

# Motorola Semiconductor Application Note

---

## AN1803

## C Coding for the Keypad Module of the MMC2001

By Glenn Jackson  
Austin, Texas

### Introduction

---

This application note assists the microcontroller developer to quickly set up the functionality of the keypad module on the MMC2001. The MMC2001 is the first publicly available general-purpose M•CORE™ offering in the M•CORE product line.

An overview of keypad design issues is explained and the software modular structure is described. Suggestions concerning future enhancements to the current software are made and detailed software for keypad decoding is listed and described.

The order of topics in this application note is:

- Keypad design issues
- Software structure
- Steps to decode the keypad
- Additional enhancements to this keypad structure
- Program listings

---

M•CORE is a trademark of Motorola, Inc.



### Keypad Design Issues Overview

Keypad operations produce several issues. These include:

- Bounce — Capturing the key depress in real time
- Unique keypad outputs — Differing configurations, matching various columns and rows
- C pseudo-code — Rising above assembler code for register bit control with C type variables
- Polling versus interrupt methods

These topics cover concerns which may not be immediately apparent for anyone who has not dealt previously with keypad issues.

### *Debounce*

The attempt to press a key on a physical keypad and have this press detected can fail as a result of several sources. These sources include:

- The high speed of sampling in the part, relative to real time
- Contacts which prove to be intermittent
- The ability of the hardware to detect an attempt at a key press versus a random spike in the circuitry

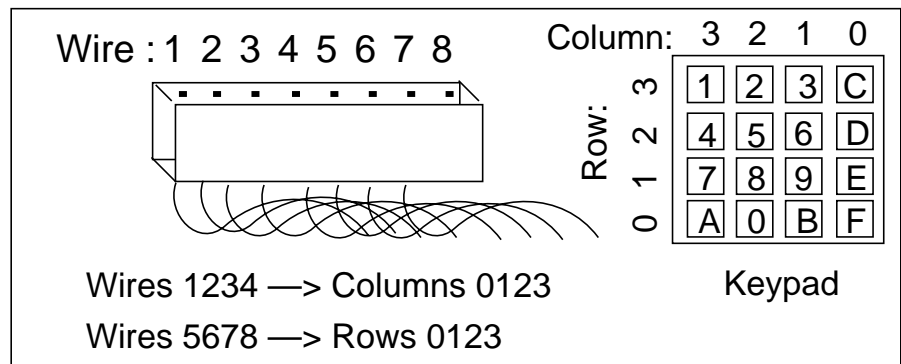
These are the various causes of what are called debounce problems. The keypad port of the MMC2001 handles this problem in hardware by requiring that a key press detection occur in four consecutive timing cycles.

When the hardware is successful in detecting a key press, the information is held until cleared by software. This eliminates all noises (glitches, spikes, etc.) of less than 16 ms in duration. The existing hardware simplifies the software writing task of detecting debounce.

### *Unique Keypad Outputs*

Keypads use a cross-connection of columns and rows to identify which key was pressed. These columns and rows may already be hard-wired to an output cable. This would produce a unique encoding for the output lines of the cable. The example in this application note uses a 4 x 4 (four columns by four rows) matrix of keys with an eight-line output cable.

**Figure 1** shows the output of the cable which is decoded in **Table 1**.



**Figure 1. Keypad Output Cable**

**Table 1. Correlation of Keys to Wires and Columns/Rows**

Key	Wires		4 x 4 Keypad	
	1 – 4	5 – 8	Column	Row
0	3	5	2	0
1	4	8	3	3
2	3	8	2	3
3	2	8	1	3
4	4	7	3	2
5	3	7	2	2
6	2	7	1	2
7	4	6	3	1
8	3	6	2	1
9	2	6	1	1
A	4	5	3	0
B	2	5	1	0
C	1	8	0	3
D	1	7	0	2
E	1	6	0	1
F	1	5	0	0

### *C Pseudo-Code Register Variable Names*

Manipulation of the registers in the keypad block involves an assembler level assignment. The KEY.H file defines variables which represent the registers and their respective bits. These variables can be treated as C character (char) or integer (int) variables in the KEY.C portion of the program. This makes the KEY.C program look like standard C.

