IBM Semea

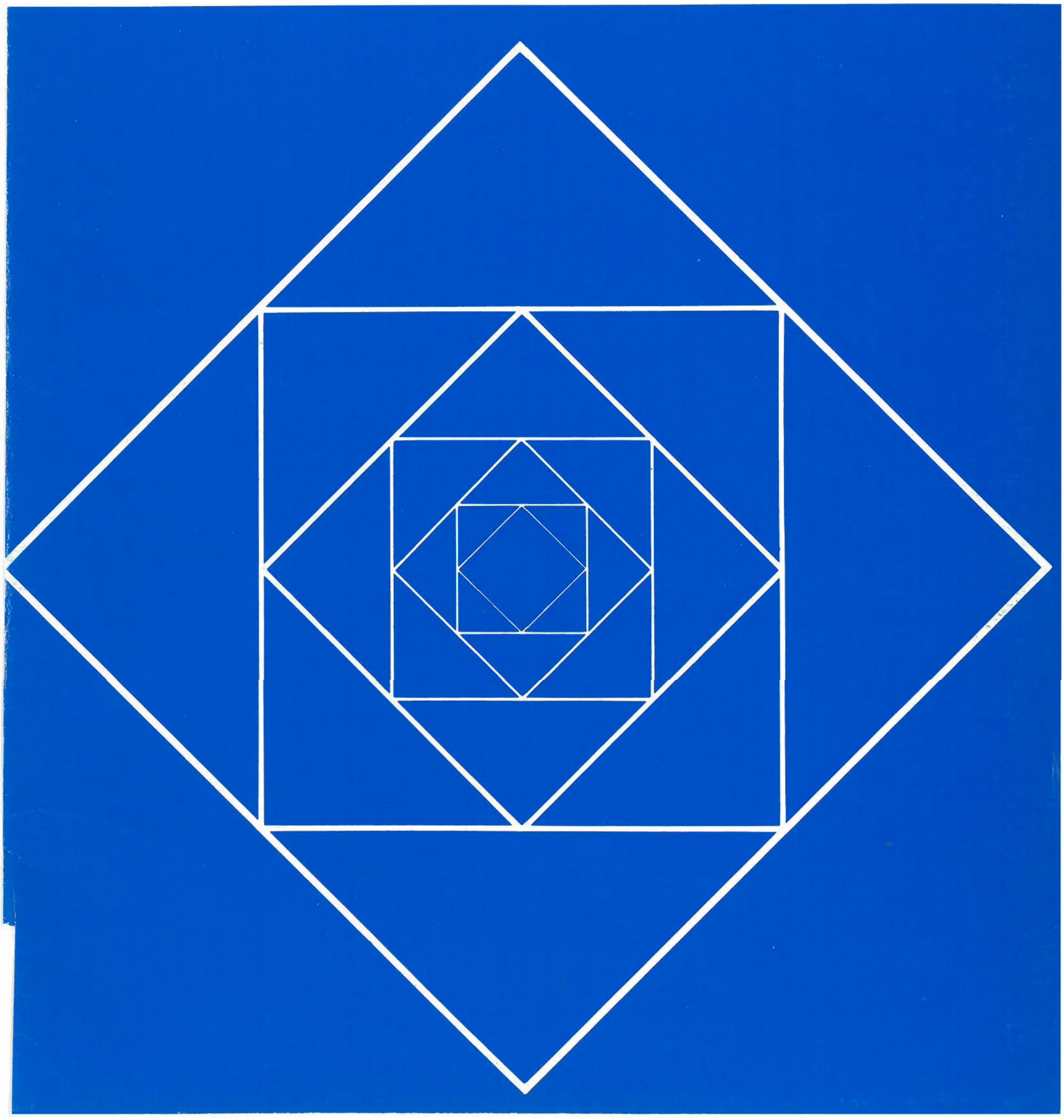
Milan Scientific Center

SCM-0001
January 1991

AGXMM - A C Library for Extended Memory Manager User's Manual

IBM Internal Use Only

A. Bondi, G. Chiozzi



AGXMM - A C Library for Extended Memory Manager User's Manual

by

A. Bondi[1], *G. Chiozzi*[2]

[1] IBM Milan Scientific Center, Milan, Italy
email: bondi@milvmsc

[2] IBM Milan Scientific Center, Milan, Italy
email: chiozzi@milvmsc

Contents

Chapter 1. Preface

Purpose of this function library is to overcome the well known DOS limitations regarding memory management (640 kbytes of memory and 64 kbytes -segment size- max for a single data block). We paid a special attention in order to create a friendly environment in which the user can handle eXtended Memory in a transparent way just as if he was working with conventional files or arrays.

This library is based upon the eXtended Memory Specification (XMS) as provided by Microsoft Corp. and requires an eXtended Memory Manager (XMM) such as HIMEM.SYS.

At the moment, only a small number of functions have been developped, but they already permit a full use of eXtended Memory; they can be divided into:

- basic XMM access functions.
- PseudoFiles functions: to use XMM as a simple but fast file system.
- Virtual Arrays functions: to address large arrays stored in eXtended Memory in a transparent way.

We plan to improve our library in order to satisfy more complex needs. User's suggestions are welcome; you should refer to the following addresses:

Alessandro Bondi: BONDI at MILVMSC; Phone: +39(2)7548.4147

Gianluca Chiozzi: CHIOZZI at MILVMSC; Phone: +39(2)7548.4516

IBM Semea - Centro Ricerca di Milano - 20090 Segrate (MI) - Italy

Chapter 2. AGXMM - A C library for eXtended Memory Manager

2.1 Introduction

This manual[1] is divided in three parts:

- This chapter: a general introduction to AGXMM
- Chapter 2: technical reference.
- Chapter 3: functions reference.

The first part contains general information on how to use the library, such as supported compilers, software and hardware requirements and so on.

Information on how the library is structured and implemented and simple general examples are given in the second chapter, while the third one is a complete list of all the implemented functions.

Several Technical Notes are added throughout the text in order to explain in more detail some topics that may be relevant for those wishing to extend the library. The mere user will not probably need to read them, and the whole manual can be profitably understood skipping them at all.

2.2 Overview

AGXMM has been created in order to put the eXtended Memory installed on 80x86 based personal computers at full disposal of C programs developped under DOS operating system. AGXMM is currently supported for the following compilers:

Compiler	**Compiler ID**
IBM C2 V1.10	IBMC2
Microsoft C	IBMC2
Borland TURBO C V2.0	TURBOC

For these compilers, precompiled libraries are available. However, everybody is welcome to port AGXMM functions to other compilers.

The library is available in two distribution versions:

- Precompiled library
- Source code

[1] This shows how the special characters used in this document look like on your printer (if they are printed at all).

```
Curly braces:                          '{' '}'
Square brackets:                       '[' ']'
Vertical bar  (logical OR in C):       '|'
Tilde         (binary NOT in C):       '¯'
Caret         (binary XOR in C):       '¬'
```

2.3 Software and Hardware requirements.

- Hardware
  - IBM PS/2 with eXtended Memory Expansion Card
- Software
  - An eXtended Memory Manager (such as Microsoft HIMEM.SYS)
  - A C compiler[2] (see above for supported list)

2.4 Using AGXMM

Using AGXMM is simple. Only a few things are needed:

1. Reading and understanding the AGXMM reference manual.
2. Installing AGXMM on your system just to use it.
3. Installing an eXtended Memory Manager (such as Microsoft's HIMEM.SYS)
4. Defining the Compiler ID symbol.
5. Including the file "agxmm.h" in your C source.
6. Linking the AGXMM library and object files.

The AGXMM include file "agxmm.h" should be included at the end of the other files included. The AGXMM library should be linked as one of the first libraries. The compiler ID must be defined as a preprocessor symbol with the value 1 (usually using the -D option of the compiler).

2.5 Developping AGXMM

Users of AGXMM are encouraged to develop it. The following is needed:

1. One or more of the supported C compilers.
2. Installing AGXMM on your system to develop it.

Whenever a new function is added to AGXMM, the following rules should be obeyed:

- A function prototype must be added to "agxmm.h".
- When new types, structures, constants, etc. appear at the function interface, they must be defined in "agxmm.h".
- All new identifiers (function names, type names, various kinds of preprocessor names) should follow well-thought naming conventions, which are up to the developper. A common practice is to use a unique prefix for the names. The new identifiers must not conflict with any previously defined identifier in the standard libraries for the particular compiler.
- Structures must be independent of whether structure packing is used or not.

The following are good sources to get information about functions already defined:

- C libraries of the most popular compilers (Borland TurboC V2.0+, IBM C/2 V1.1+, IBM C/AIX, Unix System V)
- ANSI C Standard
- IBM SAA CPI C Reference

[2] XMM is necessary in the presence of large memory demand, so the library has been developed to work under large memory model; however it is possible, with small modifications, to use it under different memory models.

2.6 Installing AGXMM

AGXMM comes in two levels of detail:

- As a user archive for a specific compiler.
- As a source archive.

In order to use AGXMM, only the user archive for that compiler is needed. In order to develop AGXMM, only the source archive is needed. To install the source archive, download it into any subdirectory.

2.7 AGXMM Files

We think it useful to provide a general list of the source files that constitute the library.

Filename	Description	Only source arc.
INCLUDE FILES		
agxmm.h	Main include file	
C SOURCE FILES		
aglibeng.c	English language messages	
agxmm.c	Basic XMM functions. To be compiled ONLY in TURBOC for C2 link agxmm.obj	X
agxmmfil.c	Pseudo-Files functions	X
agxmmvar.c	Virtual arrays functions	X
calcbyte.c	Used by agxmm.obj to be compiler independent	X
C EXAMPLE FILES		
agxmftst.c	Example program to test Pseudo File functions	
agxmmtst.c	Example program to test library	
agxmvtst.c	Example program to test virtual arrays	
LIBRARIES		
agxmmc2.lib	Functions library for IBM C2 1.1	
agxmmtc.lib	Functions library for TURBOC 2.0	
OBJECT FILES		
agxmm.obj	Basic XMM functions .obj file to be linked when using C2 compiler	X
TC PROJECT FILES		
agxmmtst.prj	TURBOC project file for agxmmtst.c compiled from library sources	X
BATCH FILES		
aglibc2.bat	To make library from sources with IBM C2	X
aglibtc.bat	To make library from sources with TURBOC	X
agxmmtst.bat	To compile agxmmtst.c program with IBM C2 from library sources	X
OTHER FILES		
batcoll.txt	Collection of example batch and TURBCC project files	

Chapter 3. AGXMM Technical Reference

3.1 EXTENDED MEMORY SPECIFICATION

The increasing demand of memory, even for PC application programs, makes the 640Kb limit of MS-DOS too narrow.

The eXtended Memory Specification (XMS) [1] defines a software interface that permits to manage memory areas not usually available to MS-DOS operating system.

Technical note 1:

XMS allows applications to allocate, resize and free three kinds of memory blocks:

- Extended Memory Blocks (EMB), that is physical memory placed at addresses above 1088Kb;
- High Memory Area (HMA), that is physical memory placed between 1024Kb and 1088Kb.

  The addressability of this area stems from how the MS-DOS manages memory. HMA corresponds to the 64Kb segment addressed by the value 0xFFFF of the segment register. There exists a bus line, named A20, that controls how these addresses are physically mapped on memory; XMS provides a hardware independent management of this line. By default, line A20 is disabled so that HMA is not used as such, but is remapped onto the first memory segment (locations 0000:0000 to 0000:FFFF); otherwise, the area is seen as a further physical segment above 1Mb.

  Some applications assume that memory wraps around the 1Mb limit (A20 disabled), so the programmer should take care to let it down after use.
- Upper Memory Blocks (UMB), that is physical memory placed between 640Kb and 1024Kb.

Up to now, this library provides functions for the support of EMB only.

3.1.1 XMS Device Driver

In order to use XMS on your computer you must first install the proper device driver, that is an eXtended Memory Manager (XMM). As an example, we will describe the procedure to install the standard Microsoft prototype (HIMEM.SYS):

- Edit your CONFIG.SYS file
- Add the line

  DEVICE = *path*\HIMEM.SYS

  where *path* indicates where the driver is placed
- IPL your system

Technical note 2:

HIMEM.SYS supports the following options:

/HMAMIN = n	that specifies the minimum size of HMA that a program can allocate (0-63Kb, def=0).
/NUMHANDLES = n	that specifies the maximum number of active XMS handles (0-128, def=32).

Warning: HIMEM.SYS may be in conflict with other devices previously installed on your system; if something goes wrong, check your device drivers installation manuals, with a special attention to other memory managers, such as Expanded Memory Manager. Only one XMM is allowed at the same time.

3.1.2 Library structure

The basic XMS functions for the EMB management constitute the first layer of AGXMM library. There are in fact two more function layers that have been designed in order to make the use of eXtended Memory easy.

