電気系-情報系間の通信方式について（案）

情報二課

1. 今までの通信方式（リモコン）

今までの通信方式（電気系-リモコン間通信）は、ボーレート（通信速度）115200 のシリアル通信であり、その通信手順は次のようであった。

(1) 電気系から命令要求（同期信号）として何か一文字送信する

(2) リモコンは命令要求を受けて制御信号（現在のボタンの押下状態）を送信する

(3) 電気系は制御信号を受けて実際にアクチュエータを制御する

(4) 始めにもどる

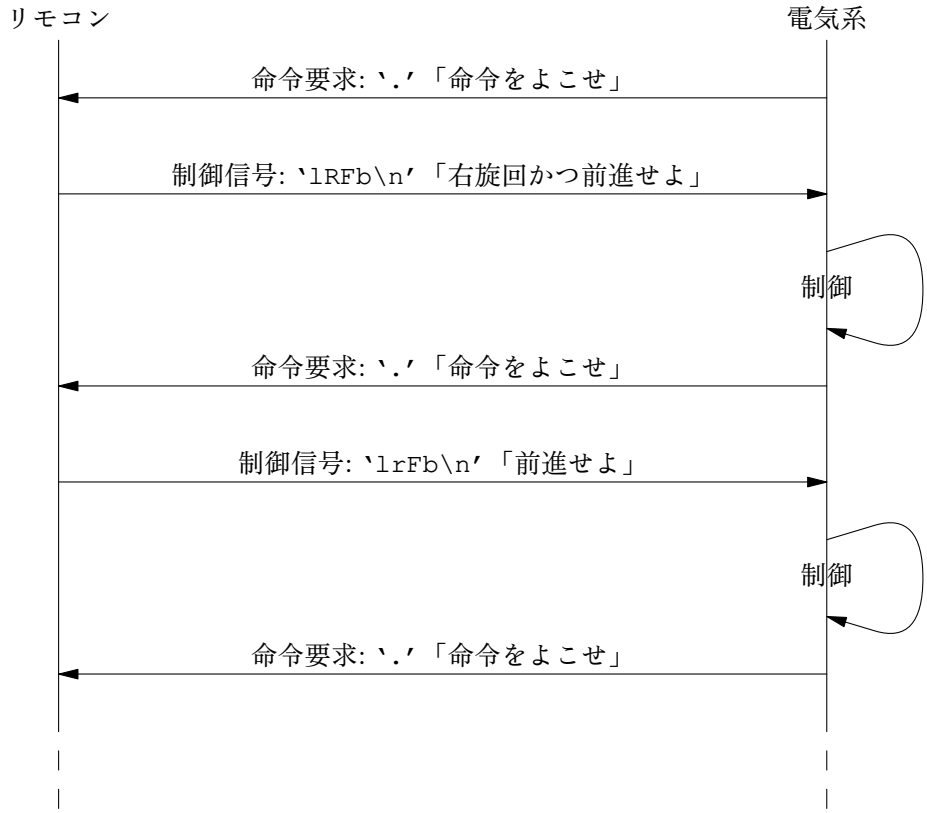

Figure 1: 今までの通信方式

2. 今までの通信方式の問題点

例えば次のようなものが挙げられる。

制御周期の決定権が電気系にある

制御信号は電気系から送信される同期信号を発端に送信される。そのため情報系（リモコン）から自発的に信号を発生させることができない。

ボーレートが速すぎる

単純な実装では容易に同期信号を取りこぼすほか、通信障害によって同期信号が消失した場合に単純な実装では制御不能に陥る。現にリモコン-木箱車間の通信では容易に制御不能になる。もっと低速な通信（ボーレート 9600 程度）でも充分実用に耐え得るだろう。

電気系からバッテリ残量などの付加情報を転送できない

電気系から生成される信号を同期信号として利用しているので、意味のある情報を転送できない。

これらの問題点を克服するために通信方式を改善する。

3. 新しい通信方式（案）

新しい通信方式はボーレート 9600 のシリアル通信であり、次のような通信手順で行う。

(1) 情報系は定期的に制御信号（リクエスト）を送信する

(2) 電気系は制御信号を受けて実際にアクチュエータを制御する

(3) 電気系は制御状態（レスポンス）を送信する

(4) 始めにもどる

さらに、次のような暴走抑制機能を搭載する。

- 情報系は、リクエストを送信すれどレスポンスを得られずに一定時間経過する場合、制御不能状態であると判断し、管理システム（サーバ）にその旨通告する。通告した後にはリクエストとして「制御停止（halt）」を送信する。
- 電気系は、前回のリクエストを受信してから新たなリクエストを受信することなく一定時間経過する場合、制御管理状態から外れたと判断して車体を安全に停止させる。停止した後にリクエストを受信した場合、レスポンスとして「制御停止（halt）」を送信する。