The address of the registers of the keypad port start at location \$10003000. The individual bytes were assigned specific names for ease of access when coding in C. These names are described in [Table 2](#).

**Table 2. Register Addresses and Name Designations**

Address	Name (Upper Byte)	Name (Lower Byte)
\$10003000	KPCR	KPRE*
\$10003002	KPSR	KPKR*
\$10003004	KDDR	KRDD*
\$10003006	KPDR**	

\*Names not found in the reference manual. These are used for C code access.

\*\*KPDR register is set up for specific bit designations. Columns 0–3 are assigned to bits KPDR 8–11, respectively. Rows 0–3 are assigned to bits KPDR 0–3, respectively.

### *Polling vs. Interrupt Methods*

The program KEY.C runs in a continuous loop until cancelled. This loop waits for a key stroke, acts on the key stroke, and returns to wait for the next key. This method, called polling, is used when the entire program is run from the keys.

An interrupt method has the processor servicing activities beyond the control of the keypad program. When a key is pressed, an interrupt is called, and the key stroke is processed. After the interrupt, the processor is released to return to its own service routines.

## Software Structure

---

The main C program (KEY.C) has several supporting files. The file KEY.H defines the variables which help the main program address the registers and bits of the keypad module on the MMC2001. The file KEY.LNK sets the address locations for the program into memory.

### KEY.C

The KEY.C program is written as a stand alone example of keypad programming. This program will yield a single “keynum” value which can be used in any other software application.

### KEY.H

The addresses in the KEY.H program can be adjusted to different locations of the MMC2001 address map. The “#pragma” block defines a function which is in the superset of ANSII C. This function is first defined in the “#pragma” section block. The function is activated in the “#pragma use\_section” block. IOASECT and IOBSECT are the names of the two defined functions for the keypad registers.

A structure of the name REGISTER is defined. REGISTER sets the bit name for each bit in a 16-bit register. The register address of KPDR is set and the structure of REGISTER is applied to the register KPDR at its register address location. The column and row bits are specifically assigned here. After this assignment, the row value is connected to its register value by using the address KPDR.rowx (where x is 0..3) in the main KEY.C program.

The same holds true for the columns.

### KEY.LNK

The KEY.LNK program sets the starting points in memory and memory size for the ROM (read-only memory), RAM (random-access memory), and stack pointer. Uninitialized data space is assigned along with compiler memory allocations and identifiers.

## Steps for Decoding the Keypad

---

A combination of hardware and software procedures is performed to convert the mechanical key press operation to the software data value.

The polling process used in this application note involves a main loop which:

- Sets the registers
- Waits for a key to be pressed
- Scans the columns and rows for the active key
- Decodes the results of the scan
- Clears the registers for another entry
- Converts the decoded data to a software function
- Repeats the loop for another key stroke

