Keywords: MAXQ, IAR, memory allocation, flash data, flash storage, SRAM

Dec 16, 2011

APPLICATION NOTE 5262

Allocate Flash and SRAM Memory on a MAXQ® Microcontroller Using the IAR Compiler

By: Sanjay Jaroli
Dec 16, 2011

Abstract: MAXQ devices provide special utility ROM functions, which are called to read and write data from program memory. However, data stored in program memory cannot be accessed directly on MAXQ microcontrollers. Instead, the utility ROM functions start addresses are integrated in IAR Embedded Workbench® to access the stored data. This application note demonstrates how to allocate and access flash and SRAM memory on a MAXQ microcontroller using IAR Embedded Workbench tools.

Introduction

The MAXQ architecture describes a powerful, single-cycle RISC microcontroller based on the classic Harvard architecture, in which the program and data memory buses are separate. This organization requires dedicated buses for each memory (Figure 1), so instructions and operands can be fetched simultaneously. Because there is no contention for a single data bus, MAXQ instructions can execute in only a single cycle.

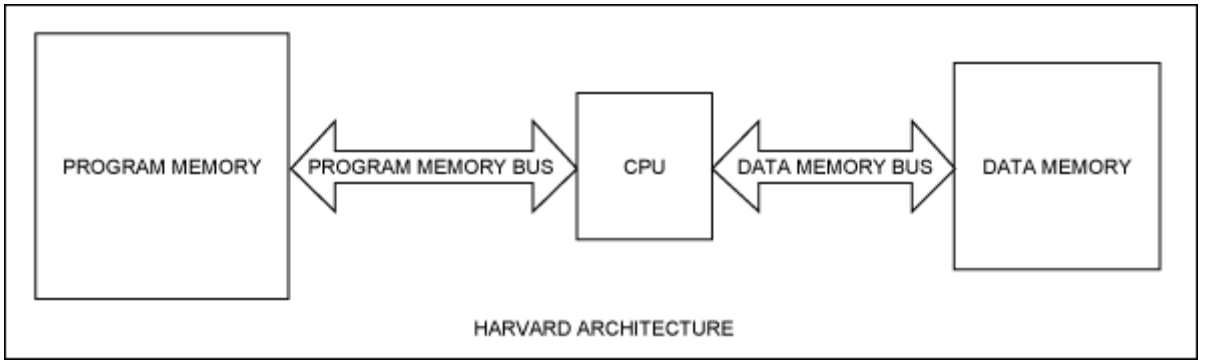

Figure 1. Harvard architecture.

Each MAXQ device incorporates the following memory types:

1. Flash memory
2. SRAM
3. Utility ROM

MAXQ devices can also execute program code from flash, utility ROM, or SRAM. While executing program code from one memory segment, the other two memory segments can be used as data memory (Refer to the Program Execution from Flash Memory and Execution Utility ROM Functions sections for further details.) This is because the program and data memory busses cannot access the same memory segment simultaneously.

Being a Harvard machine, one might assume that MAXQ microcontrollers prohibit storing data elements to nonvolatile flash memory. However, the MAXQ devices are designed with built-in utility ROM functions that allow for both reading and writing data to nonvolatile flash memory.

Program Execution from Flash Memory

In the MAXQ devices, when the application program is executing from flash memory, the data memories are SRAM (read and write) and utility ROM (read only). Refer to Table 1 for the data memory map and Figure 2 for the memory map when code is executing from flash.

1. The SRAM data memory is located in the memory map from address 0x0000 to 0x07FF (in the byte addressing mode) or from address 0x0000 to 0x03FF (in the word addressing mode).
2. The Utility ROM is located in the memory map from address 0x8000 to 0x9FFFh (byte mode) or from address 0x8000 to 0x8FFF (in the word addressing mode).

