Introduction

This manual presents a library of ARM assembly source code modules for digital signal processing (DSP) applications such as infinite impulse response (IIR) filter, finite impulse response (FIR) filter and fast Fourier transform (FFT) applicable for a range of DSP applications including VSLP vocoder. These assembly source code modules are presented for ARM mode and have been tested in an ARM9E-based STR91x platform.

In addition, the assembly source code modules have been tested in an IAR Workbench environment as well, but STMicroelectronics cannot guarantee that these assembly source code modules will be flawless for all applications.

The algorithm modules are presented "as is with no warranty".
# Contents

## 1 Definitions and related documents

1.1 Acronyms and terminology ................................................. 4
1.2 ARM and Thumb ................................................................. 4
1.3 References ........................................................................ 5

## 2 IIR ARMA 16-bit filter ......................................................... 6

2.1 Description ......................................................................... 6
2.2 Function ............................................................................. 6
2.3 Arguments and variables ..................................................... 6
  2.3.1 Calling the function from C ............................................... 6
  2.3.2 Calling the function from assembly .................................... 6
2.4 Algorithm ........................................................................... 7
2.5 Assembly code ..................................................................... 7
2.6 Requirements ..................................................................... 7
2.7 Implementation .................................................................... 8
2.8 Benchmarking ..................................................................... 8

## 3 Block FIR 16-bit filter ........................................................... 9

3.1 Description ......................................................................... 9
3.2 Function ............................................................................. 9
3.3 Arguments and variables ..................................................... 9
  3.3.1 Calling the function from C ............................................... 9
  3.3.2 Calling the function from Assembly .................................. 10
3.4 Algorithm ........................................................................... 10
3.5 Assembly code ..................................................................... 10
3.6 Requirements ..................................................................... 11
3.7 Implementation .................................................................... 11
3.8 Benchmarking ..................................................................... 11

## 4 Complex 16-bit radix-4 FFT .................................................... 12

4.1 Description ......................................................................... 12
4.2 Algorithm ........................................................................... 12
1 Definitions and related documents

1.1 Acronyms and terminology

Table 1. Definition of acronyms and terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>ARM Core</td>
</tr>
<tr>
<td>ARMA</td>
<td>Auto Regressive Moving Average</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DSPLIB</td>
<td>Digital Signal Processing Library</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>LTI</td>
<td>Linear Time-Invariant</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>STR91x</td>
<td>STR91x family of MCUs from STMicroelectronics</td>
</tr>
<tr>
<td>VSELP</td>
<td>Vector-Sum Excited Linear Prediction</td>
</tr>
</tbody>
</table>

1.2 ARM and Thumb

The Thumb set consists of 16-bit instructions that act as a compact, shorthand subset of the 32-bit ARM instructions. Every Thumb instruction could be executed via an equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset. For example, there's no way to access status or coprocessor registers in Thumb. Also, some functions that can be accomplished in a single ARM instruction can only be accomplished with a sequence of Thumb instructions.

Thumb compatible processor can operate in ARM or Thumb state. Some method is needed to switch the processor from executing instructions in one state to executing in the other. This is provided by the Branch Exchange instruction, versions of which exist both in the ARM and Thumb instruction sets. Both of these perform a branch by copying the contents of general register Rn into the program counter causing a pipeline flush and refill from address specified in Rn. Thus, BX is absolute rather than PC-relative.

In ARM state the format is:

```
BX(<cond>) Rn
```

In Thumb state the format is:

```
BX Rn
```

All ARM instructions are word-aligned, and all Thumb instructions are half-word aligned. Therefore, the least significant bit in Rn can always be considered to be zero. The
processor can actually use this bit to determine if the instruction jumped to should be executed in Thumb or ARM state:

- If bit 0 set then execute in Thumb state
- If bit 0 clear then execute in ARM state

Thumb was defined for two main reasons:

1. Better code density, as the instructions are half the size of ARM instructions (although some ARM instructions require two Thumb instructions for the same effect). You would have to compile the application for ARM and Thumb and see what gives the best result.

