CompuScope Drivers

for the

CompuScope 6012

CompuScope 1012

CompuScope 250

CompuScope 225

CompuScope 220

CompuScope LITE

DOS Driver and Windows DLL Documentation

for

Borland C 3.1 +
Microsoft C 5.1 +
Watcom C 9.0 +
Turbo Pascal for Windows 1.0 +
Visual Basic 1.0 +
Quick Basic 4.5
Protected Mode Pascal 7.0
LabWindows / CVI

Phone (514) 337-6893   Fax (514) 337-8411   BBS (514) 337-4317

Changes are periodically made to the information herein; these changes will be incorporated into new editions of the publication. Gage Applied Sciences may make improvements and/or changes in the products described in this publication at any time.

You can address your comments or queries to:

Product Manager - CompuScope
Gage Applied Sciences Inc.,
5465 Vanden Abeele
Montreal, Quebec
Canada H4S 1S1

Tel:	(514) 337-6893
Fax:	(514) 337-8411
BBS :	(514) 337-4317

CompuScope Driver Documentation

Table of Contents.

Version: 2.85

Dated: November 28, 1994.

Preface

This manual is designed to describe the routines included in the CompuScope 6012/1012/250/225/220/LITE Drivers for the Microsoft DOS or Microsoft Windows environment.

It is assumed that the programmer is familiar with the concept of DLLs, Windows 3.x programming (for DLL development) and the programming language in use. No description is included for these topics. If the programmer is not comfortable with any one of these topics, it is strongly recommended that a relevant reference manual be referred to before starting. Throughout this manual, the term CompuScope drivers will be used to refer to both the DOS drivers and the Windows DLL. Any differences between the two drivers will be discussed in the relevant sections

This manual describes the use of the driver routines for application program development. The routine descriptions, listed alphabetically, describe the C, Basic and Pascal syntax as well as provide an example call for each language (or reference to an example located elsewhere). Pascal for Windows and Protected Mode Pascal syntax for the driver routines are the generally the same. Any differences will be noted in the text. Similarly for Visual Basic and Quick Basic. Also in the descriptions are the set of named constants that can be used with each routine if/as appropriate. Every effort to use the constant's name and not the equivalent numeric value should be made as the numbers represented by the constants are subject to change but not the names of the constants.

The distribution disks contain all of the necessary files to use all types of CompuScope boards. The board specific drivers files differ slightly as appropriate to that particular CompuScope hardware, however, the basic layout and function names are consistent. The sample program source files will work with all of the board specific driver files required to control the CompuScope data acquisition card. As shipped, the drivers are configured to work with all CompuScope cards. If you wish to support just one type of card, or some other combination of cards, the drivers can easily be recompiled to do so . See Appendix B, Customizing the CompuScope Drivers, for more information.

The C drivers are coded to compile with either the Borland, Microsoft or Watcom family of C compilers using the ANSI C extensions. The drivers have been successfully compiled with Borland C versions 3.1 and 4.0, and with Microsoft C versions 5.1, 7.0 and 8.0, and Watcom C versions 9.0 and 10.0. The C sample programs on the distribution diskette are provided with project or makefiles for Borland, Micorsoft and Watcom C. Appendix C at the end of the manual describes most if not all of the sections that might need attention in order to convert any sample programs to use the Microsoft dialect of the C language. To use the older version of the Borland Turbo C compiler refer to Appendix D for coverage on converting the Borland C++ project file to the ASCII based Borland Turbo C v.2.0 format.

The description of the driver routines is broken down into four sections. The first section deals with the routines that are supported and accessible outside of the drivers, the second section shows the more low level routines, the third section deals with routines available only for DOS and the fourth section describes the board specific routines. Please note that there are a number of statically declared functions in each of the board specific driver files as well as the Gage driver supervisor file, GAGE_DRV.C. Most of these internal routines will continue to be supported, however, there is no guarantee, and there is no requirement that justifies the use of these internal routines. Some of the internal routines are subsets of the external commands and can be used intact by slightly modifying the source code. It must be stressed however, that the use of these unsupported routines may make your programs incompatible with future releases of the CompuScope drivers.

Every effort has been made to keep this manual as up-to-date and accurate as possible. However, if any discrepancies exist between the printed documentation and the examples or driver code provided on the distribution diskettes, you should consider the source code to be the most recent and accurate.

How to use the CompuScope Drivers

The CompuScope drivers are a set of files that can be used for program development with the Gage Applied Science's line of CompuScope high speed data acquisition cards. The drivers are currently useable under DOS for C, C++, Quick Basic and Borland Protected Mode Pascal development and as a dynamic link library (DLL) for any language that can support Windows DLLs. As shipped, the drivers will work with all CompuScope cards. They can be reconfigured to compile for only one card, or any combination of CompuScope cards. Please see Appendix B, Customizing the CompuScope Drivers, for more information on this. The drivers also come with sample programs written in C, Protected Mode Pascal, Quick Basic, Visual Basic and Turbo Pascal for Windows.

The Gage CompuScope drivers for both DOS and Windows support multiple boards and board types, however, from the programmers point of view only one board is accessible at any one given time. The initialization routine reads a specially formatted array to determine where the user has installed the CompuScope cards and then tries to initialize each board, determine that it is indeed present and then tests and sizes the memory on each found board. Another routine will read a binary disk file and initialize the special array, or the user can create the array with the format describe in the initialization routine, and pass it to the initialization routine. A routine is provided to select the desired active board and then all subsequent operations are applied to the active board, from data capture to configuration set up.

This section will describe the basic use of each of the drivers, what files come with it and any idiosyncrasies specific to program development using that language.

CompuScope C Drivers for DOS.

The CompuScope C driver are a set of object files that are linked in with your application program. The driver consists of the files:

GAGE_DRV.OBJ	(Driver Supervisor file)
CS112DRV.OBJ	(CS6012 and CS1012 support)
CS250DRV.OBJ	(CS250 and CS225 support)
CS220DRV.OBJ	(CS220 support)
CSLITDRV.OBJ	(CSLITE support)
TIMERS.OBJ	(high resolution timer)

The source files are included on your distribution diskette. To use the driver with your program, all that is needed is to include these object files in your project or make file. You will also have to include the header files, gage_drv.h , whichdrv.h, and optionally gage_low.h, at the top of your program. Note that whichdrv.h should be the first file included, as it contains several defines needed by the other header files.

The C drivers are coded to compile with either the Borland, Microsoft or Watcom C compilers using the ANSI C extensions. The drivers have been successfully compiled with Borland C versions 3.1 and 4.0, with Microsoft C versions 5.1, 7.0 and 8.0 and Watcom version 9.0 and 10. The C sample programs on the distribution diskette are provided with project or makefiles for the Borland, Microsoft and Watcom compilers. Appendix C at the end of the manual describes most if not all of the sections that might need attention in order to convert any sample programs to use the Microsoft dialect of the C language. To use the older version of the Borland Turbo C compiler refer to Appendix D for coverage on converting the Borland C++ project file to the ASCII based Borland Turbo C v.2.0 format.

The driver code has been compiled under the large model using the Pascal calling convention. In most cases, the application program should also be compiled in the large model. If you cannot use the large model, you should read Appendix B, Customizing the CompuScope Drivers, to find out how to recompile the drivers for a specific board. Also, the structures in the driver have been specified as byte-aligned. Application programs should also specify this in their compiler options.

If your program is making use of the **gage_read_config_file** function to read the configuration file GAGESCOP.INC, which is created with the utility GSINST.EXE that came with your CompuScope card, then the configuration file should be located in your current working directory. Alternatively, you can pass the full path name of the configuration file to the gage_read_config_file routine. A driver routine, **gage_get_config_filename**, is provided that will return the full path to GAGESCOP.INC. This routine expects to find the configuration file in your current working directory for the DOS drivers (including Pascal Protected Mode) or in your Windows directory for the DLL.

CompuScope Quick Basic Driver.

The Quick Basic driver consists of two libraries. The Quick Basic library, GAGE_DRV.QLB, is used to compile and run programs while in the Quick Basic environment. The file GAGE_DRV.LIB is used for command line compiling.

The driver library, GAGE_DRV.QLB, that is shipped with the drivers, will work for every CompuScope board. Similarly the sample program, SIMPLE.BAS, will work with all CompuScope boards. To use the driver library in the Quick Basic environment, you must start Quick Basic with the name of the library on the command line. To use the command line compiler, you must use the file GAGE_DRV.LIB. The batch files, QBS.BAT and MKSIMPLE.BAT on the distribution diskettes demonstrate both methods.

If your program is making use of the gagegetconfigfilename and gagereadconfigfile functions to read the configuration file GAGESCOP.INC created by the GSINST.EXE utility, the configuration file must be located in your current working directory. Alternatively, you can pass the full pathname of the file to the gagereadconfigfile routine.

Name Changes

Quick Basic does not allow the use of the underscore character in variable names, therefore, all the constants, variables and function and procedure names have these underscores removed, as listed in the GAGE_DRV.BAS source file. For example,

C / PASCAL language constant names	Basic language constant names
GAGE_NO_ERROR	GAGENOERROR
GAGE_DETECT_FAILED	GAGEDETECTFAILED
GAGE_ASSUME_CS250	GAGEASSUMECS250
GAGE_MEMORY_SIZE_TEST	GAGEMEMORYSIZETEST

C language function names	Basic language function names
gage_driver_initialize	gagedriverinitialize
gage_start_capture	gagestartcapture
gage_calculate_addresses	gagecalculateaddresses
gage_mem_read_dual	gagememreaddual

Since Basic does not support unsigned integers directly, all references to the type "word" in the C manual can be replaced by the type "integer". Basic also does not support the "byte" type for an 8 bit unsigned quantity, therefore, all references to byte are also changed to integer. Note that some function names may not be the same in Quick Basic as in other languages because of restrictions on the length of variable names. These distinctions will be pointed out in the description of these routines in the manual.