Table 1. Data Memory Map When Application Code Executes from Flash Memory

Addressing Mode	SRAM		Utility ROM	
	Start Address	End Address	Start Address	End Address
Byte Mode	0x0000	0x07FF	0x8000	0x9FFF
Word Mode	0x0000	0x03FF	0x8000	0x8FFF

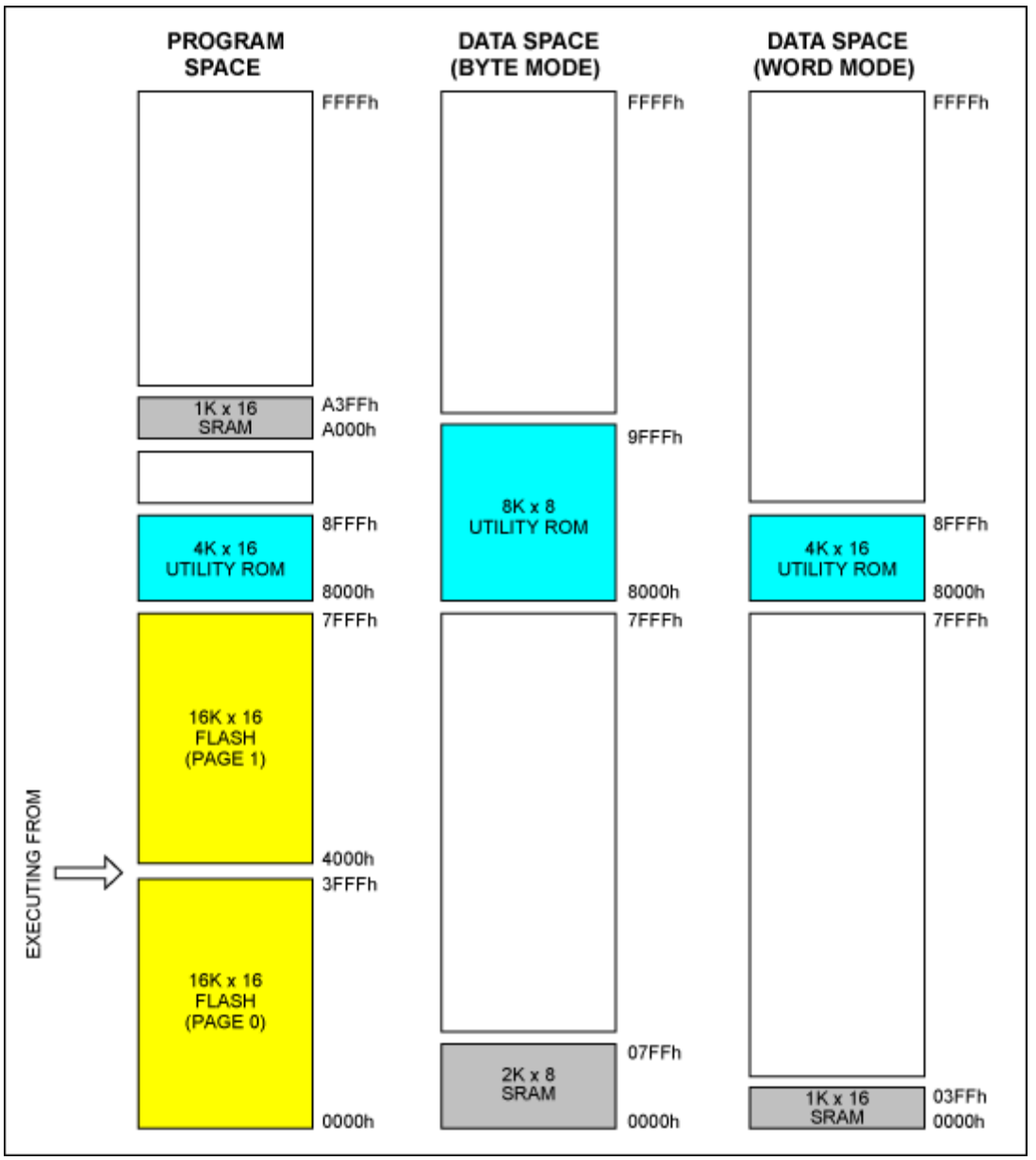

Figure 2. Memory map when application program executes code from the flash memory.

Executing Utility ROM Functions

When executing utility ROM functions, the data memories are SRAM (read and write) and flash (read and write). When the application program is executing from flash and variables or data objects are allocated in the flash memory, these variables or data objects can be read or written via utility ROM functions. By jumping program execution to the utility ROM functions, flash memory can now be accessed as data. Refer to Table 2 for the data memory map and Figure 3 for the memory map when code is executing from the utility ROM.