2. Better performance from narrow memory, as instruction fetches from smaller memory (ie: 8-bit or 16-bit) will be reduced in Thumb mode.

1.3 References


2 IIR ARMA 16-bit filter

2.1 Description

The IIR filter function performs an ARMA filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients respectively for ny output samples. This is a 16-bit fixed-point implementation. Reference [1] may be used to explore more about IIR filters.

2.2 Function

void iirarma_arm9e(void *y, void *x, short *h2, short *h1, int ny);

2.3 Arguments and variables

- Output array vector y[ny+4] (used in actual computation. first four elements must have the previous outputs. Thus the first output starts y[4] when ny = 0)
- Input array vector x[ny+4]
- Moving-average filter coefficients vector h2[5]
- Auto-regressive filter coefficients h1[5] and h1[0] are not used
- Number of output samples ny is a multiple of 4 and must be ≥ 8

2.3.1 Calling the function from C

Set the arguments and variables as appropriate in C main program and call the function. In the STR91x DSPLIB this is set in the 91x_dsp.h header file.

2.3.2 Calling the function from assembly

Use the following registers equivalent to arguments/variables.

- R0 to the address of output vector y[]
- R1 to the address of input vector x[]
- R2 to the address of coefficient vector h2[]
- R3 to the address of coefficient vector h1[]
- R12 to the address of ny

Then use the BL instruction to call the function.
2.4 Algorithm

Following is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

C equivalent IIR ARMA algorithm

```c
iir_arma_arm9e(short *y, short *x, short *h2, short *h1, int ny)
{
    int i, j;
    int sum;

    for (i=0; i<ny; i++){
        sum = h2[0] * x[4+i];
        for (j = 1; j <= 4; j++)
            sum += h2[j]*x[4+i-j]-h1[j]*y[4+i-j];
        y[4+i] = (sum >> 15);
    }
}
```

2.5 Assembly code

The assembly code for this IIR ARMA algorithm is validated in STR91x environment for ARM966E-S. The assembly code is for 16-bit fixed point applications. The assembly source code module may be found in DSPLIB.

2.6 Requirements

The assembly module is little endian and written for ARM9E in ARM mode with stack aligned to 8 bytes.

The module is for 16-bit fixed point applications where input data is expected to be 16-bit fixed point and it will produce 16-bit fixed-point output.

A caller needs to organize an input vector x[] for filter input and output vector y[] for filter output then supply the filter coefficients h1[] as well as h2[]. The vector length for x[] and y[] needs to follow the following guidelines.

- ny is multiple of 4 and greater than or equal to 8
- Input data array x[] contains ny + 4 input samples to produce ny output samples.
2.7 Implementation

- The output vector y[] contains ny + 4 locations, but its first 4 data are not used, i.e., first output is y[4] while ny = 0.
- The inner loop that iterated through the filter coefficients is completely unrolled.
- The code is little ENDIAN.
- The code is interrupt-tolerant but not interruptible.

2.8 Benchmarking

The benchmarking for this filter function is done using Chameleon Trace while the filter function is running in STR91x engine with ARM966E-S. For the Chameleon Trace in ETM Configuration Trace Port Size is set to 8 bit and Trace Port Mode is set to Normal, Half-rate clocking, while the STR91x engine runs at 96 MHz.

Table 2. Benchmarking of IIR ARMA algorithm module

<table>
<thead>
<tr>
<th>Number of taps</th>
<th>Cycle count</th>
<th>Microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>6544</td>
<td>68.167</td>
</tr>
<tr>
<td>48</td>
<td>4396</td>
<td>45.792</td>
</tr>
<tr>
<td>24</td>
<td>2248</td>
<td>23.417</td>
</tr>
</tbody>
</table>
3 Block FIR 16-bit filter

3.1 Description

This function computes a direct-form real FIR filter using the coefficients stored in vector \( h[] \) using a simple Block FIR technique, and moves delay line. The input sequence needs to be start with \( (T-1) \) zeros. This is a 16-bit fixed-point implementation. Reference [1] may be used to explore more about FIR filters.

