RX ファミリ
多倍長演算プログラムの応用例: メルセンヌ素数の計算

R01AN0230JJ0100
Rev.1.00
2011.03.14

要旨

この文書は、RX ファミリの DSP 機能命令の活用した多倍長演算プログラムの例としてメルセンヌ素数の計算方法を説明します。

動作確認デバイス

RX ファミリ

目次

1. はじめに

多倍長演算 (multiple precision arithmetic) とは、コンピュータのハードウェア命令で直接扱うことができる数の範囲を越える精度の数値計算のことです。例えば、RXのような 32 ビットマイクロコンピュータで直接扱うことができる数の範囲は 0 から 2^{32} – 4294967295 に制限されます (符号無し整数として見た場合)。このようなハードウェアの制限を越える数値の計算をしたい場合には多倍長演算を実行するプログラムが必要になります。一般に、多倍長演算はハードウェアの固定精度の計算では精度が不足する場合や、計算結果のオーバフローが問題になるような場合に使用されます。アプリケーションノート「DSP機能命令を活用した多倍長乗算プログラム」(R01AN0226JJ) では、RXのDSP機能命令を活用した多倍長数の四則演算プログラムについて説明しています。以下では多倍長演算プログラムの応用例としてメルセンヌ素数を求めるプログラムについて説明します。

なお、多倍長演算のアルゴリズムの詳細は次に示す文献を参照ください。

【注】 [1] D. E. Knuth, Seminumerical Algorithms, The Art of Computer Programming, Vol. 2, pp.265-284, 3rd edition, Addison Wesley, 1997.

2. 多倍長数のデータ表現

コンピュータのハードウェアが直接扱うことができる範囲を越える数値を多倍長数 (multiple precision number) と呼びます。本章では多倍長数のデータ表現方法について説明します。

多倍長数は符号無し整数とします。RX には 16 ビット ×16 ビットの積和演算命令があり、この積和演算命令を活用して多倍長数の乗算プログラムを実現するので、多倍長数を符号無し 16 ビット整数の配列で表現します。

符号無し 16 ビット整数は 2^{16}個の数を表現できます。配列の各要素を 2^{16}進記法の「数字」と見なすと、長さNの符号無し 16 ビット整数配列によって 2^{16}進記法でN桁の整数を表現できます。なお、N桁の「数字」は配列の中で添字 0 の要素を最下位桁とし、最上位桁に向かって昇順に並んでいるものと約束します。これを図 1 に示します。

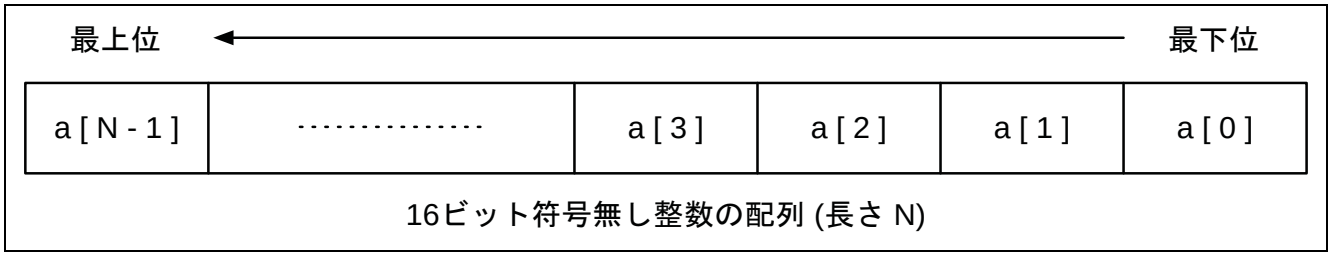

図 1　16 ビット符号無し整数配列による多倍長数のデータ表現

多倍長数 a[N]に格納されている数値は次の式で表わすことができます。

$$a[N-1]\times 2^{16\times(N-1)} + \ldots + a[2]\times 2^{32} + a[1]\times 2^{16} + a[0]$$

図 1 に対応する多倍長数のC言語プログラム例を次に示します。ただし、本アプリケーションノートでは桁数Nを 500 として、最大 $2^{8000}-1$ までの大きさの符号無し整数を扱えるようにします。

```
#include <stdint.h>
#define N      500     /* 多倍長数の長さ (符号無し 16 ビット整数配列)  */
uint16_t a[N];
```