### Set Up Registers for Keypad Input

This code module sets the registers for receiving a keypad input.

```
void set_registers(void)
{
    KPCR = 0x00;      /* Column 3-0 Open-Drain */
    KPPE = 0x0F;      /* Row 3-0 Active in Scan */
                    /* Write 0's to Data Register */
    KPDR.LCD_E = 0;   /* KPDR[15] <- 0's */
    KPDR.LCD_RW = 0;  /* KPDR[14] <- 0's */
    KPDR.LCD_RS = 0;  /* KPDR[13] <- 0's */
    KPDR.bit12 = 0;   /* KPDR[12] <- 0's */
    KPDR.col3 = 0;    /* KPDR[11] <- 0's */
    KPDR.col2 = 0;    /* KPDR[10] <- 0's */
    KPDR.col1 = 0;    /* KPDR[9] <- 0's */
    KPDR.col0 = 0;    /* KPDR[8] <- 0's */
    KPDR.bit7 = 0;    /* KPDR[7] <- 0's */
    KPDR.bit6 = 0;    /* KPDR[6] <- 0's */
    KPDR.bit5 = 0;    /* KPDR[5] <- 0's */
    KPDR.bit4 = 0;    /* KPDR[4] <- 0's */
    KPDR.row3 = 0;    /* KPDR[3] <- 0's */
    KPDR.row2 = 0;    /* KPDR[2] <- 0's */
    KPDR.row1 = 0;    /* KPDR[1] <- 0's */
    KPDR.row0 = 0;    /* KPDR[0] <- 0's */

    KDDR = 0xFF;      /* Cols are outputs, Rows are inputs */
    KRDD = 0x70;      /* Rows are inputs, Out for LCD controls */
    KPKR = 0x0F;      /* Clear KPKD bit, write "1" */
    KPSR = 0x01;      /* Set the KDIE bit */
    KPSR = 0xFD;      /* Clear the KRIE bit */
}
```

The specific bit manipulations are described as:

- The KPCR sets the column bits as totem-pole driven. The column bits may be reset to open-drain if desired. The term “open-drain” is defined as having the p-channel pullup disabled.
- The KPRE register enables the four rows zero to three [0:3] into active scan.
- Each bit of the KPDR register is initialized to zero [0].
- The KDDR register is written to set the columns to output configuration.
- The KRDD register (lower byte of KDDR) sets the row data bits as inputs.
- The status bits (KDSC, KRSS, KPKR, and KPKD) are cleared and the enable bit (KDIE) is set. A logic 1 is written to the status bits to clear their active states. By setting the interrupt enable bit (KDIE), the software is provided with a mechanism to detect a key depress.

### Have the Program Wait for Keypad Input

The wait module aligns the hardware and the software. After the registers have been set up to receive a key press, this program waits for a key press before proceeding. This is accomplished by entering a while loop which checks for any changes in the KPKR register. With the KDIE bit set along with a key press, the KPKR byte register changes to reflect the new state in the KPKD bit. This causes an exit from the while loop and returns control to the main calling program. The software needs a delay for alignment with the hardware. Otherwise, the software would run ahead of the hardware and would fail to wait for a key press. The count of 60,000 cycles was found to be sufficient and a shorter delay cycle might work just as well.

This method of waiting for a change in the KPKR register also provides a programming hook for a future interrupt version of this program. The line that feeds the KPKD bit is driven to the interrupt block for the MMC2001. Therefore, a future interrupt handling module would be located here in the program and would tap off of the same signal line.

Both the delay and wait\_key submodules are included in [Program Listings](#).

```
/* <<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> */
void delay(void)
{
int time_pass;
for(time_pass=0; time_pass<60000; time_pass++ )
{
continue;}
}
/* <<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>*/
void wait_key()
{
int first;          /* Initial value of KPKR */
delay();           /* delay to align hardware with software */
first = KPKR;      /* Original value of KPKR */
/* Check for consistent KPKR value */
while(first == KPKR)
{
continue;
}
return;
} /* end wait_key */
```

## Scan Columns and Rows for Active Key

The modules for column and row scan perform the function of converting the hardware data to software data. To scan for an active key, the columns are first prepared as outputs, charged to a high state, and then set to open drain. Next, the columns are tested consecutively along with the associated four rows for each column. A successful test will have the column set to zero [0] and then a read of each of the four rows produces a low row value. Saving the low values for the specific column and row completes the functionality of these modules.

The setting of the column values may appear redundant from the previous register setup. However, these are essential local functions which make the scan modules independent of other software in the program.