The user needs to organize the output buffer for \( y[N] \) and needs to design the filter taps \( h[T] \) and organize the coefficients in reverse order. The users also need to scale the input \( x[M] \) and filter taps \( h[T] \) to avoid overflow.

This function may be called by C or Assembly function(s).

3.2 Function

```c
typedef struct coef{
    short *h;
    unsigned int m;
}nh;

int fir_16by16_arm9e(int y[], short x[], sx *p, int N);
```

```c
nh p;
p.h = h;
p.M = T;
```

3.3 Arguments and variables

- \( h[T] \) = filter coefficient vector with \( T \) number of taps, an integer multiple of 6
- \( T \) = number of filter coefficients (taps), an integer multiple of 6
- \( N \) = filter length or vector length for output
- \( x[M] \) = filter input vector with total \( M \) samples, i.e., \( M = N + T - 1 \)
- \( y[N] \) = filter output vector with total \( N \) samples

3.3.1 Calling the function from C

Set the arguments and variables as appropriate in C main program and call the function.
3.3.2 Calling the function from Assembly

Use the following registers equivalent to arguments/variables.
● Move N to R3, N is output filter length or output vector length, a multiple of 6
● Move SP to R2, comparable &p as in C function above h[T] = filter coefficient vector with T number of taps, T is an integer multiple of 6
● Set R1 to the address of x, comparable as in x[M] filter input vector with total M samples, i.e., \( M = N + T - 1 \)
● Set R0 to the address of y, comparable as in y[N] filter output vector with total N samples

Then use BL instruction to call the function.

3.4 Algorithm

Following is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

**Block FIR Filter Reference C Source**

```c
void fir_16by16_arm9e(short input, int nt, short h[], short z[])
{
  int j;
  short acc;
  /* store input at the beginning of the delay line */
  z[0] = input;
  /* calc FIR */
  acc = 0;
  for (j = 0; j < nt; j++) {
    acc += h[j] * z[j];
  }
  /* shift delay line */
  for (j = nt - 2; j >= 0; j--) {
    z[j + 1] = z[j];
  }
  return acc;
}
```

3.5 Assembly code

The assembly code for this block FIR 16-bit fixed-point filter algorithm is validated in STR91x environment for ARM966E-S. The assembly code is for 16-bit fixed point applications, and
the filter coefficients are a multiple of 6 - more information is available in the source code. The source code module may be found in DSPLIB.

3.6 Requirements

A caller needs to organize an input vectors say x[] and output vector say y[] for filter. The caller needs to design the filter coefficients using any of the windowing functions such as Hamming, Hanning, and Blackman etc as required by the filter response.

Then upon designing the coefficients, reverse the order of the coefficients. The number of coefficients, T, must be an integer multiple of 6.

3.7 Implementation

- The filter needs T number of coefficients and T is multiple of 6
- The output vector y[N] is such that N equals (2 * T)
- The input vector x[M] is such that M equals (N + T - 1)
- The code is little ENDIAN
- The code is interrupt-tolerant but not interruptible

3.8 Benchmarking

The benchmarking for this filter function is done using Chameleon Trace while the filter function is running in STR91x engine with ARM966E-S. For the Chameleon Trace in ETM Configuration Trace Port Size is set to 8 bit and Trace Port Mode is set to Normal, Half-rate clocking, while the STR91x engine runs at 96 MHz.