Driver Definitions and the Driver Library File

The driver library, GAGE_DRV.LIB (and the Quick Library GAGE_DRV.QLB), has a support file that defines all the required constants, "user types" and function and subroutine calls required to interface the Quick Basic environment to the CompuScope hardware. The GAGE_DRV.BAS file MUST be included at the top of any basic source file that will make use of the CompuScope driver. The following line is how this is accomplished.

```
REM $INCLUDE: 'GAGE_DRV.BAS'    '  Constants and routines for the CompuScope driver.
```

This file also contains the code that converts the C language function names in the library, to the format that is used by the Basic language. To explain how this works two examples are provided. Note that the second example is a special case because it deals with strings.

```
DECLARE FUNCTION gagedriverinitialize% ALIAS "GAGE_DRIVER_INITIALIZE" (SEG records AS
INTEGER, BYVAL memory AS INTEGER)
```

The preceding line declares an external function that will be called "gagedriverinitialize" from within the basic environment, the percent sign indicates that this function will return an integer. Basic will call the function with the PASCAL calling sequence. The "ALIAS" keyword is used to define the name of the function as it appears in the CompuScope driver library, in this case "GAGE_DRIVER_INITIALIZE". The parameters and their types are listed next. The "SEG" keyword is used to specify a far pointer to an integer (an array of integers in this case). The "BYVAL" keyword specifies that this parameter's value should be passed to the function (or subroutine) and not a reference to the value. This topic is described in more detail in the Quick Basic manual and on-line help.

An example call for this function can be found in the sample program and is repeated here.

```
DIM SHARED gageboardlocation(1 TO GAGEBLBUFFERSIZE) AS INTEGER

result% = gagedriverinitialize%(gageboardlocation(1), GAGEMEMORYSIZETEST)   ' Ignore this result.
result% = gagedriverinitialize%(gageboardlocation(1), GAGEMEMORYSIZETEST)   ' Use this result.
```

The second example is very similar to the first example, but with an important exception. It is this exception that will be discussed.

```
DECLARE FUNCTION gagereadconfigfile% ALIAS "GAGE_READ_CONFIG_FILE" (BYVAL
segfilename AS INTEGER, BYVAL offfilename AS INTEGER, SEG records AS INTEGER)
```

The actual C language function has two parameters, a far pointer to a NULL terminated string and a far pointer to an array of integers. However, due to the differences between Basic and C's string handling a tricky fix was required. Basic strings are part of the language and are stored as a count of how many characters are in the string and where in the string data space it resides. Therefore, simply referring to the string would yield an error since the string does not exist at its named location as it would in the C language. The solution is to append a zero (an ASCII zero, not the character zero '0') to the string and pass the string offset and the segment address of the string space. Together these two values make a far pointer to the NULL terminated string which is exactly what the CompuScope driver expects. Therefore, it is extremely important to maintain this calling procedure when using gagereadconfigfile. The following code segment illustrates the above description.

```
boardlocations$ = "GAGESCOP.INC" + CHR$(0)
result% = gagereadconfigfile%(VARSEG(boardlocations$), SADD(boardlocations$), SEG
                                                        gageboardlocation(1))
```

Similarly, for the function gagegetconfigfilename:

```
configname$ = "This is an eighty character string used as a placeholder for the C library calls" +
                                                              CHR$(0)
result% = gagegetconfigfilename%(VARSEG(configname$), SADD(configname$))
match = INSTR(configname$, CHR$(0))
boardlocations$ = LEFT$(configname$, match)
boardlocations$ = boardlocations$ + CHR$(0)
```

It is very important to use the CALL keyword when making use of subroutines in the CompuScope driver library, for example,

```
CALL gagestartcapture(0)
```

Quick Basic and the CompuScope Libraries

In order to write basic code in the Quick Basic environment the GAGE_DRV.QLB Quick Basic Library must be specified on the command line when calling the QB.EXE program. The batch file on the distribution disk QBS.BAT will load the SIMPLE.BAS program and the Quick Library GAGE_DRV.QLB with Quick Basic. The source of this file is also included here.

```
QB /H simple.bas /L gage_drv.qlb
```

The /H parameter loads Quick Basic in the maximum available display resolution.

The /L parameter instructs Quick Basic to load the Quick Library GAGE_DRV.QLB.

Due to the size and complexity of the Quick Library, Quick Basic is unable to initialize the CompuScope driver from within the Quick Basic environment unless the SETMEM function is called to reduce the memory pool that Quick Basic uses for its own purposes. This allows the CompuScope library to dynamically allocate memory for the driver. The two lines that follow illustrate the call.

```
temp = SETMEM(-20480)  '  Minimum per 12 Bit board (CS6012 or CS1012).
or
temp = SETMEM(-10240)  '  Minimum per 8 Bit board (CS250, CS225, CS220 or CSLITE).
```

This call must be included before calling gagedriverinitialize. Note that the numbers specified are negative (they remove memory from the Quick Basic memory pool). If more that one board is to be initialized then the values used must be multiplied by the expected number of boards. The 12 bit CompuScope 1012 requires twice the memory in the driver than the 8 bit boards. If the SETMEM function is not called then the basic program must be compiled and run from the command line.

Quick Basic and Command Line Compiling

A batch file is included on the distribution disk to compile the sample program and link it with the CompuScope library. The batch file MKSIMPLE.BAT is repeated here.

```
BC SIMPLE.BAS /O /T /C:512;
LINK @MKSIMPLE.LNK
```

The linker response file MKSIMPLE.LNK is also included on the distribution disk and is listed below.

```
SIMPLE.OBJ,
SIMPLE.EXE,
NUL.MAP /NOE /NOD:BRUN45.LIB,
C:\QB\LIB\BCOM45.LIB+
GAGE_DRV.LIB;
```

The path to the Quick Basic library BCOM45.LIB may need to be modified for an individual installation of Quick Basic.

CompuScope Protected Mode Pascal Driver.

The Protected Mode Pascal driver is implemented as a dynamic link library (DLL) that can be used with Borland Pascal 7.0 in protected mode. The unit GAGE_DRV.TPP, provided on the distribution diskette, contains all the declarations needed to enable your program to communicate with the DLL and must be included in the Uses clause of the calling program. Most of the comments and descriptions listed in the section of this manual titled DLL Basics: Turbo Pascal for Windows also apply to Protected Mode Pascal with the exceptions noted below.

Protected Mode Pascal allows the use of the memory, module and resource management functions that are available under the Windows API (Application Programming Interface). This is done by using the WinAPI unit, which allows access to the Borland run-time manager, in your protected mode programs. This unit also allows the loading of dynamic link libraries and low-level access to selectors.

Because the WinAPI unit is a subset of the Windows API, any dynamic link library that doesn't use Windows specific calls can be used in protected-mode. The CompuScope DLL, GAGE_DRV.DLL, provided on your diskette is a slightly modified version of the DLL that is provided for Windows development. The modifications were done to the run time library to remove references to the WIN87EM.DLL if the program is not being compiled under Windows.

The GAGE_DRV.PAS file, which contain the functions, procedures and constants needed to call the GAGE_DRV.DLL has also been slightly modified from the Windows version. The WinAPI unit is used, and some types, such as plongint are defined in the GAGE_DRV.PAS file. The driver routine **gage_get_config_filename** looks for the configuration file, GAGESCOP.INC, created by the GSINST.EXE utility program in your current working directory rather then the Windows directory. Note that the routine uses a null-terminated string rather than the standard Pascal string. All other functions and procedures of the DLL work the same as they do when using Turbo Pascal for Windows. The distribution diskette comes with two versions of the DLL, one for Windows and one for protected mode Pascal. Currently, the version of the DLL for protected mode Pascal will work under Windows, but not vice versa. Therefore, if you are developing both Windows and protected-mode applications, it is recommended that you use the version of GAGE_DRV.DLL for protected mode Pascal.

Note that you should call a floating point routine early in your program so the Pascal floating point library, which is needed by the GAGE_DRV.DLL, is loaded. The sample program provided on your diskette, GAGETSTP.EXE, uses the line x := 1.0 as the first line in the program.

To run an application in protected mode, the two files that make up the Borland protected-mode extensions, DPMI16BI.OVL and RTM.EXE, must be on your path.

For more information specific to developing protected-mode applications, consult your Borland Pascal with Objects Language Guide.

CompuScope DLL Basics: C.

This basic introduction to the CompuScope DLL shows how to include the DLL in your code and then how to make use of these routines.

The simplest method of using the CompuScope DLL routines is to add the following DLL import code to your module definition file. These commands can also be found as part of the file GSDLLDEM.DEF for the included sample program.

