Freescale Semiconductor
Application Note

AN1952
Rev. 0, 9/2005

Using Program Memory As Data Memory

William Jiang

Contents

1. Introduction

Microcontrollers with Harvard architecture have separate program and data memory spaces. Program memory is generally used for storing program code, although it can be used for storing data; while, as its name indicates, Data memory is used for storing data. Because the CodeWarrior tool for the 56800 Family does not directly support access to Program memory from C code, using assembly code is the only way to access Program memory. To speed a product's time to market, this document includes a Program Memory Functions Library, which can be used for accessing data of various types stored in Program Memory. It is simple, easy to understand, and easy to use. For quick reference, source code is included in Section C in the Appendix.

2. 56800 Program Memory and Data Memory

Program memory and Data memory in the 56800 Family are known as P memory and X memory, respectively. Both memory spaces are 16 bits wide; i.e., one word wide. Program memory is normally used for storing program code, i.e. instructions. Data memory is used for storing program data. There is only one kind of instruction for Program memory access: MOVE(M) P:(Rj)+,HHHH. In contrast, many kinds of instructions are used to access Data memory, such as MOVE instructions, bit-manipulation instructions and so on. A detailed comparison between both memory usages is shown in **Table 2-1**.

According to this table, 56800 hybrid controllers provide far less Data memory than Program memory. A user can benefit from larger Program memory to store data when the data memory space is insufficient. For example, a user can store various constant coefficients in the Program Flash. The disadvantage of using Program memory is obvious: the access time is longer.

freescale semiconductor

Table 2-1. Usage Comparison between Program Memory and Data Memory

	Program Memory	Data Memory
How Many Instruction Types?	1	Many
Parallel Move Allowed?	No	Yes
Machine Cycles for a MOVE instruction	≥8	≥2 but < its counterpart for Program memory
56F801	9K (1K RAM/ 8K Flash)	3K (1K RAM/2K Flash)
56F802	9K (1K RAM/ 8K Flash)	3K (1K RAM/2K Flash)
56F803	32.5K (512 RAM/ 32K Flash)	6K (2K RAM/4K Flash)
56F805	32.5K (512 RAM/ 32K Flash)	6K (2K RAM/4K Flash)
56F807	62K (2K RAM/ 60K Flash)	10K (2K RAM/8K Flash)
56F826	32.5K (512 RAM/ 32K Flash)	6K (4K RAM/2K Flash)
56F827	65K (1K RAM/ 64K Flash)	8K (4K RAM/4K Flash)

3. Program Memory Functions Library

This section provides the Program memory function APIs which can be called by C code to access Program memory. Program memory is limited to the Program RAM in the case of memory write functions. For writing into the Program Flash/ROM, the relevant Flash/ROM algorithm should be used. These functions support various data types: bool; char; unsigned char; short; unsigned short; int; unsigned int; long; unsigned long; float; double; and pointer. A description of each function's performance follows.

3.1 Performance of the APIs

The performance numbers that follow are calculated under these conditions:

Table 3-2. Performance of the APIs

Program Memory Functions	Oscillator Cycles	Instructions Words
pmemreads	18	2
pmemwrites	18	2
pmemreadf	28	4
pmemwritef	28	4
pmemreadl	28	4

Table 3-2. Performance of the APIs (Continued)

Program Memory Functions	Oscillator Cycles	Instructions Words
pmemwritel	28	4
pmemreadns	16+N*10	5
pmemwritens	16+N*10	5
pmemreadnf	16+N*20	7
pmemwritenf	16+N*20	7

- The functions run from the on-chip program memory
- “N” stands for the number of data in words to be read or written

When the functions are executed from external Program memory which requires wait states, the execution will be slower.

3.2 Requirements of the APIs

Since there are no boundary checks by the Program memory functions, the user must ensure all addresses passed to the function APIs are within the scope of the device memory. The program memory functions do not use any stack space to pass parameters and return values.The stack space required for execution of a program memory function is only one word.