<table>
<thead>
<tr>
<th>Table 3. Benchmarking of Block FIR algorithm module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of taps</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>24</td>
</tr>
</tbody>
</table>
4 Complex 16-bit radix-4 FFT

4.1 Description

The DFT $X[k]$ of a complex sequence $x[n]$ of length $N$ is calculated by:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \exp(-j2\pi nk/N) \text{ for } k=0, 1, 2, \ldots, N-1$$

A direct calculation of $N$ complex values of $X[k]$ will require $4N^2$ multiplications and $4N(N-1)$ additions given the trigonometric function values. For example, if $N=128$, 65536 multiplications and 65024 additions are required.

In this report, a radix-4 FFT algorithm is implemented. More details about FFT algorithms including radix-4 are found in references [3] [4] and [5].

4.2 Algorithm

The algorithm works as follows:

**Block 1: Bit reverse**

Copy the data from the input buffer to the output buffer in bit-reversed order. It is an integer between 0 and $N-1$, with binary representation as shown, for the bit reversal of $k$, we will write. So we perform the equivalent of the following loop:

```c
int k, khat, bit;
for (k=0, khat=0; k<N; k++) {
    X[khat]=x[k];
    for (bit=N/2; (khat & bit)!=0; bit >>=1) khat ^= bit;
    khat ^= bit; /* finish incrementing khat */
}
```

The bit loop inside increments khat as a bit reversed number. If the input buffers and output buffers are equal ($X=x$), more care needs to be taken to do the action in place, but it can be performed faster as fewer elements need to be moved. Putting the elements in bit reversed order has the effect of grouping together all the Ys and Zs of the previous subsection so no further rearrangement needs to be done.
Block 2: Kernel

Perform the equivalent of the following loop:
- denotes complex multiplication

```c
#define pi 3.14159265358

for (n=2; n<=N; n <<= 1) {
    w=2*pi/n;
    for (m=0; m<N; m+=n) {
        for (k=0; k<n/2; k++) {
            y=X[m+k];
            z=X[m+k+n/2] * exp(-1j*w);
            X[m+k]=(y+z)/2;
            X[m+k+n/2]=(y-z)/2;
        }
    }
}
```

After n=2 we have performed N/2 two-element FFTs, positioned at offsets m=0, 2, 4, ..., N-2. After n=4 we have performed N/4 four-element FFTs positioned at offset m=0, 4, 8, ..., N-4. After n=N we are left with the answer to the main FFT in the buffer X.

```latex
k \ bN 1 - \ldots \ b2
b1
b0
(\ldots \ b2

Because each pass of the FFT divides the data by 4 (radix4 passes) there is no bit growth.

Four cases were tested as shown in Table 4.

<table>
<thead>
<tr>
<th>Case name</th>
<th>Description</th>
<th>Input amplitude</th>
<th>Output bin amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single sine max neg.</td>
<td>Single sine at 1600; Fs =25600; 1 complete waveform in 16 samples;</td>
<td>0x8001</td>
<td>2 bins of 0xC000 (+/- 0x2)</td>
</tr>
<tr>
<td>Single sine max pos.</td>
<td>Single sine at 1600; Fs =25600; 1 complete waveform in 16 samples;</td>
<td>0x7FFF</td>
<td>2 bins of 0x4000 (+/- 0x2)</td>
</tr>
</tbody>
</table>
### 4.3 Arguments and variables

The algorithm takes three values on input:

- A pointer `x` to the input data consisting of `N` complex numbers `x[0], ..., x[N-1]`. Each complex number is a 32-bit integer containing the 16-bit real part followed by the 16-bit imaginary part.
- A pointer `y` to the output buffer for the transformed array to be stored. This buffer is the same in size as the input buffer `x`, with same order as in input buffer for the real part as well as the imaginary part.
- An integer `nBin` giving the base 2 logarithm of the number of points `N` in the Fourier Transform.

### 4.4 Function