The file, GAGE_DRV.DLL, should be put in your Windows directory, usually C:\WINDOWS. If your program makes use of the configuration file GAGESCOP.INC , which can be created by using either the GSWINST.EXE or GSINST.EXE utility, this must also be located in your Windows directory. GSWINST will automatically put the configuration file in your Windows directory. If you use the DOS based GSINST.EXE, you must either run it from the Windows directory, copy the resulting GAGESCOP.INC file to your Windows directory or run GSINST.EXE with the -f command line switch, ie. GSINST -fc:\windows\gagescop.inc.

```
IMPORTS     GAGE_DRV.gage_32k_to_buffer
            GAGE_DRV.gage_abort
            GAGE_DRV.gage_abort_capture
            GAGE_DRV.gage_busy
            GAGE_DRV.gage_calculate_addresses
            GAGE_DRV.gage_calculate_mr_addresses
            GAGE_DRV.gage_capture_mode
            GAGE_DRV.gage_detect_multiple_record
            GAGE_DRV.gage_driver_initialize
            GAGE_DRV.gage_driver_remove
            GAGE_DRV.gage_fast_set_block_number
            GAGE_DRV.gage_forced_trigger_capture
            GAGE_DRV.gage_get_boards_found
            GAGE_DRV.gage_get_config_filename
            GAGE_DRV.gage_get_current_drv_structure
            GAGE_DRV.gage_get_data
            GAGE_DRV.gage_get_data_high
            GAGE_DRV.gage_get_data_low
            GAGE_DRV.gage_get_driver_info
            GAGE_DRV.gage_get_driver_info_structure
            GAGE_DRV.gage_get_error_code
            GAGE_DRV.gage_get_interpolate_trigger
            GAGE_DRV.gage_get_records
            GAGE_DRV.gage_get_trigger_view_offset
            GAGE_DRV.gage_init_clock
            GAGE_DRV.gage_initialize_start_capture
            GAGE_DRV.gage_input_control
            GAGE_DRV.gage_make_error_code
            GAGE_DRV.gage_mem_read_chan_a
            GAGE_DRV.gage_mem_read_chan_b
            GAGE_DRV.gage_mem_read_dual
            GAGE_DRV.gage_mem_read_single
            GAGE_DRV.gage_multiple_record
            GAGE_DRV.gage_need_ram
```

GAGE_DRV.gage_normalize_address
GAGE_DRV.gage_ram_full
GAGE_DRV.gage_read_config_file
GAGE_DRV.gage_read_master_status
GAGE_DRV.gage_reset_interpolate_trigger
GAGE_DRV.gage_select_board
GAGE_DRV.gage_select_current_board
GAGE_DRV.gage_set_block_number
GAGE_DRV.gage_set_ext_clock_variables
GAGE_DRV.gage_set_records
GAGE_DRV.gage_set_trigger_view_offset
GAGE_DRV.gage_software_clock
GAGE_DRV.gage_software_trigger
GAGE_DRV.gage_start_capture
GAGE_DRV.gage_trigger_address
GAGE_DRV.gage_trigger_control
GAGE_DRV.gage_trigger_view_transfer
GAGE_DRV.gage_triggered
GAGE_DRV.gage_triggered_aux
GAGE_DRV.gage_update_driver_info
GAGE_DRV.cs1012_enable_test_memory
GAGE_DRV.cs1012_offset_adjust
GAGE_DRV.cs1012_test_memory_chan
GAGE_DRV.cs1012_trigger_control_2
GAGE_DRV.cs250_enable_ets
GAGE_DRV.cs250_ets_average_capture
GAGE_DRV.cs250_ets_average_capture_2
GAGE_DRV.cs250_ets_capture
GAGE_DRV.cs250_ets_capture_2
GAGE_DRV.cs250_ets_detect
GAGE_DRV.cs250_mem_read_ets_data
GAGE_DRV.cs250_set_ets_delay
GAGE_DRV.cs250_set_ets_rate

When a CompuScope DLL routine is required then the name of the routine is used as listed after the "GAGE_DRV." prefix, as it is in the remainder of the manual. The CompuScope DLL can also be linked dynamically, however since most programs make regular use of the CompuScope DLL, dynamic linking does not save much time or space since the DLL would have to be loaded and unloaded too often to have any great time or memory space savings. The DLL can also be used by including the GAGE_DRV.LIB file, which comes on the distribution disks, in your project or make file. In this case, you can supply your own .DEF file or use the default one that your compiler provides.

CompuScope DLL Basics: Turbo Pascal for Windows.

This basic introduction to the CompuScope DLL shows how to include the DLL in your code and then how to make use of these routines.

The simplest method of using the CompuScope DLL routines is to add the following DLL import code to a separate unit in your program. The full set of commands can be found as part of the file GAGE_DRV.PAS for the included sample program, which must be used in your own application programs by including it in the Uses statement in the main program. Other units included on the diskette and used by the GSDLLDEM.PAS sample program are START.PAS, which defines and initializes several structures used by the sample program and INTERPOL.PAS, which provides some support routines for interpolated trigger.

The DLL file, GAGE_DRV.DLL, should be put in your Windows directory, usually C:\WINDOWS. If your program makes use of the configuration file GAGESCOP.INC , which can be created by using either the GSWINST.EXE or GSINST.EXE utility, this must also be located in your Windows directory. GSWINST will automatically put the configuration file in your Windows directory. If you use the DOS based GSINST.EXE, you must either run it from the Windows directory, copy the resulting GAGESCOP.INC file to your Windows directory or run GSINST.EXE with the -f command line switch, ie. GSINST -fc:\windows\gagescop.inc.

```
INTERFACE

        procedure gage_32k_to_buffer (var buffer : byte; high_half, start_block : integer;  var
                blocks_transfered : integer);
        procedure gage_abort;
        procedure gage_abort_capture (reset_trigger_source : integer);
        function gage_busy : integer;
        procedure gage_calculate_addresses (chan,  op_mode : integer; tbs : longint; trig,  start,
                endaddr :  plongint);

                                etc...


IMPLEMENTATION

        procedure gage_32k_to_buffer;                   external  'GAGE_DRV';
        procedure gage_abort;                           external  'GAGE_DRV';
        procedure gage_abort_capture;                   external  'GAGE_DRV';
        function    gage_busy;                          external  'GAGE_DRV';
        procedure gage_calculate_addresses;             external  'GAGE_DRV';

                                etc...
```

When a CompuScope DLL routine is required, the name of the routine is used just as in a regular Pascal subroutine call. The CompuScope DLL can also be linked dynamically, however since most programs make regular use of the CompuScope DLL, dynamic linking does not save much time or space since the DLL would have to be loaded and unloaded too often to have any great time or memory space savings.

CompuScope DLL Basics: Visual Basic.

This basic introduction to the CompuScope DLL shows how to include the DLL in your code and then how to make use of these routines.

The simplest method of using the CompuScope DLL routines is to add the following DLL import code to a separate global module in your program. The full set of commands can be found as part of the file GAGE_DRV.BAS in the included sample program. This file should be included in any Visual Basic application that you write. Please note that, although they appear here over multiple lines for clarity, the declarations for each DLL routine must be on one line in the Visual Basic module. Also, the file "GAGE_DRV.DLL" should be located in your windows directory (usually C:\WINDOWS) and the runtime Visual Basic DLL, "VBRUNxxx.DLL", should be in your windows system directory (usually C:\WINDOWS\SYSTEM).

If your program makes use of the configuration file GAGESCOP.INC , which can be created by using either the GSWINST.EXE or GSINST.EXE utility, this must also be located in your Windows directory. GSWINST will automatically put the configuration file in your Windows directory. If you use the DOS based GSINST.EXE, you must either run it from the Windows directory, copy the resulting file, GAGESCOPE.INC, to your Windows directory or run GSINST.EXE with the -f command line switch, ie. GSINST -fc:\windows\gagescop.inc.

Other files included on the diskette and used by the GSDLLDEM.BAS sample program are START.BAS, which defines and initializes several structures used by the sample program and INTERPOL.BAS, which provides some support routines for interpolated trigger.

```
Declare Sub gage_32k_to_buffer Lib "GAGE_DRV.DLL" (buffer As Integer,
        ByVal high_half As Integer, ByVal start_block As Integer,  blocks_transfered As
        Integer)
Declare Sub gage_abort Lib "GAGE_DRV.DLL" ()
Declare Sub gage_abort_capture Lib "GAGE_DRV.DLL" (ByVal reset_trigger_source As Integer)
Declare Function gage_busy Lib "GAGE_DRV.DLL" () As Integer
Declare Sub gage_calculate_addresses Lib "GAGE_DRV.DLL" (ByVal chan As Integer, ByVal
        op_mode,  As Integer, ByVal tbs As Long, trig As Long, start As Long, ending As Long)

                        etc...
```

When a CompuScope DLL routine is required, the name of the routine is used just as in a regular Visual Basic subroutine call. The CompuScope DLL can also be linked dynamically, however since most programs make regular use of the CompuScope DLL, dynamic linking does not save much time or space since the DLL would have to be loaded and unloaded too often to have any great time or memory space savings.

CompuScope Driver: types, structures and definitions.

Most definitions, structures and names are described in this section using C syntax. The equivalent Pascal or Basic syntax is usually quite similar. The exact syntax can be obtained by looking in the files containing the definitions (**GAGE_DRV.PAS** or **GAGE_DRV.BAS**). Note that because Quick Basic variables and function names cannot have underscores, the names are defined without them. Also, because neither Quick Basic or Visual Basic have word or byte types, all variables defined as such are declared as integers in Basic.

The **GAGE_DRV.H** (for C), **GAGE_DRV.PAS** (for Pascal) or **GAGE_DRV.BAS** (for Basic) files should be included with any source file that makes use of the driver functions or the variables defined in the driver. The following lines are the normal method of including the driver header file and it is typically loaded from the current directory. Note that whichdrv.h should be included before anything else.

```
#include “whichdrv.h”
#include "gage_drv.h"       (for C)

REM $INCLUDE: 'GAGE_DRV.BAS'        (for Quick Basic)

Uses GAGE_DRV;           (for Pascal)
```

Under Visual Basic, the module GAGE_DRV.BAS would be included in the makefile.

Name changes

The following driver routines and constants have been changed from previous versions of the drivers. The old names are still defined and can be used, but it is recommended the newer names be used for future compatibility.

These driver names have been changed to reflect limits on name length for different languages.

gage_calculate_multiple_record_addresses	to	gage_calculate_mr_addresses
gage_get_current_driver_structure	to	gage_get_current_drv_structure
gage_get_interpolated_trigger	to	gage_get_interpolate_trigger
gage_reset_interpolated_trigger	to	gage_reset_interpolate_trigger
gage_set_external_clock_variables	to	gage_set_ext_clock_variables

These input range constants have been changed to more accurately describe their use.