One of them defines the concept of **Pseudo-File**, that allow to set up a simple file system; by calling functions with the same sintax of *read, write, seek* and *tell*, one is able to access eXtended Memory as if he was using standard UNIX streams.

The other one provides a friendly programming interface, introducing the concept of **Virtual Arrays.** Large amounts of data, up to 8Mb (by far greater than the 64Kb limit imposed by MS-DOS), can be organized as a single array in eXtended Memory and accessed in a transparent way from any C program.

What follows is a brief description of the library structure of the implemented features in each layer:

1. Basic XMS functions:
   - Initialization of XMS.
   - Getting information on XMS status.
   - Allocation, reallocation and release of an EMB.
   - Copy of memory areas from conventional to eXtended Memory in any possible way:

     eXtended Memory ←→ eXtended Memory

     eXtended Memory ←→ Conventional Memory

     Conventional Memory ←→ Conventional Memory
2. Pseudo-Files support:
   - Pseudo-Files system initialization.
   - Opening and closing a Pseudo-File.
   - Random access to Pseudo-Files records.
3. Virtual Arrays support:
   - Virtual Arrays system initialization.
   - Virtual Arrays data exchange with disk files.
   - Virtual Arrays elements access.

Besides these subjects, the library provides functions for error management.

3.2 Basic functions layer

3.2.1 Initializing XMM

To use XMM, the first thing to do is to check if the hardware and the driver are properly set up. This can be accomplished by calling the function

XMMAccess()	that also prepares the system to access eXtended Memory.

In case of error, the global variable **XMMError** is set to the Error Code (EC) value. Moreover, XMMAccess() returns -1 on error, 0 if everything is OK. For more information on error handling techniques, refer to the section about error management.

Technical note 3:

> XMMAccess() first checks if the XMM driver is installed by generating a proper software interrupt. If the check is successful, the interrupt routine returns the entry point address of the function that will be called to communicate with XMS. This address is stored in the function pointer
>
> **XMM_call()**.
>
> Every low level XMM function call will consist of:
>
> - filling microprocessor registers with proper values; in general the AH register will be set to the XMM function op. code.
> - calling XMM_call().
> - getting return values from registers; usually EC is put in register BL.

The programmer will never need to make use of XMM_call(), since higher level functions are provided by the library.

3.2.2 Getting information

After initializing eXtended Memory functions, one may wish to get several information about the installed XMS.

The library provides some functions that permit to query the eXtended Memory Manager.

XMMVersion() returns the version number of the installed eXtended Memory Specification and of the eXtended Memory Manager along with the status flag of HMA, i.e. if it is available or not. In order to know available eXtended Memory on the system, one should use

XMMcoreleft() that, having the same syntax of standard DOS **coreleft()**, tells how many bytes of free eXtended Memory remain on your PC. It is important to stress that this memory can be fragmented; the maximum single block size is obtained by

XMMmaxblock() call.

Information on any allocated EXtended Memory Block (**EMB**) are given by the function

XMMHandleInfo() once you have passed to it the identifier of the EMB you are interested in; this identifier, called **handle**, is returned upon allocation of the block by the proper function (see below). The information given by this function concern the size of the chosen EMB, its lock count and the number of remaining free block handles.

Technical note 4:

The version numbers of XMM and XMS returned by XMMVersion() are in Binary Coded Decimal format (**BCD**). With this convention the hex number 0x456 corresponds to decimal 4.65.

Technical note 5:

The lock count should be 0 in any case with this version of AGXMM In this way an EMB can be freely moved by XMM. To enable moving, the XMM provides functions (not currently implemented) that set the lock count. This could be useful for programs that address directly eXtended Memory and use A20 line.

3.2.3 Allocating and freeing EXtended Memory Blocks

Three basic functions deal with memory allocation. These are modeled on the standard C functions alloc(), realloc() and free().

XMMAlloc() allocates an EMB taking its size as a parameter and rounding it to the nearest upper kilobyte. Notice that in this way you cannot allocate less than 1Kb of eXtended Memory, unless you wish to reserve 0 bytes, which is a permitted size; the integer value returned by this function is the EMB handle.

An EMB can be resized by calling

XMMReAlloc() and passing to it the proper handle. In the same way as above, the new size is 1Kb rounded.

Finally

XMMFree() permits to deallocate the eXtended memory previously reserved. It is very important to XMMFree() all used EMB before exiting the application, since otherwise this memory will be lost until a new IPL.

Technical note 6:

The most interesting feature of eXtended Memory is that it can be allocated whenever an application needs to do that. It is then possible to create functions that resemble very much the C language functions that manage memory allocation. For example, XMMReAlloc() can shift the position of an EMB if its resizing makes it necessary.

3.2.4 Data exchange between conventional and eXtended Memory

Perhaps the function that the user will employ more often is

XMMCopy() that transfers memory areas from conventional and eXtended Memory in any possible way. As a parameter, it accepts a structure that contains all the information about the source and the destination areas, which are identified by their handle. The structure that contains all the required information is called **XMMCOPYBLOCK** and is defined in agxmm.h by means of the following typedef declaration:

```
typedef struct                /* XMM memory block control structure  */
        {
         unsigned long bsize;   /* XMM mem. blck size (byte) ; even    */
         int src_Handle;        /* XMM memory blck source Handle       */
         union XMMoff src_off;  /* src offs or conv.mem.addr. if Hnd=0 */
         int dest_Handle;       /* XMM memory blck destination Handle  */
         union XMMoff dest_off; /* dest offs or conv.mem.addr if Hnd=0 */
        }
         XMMCOPYBLOCK;
```

The meaning of each parameters is as follows:

bsize it specifies the size of the memory area to be copied; it must be even and, for speed optimization on 80386, should be double-word aligned;

src_Handle it specifies the source EMB handle obtained from a previous XMMAlloc(); a zero value for this parameter specifies that the source area is in conventional memory;

src_off the interpretation of this parameter depends on the value of the previous one; if src_Handle contains a non zero handle number, src_off specifies an offset from the first byte of the corresponding EMB; in the case in which src_Handle is zero, src_off is a pointer to the starting location in conventional memory. To deal with this double meaning in a clean way, src_off is defined by a proper union declaration, **XMMoff**, in agxmm.h:

```
union   XMMoff                /*  XMM mem. block offset decl.      */
        {
         unsigned long offset; /*  XMM block offset                  */
         char *address;        /*  conventional memory address       */
        };
```

dest_Handle it specifies the destination EMB handle obtained from a previous XMMAlloc(); the same as for src_Handle applies;

dest_off analogous to src_off declaration.

Analogously to standard C functions of this type, it is not guaranteed that the transfer is successful if the two areas overlap, unless the source address is less than the destination one.

3.3 Pseudo-Files

3.3.1 Introduction

In order to store and access data in eXtended Memory by using XMM standard functions, the only way is to copy data buffers to and from eXtended and conventional memory. This is not a flexible procedure: a possible way to improve eXtended Memory accessibility is to set up a very simple kind of file system, whose files are stored in EMBs and can be opened, written, read and closed. Such files have been called Pseudo-Files in order to remember that they are not true files, even though the programmer can use functions with the same syntax and behaviour of the ordinary file access ones. These functions constitute the second layer of AGXMM library; resting on the basic one, they make it possible to use eXtended Memory as a very fast hard disk.

3.3.2 Pseudo-Files System initialization

After initializing XMM, by means of a call to XMMAccess(), the first step in using Pseudo-Files is to initialize the Pseudo-File System (**PFS**) by calling

XMM_files_init(). The maximum number of Pseudo-Files that can stay open at the same time is fixed by the global variable **XMM_max_files** whose default is set to 10 in the current implementation. To override this default, a new value can be assigned to this variable ***before*** the call to XMM_files_init(). Notice that this latter function can be called only once in a single program.

3.3.3 Access to Pseudo-Files

Each time the programmer needs to access a new Pseudo-File he has to open it using the function

XMMOpen(). which first verifies if there are available Pseudo-Files handles and if there is enough eXtended Memory to hold the file. It has been decided to force the user to declare the maximum size of the Pseudo-File as a parameter of XMMOpen(): in this way, all the requested eXtended Memory can be allocated by the PFS via a single call to XMMAlloc() and with no need to use XMMReAlloc() function, which is not implemented by all XMMs. Due to this choice, the syntax of XMMOpen() is sligthly different from the corresponding standard open() function, having one additional parameter. XMMOpen() returns an integer number that represents the Pseudo-File handle to be used for each access to it.

The functions that perform read and write operations over a Pseudo-File are

XMMRead() and

XMMWrite(); they are used in the same way as ordinary read() and write(). The same applies to

XMMTell() and

XMMSeek() with respect to tell() and seek() functions that are used to retrieve and set the logical pointer position in a file.

Once a Pseudo-File is no more needed, it can be closed by means of

XMMClose(). Be aware that, once a Pseudo-File is closed, all data stored in it are definitively lost and the eXtended Memory is released to XMM. The implementation of functions that save and load data from a standard disk file and Pseudo-Files (and vice-versa) is left to future developments; up to now, the user has to take care of these operations.

What follows is a simple example of a program that uses Pseudo Files. It includes system initialization and Pseudo File open and close procedures. It can be used as a skeleton to develop more complex programs.

```
#include <stdio.h>
#include <string.h>

#include "agxmm.h"

void main()
 {
  int PFile_Hand = 0;
  char *string  = "XMM Pseudo Files test string";
  char *string2 = "                              ";
  char *fname = "Filename.PF";
```

```
#include <stdio.h>
#include <string.h>

#include "agxmm.h"

void main()
 {
  int PFile_Hand = 0;
  char *fname = "Filename.PF";

  /* Tests to see if XMM is installed */
  if((XMMAccess()) != NOERROR)
   {
    printf("Unable to Access XMS\n");
    exit(1);
   }

   /* Init pseudofiles manager */
   if( XMM_files_init() != NOERROR )
    {
     printf( XMMErrorMsg( XMMError));
     exit(1);
    }

   /* Opens a pseudofile */
   if( (PFile_Hand = XMMOpen(fname, 0, 2050)) == -1 )
    {
     printf( XMMErrorMsg( XMMError));
     exit(1);
    }


   /* The array element to be accessed is XMM_fchain(PFile_Hand) */
   printf("CURRENT PSEUDO_FILE STATUS:\n");
   printf("    PFile name        :    %s\n", XMM_fchainflPFile_Hand".name );
   printf("    EMB Handle        :    %d\n",
                                       XMM_fchainflPFile_Hand".Handle );
   printf("    Curr. Pos.        :    %ld\n",
                                       XMM_fchainflPFile_Hand".offset );
   printf("    max PFile size    :    %ld Kbytes\n",
                                       XMM_fchainflPFile_Hand".bsize );
   printf("    actual PFile size:     %ld\n",
                                    XMM_fchainflPFile_Hand".filesize );
   printf("    PFile mode        :    %d\n",
                                       XMM_fchainflPFile_Hand".flags );

   /* Closes the pseudo file after use */
   if( XMMClose(PFile_Hand) != 0 )
    {
     printf( XMMErrorMsg( XMMError));
     exit(1);
    }
   printf("Closed Pseudo File\n");

  exit(0);
```

3.4 Virtual Arrays

3.4.1 Introduction

The aim we had in proposing the use of Virtual Arrays was to overcome the well known limit of 64Kb arrays for C programmers under DOS operating system; at the same time, Virtual Arrays provide a simple way to use the eXtended Memory for storing data.

The idea here implemented comes from a paper by M.Tichenor [2] who describes a similar system that uses disk space to store Virtual Array elements. With the aid of this technique, one can manage large arrays of data and access them by simple reference using predefined alias and an array index. The system array manager provides an automatic paging memory mechanism that is completely transparent to the user.

Furthermore we think that programs that have been designed without using Virtual Arrays could be modified quite easily in order to take advantage of this technique.

3.4.2 Initializing the Virtual Arrays manager

Since the Virtual Arrays System rests upon eXtended Memory **and** Pseudo Files System, it is first of all necessary to initialize XMM and PFS via subsequent calls to XMMAccess() and XMM_files_init().

3.4.3 Creating Virtual Arrays

The procedure to employ Virtual Arrays can be logically divided into two steps that have to occur in the following order in a user program:

1. the definition of the access macros used to emulate standard array access;
2. the call to the function that actually allocates eXtended Memory necessary for the Virtual array and initializes the related control structures.

#define statements are used to simplify access notations both to array elements and to complex fields within them, such as structure elements. One *#define* is required for each Virtual Arrays used. For the sake of clarity it is perhaps better to begin with the description of the second step letting the first one to the next section. There are three functions that permit the creation of Virtual Arrays according to different needs that the user may have. They are

XMMCreate_v_array(), that creates a Virtual Array and makes it available for use,

XMMInit_v_array(), that performs the same operations but uses data contained in an existing Pseudo File and

XMMLoad_v_array() that loads a file from disk to eXtended Memory and makes it available as a Virtual Array.