1. The SRAM data memory is located in the memory map from address 0x0000 to 0x07FF (in the byte addressing mode) or from address 0x0000 to 0x03FF (in the word addressing mode).
2. In the byte addressing mode, the lower half of flash is located in the memory map from address 0x8000 to 0xFFFFh when CDA0 = 0, and the upper half of flash is located in the memory map from address 0x8000 to 0xFFFFh when CDA0 = 1. In the word addressing mode, the flash is located in the memory map from address 0x8000 to 0xFFFF.

Table 2. Data Memory Map when Executing Utility ROM Functions

Addressing Mode	SRAM		Flash Memory Lower Half (CDA0 = 0)		Flash Memory Upper Half (CDA0 = 1)		Flash Memory	
	Start Address	End Address	Start Address	End Address	Start Address	End Address	Start Address	End Address
Byte Mode	0x0000	0x07FF	0x8000	0xFFFF	0x8000	0xFFFF	—	—
Word Mode	0x0000	0x03FF	—	—	—	—	0x8000	0xFFFF

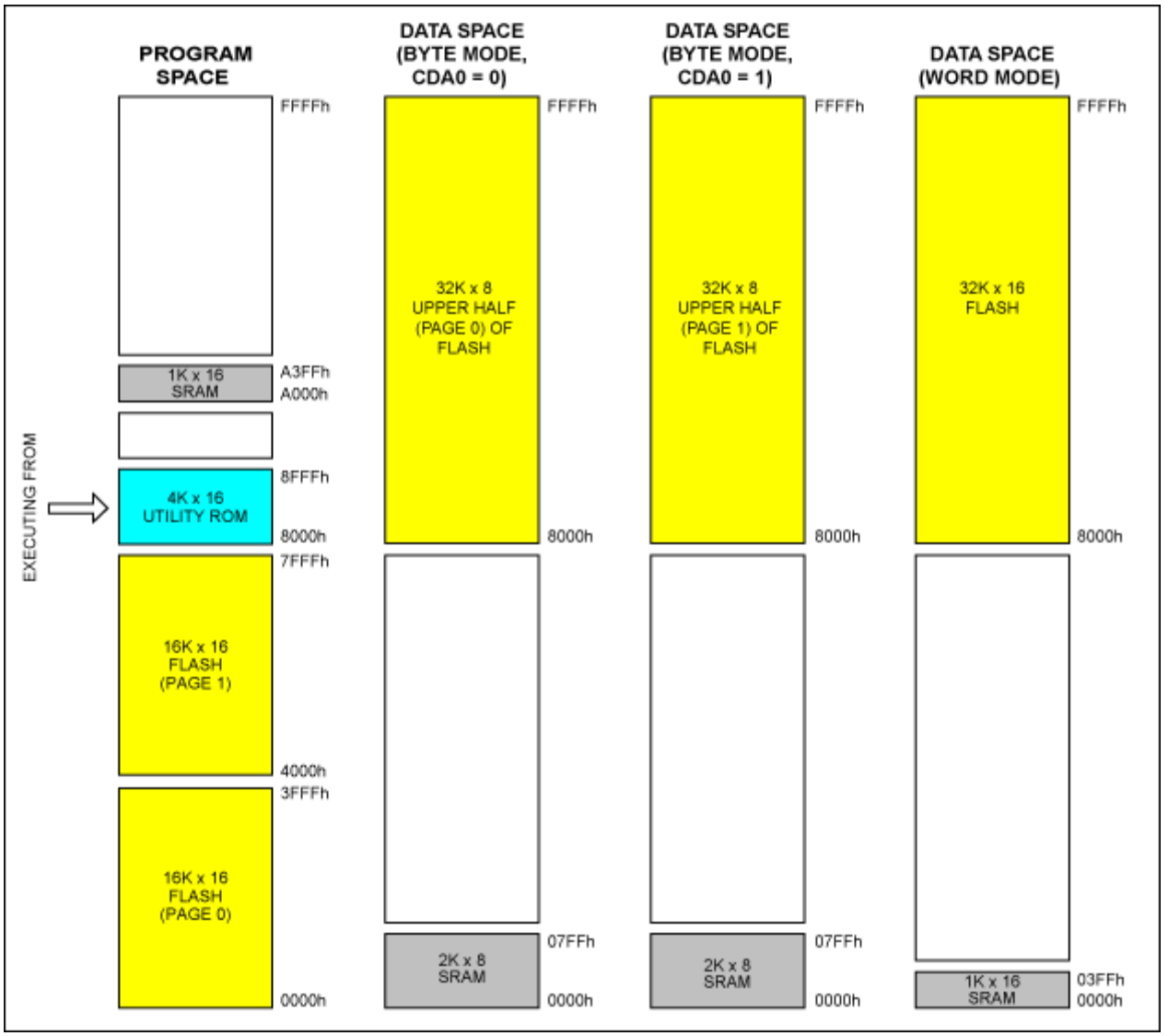

Figure 3. Memory map when executing utility ROM functions.

Memory Allocation in the flash and SRAM

IAR Embedded Work Bench IDE is used for programming MAXQ core-based microcontrollers. IAR™ C compiler (for MAXQ microcontroller) provides the option to define data objects or variables in the flash or SRAM locations. The compiler has special keywords pragma location and pragma required; by using these keywords, memory can be allocated to the data objects or variables at the absolute address. These variables or data objects must be declared with IAR keyword __no_init or const (the standard C keyword). See keyword descriptions of __no_init, const, pragma location, and program required below.

Keywords Description

pragma location

The #pragma location keyword is used to place individual global or static variables or data objects at the absolute addresses. The variables or data objects must be declared either __no_init or const. This is useful for individual data objects that must be located at a fixed address, such as variables, data objects with external or internal interfaces, or populating hardware tables.

pragma required

The #pragma required ensures that a symbol that is needed by another symbol is included in the linked output. The directive must be placed immediately before the second symbol. Use the directive if the requirement for a symbol is not otherwise visible in the application. For example, if a variable is only referenced indirectly through the segment it resides in, #pragma required must be used.

__no_init

Normally, the IAR runtime environment will initialize all global and static variables to 0 when the application is started. IAR C compiler supports the declaration of variables that will not be initialized, using the __no_init type modifier. Variables declared with __no_init are suppressed on the startup. It is not possible to give a __no_init object an initial value.

Example:__no_init char MaximChar @ 0x0200;

In this example, a __no_init declared variable is placed at an absolute address in the default data memory (SRAM).

const

The const keyword implies that an object is read only. This type of qualifier is used for indicating that a data object, accessed directly or via a pointer, is nonwritable. When const is used with keyword #pragma location and #pragma required, IAR allocates memory at the location defined by #pragma location. This is useful for configuration parameters that are accessible from an external interface. Such flash data objects can be read or written by the Utility ROM functions only.

Constant variables placed at an absolute address are not accessible in IARs default memory model. Use the option Place constants in CODE (In IAR Project → Option → General Option → Target window) to make them accessible, as shown in Figure 4.