3.3 *pmemreads*

This function reads a short value from the Program memory at the specified address. It can also be used to read other types of data: unsigned short; bool; char; unsigned char; int; and unsigned int, since they are represented in 16 bits in memory by CodeWarrior.

Prototype:

```
short pmemreads(void *address);
```

Input:

address A pointer to the location in P memory to be read

Return:

A short value

3.4 *pmemwrites*

This function writes a short value to the Program RAM at the specified address. It can also be used to write other types of data: unsigned short; bool; char; unsigned char; int; and unsigned int, since they are represented in 16 bits in memory by CodeWarrior.

Prototype:

```
void pmemwrites(void *address,short data);
```

Input:

address A pointer to the location in P RAM to write

data A short value to be written

Return:

None

3.5 *pmemreadf*

This function reads a float value from the Program memory at the specified address.

Prototype:

```
float pmemreadf(void *address);
```

Input:

address	A pointer to the location in P memory to be read

Return:

A float value

3.6 *pmemwritef*

This function reads a float value to the Program RAM at the specified address.

Prototype:

```
void pmemwritef(void *address,float data);
```

Input:

address	A pointer to the location in P RAM where the value is to be written
data	A float value to be written

Return:

None

3.7 *pmemreadl*

This function reads a long value from the Program memory at the specified address.

Prototype:

```
long pmemreadl(void *address);
```

Input:

address	A pointer to the location in P memory where data are to be read

Return:

A long value

3.8 *pmemwritel*

This function writes a long value to the Program RAM at the specified address.

Prototype:

```
void pmemwritel(void *address,long data);
```

Input:

address	A pointer to the location in P RAM where the value is to be written
data	A long value to be written

Return:

None

3.9 *pmemreadns*

This function reads an array of short values with *nelem* elements from the Program memory at *addressFrom* and stores these data into Data memory at *addressTo*. It can also be used to read other types of data: unsigned short; bool; char; unsigned char; int; and unsigned int since they are represented in 16 bits in memory by CodeWarrior.

Prototype:

```
short pmemreadns(void *addressFrom,void *addressTo,unsigned short
                  nelem);
```

Input: ,

addressFrom	A pointer to the source location in P memory from which data are to be read
addressTo	A pointer to the destination location in X memory where data are to be written
nelem	Indicates the number of elements in the short array where data are to be stored

Return:

A short value last read

3.10 *pmemwritens*

This function writes an array of short values with *nelem* elements to the program RAM memory at *addressTo* from the Data memory at *addressFrom*. It can also be used to write other types of data: unsigned short; bool; char; unsigned char; int; and unsigned int, since they are represented in 16 bits in memory by CodeWarrior.

Prototype:

```
void pmemwritens(void * addressFrom,void * addressTo,unsigned short
                  nelem);
```

Input:

addressFrom	A pointer to the source location in X memory from which data are to be read
addressTo	A pointer to the destination location in P memory where data are to be written
nelem	Indicates the number of elements in the short array to be written

Return:

None

3.11 *pmemreadnf*

This function reads an array of 32-bit values with *nelem* elements from the Program RAM memory at *addressFrom* and stores these data into Data memory at *addressTo*. It can be used to read an array of long values or float values.

Prototype:

```
short pmemreadnf(void *addressFrom,void *addressTo,unsigned short
                  nelem);
```

Input:

addressFrom	A pointer to the location in P memory from which data are to be read
addressTo	A pointer to the location in X memory where data are to be written
nelem	Indicates the number of elements in the short array where data are to be received

Return:

A short value last read

3.12 *pmemwritenf*

This function writes an array of 32-bit values with *nelem* elements to the Program RAM memory at *addressTo* from the Data memory at *addressTo*. It can be used to read an array of long values or float values.