3. メルセンヌ素数

メルセンヌ数 (Mersenne number) とは 2 の冪乗から 1 を減じた形で表わされる数です。

$$M_p = 2^p - 1$$

素数であるメルセンヌ数を特にメルセンヌ素数 (Mersenne prime) と呼びます。$p < 3000$ のメルセンヌ素数としては次のものが知られています。

p = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281

上の数列からも推察されるように、メルセンヌ数が素数である必要条件として、p が素数でなければならないことが知られています。以下では p が素数であるメルセンヌ数のみを考えることにします。

ちなみに、二進法がベースになっているデジタルコンピュータでは、メルセンヌ数の内部表現は連続するビット 1 の並びになります。例えば、p = 13 のメルセンヌ数は二進法で 1111111111111 になります。これは 1 を 13 ビット左にシフトしてから 1 を引いた値です。このようにデジタルコンピュータでは左シフト演算と減算でメルセンヌ数を簡単につくることができます。しかし、メルセンス数が素数であるかを判定することはそれほど簡単ではありません。

3.1 ルーカスの判定法

メルセンヌ数は一般に非常に巨大な整数になるので通常の方法では素数判定が困難です。そこで、メルセンヌ数の素数判定にはルーカスの判定法 (the Lucas-Lehmer primality test) として知られている方法を使います。ルーカスの判定法を次に示します。

まず、奇素数pに対して数列 S_i (i = 0, 1, 2, …) を次のように定めます。

$$S_0 = 4$$
$$S_{i+1} = S_i^{\,2} - 2$$

このときメルセンヌ数M_pが素数であるための必要十分条件は次のとおりです。

$$S_{p-2} \equiv 0 \bmod M_p$$

つまり、S_{p-2}がM_pで割り切れる (剰余が 0) ならばM_pは素数ということになります。

ただし、実際のプログラムではルーカスの判定法を次のようにして使います。まず、奇素数pに対して数列 S_i (i = 0, 1, 2, …) を次のように定めます。

$$S_0 = 4$$
$$S_{i+1} = (S_i^{\,2} - 2) \text{ を } M_p \text{ で割った剰余}$$

このときメルセンヌ数M_pが素数であるための必要十分条件は次のとおりです。

$$S_{p-2} = 0$$

この方法では各項ごとに剰余をとるのでS_iが大きくなりすぎて計算できなくなるようなことがありません。

4. 多倍長演算プログラム

本章では、メルセンヌ素数の計算に使用する多倍長数の演算プログラムを示します。これは、アプリケーションノート「DSP 機能命令を活用した多倍長乗算プログラム」(R01AN0226JJ) で説明されている演算プログラムに少し手を加えて改良したものです。

4.1 ゼロクリアと値のコピー

応用プログラムを読みやすくするために、多倍長数をゼロクリアする関数 long_clr と値のコピーを行う関数 long_cpy を定義します。プログラムを次に示します。

```
#include <string.h>

/*
  多倍長数 a を 0 クリアする
 */
void long_clr(uint16_t *a)
{
   memset(a, 0, sizeof(uint16_t) * N);
}

/*
  多倍長数 b を a へコピーする
 */
void long_cpy(uint16_t *a, uint16_t *b)
{
   memcpy(a, b, sizeof(uint16_t) * N);
}
```

4.2 加算、減算、比較

まず、多倍長数の加算関数 long_add のプログラムを示します。この関数は、多倍長数 a に多倍長数 b の値を加えた結果を a に格納します。なお、この関数はアセンブリ言語で記述されています。そのため#pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の加算
  a に結果を格納する
 */
#pragma inline_asm long_add
void long_add(uint16_t *a, uint16_t *b)
{
       mov.l   #0,r4
       mov.l   #N,r5
  ?:
       movu.w  [r1],r3
       add     r3,r4
       movu.w  [r2+],r3
       add     r3,r4
       mov.w   r4,[r1+]
       shlr    #16,r4
       sub     #1,r5
       bnz     ?-
}
```

次に、多倍長数の減算関数 long_sub のプログラムを示します。この関数は、多倍長数 a から多倍長数 b の値を引いた結果を a に格納します。ただし、a ≥ b でなければなりません。なお、この関数はアセンブリ言語で記述されています。そのため #pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の減算 (ただし、a >= b でなければならない)
  a に結果を格納する
 */
#pragma inline_asm long_sub
void long_sub(uint16_t *a, uint16_t *b)
{
       mov.l   #0,r4
       mov.l   #N,r5
  ?:
       movu.w  [r1],r3
       add     r3,r4
       movu.w  [r2+],r3
       sub     r3,r4
       mov.w   r4,[r1+]
       shar    #16,r4
       sub     #1,r5
       bnz     ?-
}
```