The active\_row variable in the scan\_key module serves a control and data function. While active\_row is equal to zero [0], the active row has not been found. This will continue the scan into the remaining columns. When an active row has been found, active\_row will be set to a value of one through four [1-4]. The final active\_row value will be translated back to a row value, zero through three [0-3], for a return value.



## Prepare Registers for Next Key Input

Since the key data is retained in memory, the registers will immediately be reset for another key press. The column values are set to zero [0] to permit a clearing of the status bits. Next, status bits KPKD and KPKR are cleared and finally the interrupt enable bit, KDIE, is reset.

If an interrupt method is incorporated into this program, then this reset is required to stop an interrupt request to the core. Therefore, the module `reg_set` is good housekeeping for the polling method but is also intended to provide an essential hook for a future interrupt version.

## Decode Key Input

This part of the program converts the input hardware data into a usable software variable.

The top line, which is commented out, is a combination of the three tasks in one line.

1. First, the column data is entered at the address location of the variable `key`. The column data is cast as a character which will be stored in four bits of data.
2. Second, the stored column data (in character format) is shifted left by one character (4 bits).
3. Third, the row data is input as a character into the lower four bits of the “key” variable.

**NOTE:** *The row data is OR-functioned into the “key” variable.*

```
char key;  
/* key = (((char)(kcol))<<4)|((char)(krow)); */  
key = (char)kcol; /* Enter Column Data into 'key' */  
key = (key<<4); /* Shift Column data to left character of 'key' */  
key = key | ((char)krow); /* Add Row data to right character of 'key' */
```

The column and row data are translated into the single keypad data. This is done with a case statement. The two input values are translated into a single case input value (`'key'`). This gives a bitwise translation from the input bits of the keypad port to the case statement. The case statement creates the output result of `keynum`.

This method is an example of how to represent the hardware bits in a C software format.

### Main Program Loop

The main program provides a loop to repeat any number of key inputs. This module is a sequence of module calls which are repeated. Generally, global variables have been used which are modified by the module calls. The `set_registers` call prepares the registers for key input. The `wait_key` call will delay the program until a key is pressed. The `scan_key` call scans each column and four associated rows to determine the data value for the active column and row. The `key_decode` call inputs the values of the active column and active row into a decryption routine. This routine yields a software value that is now independent of the hardware. The software value of the key is now available for any software application. The main loop is returned to the top to wait for the next key.

The `main` module could be used as the interrupt-handling module in an interrupt format. A key press would call for an interrupt which would call this routine when acknowledged.

## Program Enhancements

---

### Other Functions

A basic C level programming example for the keypad port of the MMC2001 is presented here, leaving room for several enhancements and alterations.

### Interrupt Key Format

With the interrupt format, the M•CORE processor is permitted to run any process independent of the keypad. When a key is pressed, an interrupt is sent to the interrupt block and is handled according to the priority of all outstanding interrupts. After the key stroke is handled, control returns to the processor.

Two hooks exist in the current program for conversion to interrupt the format. They are:

- First, the polling method used here is set to alert off of the same signal that would be sent to the interrupt block. This is found in the `wait` section of the program and could be replaced by an interrupt call module.

- Second, the `main` module could be converted into the top interrupt handling module for receiving a key stroke input.

### Dual Key Detection

The circuitry of the key port module on the MMC2001 is designed to detect two simultaneous key presses. The variable `ar1` (which is a local variable in the `get_row` submodule) is designed to be used as a test which checks for a second active row (`ar2`). The global variables would also have to be expanded to handle a second active column and a second active row. Of course, the same key decode submodule could be called a second time with the second variables passed to it. The final meaning of a double key stroke is up to the interpretation of the calling application.

### Different Keypad Size

The key port module of the MMC2001 is designed to handle keypads up to 8 x 8 in size, a maximum of eight columns and a maximum of eight rows. Additional columns would be added and checked at zero value in the `scan_key` module. Additional rows would be added and tested for a zero value in the `get_row` module. In this manner, an adjustment of any combination of columns and rows can be implemented.