GAGE_DIVIDE_10	to	GAGE_PM_10_V
GAGE_DIVIDE_5	to	GAGE_PM_5_V
GAGE_DIVIDE_2	to	GAGE_PM_2_V
GAGE_TIMES_1	to	GAGE_PM_1_V
GAGE_TIMES_2	to	GAGE_PM_500_MV
GAGE_TIMES_5	to	GAGE_PM_200_MV
GAGE_TIMES_10	to	GAGE_PM_100_MV

CompuScope memory types.

The following type definitions are used in the drivers to allow the same set of driver source files to be compiled under both DOS and Windows, and to facilitate the use of either 16 bit or 32 bit code. They are defined in GAGE_DRV.H (and in WHICHDRV.H) for C and GAGE_DRV.PAS for Pascal. If your version of these files has these types defined, they should be used in all calls to the drivers or DLL. Basic programmers should continue to use the standard integer and long types. See the example programs on the distribution diskette for more information on how to use these types.

```
typedef  unsigned char     uInt8;
typedef  unsigned short    uInt16;
typedef  unsigned long     uInt32;
typedef  signed char       int8;
typedef  signed short      int16;
typedef  signed long       int32;
#ifndef  WINDOWS_CODE
#ifdef   __WATCOMC__
typedef  char *            LPSTR;
#else
typedef  char far *        LPSTR;
#endif
#endif
```

A new data type has been defined in GAGE_DRV.H (and WHICHDRV.H) as follows:

```
/*       Calling convention.       */

#ifdef   __WATCOMC__
#define  GAGEAPI pascal
#ifdef   far
#undef   far
#define  far
#endif
#else
#ifndef GAGEAPI
#ifdef   WINDOWS_CODE
#define GAGEAPI           FAR PASCAL
#else
#define GAGEAPI           far pascal
#endif
#endif
#endif
```

This allows the same set of driver source files to be used under both DOS and Windows. The driver source files are now built with the PASCAL calling convention under both DOS and Windows. GAGEAPI is defined as far pascal under DOS and FAR PASCAL under Windows.

The drivers are now also compatible with the Watcom 32 bit C compiler. If the Watcom compiler is being used, GAGEAPI is defined as pascal and the far keyword is not defined.

CompuScope error definitions.

If a routine fails, a global error variable is available that describes the error that occurred and the board that caused the error. This variable, called **gage_error_code**, is an encoded integer with the high byte containing the board in error and the low byte is equal to one of the following defined error constants.

GAGE_NO_ERROR
GAGE_NO_SUCH_BOARD
GAGE_NO_SUCH_MODE
GAGE_NO_SUCH_INPUT
GAGE_INVALID_SAMPLE_RATE
GAGE_NO_SUCH_COUPLING
GAGE_NO_SUCH_CHANNEL
GAGE_NO_SUCH_GAIN
GAGE_NO_SUCH_TRIG_DEPTH
GAGE_NO_SUCH_TRIG_POINT
GAGE_NO_SUCH_TRIG_SLOPE
GAGE_NO_SUCH_TRIG_SOURCE
GAGE_MISC_ERROR

A C language program that needs access to the **gage_error_code** variable will require the **GAGE_DRV.H** header file to be included in the source file that will make use of this variable. The header file automatically makes the external definition when the header file is not included in the **GAGE_DRV.C** driver source file. Alternatively, the driver routine **gage_get_error_code** can be used.

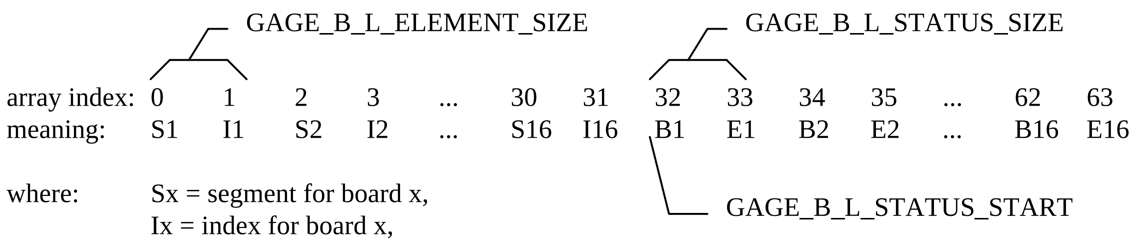
Driver boolean values.

Since the C language does not have true boolean types and constants, in the Pascal sense of the term, two constants have been provided for this purpose to synthesize the boolean meanings. Substituting the equivalent numeric values will be guaranteed to work, since the routines that use the true and false values only want a non-zero or zero value respectfully.

```
#define  TRUE   1        /*  A true value.  */
#define  FALSE  0        /*  A false value.  */
```

Locating the CompuScope boards in the PC.

The **gage_driver_initialize** routine uses an array called **gage_board_location** to pass the desired locations for the CompuScope hardware to the driver code. The following section illustrates the relationship between the **gage_board_location** array and the **GAGE_B_L_xxxxxxx** constants which layout the pseudo structure.
CompuScope **gage_board_location** array "pseudo structure".



array index:	0	1	2	3	...	30	31	32	33	34	35	...	62	63
meaning:	S1	I1	S2	I2	...	S16	I16	B1	E1	B2	E2	...	B16	E16

where:
Sx = segment for board x,
Ix = index for board x,
Bx = returned board type for board x,
Ex = returned status error for board x.

/* CompuScope definitions for gage_board_location sizing. */

```
#define  GAGE_B_L_MAX_CARDS               16
#define  GAGE_B_L_SIZEOF_ELEMENT          2
#define  GAGE_B_L_ELEMENT_SIZE            2
#define  GAGE_B_L_STATUS_SIZE             2
#define  GAGE_B_L_STATUS_START            (GAGE_B_L_MAX_CARDS *
                                          GAGE_B_L_ELEMENT_SIZE)
#define  GAGE_B_L_BUFFER_SIZE             (GAGE_B_L_MAX_CARDS *
                                          (GAGE_B_L_ELEMENT_SIZE +
                                          GAGE_B_L_STATUS_SIZE))
```

The easiest method of setting up this array is to call the **gage_read_config_file** routine that uses the same **GAGESCOP.INC** file as the **GAGESCOP.EXE** program uses for locating the CompuScope hardware. The **gage_board_location** array can also be initialized by calling the **gage_set_records** routine, see the description for this routine for an example explaining the initialization of the **gage_board_location** array. This array is declared as:

```
uInt16   gage_board_location[GAGE_B_L_BUFFER_SIZE];
```

the first segment and I/O index locations are:

```
gage_board_location[0] = segment;     /*  "segment" is a word that is set to equal the desired
                                      segment for board 1.  */
gage_board_location[1] = index;       /*  "index" is a word that is set to equal the desired I/O index
                                      for board 1.  */
```

When the **gage_driver_initialize** routine finds a CompuScope board at the specified locations the board type is returned via the **gage_board_location** array. The board type for the first board is located in the **gage_board_location** array using the driver constants available for accessing this array.

```
board_type = gage_board_location[GAGE_B_L_STATUS_START];   /*  "board_type" is a word that will
                                      equal one of the board type constants listed below.  */
```

If the driver detects a board during initialization then one of the following "ORable" constants is set in the **gage_board_location** status position. The board type definitions are:

```
        GAGE_ASSUME_NOTHING
        GAGE_ASSUME_CSLITE
        GAGE_ASSUME_CS220
        GAGE_ASSUME_CS225
        GAGE_ASSUME_CS250
        GAGE_ASSUME_CSLITE15        /*  CSLITE version 1.5 and newer.  */
        GAGE_ASSUME_CS1012
        GAGE_ASSUME_CS6012
        GAGE_ASSUME_CS2125
        GAGE_ASSUME_RESERVED
        GAGE_ASSUME_ALL_BOARDS
```

The **GAGE_ASSUME_ALL_BOARDS** constant is included to provide a quick means to determine if any CompuScope hardware was encountered during initialization.

If the **gage_driver_initialize** routine encounters an error during initialization, it returns the error code(s) detected also via the **gage_board_location** array. The error status for the first board is:

status = gage_board_location[GAGE_B_L_STATUS_START + 1]; /* "status" is a **word** that will have zero or more of the error constant bits set. */

The value returned can be one or more of the following constants. Note that these constants are individual bits and can be "ORed" together to form a mask. The CompuScope definitions for **gage_board_location** status errors are:

GAGE_BAD_LSB_SEGMENT
GAGE_BAD_MSB_SEGMENT
GAGE_BAD_LSB_INDEX
GAGE_BAD_MSB_INDEX
GAGE_DETECT_FAILED
GAGE_MEMORY_FAILED
GAGE_BAD_MEMORY_SIZE
GAGE_ALL_GAGE_STATUS_ERRORS

The **GAGE_ALL_GAGE_STATUS_ERRORS** constant is included to provide a quick means to determine if any errors were encountered during initialization.

The above method can be used to set and retrieve the **gage_board_location** array settings, but an easier solution is to use the **gage_set_records** and **gage_get_records** routines.

Verifying the CompuScope memory size.

The driver is normally instructed to check the encountered CompuScope hardware for the size of memory installed by using the **GAGE_MEMORY_SIZE_TEST** predefined constant. This method of operation can be overridden for special purposes by using the following constants for overriding the built in memory detection code:

GAGE_MEMORY_SIZE_TEST
GAGE_MEMORY_SIZE_016K
GAGE_MEMORY_SIZE_032K
GAGE_MEMORY_SIZE_064K
GAGE_MEMORY_SIZE_128K
GAGE_MEMORY_SIZE_256K
GAGE_MEMORY_SIZE_512K
GAGE_MEMORY_SIZE_001M
GAGE_MEMORY_SIZE_002M
GAGE_MEMORY_SIZE_004M
GAGE_MEMORY_SIZE_008M

Driver structure for board description.

