cyan technology

AN047 – Accessing Large Data Areas

Version 1.1

Contents

1 Introduction 4
2 Glossary 4
3 Installation Instructions 5
3.1 LMA.def 5
3.2 LMA.h 7
4 Operation 8
5 Notes 8
6 API Functions 9

Confidential and Proprietary Information

©Cyan Technology Ltd, 2007

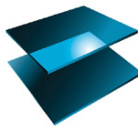
This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.



cyan technology

Revision History

Version	Date	Notes
V1.0	21/11/2006	First release
V1.1	16/07/2007	Modified for CyanIDE V1.4 and eCOG1X. Added word access API functions.

1 Introduction

The eCOG1k and eCOG1X have a linear 16-bit logical address range (64Kwords) in data space. This is mapped into the 24-bit physical address range (16Mwords) using the MMU peripheral. Direct access to the full 16Mword address range is difficult because the C compiler only uses the 64Kwords of logical data space. Some applications may require access to large constant data arrays, for example graphics libraries or large font tables, where this 64Kword size limit is a significant restriction.

This application note describes a method for storing large constant data tables in memory and accessing them at run-time. The software includes two main components:

(1) A Python script extension to CyanIDE that allows constant data tables to be integrated with ROM code image files and downloaded to internal or external memory.

(2) A run-time support library that implements paged access to the data tables using the configurable data space memory translators in the MMU.

The complete solution allows character or integer access to large data tables in memory using absolute addressing via a simple API. CyanIDE automatically downloads the generated and updated constant data files to the eCOG1 when the application project is compiled and executed.

2 Glossary

A table of abbreviations used in this document.

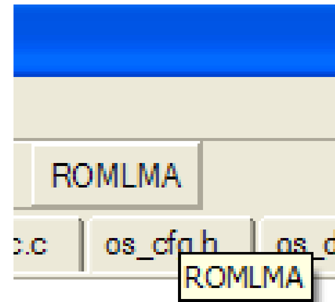
API	Application Programming Interface
eCOG1	Cyan Technology target micro controller
EMI	External Memory Interface
MMU	Memory Management Unit
LMA	Large Memory Access
ROM	Read-Only Memory

3 Installation Instructions

The zip file *AN047SW.zip* provided with this application note contains the following four files:

LMA.c	C source support routines for large memory access
LMA.h	Header file for the support routines
LMA.def	Definition file for the current project
LMA.py	Python script macro file

Add the *LMA.py* file to the CyanIDE project using *Project → Add Files to Project*. Once the file has been added to the project, right click on the file in the navigator pane, select *Properties* and set the property *Run on project load* to *Yes*. This sets it up such that the macro is imported into CyanIDE automatically, either by reloading the project or by typing “reload (LMA)” in the command window. Once this is complete, a button labelled *ROMLMA* appears on the macros toolbar in CyanIDE, next to the *FileHeader* button.



Extract the remaining files *LMA.c, LMA.h* and *LMA.def* from the zip file to the current project directory, and add them to the project using *Project → Add Files to Project* from the main menu.

The file *LMA.h* must be included in any source file that requires large memory access, with the statement *#include “LMA.h”*. This allows the application software to use the LMA API functions to read information from the constant data tables in memory.

3.1 LMA.def

The file *LMA.def* must be edited to set the parameters for the project; an example is shown below. Note that the parameter names must not be changed.

```
# Definition file for the Large Memory Access Controller
# DO NOT edit the parameter names

MCU_TYPE        eCOG1X          // Define MCU type

ROMNAME         "gb_font.rom"   // The ROM filename, enclosed in ""
USE_MEM         ROM             // Memory device (ROM, EM0 or EM1)
USE_TRAN_NUM    1               // Translator 0 or 1
TRAN_LOG        0xfa00          // Translator logical address
TRAN_SIZE       0x0100          // Translator block size
RESTORE_TRAN    0               // Set to 1 to preserve translator state
```

3.1.1 MCU_TYPE

This is the name of the MCU to be used in this project. Valid values are eCOG1k and eCOG1X.