最後に、多倍長数の比較関数 long_cmp のプログラムを次に示します。この関数は、2 つの多倍長数 a と b を比較し、a = b の場合に 0 を、a < b の場合に–1 を、a > b の場合に 1 を返します。

```
/*
  多倍長数の比較
  a > b なら 1、a == b なら 0, a < b なら-1 を返す
 */
int long_cmp(uint16_t *a, uint16_t *b)
{
    int i;
    int32_t c;

    for (i = N - 1; i >= 0; i--) {
        c = (int32_t)a[i] - (int32_t)b[i];
        if (c < 0) {
            return -1;
        }
        if (c > 0) {
            return 1;
        }
    }
    return 0;
}
```

4.3　乗算

多倍長数の乗算を行う関数 long_mul のプログラムを次に示します。この関数は、多倍長数 a と b の乗算を行い、結果を a に格納します。なお、この関数は、補助関数として mul16 と add16 を参照します。なお、関数 mul16 はアセンブリ言語で記述されています。そのため #pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の乗算
  a に結果を格納する
 */
void long_mul(uint16_t *a, uint16_t *b)
{
    int i, j;
    uint32_t x;
    static uint16_t res[N];

    memset(res, 0, sizeof res);
    for (i = 0; i < N; i++) {
        if (a[i] != 0) {
            for (j = 0; j < N; j++) {
                if (b[j] != 0 && i + j < N) {
                    x = mul16(a[i], b[j]);
                    add16(res, i + j, (x & 0xffff));
                    add16(res, i + j + 1, (x >> 16));
                }
            }
        }
    }
    memcpy(a, res, sizeof res);
}

/*
  符号無し 16 ビット整数の乗算
  結果は符号無し 32 ビット整数で返す
 */
#pragma inline_asm mul16
static uint32_t mul16(uint16_t a, uint16_t b)
{
       push.l  r6
       mov.l   r1,r3
       and     #7fffh,r3
       mov.l   r2,r4
       and     #7fffh,r4
       mov.l   #0,r5
       tst     #8000h,r1
       bz      ?+
       mov.l   r4,r6
       shll    #15,r6
       add     r6,r5
  ?:
       tst     #8000h,r2
       bz      ?+
       mov.l   r3,r6
       shll    #15,r6
       add     r6,r5
       tst     #8000h,r1
       bz      ?+
       add     #40000000h,r5
```

```
  ?:
        mullo   r3,r4
        mvfacmi r1
        add     r5,r1
        pop     r6
}

/*
  符号無し 16 ビット整数 b を多倍長数 a の i 桁目に加算する
 */
static void add16(uint16_t *a, int i, uint16_t b)
{
    uint32_t c;

    for (c = b ; c > 0 && i < N; i++) {
        c += a[i];
        a[i] = c;  // c の下位 16 ビットのみ転送される
        c >>= 16;  // c の上位 16 ビットには桁上りの値が入っている
    }
}
```

4.4　除算

多倍長数の除算関数 long_div のプログラムを次に示します。この関数は、多倍長数 a の値を多倍長数 b の値で割り算をして、商を q、剰余を r に格納します。ただし、b > 0 でなければなりません。この関数は、補助関数として guess、lshl、lshr および llen を参照します。なお、関数 guess はアセンブリ言語で記述されているため #pragma inline_asm 宣言を使用します。