## Program Listings

---

**KEY.C Listing**            The KEY.C listing provides the main keypad program. This part represents transportable C code. The other listings introduce specific hardware and compiler structures.

```
/* This program is for the M.CORE MMC2001 keypad entry.
* Program Name: key.c
* Written by Glenn Jackson
* Date: 9/14/98
* Description:
* Part 1: Set up the registers for the Keypad input
* Part 2: Enter delay / wait for keypad input
* Part 3: Prepare registers for next key input
* Part 4: Scan columns and rows for active key
* Part 5: Decode the key input
* Part 6: Main Loop: Cycle from Part 1 to Part 5
*/

/*****
/* include section *****/

#include "key.h"

/* Declare the functions or Subroutines *****/

/* Define the addresses */
int act_col;
int act_row;
int key_col;
int key_row;
char keynum;
/* Part 1: Set up registers */
/*****
/* Modules for PART 1:            Set up Registers            */
/*****
/* <<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> */
void set_registers(void)
{
KPCR = 0x00; /* Column 3-0 Open-Drain */
KPRE = 0x0F; /* Row 3-0 Active in Scan */
/* Write 0's to Data Register */
KPCR.LCD_E = 0; /* KPDR[15] <- 0's */
KPCR.LCD_RW = 0; /* KPDR[14] <- 0's */
KPCR.LCD_RS = 0; /* KPDR[13] <- 0's */
KPCR.bit12 = 0; /* KPDR[12] <- 0's */
KPCR.col3 = 0; /* KPDR[11] <- 0's */
KPCR.col2 = 0; /* KPDR[10] <- 0's */
KPCR.col1 = 0; /* KPDR[9] <- 0's */
KPCR.col0 = 0; /* KPDR[8] <- 0's */
KPCR.bit7 = 0; /* KPDR[7] <- 0's */
KPCR.bit6 = 0; /* KPDR[6] <- 0's */
KPCR.bit5 = 0; /* KPDR[5] <- 0's */
KPCR.bit4 = 0; /* KPDR[4] <- 0's */
}
```



## Application Note

```

{
int ar1;  /* Active row in get_row module */
ar1 = 0;
if (KPDR.row0==0)
    {
        ar1 = 1;
        return(ar1);
    }
if (KPDR.row1==0)
    {
        ar1 = 2;
        return(ar1);
    }
if (KPDR.row2==0)
    {
        ar1 = 3;
        return(ar1);
    }
if (KPDR.row3==0)
    {
        ar1 = 4;
        return(ar1);
    }
return(ar1);
} /* End of get_row */

/* <<<<<<<<<>>>>>>>>>>>>>>>> */
/* Scan for active key */
void scan_key(int *acol, int *arow)
/* int acol;
int arow; */
{
extern char KPCR;
int active_row;

*acol = 9; /* "9" is unique here (for debug) */
*arow = 8; /* "8" is unique here (for debug) */
active_row = 0;

/* Disable keypad interrupts */
KPSR = 0x00; /* disable KRIE,KDIE */

/* Write 1's to Column data */
/* Set up for scan */
KPDR.col0 = 1; /* KPDR[8] <- 1's */
KPDR.col1 = 1; /* KPDR[9] <- 1's */
KPDR.col2 = 1; /* KPDR[10] <- 1's */
KPDR.col3 = 1; /* KPDR[11] <- 1's */
KDDR = 0xFF; /* Columns to Outputs */
KPCR = 0x00; /* Columns to totem-pole outputs, to charge 1 */
KPCR = 0x0F; /* Columns [3:0] to Open Drain */

/* Column-Row Scan */

/* ***** GET COLUMN ***** */
/* Walk a zero in columns 0 to 3 */

/* Zero Column 0 Only */
KPDR.col0 = 0; /* KPDR[8] <- 0's */
KPDR.col1 = 1; /* KPDR[9] <- 1's */