```c
extern void cr4fft1k16_arm9e(void *pssOUT, void *pssIN, int nBin);
```

The input to the function is an array of complex data with each even index the real part and imaginary part being in odd index, e.g., `x[0]` real part and its imaginary part `x[1]`; `x[2]` real part and its imaginary part `x[3]`; and so on ... Similarly, the output is also in the same order.

The block outline of this FFT algorithm is shown in Table 5.

### Table 5. Block outline of FFT routine

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_context</td>
<td>Save all registers used by the routine</td>
<td></td>
</tr>
<tr>
<td>d_main</td>
<td>Main loop; transfer 16 bytes per iteration</td>
<td>single loop</td>
</tr>
<tr>
<td>d_swit</td>
<td>Second level switch: transfer 4, 8 or 12 bytes</td>
<td>linear routine</td>
</tr>
<tr>
<td>d_switch</td>
<td>Third level switch: transfer 1, 2 or 3 bytes</td>
<td>switch case</td>
</tr>
<tr>
<td>restore_context</td>
<td>Restore all registers</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.1 Calling the FFT function from C

To call the FFT routine from C, include both `91x_dsp.c` and `91x_dsp.h` which declares the type structure and the function FFT as:

```c
extern void cr4fft1k16_arm9e(void *pssOUT, void *pssIN, int nBin);
```
Then to call the function from main use the following in main program as global:

