速カウンタの場合＞

```
' GPG-6320のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/ifhscnt.h")       ' GPG-6320 要インストール
CDECLARE "libgpg6320hs.so", s$
```

GPG-6320のヘルプファイルでは、『実行手順』の章で、初期化、カウンタ値の取得などの事例などが紹介されています。
この中で、よく使われると思われる関数を、以下に列挙します。

万能カウンタの場合＞

機能	関数名	説明
初期化	UcntOpen	デバイスのオープン
モード設定	UcntSetPulseCountMode	パルスカウントモードの設定
機能設定	UcntSetLoadData	プリロードデータの設定
カウンタ値取得	UcntReadCounter	カウンタ値の取得
終了処理	UcntClose	デバイスのクローズ

高速カウンタの場合＞

機能	関数名	説明
初期化	HScntOpen	デバイスのオープン
モード設定	HScntSetMode	動作モード、カウンタ方向などを設定
カウンタ開始・停止	HScntStartCount HScntStopCount	カウンタを開始 カウンタを停止
カウンタ値取得	HScntReadCounter	カウンタ値の取得
終了処理	HScntClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、万能カウンタのデバイスオープンの UcntOpen と デバイスクローズの UcntClose と、高速カウンタのデバイスオープンの HScntOpen と デバイスクローズの HScntClose です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

＜万能カウンタの場合＞

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// 万能カウンタのデバイスオープン
nRet = UcntOpen(nDevice);
	’ 万能カウンタのデバイスオープン
	nDevice = 1
	errno = CFUNCALL(nRet%, "UcntOpen", nDevice)

int nRet;
int nDevice = 1;
nRet = UcntOpen(nDevice);
if(!nRet){
  ... <略>
  // 万能カウンタのデバイスクローズ
  nRet = UcntClose(nDevice);
}
	nDevice = 1
	errno = CFUNCALL(nRet%, "UcntOpen", nDevice)
	IF nRet = 0 THEN
	  ... <略>
	  ’ 万能カウンタのデバイスクローズ
	  errno=CFUNCALL(nRet%, "UcntClose", nDevice)
	END IF

＜高速カウンタの場合＞

C言語の例	AJANの例

int nRet;
int nDevice = 1
// 高速カウンタのデバイスオープン
nRet = HScntOpen(nDevice);
	’ 高速カウンタのデバイスオープン
	nDevice = 1
	errno = CFUNCALL(nRet%, "HScntOpen", nDevice)

int nRet;
int nDevice = 1;
nRet = HScntOpen(nDevice);
if(!nRet){
  ... <略>
  // 高速カウンタのデバイスクローズ
  nRet = HScntClose(nDevice);
}
	nDevice = 1
	errno = CFUNCALL(nRet, "HScntOpen", nDevice)
	IF nRet = 0 THEN
	  ... <略>
	  ’ 高速カウンタのデバイスクローズ
	  errno=CFUNCALL(nRet%,"HScntClose",nDevice)
	END IF

次に、万能カウンタで使用するパルスカウントモード設定を行う UcntSetPulseCountMode と、プリロードデータ設定を行う UcntSetLoadData 、カウンタ値を取得する UcntReadCounter です。

C言語の例	AJANの例

int nRet;
// パルスカウントモードに設定
nRet=UcntSetPulseCountMode(nDevice, 1, IFUCNT_COUNT_PHASE_4, 0x01, 0x00);
	’ パルスカウントモードに設定
	errno=CFUNCALL(nRet%,"UcntSetPulseCountMode", nDevice, 1, &HA/* IFUCNT_COUNT_PHASE_4 */, &H01, &H01)

int nRet;
 	
// プリロードデータを設定
nRet = UcntSetLoadData(nDevice, 1, 0);
	’ プリロードデータを設定
	errno = CFUNCALL(nRet%, "UcntSetLoadData", nDevice, 1, 0)

int nRet; unsigned long dwCounter[4];	DIM dwCounter&(3)

// カウンタ値の取得
nRet=UcntReadCounter(nDevice,0x0f,
dwCounter);
	' カウンタ値の取得
	errno = CFUNCALL(nRet%, "UcntReadCounter",
	nDevice, &H0f, dwCounter&)

次に、高速カウンタで使用するカウンタの外部制御モードを設定する HScntSetMode と、カウントを開始する HScntStartCount、カウントを停止する HScntStopCount、カウンタ値を取得する HScntReadCounter です。

C言語の例	AJANの例

int nRet;
 	
// カウンタの外部制御モードを設定
nRet = HScntSetMode(nDevice, 1, 0x06);
	' カウンタの外部制御モードを設定
	errno = CFUNCALL(nRet%, "HScntSetMode",
	nDevice, 1, &H06)

int nRet;
 	
// カウントを開始する
nRet = HScntStartCount(nDevice, 0x03,
IFHSCNT_CMD_START);
	' カウントを開始する
	errno = CFUNCALL(nRet%, "HScntStartCount",
	nDevice, &H03, &H01/* IFHSCNT_CMD_START */)

int nRet;
 	
// カウントを停止する
nRet = HScntStopCount(nDevice, 0x01,
IFHSCNT_CMD_STOP);
	' カウントを停止する
	errno = CFUNCALL(nRet%, "HScntStopCount",
	nDevice, &H01, &H04/* IFHSCNT_CMD_STOP */)

int nRet;
unsigned long dwCounter[2];
 	
// カウンタ値を取得する
nRet = HScntReadCounter(nDevice, 0x03,
dwCounter);
	DIM dwCounter&(1)
	
	' カウンタ値を取得する
	errno = CFUNCALL(nRet%, "HScntReadCounter",
	nDevice, &H03, dwCounter&)

4.11.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-6320は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-6320のオンラインヘルプでは『導入方法』の章の『デバイスを動かすまで』項が該当します。

参考に、PCI-632206をPCIスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg6320/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insmcnt.sh
④ドライバモジュール(cp6320, cp6320u, cp6320hs)が組み込まれた事を確認
# lsmod | grep cp
cp6320                 16384  0
cp6320u                53248  1 cp6320
cp6320hs               32768  1 cp6320
acpi_pad               24576  0
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

*************************************************
 Setup Utility
-------------------------------------------------
 Version: 1.60-10
-------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
*************************************************

Enter the model number of the product: GPG/GPH-6320u   <= ②「6320u」を入力
=================================================================
 Ref.ID | Model                          | RSW1 |  TYPE | Device No.
-----------------------------------------------------------------
      1 | PCI/PAZ-632206(Tx)             |    0 |  ucnt |     1
=================================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
```

```
 99. Exit the program.
**************************************
Enter the command number: 99

③再度、デバイス番号設定ユーティリティを起動
# bash setup.sh

************************************************
 Setup Utility
------------------------------------------------
 Version: 1.60-10
------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
************************************************

Enter the model number of the product: GPG/GPH-6320hs   <= ④「6320hs」を入力
================================================================
 Ref.ID | Model                          | RSW1 |  TYPE | Device No.
----------------------------------------------------------------
      1 | PCI/PAZ-632206(Tx)             |    0 | hscnt |     1
================================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
**************************************
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、万能カウンタ(6320u)の場合 UcntOpen関数の呼び出しで、高速カウンタ(6320hs)の場合 HScntOpen関数の呼び出しで、指定するデバイス番号です。
上の例では、PCI-632206のデバイス番号は 1 です。

4.12 GPG-7400 パルスモーションドライバの制御事例

4.12.1 基本手順

GPG-7400は、弊社パルスモーションコントローラ機能を制御するドライバおよびライブラリです。
共有ライブラリとヘッダファイルは、以下のファイル名です。

共有ライブラリ	libgpg7400.so
ヘッダファイル	fbimtn.h

これにより、以下のように呼び出す事で、呼び出しの基本準備が整います。

```
' GPG-7400のヘッダファイルを読み取ります
s$ = str_freadall$("/usr/include/fbimtn.h")        ' GPG-7400 要インストール
CDECLARE "libgpg7400.so", s$
```

GPG-7400ヘルプファイルでは、『制御手順』の章で、初期化、動作パラメータ設定、動作起動などの事例などが紹介されています。
この中で、よく使われると思われる関数を、以下に抜粋します。

機能	関数名	説明
初期化	MtnOpen	デバイスのオープン
初期設定	MtnSetPulseOut	パルス出力の設定
	MtnSetLimitConfig	制御信号の各種設定
動作パラメータ設定	MtnSetMotion	独立動作パラメータの設定
動作起動	MtnStartMotion	各種動作の起動
動作停止	MtnStopMotion	動作を停止
モニタリング	MtnGetStatus	各種ステータスの取得
カウンタ読み込み	MtnReadCounter	カウンタ値の読み込み
終了処理	MtnClose	デバイスのクローズ

幾つかの関数で、C言語の使用例を元に、AJANでの記述事例を示します。

まず、デバイスオープンの MtnOpen と デバイスクローズの MtnClose です。
関数の引数と戻り値は、int 型と単純な為、以下のように簡単に呼び出し可能です。

C言語の例	AJANの例

int nRet;
int nDevice = 1;
// デバイスオープン
nRet = MtnOpen(nDevice, MTR_FLAG_NORMAL);
	' デバイスオープン
	nDevice = 1
	errno = CFUNCALL(nRet, "MtrOpen", nDevice, 0/*
	MTR_FLAG_NORMAL */)

int nRet;
int nDevice = 1;
nRet = MtrOpen(nDevice, 0);
if(!nRet){
	
	errno = CFUNCALL(nRet, "MtrOpen", nDevice, 0)
	IF nRet = 0 THEN

... 〈略〉
// デバイスクローズ
nRet = MtrClose(nDevice);
}
	... 〈略〉
	' デバイスクローズ
	errno = CFUNCALL(nRet, "MtrClose", nDevice)
	END IF

次にパルス出力の設定を行う MtnSetPulseOut と、制御信号の設定を行う MtnSetLimitConfig です。

C言語の例	AJANの例

int nRet;
// パルス出力の設定
nRet = MtnSetPulseOut(nDevice, 0x0F, MTR_METHOD, 0x00);
	' パルス出力の設定
	errno = CFUNCALL(nRet%, "MtnSetPulseOut", nDevice, &H0F, 0/*MTR_METHOD*/, &H00)

int nRet;
// 制御信号の設定
nRet = MtnSetLimitConfig(nDevice, 0x02, MTR_LOGIC, 0x04);
	' 制御信号の設定
	errno = CFUNCALL(nRet%, "MtnSetLimitConfig", nDevice, &H02, 1/*MTR_LOGIC*/, &H04)

C言語の例で、定数名「MTR_METHOD」または「MTR_LOGIC」があります。
これは、ヘッダファイル(fbimtn.h)にて、以下のように定義されていますので、即値を指定します。
(0x01 はC言語で16進数の定数値なので、AJANでは &H01 と記述します)

```
... 〈略〉
/* Set(Get)PulseOut */
#define MTR_METHOD          0x00
#define MTR_IDLING          0x01
... 〈略〉
/* Set(Get)LimitConfig */
#define MTR_MASK            0x00
#define MTR_LOGIC           0x01
#define MTR_SD_FUNC         0x02
... 〈略〉
```

次は独立動作パラメータの設定を行う MtnSetMotion です。

C言語の例	AJANの例

int nRet;
MTNMOTION Motion[4];
 	
// 独立動作パラメータの設定
Motion[0].wClock = 299;
Motion[0].wAccMode = MTR_ACC_SIN;
Motion[0].fLowSpeed = 50;
Motion[0].fSpeed = 1200;
Motion[0].ulAcc = 500;
Motion[0].ulDec = 700;
Motion[0].fSAccSpeed = 200;
Motion[0].fSDecSpeed = 300;
Motion[0].lStep = 2500;
Motion[1].wClock = 299;
	OBJECT Motion@ AS MEMORY
	Motion@ = CSTRUCT@("MTNMOTION", 4)
	
	' 独立動作パラメータの設定
	Motion@.SETMEMVAL(0, "wClock", 299)
	Motion@.SETMEMVAL(0, "wAccMode", &h01/*MTR_ACC_SIN*/)
	Motion@.SETMEMVAL(0, "fLowSpeed", 50)
	Motion@.SETMEMVAL(0, "fSpeed", 1200)
	Motion@.SETMEMVAL(0, "ulAcc", 500)
	Motion@.SETMEMVAL(0, "ulDec", 700)
	Motion@.SETMEMVAL(0, "fSAccSpeed", 200)
	Motion@.SETMEMVAL(0, "fSDecSpeed", 300)
	Motion@.SETMEMVAL(0, "lStep", 2500)