An important structure, defined in GAGE_DRV.H, GAGE_DRV.PAS and GAGE_DRV.BAS is gage_driver_info_type. This structure is used to obtain information about the currently selected board by using the gage_get_driver_info routine. Note that this structure can not be used to change any parameters.

index	base I/O port index of CompuScope, used to transfer data from the CompuScope card. The value is typically read from the configuration file GAGESCOP.INC.
segment	absolute address of memory segment, typically read from GAGESCOP.INC.

board_type	numeric constant representing the board type. as defined in the definition file.
max_memory	memory size of CompuScope board (in samples).
max_available_memory	available memory (in samples) in current mode, ie. max_memory in single channel, max_memory / 2 in dual channel.
bank_offset_value	block number where channel B's data starts.
mode	current mode of the board, **GAGE_SINGLE_CHAN** or **GAGE_DUAL_CHAN** as defined in the definition file.
enable_a	current enable status of channel A. Ram is either connected or disconnected to the transceiver by calling **gage_capture_mode** with either **GAGE_INPUT_ENABLE** or **GAGE_INPUT_DISABLE**.
enable_b	same for as enable_a for channel B.
rate	used in **gage_capture_mode**.
multiplier	used in **gage_capture_mode**.
coupling_a	status of the coupling for channel A, either AC or DC. Use **GAGE_DC** or **GAGE_AC** with **gage_input_control**.
coupling_b	same as coupling_a for channel B.
coupling_ext	status of the coupling for the external trigger. Use **GAGE_AC** or **GAGE_DC** with **gage_trigger_control**.
gain_a	status of the gain value for channel A. One of the following predefined constants in the definition file should be used with **gage_input_control** to set this parameter: **GAGE_PM_5_V** **GAGE_PM_2_V** **GAGE_PM_1_V** **GAGE_PM_500_MV** **GAGE_PM_200_MV** **GAGE_PM_100_MV** Note that not all constants can be used with all boards.
gain_b	same for gain_a for channel B.
gain_ext	status of the gain for the external trigger. To set, use either **GAGE_PM_5_V** or **GAGE_PM_1_V** with **gage_trigger_control**. Note that the CompuScope LITE can only use **GAGE_PM_1_V**.
trigger_depth	number of points captured after the trigger event.
trigger_level	level at which trigger event occurred. Can be any value between 0 and 255, with 0 as the minimum and 255 as the maximum..
trigger_slope	status of the current trigger slope. To set, use either **GAGE_POSITIVE** or **GAGE_NEGATIVE** with a call to **gage_trigger_control**.. This has no effect when using a software trigger.
trigger_source	status of current trigger source. To set, either **GAGE_CHAN_A**, **GAGE_CHAN_B**, **GAGE_EXTERNAL** or **GAGE_SOFTWARE** with a call to **gage_trigger_control**.
trigger_res	resolution of trigger depth. On a CS6012, CS1012, CS250 or CS225, this is 64 bytes. Therefore trigger_depth is always a multiple of 64. Similarly, for a CSLITE trigger depth must be a multiple of 16. For a CS220, trigger_res is 0, which means that the trigger_depth must be a power of 2.
multiple_recording	flag to determine if multiple record is currently in operation.
sample_offset	the value of the data returned from the card that equals 0 volts. This value is 128 for 8 bit cards and 0 for 12 bit cards.

sample_resolution	number of levels between 0 and positive full scale or 0 and negative full scale. This value is 128 for 8 bit cards and 2048 for 12 bit cards.
sample_bits	number of sample bits in the installed CompuScope card(s). This will be 8 for an 8 bit card and 12 for a 12 bit card.
external_clock_delay	current clock delay used to synchronize an external clock with the CompuScope card. Should be 0 if the sample rate is greater then 5 MHz or the greater of 1 or 10,000 / external_clock_rate otherwise.
external_clock_rate	current external clock rate when using an external clock. This value is the sample rate in single channel mode and twice the sample rate in dual channel mode.
sample_rate	current sample rate.
memptr	pointer to the segment:offset address of the start of CompuScope ram.

Note that these definitions are defined in GAGE_DRV.H, GAGE_DRV.PAS and GAGE_DRV.BAS for C, Pascal and Basic programs respectively. These files can be found on the diskette and should be referred to for the values of the predefined constants. These predefined constants should be used rather then their numeric values to ensure future compatibility with the GAGESCOPE drivers.

* Prior to version 2.40 of the drivers, trigger_level was represented as a byte.
* Prior to version 2.61 of the drivers, trigger_address, busy, ram_full and triggered specified.
* The version 2.70 structure will maintain this layout. Additional variables will be appended to the structure as required by the driver to support additional features or different versions of hardware.

The normal method of obtaining information about the driver is accomplished via this structure. The following code segment illustrates the process:

```
if (gage_select_board (1))  {
    gage_driver_info_type   gdi;
    gage_get_driver_info (&gdi);
/*  Use the information from the driver.  */
}
```

In Pascal, this would be:

```
gdi:        gage_driver_info_type;
if (gage_select_board (1) = 1) then
   gage_get_driver_info (gdi);
```

In Visual Basic:

```
gdi As gage_driver_info_type
ret = gage_select_board (1)
If ret = 1 Then
   CALL gage_get_driver_info (gdi)
```

In Quick Basic:

```
gdi As gagedriverinfotype
ret = gageselectboard (1)
If ret = 1 Then
   CALL gagegetdriverinfo (SEG gdi)
```

The driver defines the variable **gage_driver_info** as follows:

```
gage_driver_info_type       gage_driver_info;
```

and has the equivalent **extern** reference in the **GAGE_DRV.H** file. For C programs, the previous code segment can be re-written to use the static version of the variable.

```
if (gage_select_board (1))  {
   gage_get_driver_info (&gage_driver_info);
/*  Use the information from the driver.  */
}
```

The advantage of using the predefined **gage_driver_info** variable is that it will always be defined correctly of the appropriate size and be available to the whole application program and maintain its value between calls and different procedures.

Driver structure for board functionality and operation.

The second structure is defined as the **gage_board_node** and ***gage_board_type** structures. This linked list structure is used by the driver to maintain information on the installed CompuScope hardware. The structure can be used by the application program, however, the structure is subject to change *without* notice and can cause the program to crash if improperly used. There is a good performance advantage when using the internal driver structure elements. **gage_current_card**, which is a **gage_board_type** pointer, always points to the current CompuScope card being maintained by the driver. The **gage_boards_master** pointer is of the same type and always points to the first CompuScope board, which is the master in a master/slave installation. Single card installations will have **gage_current_card** and **gage_boards_master** point to the same driver node. The **gage_current_card** pointer is changed during a call to the **gage_select_board** routine. It is recommended that program development start without the use of this structure by the application program. After the program is running and there is a need to increase performance then this structure can be used, but please be very careful to only read these structures. Two routines are available to allow Windows applications to access these structures, **gage_get_current_drv_structure** and **gage_select_current_card.** Note that these structures are currently only accessible if you using C for program development.

Memory Organization of the CompuScope

Memory Architecture.

The A/D speeds at which CompuScope cards produce digital data are too fast for the IBM PC bus (ISA bus) to keep up with (the maximum transfer rate for the ISA bus is approximately 1.5 MBytes / Second). As such, all CompuScope boards have high speed, on-board memory to store the digital data for the IBM PC to access it in a post-processing mode.

Interface for ISA Bus

In order to allow optimum data transfer rates from the CompuScope memory to PC's memory or extended memory, the on-board memory is mapped within the memory map of the 80x86 processor, between 640K and 1M (factory default is D000H - D0FFH for CS250, CS225, CS220 and CSLITE and D000H - D1FFH for CS6012 and CS1012).

The CS250, CS225, CS220 and CSLITE all take only 4 kilobytes of memory space between 640K and 1M and the CS6012 and CS1012 take only 8 kilobytes. This memory address is configurable by writing to the on-board segment register, i.e. it is configured by software, not by DIP switches etc. The address is typically set by the configuration program GSINST.EXE and is read in by the driver routine **gage_read_config_file**. This small memory window and software configuration means that there is very little chance of memory conflicts in any PC.

All CompuScope cards have memory depth much greater than 4 KB or 8 KB. As such, the on-board memory is addressed in a segmented manner. The on-board memory is divided into 4 KB or 8 KB blocks as follows :

On-Board Memory Address	**Block Number**
0 to 4095	0
4096 to 8191	1
8192 to 12287	2
12288 to 16383	3
...	...
...	...

Using the above method, any byte of on-board memory can be addressed using the BLOCK NUMBER and an OFFSET.

BLOCK NUMBER is a number between

0 and (Memory Depth / 4096) for CS250, CS225, CS220 and CSLITE, or

0 and (Memory Depth / 8192) for CS6012 and CS1012.

OFFSET is a number between

0 and 4095 for CS250, CS225, CS220 and CSLITE.

0 and 8191 for CS6012 and CS1012.

Note that the CS6012 and CS1012 return a 16 bit word for each sample, so a block really contains 4096 samples for all boards.