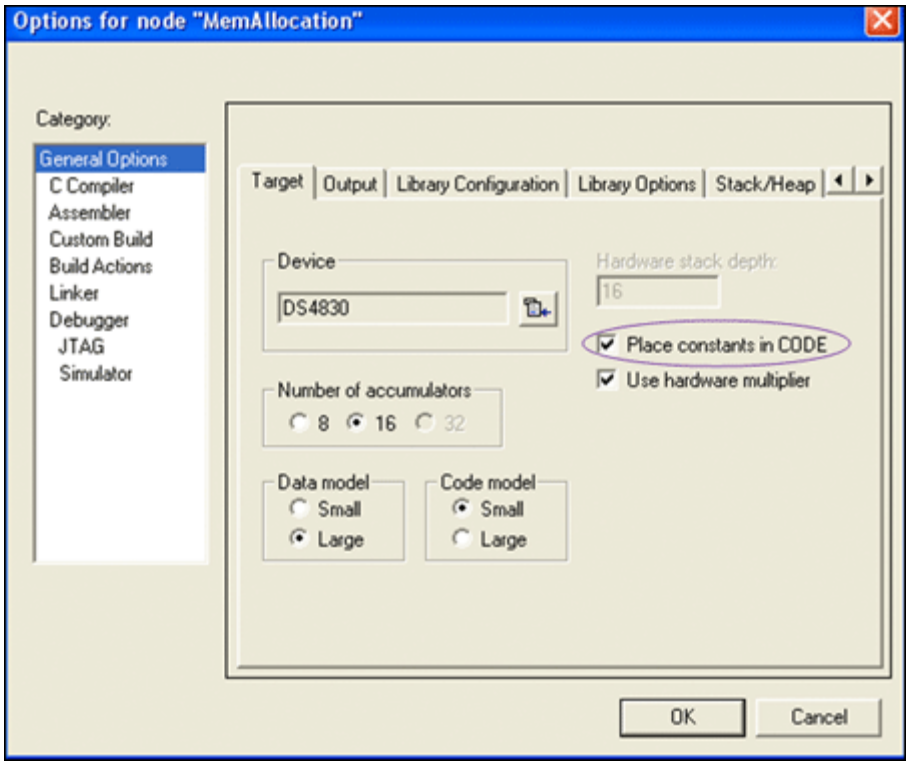

Figure 4. IAR Project Option window.

Example 1

```
const int FLASH_DATA0;
//FLASH_DATA0 is initialized to 0x0000 and linker will allocate memory address.
```

Example 2

```
#pragma location = 0xA000
const int FLASH_DATA1 = 0x1234;
#pragma required = FLASH_DATA1
```

Here memory is allocated at flash address 0xA000 and initialized to 0x1234.

Example 3

```
#pragma location = 0xA002
__no_init const int FLASH_DATA2 //Memory is allocated at the address 0xA002 (byte address)
#pragma required = FLASH_DATA2
```

Here memory is allocated at flash address 0xA002 without initialization.

In the above examples, there are three const declared objects, where the first is initialized to zero, the second is initialized to a specific value, and the third is uninitialized. All three variables are placed in the flash.

Keyword Examples

Example 1

In the following example, FLASH_CONFIG is a FlashMemoryMap structure variable. This structure variable's start address is explicitly defined at location "CONFIG_FLASH" (0xEE00) using keywords `#pragma location` and `#pragma required`.

```
//Structure for Memory Map
typedef struct
{
  unsigned char SYSTEM_CONFIG;                //Address 0x00
  unsigned char TEMP_CONFIG;                  //Address 0x01
  unsigned char SLAVE_ADDR_A0;                //Address 0x02
  unsigned char NULL_A0_3;                    //Address 0x03
  signed   int  INTERNAL_TEMP_THRES;          //Address 0x04-5
  signed   int  EXTERNAL_TEMP_THRES;          //Address 0x06-7
  signed   int  DS75_TEMP_THRES;              //Address 0x08-9
}FlashMemoryMap;

#define CONFIG_FLASH = 0xEE00 //Flash Address

#pragma location = CONFIG_FLASH
const FlashMemoryMap FLASH_CONFIG =  //Initialize data objects variable
                              {
                                0x00,           // SYSTEM_CONFIG
                                0xFE,           // TEMP_CONFIG
                                0xA0,           // SLAVE_ADDR_A0
                                0x00,           // NULL_A0_3
                                0x3200,         // INTERNAL_TEMP_THRES
                                0x4200,         // EXTERNAL_TEMP_THRES
                                0x5200          // DS75_TEMP_THRES
                              };
#pragma required = FLASH_CONFIG
```

To see memory allocation and initialization in IAR Embedded Work Bench IDE, go to View → Memory. In the displayed edit box, type 0xEE00 in the Go to box and select Code from dropdown box, as shown in Figure 5.

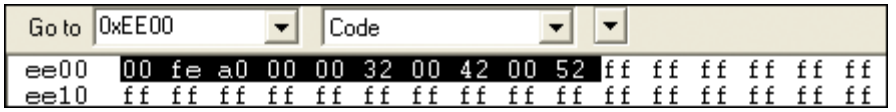

Figure 5. Memory allocation.