Motion[1].wAccMode = MTR_ACC_NORMAL; Motion[1].fLowSpeed = 100; Motion[1].fSpeed = 1000; Motion[1].ulAcc = 900; Motion[1].ulDec = 1200; Motion[1].fSAccSpeed = 0; Motion[1].fSDecSpeed = 0; Motion[1].lStep = -100; nRet = MtnSetMotion(nDevice, 0x03, MTR_PTP, Motion);	Motion@.SETMEMVAL(1, "wClock", 299) Motion@.SETMEMVAL(1, "wAccMode", &h00/*MTR_ACC_NORMAL*/) Motion@.SETMEMVAL(1, "fLowSpeed", 100) Motion@.SETMEMVAL(1, "fSpeed", 1000) Motion@.SETMEMVAL(1, "ulAcc", 900) Motion@.SETMEMVAL(1, "ulDec", 1200) Motion@.SETMEMVAL(1, "fSAccSpeed", 0) Motion@.SETMEMVAL(1, "fSDecSpeed", 0) Motion@.SETMEMVAL(1, "lStep", -100) errno = CFUNCALL(nRet%, "MtnSetMotion", nDevice, &H03, &h02/*MTR_PTP*/, Motion@)

C言語の例で、MTNMOTION構造体を使った「Motion」配列変数を、AJANでは OBJECT型の変数を宣言し、CSTRUCT@ 関数の引数に "MTNMOTION" を指定して構造体に必要なメモリ量と、要素数4 を指定して、全体で必要なメモリを確保しています。

CSTRUCT@ 関数で確保した OBJECT型の変数では、SETMEMVAL や GETMEMVAL などのメソッドを使って、C言語で構造体のメンバを操作するのと同じように、アクセス操作が可能です。

次に動作開始を行う MtnStartMotion と、動作停止を行う MtnStopMotion です。
これにより、モータを動作させたり、停止することができます。

C言語の例	AJANの例

int nRet;
// 動作開始
nRet = MtnStartMotion(nDevice, 0x0F, MTR_ACC, MTR_JOG);
	' 動作開始
	errno = CFUNCALL(nRet%, "MtnStartMotion", nDevice, &H0F, &h00/*MTR_ACC*/, &h00/*MTR_JOG*/)

int nRet;
// 動作停止
nRet = MtnStopMotion(nDevice, 0x01, MTR_IMMEDIATE_STOP);
	' 動作停止
	errno = CFUNCALL(nRet%, "MtnStopMotion", nDevice, &H01, 1/*MTR_IMMEDIATE_STOP*/)

次にステータスの取得を行う MtnGetStatus と、カウンタ値の読み込みを行う MtnReadCounter です。

C言語の例	AJANの例

int nRet;
unsigned long ulStatus[4];
// ステータス取得
nRet = MtnGetStatus(nDevice, 0x08, MTR_BUSY, ulStatus);
printf("U 軸動作状態:%lx¥n", ulStatus[3]);
	DIM ulStatus&(3)
	' ステータス取得
	errno = CFUNCALL(nRet%, "MtnGetStatus", nDevice, &H08, &h00/*MTR_BUSY*/, ulStatus&)
	PRINT "U 軸動作状態:"; HEX$(ulStatus&(3))

int nRet;
long lPos[4];
// カウンタ値読み込み
nRet=MtnReadCounter(nDevice,0x02,MTR_COUNTER, lPos);
	DIM lPos&(3)
	' カウンタ値読み込み
	errno = CFUNCALL(nRet%, "MtnReadCounter", nDevice, &H02, 1/*MTR_COUNTER*/, lPos&)

printf("Y 軸の出力パルスカウンタ値:%ld¥n", lPos[1]);	PRINT "Y 軸の出力パルスカウンタ値:"; lPos&(1)

C言語の例では、ステータスの取得やカウンタ値の取得に、long型の配列を指定しています。
long型は、AJANでは 倍精度整数 に相当するので、倍精度整数の配列を確保し、呼び出し時に指定しています。
他に、OBJECT型の変数を宣言し、CALLOC@ 関数で必要なメモリ量を確保して渡す事も可能です。

4.12.2 ポイント＞初回にドライバモジュール組み込みを忘れない

readmeおよびオンラインヘルプで注意喚起されているように、GPG-7400は初回時、ドライバモジュールの組み込みが必要です。

組み込み手順の説明は、GPG-7400のオンラインヘルプでは『実行手順』の章の『デバイスを動かすまで』項が該当します。

参考に、PEX-H741444VをPCI Expressスロットに挿して、ドライバモジュールを組み込む作業を例示します。

```
①スーパーユーザーになる
$ su
②フォルダ移動
# cd /usr/src/interface/gpg7400/x86_64/linux/drivers
③ドライバモジュールの組み込み
# bash insmtn.sh
④ドライバモジュール(cp7400)が組み込まれた事を確認
# lsmod | grep cp
cp7400                241664  0
ifcpmgr                81920  3
...
```

ドライバモジュールが組み込まれたことを確認した後、デバイス番号設定ユーティリティで、デバイスノードを作ります。

```
①デバイス番号設定ユーティリティを起動
# bash setup.sh

*************************************************
 Setup Utility
-------------------------------------------------
 Version: 1.60-10
-------------------------------------------------
 Copyright 2003, 2011      Interface Corporation.
                            All rights reserved.
*************************************************

Enter the model number of the product: GPG/GPH-7400  <= ②「7400」を入力
=================================================================
 Ref.ID | Model                          | RSW1 | Device No.
-----------------------------------------------------------------
      1 | PEX-H741444V                   |    0 |     1
=================================================================

************** Command ***************
  1. Change the device number.
  2. Delete the device number.
  3. Load new device setting file.
  4. Run the initialization program.
  5. Run the CardBus ID setup utility.
 99. Exit the program.
```

```
************************************
Enter the command number: 99
```

デバイス番号設定ユーティリティで、デバイス番号(Device No.)を確認できます。
この値は、MtnOpen関数の呼び出しで、指定するデバイス番号です。
上の例では、PEX-H741444Vのデバイス番号は 1 です。

第5章 リファレンス

使用できる拡張コマンドの使い方について記載します。

!	制限事項については、「注意」に記載しています。
!	使用例は動作を保証するものではありません。 実際の使い方は各種サンプルプログラムを参照してください。

5.1 コマンド一覧

コマンド名	機能
基本制御に関する関数・命令	
CDECLARE	C言語連携機能を用いて呼び出したい、共有ライブラリおよび関数の定義を行います。
CSUBCALL	C言語連携機能を用いて、共有ライブラリの戻り値の無い関数を呼び出します。
CFUNCALL	C言語連携機能を用いて、共有ライブラリの戻り値の有る関数を呼び出します。
CGETADRS@	C言語連携機能を用いて、指定した関数名のアドレス情報を得ます。
CIMPORT	指定した外部C言語ファイルのプログラムを、自身に埋め込んでコンパイルします。
型の宣言や変換に関する関数・命令	
OBJECT	オブジェクト変数を宣言します。
LOCAL OBJECT	オブジェクトローカル変数を宣言します。
OBJDELETE	オブジェクト変数をクリアします。
OBJTYPENAME$	オブジェクト変数のオブジェクト型名を取得します。
POINTER型に関する関数・命令	
<POINTER>.FROMLNG	POINTER型変数に対して、アドレス値をセットします。
<POINTER>.TOLNG	POINTER型変数から、アドレス値が得られます。
MEMORY型に関する関数・命令	
CALLOC@	指定したバイトサイズのメモリブロックを確保します。
CMEMMAP@	指定したアドレスからバイトサイズの領域をアクセス可能領域とします。
CSTRUCT@	指定したC言語構造体名に該当するサイズのメモリブロックを確保します。
CMEMMOVE	MEMORY型オブジェクトのメモリの特定領域をコピーします。
<MEMORY>.PEEKBYTE	MEMORY型オブジェクトの指定したメモリ位置からバイトデータ値を読み取ります。
<MEMORY>.PEEKHALF	MEMORY型オブジェクトの指定したメモリ位置からワードデータ値を読み取ります。
<MEMORY>.PEEKINT	MEMORY型オブジェクトの指定したメモリ位置からダブルワードデータ値を読み取ります。
<MEMORY>.PEEKLNG	MEMORY型オブジェクトの指定したメモリ位置からロングダブルワードデータ値を読み取ります。

コマンド名	機能
<MEMORY>.PEEKSNG	MEMORY型オブジェクトの指定したメモリ位置から単精度実数値を読み取ります。
<MEMORY>.PEEKDBL	MEMORY型オブジェクトの指定したメモリ位置から倍精度実数値を読み取ります。
<MEMORY>.PEEKSTR$	MEMORY型オブジェクトの指定したメモリ位置から指定サイズ長のバイナリ文字列を読み取ります。
<MEMORY>.POKEBYTE	MEMORY型オブジェクトの指定したメモリ位置に、バイトデータ値を書き込みます。
<MEMORY>.POKEHALF	MEMORY型オブジェクトの指定したメモリ位置に、ワードデータ値を書き込みます。
<MEMORY>.POKEINT	MEMORY型オブジェクトの指定したメモリ位置に、ダブルワードデータ値を書き込みます。
<MEMORY>.POKELNG	MEMORY型オブジェクトの指定したメモリ位置に、ロングダブルワードデータ値を書き込みます。
<MEMORY>.POKESNG	MEMORY型オブジェクトの指定したメモリ位置に、単精度実数値を書き込みます。
<MEMORY>.POKEDBL	MEMORY型オブジェクトの指定したメモリ位置に、倍精度実数値を書き込みます。
<MEMORY>.POKESTR	MEMORY型オブジェクトの指定したメモリ位置に、バイナリ文字列値を書き込みます。
<MEMORY>.GETMEMOFF	CSTRUCT@関数の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当するオフセット位置を得ます。
<MEMORY>.GETMEMVAL	CSTRUCT@関数の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当するメンバ位置の、メモリ値を取得します。
<MEMORY>.GETMEMSTR$	CSTRUCT@関数の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当するメンバ位置の文字列を取得します。
<MEMORY>.SETMEMVAL	CSTRUCT@関数の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当するメンバ位置の、メモリに対して値を書き込みます。
<MEMORY>.SIZEOF	MEMORY型オブジェクト変数のメモリブロックサイズが得られます。
<MEMORY>.TOJSON$	CSTRUCT@関数の引数で与えたC言語の構造体型に対して、構造体の各メンバ名とメモリから取得した値を、JSON形式の文字列で得られます。
<MEMORY>.TOLNG	MEMORY型オブジェクトのメモリブロックのアドレス値が得られます。
<MEMORY>.TOSTRTYPE$	MEMORY型オブジェクトの指定したメモリ位置から指定した型名の数値または文字列を文字列化します。
JSON型に関する関数・命令	
JSON_PARSE@	JSON形式の文字列を解析し、JSON型オブジェクトを得ます。
JSON_TRYPARSE	JSON形式の文字列の解析に挑戦し、その可否を返します。解析に成功すれば、JSON型オブジェクトをセットします。
JSON_NEW@	指定した数値または文字列からJSON型オブジェクトを得ます。
JSON_DUMP$	JSON型オブジェクトからJSON形式の文字列が得られます。
<JSON>.TYPEOF	JSON型オブジェクトの種類を調べます。
<JSON>.GET_CSTR$	JSON型オブジェクトから文字列を得ます。
<JSON>.GET_CLNG	JSON型オブジェクトから整数値を得ます。
<JSON>.GET_CDBL	JSON型オブジェクトから実数値を得ます。
<JSON>.GET_CBOOL	JSON型オブジェクトから真偽値を得ます。
<JSON>.ARRAY_SIZE	JSON型オブジェクトから配列の要素数を得ます。
<JSON>.ARRAY_GET@	JSON型オブジェクトから配列の指定した添字のオブジェクトを得ます。
<JSON>.ARRAY_SET	JSON型オブジェクトの配列の指定した添字に値を設定します。

コマンド名	機能
<JSON>.ARRAY_INSERT	JSON型オブジェクトの配列の指定した添字位置に値を挿入／追加します。
<JSON>.ARRAY_REMOVE	JSON型オブジェクトの配列の指定した添字位置の値を削除します。
<JSON>.ARRAY_CLEAR	JSON型オブジェクトの配列の全ての要素を削除します。
<JSON>.OBJECT_SIZE	JSON型オブジェクトから、オブジェクト(連想配列)の要素数を得ます。
<JSON>.OBJECT_KEYS$	JSON型オブジェクトから、オブジェクト(連想配列)のキー配列を得ます。
<JSON>.OBJECT_GET@	JSON型オブジェクトから、オブジェクト(連想配列)の指定したキーのオブジェクトを得ます。
<JSON>.OBJECT_SET	JSON型オブジェクトの、オブジェクト(連想配列)の指定したキーに対して、値を設定します。
<JSON>.OBJECT_DEL_KEY	JSON型オブジェクトの、オブジェクト(連想配列)の指定したキーに紐付けされたデータを削除します。
<JSON>.OBJECT_CLEAR	JSON型オブジェクトの、オブジェクト(連想配列)の全ての要素を削除します。
<JSON>.ISEQ	JSON型オブジェクト同士が、同じ内容(TRUE)か否(FALSE)か比較します。
<JSON>.CLONE@	JSON型オブジェクトの複製を作ります。
<JSON>.TOLNG	JSON型の変数から、内部のアドレス値が得られます。