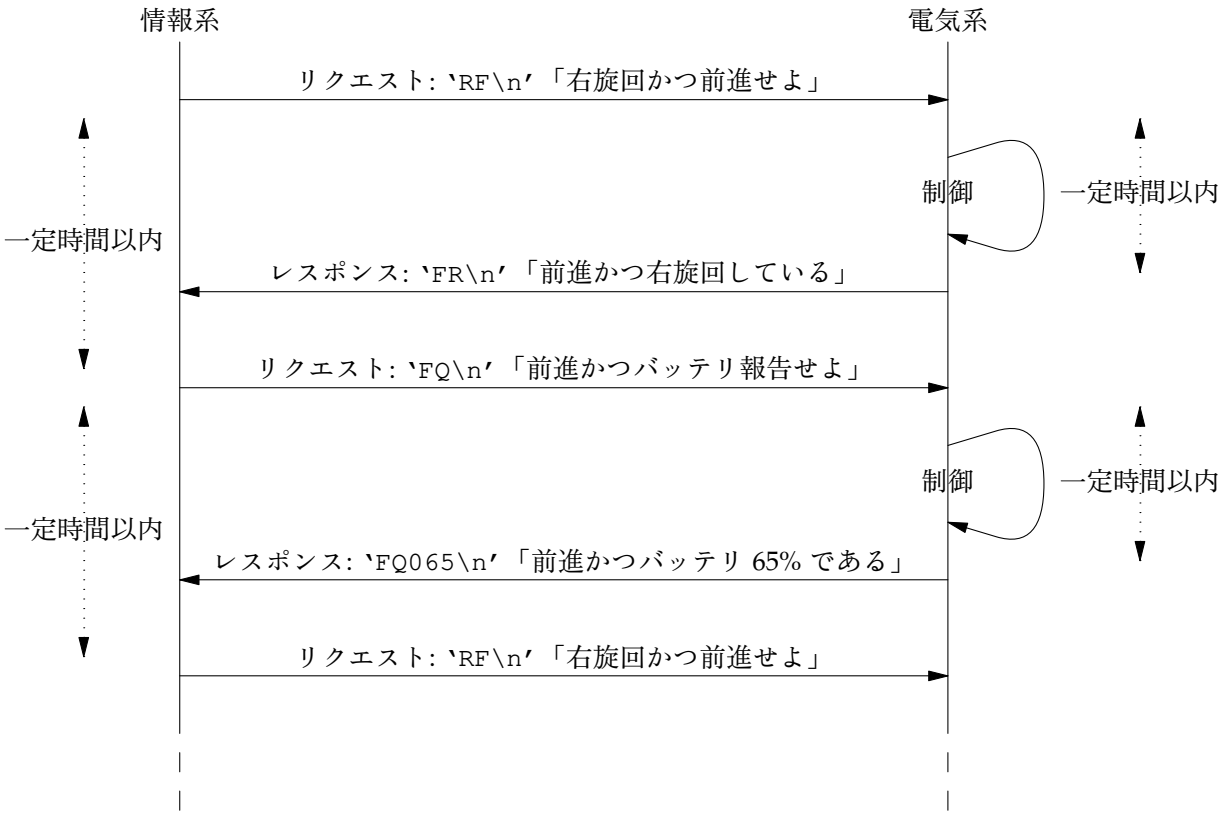

Figure 2: 新しい通信方式

4. リクエスト及びレスポンスの書式

リクエスト及びレスポンス（メッセージ）の書式は次のようにする。

- それぞれのメッセージはデリミタ文字 '\n' (0x0A; NL) で区切られる。
- ひとつのメッセージは 72 文字以内である。ただし、デリミタ文字はこれに含まれない。

4.1. リクエストの書式

形式的な定義では次の通り（たぶんね）:

```
<request>
    : <reqbody> <delimiter>
    ;

<reqbody>
    : <reqbody> <reqmesg>
    | <reqmesg>
    ;

<reqmesg>
    : <battery>
    | <diretion>
    | <halt>
    | <noope>
    | <sleep>
    | <turn>
    ;

<direction>
    : <backward>
    | <forward>
    ;

<turn>
    : <lturn>
    | <rturn>
    ;

<backward>   : 'B' ;      /* 後退せよ */
<battery>    : 'Q' ;      /* バッテリ報告せよ */
<delimiter>  : '\n';      /* デリミタ文字 */
<forward>    : 'F' ;      /* 前進せよ */
<halt>       : 'H' ;      /* 制御停止せよ */
<lturn>      : 'L' ;      /* 左旋回せよ */
<rturn>      : 'R' ;      /* 右旋回せよ */
<sleep>      : 'Z' ;      /* 停車せよ */

<noope>                   /* 意味を持たない */
    : { <backward>、<battery>、<delimiter>、<forward>、
        <halt>、<lturn>、<rturn> 及び <sleep> のいづれでもない文字 }
```