```c
struct complex_fft *test_fft = NULL;
```

and call the function using

```c
test_fft->calc_cr4fft_1k(y, x, N);
```

The function can also be called directly using simple function call by main program. Note the following points:

- The buffers x and y may coincide. In this case the FFT is said to be done in place rather than out of place.
- To prevent overflow within the algorithm, the real and imaginary values in the array x should be sign extended 16-bit quantities (between -32768 and +32767).
- The values of N allowed are 64, 256, 1024.

### 4.4.2 Calling the FFT function from assembly

To call the FFT routine from assembler, import the symbol FFT and set up the registers as follows:

- R0 to the address of the input buffer x
- R1 to the address of the output buffer y
- R2 to N

Call the FFT routine using BL FFT. On exit R0 will contain the exit code (0 if successful), R1-R3 and R14 will have been corrupted and R4-R13 preserved.

### 4.5 The FFT function characteristics

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Complex FFT</td>
<td></td>
</tr>
<tr>
<td>Algorithm choices</td>
<td>Radix4, DIT, Bit reverse in input, Not in place</td>
<td></td>
</tr>
<tr>
<td>Number of points</td>
<td>64 or 256 or 1024</td>
<td></td>
</tr>
<tr>
<td>Input data size</td>
<td>2 x 16-bit</td>
<td></td>
</tr>
<tr>
<td>Input data organization</td>
<td>imag_real (little endian) such as real = address imag =address +2</td>
<td></td>
</tr>
<tr>
<td>Input data order</td>
<td>Linear (bit reverse is done during the first pass; this is transparent to the user)</td>
<td></td>
</tr>
<tr>
<td>Computation radix</td>
<td>Radix 4 (Burrus Parks)</td>
<td></td>
</tr>
<tr>
<td>Overflow prevention</td>
<td>for every pass (radix 4) the data is right shifted by 2 so that the resulting value does not overflow;</td>
<td>see normalized note</td>
</tr>
<tr>
<td>Saturate on overflow</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Rounding</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Output data size</td>
<td>2 x 16-bit</td>
<td></td>
</tr>
</tbody>
</table>
The STR91x implementation of the bit reversal stage is straightforward and contained between the labels FFT and FFTSTART. The second stage begins at the label FFTSTART.

4.6 Performance benchmarking

We start by listing the performance for the speed optimized algorithm. Timings are given for the STR91x running at 96 MHz. The code size for the optimized algorithm is 1592+ N/2 bytes (including the lookup-table) and the data size is 64 bytes (not including the input and output buffers).

<table>
<thead>
<tr>
<th>Radix-4 complex FFT</th>
<th>Operation mode</th>
<th>Cycle count</th>
<th>Microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Point</td>
<td>Program in Flash &amp; Data in SRAM</td>
<td>2701</td>
<td>28.135</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in SRAM</td>
<td>3432</td>
<td>35.750</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in Flash</td>
<td>3705</td>
<td>38.594</td>
</tr>
<tr>
<td>256 Point</td>
<td>Program in Flash &amp; Data in SRAM</td>
<td>13740</td>
<td>143.125</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in SRAM</td>
<td>18079</td>
<td>188.323</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in Flash</td>
<td>19908</td>
<td>207.375</td>
</tr>
<tr>
<td>1024 Point</td>
<td>Program in Flash &amp; Data in SRAM</td>
<td>68534</td>
<td>713.896</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in SRAM</td>
<td>90422</td>
<td>941.896</td>
</tr>
<tr>
<td></td>
<td>Program &amp; Data in Flash</td>
<td>101151</td>
<td>1053.656</td>
</tr>
</tbody>
</table>

All these results were obtained on a STR910-EVAL best case configuration for speed. The clock is 96 MHz; the number of wait states is minimum for Flash and for SRAM. The code is run from Flash.
### 4.7 Fixed-point error benchmarking

Table 8 shows fixed-point error analysis of FFT function (ARM966E-S) with respect to floating-point and fixed-point C function, and amplitude, for a 16-point FFT.

**Table 8. The FFT function fixed-point error benchmarking**

<table>
<thead>
<tr>
<th>ARM966E-S</th>
<th>Real Part</th>
<th>Imaginary Part</th>
<th>Amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM966E-S</td>
<td>Floating point</td>
<td>ARM966E-S</td>
</tr>
<tr>
<td>-5552</td>
<td>-5550</td>
<td>-5552</td>
<td>-5550</td>
</tr>
<tr>
<td>21318</td>
<td>21319</td>
<td>-5841</td>
<td>-5840</td>
</tr>
<tr>
<td>6297</td>
<td>6298</td>
<td>-1503</td>
<td>-1502</td>
</tr>
<tr>
<td>4100</td>
<td>4100</td>
<td>-402</td>
<td>-401</td>
</tr>
<tr>
<td>3146</td>
<td>3146</td>
<td>208</td>
<td>208</td>
</tr>
<tr>
<td>2573</td>
<td>2574</td>
<td>632</td>
<td>631</td>
</tr>
<tr>
<td>2165</td>
<td>2165</td>
<td>967</td>
<td>967</td>
</tr>
<tr>
<td>1836</td>
<td>1836</td>
<td>1262</td>
<td>1262</td>
</tr>
<tr>
<td>1543</td>
<td>1544</td>
<td>1543</td>
<td>1544</td>
</tr>
<tr>
<td>1261</td>
<td>1262</td>
<td>1836</td>
<td>1836</td>
</tr>
<tr>
<td>966</td>
<td>967</td>
<td>2164</td>
<td>2165</td>
</tr>
<tr>
<td>631</td>
<td>631</td>
<td>2574</td>
<td>2574</td>
</tr>
<tr>
<td>208</td>
<td>208</td>
<td>3146</td>
<td>3146</td>
</tr>
<tr>
<td>-402</td>
<td>-401</td>
<td>4101</td>
<td>4100</td>
</tr>
<tr>
<td>-1501</td>
<td>-1502</td>
<td>6298</td>
<td>6298</td>
</tr>
<tr>
<td>-5840</td>
<td>-5840</td>
<td>21320</td>
<td>21319</td>
</tr>
</tbody>
</table>
5 Revision history

Table 9. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18-Jan-2007</td>
<td>1</td>
<td>Initial release</td>
</tr>
<tr>
<td>22-Jan-2007</td>
<td>2</td>
<td>Added references 3, 4 and 5 in Section 1.3 on page 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Section 4.1 on page 12</td>
</tr>
<tr>
<td>09-Jun-2008</td>
<td>3</td>
<td>Removed references to obsolete products.</td>
</tr>
</tbody>
</table>
Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale. Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS “AUTOMOTIVE GRADE” MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2008 STMicroelectronics - All rights reserved

STMicroelectronics group of companies
Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

19/19