```

```

KPDR.col2 = 1; /* KPDR[10] <- 1's */
KPDR.col3 = 1; /* KPDR[11] <- 1's */

active_row = get_row();
if (active_row > 0)
    {
        *acol = 0;
        *arow = active_row - 1; /* '-1' for alignment */
    }
/* End Column 0 check */
if (active_row==0) /* No previous column */
{
/* Zero Column 1 Only */
KPDR.col0 = 1; /* KPDR[8] <- 1's */
KPDR.col1 = 0; /* KPDR[9] <- 0's */
KPDR.col2 = 1; /* KPDR[10] <- 1's */
KPDR.col3 = 1; /* KPDR[11] <- 1's */

active_row = get_row();
if (active_row > 0)
    {
        *acol = 1;
        *arow = active_row - 1; /* '-1' for alignment */
    }
} /* End column 1 check */
if (active_row==0) /* No previous column */
{
/* Zero Column 2 Only */
KPDR.col0 = 1; /* KPDR[8] <- 1's */
KPDR.col1 = 1; /* KPDR[9] <- 1's */
KPDR.col2 = 0; /* KPDR[10] <- 0's */
KPDR.col3 = 1; /* KPDR[11] <- 1's */

active_row = get_row();
if (active_row > 0)
    {
        *acol = 2;
        *arow = active_row - 1; /* '-1' for alignment */
    }
} /* End column 2 check */
if (active_row==0) /* No previous column */
{
/* Zero Column 3 Only */
KPDR.col0 = 1; /* KPDR[8] <- 1's */
KPDR.col1 = 1; /* KPDR[9] <- 1's */
KPDR.col2 = 1; /* KPDR[10] <- 1's */
KPDR.col3 = 0; /* KPDR[11] <- 0's */

active_row = get_row();
if (active_row > 0)
    {
        *acol = 3;
        *arow = active_row - 1; /* '-1' for alignment */
    }
} /* End column 3 check */
/* default return if no-key */
} /* End of Scan_Key */

```

## Application Note

```

/*****
/* Modules for PART 5:      Column Row Decode                          */
/*****
/* <<<<<<<<<>>>>>>>> */
char key_decode (int kcol, int krow)
/*  *kcol -- Active keyed column */
/*  *krow -- Active keyed row */
{
extern char keynum; /* Returns the decoded keystroke */
char key;
/*  key = (((char)(kcol))<<4)|((char)(krow)); */
key = (char)kcol; /* Enter Column Data into 'key' */
key = (key<<4); /* Shift Column data to left character of 'key' */
key = key | ((char)krow); /* Add Row data to right character of 'key' */
switch ( key )
{ /* This decode is for a specific 0-F keypad */
/* YMMV -- Your milage may vary. */
case 0x00: keynum = 'F';
break;
case 0x01: keynum = 'E';
break;
case 0x02: keynum = 'D';
break;
case 0x03: keynum = 'C';
break;
case 0x10: keynum = 'B';
break;
case 0x11: keynum = '9';
break;
case 0x12: keynum = '6';
break;
case 0x13: keynum = '3';
break;
case 0x20: keynum = '0';
break;
case 0x21: keynum = '8';
break;
case 0x22: keynum = '5';
break;
case 0x23: keynum = '2';
break;
case 0x30: keynum = 'A';
break;
case 0x31: keynum = '7';
break;
case 0x32: keynum = '4';
break;
case 0x33: keynum = '1';
break;
default: keynum = '*';
} /* End of Switch */
return(keynum);
} /* End of key_decode */

/* ****
/*              End of modules                                      */
/* ****

```





## Application Note

```
unsigned short bit5 :1;
unsigned short bit4 :1;
unsigned short row3 :1;
unsigned short row2 :1;
unsigned short row1 :1;
unsigned short row0 :1;
}REGISTER;

#define KPDR_DEF (unsigned long) 0x10003006 /* Set PORTB data GPIO/INT */
#define KPDR (*(volatile REGISTER*) (KPDR_DEF+0))
```