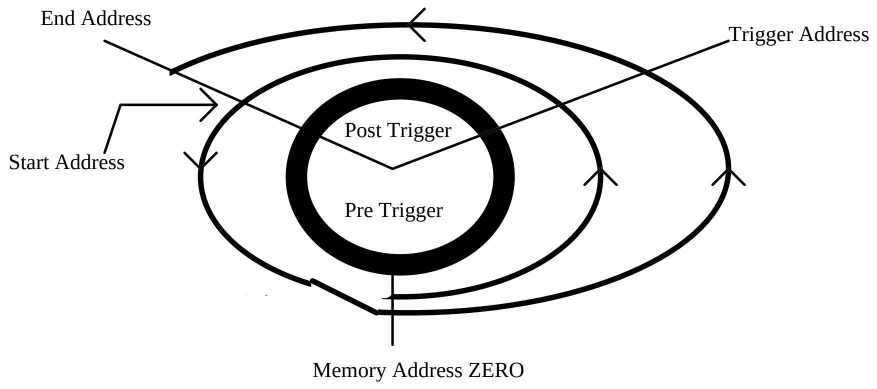
A/D Data Storage

The data coming out of the A/D converters is stored in the on-board Static Memory (SRAM) which is configured as a circular buffer. A circular buffer is used to guarantee that the system will keep on capturing data indefinitely until a trigger event is detected.

The sequence of events is as follows :

> 80x86 processor tells the CompuScope to GET DATA using the Get Data bit.
>
> BUSY flag is set by the CompuScope. PC bus is denied any further access to the on-board memory.
>
> The on-board memory counters initialize to ZERO and start counting up, thereby starting data storage at memory address ZERO.
>
> The system waits for a trigger event to occur while it is storing data in the on-board memory. This data is called Pre-Trigger data.
>
> Once the trigger event is received, a specified number of Post Trigger points is captured. The number of Post Trigger Points can be specified by writing to a register on the CompuScope.
>
> After storing the specified number of Post Trigger Points after receiving the trigger event, acquisition is stopped, BUSY flag is reset and PC bus is allowed access to the on-board memory.

A graphical representation of the above sequence is as follows :



In the diagram above, the circular memory buffer is shown as an annulus with the physical memory address ZERO at the bottom. Data storage is shown as a spiralling line going counter-clockwise.

Storage starts at address ZERO and keeps on writing into the memory until it is filled (the spiralling line completes a circle) and then starts overwriting the data stored in addresses ZERO, 1, 2, ...

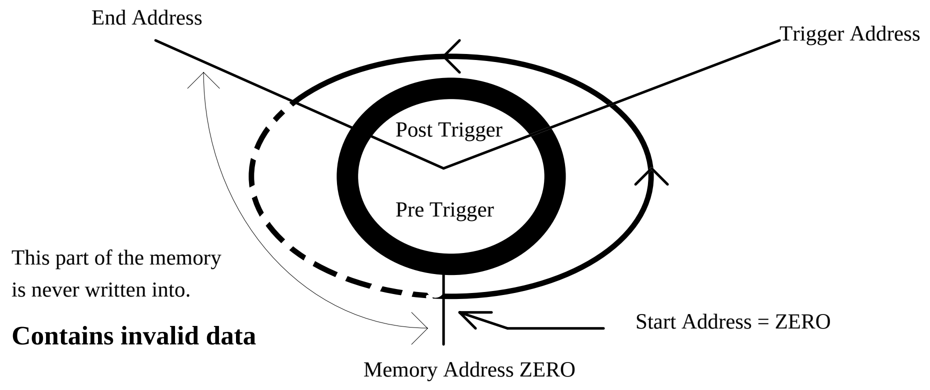
Once a trigger event is detected, the address to which the data was being written into is tagged as the Trigger Address, a specified number of Post Trigger points are captured and then the acquisition is stopped.

The memory address at which the acquisition is stopped is designated as the End Address and the address after that one is called Start Address.

Now, Pre Trigger data lies between Start Address and Trigger Address and Post Trigger data between Trigger Address and End Address.

It is clear from the diagram shown above that memory address ZERO is not necessarily the first point, or Start Address, of the signal being captured. In fact, the physical address ZERO has very little significance in such a system, as the trigger can happen at any time.

One case in which ZERO is the Start Address is when a trigger is received right after the 80x86 tells the CompuScope to GET DATA and data storage stops before the memory has been filled up. This situation is illustrated below :



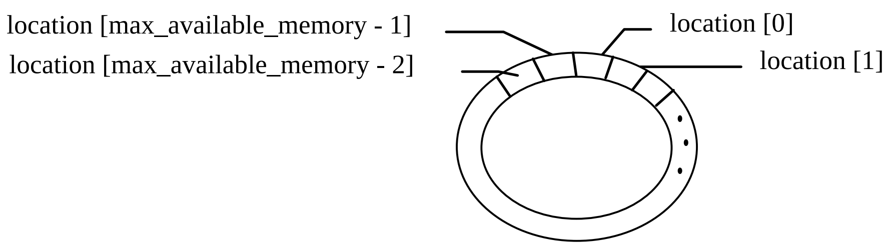
This condition can be detected by looking at the RAMFULL bit in the STATUS register. This bit is reset to ZERO when a GET DATA command is issued and is set to ONE when the memory counters overflow from FFFF to ZERO, for example.

In this case, Pre-Trigger data lies between ZERO and Trigger Address and Post Trigger between Trigger Address and End Address.

Memory: Programmer's View.

The total memory of a CompuScope card is divided into two channels, A and B. In dual channel mode, each channel gets half the memory of the card. For example, in a 256K card channel A will have the first 128K and channel B the second 128K. In single channel mode, all 256K will belong to channel A but it is interleaved between the data space for channel A and B. Even addresses are from channel A and odd addresses from channel B. The total available memory to a channel can be retrieved through the **gage_driver_info_type** field, max_available_memory. Each channel's memory is further divided into 4K regions called blocks.

The memory of each channel of the CompuScope cards is organized as a circular buffer. The addresses returned from the routine **gage_calculate_addresses** represent the trigger address, starting address and ending address of the captured data. These addresses can then be used to loop through the desired locations to retrieve the captured data one sample at a time using the **gage_mem_read_xxxx** functions. Although the **gage_mem_read_xxxxxx** routines return one point at a time, internally they still transfer a full four kilosample block of memory. This speeds up the access to the CompuScope memory dramatically for data that is accessed sequentially. This is the preferred method for most applications that need to use discrete data points since the data is corrected by the driver for any idiosyncrasies in memory layout and polarity (the data is always returned as an integer with the smallest value representing the largest voltage and the largest data sample value representing the smallest voltage). Because of the circular buffer, care must be taken to wrap around at the end of the memory address space. This is usually done by using the MOD function with the index and gdi.max_available_memory. If the memory depth is a power of 2, then a faster way is to use the AND function with gdi.max_available_memory - 1. See the example routines and programs for more information on how this is done.


location [max_available_memory - 1]
location [max_available_memory - 2]
location [0]
location [1]

If one of the block transfer routines, **gage_trigger_view_transfer**, **gage_32k_to_buffer** or a user-defined routine is used, then the memory is split between channel A and channel B. In single channel mode, the data is interleaved between the two channels, with the first data point in channel A, the second in channel B, the third in A, etc., as shown below. See the Sample Routines section (transfer_data) for examples of uninterleaving the data.

Channel A	Data 0	Data 2	Data 4	Etc.
Channel B	Data 1	Data 3	Data 5	Etc.

Because the block transfer routines do not correct for any idiosyncrasies of the memory layout of the CompuScope hardware, this must be taken into account. The data returned from the CS220's internal buffers should be inverted (one's complement) to match the polarity of the other CompuScope boards. That is, before inversion, the values returned have the smallest value (0) representing the smallest voltage and the largest value (255) representing the largest voltage. In addition, on the CS220, the least significant bit of the address will be flipped under the following circumstances:

a) You are in single channel mode and the board has 1 M of memory or greater, then the least significant bit of channel A's data will be flipped. This situation is shown below:

Logical Address	CHA	CHB	Actual CHA	Actual CHB
0	0	-	1	-
1	-	0	-	0
2	1	-	0	-
3	-	1	-	1
4	2	-	3	-
5	-	2	-	2
6	3	-	2	-
7	-	3	-	3
8	4	-	5	-
etc.	etc.	etc.	etc.	etc.

Note that the numbers in the columns represent the addresses where the samples are located. When using the block transfer routines, you would typically use one buffer for channel A and one for channel B. For boards other than the CompuScope 220, the data is returned according to the second and third columns in single channel mode. That is, the first data point is in channel A's data space, the second in channel B's, the third in A's, etc. For the CompuScope 220, the last two columns represent how the data is returned. The addresses in channel A have the least significant bit flipped.

b) You are in dual channel mode, the CompuScope 220 has less than 1M of memory and you are capturing channel B's data. The least significant bit gets flipped as follows:

Address	Binary	Binary (after bit-flipping)	New Address
0	0x0000	0x0001	1
1	0x0001	0x0000	0
2	0x0010	0x0011	3
3	0x0011	0x0010	2

In this case, the first sample will be in location 1, the second sample in location 0, the third in location 3, the fourth in location 2, etc.

As mentioned above, the CompuScope cards internally transfer a 4 kilosample block of memory. Sample size is one byte for 8 bit cards and 12 bits (sign extended to 16 bits) for 12 bit cards. Specific blocks can be accessed by using the driver routines **gage_set_block_number.** The memory blocks are organized with all of channel A's data occupying the first set of four kilobyte blocks up to the total size of one channel's maximum memory size. The same number of blocks are available for channel B. For example, a 16K CSLITE would have 16K / 4 kilosamples = 4 blocks. In dual channel mode half the blocks would belong to channel A and half to channel B. Block 0 and block 1 would belong to channel A and block 0 + bank_offset_value and block 1 + bank_offset_value would belong to channel B. In single channel mode all the banks would belong to channel A. The value of bank_offset_value is defined in the source files for each CompuScope card and is available through the **gage_driver_info_type** structure.