4.2. レスポンスの定義

形式的な定義では次の通り（たぶんね）:

```
<response>
    : <resbody> <selimiter>
    ;

<resbody>
    : <resbody> <resmesg>
    | <resmesg>
    ;

<resmesg>
    : <batteryinfo>
    | <direction>
    | <halt>
    | <noope>
    | <sleep>
    | <turn>
    ;

<batteryinfo>
    | <battery> <percent>
    ;

<percent>
    | <number> <number> <number>
    ;

<direction>
    : <backward>
    | <forward>
    ;

<turn>
    : <lturn>
    | <rturn>
    ;

<backward>   : 'B' ;      /* 後退している */
<battery>    : 'Q' ;      /* バッテリ報告 */
<delimiter>  : '\n';      /* デリミタ文字 */
<forward>    : 'F' ;      /* 前進している */
<halt>       : 'H' ;      /* 制御停止している */
<lturn>      : 'L' ;      /* 左旋回している */
<rturn>      : 'R' ;      /* 右旋回している */
<sleep>      : 'Z' ;      /* 停車している */

<noope>                   /* 意味を持たない */
    : { <backward>、<battery>、<delimiter>、<forward>、<halt>、
        <lturn>、<number>、<rturn> 及び <sleep> のいづれでもない文字 }
<number>
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;
```

4.3. メッセージの意味

それぞれのメッセージ文字は次のような意味を持つ。

<backward>
リクエストならば「後退せよ」、レスポンスならば「後退している」。リクエストに <forward> と同時に出現する場合は互いに打ち消し合う。すなわち前進も後退もしない。レスポンスに <forward> と同時に出現しては**いけない**。

<battery>
リクエストならば「バッテリ残量を報告せよ」、レスポンスならば <batteryinfo> の形式で「バッテリ残量は XXX% である」。

<forward>
リクエストならば「前進せよ」、レスポンスならば「前進している」。リクエストに <backward> と同時に出現する場合は互いに打ち消し合う。すなわち前進も後退もしない。レスポンスに <backward> と同時に出現しては**いけない**。

<halt>
リクエストならば「制御を停止せよ」、レスポンスならば「制御を停止している」。制御停止状態から復帰するには本体のリセットが必要である。

<lturn>
リクエストならば「左旋回せよ」、レスポンスならば「左旋回している」。リクエストに <rturn> と同時に出現する場合は互いに打ち消し合う。すなわち旋回しない。レスポンスに <rturn> と同時に出現しては**いけない**。

<rturn>
リクエストならば「右旋回せよ」、レスポンスならば「右旋回している」。リクエストに <lturn> と同時に出現する場合は互いに打ち消し合う。すなわち旋回しない。レスポンスに <lturn> と同時に出現しては**いけない**。

<sleep>
リクエストならば「停車せよ」、レスポンスならば「停車している」。レスポンスに <backward>、<forward>、 <lturn> あるいは <rturn> と同時に出現しては**いけない**。

<noope>
リクエストとレスポンスのいづれにおいても無視される。開発者が巫山戯るためだけにある。どんなときであっても遊び心を忘れてはいけない。

4.4. メッセージの優先順位

4.4.1. リクエストの優先順位

リクエストの優先順位は次の通りである。メッセージの意味が相反する場合、より上にあるものが勝つ。

1. <halt>
2. <sleep>
3. <battery>
4. <lturn>、<rturn>
5. <backward>、<forward>
6. <noope>