5.2 基本制御に関する関数・命令

ここでは、C言語で作られた共有ライブラリと連携するためのコマンドおよび関数群を紹介します。

5.2.1 CDECLARE

命令

機　能

C言語連携機能を用いて呼び出したい、共有ライブラリおよび関数、構造体の定義を行います。

書　式

CDECLARE 〈①ライブラリ名〉, 〈②呼び出し関数と構造体定義〉

パラメータ

①	<ライブラリ名>	文字列

C言語連携機能で、呼び出したい共有ライブラリ名を指定します。
空文字列を指定するとCIMPORT命令で組み込んだ自身を指定します。

②	<呼び出し関数と構造体定義>	文字列

C言語連携機能で、呼び出したい関数と書式を、C言語の関数プロトタイプ宣言形式で定義します。
関数を複数定義したい場合、改行して続けて記述できます。
また、C言語の構造体の定義を読み込ませて、CSTRUCT@関数で、メモリブロックを確保して、他で利用できます。

関数を定義する際に用いる事ができるC言語の型キーワードを、以下に列挙します。

型キーワード	解説
void	返値に void を指定すると、値を返さない意味になります。
char*	引数に char* を指定すると、文字列を渡す意味になります。 AJANでは文字列で代替します。
void*	引数に void* を指定すると、ポインタを渡す意味になります。 AJANでは、オブジェクト型を渡します。
int8_t	1バイト長の整数値を意味します。 AJANでは単精度整数で代替します。
int16_t	2バイト長の整数値を意味します。 AJANでは単精度整数で代替します。
int32_t	4バイト長の整数値を意味します。 AJANでは単精度整数で代替します。
int64_t	8バイト長の整数値を意味します。 AJANでは倍精度整数で代替します。
float	4バイト長の浮動小数点数を意味します。 AJANでは単精度実数で代替します。
double	8バイト長の浮動小数点数を意味します。 AJANでは倍精度実数で代替します。
const XXX	XXXの部分は、char* などの型キーワードを指定し、型に対する定数性を指示します。
struct 構造体名*	構造体名を指定したポインタ定義を指定すると、ポインタを渡す意味になります。 AJANでは、オブジェクト型を渡します。

備　考

・呼び出し関数の定義(C言語でいうプロトタイプ宣言)で定義した関数名が、指定したライブラリに見つからない場合、「not found 〈関数名〉」の形式で警告表示されます。
・C言語の共用体の定義は認識できません。また、構造体の中に共用体を含ませた場合も認識できません。
・C言語の可変個引数記号(「...」の形式)の定義は指定できません。

使用例1

```
CDECLARE "libc.so.6", "int strcmp(const char* s1, const char* s2);"
```

libc共有ライブラリに含まれている、strcmp関数を呼び出せるように定義します。

使用例2

```
' C言語のHOGE構造体 を定義して、これをAJANに読み込ませます。
S$ = '''
```

```
struct HOGE {
    int    id;
    char  msg[20];
};
'''
CDECLARE "", S$

OBJECT PTR@ AS MEMORY
' HOGE構造体が要求するサイズのメモリブロックを確保します
PTR@ = CSTRUCT@("HOGE")
' メモリブロックのポインタのアドレス値を表示します。
PRINT HEX$(PTR@.TOLNG())
```

5.2.2 CSUBCALL

関数			
機　能	C言語連携機能を用いて、共有ライブラリの戻り値の無い関数を呼び出します。		
書　式	〈(戻り値)errno値〉 = CSUBCALL(〈①関数名〉 [,〈②引数値〉, ...])		
戻り値	戻り値	**<errno値>**	数値
	共有ライブラリの関数を呼び出した結果のerrno値を得られます。		
パラメータ	①	**<関数名>**	文字列
	呼び出したい関数名を指定します。 CDECLAREの〈呼び出し関数定義〉で定義した関数名を指定します。		
	②	**<引数値>**	値
	呼び出したい関数に渡す引数を列挙します。 CDECLAREの〈呼び出し関数定義〉で定義した関数プロトタイプ宣言の引数通りに、値を渡します。		
備　考	・		
使用例	libc共有ライブラリに含まれている、perror関数を呼び出す事例です。 CDECLARE "libc.so.6", "void perror(const char* s);" PRINT CSUBCALL("perror", "hello")		

5.2.3 CFUNCALL

関数			
機　能	C言語連携機能を用いて、共有ライブラリの戻り値が有る関数を呼び出します。		
書　式	〈(戻り値)errno値〉 = CFUNCALL(〈①戻り値受け取り〉, 〈②関数名〉 [, 〈③引数値〉, ...])		
戻り値	戻り値	**<errno値>**	数値
	共有ライブラリの関数を呼び出した結果のerrno値を得られます。		
パラメータ	①	**<戻り値受け取り>**	値
	呼び出した関数の戻り値を受け取る変数を指定します。		
	②	**<関数名>**	文字列
	呼び出したい関数名を指定します。 CDECLAREの〈呼び出し関数定義〉で定義した関数名を指定します。		
	③	**<引数値>**	値
	呼び出したい関数に渡す引数を列挙します。 CDECLAREの〈呼び出し関数定義〉で定義した関数プロトタイプ宣言の引数通りに、値を渡します。		
備　考	・		
使用例	libc共有ライブラリに含まれている、strcmp関数を呼び出す事例です。 CDECLARE "libc.so.6", "int strcmp(const char* s1, const char* s2);" RET = 0 ERRNO = CFUNCALL(RET, "strcmp", "hello", "Hello") PRINT "strcmpの戻り値="; RET		

5.2.4 CGETADRS@

関数			
機　能	C言語連携機能を用いて、指定した関数名のアドレス情報を得ます。		
書　式	〈(戻り値)POINTER型オブジェクト〉 = CGETADRS@(〈①関数名〉)		
戻り値	戻り値	**<POINTER型オブジェクト>**	オブジェクト型
	関数名のアドレス値を、POINTER型オブジェクトにセットして得られます。		
パラメータ	①	**<関数名>**	文字列
	アドレス情報を得たい関数名を指定します。 CDECLAREの〈呼び出し関数定義〉で定義した関数名を指定します。		
備　考	・		
使用例	OBJECT PT@ AS POINTER CDECLARE "libc.so.6", "int strlen(const char* s);" PT@ = CGETADRS@("strlen") PRINT PT@ libc共有ライブラリに含まれている、strlen関数のアドレス値を得ます。		

5.2.5 CIMPORT

命令			
機　能	指定した外部C言語ファイルのプログラムを、自身に埋め込んでコンパイルします。		
書　式	CIMPORT ″<①C言語ファイル名>″ [, ″<②追加オプション>″]		
パラメータ	①	**<C言語ファイル名>**	文字列
	自身に埋め込んでコンパイルする為の、C言語ファイル名を指定します。 文字列定数で指定します。変数は使用できません。		
	②	**<追加オプション>**	文字列
	コンパイル時に、コンパイラに対するオプション指示を追加できます。		
注　意	・指定するC言語ファイル名の先頭部分に「AJAN-」を付けないでください。エラーとなります。 ・C言語ファイル名の拡張子は、「.c」を指定してください。 ・指定するC言語のプログラム中のシンボル名は、「c_」または「test」で始まるように指定してください。それ以外はエラーとなります。 ・C言語のプログラムがコンパイル不可と検知すると、エラーとなります。		
備　考	・処理内容的には、AJANをコンパイルする際に、指定したC言語ファイルを一緒に埋め込んでコンパイルします。 この為、ここで指定するC言語ファイルの関数名(＝シンボル)が、AJANの内部関数名と衝突しないように、注意が必要です。 ・埋め込んだ C言語ファイルの関数を呼び出すには、CDECLARE命令のライブラリ名に、空文字列を指定します。 ・C言語ファイルを相対パスで指定した時、INCLUDE命令と同じ基準で、ファイルを検索します。		
備　考	・2021/2より実験的に、C++言語ファイル(拡張子は「.cpp」)を組み込めるようになりました。シンボル名を指定する関数は「extern ″C″」を修飾し、C言語の関数名としてアクセスできるように指定してください。		
使用例	CIMPORT ″test.c″　　' test.c内に、int c_test(int a, int b); が定義されていると仮定 CDECLARE ″″, ″int c_test(int a, int b);″ RET = 0 ERRNO = CFUNCALL(RET, ″c_test″, 123, 456) PRINT ″c_test=″; RET		

5.3 型の宣言や変換に関する関数・命令

5.3.1 OBJECT

命令			
機　能	オブジェクト変数を宣言します。		
書　式	OBJECT 〈①変数名〉 AS 〈②オブジェクト型名〉		
パラメータ	①	**〈変数名〉**	変数名
	オブジェクト変数名を指定します。 オブジェクト型は、変数名の最後に「@」を付けることにより区別します。		
	②	**〈オブジェクト型名〉**	オブジェクト型名
	オブジェクト型名を指定します。 指定可能なオブジェクト型名は、「5.3.1.1 ＜オブジェクト型名一覧＞」を参照してください。		
備　考	・		
使用例	MEMORY型オブジェクト変数を宣言します。 OBJECT HOGE@ AS MEMORY		

5.3.1.1 ＜オブジェクト型名一覧＞

オブジェクト型名	解説	ページ
POINTER	メモリのアドレス値を管理します。	P.122
MEMORY	ヒープメモリを確保し、メモリ操作を行えます。	P.123
JSON	JSON型のオブジェクトを管理します。	P.142

5.3.2 LOCAL OBJECT

命令			
機　能	オブジェクトローカル変数を宣言します。		
書　式	LOCAL OBJECT 〈①変数名〉 AS 〈②オブジェクト型名〉		
パラメータ	①	**〈変数名〉**	変数名
	オブジェクトローカル変数名を指定します。 オブジェクト型は、変数名の最後に「@」を付けることにより区別します。		
	②	**〈オブジェクト型名〉**	オブジェクト型名
	オブジェクト型名を指定します。 指定可能なオブジェクト型名は、「5.3.1.1 ＜オブジェクト型名一覧＞」を参照してください。		
備　考	・		
使用例	MEMORY型オブジェクトローカル変数を宣言します。 SUB TEST() 　LOCAL OBJECT HOGE@ AS MEMORY 　...		

5.3.3 OBJDELETE

命令			
機　能	オブジェクト変数をクリアします。		
書　式	OBJDELETE〈①オブジェクト変数〉		
パラメータ	①	**〈オブジェクト変数〉**	変数名
	クリアしたいオブジェクト変数を指定します。		
備　考	OBJECT命令で宣言した直後の状態に戻します。		
使用例	MEMORY型オブジェクト変数 ME@ をクリアします。 OBJECT ME@ AS MEMORY ME@ = CALLOC@(100) OBJDELETE ME@		

5.3.4 OBJTYPENAME$

関数			
機　能	オブジェクト変数のオブジェクト型名を取得します。		
書　式	〈(戻り値)オブジェクト型名〉 = OBJTYPENAME$(〈①オブジェクト変数〉)		
戻り値	戻り値	**<オブジェクト型名>**	文字列
	オブジェクト変数のオブジェクト型名が得られます。		
パラメータ	①	**〈オブジェクト変数〉**	オブジェクト型
	調べたいオブジェクト変数を記述します。		
備　考	・得られるのは、OBJECT命令などで指定したオブジェクト型名です。 ・存在しないオブジェクト変数を与えると、空文字列が得られます。		
使用例	OBJECT PTR@ AS POINTER PRINT VARTYPE(PTR@)　　’8192が出力されます PRINT OBJTYPENAME$(PTR@)　’POINTERが出力されます		

5.4 POINTER型に関する関数・命令

POINTER型は、メモリなどのアドレス値を保持するためのオブジェクト変数です。
C言語連携を使って、共有ライブラリの関数を呼び出したりする際、使用したい関数がアドレス値を扱う関数だったときに、アドレス値をAJANとの間で授受させるなどに用います。

5.4.1 <POINTER>.FROMLNG

命令			
機　能	POINTER型オブジェクト変数に対して、アドレス値をセットします。		
書　式	<①POINTER型変数> . FROMLNG(<②アドレス値>)		
パラメータ	①	**<POINTER型変数>**	オブジェクト型
	POINTER型オブジェクト変数を指定します。		
	②	**<アドレス値>**	数値
	POINTER型オブジェクト変数にセットするアドレス値を指定します。 倍精度整数で指定します。		
使用例	OBJECT PTR@ AS POINTER PTR@.FROMLNG(&H12345) PRINT "アドレス="; HEX$(PTR@.TOLNG())		

5.4.2 <POINTER>.TOLNG

関数			
機　能	POINTER型オブジェクト変数から、アドレス値が得られます。		
書　式	<(戻り値)アドレス値> = <①POINTER型変数> . TOLNG( )		
戻り値	戻り値	**<アドレス値>**	数値
	POINTER型オブジェクト変数にセットされているアドレス値が、倍精度整数値で得られます。		
パラメータ	①	**<POINTER型変数>**	オブジェクト型
	POINTER型オブジェクト変数を指定します。		
使用例	OBJECT PTR@ AS POINTER PTR@.FROMLNG(&H12345) PRINT "アドレス="; HEX$(PTR@.TOLNG())		