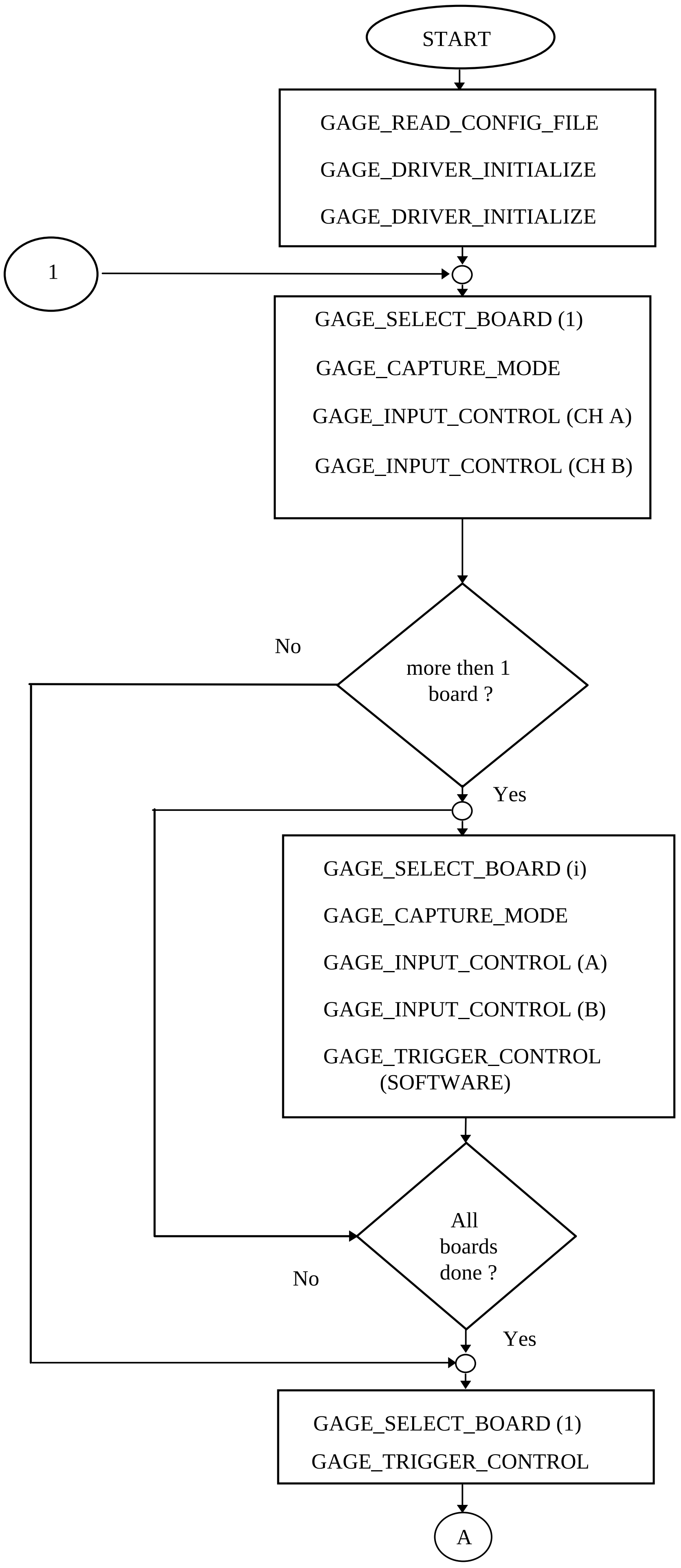
GageScope signal files are saved as a memory image of the CompuScope card's buffers. Therefore, reading a signal file is much like reading the CompuScope memory. The trigger address, trigger depth and sample depth are available from the header of the signal file. Sample depth is equal to the available memory size if the Save All option was used in GageScope. This would be the same as the max_available_memory field in the **gage_driver_info_type** structure. In this case, reading the file is the same as reading directly from CompuScope memory and the same precautions concerning buffer wraparound. If the file was saved with the Normal or User Defined options, the file will be normalized and the number of post-trigger samples will be ending address - trigger address + 1. If the starting address < = trigger address < = ending address, then the file can be considered to be normalized. See the sections on diskfile.h and the GageScope signal file format for more information.