Since these functions are very similar to each other we will describe only the the first one, which is probably the most useful (refer to the function reference for a more complete description of all of them). Since the Virtual Array is stored in a Pseudo File, the first parameter of XMMCreate_v_array() is the Pseudo File name that is used to open it. The function calculates also the required size of the Pseudo File on the basis of the number of elements, given by the parameter *elnum*, and of the elements' size, *elsize*[3] Furthermore, the function initializes Virtual array elements, if needed, to the predefined mask *fillchar*. To optimize access performances, the function accepts two additional parameters: the buffer size and the number of elements per segment. The Virtual Array manager access array elements via a paging algorithm: a certain number of elements are stored in a buffer, placed in conventional memory, whose size is defined by the parameter *bsize*; moreover, this buffer is divided into logical segments that contain a number of consecutive elements of the virtual array, that is given by the parameter *segsize*. Each time an element is referenced, the manager checks if it is already placed in the buffer and, if not, loads it from eXtended Memory, swapping a whole segment; as a consequence, the access of sequential elements takes benefit from large segments, while random access is faster with smaller segments. The buffer size, whose upper limit is 64 Kb, depends mainly on the available amount of conventional memory and does not affect the performances in a sensible way. In order to optimize the use of the buffer memory, *bsize*, *segsize* and *elsize* should be balanced so that *bsize* is the nearest possible to a multiple of the product of *segsize* and *elsize*. XMMCreate_v_array() returns a pointer to the Virtual Array Control Structure of type

[3] As previously noticed, the function XMMReAlloc is not supported by all the XMM, so it has been chosen to avoid its use; as a consequence, the Pseudo File that holds the Virtual Array is of fixed dimension; by using dynamic size Pseudo Files virtual arrays of dynamic dimension could be easily implemented.

XMM_VACS which contains all the relevant information to be used by access functions. This structure will be described later on only in a technical note since the programmer should never use explicitly data contained in it.

3.4.4 Access notation

The access macros are defined in terms of *#define* declarations that involve a call to

XMMAccess_v_array() or

XMMFast_Access_v_array() that are the functions that actually retrieve information from eXtended Memory by using the Virtual Array Control Structure of type XMM_VACS.

XMMFast_Access_v_array() is considerably faster than the first function, but can be used only when all the operations that the user wishes to perform on a Virtual Array (for example loaded from a disk file) are of read only type.

These functions should not be called directly but only used in macros definitions. Let us suppose, as a first simple example, that we need to access the array of integers **BigArray**: if it was a standard array it would be sufficient to write a line like this:

```
BigArray[item] = 5;
```

In order to perform the same operation with a similar syntax in the case of Virtual Arrays it is necessary to define the following macro:

```
#define     BigArray(i)     (*(int *)XMMAccess_v_array(v_array, i))
```

After the creation of the Virtual Array any of its element can be accessed with a construct like the following one

```
BigArray(item) = 5;
```

Let us now analyze the macro definition. The access functions return a void pointer to the referenced array element. This pointer must be cast to the desired type, in this case a pointer to integer. Then the actual array element is obtained by taking the content of this pointer; this notation works well both for read and write operations[4].

```
n = BigArray(item);                /*  read  access     */

BigArray(item) = 5;                /*  write access     */
```

XMMAccess_v_array() requires, as the first parameter, the pointer to the Virtual Array Control Structure of type XMM_VACS that identifies the required Virtual Array. This pointer is set by the call to any of the creation functions described above, that must be performed before any access to the Virtual Array. The second parameter is plainly the index number of the searched element.

[4] Pay attention that write operations are not permitted when using XMMFast_Access_v_array() because of the particular way in which eXtended Memory is managed.

What follows is an example of the whole procedure:

```
#include <stdio.h>
#include agxmm.h

/*     Access macro definition       */
#define BigArray(i)    (*(int *)XMMAccess_v_array(v_array, i))

main()
{
 XMM_VACS *v_array;           /* Virtual Array Control Structure */
                              /*       pointer declaration       */
 long arrsize = 50000;        /* desired v_array size            */
 int i=0, n;


  ..... XMM and PFS  initializations .....


 /* creates the virtual array setting element size */
 /* the size of item structure and setting the     */
 /* initialization value to i;                     */
 /* returns pointer to XMM_VACS structure v_array  */

 if( (v_array = XMMCreate_v_array("PFname", arrsize, sizeof(int),
                        (char *)&i, 0, 0) == NULL)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }


 for( i=0; i<arrsize; i++)
  {
   /*  access the i-th element assigning a value to it.... */
   BigArray(i) = i;

   /*  ....... or retrieving a value from it.... */
   n = BigArray(i);
  }

  .........  other operations ..........

}      /*   end   main()    */
```

In a more complex case the user may desire to access arrays of structures and fields within them. He can use the following example:

```
#include <stdio.h>
#include agxmm.h

/****************************************/
/*  Array Elements Structure typedef    */
/****************************************/

typedef struct
        {
        long v_item, v_qty;
        char v_desc[24];
        }
         items;

/*   Access Macros definitions: single structure....    */
#define XMM_VREC(i)    ((items *)XMMAccess_v_array(item_array, i))

/*  ...... and various fields in them                    */
#define item(i)        XMM_VREC(i)->v_item
#define qty(i)         XMM_VREC(i)->v_qty
#define desc(i)        XMM_VREC(i)->v_desc

main()
{
 XMM_VACS *item_array;
 unsigned long i;
 items fillchar;


 /*     single structure  prototype initialization     */
 fillchar.v_item = -1;
 fillchar.v_qty  = -1;
 strcpy(fillchar.v_desc, "Null_array_item________");


  ..... XMM and PFS  initializations .....


  ..... Virtual Array creation  (returns XMM_VACS pointer).......



 /* fills in 50 array items                          */

 for( i=0 ; i<50 ; i++)
 {
  item(i)  = i+1;
  qty(i)   = 0;
  sprintf(desc(i), "item # %ld", i+1);
 }


}   /*  end main()  */
```

It is also possible to reference multidimensional arrays as shown in the following example:

```
#include <stdio.h>
#include agxmm.h

/*****************************************/
/*  Array Elements Structure typedef    */
/*****************************************/

typedef struct
        {
        int rowitem[100];
        }
         matrix;
```

```
/*   Access Macros definitions: get whole column....     */
#define XMM_VREC(y)    ((matrix *)XMMAccess_v_array(item_array, y))

/*  ...... and corresponding row                         */
#define mat(x,y)       XMM_VREC(y)->rowitem[x]

main()
{
 XMM_VACS *item_array;
 unsigned long i;


  ..... XMM and PFS  initializations .....


  ..... Virtual Array creation  (returns XMM_VACS pointer).......



 /* fills in a few elements in the matrix                   */

 for( i=0 ; i<50 ; i++)     mat(i,i) = i+1;


}   /*  end main()  */
```

There are a few things to be noticed about this last example:

1. each row in the matrix is defined as a single element in the Virtual Array item;
2. consequently a row cannot be extended beyond a certain limit whose upper value is 64Kb, the 8086 addressing segment size. The actual limit is indeed lower, since it depends on the parameters used for the creation of the Virtual Array.

3.4.5 Closing Virtual Arrays

When the user no longer needs to reference the elements of a Virtual Array he can close it by calling

XMMClose_v_array() This function does not actually discard data contained in the Virtual Array but rather stores them in the related Pseudo File, whose handle is returned, and frees that region in conventional memory needed for Virtual Array management.

The programmer has to take care of the use of the data now contained in the Pseudo File: he can, for example, save them on disk, or discard them by simply closing also the Pseudo File.

3.4.6 Cautions

There are a few remarks to keep in mind when using Virtual Arrays. Although the access syntax is quite similar to the standard one, the paging algorithm can generate some collateral effects. Since only a small number of elements is in conventional memory at access time and these elements are loaded using a hashing scheme

- there is no warranty that logical consecutive elements are placed in consecutive memory locations in the buffer;
- when accessed, different elements with the same hashing key are loaded from eXtended Memory to the same buffer address.

These two facts lead to some disappointing consequences. For example, pointer autoincrementation does not lead, in general, to the next item of the Virtual Array. Moreover, contemporary access to different elements can cause buffer collision; for example, memory copy functions, such as *strcpy*, used to copy data directly from one array element to another, are unreliable because the two may occupy the same buffer locations. In particular

```
strcpy(StringArray(n), StringArray(m));
```

will not work if *n* and *m* have the same hashing keys. Troubles can be avoided by using a temporary string buffer as follows:

```
strcpy(temp_string, StringArray(m));
strcpy(StringArray(n), temp_string);
```

On the other hand, statements like

```
BigArray(n) = BigArray(m) + 1;
```

work correctly because memory copying is not involved and the compiler calculates the assignment value before the addresses for the assignment.

The choice of using a buffered access also forbids the employement of such things as in-memory sort utilities like *qsort*

Finally, take care not to overrun the end of the array elements when copying data into them since this could spoil some data management information.

Technical note 8:

The management of Virtual Arrays has required the implementation of a paging algorithm that transfers areas of eXtended Memory into conventional memory whenever this is necessary in order to satisfy user's requests of referencing an element in a Virtual Array. Things have been arranged so that the entire procedure is hidden from the user; this is how the main purpose of handling in a transparent way very large amounts of data under DOS operating system is attained.

For the sake of completeness, we will now outline this algorithm, though it is clear that normally one does not have to care about the operation of memory management.

The algorithm can be thought of as a simple kind of hashing function that associates to any element of a Virtual Array a location in a buffer placed in conventional memory. This buffer, that we call **data_buffer** and whose dimension **bsize** can be set by the user, contains the data that have to be transferred from eXtended Memory in order to permit Virtual Array element access. It is at the basis of the procedure, along with the **index_buffer** that contains information about the eXtended Memory areas present in data_buffer; both are allocated by any one of the Virtual Array creation functions that fulfill also other fundamental tasks. Among them, there is the logical operation of dividing the Virtual Array into segments whose size can be chosen by the user if he sets the proper parameter **segsize** when calling such a function. The segmentation of a Virtual Array has a correspondence in the logical segmentation of the data_buffer that of course can contain much less segments.

A **segment** represents the amount of memory which can be transferred back and forth between eXtended and conventional memory whenever it is necessary. In order to keep track of which segments of the Virtual Array are present in data_buffer at any given moment, the index_buffer is built as a vector of integers having the following characteristics:

- its dimension is equal to the total number of segments that can be contained in the data_buffer
- its index identifies the segment number in data_buffer
- each one of its items contains two kinds of information
  1. the segment number in Virtual Array is stored in the first 15 bits
  2. the highest bit is employed as a flag in order to indicate if any element in that segment has been changed by the user

When a new Virtual Array is created, it is necessary to open the Pseudo File that will contain it, to determine the segmentation in it and the dimension of data_buffer, to allocate memory for the data_buffer and the index_buffer and to initialize index_buffer and data_buffer elements.

Once the the Virtual Array segmentation and the data_buffer dimension are given, it is possible to outline how the hashing transformation works in order to assign a location in conventional memory to the n-th element of the Virtual Array. All we have to do is to compute the following quantities:

- to which segment **v_seg** of the Virtual Array its n-th element belongs;
- which will be the offset of the element in data_buffer, in terms of element size units;
- once the offset is known, which will be the segment **d_seg** in data_buffer into which the n-th element will fall.

The second parameter gives the new location of the n-th element in conventional memory, while the first and the last one identify which are the segments that have to be swapped.

An example may better explain how these values are evaluated. Let us consider the array element **elind** of size **elsize**: its segment in the Virtual Array is computed by the following expression

```
v_seg = (int)(elind / segsize);
```

while its offset in the data_buffer, which is just the hashing key, is given by

```
b_off = (int)(elind % ((bsize/(elsize*segsize))*segsize));
```

where all the operations have to be taken in the order given by parentheses, paying attention to the fact that only integer operations are involved, and the division could cause rounding effects. Finally, the segment number in data_buffer is calculated as follows

```
bseg_off = b_off / segsize;
```

If the n-th element is referenced by the user, and if its segment is not already present in data_buffer, v_seg will be placed at d_seg in conventional memory and the corresponding item of the index_buffer will be updated.