```
/*
  多倍長数の除算 (ただし、b > 0 でなければならない)
  q に商を、r に剰余をそれぞれ返す
 */
void long_div(uint16_t *a, uint16_t *b, uint16_t *q, uint16_t *r)
{
    int i, m, n, shift;
    uint32_t u, quot;
    static uint16_t c[N], d[N], e[N];

#define ZERO(x)       memset(x, 0, sizeof(uint16_t) * N)
#define COPY(x, y)     memcpy(x, y, sizeof(uint16_t) * N)

    /* initialize */
    ZERO(e);
    ZERO(q);
    COPY(r, a);
    n = llen(b) - 1;
    if (long_cmp(a, b) < 0 || n < 0) {
        return;
    }
    /* normalize */
    for (shift = 0, u = b[n]; (u & 0x8000) == 0; u <<= 1) {
        shift++;
    }
    lshl(r, shift);
    lshl(b, shift);
    /* loop */
    while (long_cmp(r, b) >= 0) {
```

```
        m = llen(r) - 1;
        if (r[m] >= b[n]) {
            ZERO(c);
            for (i = 0; i <= n; i++) { c[m - n + i] = b[i]; }
            if (long_cmp(r, c) >= 0) {
                q[m - n] = 1;
                long_sub(r, c);
                continue;
            }
        }
        quot = guess(r, b, m, n);
        ZERO(c);
        for (i = 0; i <= n; i++) { c[m - n - 1 + i] = b[i]; }
        COPY(d, c);
        e[0] = quot;
        long_mul(c, e);
        while (long_cmp(r, c) < 0) {
            long_sub(c, d);
            quot--;
        }
        q[m - n - 1] = quot;
        long_sub(r, c);
    }
    /* unnormalize */
    lshr(r, shift);
    lshr(b, shift);

#undef  ZERO
#undef  COPY
}

/* long_div の補助関数 */
#pragma inline_asm guess
static uint32_t guess(uint16_t *a, uint16_t *b, int c, int d)
{
        shll    #01h,r3,r5
        add     r1,r5
        movu.w  [r5],r1
        sub     #02h,r5
        shll    #10h,r1
        add     [r5].uw,r1
        movu.w  [r4,r2],r5
        divu    r5,r1
        cmp     #0ffffh,r1
        bleu    ?+
        mov.l   #0ffffh,r1
  ?:
}

/*
  多倍長数 a を左へ n ビットシフト
  ただし、0 <= n <= 15 でなければならない
 */
static void lshl(uint16_t *a, int n)
{
    int i;
    uint32_t c = 0;
    uint32_t t;
```

```
    if (n == 0) {
        return;
    }
    for (i = 0; i < N; i++) {
        t = (uint32_t)a[i];
        t <<= n;
        t |= c;
        a[i] = t;
        c = (t >> 16);
    }
}

/*
  多倍長数 a を右へ n ビットシフト
  ただし、0 <= n <= 15 でなければならない
 */
static void lshr(uint16_t *a, int n)
{
    int i;
    uint16_t c = 0;
    uint16_t t;

    if (n == 0) {
        return;
    }
    for (i = N - 1; i >= 0; i--) {
        t = a[i];
        a[i] = (c | (t >> n));
        c = (t << (16 - n));
    }
}

/*
  多倍長数の桁数を返す
  ただし、a == 0 のときは 0 を返す
 */
static int llen(uint16_t *a)
{
    int i;

    for (i = N - 1; i >= 0; i--) {
        if (a[i] != 0) {
            return i + 1;
        }
    }
    return 0;
}
```

4.5　ビットシフト

多倍長数をビット単位で左または右へシフトする関数 long_shl と long_shr のプログラムを次に示します。これらの関数は、ビット単位で左または右に n ビットだけ多倍長数 a をシフトします。シフトするビット数 n には 15 以上の値を指定することができます。これらの関数は前節で説明した除算プログラムの補助関数 lshl, lshr, llen を参照します。

```
/*
  多倍長数 a を左へ n ビットシフト
 */
void long_shl(uint16_t *a, int n)
{
    int i, j, k;
    static uint16_t t[N];

    if (n <= 15) {
        lshl(a, n);
    }
    else {
        j = n / 16;     /* 左へシフトする桁数 */
        n -= j * 16;    /* 残りのビットシフト数 */
        k = llen(a);    /* a の桁数 */
        memset(t, 0, sizeof t);
        for (i = 0; i < k && i + j < N; i++) {
            t[i + j] = a[i];
        }
        lshl(t, n);
        memcpy(a, t, sizeof t);
    }
}