3.1.2 ROMNAME

This is the name of the binary ROM image file to be inserted into program memory. The entire file is read and inserted, so care should be taken to ensure that it fits into the available memory and contains only necessary information. The ROM file itself should be a byte-wise or word-wise representation of the constant data. The byte ordering for multi-byte data values such as short or long items is big-endian.

3.1.3 USE_MEM

This is the name of the internal/external memory device to program with the data. Valid values are ROM (for eCOG1X only), EM0 and EM1. This parameter determines the extension of the output file and whether the input file should be inserted in a location within the existing file. If the file does not exist, then a new file is created automatically.

3.1.4 USE_TRAN_NUM

For physical memories that have more than one address translator, it is necessary to specify which translator is to be used for data accesses. For example, chip select CS1 has two data space translators available, ***ext_cs1_data0*** and ***ext_cs1_data1***. This option is provided to allow the software to reserve one translator for the large memory access functions, and the application software to use the other translator. This value is used only for physical memories that have more than one data space translator, and has no effect for physical memories with only one single translator.

3.1.5 TRAN_LOG

This is the logical base address for the MMU translator, used by the software when mapping a segment of the large memory area into data space during operation. Note that normal MMU rules apply to the block size and logical address, in that the translator logical address must be a multiple of the block size.

3.1.6 TRAN_SIZE

This is the block size for the MMU translator, used by the software when mapping a segment of the large memory area into data space during operation. Note that normal MMU rules apply to the block size and logical address, in that the translator logical address must be a multiple of the block size.

3.1.7 RESTORE_TRAN

This parameter is used when the application already makes extensive use of the MMU data space translators. When this is the case, set this value to ‘1’; this configures the software to save the address translator settings on entry to the function and restore them at exit. This adds a small overhead to the function call but allows the application software to operate without knowing the current status of the LMA manager.

Note: when using this feature, only the buffered API call (see section 6) is available, as the non-buffered call relies upon having the translator value mapped to the large memory area when it returns. The application software can still use the unbuffered API call if desired, but it must then be responsible for controlling the state of the translators after the API call.

3.2 LMA.h

The *LMA.h* include file is as follows:

```
#ifndef LMA_H
#define LMA_H

#define LMARead_EN               0 // =1 enable this function
#define LMAReadWord_EN           1 // =1 enable this function
#define LMASizeRtn_EN            0 // =1 LMARead/ReadWord returns size,
                                   // =0 LMARead/ReadWord is void function
#define LMAMMU_EN                1 // =1 MMU enabled in LMARead/ReadWord,
                                   // =0 MMU enabled only in LMAReadToBuffer/WordBuffer
#define LMAReadToBuffer_EN       0 // =1 enable this function,
                                   // LMARead_EN=1 & LMASizeRtn_EN=0/1 & /LMAMMU_EN=0/1
#define LMAReadWordBuffer_EN     0 // =1 enable this function,
                                   // LMAReadWord_EN=1 & LMASizeRtn_EN=0/1 & LMAMMU_EN=0/1
#define LMAWrite_EN              0 // =1 enable this function,
                                   // LMARead_EN=1 & LMASizeRtn_EN=0/1 & LMAMMU_EN=0/1
#define LMAWriteWord_EN          0 // =1 enable this function,
                                   // LMAReadWord_EN=1 & LMASizeRtn_EN=1 & LMAMMU_EN=0/1

/* Prototypes */
#if     LMASizeRtn_EN==1
unsigned int LMARead(unsigned char **pOut, unsigned long offset);
#else
void LMARead(unsigned char **pOut, unsigned long offset);
#endif

#if     LMASizeRtn_EN==1
unsigned int LMAReadWord(unsigned int **pOut, unsigned long offset);
#else
void LMAReadWord(unsigned int **pOut, unsigned long offset);
#endif

void LMAReadToBuffer(unsigned char *out, unsigned long offset, unsigned int size);
void LMAReadWordBuffer(unsigned int *out, unsigned long offset, unsigned int size);
void LMAWrite(unsigned char *in, unsigned long offset, unsigned int size);
void LMAWriteWord(unsigned int *in, unsigned long offset, unsigned int size);

// DO NOT EDIT THE FOLLOWING LINES, THIS IS DONE AUTOMATICALLY BY THE PYTHON SCRIPT
// LAST UPDATED                    Fri Jul 06 13:37:30 2007
#define MMU_RESTORE_TRANSLATOR     0
#define MMU_TRANSLATOR_LOG         rg.mmu.ext_cs1_data1_log
#define MMU_TRANSLATOR_PHY         rg.mmu.ext_cs1_data1_phy
#define MMU_TRANSLATOR_SIZE        rg.mmu.ext_cs1_data1_size
#define MMU_TRANSLATOR_EN          fd.mmu.translate_en.ext_cs1_data1
#define LMA_BLOCK_LOG              0x4000
#define LMA_BLOCK_SIZE             0x100
#define LMA_OFFSET                 0x0L
// END OF DO NOT EDIT

#endif   // LMA_H
```