The previous content of the segment d_seg in data_buffer can be possibly stored back in eXtended Memory, but only if some of its elements have been changed by the user. This condition is actually controlled by the function XMMAccess_v_array() but not by XMMFast_Access_v_array(), which are the functions that implement the memory management and allow Virtual Array elements reference; furthermore, when closing a Virtual Array, XMMClose_v_array() checks which are the data_buffer segments that are to be saved in eXtended Memory. All these functions inspect the status bit of index_buffer items which is risen whenever an element in the corresponding data_buffer segment has been changed. The mechanism used to set this status bit works as follows: each time an array element is accessed, a copy of it is stored in a safe place; at the next access to an array element (the same as before or any other one), a comparison is performed between the copy and the current value of the last accessed element; if they are not equal, this means that a write operation has occurred on that array item and the corresponding segment status bit must be risen. This test is done one access request later than the reference operation that modified the element, because element processing is done on return of the XMMAccess_v_array() and it is impossible to know in advance if the program will perform a write or read only access. The only other way would be to save always any accessed segment each time it must be swapped out of buffer without worrying about modifications, but this would decrease sensibly the access speed performances.

All the information needed to manage memory swapping in this way is contained in a Virtual Array Control Structure, whose type is defined as XMM_VACS, which is allocated and properly initialized by any one of the Virtual Array creation functions, such as XMMCreate_v_array(). The related typedef declaration is as follows:

```
typedef struct              /* XMM virtual array control structure */
  {
   int PFile_Handle;        /* PseudoFile handle                 */
   unsigned long elnum;     /* # of array elements in file       */
   int elsize;              /* # of bytes per element            */
   int  *index_buffer;      /* table of data address             */
   char *data_buffer;       /* table of data                     */
   int dbuf_size;           /* # of elements in buffer           */
   int dseg_size;           /* # of segments in buffer           */
   int lastbel;             /* Last accessed element buffer #    */
   char *lastel;            /* Pointer to Internal copy of last  */
                            /* accessed element                  */
   unsigned long bsize;     /* Max Data Buffer Size in byte      */
   unsigned long segsize;   /* # of elements per segment         */

  }
   XMM_VACS;
```

The meaning of each parameter appearing in this definition is summarized in the following list:

PFile_Handle	it is the handle of the Pseudo-File that contain the Virtual Array;
elnum	it specifies the total number of elements in the Virtual Array;
elsize	it is the size, in bytes, of each element of the Virtual Array;

index_buffer	it is the address of the buffer that contains the information about the Virtual Array segments that are placed in conventional memory at any time;
data_buffer	it is the address of the conventional memory buffer that contains Virtual Array data;
dbuf_size	it is the total number of elements that can be stored in the data_buffer; it is used in order to compute which would be the offset of a Virtual Array element in the data_buffer;
dseg_size	it is the total number of segments that can be stored in the data_buffer;
lastbel	it is the position in the data_buffer (expressed as an offset) of the last accessed element;
lastel	it is the pointer to the internal copy of the last accessed element; it is used to test if that element has been changed, and hence to see if it is necessary to rise the status bit.
bsize	it is the maximum size of the data_buffer in bytes;
segsize	it is the number of Virtual Array elements that can be stored in each segment; it is used to compute to which segment in data_buffer a Virtual Array element will belong.

As it has been already stressed, the normal use of this structure does not embrace the direct access to these information, since otherwise the transparency of Virtual Array elements referencing would be lost. What is needed is just to store the pointer to the XMM_VACS structure, which is returned by creation functions, and to pass it as a parameter to the access functions when defining the access macros. Refer to the example in order to see how this is done.

3.4.7 Error management for Virtual Arrays

The error management is consistent with that of previous layers, whenever this is possible: the global variable *XMMError* is set by every function and can be read to test error conditions. However, there are situations in which this is not profitable; in particular, during Virtual Array elements access it is not possible to test *XMMError* without loosing the appearance of using standard arrays. To cope with this difficulty, the user can define his own error management function that takes care of communicating error conditions and messages. The global variable

void (*XMM_Varr_error)(int ErrCode) is set to NULL by default: this means no error communication. The user can change the value of this pointer by assigning to it the address of his own error function; at any time an error occurs while accessing Virtual Arrays, the system will call this function passing to it, as a parameter, the corresponding error code stored in XMMError. This error code will be processed as needed, possibly obtaining from the system the corresponding error message string by means of a call to XMMErrorMsg(XMMError).

What follows is a simple example of this procedure.

```
#include <stdio.h>
#include agxmm.h

main()
{


 /*  setting pointer to error function    */
 XMM_Varr_error = MyErrorFunc;

 if( XMMAccess() != 0)
 {
  printf("Unable to access XMS\n");
  exit(1);
 }

 if(XMM_files_init() != NOERROR)
 {
  printf("Unable to access XMS PseudoFiles\n");
  exit(1);
 }
 ......   Virtual Array access.......



}         /* end main */


void MyErrorFunc( int ErrCode)
  {
    printf( "%s\n", XMMErrorMsg( ErrCode));
    exit(1);
  }
```

Chapter 4. AGXMM Function Reference

4.1 XMM_files_init

Name:

XMM_files_init - Initializes the Pseudo File System

Declaration:

```
int XMM_files_init( void );
```

Synopsis:

```
#include "agxmm.h"

int EC;

EC = XMM_files_init();
```

Description:

The function sets all the handles of the Pseudo File System to UNUSED; in this way it makes them available. This function can be called only once in a single program. It also sets XMMError.

Return value:

The function returns the error code stored in XMMError.

Example:

```
#include "agxmm.h"

/* Init pseudofiles manager */

if( XMM_files_init() != NOERROR )
 {
  printf( XMMErrorMsg( XMMError));
  exit(1);
 }

       ------    Pseudo Files  Operations   -----
```

Related Functions:

4.2 XMMAccess

Name:

XMMAccess - Initializes and accesses eXtended Memory Manager

Declaration:

```
int XMMAccess(void)
```

Synopsis:

```
#include "agxmm.h"

int RetCode;

RetCode = XMMAccess( );
```

Description:

The function checks if the hardware and the XMM are properly set up and also prepares the system to access eXtended Memory. In case of error, the global variable XMMError is set to the error code value.

Return value:

The function returns the error code value stored in XMMError (NOERROR if everything is OK).

Example:

The following example shows a typical call to XMMAccess() before accessing any other XMM function

```
#include "agxmm.h"

  if((XMMAccess()) != NOERROR)
   {
    printf("Unable to Access XMS\n");
    exit(1);
   }
```

Related Functions:

XMMInstalled()

4.3 XMMAccess_v_array

Name:

XMMAccess_v_array - Accesses a Virtual Array element

Declaration:

```
void *XMMAccess_v_array( XMM_VACS *vacs, unsigned long elind )
```

Synopsis:

```
#include "agxmm.h"

unsigned long elind;          /* index of referenced element     */
XMM_VACS vacs;                /* Virtual Array control structure */
<desired type> content;       /* store the referenced element    */

content = (<desired type> *)XMMAccess_v_array( &vacs, elind);
```

Description:

This function is to be used in the declaration of the macros that are used to emulate standard array access. The function accesses a Virtual Array element implementing a paging algorithm that manages the swapping of memory areas between conventional and eXtended Memory. First of all, the function checks if the last accessed element has been changed or not; if yes, it rises the status bit of the corresponding logical segment in the conventional memory buffer allocated by XMMCreate_v_array(). Then, it performs a test to verify if the referenced element is already loaded in the buffer. In doing this operation, XMMAccess_v_array() uses the information stored in the Virtual Array Control Structure of type XMM_VACS. If the element is not placed in any segment, the function looks for it in the Virtual Array, identifies its segment and transfers it entirely in the buffer. In this way a segment already in the buffer is replaced after checking its status bit: if it is high, the segment is copied back in eXtended Memory before replacement. The user must take care of referencing an element that does not exceed the maximum number of elements in the Virtual Array, declared when calling the function XMMCreate_v_array().

Return value:

The function returns NULL on error; the address in the buffer of the accessed element otherwise. The function sets XMMError.

Example:

We report here the simple case of a Virtual Array of integers. Refer to chapter 2 for more complex examples.

```
#include <stdio.h>
#include agxmm.h

/*     Access macro definition       */
#define BigArray(i)    (*(int *)XMMAccess_v_array(v_array, i))

main()
{
 XMM_VACS *v_array;
```

```
    int i=0;

     ..... XMM and PFS  initializations .....


     ..... Virtual Array creation  (returns v_array pointer).......

    /*  access the i-th element assigning a value to it.... */
    BigArray(5) = i;

    /*  ....... or retrieving a value from it.... */
    i = BigArray(5)


  }       /*   end   main()   */
```

Related Functions:

XMMFast_Access_v_array()

4.4 XMMAlloc

Name:

XMMAlloc - Allocates XMM memory, rounding up to Kbyte

Declaration:

```
int XMMAlloc( unsigned long sizebytes);
```

Synopsis:

```
#include "agxmm.h"

unsigned long size;           /* number of bytes to be allocated */
int handle;                   /* XMM handle number               */

handle = XMMAlloc( size);
```

Description:

The function allocates an EXtended Memory Block (EMB). The eXtended Memory Manager reserves memory in multiples of Kbytes, so the function rounds up to the lower number of Kbytes necessary to store the requested number of bytes. The prototype requires the size in bytes to be functionally equivalent to malloc(). The number of blocks that can be allocated depends on your XMM: for example, HIMEM.SYS gives by default 32 different handles, but this parameter can be varied.

Return value:

The function returns the EMB handle number or 0 on error. The function sets XMMError.

Example:

The following example shows how to allocate a 3Kb EMB, how to get the corresponding handle and how to manage an error condition:

```
#include "agxmm.h"

int Handle;
unsigned long size = 1024*3;

if( (Handle = XMMAlloc(size)) == 0)
 {
  printf( "%s\n", XMMErrorMsg( XMMError));
  exit(1);
 }
```

Related Functions:

XMMReAlloc(), XMMFree()

4.5 XMMClose

Name:

XMMClose - Closes a Pseudo-File, releasing the related eXtended Memory to the system.

Declaration:

```
int XMMClose( int Pseudo_File_Handle);
```

Synopsis:

```
#include "agxmm.h"

int handle;                    /* XMM handle number                */
int EC;

EC = XMMClose( handle );
```

Description:

The function closes a previously opened Pseudo-File, by freeing the EMB that contained it and by setting its handle to UNUSED. User should take care of saving of data stored in the Pseudo-File before closing it, since otherwise they are completely lost. The function sets XMMError.

Return value:

The function returns the error code stored in XMMError.

Example:

```
#include "agxmm.h"

int PFile_Hand = 0;
char *fname = "Prova";


 ----------    PSEUDO FILE SYSTEM INITIALIZATION    -----------


/* Opens a pseudofile */

if( (PFile_Hand = XMMOpen(fname, 0, 2048)) == -1 )
 {
  printf( XMMErrorMsg( XMMError));
  exit(1);
 }


 ----------    READ/WRITE OPERATIONS    -----------


/* Closes the pseudo file after use */

if( XMMClose(PFile_Hand) != 0 )
```

```
    {
     printf( XMMErrorMsg( XMMError));
     exit(1);
    }
```

Related Functions:

XMMOpen()

4.6 XMMClose_v_array

Name:

XMMClose_v_array - Closes a Virtual Array storing data in its Pseudo File

Declaration:

```
int XMMClose_v_array( XMM_VACS *vacs );
```

Synopsis:

```
#include "agxmm.h"

XMM_VACS vacs;                 /* Virtual Array control structure */
int PF_handle;                 /* store the referenced element    */

PF_handle = XMMClose_v_array( &vacs );
```

Description:

The function performs the following operation. First it checks if the last accessed element has been changed, finds which segments have been modified and store them in the Pseudo File that contained the Virtual Array. Then it frees the conventional memory buffers and the Virtual Array control structure. The function returns the Pseudo File handle, so that the user has to take care of its closure, possibly after some other operations on it.

Return value:

The function returns -1 on error; a Pseudo File handle otherwise. The function sets XMMError.

Example:

```
#include "agxmm.h"

XMM_VACS vacs;                 /* Virtual Array control structure */
int PF_handle;                 /* store the referenced element    */

 .....    Virtual Array initialization and creation  .....

 .......     Virtual Array  operations ...........


 /*  closes virtual array  */

 if( (PFile_Handle = XMMClose_v_array(item_array)) == -1)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }
```

Related Functions:

XMMCreate_v_array(), XMMInit_v_array(), XMMLoad_v_array()

4.7 XMMCopy

Name:

XMMCopy - Copies memory areas from eXtended to conventional memory in any possible way

Declaration:

```
int XMMCopy( XMMCOPYBLOCK *areainfos);
```

Synopsis:

```
#include "agxmm.h"

XMMCOPYBLOCK areainfos;      /* memory areas control structure  */
int RetCode;                 /* Return value                    */

RetCode = XMMCopy( &areainfos );
```

Description:

The function copies an area between eXtended Memory and conventional memory in any possible way depending on the values of the control structure parameters. The structure XMMCOPYBLOCK is described in detail in the Technical Reference chapter.

Return value:

The function returns the Error Code and sets XMMError.