/*
  多倍長数 a を右へ n ビットシフト
 */
void long_shr(uint16_t *a, int n)
{
    int i, j, k;
    static uint16_t t[N];

    if (n <= 15) {
        lshr(a, n);
    }
    else {
        j = n / 16;     /* 右へシフトする桁数 */
        n -= j * 16;    /* 残りのビットシフト数 */
        k = llen(a);    /* a の桁数 */
        memset(t, 0, sizeof t);
        for (i = k - 1; i - j >= 0; i--) {
            t[i - j] = a[i];
        }
        lshr(t, n);
        memcpy(a, t, sizeof t);
    }
}
```

5. メルセンヌ素数の計算

本章では、メルセンヌ素数を求めるプログラムを示します。メインプログラム (main関数) は、5 から順番に 3000 より小さい奇素数pに対応するメルセンヌ数 $(2^p - 1)$ が素数であるかをルーカスの判定法で調べます。

メインプログラムは divisor、next_prime、print_mersenne_prime の 3 つの補助関数を参照します。

divisor は、整数 n の最小の約数を返す関数です。divisor は、3 から sqrt(n)までの大きさの奇数で順番に n を割ってみて n の最小の約数 (1 以外) を探します。

next_prime 関数は、指定された整数 n より大きい最小の素数を返します。next_prime は divisor を呼び出して、1 以外の最小の約数がその数に等しいことにより素数を判定します。

print_mersenne_prime は、メルセンヌ素数が見つかった場合にメインプログラムから呼び出される関数です。統合開発環境 (High-performance Embedded Workshop) でプログラムを動かす場合、この関数にブレークポイントを設定することにより求めたメルセンヌ素数を確認できます。

```
#include <stdint.h>
#include <math.h>
#include "RXlong.h"

/* n の最小の約数を返す */
static int divisor(n)
{
    int i, root;

    root = (int)sqrt(n);
    for (i = 3; i <= root; i += 2) {
        if (n % i == 0) {
            return i;
        }
    }
    return n;
}

/* n 以上の最小の素数を返す */
static int next_prime(int n)
{
    if (n <= 3) {
        return 3;
    }
    if (n % 2 == 0) {
        n += 1;  /* 奇素数 */
    }
    while (divisor(n) != n) {
        n += 2;
    }
    return n;
}

/* メルセンヌ素数 p の表示 */
static void print_mersenne_prime(int p)
{
    /* printf("mersenne prime: p = %d¥n", p); */
}

void main(void)
{
    int p, i;
```

```
    static uint16_t M[N], S[N], a[N], q[N];

    /*
       5 から昇順に 3000 を越えない奇素数 p に対して、ルーカスの判定法 (the
       Lucas-Lehmer primality test) を使って(2 ^ p - 1)がメルセンヌ素数
       であるかを判定する
    */
    for (p = 5; p < 3000; p = next_prime(p + 2)) {
        /* M <-- 2 ^ p - 1 */
        long_clr(M);
        M[0] = 1;
        long_shl(M, p);
        long_clr(a);
        a[0] = 1;
        long_sub(M, a);

        /* S <-- 4 */
        long_clr(S);
        S[0] = 4;
        for (i = 1; i < n - 1; i++) {
            /* S <-- S ^ 2 - 2 */
            long_cpy(a, S);
            long_mul(S, a);
            long_clr(a);
            a[0] = 2;
            long_sub(S, a);
            /* S <-- S mod M */
            long_div(S, M, q, a);
            long_cpy(S, a);
        }
        if (long_nil(S)) {
            /* S == 0 ならメルセンヌ素数である */
            print_mersenne_prime(p);
        }
    }
}
```

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/inquiry

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2011.03.14	—	初版発行

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. リザーブアドレス（予約領域）のアクセス禁止
   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. クロックについて
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. 製品間の相違について
   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。
   同じグループのマイコンでも型名が違うと、内部 ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

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

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

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

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

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

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

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル） (03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

Colophon 1.0