A list of *#define* parameters (in blue above) are used to control conditional compilation in LMA.c, The following limitations apply.

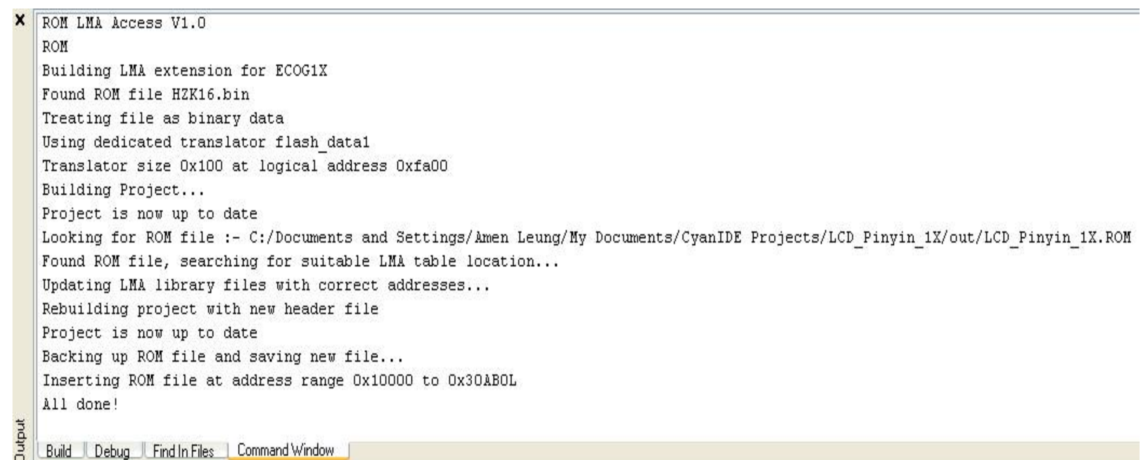
- If `LMAReadToBuffer_EN` is set to 1 then `LMARead_EN` must be set to 1.
- If `LMAReadWordBuffer_EN` is set to 1 then `LMAReadWord_EN` must be set to 1.
- If size of the data available within the current translator settings is required, then `LMASizeRtn_EN` must be set to 1 for `LMARead_EN` and `LMAReadWord_EN`.

A list of *#define* parameters (in red above) are set when the Python macro is executed. It is essential that these *#define* symbol names are not edited without the corresponding changes being made to the Python script in *LMA.py*, as these are automatically updated from the values set in the file *LMA.def* when the macro is executed. It is also essential that these *#define* values are valid for the first run of *LMA.py*; this depends on the MCU_TYPE value defined in *LMA.def*. For example, *fd.mmu.translate_en.ext_cs1_data1* is a valid translator name for eCOG1k and *fd.mmu.translate_en0.ext_cs1_data1* is a valid translator name for eCOG1X.

4 Operation

The source project should set up the configuration parameters as described above and include the function calls detailed in section 6 where necessary. Note that addresses within the data file are specified as a byte or word offset from the start of the file, regardless of where the file is located in physical memory.

When the application is ready to be built, click on the *ROMLMA* button on the macros toolbar. The command window displays current progress. During the operation, the project is built, the header file is updated with the correct parameters, and the project is rebuilt. The output files are then updated or created to insert the user binary ROM data. An example of the output from the command window is shown below, for the definitions given previously:



5 Notes

Only basic error checking is performed on parameters, so the user must take care to specify everything correctly.