Example:

The following example shows how to copy a string of characters from conventional to eXtended Memory 8 bytes after the beginning of the chosen EMB

```
#include "agxmm.h"

char string[] = "string to be copied";
XMMCOPYBLOCK test;
int Handle;

----------     XMM Initialization  and  Memory Allocation

/*** Copy from conventional memory to XMM ***/

test.bsize = strlen(string)+1;   /* compute block size            */
test.src_Handle = 0;             /* source is in conventional mem. */
test.src_off.address = string;   /* source address in conv. mem.   */
test.dest_Handle = Handle;       /* dest. Handle from XMMAlloc()   */
test.dest_off.offset = 8l;       /* EMB offset from its beginning  */

if( XMMCopy(&test) != NOERROR )       /* XMMCopy()  call and        */
 {                                    /*          Error Management  */
  printf( XMMErrorMsg( XMMError));    /* XMMCopy() sets XMMError    */
  exit(1);
 }
```

```
/*  OK message if successful    */
printf("  '%s' copied from conv mem to XMM\n",stringa);
```

Related Functions:

4.8 XMMcoreleft

Name:

XMMcoreleft - Calculates how many bytes of free eXtended Memory remain on your PC

Declaration:

```
unsigned long XMMcoreleft(void);
```

Synopsis:

```
#include "agxmm.h"

unsigned long Freemem;

Freemem = XMMcoreleft( );
```

Description:

The function calculates the number of available bytes in eXtended Memory. XMM allows to fragment and allocate memory any time that is needed; the number returned by this function refers to all unused eXtended Memory. If memory has been fragmented by previous XMMAlloc(), XMMReAlloc() and XMMFree(), usually it is not true that a unique EMB of that size can be allocated. The maximum allowable single block size is given by the related function XMMmaxblock().

Return value:

The function returns the size of unused eXtended Memory or 0 if an error occurs or if there is no more memory. XMMError is set.

Example:

The following example shows how to get information about eXtended Memory left for use:

```
#include "agxmm.h"

unsigned long memsize;

if( (memsize =XMMcoreleft()) == 0)
 {
  printf( "%s\n", XMMErrorMsg( XMMError));       /* Error management */
  exit(1);
 }
printf("Found  %lu  XMM bytes unused\n", memsize);
```

Related Functions:

XMMmaxblock()

4.9 XMMCreate_v_array

Name:

XMMCreate_v_array - Creates a Virtual Array and makes it available for use

Declaration:

```
XMM_VACS *XMMCreate_v_array(char *PFname, unsigned long elnum,
int elsize, char *filch, unsigned long bsize, unsigned long segsize)
```

Synopsis:

```
#include "agxmm.h"

char PFname[12]      /* name of the Pseudo File containing  */
                               /* the Virtual Array                 */
unsigned long elnum;           /* number of elements in Virtual Array */
int elsize;                    /* size of elements in bytes         */
char filch[10]       /* filling character for empty V.Array */
unsigned long bsize;           /* conv. mem. buffer size in bytes   */
                               /* takes default value if zero       */
unsigned long segsize;         /* number of elements per segment    */
                               /* takes default value if zero       */
XMM_VACS *vacs;                /* Virtual Array Control Structure   */

vacs = XMMCreate_v_array(PFname, elnum, elsize, filch, bsize, segsize);
```

Description:

The function creates a Virtual Array by opening the Pseudo File that will contain it and allocating the Virtual Array Control Structure, the buffers for data and their addresses and initializing all the elements to the predefined mask **filch**. The name of the Pseudo File is used to open it, while **elnum** and **elsize** are needed to compute its size. The user can change the parameters that controls the buffer and segment sizes, **bsize** and **segsize**, in order to improve Virtual Array access performance; setting to zero these variables will cause the system to take default values, **bsize** = 10240 b and **segsize** = 48 elements. The pointer to the Virtual Array Control Structure is returned after allocating space for it.

Return value:

The function returns NULL on error; the pointer to the Virtual Array Control Structure otherwise.

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include "agxmm.h"

/********************************/
/*      Access Macros          */
/********************************/
```

```
#define XMM_VREC(i)     ((items *)XMMAccess_v_array(item_array, i))
#define item(i)         XMM_VREC(i)->v_item
#define qty(i)          XMM_VREC(i)->v_qty
#define desc(i)         XMM_VREC(i)->v_desc

/****************************************/
/*  Array Elements Structure typedef    */
/****************************************/

typedef struct
        {
        long v_item, v_qty;
        char v_desc[24];
        }
         items;

main()
{
 XMM_VACS *item_array;
 unsigned long i;
 int PFile_Handle;
 unsigned long arrsize, bsize, segsize;
 items fillchar;


 fillchar.v_item = -1;
 fillchar.v_qty  = -1;
 strcpy(fillchar.v_desc, "Null_array_item________");

  XMM_Varr_error = MyErrorFunc;

 if( XMMAccess() != 0)
 {
  printf("Unable to access XMS\n");
  exit(1);
 }

 if(XMM_files_init() != NOERROR)
 {
  printf("Unable to access XMS PseudoFiles\n");
  exit(1);
 }
 /* create a virtual array setting element size */
 /* the size of item structure and setting the  */
 /* initialization char to space char           */

 if( (item_array = XMMCreate_v_array("Prova", arrsize, sizeof(items),
                        (char *)&fillchar, bsize, segsize)) == NULL)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }


  ..... Virtual Array  operation   ..........
```

Related Functions:

XMMInit_v_array(), XMMLoad_v_array()

4.10 XMMErrorMsg

Name:

XMMErrorMsg - Returns the pointer to the proper error message

Declaration:

```
char *XMMErrorMsg( int ErrorCode );
```

Synopsis:

```
#include "agxmm.h"

char *string;                /* stores pointer to error message */
int EC;                      /* Error Code value                */

string = XMMErrorMsg( EC );
```

Description:

The function returns the pointer to the error message identified by its argument. At the moment, error messages are available only in english language.

Return value:

The function returns the pointer to the error messsage.

Example:

Refer to the examples for XMMCopy() or XMMcoreleft() that also show simple uses of XMMErrorMsg().

Related Functions:

4.11 XMMFast_Access_v_array

Name:

XMMFast_Access_v_array - Accesses a Virtual Array element without checking previous data modifications

Declaration:

```
void *XMMFast_Access_v_array( XMM_VACS *vacs, unsigned long elind )
```

Synopsis:

```
#include "agxmm.h"

unsigned long elind;          /* index of referenced element     */
XMM_VACS vacs;                /* Virtual Array control structure */
<desired type> content;       /* store the referenced element    */

content = (<desired type> *)XMMFast_Access_v_array( &vacs, elind);
```

Description:

This function is to be used in the declaration of the macros that are employed to emulate standard array access. It has the same syntax of XMMAccess_v_array(), but operates with higher speed.
The function does not perform any check on the possible modification of the data loaded in the buffer, so that the segment swapping takes place without saving the segment that has to be replaced. This function can be used if no data modification is needed, for example when reading data or displaying images. A further speed improvement is obtained via direct call to Basic Layer functions. These are the only differences between this and XMMAccess_v_array() function. Refer to the documentation on that for more information.

Return value:

The function returns NULL on error; the address in the buffer of the accessed element otherwise. The function sets XMMError.

Example:

We report here the simple case of a Virtual Array of integers. Refer to chapter 2 for more complex examples.

```
#include <stdio.h>
#include agxmm.h

/*     Access macro definition       */
#define BigArray(i)    (*(int *)XMMFast_Access_v_array(v_array, i))

main()
{
 XMM_VACS *v_array;
 int i=0;

  ..... XMM and PFS  initializations .....
  ..... Virtual Array creation  (returns v_array pointer).......

 /*  access the i-th element assigning a value to it.... */
 BigArray(5) = i;

 /*  ....... or retrieving a value from it.... */
 i = BigArray(5)


}      /*   end   main()   */
```

Related Functions:

XMMAccess_v_array()

4.12 XMMFree

Name:

XMMFree - Releases previously allocated XMM memory

Declaration:

```
int XMMFree( int Handle );
```

Synopsis:

```
#include "agxmm.h"

int Handle;                    /* XMM handle number            */
int EC;                        /* stores XMMError value        */

EC = XMMFree( Handle );
```

Description:

The function frees an EXtended Memory Block (EMB) previously allocated by XMMAlloc(), which also returned the handle identifier.

Return value:

The function returns the value of XMMError after setting it.

Example:

The Synopsis paragraph reports the simplest call to XMMFree()

Related Functions:

XMMAlloc(), XMMReAlloc()

4.13 XMMHandleInfo

Name:

XMMHandleInfo - Gets EMB handles information

Declaration:

```
int XMMHandleInfo( int Handle, unsigned long *blocksize, int *freehnd,
int *lockmode)
```

Synopsis:

```
#include "agxmm.h"

int Handle;                    /* Handle whose infos are needed    */
unsigned long blocksize;       /* EMB size in bytes                */
int freehnd;                   /* # of remaining free handles      */
int lockmode;                  /* # of locks (0 = not locked)      */
int EC;

EC = XMMHandleInfo( Handle, &blocksize, &freehnd, &lockmode);
```

Description:

The function returns in the parameter passed variables the information about the requested EMB handle, previously allocated by a call to XMMAlloc(). "blocksize" is the size of the EMB; "freehnd" is the number of memory blocks you can still allocate; "lockmode" is the lock count associated to the EMB: this should be 0 for each practical purpose.

Return value:

The function returns the error code stored in XMMError.

Example:

```
#include "agxmm.h"

int Handle;
unsigned long blksize;
int freehnd;
int lockmode;

  /*  Allocates 3Kb Extended Memory Block    */
  if( (Handle = XMMAlloc(1024*3)) == 0)
   {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
   }

  /*  Get information about allocated EMB    */
  if( XMMHandleInfo(Handle, &blksize, &freehnd, &lockmode) != NOERROR)
   {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
   }
```

```
printf("You can still allocate %d EMB\n",freehnd);

printf("Size of EMB %d is %ld bytes \n", Handle, blksize);
```

Related Functions:

XMMAlloc()

4.14 XMMInit_v_array

Name:

XMMInit_v_array - Creates a Virtual Array and makes it available for use with data contained in an existing Pseudo File

Declaration:

```
XMM_VACS *XMMInit_v_array(int PF_hnd, unsigned long elnum,
int elsize, unsigned long bsize, unsigned long segsize)
```

Synopsis:

```
#include "agxmm.h"

int PF_hnd;                    /* handle of  Pseudo File containing    */
                               /* data for the Virtual Array           */
unsigned long elnum;           /* number of elements in Virtual Array */
int elsize;                    /* size of elements in bytes            */
unsigned long bsize;           /* conv. mem. buffer size in bytes      */
                               /* takes default value if zero          */
unsigned long segsize;         /* number of elements per segment       */
                               /* takes default value if zero          */
XMM_VACS *vacs;                /* Virtual Array Control Structure      */

vacs = XMMInit_v_array(PF_hnd, elnum, elsize, bsize, segsize);
```

Description:

The function takes a previously opened Pseudo File handle to use it as a Virtual Array. It allocates the Virtual Array Control Structure and the buffers for data and their addresses. The data must have been initialized on the Pseudo File just as if they were array elements, i.e. consecutive elements must be in sequence in the Pseudo File with the expected size for array elements. The user must take care that the Pseudo File size is equal or greater than the memory needed to store the declared number of elements. The user can change the parameters that control the buffer and segment sizes, **bsize** and **segsize**, in order to improve Virtual Array access performance; setting to zero these variables will cause the system to take default values, **bsize** = 10240 b and **segsize** = 48 elements. The pointer to the Virtual Array Control Structure is returned after allocating space for it.

Return value:

The function returns NULL on error; the pointer to the Virtual Array Control Structure otherwise.

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include "agxmm.h"
```

```
/*********************************/
/*      Access Macros            */
/*********************************/

#define element(i)    (*(int *)XMMAccess_v_array(item_array, i))


main()
{
 XMM_VACS *item_array;
 int elem[1024];
 int PFile_Hand;

    ...... XMM and PFS initializations  ........


/* Opens a pseudofile */

if( (PFile_Hand = XMMOpen("PFname", 0, 2048)) == -1 )
 {
  printf( XMMErrorMsg( XMMError));
  exit(1);
 }

    ..... Pseudo File operations     ........

 /* take the Pseudo File and use it as a Virtual Array  */
 if( (item_array = XMMInit_v_array("Prova", arrsize, sizeof(int), 0, 0))
                                                              == NULL)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }


  ..... Virtual Array  operations such as  ...

  element(10) = elem[10];
```

Related Functions:

XMMCreate_v_array(), XMMLoad_v_array()

4.15 XMMInstalled

Name:

XMMInstalled - Checks if XMM is installed

Declaration:

```
int XMMInstalled( void );
```

Synopsis:

```
#include "agxmm.h"

int RetCode;

RetCode = XMMInstalled( );
```

Description:

The function tests the presence of an eXtended Memory Manager by generating the interrupt 0x2f with value 0x4300 in register AX. The interrupt handler returns the value 0x80 in AL register if a XMM is active. This function is called by each AGXMM function before any attempt to use eXtended Memory.