5.5 MEMORY型に関する関数・命令

MEMORY型は、C言語のmalloc 関数で扱うヒープメモリのメモリブロックを扱うオブジェクト変数です。C言語連携を使って共有ライブラリの関数を使用する際、その関数がメモリブロックを扱う関数の場合に、メモリブロックおよび内容をAJANとの間で授受させるなどに用います。

5.5.1 CALLOC@

関数			
機　能	指定したバイトサイズのメモリブロックを確保します。		
書　式	<(戻り値)MEMORY型オブジェクト> = CALLOC@(<①バイトサイズ>)		
戻り値	戻り値	**<MEMORY型オブジェクト>**	オブジェクト型
	MEMORY型オブジェクトが得られます。		
パラメータ	①	**<バイトサイズ>**	数値
	確保したいメモリブロックのサイズを、バイト単位で指定します。		
備　考	・メモリブロックの確保は、C言語の malloc 関数が使用されます。		
使用例	OBJECT PTR@ AS MEMORY ’100バイトのメモリブロックを確保します PTR@ = CALLOC@(100) ’メモリブロックのポインタのアドレス値を表示します。 PRINT HEX$(PTR@.TOLNG())		

5.5.2 CMEMMAP@

関数			
機　能	指定したアドレスからバイトサイズの領域をアクセス可能領域とします。		
書　式	〈(戻り値)MEMORY型オブジェクト〉 = CMEMMAP@(〈①アドレス値〉, 〈②バイトサイズ〉)		
戻り値	戻り値	**<MEMORY型オブジェクト>**	オブジェクト型
	MEMORY型オブジェクトが得られます。		
パラメータ	①	**〈アドレス値〉**	数値
	AJAN以外の関数呼び出し等で確保したメモリブロックの、アクセスしたい先頭アドレスを指定します。 アドレス値は、倍精度整数かPOINTER型で指定してください。		
	②	**〈バイトサイズ〉**	数値
	アクセスしたいメモリブロックのサイズを、バイト単位で指定します。		
備　考	・アクセスするメモリブロックは、AJAN以外の関数呼び出しで確保したものに対して行ってください。 ・本関数はメモリブロックの実体の管理を行いません。実体の無い不正なアドレスを指定したり、メモリブロックを free 関数などで解放されると、Segmentation Faultの発生や、場合によってはOSが危険な状態に陥る可能性がありますので呼び出しは慎重に行ってください。		
使用例	’C言語のメモリ確保関数を定義します CDECLARE ″libc.so.6″, ″void* malloc(size_t size);″ ’C言語のmalloc 関数を呼び出して、アドレス値を ptr& に受け取ります errno = CFUNCALL(ptr&, ″malloc″, 100) OBJECT MEM@ AS MEMORY ’C言語のmalloc関数で確保した領域を指定します MEM@ = CMEMMAP@(ptr&, 100) ’メモリブロックのポインタのアドレス値を表示します。 PRINT HEX$(MEM@.TOLNG())		

5.5.3 CSTRUCT@

関数			
機　能	指定したC言語構造体名に該当するサイズのメモリブロックを確保します。		
書　式	〈(戻り値)MEMORY型オブジェクト〉= CSTRUCT@(〈①C言語構造体型名〉[,〈②要素数〉])		
戻り値	戻り値	**<MEMORY型オブジェクト>**	オブジェクト型
	MEMORY型オブジェクトが得られます。		
パラメータ	①	**〈C言語構造体型名〉**	文字列
	CDECLARE 命令で、定義した C言語の構造体の型名を指定します。 指定した構造体型の確保に必要なメモリサイズ × 要素数 を計算し、メモリブロックを確保します。		
	②	**〈要素数〉**	数値
	後から配列のようにアクセスできるようにする為に、指定した構造体型の必要メモリサイズに掛ける要素数を指定します。 省略すると、要素数は1として扱われます。		
備　考	・メモリブロックの確保は、C言語の malloc 関数が使用されます。		

使用例	' C言語のHOGE構造体 を定義して、これをAJANに読み込ませます。
	S$ = '''
	struct HOGE {
	　int　 id;
	　char　msg[20];
	};
	'''
	
	CDECLARE "", S$
	
	OBJECT PTR@ AS MEMORY
	' HOGE構造体が要求するサイズのメモリブロックを確保します
	PTR@ = CSTRUCT@("HOGE")
	' メモリブロックのポインタのアドレス値を表示します。
	PRINT HEX$(PTR@.TOLNG())		

5.5.4 CMEMMOVE

命令			
機　能	MEMORY型オブジェクトのメモリの特定領域をコピーします。		
書　式	CMEMMOVE〈①転送先MEMORY型変数〉,〈②転送先オフセット〉,〈③転送元MEMORY型変数〉,〈④転送元オフセット〉,〈⑤転送バイトサイズ〉		
パラメータ	①	**<転送元MEMORY型変数>**	オブジェクト型
	転送先となる、MEMORY型オブジェクト変数を指定します。		
	②	**<転送先オフセット>**	数値
	転送先のオフセット位置を0始まりのバイト単位で指定します。		
	③	**<転送先MEMORY型変数>**	オブジェクト型
	転送元となる、MEMORY型オブジェクト変数を指定します。		
	④	**<転送元オフセット>**	数値
	転送元のオフセット位置を0始まりのバイト単位で指定します。		
	⑤	**<転送バイトサイズ>**	数値
	転送元から転送先に、メモリ内容をコピーするサイズを、バイト単位で指定します。		
備　考	・転送する際、オフセットと転送バイトサイズが、確保しているメモリブロックのサイズを超えないでください。		
使用例	OBJECT DST@ AS MEMORY DST@ = CALLOC@(100) OBJECT SRC@ AS MEMORY SRC@ = CALLOC@(100) CMEMMOVE @DST, 10, SRC@, 20, 30		

5.5.5 <MEMORY>.PEEKBYTE

関数	
機　能	MEMORY型オブジェクトの指定したメモリ位置からバイトデータ値を読み取ります。
書　式	<(戻り値)バイトデータ値> = <①MEMORY型変数> . PEEKBYTE(<②オフセット>)
戻り値	戻り値 | **<バイトデータ値>** | 数値 オフセット位置からバイトデータ(8bit長の単精度整数値)で、読み取った値が得られます。
パラメータ	① | **<MEMORY型変数>** | オブジェクト型 メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。 ② | **<オフセット>** | 数値 オフセット位置を0始まりのバイト単位で指定します。
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKBYTE(10)

5.5.6 <MEMORY>.PEEKHALF

関数	
機　能	MEMORY型オブジェクトの指定したメモリ位置からワードデータ値を読み取ります。
書　式	<(戻り値)ワードデータ値> = <①MEMORY型変数> . PEEKHALF(<②オフセット>)
戻り値	戻り値 | **<ワードデータ値>** | 数値 オフセット位置からワードデータ(16bit長の単精度整数値)で、読み取った値が得られます。
パラメータ	① | **<MEMORY型変数>** | オブジェクト型 メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。 ② | **<オフセット>** | 数値 オフセット位置を0始まりのバイト単位で指定します。
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKHALF(10)

5.5.7 <MEMORY>.PEEKINT

関数			
機　能	MEMORY型オブジェクトの指定したメモリ位置からダブルワードデータ値を読み取ります。		
書　式	<(戻り値)ダブルワードデータ値> = <①MEMORY型変数> . PEEKINT(<②オフセット>)		
戻り値	戻り値	**<ダブルワードデータ値>**	数値
	オフセット位置からダブルワードデータ(32bit長の単精度整数値)で、読み取った値が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKINT(10)		

5.5.8 <MEMORY>.PEEKLNG

関数			
機　能	MEMORY型オブジェクトの指定したメモリ位置からロングダブルワードデータ値を読み取ります。		
書　式	<(戻り値)ロングダブルワードデータ値> = <①MEMORY型変数> . PEEKLNG(<②オフセット>)		
戻り値	戻り値	**<ロングダブルワードデータ値>**	数値
	オフセット位置からロングダブルワードデータ(64bit長の倍精度整数値)で、読み取った値が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKLNG(10)		

5.5.9 <MEMORY>.PEEKSNG

関数			
機　能	MEMORY型オブジェクトの指定したメモリ位置から単精度実数値を読み取ります。		
書　式	<(戻り値)単精度実数値> = <①MEMORY型変数> . PEEKSNG(<②オフセット>)		
戻り値	戻り値	**<単精度実数値>**	数値
	オフセット位置から単精度実数(32bit長)で、読み取った値が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKSNG(10)		

5.5.10 <MEMORY>.PEEKDBL

関数			
機　能	MEMORY型オブジェクトの指定したメモリ位置から倍精度実数値を読み取ります。		
書　式	<(戻り値)倍精度実数値> = <①MEMORY型変数> . PEEKDBL(<②オフセット>)		
戻り値	戻り値	**<倍精度実数値>**	数値
	オフセット位置から倍精度実数(64bit長)で、読み取った値が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKDBL(10)		

5.5.11 <MEMORY>.PEEKSTR$

関数			
機　能	MEMORY型オブジェクトの指定したメモリ位置から指定サイズ長のバイナリ文字列を読み取ります。		
書　式	〈(戻り値)バイナリ文字列値〉＝〈①MEMORY型変数〉．PEEKSTR$(〈②オフセット〉,〈③バイトサイズ〉)		
戻り値	戻り値	**<バイナリ文字列値>**	文字列
	オフセット位置から指定したバイトサイズ分の、読み取ったバイナリ文字列値が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<バイトサイズ>**	数値
	メモリから読み取るサイズをバイト単位で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "値="; PTR@.PEEKSTR$(10, 20)		

5.5.12 <MEMORY>.POKEBYTE

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、バイトデータ値を書き込みます。		
書　式	<①MEMORY型変数> . POKEBYTE(<②オフセット>, <③バイトデータ値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<バイトデータ値>**	数値
	オフセット位置に書き込む値を、バイトデータ(8bit長の単精度整数値)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKEBYTE(10, &H12)		

5.5.13 <MEMORY>.POKEHALF

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、ワードデータ値を書き込みます。		
書　式	<①MEMORY型変数> . POKEHALF(<②オフセット>, <③ワードデータ値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<ワードデータ値>**	数値
	オフセット位置に書き込む値を、ワードデータ(16bit長の単精度整数値)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKEHALF(10, &H1234)		

5.5.14 <MEMORY>.POKEINT

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、ダブルワードデータ値を書き込みます。		
書　式	<①MEMORY型変数> . POKEINT(<②オフセット>, <③ダブルワードデータ値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<ダブルワードデータ値>**	数値
	オフセット位置に書き込む値を、ダブルワードデータ(32bit長の単精度整数値)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKEINT(10, &H12345678)		

5.5.15 <MEMORY>.POKELNG

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、ロングダブルワードデータ値を書き込みます。		
書　式	<①MEMORY型変数> . POKELNG(<②オフセット>, <③ロングダブルワードデータ値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<ロングダブルワードデータ値>**	数値
	オフセット位置に書き込む値を、ロングダブルワードデータ(64bit長の倍精度整数値)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKELNG(10, &H1234567812345678&)		

5.5.16 <MEMORY>.POKESNG

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、単精度実数値を書き込みます。		
書　式	<①MEMORY型変数> . POKESNG(<②オフセット>, <③単精度実数値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となる、MEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<単精度実数値>**	数値
	オフセット位置に書き込む値を、単精度実数(32bit長)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKESNG(10, 123.45)		

5.5.17 <MEMORY>.POKEDBL

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、倍精度実数値を書き込みます。		
書　式	<①MEMORY型変数> . POKEDBL(<②オフセット>, <③倍精度実数値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となる、MEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<倍精度実数値>**	数値
	オフセット位置に書き込む値を、倍精度実数(64bit長)で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKEDBL(10, 123.45)		

5.5.18 <MEMORY>.POKESTR

命令			
機　能	MEMORY型オブジェクトの指定したメモリ位置に、バイナリ文字列値を書き込みます。		
書　式	<①MEMORY型変数> . POKESTR(<②オフセット>, <③バイナリ文字列値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<オフセット>**	数値
	オフセット位置を0始まりのバイト単位で指定します。		
	③	**<バイナリ文字列値>**	文字列
	オフセット位置に書き込む値を、バイナリ文字列で指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PTR@.POKESTR(10, "Hello AJAN")		

5.5.19 <MEMORY>.GETMEMOFF

関数

機　能

「CSTRUCT@」の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当するオフセット位置を得ます。

書　式

<(戻り値)オフセット位置> = <①MEMORY型変数> . GETMEMOFF([<②添字位置>,] <③メンバ名>)

戻り値

戻り値	**<オフセット位置>**	数値

メンバ名で指定したオフセット位置を得ます。

パラメータ

①	**<MEMORY型変数>**	オブジェクト型

メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。

②	**<添字位置>**	数値

「CSTRUCT@」に要素数を2以上指定した後、
メモリブロックを構造体配列としてアクセスするための添字位置を指定します。
省略時、0 として扱います。

③	**<メンバ名>**	文字列

「CSTRUCT@」に指定した C言語の構造体型に対して、アクセスしたいメンバ名を指定します。
配列にアクセスする場合、C言語の指定と同じように [] (角カッコ)で指定します。
メンバ内メンバにアクセスする場合、C言語と同じように「.(ドット)」で指定します。