Currently only a single LMA file is allowed. Future versions may support multiple files and multiple address translators.

While the software automatically manages the translator addresses, care must be taken to ensure that the logical address and size values are valid for the MMU configuration. Failure to respect this is likely to result in address exceptions and incorrect data.

6 API Functions

`unsigned int LMARead(unsigned char **pOut, unsigned long offset)`

`void LMARead(unsigned char **pOut, unsigned long offset)`

Sets up the MMU address translator to map the data located at byte address ***offset*** within the data file, using the specified translator parameters, then sets the pointer ***pOut*** to the mapped logical address for the requested data. If the symbol `LMASizeRtn_EN` is set to 1, then it returns the size of the data available within the current translator settings.

Parameters

***pOut*	Address of a char pointer that is to be set to the logical address of the requested data once the MMU translator is configured.
offset	The byte offset within the stored data file of the data to be accessed.

Returns (if `LMASizeRtn_EN` is set to 1)

The size of the data available within the current translator block (in bytes). The return value is the difference between the MMU translator block size and the position of the requested data relative to the beginning of this block, and represents the number of bytes that can be read from this address without changing the MMU translator setting.

`unsigned int LMAReadWord(unsigned int **pOut, unsigned long offset)`

`void LMAReadWord(unsigned int **pOut, unsigned long offset)`

Sets up the MMU address translator to map the data located at word address ***offset*** within the data file, using the specified translator parameters, then sets the pointer ***pOut*** to the mapped logical address for the requested data. If the symbol `LMASizeRtn_EN` is set to 1, then it returns the size of the data available within the current translator settings.

Parameters

***pOut*	Address of a word pointer that is to be set to the logical address of the requested data once the MMU translator is configured.
offset	The word offset within the stored data file of the data to be accessed.

Returns (if `LMASizeRtn_EN` is set to 1)

The size of the data available within the current translator block (in words). The return value is the difference between the MMU translator block size and the position of the requested data relative to the beginning of this block, and represents the number of words that can be read from this address without changing the MMU translator setting.

```
void LMAReadToBuffer(unsigned char *out,
                     unsigned long offset,
                     unsigned int size)
```

Sequentially calls ***LMARead()*** to fill the char buffer of ***size*** bytes, pointed to by ***out***, with the data starting at absolute offset ***offset***.

Parameters

**out*	A char pointer to the buffer to be filled.
offset	The byte offset within the stored data file of the data to be accessed.
Size	The number of bytes to copy to the buffer.

Returns

Nothing.

```
void LMAReadWordBuffer(unsigned int *out,
                       unsigned long offset,
                       unsigned int size)
```

Sequentially calls ***LMAReadWord()*** to fill the unsigned int buffer of ***size*** words, pointed to by ***out***, with the data starting at absolute offset ***offset***.

Parameters

**out*	An unsigned int pointer to the buffer to be filled.
offset	The word offset within the stored data file of the data to be accessed.
size	The number of words to copy to the buffer.

Returns

Nothing.

```
void LMAWrite(unsigned char *in,
              unsigned long offset,
              unsigned int size)
```

Sequentially calls ***LMARead()*** to fill the memory starting at absolute offset ***offset*** with ***size*** bytes from the address pointed to by ***in***. Note that this requires that the memory containing the LMA data table allows write access.

Parameters

**in*	A char pointer to the buffer containing the data to be written.
offset	The byte offset within the stored data file of the data to be accessed.
Size	The number of bytes to copy to the buffer.

Returns

Nothing.

```
void LMAWriteWord(unsigned int *in,
                  unsigned long offset,
                  unsigned int size)
```

Sequentially calls ***LMAReadWord()*** to fill the memory starting at absolute offset ***offset*** with ***size*** words from the address pointed to by ***in***. Note that this requires that the memory containing the LMA data table allows write access.

Parameters

**in*	A unsigned int pointer to the buffer containing the data to be written.
Offset	The word offset within the stored data file of the data to be accessed.
Size	The number of words to copy to the buffer.

Returns

Nothing.