Return value:

The function returns 0 if no XMM is active, 1 otherwise.

Example:

```
void User_XMM_function()
{

 /* First tests if XMM is installed */
 if (!XMMInstalled() )
  {
   XMMError = NOTINSTALLED;
   return(0);
  }

  --------------------   User_XMM_function   body

}       /*  end User_XMM_function() */
```

Related Functions:

4.16 XMMLoad_v_array

Name:

XMMLoad_v_array - Loads a file from disk to eXtended Memory and makes it available as a Virtual Array

Declaration:

```
XMM_VACS *XMMLoad_v_array(char *fname, unsigned long elnum,
int elsize, unsigned long bsize, unsigned long segsize)
```

Synopsis:

```
#include "agxmm.h"

char *fname;                /* name of the disk file to be loaded  */
unsigned long elnum;        /* number of elements in Virtual Array */
int elsize;                 /* size of elements in bytes           */
unsigned long bsize;        /* conv. mem. buffer size in bytes     */
                            /* takes default value if zero         */
unsigned long segsize;      /* number of elements per segment      */
                            /* takes default value if zero         */
XMM_VACS *vacs;             /* Virtual Array Control Structure     */

vacs = XMMLoad_v_array(fname, elnum, elsize, bsize, segsize);
```

Description:

The function creates a Pseudo File loading data from a disk file, and then makes it available as a Virtual Array. It allocates the Virtual Array Control Structure and the buffers for data and their addresses. The data must have been stored on the disk file just as if they were array elements, i.e. consecutive elements must be in sequence with the expected size for array elements. The user must take care that the size of the Virtual Array is equal to that of the disk file. The user can change the parameters that control the buffer and segment sizes, **bsize** and **segsize**, in order to improve Virtual Array access performance; setting to zero these variables will cause the system to take default values, **bsize** = 10240 b and **segsize** = 48 elements. The pointer to the Virtual Array Control Structure is returned after allocating space for it.

Return value:

The function returns NULL on error; the pointer to the Virtual Array Control Structure otherwise.

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include "agxmm.h"

/*********************************/
/*      Access Macros            */
/*********************************/
```

```
#define element(i)    (*(int *)XMMAccess_v_array(item_array, i))


main()
{
 XMM_VACS *item_array;
 int elem[1024];
 unsigned long arrsize = 1024;

    ...... XMM and PFS initializations  ........


 /* take the disk file and load it as a Virtual Array  */
 if( (item_array = XMMLoad_v_array("fname", arrsize, sizeof(int), 0, 0))
                                                                   == NULL)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }


  ..... Virtual Array  operations such as  ...

  element(10) = elem[10];
```

Related Functions:

XMMCreate_v_array(), XMMInit_v_array()

4.17 MapError

Name:

MapError - Internally used, calculates values for XMMError variable

Declaration:

```
static int MapError( int XMMErr );
```

Synopsis:

```
#include "agxmm.h"

extern int XMMError;
int EC;

XMMError = MapError( EC );
```

Description:

The function is static into AGXMM.C module and is not accessible out of it. It is used by some functions to calculate the proper error code to be stored in XMMError global variable.

Return value:

The function returns the error code to be stored in XMMError.

Example:

Related Functions:

4.18 XMMmaxblock

Name:

XMMmaxblock - Calculates the size in bytes of the biggest free single EMB

Declaration:

```
unsigned long XMMmaxblock(void);
```

Synopsis:

```
#include "agxmm.h"

unsigned long size;           /* size of biggest single EMB   */

size = XMMmaxblock();
```

Description:

The function computes the size of the biggest single EMB which is still free at the moment of the call. The size is expressed in bytes.

Return value:

The function returns the size in bytes. A zero value indicates an error or the fact that there is no more available memory. The function sets XMMError.

Example:

```
#include "agxmm.h"

unsigned long size;

if( (size =XMMmaxblock()) == 0)
 {
  printf( "%s\n", XMMErrorMsg( XMMError));       /* Error management */
  exit(1);
 }
printf("Found  %lu  bytes for max. free EMB \n", size);
```

Related Functions:

XMMcoreleft()

4.19 XMMOpen

Name:

XMMOpen - Opens a Pseudo File, allocating an EMB of the desired size and setting properly its control structure XMMBLOCK

Declaration:

```
int XMMOpen( char *name, int flags, unsigned long size );
```

Synopsis:

```
#include "agxmm.h"

unsigned long size;          /* maximum allowed size in byte */
int PF_handle;               /* Pseudo File handle           */
int flag;                    /* mode flag                    */

PF_handle = XMMOpen( "Pseudo_name", flag, size );
```

Description:

This function opens a Pseudo File, once PFS has been initialized. The maximum allowed size must be passed as a parameter in order to allocate only once all the needed eXtended Memory and avoid possible troubles with XMMs that do not support XMMReAlloc() function. XMMOpen() checks if there are available handles in PFS; if yes, it allocates the requested amount of eXtended Memory and sets all the parameters in the proper XMMBLOCK control structure. XMMError is also set.

Return value:

The function returns -1 on error, or the handle identifier of the Pseudo File otherwise.

Example:

```
#include "agxmm.h"

int PFile_Hand = 0;
char *fname = "Prova";


 ----------    PSEUDO FILE SYSTEM INITIALIZATION    -----------


/* Opens a pseudofile */

if( (PFile_Hand = XMMOpen(fname, 0, 2048)) == -1 )
 {
  printf( XMMErrorMsg( XMMError));
  exit(1);
 }


 ----------    READ/WRITE OPERATIONS    -----------
```

```
/* Closes the pseudo file after use */

if( XMMClose(PFile_Hand) != 0 )
 {
  printf( XMMErrorMsg( XMMError));
  exit(1);
 }
```

Related Functions:

XMMClose()

4.20 XMMRead

Name:

XMMRead - Reads from a Pseudo File

Declaration:

```
int XMMRead( int PFile_Handle, char *buf, unsigned int len );
```

Synopsis:

```
#include "agxmm.h"

unsigned int len;                /* number of bytes to be read */
int PF_handle;                   /* Pseudp File handle         */
char buf[65530];       /* buffer in conv. memory     */
int reading;                     /* collect # of bytes read    */

reading = XMMRead( PF_handle, buf, len );
```

Description:

The function reads from a Pseudo File len bytes and puts them in a buffer allocated in conventional memory. It starts reading from the current position in the Pseudo File; such position can be changed using the function XMMSeek(). XMMRead() adjusts the parameter len if it exceeds the total real size of the Pseudo File, in such a way as to read from the current position up to the true end of it. The function updates the current position pointer in the Pseudo File and also sets XMMError. Pay attention to the fact that the maximum allowed value for the parameter len is 64Kb due to the well known DOS segmentation limit.

Return value:

The function returns -1 on error, 0 upon reaching the true end of the Pseudo File and the number of bytes read otherwise.

Example:

```
#include "agxmm.h"

int PFile_Hand = 0;
char buffer[65530];


 ----------    PSEUDO FILE SYSTEM INITIALIZATION    -----------

                -- Open a Pseudo File --

                --  Write operations  --

 if( XMMRead(PFile_Hand, buffer, 65530 ) == -1 )
  {
   printf( XMMErrorMsg( XMMError));
   exit(1);
  }
```

```
                    -- Close  Pseudo File --
```

Related Functions:

XMMWrite(), XMMTell(), XMMSeek()

4.21 XMMReAlloc

Name:

XMMReAlloc - Resizes an allocated EMB, rounded up to Kbyte

Declaration:

```
int XMMReAlloc( in Handle, unsigned long sizebytes);
```

Synopsis:

```
#include "agxmm.h"

unsigned long size;           /* new size of EMB     */
int handle;                   /* XMM handle number   */
int EC;

EC = XMMReAlloc(handle, size);
```

Description:

The function resizes a previously allocated Extended Memory Block. As in the case of XMMAlloc(), the new dimension is rounded up to Kb. If necessary, XMM varies the position of the EMB in order to resize it. This function is surely implemented by the device driver HIMEM.SYS, but other drivers may not provide it.

Return value:

The function returns the error code stored in XMMError.

Example:

```
#include "agxmm.h"

int Handle;
unsigned long blksize;
int freehnd;
int lockmode;

  /*  Allocates 3Kb Extended Memory Block   */
  if( (Handle = XMMAlloc(1024*3)) == 0)
   {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
   }

  /* resize previously allocated EMB  */
  if( XMMReAlloc(Handle,1024*5) != NOERROR)
   {
    /* Not all XMM support this function (e.g. QEMM 5.0 does not ) */
    if(XMMError == UNKFUNCTION)
     {
      printf("ATTENTION: This XMM does not support EMB reallocation!\n");
     }
    else
```

```
            {
             printf( "%s\n", XMMErrorMsg( XMMError));
             XMMFree(Handle);
             exit(1);
            }
         }
       else
         {
          if( XMMHandleInfo(Handle, &blksize, &freehnd, &lockmode) != NOERROR)
            {
             printf( "%s\n", XMMErrorMsg( XMMError));
             XMMFree(Handle);
             exit(1);
            }

          printf("Reallocated 2nd XMM block:  Handle = %d  \n", Handle);
          printf("Infos: size = %5luKb, free handles = %3d, lock mode = %3d\n",
                 blksize, freehnd, lockmode );
         }
```

Related Functions:

XMMAlloc()

4.22 XMMSeek

Name:

XMMSeek - Resets the current position pointer in a Pseudo File

Declaration:

```
long XMMSeek( int PFile_Handle, long offset, int fromwhere );
```

Synopsis:

```
#include <stdio.h>
#include "agxmm.h"

unsigned long pos;            /* stores current position  */
int PF_handle;                /* Pseudo File handle       */
long offset;                  /* offset to be used        */

pos = XMMSeek( PF_handle, offset, SEEK_SET);
```

Description:

The function changes the current position pointer in a Pseudo File. The starting position from which to compute the new offset can be specified by means of the following macros defined in < stdio.h > :

SEEK_SET	that corresponds to the beginning of the Pseudo File
SEEK_CUR	that means starting from current position
SEEK_END	that specifics the end of the Pseudo File

The function sets XMMError.

Return value:

The function returns -1 on error, the new current position otherwise.

Example:

```
#include <stdio.h>
#include "agxmm.h"

int PF_handle = 0;


 ----------    PSEUDO FILE SYSTEM INITIALIZATION    -----------

                    -- Open a Pseudo File --

                    --  Write operations  --

 /* Rewinds Pseudo file  */

 if( XMMSeek(PF_handle, 0, SEEK_SET) == -1 )
  {
    printf( XMMErrorMsg( XMMError));
    exit(1);
   }
                      --  Read operations  --

                      -- Close  Pseudo File --
```

Related Functions:

XMMWrite(), XMMTell(), XMMRead()

4.23 XMMTell

Name:

XMMTell - Returns the current position in a Pseudo File

Declaration:

```
long XMMTell( int PFile_handle );
```

Synopsis:

```
#include "agxmm.h"

unsigned long pos;            /* stores current position  */
int PF_handle;                /* Pseudo File Handle       */

pos = XMMTell( PF_handle );
```

Description:

The function gives the current position in the specified Pseudo File, computing it from its beginning.

Return value:

The function returns -1 on error, the current position otherwise.

Example:

```
#include "agxmm.h"

int PF_handle = 0;
int PF_pos;


 ----------     PSEUDO FILE SYSTEM INITIALIZATION    -----------

                   -- Open a Pseudo File --

                   --  Write operations  --

 /* Tells position in Pseudo File  */

 if( (PF_pos = XMMTell(PF_handle)) == -1 )
  {
   printf( XMMErrorMsg( XMMError));
   exit(1);
  }

  printf("Current Pseudo File position is %ld",PF_pos);

                   -- Rewind and read operations  --

                        -- Close  Pseudo File --
```

Related Functions:

4.24 XMMWrite

Name:

XMMWrite - Writes in a Pseudo File

Declaration:

```
int XMMWrite( int PFile_Handle, char *buf, unsigned int len );
```

Synopsis:

```
#include "agxmm.h"

unsigned int len;                /* number of bytes to be written */
int PF_handle;                   /* Pseudo File handle            */
char buf[65530];        /* buffer in conv. memory        */
int written;                     /* collect # of bytes written    */

written = XMMWrite( PF_handle, buf, len );
```

Description:

The function writes in a Pseudo File len bytes taking them from a buffer allocated in conventional memory. It starts writing from the current position in the Pseudo File; such a position can be changed using the function XMMSeek(). XMMWrite() checks if the parameter len exceeds the total true size of the Pseudo File, signaling an error if it is so. The function updates the current position pointer in the Pseudo File and also sets XMMError. Pay attention to the fact that the maximum allowed value for the parameter len is 64Kb due to the well known DOS segmentation limit.

Return value:

The function returns -1 on error, 0 at end of Pseudo File and the number of bytes written otherwise.

Example:

```
#include "agxmm.h"

int PFile_Hand = 0;
char buffer[] = "stringa";


 ----------    PSEUDO FILE SYSTEM INITIALIZATION    -----------

                    -- Open a Pseudo File --


 if( XMMWrite(PFile_Hand, buffer, strlen(buffer) ) == -1 )
  {
   printf( XMMErrorMsg( XMMError));
   exit(1);
  }

                    -- Close  Pseudo File --
```

Related Functions:

XMMRead(), XMMSeek(), XMMTell()

4.25 XMMVersion

Name:

XMMVersion - Tells which is the version of XMS, XMM and the status of HMA

Declaration:

```
int XMMVersion( int *XMS_ver, int *XMM_ver, int *HMA_flag);
```

Synopsis:

```
#include "agxmm.h"

int XMS_ver;                    /* XMS version number  */
int XMM_ver;                    /* XMM version number  */
int HMA_flag;                   /* HMA status flag     */
int EC;

EC = XMMVersion(int *XMS_ver, int *XMM_ver, int *HMA_flag);
```

Description:

The function works out the version number of XMS and XMM along with the status flag of HMA. The latter variable is 0 if HMA is not available, 1 otherwise. Version numbers are given in BCD format.

Return value:

The function returns the error code stored in XMMError.

Example:

```
#include "agxmm.h"

int XMS_ver;                    /* XMS version number  */
int XMM_ver;                    /* XMM version number  */
int HMA_flag;                   /* HMA status flag     */

  /*  query version numbers and HMA flag  */
  if( XMMVersion(&XMS_ver, &XMM_ver, &HMA_flag) != NOERROR)
   {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
   }

  printf("eXtended Memory Specification  version %x.%x installed\n"
         "      eXtended Memory Manager  version %x.%x installed\n"
         "                High Memory Area  status  %x \n\n",
        XMS_ver>>8,XMS_ver&0x00ff,XMM_ver>>8,XMM_ver&0x00ff,
        HMA_flag);
```

Related Functions:

Appendix A. Table of functions per library layer

Function	Description
BASIC XMM LAYER	
XMMAccess()	Initializes and accesses the eXtended Memory Manager
XMMAlloc()	Allocates XMM memory
XMMCopy()	Copies memory areas to/from eXtended memory
XMMcoreleft()	Computes remaining free eXtended memory
XMMErrorMsg()	Returns the pointer to a proper error message
XMMFree()	Releases previously allocated XMM memory
XMMHandleInfo()	Gets EMB handles information
XMMInstalled()	Checks if XMM is installed
MapError()	INTERNAL! Calculates values for XMMError variable
XMMmaxblock()	Calculates the size in bytes of the biggest free EMB
XMMReAlloc()	Resizes an allocated EMB
XMMVersion()	Returns XMs and XMM versions and HMA status
PSEUDO FILES	
XMMClose()	Closes a Pseudo File, releasing eXtended Memory
XMM_files_init()	Initializes the Pseudo Files System
XMMOpen()	Opens a Pseudo File of given size
XMMRead()	Reads data from a Pseudo File
XMMSeek()	Sets the current position pointer in a Pseudo File
XMMTell()	Returns the current position in a Pseudo File
XMMWrite()	Writes data in a Pseudo File
VIRTUAL ARRAYS	
XMMAccess_v_array()	Accesses a Virtual Array element
XMMClose_v_array()	Closes a Virtual Array
XMMCreate_v_array()	Creates a Virtual Array
XMMFast_Access_v_array()	Accesses a Virtual Array element (fast read)
XMMInit_v_array()	Creates a Virtual Array using data in an existing Pseudo File
XMMLoad_v_array()	Loads a file from disk and makes it available as a Virtual Array

Appendix B. Example Programs Source Files

B.1 AGXMMTST.C - general test program.

```
/*****************************************************************************
 * Module:   Sources AGLIB.LIB -> AGXMMTST.C
 *
 * Use:  XMM memory management test program.
 *
 * Alessandro Bondi - Gianluca Chiozzi
 *
 * Date: 20/07/90       Last Rev. :08/10/90
 *
 *****************************************************************************/



#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <string.h>

#include "agxmm.h"



/*****************/
/* Main function */
/*****************/

void main()
 {
  int  Handle1, Handle2, freehnd, lockmode;
  unsigned long blksize;
  int XMS_ver, XMM_ver, HMA_flag;
  unsigned long memsize;
  XMMCOPYBLOCK  test;
  char *string  = "XMM Allocation and Copy test string";
  char *string2 = "                                    ";

  printf("\n\n*******************************************\n");
  printf(    "*                                         *\n");
  printf(    "*        Test for XMM aglib functions     *\n");
  printf(    "*                                         *\n");
  printf(    "*                                         *\n");
  printf(    "*          A.Bondi - G. Chiozzi           *\n");
  printf(    "* Centro Ricerca Milano  - IBM Semea Srl  *\n");
  printf(    "*               25 Luglio 1990            *\n");
  printf(    "*******************************************\n\n");
  printf(    "                  -- IBM Internal Use Only --\n");

  /*************************************/
  /* Tests to see if XMM is installed */
  /*************************************/

  if((XMMAccess()) != NOERROR)
   {
    printf("Unable to Access XMS\n");
   exit(1);
  }


 /*************************************/
 /*        Gets XMM version           */
 /*************************************/

 if( XMMVersion(&XMS_ver, &XMM_ver, &HMA_flag) != NOERROR)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   exit(1);
  }

 printf("eXtended Memory Specification  version %x.%x installed\n"
        "      eXtended Memory Manager  version %x.%x installed\n"
        "              High Memory Area  status  %x \n\n",
       XMS_ver>>8,XMS_ver&0x00ff,XMM_ver>>8,XMM_ver&0x00ff,
       HMA_flag);

 /*************************************/
 /*       Gets XMM free mem           */
 /*************************************/

 if( (memsize =XMMcoreleft()) == 0)                    /* Total free mem */
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   exit(1);
  }

 printf("Found   %lu  XMM bytes unused\n", memsize);  /* Max free block */

 if( (memsize =XMMmaxblock()) == 0)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   exit(1);
  }

 printf("Found   %lu  bytes for max free block size\n", memsize);


 /**************************************************/
 /*       Allocates 2  XMM memory blocks           */
 /**************************************************/

 if( (Handle1 = XMMAlloc(1024*3)) == 0)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   exit(1);
  }
 if( XMMHandleInfo(Handle1, &blksize, &freehnd, &lockmode) != NOERROR)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   exit(1);
  }

 printf("Allocated 1st XMM block:  Handle = %d  \n", Handle1);
 printf("   Infos:  size = %5luKb, free handles = %3d, lock mode = %3d\n",
       blksize, freehnd, lockmode );

 /* Allocates Handle2 for a 2Kb block */
 if( (Handle2 = XMMAlloc(1024*2)) == 0)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   exit(1);
  }
 if( XMMHandleInfo(Handle2, &blksize, &freehnd, &lockmode) != NOERROR)
  {
   printf( "%s\n", XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   XMMFree(Handle2);
   exit(1);
  }

 printf("Allocated 2nd XMM block:  Handle = %d  \n", Handle2);
 printf("   Infos:  size = %5luKb, free handles = %3d, lock mode = %3d\n",
       blksize, freehnd, lockmode );


 /*************************************************/
 /* Reallocates 2nd block for a 3kb size from 2Kb */
 /*************************************************/

 if( XMMReAlloc(Handle2,1024*3) != NOERROR)
  {
   /* Not all XMM support this function (for example QEMM 5.0 does not ) */
   if(XMMError == UNKFUNCTION)
    {
     printf( "\n\nATTENTION: This XMM does not support EMB reallocation!\n\n");
    }
   else
    {
     printf( "%s\n", XMMErrorMsg( XMMError));
     XMMFree(Handle1);
     XMMFree(Handle2);
     exit(1);
    }
  }
 else
  {
   if( XMMHandleInfo(Handle2, &blksize, &freehnd, &lockmode) != NOERROR)
    {
     printf( "%s\n", XMMErrorMsg( XMMError));
     XMMFree(Handle1);
     XMMFree(Handle2);
     exit(1);
    }

   printf("Reallocated 2nd XMM block:  Handle = %d  \n", Handle2);
   printf("   Infos:  size = %5luKb, free handles = %3d, lock mode = %3d\n",
       blksize, freehnd, lockmode );
  }


 /**********************************************/
 /*       Test for XMM copy functions          */
```

```
/**********************************************/

printf("XMM copy test:\n");

/*** Copy from conventional memory to conventional memory ***/

test.bsize = strlen(string)+1;
test.src_Handle = 0;              /*  0 to use conventional memory       */
test.src_off.address = string;
test.dest_Handle = 0;
test.dest_off.address = string2;

if( XMMCopy(&test) != NOERROR )
 {
  printf( "%s\n", XMMErrorMsg( XMMError));
  XMMFree(Handle1);
  XMMFree(Handle2);
  exit(1);
 }
printf("  '%s' copied from conv mem to conv mem\n",string2);

/*** Copy from conventional memory to XMM ***/

test.bsize = strlen(string)+1;
test.src_Handle = 0;              /*  0 to use conventional memory       */
test.src_off.address = string;
test.dest_Handle = Handle1;
test.dest_off.offset = 0l;

if( XMMCopy(&test) != NOERROR )
 {
  printf( XMMErrorMsg( XMMError));
  XMMFree(Handle1);
  XMMFree(Handle2);
  exit(1);
 }
printf("  '%s' copied from conv mem to XMM\n",string);

/*** Copy from XMM to XMM ***/

test.bsize = strlen(string)+1;
test.src_Handle = Handle1;
test.src_off.address = 0l;
test.dest_Handle = Handle2;
test.dest_off.offset = 0l;

if( XMMCopy(&test) != NOERROR )
 {
  printf( XMMErrorMsg( XMMError));
  XMMFree(Handle1);
  XMMFree(Handle2);
  exit(1);
 }
printf("  '%s' copied from XMM to XMM\n",string);

/*** Copy from XMM memory to conventional memory ***/

test.bsize = strlen(string)+1;
test.src_Handle = Handle2;
test.src_off.offset = 0l;
test.dest_Handle = 0;
test.dest_off.address = string;

if( XMMCopy(&test) != NOERROR )
 {
  printf( XMMErrorMsg( XMMError));
  XMMFree(Handle1);
  XMMFree(Handle2);
  exit(1);
 }
printf("  '%s' copied from XMM to conv mem\n",string);


/*************************************************/
/*       Test for PseudoFile funcs (dusty)       */
/*************************************************/
{
 int PFile_Hand = 0;
 char *fname = "filename.PF";

 printf("\n\nXMM Pseudo Files Test\n");

 /* Init pseudofiles manager */

 if( XMM_files_init() != NOERROR )
  {
   printf( XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   XMMFree(Handle2);
   exit(1);
  }


 /* Opens a pseudofile */

 if( (PFile_Hand = XMMOpen(fname, 0, 2048)) == -1 )
  {
   printf( XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   XMMFree(Handle2);
   exit(1);
  }
 printf("Opened Pseudo File %s (PFile_Handle  %d)\n",fname,PFile_Hand);

 /* Writes string on Pseudo File */

 if( XMMWrite(PFile_Hand,string, strlen(string)+1) == -1 )
  {
   printf( XMMErrorMsg( XMMError));
   XMMFree(Handle1);
   XMMFree(Handle2);
   exit(1);
  }

 printf("Written string: current position is  %ld \n", XMMTell(PFile_Hand));

 /* Rewinds Pseudo file and reads string again */
```

```
   if( XMMSeek(PFile_Hand, 0, SEEK_SET) == -1 )
    {
     printf( XMMErrorMsg( XMMError));
     XMMFree(Handle1);
     XMMFree(Handle2);
     exit(1);
    }

   strcpy(string2,"teststr");
   if( XMMRead(PFile_Hand,string2, strlen(string)+1) == -1 )
    {
     printf( XMMErrorMsg( XMMError));
     XMMFree(Handle1);
     XMMFree(Handle2);
     exit(1);
    }

   printf("The read string is: %s \n", string2);

   /* Closes the pseudo file after use */

   if( XMMClose(PFile_Hand) != 0 )
    {
     printf( XMMErrorMsg( XMMError));
     XMMFree(Handle1);
     XMMFree(Handle2);
     exit(1);
    }
   printf("Closed Pseudo File\n");


  } /* end test for pseudo files */


  /************************************/
  /*       Gets XMM free mem          */
  /************************************/