注　意

「CSTRUCT@」で作成した MEMORY型変数のみ、呼び出し可能です。

使用例

```
' C言語のHOGE構造体を定義して、これをAJANに読み込ませます。
S$ = '''
struct HOGE {
    int    id;
    int    ary[5];
};
'''

CDECLARE "", S$

OBJECT PTR@ AS MEMORY
' HOGE構造体が要求するサイズのメモリブロックを確保します
PTR@ = CSTRUCT@("HOGE")
' HOGE構造体の ary[1]メンバ に相当するオフセット位置を得ます
OFF = PTR@.GETMEMOFF("ary[1]")
```

5.5.20 <MEMORY>.GETMEMVAL

関数

機　能

「CSTRUCT@」の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当する位置の値を取得します。

書　式

<(戻り値)メンバ値> = <①MEMORY型変数> . GETMEMVAL([<②添字位置>,] <③メンバ名>)

戻り値

戻り値	**<メンバ値>**	数値

メンバ名で指定した位置のメモリの値を数値で得ます。

パラメータ

①	**<MEMORY型変数>**	オブジェクト型

メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。

②	**<添字位置>**	数値

「CSTRUCT@」に要素数を2以上指定した後、
メモリブロックを構造体配列としてアクセスするための添字位置を指定します。
省略時、0 として扱います。

③	**<メンバ名>**	文字列

「CSTRUCT@」に指定したC言語の構造体型のアクセスしたいメンバ名を指定します。
配列にアクセスする場合、C言語の指定と同じように []（角カッコ)で指定します。
メンバ内メンバにアクセスする場合、C言語と同じように「.(ドット)」で指定します。

注　意

「CSTRUCT@」で作成した MEMORY型変数のみ、呼び出し可能です。

使用例

```
' C言語のHOGE構造体を定義して、これをAJANに読み込ませます。
S$ = '''
struct HOGE {
    int    id;
    int    ary[5];
};
'''
CDECLARE "", S$

OBJECT PTR@ AS MEMORY
' HOGE構造体が要求するサイズのメモリブロックを確保します
PTR@ = CSTRUCT@("HOGE")
' HOGE構造体の ary[1]メンバ に相当する値を得ます
PRINT PTR@.GETMEMVAL("ary[1]")
```

5.5.21 <MEMORY>.GETMEMSTR$

関数			
機　能	「CSTRUCT@」の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当する位置の文字列を取得します。		
書　式	<(戻り値)メンバ文字列> = <①MEMORY型変数> . GETMEMSTR$([<②添字位置>,] <③メンバ名>)		
戻り値	戻り値	**<メンバ文字列>**	文字列
	メンバ名で指定した位置のメモリの値を文字列で得ます。 文字列として取得する範囲は、メンバの文字列配列の要素数です。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<添字位置>**	数値
	「CSTRUCT@」に要素数を2以上を指定した後、 メモリブロックを構造体配列としてアクセスするための添字位置を指定します。 省略時、0 として扱います。		
	③	**<メンバ名>**	文字列
	「CSTRUCT@」に指定したC言語の構造体型のアクセスしたいメンバ名を指定します。 配列にアクセスする場合、C言語の指定と同じように []（角カッコ）で指定します。 メンバ内メンバにアクセスする場合、C言語と同じように「.（ドット）」で指定します。		
注　意	「CSTRUCT@」で作成した MEMORY型変数のみ、呼び出し可能です。		

使用例	’ C言語のHOGE構造体 を定義して、これをAJANに読み込ませます。
	S$ = '''
	struct HOGE {
	　int　 id;
	　char　 name[10];
	};
	'''
	
	CDECLARE "", S$
	
	OBJECT PTR@ AS MEMORY
	’ HOGE構造体が要求するサイズのメモリブロックを確保します
	PTR@ = CSTRUCT@("HOGE")
	’ HOGE構造体の nameメンバ に相当する値を得ます
	PRINT PTR@.GETMEMVAL("name")		

5.5.22 <MEMORY>.SETMEMVAL

命令			
機　能	「CSTRUCT@」の引数で与えたC言語の構造体型に対して、指定したメンバ名に相当する位置に値を書き込みます。		
書　式	<①MEMORY型変数> . SETMEMVAL([<②添字位置>,] <③メンバ名>, <④設定値>)		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。		
	②	**<添字位置>**	数値
	「CSTRUCT@」に要素数を2以上を指定した後、 メモリブロックを構造体配列としてアクセスするための添字位置を指定します。 省略時、0 として扱います。		
	③	**<メンバ名>**	文字列
	「CSTRUCT@」に指定したC言語の構造体型のアクセスしたいメンバ名を指定します。 配列にアクセスする場合、C言語の指定と同じように []（角カッコ）で指定します。 メンバ内メンバにアクセスする場合、C言語と同じように「.（ドット）」で指定します。		
	④	**<設定値>**	値
	書き込みたい値を数値または文字列で指定します。		
注　意	「CSTRUCT@」で作成したMEMORY型オブジェクト変数にのみ使用可能です。		

使用例	' C言語のHOGE構造体 を定義して、これをAJANに読み込ませます。
	S$ = '''
	struct HOGE {
	　int　id;
	　int　ary[5];
	};
	'''
	
	CDECLARE "", S$
	
	OBJECT PTR@ AS MEMORY
	' HOGE構造体が要求するサイズのメモリブロックを確保します
	PTR@ = CSTRUCT@("HOGE")
	' HOGE構造体の ary[1]メンバ に相当する位置に値を書き込みます。
	PTR@.SETMEMVAL("ary[1]", 123)
	' HOGE構造体の ary[1]メンバ に相当する値を得ます(123が得られるハズです)
	PRINT PTR@.GETMEMVAL("ary[1]")		

5.5.23 <MEMORY>.SIZEOF

関数			
機　能	MEMORY型オブジェクト変数のメモリブロックサイズが得られます。		
書　式	<(戻り値)メモリサイズ長> = <①MEMORY型変数> . SIZEOF( )		
戻り値	戻り値	**<メモリサイズ長>**	数値
	メモリブロックのメモリサイズが、バイト単位で得られます。 「CALLOC@」で指定したバイトサイズが得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	MEMORY型オブジェクト変数を指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "メモリサイズ="; PTR@.SIZEOF()　' 100が得られます		

5.5.24 <MEMORY>.TOJSON$

関数			
機　能	「CSTRUCT@」の引数で与えたC言語の構造体型に対して、構造体の各メンバ名とメモリから取得した値を、JSON形式の文字列で得られます。		
書　式	<(戻り値)JSON形式の文字列> = <①MEMORY型変数> . TOJSON$( )		
戻り値	戻り値	**<JSON形式の文字列>**	文字列
	JSON形式の文字列が得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	MEMORY型オブジェクト変数を指定します。		
備　考	・構造体の各メンバ名と値が、JSON形式で表現されます。 ・C言語の各メンバの型で、char型の配列は 文字列様式で表現されます。 それ以外の型は、数値様式で表現されます。 ポインタ型も、ポインタアドレス値を数値様式で表現されます。		
注　意	・「CSTRUCT@」で作成したMEMORY型オブジェクト変数にのみ使用可能です。		

使用例	' C言語のHOGE構造体 を定義して、これをAJANに読み込ませます。
	S$ = '''
	struct HOGE {
	　int　id;
	　char　name[10];
	};
	'''
	
	CDECLARE "", S$
	
	OBJECT PTR@ AS MEMORY
	' HOGE構造体が要求するサイズのメモリブロックを確保します
	PTR@ = CSTRUCT@("HOGE")
	' HOGE構造体の各メンバとメモリの値を、JSON形式の文字列で得られます。
	PRINT PTR@.TOJSON$()		

5.5.25 <MEMORY>.TOLNG

関数			
機　能	MEMORY型で指定した、メモリブロックのアドレス値が得られます。		
書　式	<(戻り値)アドレス値> = <①MEMORY型変数> . TOLNG( )		
戻り値	戻り値	**<アドレス値>**	数値
	メモリブロックのアドレス値が、倍精度整数値で得られます。		
パラメータ	①	**<MEMORY型変数>**	オブジェクト型
	MEMORY型オブジェクト変数を指定します。		
使用例	OBJECT PTR@ AS MEMORY PTR@ = CALLOC@(100) PRINT "アドレス="; HEX$(PTR@.TOLNG())		

5.5.26 <MEMORY>.TOSTRTYPE$

関数

機　能

MEMORY型オブジェクトの指定したメモリ位置から指定した型名の数値または文字列を文字列化します。

書　式

<(戻り値)文字列値> = <①MEMORY型変数> . TOSTRTYPE$(<②型名>, <③オフセット>, [<④バイトサイズ>])

戻り値

戻り値	<文字列値>	文字列

オフセット位置から指定した型名の数値または文字列を文字列化します。

パラメータ

①	<MEMORY型変数>	オブジェクト型

メモリ操作の対象となるMEMORY型オブジェクト変数を指定します。

②	<型名>	文字列

オフセット位置から数値ないしは文字列として取り出したい型名を指定します。

型名	動作
int8	オフセットから1バイトの整数値を読み取り、-128 ～ 127の数文字列として返します。
uint8	オフセットから1バイトの整数値を読み取り、0 ～ 255 の数文字列として返します。
int16	オフセットから2バイトの整数値を読み取り、-32,768 ～ 32,767の数文字列として返します。
uint16	オフセットから2バイトの整数値を読み取り、0 ～ 65,535の数文字列として返します。
int32	オフセットから4バイトの整数値を読み取り、-2,147,483,648 ～ 2,147,483,647 の数文字列として返します。
uint32	オフセットから4バイトの整数値を読み取り、0 ～ 4,294,967,295 の数文字列として返します。
int64	オフセットから8バイトの整数値を読み取り、 -9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807 の数文字列として返します。
uint64	オフセットから8バイトの整数値を読み取り、 0 ～ 18,446,744,073,709,551,615 の数文字列として返します。
float	オフセットから4バイトの実数値を読み取り、-3.402823E+38 ～ 3.402823E+38 の数文字列として返します。
double	オフセットから8バイトの実数値を読み取り、-1.79769313486232E+308 ～ 1.79769313486232E+308 の数文字列として返します。
str	オフセットからバイトサイズまでをバイナリ文字列として返します。

③	<オフセット>	数値

オフセット位置を0始まりのバイト単位で指定します。

④	<バイトサイズ>	数値

メモリから読み取るサイズをバイト単位で指定します。
型名で「str」を指定した時に有効です。

使用例

```
OBJECT v@ AS MEMORY
v@ = CALLOC@(100)
v@.POKEHALF(0, 1234)
? v@.TOSTRTYPE$("int16", 0)’ 1234が表示されます
? v@.TOSTRTYPE$("uint16", 0)      ’ 1234が表示されます
v@.POKEHALF(0, &HFFFF)
? v@.TOSTRTYPE$("int16", 0)’ -1が表示されます
? v@.TOSTRTYPE$("uint16", 0)      ’ 65535が表示されます
```

5.6 JSON型に関する関数・命令

JSON型は、JSON(JavaScript Object Notation)というデータ交換用の書式で書かれた文字列を任意に扱うためのオブジェクト変数です。
JSONは、Webブラウザ上で動作する JavaScript と サーバとの間でやり取りする、データ(情報)交換の記述書式に JSON 形式が多く使われており、最近では他のプログラミング言語やデータベースの格納書式にも採用が拡がっています。
AJANのJSON型は、このJSON形式のデータを読み取ったり、JSON形式の文字列を作るために用います。

5.6.1 JSON_PARSE@

関数			
機　能	JSON形式の文字列を解析し、JSON型オブジェクトを得ます。		
書　式	<(戻り値)JSONオブジェクト> = JSON_PARSE@(<①JSON形式の文字列>)		
戻り値	戻り値	**<JSON型オブジェクト>**	オブジェクト型
	JSON型オブジェクトが得られます。		
パラメータ	①	**<JSON形式の文字列>**	文字列
	JSON形式で記述された文字列を指定します。		
備　考	・解析の可否判定を行いたい場合は、JSON_TRYPARSE 関数を使ってください。		
使用例	OBJECT PT@ AS JSON ' JSON形式文字列からオブジェクトを生成する PT@ = JSON_PARSE@("{ ""name"": ""ayumu"", ""age"": 7 }") ' JSON型オブジェクトを文字列化します PRINT JSON_DUMP$(PT@)		

5.6.2 JSON_TRYPARSE

関数			
機　能	JSON形式文字列の解析に挑戦し、その可否を返します。解析に成功すれば、JSON型オブジェクトをセットします。		
書　式	<(戻り値)解析可否> = JSON_TRYPARSE(<①JSON型変数>, <②JSON形式の文字列>)		
戻り値	戻り値	**<解析可否>**	真偽値
	JSON形式文字列の解析に成功すると TRUE、失敗すると FALSE を返します。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSON形式の文字列の解析に成功すると、この変数にJSON型オブジェクトをセットします。		
	②	**<JSON形式の文字列>**	文字列
	JSON形式で記述された文字列を指定します。		