**KEY.LNK Listing**      The KEY.LNK listing is specific to the Diab Data C compiler. KEY.LNK allocates memory for the program RAM, ROM, and stack. Program control variables also are defined and set into memory in KEY.LNK.

```
/* Program File Name: key.lnk */
MEMORY
{
    rom:    org = 0x30000000, len = 0x1000
    ram:    org = 0x30001000, len = 0x5900
    stack:  org = 0x30006900, len = 0x1000
}

SECTIONS
{
/* The first group contains code and constant data */

GROUP : {
/* First take all code from all objects and libraries */

.text (TEXT) : {
*(.text) *(.rodata) *(.init) *(.fini) *(.eini)
. = (.+15) & ~15;
}
/* Next take all small CONST data */
.sdata2 (TEXT) : {}
} > ram

/* The second group will allocate space for the initialized data
* (.data/.sdata) and the uninitialized data (.bss/.sbss) in the "ram" section.
*/
GROUP : {
.data (DATA) : {}
/* .sdata contains small address data */
.sdata (DATA) : {}

/* This will allocate the .bss symbols */
.sbss (BSS) : {}
.bss (BSS) : {}
.debug : {}
.debug_sfnames : {}
.debug_srcinfo : {}
.line : {}
}
```

```

/* Any space left over will be used as a heap */
} >ram
}


/* Definitions of identifiers used by key.c, key.h and the different
 * crt0.s files. Their purpose is to control initialization and memory
 * allocation.
 *
 * __HEAP_START : Used by sbrk.c. Start of memory used by malloc() etc.
 * __HEAP_END   : Used by sbrk.c. End of heap memory
- * __HEAP_END   : Used by sbrk.c. End of heap memory
 * __SP_INIT    : Used by crt0.s. Initial address of stack pointer
 * __SP_END     : Used by sbrk.c. Only used when stack probing
 * __DATA_ROM   : Used by init.c. Address of initialized data in ROM
 * __DATA_RAM   : Used by init.c. Address of initialized data in RAM
 * __DATA_END   : Used by init.c. End of allocated initialized data
 * __BSS_START  : Used by init.c. Start of uninitialized data
 * __BSS_END    : Used by init.c. End of data to be cleared
 * ----- */

__HEAP_START = ADDR(.line) + SIZEOF(.line);
__SP_INIT    = ADDR(stack)+SIZEOF(stack);
__HEAP_END   = ADDR(ram)+SIZEOF(ram);
__SP_END     = ADDR(stack);
__DATA_ROM   = ADDR(.sdata2)+SIZEOF(.sdata2);
__DATA_RAM   = ADDR(.data);
__DATA_END   = ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START  = ADDR(.sbss);
__BSS_END    = ADDR(.bss)+SIZEOF(.bss);

/* Some targets use an extra underscore in front of identifiers
 * ----- */
__HEAP_START = __HEAP_START;
__HEAP_END   = __HEAP_END;
__SP_INIT    = __SP_INIT;
__SP_END     = __SP_END;
__DATA_ROM   = __DATA_ROM;
__DATA_RAM   = __DATA_RAM;
__DATA_END   = __DATA_END;
__BSS_START  = __BSS_START;
__BSS_END    = __BSS_END;

```

# Application Note

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

## How to reach us:

**USA/EUROPE/Locations Not Listed:** Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1-303-675-2140 or 1-800-441-2447

**JAPAN:** Motorola Japan Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu, Minato-ku, Tokyo 106-8573 Japan. 81-3-3440-3569

**ASIA/PACIFIC:** Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.  
852-26668334

**Technical Information Center: 1-800-521-6274**

**HOME PAGE:** <http://www.motorola.com/semiconductors/>



**MOTOROLA**

© Motorola, Inc., 2000

AN1803/D