4.4.2. レスポンスの優先順位

レスポンスは状況を説明しているだけであるので全てのメッセージ文字は対等である。

4.5. リクエスト評価の例

リクエストを評価する例を示す。

`'F\n'`
: 前進する。

`'FR\n'`
: 右旋回しながら前進する。

`'RF\n'`
: 上と同じ。メッセージは順不同である。

`'BF\n'`
: 停止する。前進と後退は互いに打ち消し合う。

`'FFFFFFFFFF\n'`
: 前進する。重複するメッセージはひとつにまとめられる。

`'FBFFBFB\n'`
: 停止する。重複するメッセージがまとめられた後に打ち消し合う。

`'RLF\n'`
: 前進する。

`'Z\n'`
: 停止する。

`'\n'`
: 停止する。メッセージが無い場合の規定の動作は停止である。

`'Zzz...\n'`
: 停止する。無効な文字は単に無視される。

`'SLEEP!\n'`
: 左旋回する。

`'BBQ! BBQ!\n'`
: 後退しながらバッテリ報告する。もちろん一回だけ報告すれば良い。

`'BATTERY\n'`
: 右旋回しながら後退する。

`'HZQLRBF\n'`
: 制御停止する。制御停止は他のどんな動作よりも優先する。

5. 実装のヒント

ここでは実装のヒントとなるようなコード片をいくつか紹介する。**ここに紹介しているコード片は正しいと思われるが、実際に動作を試していないので注意すること**。機種依存の処理のいくつかは〈https://blog.goo.ne.jp/lm324/e/c7c8705f1f110a74a251b3abf254024c〉に紹介されている関数を利用している。

5.1. メッセージを受けとる

関数 `getMessage()` はメッセージを引数 `buf` の指す領域に格納する。引数 `buf` が指す領域は caller（呼び出し側）によって適切に 73 バイト以上（80 バイト程度）確保されている必要がある。

関数 `getMessage()` は成功すると 0 を、失敗した場合（例えば、72 バイト以上のメッセージを受信した場合など）に –1 を返す。

```
#define DELIM    '\n'          /* デリミタ文字 */
#define MSGLEN   72            /* メッセージの最大長 */

int
getMessage(char *buf)
{
    int c, i, res;

    res = 0;
    for (i = 0; i < MSGLEN; i++) {
        c = SCI_get();         /* 一文字受信する */
        if (c == DELIM)        /* デリミタ文字（メッセージの終わり）か？ */
            break;             /* 受信終了 */
        buf[i] = c;            /* バッファに格納 */
    }
    buf[i] = '\0';             /* 文字列終端 */
    if (i == MSGLEN) {         /* メッセージが最大長を越えているか？ */
        res = -1;
        while (SCI_get() != DELIM)
            (void)0;           /* デリミタ文字まで読み飛ばす */
    }

    return res;
}
```

5.2. リクエストを解析する

関数 parseRequest() は引数 req で指定されるリクエストを解析する。解析された結果を返り値として返す。

```
#define BACKWARD 'B'
#define BATTERY  'Q'
#define FORWARD  'F'
#define HALT     'H'
#define LTURN    'L'
#define RTURN    'R'
#define SLEEP    'Z'

#define F_BACK   0x01
#define F_FORW   0x02
#define F_LTRN   0x04
#define F_RTRN   0x08
#define F_BTRY   0x10
#define F_SLEP   0x20
#define F_HALT   0x40

int
parseRequest(const char *req)
{
    int reqtab[256];
    int i, ret;

    /* リクエストテーブルを初期化する（本来はハードコードしたい） */
    for (i = 0; i < sizeof(reqtab)/sizeof(reqtab[0]); i++)
        reqtab[i] = 0;
    reqtab[BACKWARD] = F_BACK;
    reqtab[BATTERY]  = F_BTRY;
    reqtab[FORWARD]  = F_FORW;
    reqtab[HALT]     = F_HALT;
    reqtab[LTURN]    = F_LTRN;
    reqtab[RTURN]    = F_RTRN;
    reqtab[SLEEP]    = F_SLEP;

    /* リクエストメッセージを舐めてフラグをセットする */
    ret = 0;
    while (*req)
        ret |= reqtab[*req++];

    return ret;
}
```

5.3. 解析した結果を用いて実際に制御する

関数 drive() は引数 arg に格納された解析結果を用いて実際に車体を制御する。制御する部分のコードを記述することは~~したくない~~できないのでコメントに代えてある。

```
/* 各種 F_* の定義は 5.2. と同様 */

int
drive(int arg)
{
    /* 同時に前進と後退が指示されているなら打ち消す */
    if (arg & F_BACK && arg & F_FORW)
        arg ^= F_BACK | F_FORW;

    /* 同時に左旋回と右旋回が指示されているなら打ち消す */
    if (arg & F_LTRN && arg & F_RTRN)
        arg ^= F_LTRN | F_RTRN;

    /* 優先順位の上の方から順に指示を調べる */
    if (arg & F_HALT) {
        /* 制御停止 */
        return 0;
    }
    if (arg & F_SLEP) {
        /* 停車 */
        return 0;
    }
    if (arg & F_BTRY) {
        /* バッテリ報告 */
    }
    if (arg & F_LTRN) {
        /* 左旋回 */
    }
    if (arg & F_RTRN) {
        /* 右旋回 */
    }
    if (arg & F_BACK) {
        /* 後退 */
    }
    if (arg & F_FORW) {
        /* 前進 */
    }

    return 0;
}
```