Prototype:

```
void pmemwritenf(void * addressFrom,void * addressTo,unsigned short
                  nelem);
```

Input:

addressFrom	A pointer to the location in X memory from which data are to be read
addressTo	A pointer to the location in P memory where data are to be written
nelem	Indicates the number of elements in the short array to be written

Return:

None

4. How to Use the Program Memory Functions Library

There are some tips and tricks for using the Program Memory Functions Library. Here is an example of storing constant variables in the program memory on the 56F827. The framework of the sample code is generated with Freescale's Embedded Stationery. The global constant variables defined in *main.c* will be stored in the external Program RAM at load time. This can be achieved by setting the compiler to emit const defined data to the *.rodata* section and by using the linker command file as shown. The *.ApplicationCode* section contains code lines similar to these:

```
# rodata: Constants to be placed in Program memory
      _Prodata_start_addr_in_RAM = .;
      F_Prodata_start_addr_in_RAM = .;
      main.c (.rodata)
      F_Prodata_size = . - _Prodata_start_addr_in_RAM;
```

These code lines direct the linker to put the global constant variables defined in *main.c* into the Program RAM. After the code is loaded by the CodeWarrior debugger, the Program RAM contains these global constants.

However, take care when using strings in CodeWarrior.

4.1 Tips and Tricks

Without the keyword "*const*" just before the variable name *ErrorMessages*, the following definition results in that data will be stored in *.data* section instead of in the *.rodata* section.

```
const char * ErrorMessages [] =
{
"IO ERROR",
   "TRBLE MODE ERR",
   "DATA ERROR",
   "DATA CORRUPTED",
   "DISPLAY ERROR",
   "MEMORY ERROR",
   "INVALID PARAMETER"
};
```

In the code above, there are constant strings which will be stored in Data memory instead of in Program memory, since CodeWarrior puts those strings in the *.data* section. To direct CodeWarrior to store constant strings in the *.rodata* section and then in the Program memory, check "*Make Strings ReadOnly*" in the Code Generation\M56800 Processor panel. In general, the user can refer to the linker map file for the Program and Data memory usages.