Example 2

In the following example, a DATA SRAMMemoryMap structure variable (DATA_MONITOR) is created at address 0x0116, and it will not be initialized (using `__no_init` type modifier).

```
typedef struct
{
  //Read Only
  signed int    INTERNAL_TEMP;                //Address = OFFSET + 0x00-1
  signed int    EXTERNAL_TEMP;                //Address  = OFFSET + 0x02-3
  signed int    DS75_TEMP;                    //Address  = OFFSET + 0x04-5
  signed int    VOLTAGE0;                     //Address  = OFFSET + 0x06-7
  signed int    VOLTAGE1;                     //Address  = OFFSET + 0x08-9
}SRAMMemoryMap;

#define CONFIG_SRAM 0x0116                    //SRAM Address 0x0116

#pragma location = CONFIG_SRAM
__no_init SRAMMemoryMap DATA_MONITOR;
#pragma required = DATA_MONITOR
```

To see the contents of the structure variable when debugging in IAR, select the variable, right click, and choose the Add to Watch option. See Figure 6.

Watch

Expression	Value	Location	Type
⊟ DATA_MONITOR	<struct>	Data:0x116	SRAMMemoryMap
├ INTERNAL_TEMP	0	Data:0x116	int
├ EXTERNAL_TEMP	0	Data:0x118	int
├ DS75_TEMP	0	Data:0x11A	int
├ VOLTAGE0	0	Data:0x11C	int
└ VOLTAGE1	0	Data:0x11E	int

Figure 6. An IAR Watch window.

Viewing Allocated Memory in the Intel® HEX File

Memory allocated for data objects in the code memory can be viewed in the Intel HEX file generated by IAR Embedded Workbench. See the highlighted area in Figure 7. Here, data objects are allocated memory in the flash between 0xEE00 and 0xEE15.

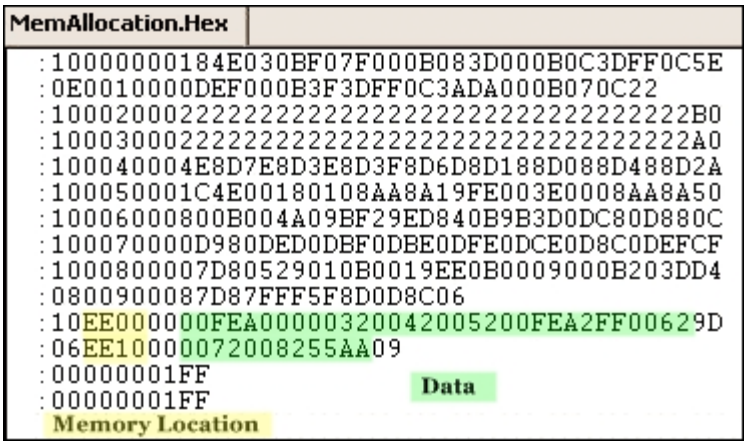

Figure 7. An IAR-generated HEX file in the Release mode.

Example Code

The included example code has files that demonstrate how to allocate memory for variables in flash and SRAM, and also how to read and write to the variables in flash. The included files are:

1. main.c demonstrates reading and writing to flash memory.
2. memory.h demonstrates data object creation and initialization in flash and SRAM. This file uses the __no_init, const, pragma location, and pragma required keywords.
3. flash.c has flash read and write functions (C functions). These functions call assembly functions, which are defined in the assembly.asm file.
4. flash.h declares flash read and write function prototypes. These functions have definitions in flash.c and assembly.asm.
5. assembly.asm has various assembly functions to read and write flash memory. All functions have Utility ROM function calls.

Download complete project.

Related Parts		
DS4830	Optical Microcontroller	Free Samples

Next Steps	
EE-Mail	Subscribe to EE-Mail and receive automatic notice of new documents in your areas of interest.
Download	Download, PDF Format (322.2kB)
Share	Other Channels E-Mail this page to an associate or friend.

More Information

For Technical Support: http://www.maxim-ic.com/support
For Samples: http://www.maxim-ic.com/samples
Other Questions and Comments: http://www.maxim-ic.com/contact

Application Note 5262: http://www.maxim-ic.com/an5262
APPLICATION NOTE 5262, AN5262, AN 5262, APP5262, Appnote5262, Appnote 5262