使用例	OBJECT PT@ AS JSON
	' JSON形式文字列からオブジェクトを生成する
	IF JSON_TRYPARSE(PT@, "{ ""name"": ""ayumu"", ""age"": 7 }") = TRUE THEN
	　' JSON型オブジェクトを文字列化します
	　PRINT JSON_DUMP$(PT@)
	ELSE
	　PRINT "JSON解析に失敗しました"
	END IF		

5.6.3 JSON_NEW@

関数			
機　能	指定した数値または文字列からJSON型オブジェクトを得ます。		
書　式	<(戻り値)JSON型オブジェクト> = JSON_NEW@(<①文字列／数値>)		
戻り値	戻り値	**<JSON型オブジェクト>**	オブジェクト型
	JSON型オブジェクトが得られます。		
パラメータ	①	**<文字列／数値>**	値
	変換したい、文字列／数値を指定します。 指定する値の型により、得られる JSONの内部の型が異なります。		

AJANの型(引数)	JSONの内部の型(戻り値)
文字列	文字列
単精度整数、倍精度整数 単精度実数、倍精度実数	数値
TRUE, FALSE	真偽値

備　考	・JSONの仕様では、数値は本来、浮動小数点数(AJANで言う倍精度実数)で まとめて扱われます。しかし、AJANでは整数型の数値を明示的に渡す事で、整数の数値として扱う事が可能です。 ・JSONの空のオブジェクトを得たい場合、JSON_PARSE@ で以下のように書きます。 「JSON_PARSE@("{}")」 ・JSONの空の配列を得たい場合、JSON_PARSE@ で以下のように書きます。 「JSON_PARSE@("[]")」 ・JSONのnull値を得たい場合、JSON_PARSE@ で以下のように書きます。 「JSON_PARSE@("null")」
使用例	OBJECT PT@ AS JSON ' 123という数値から、JSON型オブジェクトを生成する PT@ = JSON_NEW@(123) '「Hello」という文字列から、JSON型オブジェクトを生成する PT@ = JSON_NEW@("Hello")

5.6.4 JSON_DUMP$

関数			
機　能	JSON型オブジェクトからJSON形式の文字列が得られます。		
書　式	<(戻り値)JSON形式の文字列> = JSON_DUMP$(<①JSON型変数>)		
戻り値	戻り値	**<JSON形式の文字列>**	文字列
	JSON形式の文字列が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON ' JSON形式文字列からJSON型オブジェクトを生成する PT@ = JSON_PARSE@("{ ""name"": ""ayumu"", ""age"": 7 }") ' JSON型オブジェクトを文字列化します PRINT JSON_DUMP$(PT@)		

5.6.5 〈JSON〉.TYPEOF

関数

機　能

JSON型オブジェクトの種類を調べます。

書　式

〈(戻り値)種別情報〉 = 〈①JSON型変数〉 . TYPEOF( )

戻り値

戻り値	<種別情報>	数値

JSON型オブジェクトの種別情報(内部の型)が、数値で得られます。

戻り値	JSON型オブジェクト種別
0	オブジェクト(連想配列)
1	配列
2	文字列
3	整数値
4	実数値
5	TRUE
6	FALSE
7	NULL

パラメータ

①	<JSON型変数>	オブジェクト型

JSON型オブジェクト変数を指定します。

使用例

```
OBJECT PT@ AS JSON
' JSON型オブジェクトを作ります
PT@ = JSON_PARSE@("{ ""name"": ""ayumu"", ""age"": 7 }")
' JSON型オブジェクトの種別を得ます。0が表示されます。
PRINT PT@.TYPEOF()
' 文字列からJSON型オブジェクトを作ります
PT@ = JSON_NEW@("Hello")
' JSON型オブジェクトの種別を得ます。2が表示されます。
PRINT PT@.TYPEOF()
```

5.6.6 <JSON>.GET_CSTR$

関数			
機　能	JSON型オブジェクトから文字列を得ます。		
書　式	<(戻り値)文字列> = <①JSON型変数> . GET_CSTR$( )		
戻り値	戻り値	**<文字列>**	文字列
	JSON型オブジェクトから、文字列が得られます。 JSONの内部の型が文字列以外だと、エラーになります。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON '文字列のJSON型オブジェクトを得ます PT@ = JSON_NEW@("Hello") ' JSON型オブジェクトから、文字列を得ます。「Hello」が表示されます PRINT PT@.GET_CSTR$()		

5.6.7 <JSON>.GET_CLNG

関数			
機　能	JSON型オブジェクトから整数値を得ます。		
書　式	<(戻り値)整数値> = <①JSON型変数> . GET_CLNG( )		
戻り値	戻り値	**<整数値>**	数値
	JSON型オブジェクトから、倍精度整数値が得られます。 JSONの内部の型が整数値以外だと、エラーになります。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON ' 数値のJSON型オブジェクトを得ます PT@ = JSON_NEW@(123%) ' JSON型オブジェクトから整数値を得ます。「123」が表示されます PRINT PT@.GET_CLNG()		

5.6.8 〈JSON〉.GET_CDBL

関数			
機　能	JSON型オブジェクトから実数値を得ます。		
書　式	〈(戻り値)実数値〉 = 〈①JSON型変数〉 . GET_CDBL( )		
戻り値	戻り値	**<実数値>**	数値
	JSON型オブジェクトから、倍精度実数値が得られます。 JSONの内部の型が整数または実数値以外だと、エラーになります。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON ’数値のJSON型オブジェクトを得ます PT@ = JSON_NEW@(123.45) ’JSON型オブジェクトから実数値を得ます。「123.45」が表示されます PRINT PT@.GET_CDBL()		

5.6.9 〈JSON〉.GET_CBOOL

関数			
機　能	JSON型オブジェクトから真偽値を得ます。		
書　式	〈(戻り値)真偽値〉 = 〈①JSON型変数〉 . GET_CBOOL( )		
戻り値	戻り値	**<真偽値>**	数値
	JSON型オブジェクトから、真偽値が得られます。 JSONの内部の型が真偽値以外だと、エラーになります。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON ’真偽値のJSON型オブジェクトを得ます PT@ = JSON_NEW@(TRUE) ’JSON型オブジェクトから、真偽値を得ます。「TRUE」が表示されます PRINT PT@.GET_CBOOL()		

5.6.10 <JSON>.ARRAY_SIZE

関数			
機　能	JSON型オブジェクトから配列の要素数を得ます。		
書　式	<(戻り値)配列の要素数> = <①JSON型変数> . ARRAY_SIZE( )		
戻り値	戻り値	**<配列の要素数>**	数値
	JSON型オブジェクトから、配列の要素数が得られます。 JSONの内部の型が配列以外だと0が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、配列のものを指定してください。		
使用例	OBJECT PT@ AS JSON ’配列のJSON型オブジェクトを得ます PT@ = JSON_NEW@([1; 2; 3]) ’JSON型オブジェクトから、配列の要素数を得ます。3が表示されます。 PRINT PT@.ARRAY_SIZE()		

5.6.11 <JSON>.ARRAY_GET@

関数			
機　能	JSON型オブジェクトから配列の指定した添字のJSON型オブジェクトを得ます。		
書　式	<(戻り値)JSON型オブジェクト> = <①JSON型変数> . ARRAY_GET@(<②配列の添字>)		
戻り値	戻り値	**<JSON型オブジェクト>**	オブジェクト型
	JSON型オブジェクトに対して、指定した配列の添字から、 JSON型オブジェクトを得ます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、配列のものを指定してください。		
	②	**<配列の添字>**	数値
	取得したいオブジェクトの配列の添字を指定します。 0始まりで指定します。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’配列のJSON型オブジェクトを得ます PT@ = JSON_NEW@([1; 2; 3]) ’JSON型オブジェクトから、添字=1のオブジェクトを得ます。 P2@ = PT@.ARRAY_GET@(1)		

5.6.12 〈JSON〉.ARRAY_SET

<table>
<tr><td colspan="3">命令</td></tr>
<tr><td>機　能</td><td colspan="2">JSON型オブジェクトの配列の指定した添字に値を設定します。</td></tr>
<tr><td>書　式</td><td colspan="2">〈①JSON型変数〉. ARRAY_SET(〈②配列の添字〉, 〈③設定値〉)</td></tr>
<tr><td>パラメータ</td><td>①</td><td><JSON型変数>　オブジェクト型
JSON型オブジェクト変数を指定します。
JSONの内部の型が、配列のものを指定してください。</td></tr>
<tr><td></td><td>②</td><td><配列の添字>　数値
設定したい配列の添字を指定します。
0始まりで指定します。</td></tr>
<tr><td></td><td>③</td><td><設定値>　値
設定する値を指定します。
数値、文字列、JSON型値が指定できます。</td></tr>
<tr><td>備　考</td><td colspan="2">JSON_NEW@で指定できる引数が、設定値で使用できます。</td></tr>
<tr><td>使用例</td><td colspan="2">OBJECT PT@ AS JSON, P2@ AS JSON
’配列のJSON型オブジェクトを得ます
PT@ = JSON_NEW@([1; 2; 3])
’JSON型オブジェクトから、添字=1の値を書き換えます。
PT@.ARRAY_SET(1, 123)
’JSON型オブジェクトを文字列化します。「1,123,3」と表示されます。
PRINT JSON_DUMP$(PT@)</td></tr>
</table>

5.6.13 〈JSON〉.ARRAY_INSERT

命令			
機　能	JSON型オブジェクトの配列の指定した添字位置に値を挿入／追加します。		
書　式	〈①JSON型変数〉. ARRAY_INSERT(〈②配列の添字位置〉, 〈③設定値〉)		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、配列のものを指定してください。		
	②	**<配列の添字位置>**	数値
	挿入／追加したい配列の添字位置を指定します。 0始まりで指定し、指定した位置に値を挿入します。 -1を指定すると、末尾に値を追加します。		
	③	**<設定値>**	値
	挿入／追加する値を指定します。 数値、文字列、JSON型値が指定できます。		
備　考	JSON_NEW@ で指定できる引数が、設定値で使用できます。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’配列のJSON型オブジェクトを得ます PT@ = JSON_NEW@([1; 2; 3]) ’JSON型オブジェクトから、配列の末尾に値を追加します。 PT@.ARRAY_INSERT(-1, 123) ’JSON型オブジェクトを文字列化します。「1,2,3,123」と表示されます。 PRINT JSON_DUMP$(PT@)		

5.6.14 <JSON>.ARRAY_REMOVE

関数			
機　能	JSON型オブジェクトの配列の指定した添字位置の値を削除します。		
書　式	<(戻り値)成功可否> = <①JSON型変数> . ARRAY_REMOVE(<②配列の添字位置>)		
戻り値	戻り値	**<成功可否>**	真偽値
	配列に対して、値を削除できたら TRUEを、 失敗したら FALSE が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、配列のものを指定してください。		
	②	**<配列の添字位置>**	数値
	削除したい配列の添字位置を指定します。 0始まりで指定し、指定した位置の値を削除します。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’配列のJSON型オブジェクトを得ます PT@ = JSON_NEW@([1; 2; 3]) ’JSON型オブジェクトから、配列の添字=1の値を削除します。 PRINT PT@.ARRAY_REMOVE(1) ’JSON型オブジェクトを文字列化します。「1,3」と表示されます。 PRINT JSON_DUMP$(PT@)		

5.6.15 <JSON>.ARRAY_CLEAR

命令			
機　能	JSON型オブジェクトの、配列の全ての要素を削除します。		
書　式	<①JSON型変数> . ARRAY_CLEAR( )		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、配列のものを指定してください。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’配列のJSON型オブジェクトを得ます PT@ = JSON_NEW@([1; 2; 3]) ’JSON型オブジェクトから、配列の全ての要素を削除します。 PT@.ARRAY_CLEAR() ’JSON型オブジェクトの、配列の要素数を調べます。0が表示されます。 PRINT PT@.ARRAY_SIZE()		

5.6.16 <JSON>.OBJECT_SIZE

関数			
機　能	JSON型オブジェクトから、オブジェクト(連想配列)の要素数を得ます。		
書　式	<(戻り値)連想配列の要素数> = <①JSON型変数> . OBJECT_SIZE( )		
戻り値	戻り値	**<配列の要素数>**	数値
	JSON型オブジェクトから、オブジェクト(連想配列)の要素数が得られます。 JSONの内部の型がオブジェクト以外だと、0が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(" { ""name"": ""ayumu"", ""age"": 7 } ") ’JSON型オブジェクトから、連想配列の要素数を得ます。2が表示されます。 PRINT PT@.OBJECT_SIZE()		

5.6.17 <JSON>.OBJECT_KEYS$

関数			
機　能	JSON型オブジェクトから、オブジェクト(連想配列)のキー配列を得ます。		
書　式	<(戻り値)連想配列のキー配列> = <①JSON型変数> . OBJECT_KEYS$( )		
戻り値	戻り値	**<連想配列のキー配列>**	配列
	JSON型オブジェクトから、オブジェクト(連想配列)のキー文字列の1次元配列が得られます。 JSONの内部の型がオブジェクト以外だと、0が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(" { ""name"": ""ayumu"", ""age"": 7 } ") ’JSON型オブジェクトから、連想配列のキー配列を得ます。 PRINT PT@.OBJECT_KEYS$()		