4.2 Linker Command File for 56F827 External RAM

```
# Linker.cmd file for DSP56827EVM External RAM
#        using both internal and external data memory (EX = 0)
#    and using external program memory (Mode = 3)
#*************************************************************************
MEMORY {
  .pInterruptVector   (RX)  : ORIGIN = 0x0000, LENGTH = 0x0086
.pExtRAM            (RWX) : ORIGIN = 0x0086, LENGTH = 0xFF7A
.xAvailable         (RW)  : ORIGIN = 0x0000, LENGTH = 0x0030
.xCWRegisters       (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
.xIntRAM_DynamicMem (RW)  : ORIGIN = 0x0040, LENGTH = 0x0FC0
.xPeripherals       (RW)  : ORIGIN = 0x1000, LENGTH = 0x0400
.xReserved1         (R)   : ORIGIN = 0x1400, LENGTH = 0x0C00
.xFlash             (R)   : ORIGIN = 0x2000, LENGTH = 0x1000
.xReserved2         (R)   : ORIGIN = 0x3000, LENGTH = 0x1000
.xExtRAM            (RW)  : ORIGIN = 0x4000, LENGTH = 0xA000
.xExtRAM_DynamicMem (RW)  : ORIGIN = 0xE000, LENGTH = 0x1200
.xStack             (RW)  : ORIGIN = 0xF200, LENGTH = 0x0D80
.xCoreRegisters     (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0080
}
#*************************************************************************
FORCE_ACTIVE {FconfigInterruptVector}
SECTIONS {
#
# Data (X) Memory Layout
#
 _EX_BIT     = 0;
 # Internal Memory Partitions (for mem.h partitions)
 _NUM_IM_PARTITIONS = 1;  # IM_ADDR_1 (no IM_ADDR_2 )
 # External Memory Partition (for mem.h partitions)
 _NUM_EM_PARTITIONS = 1;   # EM_ADDR_1

#*************************************************************************
.ApplicationInterruptVector :
{
 vector.c (.text)

} > .pInterruptVector
#*************************************************************************
.ApplicationCode :
{
```

```
     # Place all code into External Program RAM
 * (.text)
 * (rtlib.text)
 * (fp_engine.text)
 * (user.text)

 # data to be placed into Program RAM

 F_Pdata_start_addr_in_ROM = 0;
 F_Pdata_start_addr_in_RAM = .;
     pramdata.c (.data)

    # rodata: Constants to be placed in Program memory

 _Prodata_start_addr_in_RAM = .;
 F_Prodata_start_addr_in_RAM = .;
 main.c (.rodata)
 F_Prodata_size = . - _Prodata_start_addr_in_RAM;

 F_Pdata_ROMtoRAM_length = 0;
  F_Pbss_start_addr = .;
 _P_BSS_ADDR = .;
 pramdata.c (.bss)
 F_Pbss_length = . - _P_BSS_ADDR;
F_Pram_end_addr = .;
} > .pExtRAM

#*****************************************************************************
.ApplicationData :
{
 # Define variables for C initialization code

 F_Xdata_start_addr_in_ROM = ADDR(.xFlash) + SIZEOF(.xFlash) / 2;
 F_StackAddr               = ADDR(.xStack);
 F_StackEndAddr            = ADDR(.xStack) + SIZEOF(.xStack) / 2  - 1;
 F_Xdata_start_addr_in_RAM = .;

 # Define variables for mem library
 FmemEXbit = .;
    WRITEH(_EX_BIT);
 FmemNumIMpartitions = .;
    WRITEH(_NUM_IM_PARTITIONS);
 FmemNumEMpartitions = .;
    WRITEH(_NUM_EM_PARTITIONS);
 FmemIMpartitionList = .;
    WRITEH(ADDR(.xIntRAM_DynamicMem));
    WRITEH(SIZEOF(.xIntRAM_DynamicMem) / 2);
```

```
 FmemEMpartitionList = .;
    WRITEH(ADDR(.xExtRAM_DynamicMem));
    WRITEH(SIZEOF(.xExtRAM_DynamicMem) /2);

 # Place rest of the data into External RAM

 * (.data)
 * (fp_state.data)
 * (rtlib.data)

 F_Xdata_ROMtoRAM_length = 0;

 F_Xbss_start_addr = .;
 _X_BSS_ADDR = .;

  * (rtlib.bss.lo)
 * (.bss)

 F_Xbss_length = . - _X_BSS_ADDR;  # Copy DATA

} > .xExtRAM
#*****************************************************************************
FArchIO   = ADDR(.xPeripherals);
FArchCore = ADDR(.xCoreRegisters);
}
```