  XMMFree(Handle1);
  XMMFree(Handle2);

  printf("Releasing XMM blocks\n");
  if( (memsize =XMMcoreleft()) == 0)      /* Release memory block */
   {
    printf( XMMErrorMsg( XMMError));
    exit(1);
   }

  printf("   now   %lu  XMM bytes unused\n", memsize);

  exit(0);

 } /* end main() */
```

B.2 AGXMFTST.C - Pseudo File use example program

```
/*****************************************************************************
 * Module:   Sources AGLIB.LIB -> AGXMFTST.C
 *
 * Use:  XMM Pseudo File management test program.
 *
 * Alessandro Bondi - Gianluca Chiozzi
 *
 * Date: 23/10/90       Last Rev. :23/10/90
 *
 *****************************************************************************/



#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <string.h>

#include "agxmm.h"



/*****************/
/* Main function */
/*****************/

void main()
 {
  int PFile_Hand = 0;
  char *string  = "XMM Pseudo Files test string";
  char *string2 = "                                   ";
  char *fname = "Filename.PF";

  printf("\n\n*******************************************\n");
  printf(    "*                                         *\n");
  printf(    "*   Test for XMM PseudoFiles functions    *\n");
  printf(    "*                                         *\n");
  printf(    "*                                         *\n");
  printf(    "*          A.Bondi - G. Chiozzi           *\n");
  printf(    "* Centro Ricerca Milano  - IBM Semea Srl  *\n");
  printf(    "*              25 Luglio 1990             *\n");
  printf(    "*******************************************\n\n");
  printf(    "                -- IBM Internal Use Only --\n");

  /*************************************/
  /* Tests to see if XMM is installed */
  /*************************************/

  if((XMMAccess()) != NOERROR)
   {
    printf("Unable to Access XMS\n");
    exit(1);
   }

  /********************************************/
  /*       Test for PseudoFile funcs         */
  /********************************************/
```

```
    /* Init pseudofiles manager */

    if( XMM_files_init() != NOERROR )
     {
      printf( XMMErrorMsg( XMMError));
      exit(1);
     }


    /* Opens a pseudofile */

    if( (PFile_Hand = XMMOpen(fname, 0, 2050)) == -1 )
     {
      printf( XMMErrorMsg( XMMError));
      exit(1);
     }
    printf("Opened Pseudo File %s (PFile_Handle  %d)\n",fname,PFile_Hand);

    /* Writes string on Pseudo File */

    printf("String to be written on file: %s\n",string);

    if( XMMWrite(PFile_Hand,string, strlen(string)+1) == -1 )
     {
      printf( XMMErrorMsg( XMMError));
      exit(1);
     }

    printf("Written string: current position is  %ld \n", XMMTell(PFile_Hand));

    /* Rewinds Pseudo file and reads string again */

    if( XMMSeek(PFile_Hand, 0, SEEK_SET) == -1 )
     {
      printf( XMMErrorMsg( XMMError));
      exit(1);
     }

    /* Reads string from file and puts it on a new buffer */

    if( XMMRead(PFile_Hand,string2, strlen(string)+1) == -1 )
     {
      printf( XMMErrorMsg( XMMError));
      exit(1);
     }

    printf("The read string is: %s \n", string2);

    /*******************************************************/
    /* Access file information on XMM_fchain data structure */
    /*******************************************************/

    /* The array element to be accessed is XMM_fchain(PFile_Hand) */

    printf("CURRENT PSEUDO_FILE STATUS:\n");
    printf("   PFile name        :    %s\n", XMM_fchainﬄPFile_Hand".name );
    printf("   EMB Handle        :    %d\n", XMM_fchainﬄPFile_Hand".Handle );
    printf("   Curr. Pos.        :    %ld\n", XMM_fchainﬄPFile_Hand".offset );
```

```
   printf("   max PFile size   :    %ld Kbytes\n",
                            XMM_fchain¶PFile_Hand".bsize );
   printf("   actual PFile size:    %ld\n",
                            XMM_fchain¶PFile_Hand".filesize );
   printf("   PFile mode       :    %d\n", XMM_fchain¶PFile_Hand".flags );

   /* Closes the pseudo file after use */

   if( XMMClose(PFile_Hand) != 0 )
    {
     printf( XMMErrorMsg( XMMError));
     exit(1);
    }
   printf("Closed Pseudo File\n");

  exit(0);

} /* end main() */
```

B.3 AGXMVTST.C - Virtual Arrays use example program

```
/*****************************************************************************
 * Module: AGLIB -> AGXMVTST.C
 *
 * Use: Test Program for XMM Virtual Arrays Functions
 *
 * Command Line : agxmvtst ﬂarray size ﬂprintout step ﬂbuf.size ﬂseg.size""""
 *                          # of els.                  byte    # of els.
 *
 * Author: Alessandro Bondi - Gianluca Chiozzi
 *
 * Date: 25/07/90       Last Rev. : 28/08/90
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <io.h>

#include "agxmm.h"

unsigned _stklen = 0xffff;


/*********************************/
/*      Access Macros           */
/*********************************/

#define XMM_VREC(i)     ((items *)XMMAccess_v_array(item_array, i))
#define item(i)         XMM_VREC(i)->v_item
#define qty(i)          XMM_VREC(i)->v_qty
#define desc(i)         XMM_VREC(i)->v_desc

/*****************************************/
/*  Array Elements Structure typedef    */
/*****************************************/

typedef struct
        {
        long v_item, v_qty;
        char v_descﬂ24";
        }
         items;

/*****************************************/
/*  Local function prototypes           */
/*****************************************/

void MyErrorFunc( int ErrCode);

#ifdef IBMC2
 void randomize(void);
#endif
```

```
/*****************************************/
/*          Start Main Program           */
/*****************************************/

main(int argc, char *argvffl")
{
 int step = 50;
 XMM_VACS *item_array;
 unsigned long i;
 int PFile_Handle;
 unsigned long arrsize, bsize, segsize;
 long temp;
 items fillchar;
 clock_t t1,t2, empty_t, full_t;              /*  timing variables  */
 FILE *flog;                                  /* measurements file  */
 ldiv_t ltemp;


 fillchar.v_item = -1;
 fillchar.v_qty  = -1;
 strcpy(fillchar.v_desc, "Null_array_item________");

 if( argc == 1)
   arrsize = 1000;
 else
   arrsize = atol(argvffl1");

 if( argc >= 3)
   step = atoi(argvffl2");

 if( argc >= 4)
   bsize   = atol(argvffl3");    /* Max Data Buffer Size in byte    */
 else
   bsize = 0;

 if( argc >= 5)
   segsize = atol(argvffl4");    /* # of elements per segment       */
 else
   segsize = 0;

 if ( access( "XMVARTST.TIM", 0 ) !=0 )
 {
   flog= fopen("XMVARTST.TIM","w");
   fprintf(flog,"\n\n*******************************************\n");
   fprintf(flog,    "*                                         *\n");
   fprintf(flog,    "*  Test for XMM virtual arrays functions  *\n");
   fprintf(flog,    "*                                         *\n");
   fprintf(flog,    "*          A.Bondi - G. Chiozzi           *\n");
   fprintf(flog,    "* Centro Ricerca Milano  - IBM Semea Srl  *\n");
   fprintf(flog,    "*              25 Luglio 1990             *\n");
   fprintf(flog,    "*******************************************\n\n");
   fprintf(flog,    "                -- IBM Internal Use Only --\n");

   fprintf(flog, "\n\n Total Size (Kb) | Buffer Size (Kb) | "
            "Segment Size(b)  |    Mean Access Rate(b/sec)\n\n\n");
 }
 else
   flog = fopen("XMVARTST.TIM","a");
 printf("\n\n*******************************************\n");
 printf(    "*                                         *\n");
 printf(    "*  Test for XMM virtual arrays functions  *\n");
 printf(    "*                                         *\n");
 printf(    "*          A.Bondi - G. Chiozzi           *\n");
 printf(    "* Centro Ricerca Milano  - IBM Semea Srl  *\n");
 printf(    "*              25 Luglio 1990             *\n");
 printf(    "*******************************************\n\n");
 printf(    "              -- IBM Internal Use Only --\n");

 printf("Testing for an array of %lu elements %d bytes long\n\n",
        arrsize, sizeof(items));

 printf(" - Command Line : \n"
        "    agxmvtst ﬄarray size ﬄprintout step ﬄbuf.size ﬄseg.size""""\n"
        "              # of els.                  byte    # of els.\n");

 XMM_Varr_error = MyErrorFunc;

if( XMMAccess() != 0)
{
 printf("Unable to access XMS\n");
 exit(1);
}

if(XMM_files_init() != NOERROR)
{
 printf("Unable to access XMS PseudoFiles\n");
 exit(1);
}
/* create a virtual array setting element size */
/* the size of item structure and setting the  */
/* initialization char to space char           */

if( (item_array = XMMCreate_v_array("Prova", arrsize, sizeof(items),
                      (char *)&fillchar, bsize, segsize)) == NULL)
 {
   printf( "%s\n", XMMErrorMsg( XMMError));
   exit(1);
 }


bsize   = item_array->bsize;
segsize = item_array->segsize;

/* fills in arrsize array items  and watch time */

for( i=0 ; i<arrsize ; i++)
{
 item(i) = i+1;
 qty(i)  = 0;
 sprintf(desc(i), "item # %ld", i+1);
}

/* prints content of filled items  */

for( i=0 ; i<arrsize ; i+= step)
{
 printf("Element # %ld   Item = %ld  Qty = %ld   Desc = %s   %d\n",
        i, item(i), qty(i), desc(i), (int)desc(i)ffl23");
 }

/*
                Sequential Access Test  Loop
*/

 printf("WAIT!  : Testing Sequential Access Performances\n");

 t1 = clock();                      /*  Empty Sequential Acces loop     */
 for( i=0 ; i<arrsize ; i++)
   temp = i;
 t2 = clock();
 empty_t = t2 - t1;

 t1 = clock();                      /*  True Sequential Acces loop     */
 for( i=0 ; i<arrsize ; i++)
   temp = item(i);
 t2 = clock();
 full_t = t2 - t1;


 printf("Sequential Access Rate:   %12.3f bytes/sec\n",
        (CLK_TCK*arrsize*sizeof(items))/(full_t-empty_t+0.00001));

 fprintf(flog,"      %4lu       |       %4lu       | "
           "      %4lu       | Seq   %#12.3f        \n\n",
           arrsize*sizeof(items)/1024,  bsize/1024,
           segsize*sizeof(items),
           (CLK_TCK*arrsize*sizeof(items))/(full_t-empty_t+0.00001));


/*
                Random Access Test  Loop
*/

 printf("WAIT!  : Testing Random Access Performances\n");

 randomize();                       /*  Initialize random # generator  */

 t1 = clock();                      /*  Empty reference loop           */
 for( i=0 ; i<arrsize ; i++)
  ltemp = ldiv( ((unsigned long)rand() * rand()) , arrsize);
 t2 = clock();
 empty_t = t2 - t1;

 t1 = clock();                      /*  True Random Access Loop        */
 for( i=0 ; i<arrsize ; i++)
 {
  ltemp = ldiv( ((unsigned long)rand() * rand()) , arrsize);
  temp = item(ltemp.rem);
 }
 t2 = clock();
 full_t = t2 - t1;

 printf("Random Access Rate:   %12.3f bytes/sec\n",
        (CLK_TCK*arrsize*sizeof(items))/(full_t-empty_t+0.00001));

 fprintf(flog,"                 |                  | "
            "                    | Ran    %#12.3f         \n\n",
            (CLK_TCK*arrsize*sizeof(items))/(full_t-empty_t+0.00001));


 fclose(flog);


 /*  closes virtual array  */

 if( (PFile_Handle = XMMClose_v_array(item_array)) == -1)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }

 if( XMMClose(PFile_Handle) != NOERROR)
  {
    printf( "%s\n", XMMErrorMsg( XMMError));
    exit(1);
  }

 return(0);

}         /* end main */


void MyErrorFunc( int ErrCode)
  {
    printf( "%s\n", XMMErrorMsg( ErrCode));
    exit(1);
  }



#ifdef IBMC2

/***************************************************************************
 * Function:  void randomize(void)
 *
 * Use: For IBMC2  (it is a standard TURBOC function): seeds the
 *      random number generator to a random value (uses time() so
 *      time.h must be included)
 *
 * Arguments:
 *
 * Returns:
 *
 * Date: 28/08/90    Last rev: _______
 *
 ***************************************************************************/

 void randomize(void)

 {
  time_t now;

  srand( (unsigned int)time(&now) );

 } /* end randomize */
```

```
#endif
```

Appendix C. Bibliography

[1] Ray Duncan - *MS-DOS Extensions*, 1989, Microsoft Press

[2] Mark Tichenor - *Virtual Arrays in C*, May 1988, Dr Dobb's Journal

IBM Semea
Milan Scientific Center
Circonvallazione Idroscalo
I-20090 Segrate (Milano, Italy)