5.6.18 〈JSON〉.OBJECT_GET@

関数			
機　能	JSON型オブジェクトから、オブジェクト(連想配列)の指定したキーのオブジェクトを得ます。		
書　式	〈(戻り値)JSON型オブジェクト〉=〈①JSON型変数〉. OBJECT_GET@(〈②キー名〉)		
戻り値	戻り値	**<JSON型オブジェクト>**	オブジェクト型
	JSON型オブジェクトに対して、指定したオブジェクト(連想配列)のキーから、対応した JSON型オブジェクトを得ます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
	②	**<キー名>**	文字列
	取得したい連想配列のキー名を指定します。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(″ { ″″name″″: ″″ayumu″″, ″″age″″: 7 } ″) ’JSON型オブジェクトから、「name」キーのオブジェクトを得ます。 PRINT PT@.OBJECT_GET@(″name″)		

5.6.19 〈JSON〉.OBJECT_SET

命令			
機　能	JSON型オブジェクトの、オブジェクト(連想配列)の指定したキーに対して、値を設定します。		
書　式	〈①JSON型変数〉. OBJECT_SET(〈②キー名〉,〈③設定値〉)		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
	②	**<キー名>**	文字列
	設定したいオブジェクトの連想配列のキー名を指定します。		
	③	**<設定値>**	値
	設定する値を指定します。 数値、文字列、JSON型値が指定できます。		
備　考	JSON_NEW@ で指定できる引数が、設定値で使用できます。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(″ { ″″name″″: ″″ayumu″″, ″″age″″: 7 } ″) ’JSON型オブジェクトから、「age」キーの値を書き換えます。 PT@.OBJECT_SET(″age″, 8) ’JSON型オブジェクトに「height」キーと値を追加します。 PT@.OBJECT_SET(″height″, 120)		

5.6.20 <JSON>.OBJECT_DEL_KEY

命令			
機　能	JSON型オブジェクトの、オブジェクト(連想配列)の指定したキーに紐付けされたデータを削除します。		
書　式	<(戻り値)成功可否> = <①JSON型変数> . OBJECT_DEL_KEY(<②キー名>)		
戻り値	戻り値	**<成功可否>**	真偽値
	連想配列に対して、指定したキーのデータを削除できたら TRUEを、 失敗したら FALSE が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
	②	**<キー名>**	文字列
	削除したいオブジェクトの連想配列のキー名を指定します。 0始まりで指定します。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(″ { ″″name″″: ″″ayumu″″, ″″age″″: 7 } ″) ’JSON型オブジェクトから、「age」キーのデータを削除します。 PRINT PT@.OBJECT_DEL_KEY(″age″) ’キーの数を得ます。1が表示されます。 PRINT PT@.OBJECT_SIZE()		

5.6.21 <JSON>.OBJECT_CLEAR

命令			
機　能	JSON型オブジェクトの、オブジェクト(連想配列)の全ての要素を削除します。		
書　式	<①JSON型変数> . OBJECT_CLEAR( )		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。 JSONの内部の型が、オブジェクト(連想配列)のものを指定してください。		
使用例	OBJECT PT@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(″ { ″″name″″: ″″ayumu″″, ″″age″″: 7 } ″) ’JSON型オブジェクトから、全てのデータを削除します。 PT@.OBJECT_CLEAR( ) ’キーの数を得ます。0が表示されます。 PRINT PT@.OBJECT_SIZE()		

5.6.22 〈JSON〉.ISEQ

命令			
機　能	JSON型オブジェクト同士が、同じ内容(TRUE)か否(FALSE)か比較します。		
書　式	〈(戻り値)成功可否〉 = 〈①JSON型変数〉 . ISEQ(〈②比較JSON型変数〉)		
戻り値	戻り値	**<成功可否>**	真偽値
	JSON型オブジェクト同士が、同じ内容であれば TRUE を、 異なれば FALSE が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
	②	**<比較JSON型変数>**	オブジェクト型
	比較対象となる、JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’配列のJSON型オブジェクトを作ります。 PT@ = JSON_NEW@([1; 2; 3]) P2@ = JSON_NEW@([1; 2; 3; 4]) ’JSON型オブジェクト同士を比較します。中身が異なるので FALSE が表示されます。 PRINT PT@.ISEQ(P2@)		

5.6.23 〈JSON〉.CLONE@

関数			
機　能	JSON型オブジェクトの複製を作ります。		
書　式	〈(戻り値)JSON型オブジェクト〉 = 〈①JSON型変数〉 . CLONE@( )		
戻り値	戻り値	**<JSON型オブジェクト>**	オブジェクト型
	JSON型オブジェクトの複製が得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON, P2@ AS JSON ’連想配列のJSON型オブジェクトを得ます PT@ = JSON_PARSE@(" { ""name"": ""ayumu"", ""age"": 7 } ") ’JSON型オブジェクトの複製を作ります P2@ = PT@.CLONE@()		

5.6.24 <JSON>.TOLNG

関数			
機　能	JSON型の変数から、内部のアドレス値が得られます。		
書　式	<(戻り値)アドレス値> = <①JSON型変数> . TOLNG( )		
戻り値	戻り値	**<アドレス値>**	数値
	JSON型オブジェクト変数の内部のアドレス値が、倍精度整数値で得られます。		
パラメータ	①	**<JSON型変数>**	オブジェクト型
	JSON型オブジェクト変数を指定します。		
使用例	OBJECT PT@ AS JSON ’ アドレス値を得ます。未使用なので、0が得られます。 PRINT ″アドレス=″; HEX$(PT@.TOLNG()) ’ JSON形式文字列からJSON型オブジェクトを生成する PT@ = JSON_PARSE@(″{ ″″name″″: ″″ayumu″″, ″″age″″: 7 }″) ’ アドレス値を得ます。使用中なので、0以外が得られます。 PRINT ″アドレス=″; HEX$(PT@.TOLNG())		

第6章 サンプルプログラム

サンプルプログラムは「/usr/share/interface/AJANPro/samples/EXT/IO/」に格納されています。
AJAN統合開発環境を起動すると､左ペインのエクスプローラウィンドウ内の「Samples/EXT/IO/」に、ファイルが取り込まれて配置されます。

6.1 サンプルプログラム

#	ファイル名	内容
1	CFUNC_GPG2X72C_DiBm_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の DiBm_pae.c サンプルの移植例です。 サンプリングモードでサンプリングを行う単純なサンプルプログラムです。
2	CFUNC_GPG2X72C_DoBm_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の DoBm_pae.c サンプルの移植例です。 パターン出力モードで出力を行うサンプルプログラムです。 LPC/PEX-291144、LPC/PEX-292144 で出力使用時は、「dwRedirectWidth」 の設定を &H04 でなく &H02 を設定するようにしてください。
3	CFUNC_GPG2X72C_ExBm_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の ExBm_pae.c サンプルの移植例です。 バスマスタ DI 部とバスマスタ DO 部を同時に動作させるサンプルプログラムです。 LPC/PEX-291144、LPC/PEX-292144 で出力使用時は、「dwRedirectWidth」 の設定を &H04 でなく &H02 を設定するようにしてください。
4	CFUNC_GPG2X72C_exbm_event_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の exbm_event_pae.c サンプルの移植例です。 バスマスタ DI 部とバスマスタ DO 部を同時に動作させるサンプルプログラムです。 （サンプリング完了のコールバックを使用して同期します） LPC/PEX-291144、LPC/PEX-292144 で出力使用時は、「dwRedirectWidth」 の設定を &H04 でなく &H02 を設定するようにしてください。
5	CFUNC_GPG2X72C_patternout_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の patternout_pae.c サンプルの移植例です。 パターン出力モードで出力を行う単純なサンプルプログラムです。 （パターン出力に必要となる最小限の設定のみ実行します） LPC/PEX-291144、LPC/PEX-292144 で出力使用時は、「dwRedirectWidth」 の設定を &H04 でなく &H02 を設定するようにしてください。

#	ファイル名	内容
6	CFUNC_GPG2X72C_sampling_pae.AJN	C 言語連携機能を使って、GPG-2X72C のバスマスタライブラリを呼び出すサンプルプログラムです。 GPG-2X72C 付属の sampling_pae.c サンプルの移植例です。 サンプリングモードでサンプリングを行う単純なサンプルプログラムです。 （サンプリングに必要となる最小限の設定のみ実行します）
7	CFUNC_GPG2000_event.AJN	C 言語連携機能を使って、GPG-2000の DIO ライブラリを呼び出すサンプルプログラムです。 GPG-2000付属の event.c サンプルの移植例です。 割り込みイベントを捉えて表示します。
8	CFUNC_GPG2000_inpoint.AJN	C 言語連携機能を使って、GPG-2000の DIO ライブラリを呼び出すサンプルプログラムです。 GPG-2000付属の inpoint.c サンプルの移植例です。 「DioInputPoint」 関数を用いて、指定点数のデジタル入力を実施した結果を表示します。
9	CFUNC_GPG2000_inputbyte.AJN	C 言語連携機能を使って、GPG-2000の DIO ライブラリを呼び出すサンプルプログラムです。 GPG-2000付属の inputbyte.c サンプルの移植例です。 「DioInputByte」 関数を用いて、1件のデジタル入力を実施した結果を表示します。
10	CFUNC_GPG2000_outpoint.AJN	C 言語連携機能を使って、GPG-2000の DIO ライブラリを呼び出すサンプルプログラムです。 GPG-2000付属の outpoint.c サンプルの移植例です。 「DioOutputPoint」 関数を用いて、指定点数のデジタル出力を実施した結果を表示します。
11	CFUNC_GPG2000_outputbyte.AJN	C 言語連携機能を使って、GPG-2000の DIO ライブラリを呼び出すサンプルプログラムです。 GPG-2000付属の outputbyte.c サンプルの移植例です。 「DioOutputByte」 関数を用いて、1件のデジタル出力を実施した結果を表示します。
12	CFUNC_GPG3100_AdInputAD.AJN	C 言語連携機能を使って、GPG-3100の AD ライブラリを呼び出すサンプルプログラムです。 GPG-3100付属の AdInputAD.c サンプルの移植例です。 「AdInputAD」 関数を用いて、1件のアナログ入力を実施した結果を表示します。
13	CFUNC_GPG3100_AdSampling.AJN	C 言語連携機能を使って、GPG-3100の AD ライブラリを呼び出すサンプルプログラムです。 GPG-3100付属の AdSampling.c サンプルの移植例です。 「AdStartSampling」 関数を用いて、サンプリング入力を実施した結果を表示します。
14	CFUNC_GPG3100_AdWaitEvent.AJN	C 言語連携機能を使って、GPG-3100の AD ライブラリを呼び出すサンプルプログラムです。 GPG-3100付属の AdWaitEvent.c サンプルの移植例です。 コールバックを使ったサンプリング入力を行います。
15	CFUNC_GPG3100_inputad.AJN	C 言語連携機能を使って、GPG-3100の AD ライブラリを呼び出すサンプルプログラムです。 GPG-3100付属の inputad.c サンプルの移植例です。 「AdInputAD」 関数を用いて、1件のアナログ入力を実施した結果を表示します。
16	CFUNC_GPG3300_DaOutputDA.AJN	C 言語連携機能を使って、GPG-3300の DA ライブラリを呼び出すサンプルプログラムです。 GPG-3300付属の DaOutputDA.c サンプルの移植例です。 「DaOutputDA」 関数を用いて、1件のアナログ出力を実施