4.3 *main.c*

```
/* File: main.c */
#include "pmemop.h"

/*******************************************************
* Skeleton C main program for use with Embedded Software
*******************************************************/
#define assert(a)\
if(!a)\
{   \
 asm(debug)\
}

extern void * _Prodata_start_addr_in_RAM;
extern void * _Prodata_size;
extern void * _Pram_end_addr;

#pragma use_rodata on

const short ShortArray [3][3] =
{ {0x4000, 0x5000, 0x6000},
```

```
 {0x7000, 0x8000, 0x9000},
 {0xa000, 0xb000, 0xc000}
};

const float FloatArray [3][3] =
{ {0.5, 1.0, 2.5},
 {3.0, 3.25, 4.0},
 {0.25, 4.25, 5.5}
};

const long LongArray[3][3] =
{
{0x55aaff00,0xff0055aa,0x0055ffaa},
{0x12345678,0x23456789,0x3456789a},
{0x456789ab,0x56789abc,0x6789abcd}
};

/*
 * Without 'const' just before the variable name,the following defintion results
in that
 * data will be stored in .data section instead of .rodata section
 * This is testified by the linker map file.
 * Please check "Make Strings ReadOnly" in the Code Generation\M56800 Processor
panel.
 */
const char * const ErrorMessages [] =
{ "IO ERROR",
 "TRBLE MODE ERR",
 "DATA ERROR",
 "DATA CORRUPTED",
 "DISPLAY ERROR",
 "MEMORY ERROR",
 "INVALID PARAMETER"
};
#pragma use_rodata off


void main (void)
{
int i,j;
shorts[3][3];           // stores the array of short values
floatf[3][3];           // stores the array of float values
longl[3][3];            // stores the array of long values
char*msg_pointer[7];// stores the pointers to the strings
charmsg[50];            // stores the actual string
longldata = 0x66885588L;
```

```
/*
 * CASE 1: test pmemreads to read a single short value and pmemreadf to read a
single float value and
 *      pmemreadl to read a long value.
 */
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    s[i][j] = pmemreads((void*)&ShortArray[i][j]);// read short values from p
memory at &ShortArray [i][j]
    f[i][j] = pmemreadf((void*)&FloatArray[i][j]);// read float values from p
memory at &FloatArray [i][j]
    l[i][j] = pmemreadl((void*)&LongArray[i][j]);// read long values from p
memory at &LongArray [i] [j]
 }
}
/*
 * CASE 2: test pmemreadns to read an array of short values.
 */
pmemreadns((void*)ErrorMessages,msg_pointer,7);// read pointers to the desired
strings
for (i = 0; i < 7;i++)
{
 pmemreadns(msg_pointer[i],msg,20);// check msg to see whether it contains the
desired string
}

/*
 * CASE 3: test pmemwritens to write an array of short values.
 */
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    s[i][j] = (i+1)*j; // initialize the short array
 }
}

pmemwritens((void*)s,(void*)ShortArray,9);// write 9 shorts in the program memory
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    s[i][j] = 0; // reset the short array
 }
}
pmemreadns((void*)ShortArray,(void*)s,9);
```

```
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    assert(s[i][j] == (i+1)*j); // check the returned value
 }
}
/*
 * CASE 4: test pmemwritenf to write an array of float values.
 */
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    f[i][j] = (i+10.0)*j; // initialize the short array
 }
}

pmemwritenf(f,(void*)FloatArray,9);// write 9 shorts in the program memory
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    f[i][j] = 0.0;           // reset the short array
 }
}
pmemreadnf((void*)FloatArray,(void*)f,9);

for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    assert(f[i][j] == (i+10.0)*j);// check the returned value
 }
}
/*
 * CASE 5: test pmemwritenf to write an array of long values.
 */
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    l[i][j] = (i+0x20000000L)*j+0x10000000L; // initialize the long array
 }
}
```

```
pmemwritenf((void*)l,(void*)LongArray,9);// write 9 long values to the program
memory
for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    l[i][j] = 0; // reset the short array
 }
}
pmemreadnf((void*)LongArray,(void*)l,9);

for (i = 0; i < 3; i++)
{
 for(j = 0; j < 3; j++)
 {
    assert(l[i][j] == (i+0x20000000L)*j+0x10000000L); // check the returned value
 }
}
/*
 * CASE 6: test pmemwritel to write a long value
 */
ldata = 0x66885588L;
pmemwritel((void*)&_Pram_end_addr,ldata);
ldata = pmemreadl((void*)&_Pram_end_addr);
assert(ldata == 0x66885588L);

/*
 * CASE 7: test pmemwrites to write a short value
 */
pmemwrites((void*)&_Pram_end_addr,0x1234);
s[0][0]= pmemreads((void*)&_Pram_end_addr);
assert(s[0][0]==0x1234);
/*
 * CASE 8: test pmemwritefto write a float value
 */
pmemwritef((void*)&_Pram_end_addr,123.456);
f[0][0]= pmemreadf((void*)&_Pram_end_addr);
assert(f[0][0]==123.456);
}
```

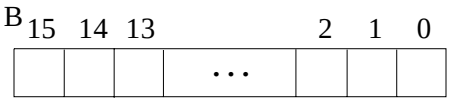
APPENDIX

Before reading the source code of the Program Memory Functions Library, a user must understand CodeWarrior Data Representation and the C Calling Conventions, which are shown here for reference.