#	ファイル名	内容
		した結果を表示します。
17	CFUNC_GPG3300_DaSampling.AJN	C 言語連携機能を使って、GPG-3300の DA ライブラリを呼び出すサンプルプログラムです。 GPG-3300付属の DaSampling.c サンプルの移植例です。 「DaStartSampling」 関数を用いて、連続出力を実施した結果を表示します。
18	CFUNC_GPG3300_DaWaitEvent.AJN	C 言語連携機能を使って、GPG-3300の DA ライブラリを呼び出すサンプルプログラムです。 GPG-3300付属の DaWaitEvent.c サンプルの移植例です。 コールバックを使った連続出力を行います。
19	CFUNC_GPG4116_FrameSendRecv.AJN	C 言語連携機能を使って、GPG-4116の HDLC ライブラリを呼び出すサンプルプログラムです。 GPG-4116付属の FrameSendRecv/sample.c サンプルの移植例です。 送信用のポートからデータを送信し、受信用のポートで受信した結果を表示します。
20	CFUNC_GPG4116_ReceiveMessage.AJN	C 言語連携機能を使って、GPG-4116の HDLC ライブラリを呼び出すサンプルプログラムです。 GPG-4116付属の ReceiveMessage.c サンプルの移植例です。 メッセージを送信します。
21	CFUNC_GPG4116_SendMessage.AJN	C 言語連携機能を使って、GPG-4116の HDLC ライブラリを呼び出すサンプルプログラムです。 GPG-4116付属の SendMessage.c サンプルの移植例です。 メッセージを受信します。
22	CFUNC_GPG4141_fd_receive.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の fd_receive.c サンプルの移植例です。 全二重通信でメッセージを受信します。
23	CFUNC_GPG4141_fd_send.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の fd_send.c サンプルの移植例です。 全二重通信でメッセージを送信します。
24	CFUNC_GPG4141_hd_receive.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の hd_receive.c サンプルの移植例です。 半二重通信でメッセージを受信します。
25	CFUNC_GPG4141_hd_send.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の hd_send.c サンプルの移植例です。 半二重通信でメッセージを受信します。
26	CFUNC_GPG4141_receive.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の receive.c サンプルの移植例です。 メッセージを受信すると終了します。
27	CFUNC_GPG4141_send.AJN	C 言語連携機能を使って、GPG-4141の調歩同期シリアル通信ライブラリを呼び出すサンプルプログラムです。 GPG-4141付属の send.c サンプルの移植例です。 メッセージを送信して終了します。
28	CFUNC_GPG4301_find.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 「PciGpibExExecFindListener」 を呼び出して、デバイス（リスナ）を探索し、見つかったデバイスに対して、IEEE 488.1共通コマンドの「*IDN?（識別問い合わせ）」コマンドを送信して、その受信結果を表示します。

#	ファイル名	内容
		「*IDN?」コマンドを搭載しているデバイスは、デバイスのメーカや型式などの情報が得られます。
29	CFUNC_GPG4301_hp33120a_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent 33120A（ファンクションジェネレータ）を制御します。 Agilent 33120A の『6章 APPLy コマンドの使い方』の事例を再現します。 ・APPLy コマンドと低レベルのコマンドを使ってバーストを設定する方法。 開始位相を270度に設定し、オフセット電圧を追加すると、バーストのある「ハーバサイン」波形が生成されます。
30	CFUNC_GPG4301_hp33120a_B.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent 33120A（ファンクションジェネレータ）を制御します。 Agilent 33120A の『6章 低レベルコマンドの使い方』の事例を再現します。 ・低レベル・コマンドを使って AM 波形を設定する方法 ・*SAV コマンドを使って、機器構成をメモリにストアする方法
31	CFUNC_GPG4301_hp33120a_C.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent 33120A（ファンクションジェネレータ）を制御します。 Agilent 33120A の『6章 HP-IB を介して任意波形をダウンロードする方法』の事例を再現します。 ・HP-IB インタフェースを介して波形ポイントのセットを定義し、揮発性メモリにポイントをダウンロードする方法。 ダウンロードした波形は、計算した立ち上がり時間および立ち下がり時間の方形波パルス（4,000ポイント）です。 ・ダウンロードした波形を不揮発性メモリにコピーする方法。
32	CFUNC_GPG4301_hp34401a.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 GPG-4301付属の hp34401a.c サンプルの移植例です。 Agilent 34401A（マルチメータ）を制御して、電圧値を読み取ります。
33	CFUNC_GPG4301_hp34401a_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent 34401A（マルチメータ）を制御します。 Agilent 34401A の『6章 MEASure?コマンドを使用したシングル測定』の事例を再現します。 ・MEASure?コマンドを使って1回の AC 電流測定を行います。 これは、マルチメータを測定用にプログラムする一番簡単な例です。
34	CFUNC_GPG4301_hp34401a_B.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent 34401A（マルチメータ）を制御します。 Agilent 34401A の『6章 CONFigure コマンドを使用した演算機能』の事例を再現します。 ・CONFigure コマンドを dBm 演算機能で使用します。 CONFigure コマンドは、MEASure?コマンドに比べてプログラミングに多少柔軟性があります。

#	ファイル名	内容
		このため、マルチメータの構成を「段階的」に変更できます。
35	CFUNC_GPG4301_hpe3645a_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Agilent E3645A（プログラマブル DC 電源）を制御します。 Agilent E3645A の『5章 アプリケーション・プログラム』の事例を再現します。 ・Volt コマンドを使って、0.6V から0.8V まで、0.02 ずつ電圧を上げていきます。
36	CFUNC_GPG4301_tk_tds224_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Tektronix TDS224（オシロスコープ）を制御します。 Tektronix TDS224の CH1の波形データを、カンマ区切り形式で読み取ったり、任意ブロック形式で読み取ったりします。
37	CFUNC_GPG4301_tk_tds224_B.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 Tektronix TDS224（オシロスコープ）を制御します。 Tektronix TDS224の「Status and Events」章の「Synchronization」の「Using the *OPC Query」項を元にしています。
38	CFUNC_GPG4301_yk7561.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 GPG-4301付属の yk7561.c サンプルの移植例です。 YOKOGAWA 7561（マルチメータ）を制御して、電圧値を読み取ります。
39	CFUNC_GPG4301_yk7561_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 YOKOGAWA 7561（マルチメータ）を制御します。 YOKOGAWA 7561の『7.4 サンプルプログラム集』の、（4）の事例を再現します。 ・『（4） SINGLE モードでトリガをかけてデータを読む場合』 （AJAN では、GOTO で永久ループせず、10回程度の FOR ループとしています）
40	CFUNC_GPG4301_yk7561_B.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 YOKOGAWA 7561（マルチメータ）を制御します。 YOKOGAWA 7561の『7.4 サンプルプログラム集』の、（5）の事例を再現します。 ・『（5）高速サンプルでデータをストアし、リコールする場合』
41	CFUNC_GPG4301_yk7651_A.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 YOKOGAWA 7651（プログラマブル直流電圧／電流源）を制御します。 YOKOGAWA 7651の『6.4 サンプルプログラム集』の、（1）の事例を再現します。 ・『（1）7651のレンジを10mV とし、+5.0000mV を設定した後、出力を on にする。 また、設定データを読み出し、表示する。』
42	CFUNC_GPG4301_yk7651_B.AJN	C 言語連携機能を使って、GPG-4301の GPIB ライブラリを呼び出すサンプルプログラムです。 YOKOGAWA 7651（プログラマブル直流電圧／電流源）を制御します。

#	ファイル名	内容
		YOKOGAWA 7651の『6.4 サンプルプログラム集』の、(2)の事例を再現します。 ・『(2) 7651のプログラム機能を使って、0V, +5V の方形波を発生させる。 何かキーが押されたら、プログラムを止める。』 (AJAN では、INKEY$ の代わりに INPUT 命令で、Enter入力するまでとしています)
43	CFUNC_GPG4851_canmon.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の canmon.c サンプルの移植例です。 CAN バス上を流れるメッセージを読み取り、画面に表示します。
44	CFUNC_GPG4851_canmonfd.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の canmonfd.c サンプルの移植例です。 CAN FD バス上を流れるメッセージを読み取り、画面に表示します。
45	CFUNC_GPG4851_cyclic.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の cyclic.c サンプルの移植例です。 メッセージの周期送信を行います。
46	CFUNC_GPG4851_receive.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の receive.c サンプルの移植例です。 メッセージの受信を行います。
47	CFUNC_GPG4851_receivefd.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の receive.c サンプルを CAN FD 対応に移植したものです。 CAN FD メッセージの受信を行います。
48	CFUNC_GPG4851_send.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の send.c サンプルの移植例です。 メッセージの送信を行います。
49	CFUNC_GPG4851_sendfd.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の send.c サンプルを CAN FD 対応に移植したものです。 CAN FD メッセージの送信を行います。
50	CFUNC_GPG4851_sendrecvfd.AJN	C 言語連携機能を使って、GPG-4851の CAN 通信ライブラリを呼び出すサンプルプログラムです。 GPG-4851付属の sendrecvfd.c サンプルの移植例です。 CAN FD メッセージの送受信を行います。
51	CFUNC_GPG6204_simplecount.AJN	C 言語連携機能を使って、GPG-6204の3モードカウンタライブラリを呼び出すサンプルプログラムです。 GPG-6204付属の simplecount.c サンプルの移植例です。 位相差パルスをカウントして連続表示します。
52	CFUNC_GPG6320_encoderpulsecount.AJN	C 言語連携機能を使って、GPG-6320のユニバーサルカウンタライブラリを呼び出すサンプルプログラムです。 GPG-6320付属の encoderpulsecount.c サンプルの移植例です。 エンコーダ出力の位相差パルスをカウントして連続表示します。
53	CFUNC_GPG6320_singlepulsecount.A	C 言語連携機能を使って、GPG-6320のユニバーサルカウンタ

#	ファイル名	内容
	JN	ライブラリを呼び出すサンプルプログラムです。 GPG-6320付属の singlepulsecount.c サンプルの移植例です。 単層パルスをカウントします。
54	CFUNC_GPG7400_arc.AJN	C 言語連携機能を使って、GPG-7400のモーションコントローラライブラリを呼び出すサンプルです。 GPG-7400付属の arc.c サンプルの移植例です。 円弧補間動作を行います。 F2キーを押すと、処理を停止します。
55	CFUNC_GPG7400_line.AJN	C 言語連携機能を使って、GPG-7400のモーションコントローラライブラリを呼び出すサンプルです。 GPG-7400付属の line.c サンプルの移植例です。 直線補間動作を行います。 F2キーを押すと、処理を停止します。
56	CFUNC_GPG7400_motion.AJN	C 言語連携機能を使って、GPG-7400のモーションコントローラライブラリを呼び出すサンプルです。 GPG-7400付属の motion.c サンプルの移植例です。 独立動作を行います。 F2キーを押すと、処理を停止します。
57	CFUNC_GPG7400_ptp.AJN	C 言語連携機能を使って、GPG-7400のモーションコントローラライブラリを呼び出すサンプルです。 GPG-7400付属の ptp.c サンプルの移植例です。 PTP 動作を行います。起動→停止までの基本的な処理を行います。
58	CFUNC_GPG7400_setcmd.AJN CFUNC_GPG7400_startcmd.AJN	C 言語連携機能を使って、GPG-7400のモーションコントローラライブラリを呼び出すサンプルです。 GPG-7400付属の setcmd.c および startcmd.c サンプルの移植例です。 setcmd は、コマンドバッファ機能を行います。 startcmd は、コマンドバッファ機能を起動・停止します。

第7章 索引

第8章 重要な情報

保証の内容と制限

弊社は本ドキュメントに含まれるソースプログラムの実行が中断しないこと、またはその実行に誤りが無いことを保証していません。
本製品の品質や使用に起因する、性能に起因するいかなるリスクも使用者が負うものとします。

弊社はドキュメント内の情報の正確さに万全を期しています。万一、誤記または誤植などがあった場合、弊社は予告無く改訂する場合があります。ドキュメントまたはドキュメント内の情報に起因するいかなる損害に対しても弊社は責任を負いません。
ドキュメント内の図や表は説明のためであり、ユーザ個別の応用事例により変化する場合があります。

著作権、知的所有権

弊社は本製品に含まれるおよび本製品に対する権利や知的所有権を保持しています。
本製品はコンピュータ ソフトウェア、映像/音声(例えば図、文章、写真など)を含んでいます。

医療機器/器具への適用における注意

弊社の製品は人命に関わるような状況下で使用される機器に用いられる事を目的として設計、製造された物では有りません。
弊社の製品は人体の検査などに使用するに適する信頼性を確保する事を意図された部品や検査機器と共に設計された物では有りません。
医療機器、治療器具などの本製品の適用により、製品の故障、ユーザ、設計者の過失などにより、損傷/損害を引き起こす場合が有ります。

複製の禁止

弊社の許可なく、本ドキュメントの全て、または一部に関わらず、複製、改変などを行うことはできません。

責任の制限

弊社は､弊社または再販売者の予見の有無にかかわらず発生したいかなる特別損害､偶発的損害、間接的な損害、重大な損害について、責任を負いません。

本製品(ハードウェア，ソフトウェア)のシステム組み込み、使用、ならびに本製品から得られる結果に関する一切のリスクについては、本製品の使用者に帰属するものとします。

本製品に含まれる不都合、あるいは本製品の供給(納期遅延)、性能もしくは使用に起因する付帯的損害もしくは間接的損害に対して、弊社に全面的に責がある場合でも、弊社はその製品に対する改良(有償サービスの利用)、代品交換までとし、製品の予防交換並びに、代金減額等、金銭面での賠償の責任は負わないものとします。

本製品は、日本国内仕様です。

商標/登録商標

本書に掲載されている会社名、製品名は、それぞれ各社の商標または登録商標です。

改訂履歴

Ver.	年　月	改　訂　内　容
0.90	2020年8月	新規作成
0.91	2021年10月	最新情報に更新
1.00	2022年1月	誤記修正。 コマンド追加。CMEMMAP@ GPG-2X72C、GPG-7400 の事例紹介の追加、サンプルの追加。
1.10	2023年3月	GPG-4141, GPG-4851 の事例紹介の追加、サンプルの追加。 コマンド追加。MEMORY.TOSTRTYPE$

このマニュアルは、製品の改良その他により将来予告なく改訂しますので、予めご了承ください。