A. CodeWarrior Data Representation

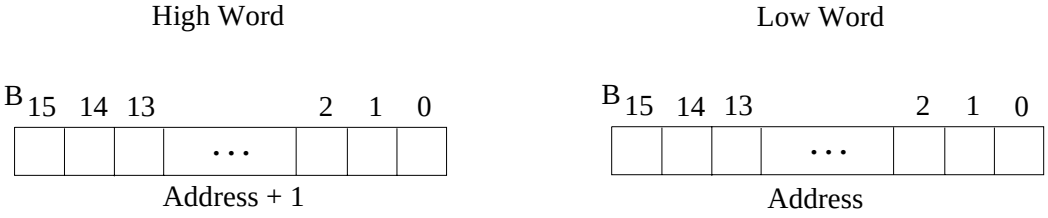
A.1 Bool; Char; Short; Int; and Pointer Types

These types are represented in one word; i.e., 16 bits, as shown below:



A.2 Long; Float and Double Types

These types are represented in two words in Little Endian Mode: the high address word stores high data word, and the low address word stores low data word.



B. CodeWarrior Calling Conventions

CodeWarrior for Freescale 56800 may store data and calls functions differently from other target platforms.

- Registers A, R2, R3, Y0 and Y1 pass parameters to functions
- When a function is called, the parameter list is scanned from left to right.

The parameters are passed in this way:

1. The first long fixed-point value is placed in A
2. The first two 16-bit fixed-point values are placed in Y0 and Y1
3. The first two 16-bit addresses are placed in R2 and R3
4. If there are no long fixed-point parameters, the first non-fixed-point 32-bit value or 19-bit address is placed in A.
5. If there are no 16-bit fixed-point parameters, the first two 16-bit non-fixed-point, non-address values are placed in Y0 and Y1 (and Y0 receives the single value when only one is passed)

6. All remaining parameters are pushed onto the stack, beginning with the right-most parameter. Multiple-word parameters (19- and 32-bit values) have their least significant word pushed onto the stack first.

When calling a routine that returns a structure, the caller passes an address in R0, which specifies where to copy the structure.

The registers A, R0, R2, and Y0 are used to return function results as follows:

- Long fixed-point values are returned in A
- All 19-bit addresses and 32-bit values are returned in A
- 16-bit addresses are returned in R2
- All 16-bit non-address values are returned in Y0

C. Source Code of Program Memory Functions Library

C.1 *pmemop.h*

```
#ifndef__pmemop_h
#define __pmemop_h
short pmemreads(void *address);
long pmemreadl(void *address);
float pmemreadf(void *address);
short pmemreadns(void *addressFrom,void *addressTo,unsigned short nelem);
short pmemreadnf(void *addressFrom,void *addressTo,unsigned short nelem);
void pmemwrites(void *address,short data);
void pmemwritel(void *address,long data);
void pmemwritef(void *address,float data);
void pmemwritens(void * addressFrom,void * addressTo,unsigned short nelem);
void pmemwritenf(void * addressFrom,void * addressTo,unsigned short nelem);
#endif
```

C.2 *pmemop.c*

```
#include "pmemop.h"

/*
 * Function: pmemreads
 * Description: reads a short value from the program memory at address
 * Input:
 * address pointer to the location in p memory to be read from
 * Return:
 *short value
 */

asm short pmemreads(void *address)
{
movem P:(R2)+,y0
rts
}
```

```
/*
 * Function: pmemreadns
 * Description: reads an array of short values with nelem elements from the
program memory at addressFrom and
 *              stores these data into data memory at addressTo.
 * Input:
 * addressFrom pointer to the location in p memory to be read
 *addressTo pointer to the location in x memory where data are to be written
 *nelem         indicates the number of elements in the short array to be read
 * Return:
 *short value last read
 */

asm short pmemreadns(void * addressFrom,void * addressTo,unsigned short nelem)
{
do Y0,endr
 movem P:(R2)+,y0//addressFrom
 move  y0,x:(R3)+//addressTo
endr:
rts
}

/*
 * Function: pmemreadf
 * Description: reads a float value from the program memory at address
 * Input:
 * address pointer to the location in p memory to be read
 * Return:
 *float value
 */
asm float pmemreadf(void *address)
{
movem P:(R2)+,x0
movem P:(R2)+,a
move  x0,a0
rts
}

/*
 * Function: pmemreadnf
 * Description: reads an array of float values with nelem elements from the
program memory at addressFrom and
 *              stores these data into data memory at addressTo.
 * Input:
 * addressFrom pointer to the location in p memory to be read from
 *addressTo pointer to the location in x memory where data are to be written
 *nelem         indicates the number of elements in the short array to be read
 * Return:
 *short value last read
 */
```

```
asm short pmemreadnf(void * addressFrom,void * addressTo,unsigned short nelem)
{
do Y0,endr
 movem P:(R2)+,y0//addressFrom : low word
 move  y0,x:(R3)+//addressTo
 movem P:(R2)+,y0//addressFrom : high word
 move  y0,x:(R3)+//addressTo
endr:
rts
}

/*
 * Function: pmemreadl
 * Description: reads a long value from the program memory at address
 * Input:
 * address pointer to the location in p memory to be read
 * Return:
 *long value
 */
asm long pmemreadl(void *address)
{
movem P:(R2)+,x0
movem P:(R2)+,a
move  x0,a0
rts
}

/*
 * Function: pmemwrites
 * Description: writes a short value to the program memory at address
 * Input:
 * address pointer to the location in p memory to be written to
  * data         a short value to be written;
 * Return:
 *none
 */

asm void pmemwrites(void *address,short data)
{
move y0,P:(R2)+
rts
}

/*
 * Function: pmemwritens
 * Description: writes an array of short values with nelem elements to the program
memory at addressTo
 *              from the data memory at addressFrom
 * Input:
 * addressFrom pointer to the location in x memory to be read
 *addressTo pointer to the location in p memory where data are to be written
```

```
 *nelem          indicates the number of elements in the short array to be read
 * Return:
 *none
 */

asm void pmemwritens(void * addressFrom,void * addressTo,unsigned short nelem)
{
do Y0,endr
  move  X:(R2)+,y0//addressFrom
  move  y0,P:(R3)+//addressTo
endr:
rts
}

/*
 * Function: pmemwritel
 * Description: writes a long value to the program memory at address
 * Input:
 * address pointer to the location in p memory to be written to
  * data          a long value to be written;
 * Return:
 *none
 */

asm void pmemwritel(void *address,long data)
{
move a0,x0
move x0,P:(R2)+
move a1,P:(R2)+
rts
}
/*
* Function: pmemwritef
* Description: writes a float value to the program memory at address
* Input:
* address pointer to the location in p memory to be writen to;
* data a float value to be writen;
* Return:
* none
*/
asm void pmemwritef(void *address,float data)
{
move a0,x0
move x0,P:(R2)+
move a1,P:(R2)+
rts
}

/*
* Function: pmemwritenf
* Description: writes an array of float values with nelem elements to the program
memory at addressTo from
* the data memory at addressTo.
```

```
* Input:
* addressFrom pointer to the location in x memory to be read
* addressTo pointer to the location in p memory where data are to be writen
* nelem indicates the number of elements in the short array to be read
* Return:
* none
*/
asm void pmemwritenf(void * addressFrom,void * addressTo,unsigned short nelem)
{
do Y0,endr
move X:(R2)+,y0 //addressFrom : low word
move y0,P:(R3)+ //addressTo
move X:(R2)+,y0 //addressFrom : high word
move y0,P:(R3)+ //addressTo
endr:
rts
}
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN1952

Rev. 0
9/2005