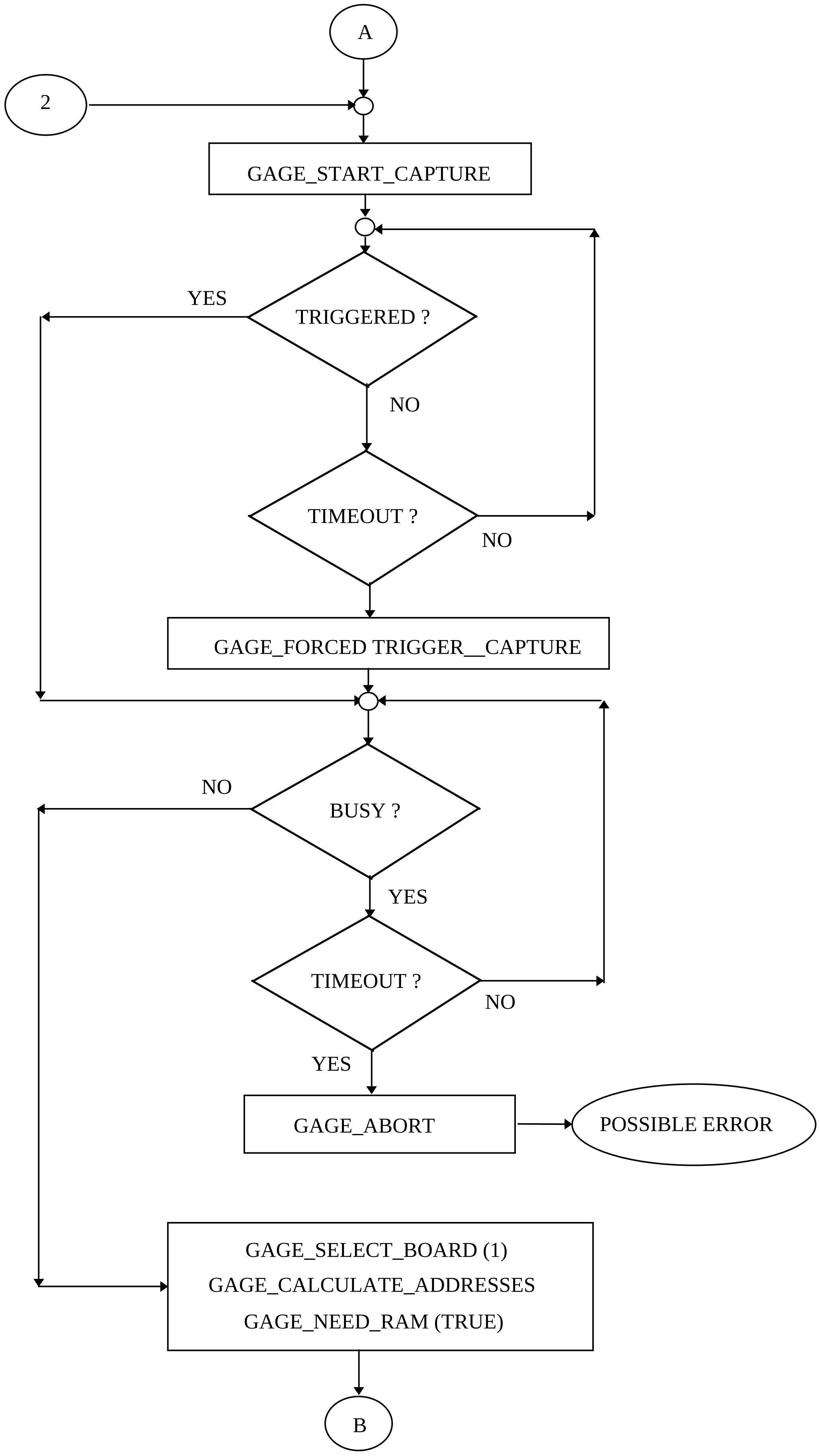
Application Development

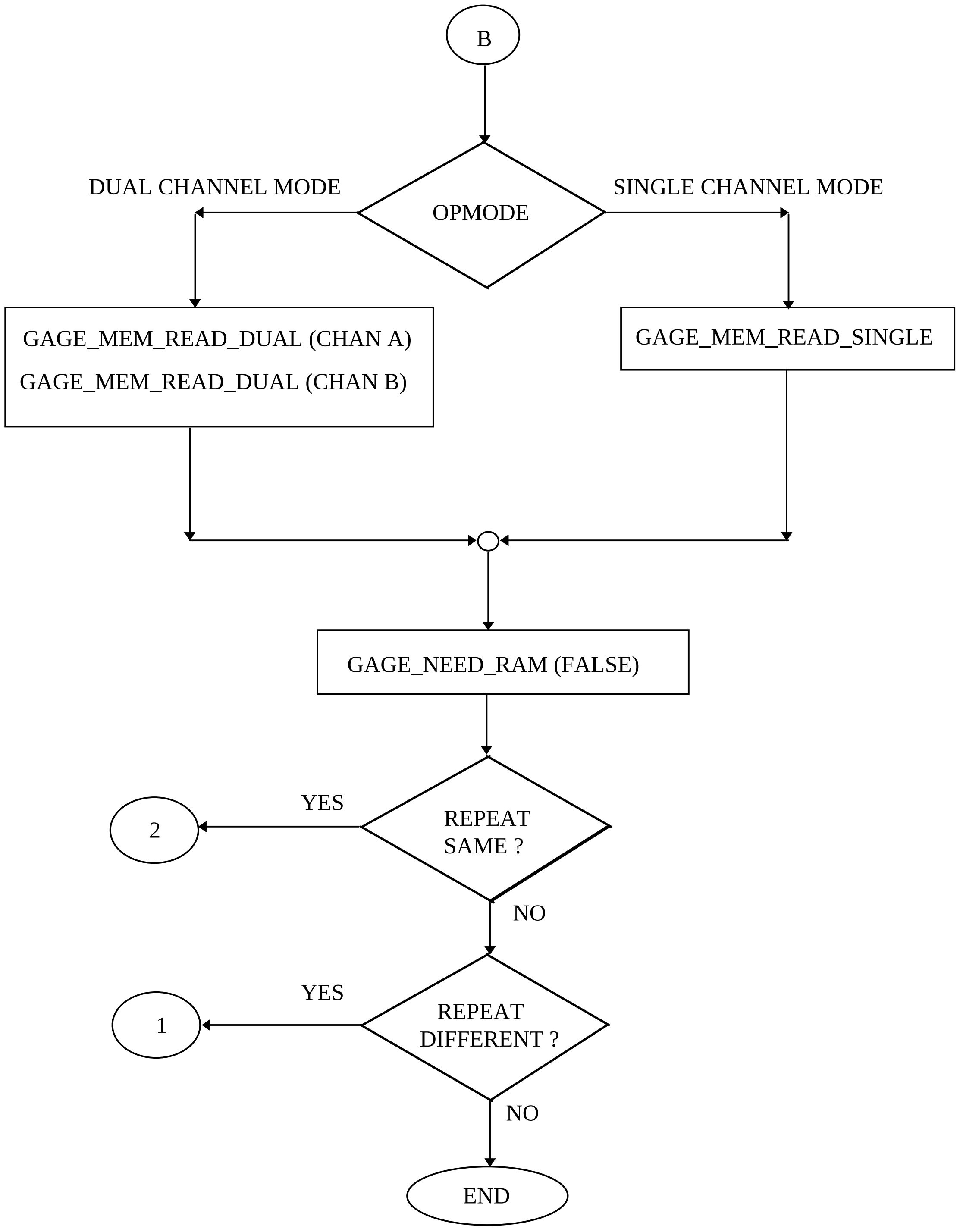
Application development using the CompuScope drivers generally follow the same basic algorithm regardless of the operating system or language being used. The following flow chart and tutorial included in this section demonstrate the basic steps needed to initialize the driver and the hardware, start the data acquisition and access the captured data on the CompuScope card. Note that most of the sample routines and programs discussed in this manual are general in nature. They will work with single or multiple board systems and will work with any CompuScope card. Exceptions to this will be noted when discussing the sample programs. Some optimization is possible by writing your program for one specific board. Optimization is also possible in the transfer of data by using the block transfer routines, **gage_trigger_view_transfer** and **gage_32k_to_buffer**, or by utilizing your own memory transfer routines. See the sample program, ACQ2DISK.C, for an example of this. Another place to look for examples of this is the driver code, which shows how routines such as **gage_trigger_view_transfer** are implemented.

The flowchart on the following pages shows the general form an application must follow to use the CompuScope drivers. The chart is shown on three pages, with the first part showing driver initialization, the second data capture and the third part showing data transfer. The next section describes how to build a simple application using the CompuScope drivers and the steps involved. The complete source code for this tutorial is on the distribution diskettes.

A tool that is very useful in writing application programs for the CompuScope drivers is the GAGESCOP.EXE program that comes with your CompuScope card. GAGESCOP is a general purpose data acquisition and display program that has been written using the C drivers and is thoroughly tested. It is often used during program development to determine if a problem is a hardware bug, a driver bug or an application bug. In general, if you are getting good results with GAGESCOP and not with your application program, the problem is in the program and not in the drivers or with the hardware.


START
GAGE_READ_CONFIG_FILE
GAGE_DRIVER_INITIALIZE
GAGE_DRIVER_INITIALIZE
1
GAGE_SELECT_BOARD (1)
GAGE_CAPTURE_MODE
GAGE_INPUT_CONTROL (CH A)
GAGE_INPUT_CONTROL (CH B)
No
more then 1 board ?
Yes
GAGE_SELECT_BOARD (i)
GAGE_CAPTURE_MODE
GAGE_INPUT_CONTROL (A)
GAGE_INPUT_CONTROL (B)
GAGE_TRIGGER_CONTROL (SOFTWARE)
All boards done ?
No
Yes
GAGE_SELECT_BOARD (1)
GAGE_TRIGGER_CONTROL
A


A
2
GAGE_START_CAPTURE
TRIGGERED ?
YES
NO
TIMEOUT ?
NO
GAGE_FORCED TRIGGER__CAPTURE
BUSY ?
NO
YES
TIMEOUT ?
NO
YES
GAGE_ABORT
POSSIBLE ERROR
GAGE_SELECT_BOARD (1)
GAGE_CALCULATE_ADDRESSES
GAGE_NEED_RAM (TRUE)
B


B
DUAL CHANNEL MODE
OPMODE
SINGLE CHANNEL MODE
GAGE_MEM_READ_DUAL (CHAN A)
GAGE_MEM_READ_DUAL (CHAN B)
GAGE_MEM_READ_SINGLE
GAGE_NEED_RAM (FALSE)
YES
2
REPEAT SAME ?
NO
YES
1
REPEAT DIFFERENT ?
NO
END

A simple application using the CompuScope C drivers

The first step in writing any program is to establish what function the program is to perform. Our sample program, AMEANRMS.C, will acquire 512 samples and calculate the mean and the RMS average of the captured data. The example code will be shown only in C for brevity. Similar programs for Protected Mode Pascal and Quick Basic are in the files AMEANRMS.PAS and AMEANRMS.BAS on the distribution diskette. The results will be presented in volts. The next step is to establish what is required to achieve this goal. The basic algorithm for any data acquisition application based on Gage's CompuScope cards is:

1) Initialize program structures and variables.
2) Initialize the driver and the CompuScope hardware.
3) Initialize the board with the desired settings.
4) Start the acquisition.
5) Wait for the acquisition to be completed.
6) Access the data on the CompuScope card.
7) Use the acquired data.
8) Another acquisition is required? No, go to step 11.
9) Yes. Change the board settings? No, go to step 4.
10) Yes. Go to step 3.
11) End program.

Useful program structures and variables have been included in the sample program. Three important structures, **boarddef**, **srtype** and **irtype** , are defined in the file **STRUCTS.H** and initialized in the file **STRUCTS.C**, both of these files are located on the distribution disk for the CompuScope C drivers. The **boarddef**, **srtype** and **irtype** structures are reproduced here, however, the initialization of the structures is too large to be listed below but the external references follow the definitions.

```
typedef struct  {
        int16           opmode, srindex;        /*  For gage_capture_mode.  */
        int16           range_a, couple_a;      /*  For gage_input_control.  */
        int16           range_b, couple_b;      /*  For gage_input_control.  */
        int16           source, slope, level;   /*  For gage_trigger_control.  */
        int16           range_e, couple_e;      /*  For gage_trigger_control.  */
        int32           depth;                  /*  For gage_trigger_control.  */
} boarddef;
typedef struct {
        int16           rate;                   /*  For gage_capture_mode.  */
        int16           mult;                   /*  For gage_capture_mode.  */
        uInt32          sr_flag;                /*  Board supports sample rate.  */
        float           sr_calc;                /*  Time between samples (in ns).  */
        char            *sr_text;               /*  Sample rate text.  */
} srtype;
typedef struct {
        int16           constant;               /*  For gage_input_control.  */
        uInt32          gf_flag;                /*  Flag to see if board supports input range.  */
        double          gf_calc;                /*  Voltage amplitude.  */
        char            *gf_text;               /*  Input range text.  */
} irtype;

extern boarddef   board;
extern srtype     srtable[];
extern irtype        ranges[];
```

The **boarddef** structure defines all of the settings that may be changed on the CompuScope card. The **srtable** structure defines all of the possible sample rate settings that any of the CompuScope card may use. The top 16 bits of the flag is used to specify if the sample rate is available single channel and the bottom 16 bits are used to specify that the sample rate is available when acquiring data in the dual channel mode. The **irtype** structure similarly defines which input ranges are available for the installed CompuScope hardware. The structure has a flag field whose operation is similar to the flag used by the **srtype** structure.

Initializing the CompuScope driver and hardware is accomplished in two parts. First, it must be established where the installed hardware is located in the I/O and memory maps and then to actually initialize the driver/hardware. There are two methods to define the location of the hardware. First, use the **GAGESCOP.INC** file as created by the **GSINST.EXE** program that accompanies both GageScope and the drivers. This program creates a file that contains the I/O Index and the Memory Segment for each CompuScope board in the system. The **GSINST** program can be used to test the desired location for compatibility between the host computer and the CompuScope hardware. The driver can read this file, or any other file with the same format, with a call to the **gage_read_config_file** routine which initializes the **gage_board_location** array. The second method is to use the **gage_set_records** routine to initialize the **gage_board_location** array directly. This method is faster and does not require the **GAGESCOP.INC** file to be maintained. Regardless of which method was used to initialize the **gage_board_location** array the next step is to initialize the driver and the board(s) using the **gage_driver_initialize** routine. This routine should be called twice when initializing CompuScope boards in a master/slave system to "pre-initialize" the registers on the slave boards. Once the driver has been initialized these routines do not need to be called again.

```
int16      init_driver_hardware (void)
{
           if (gage_read_config_file ("GAGESCOP.INC", gage_board_location) < 0)  {
                      printf ("Errors in GAGESCOP.INC, defaults used.\n");
                      gage_set_records (gage_board_location, 0, 0xd000, 0x0200, 0, 0);
                      gage_set_records (gage_board_location, 1, 0, 0, 0, 0);
           }
           gage_driver_initialize(gage_board_location, GAGE_MEMORY_SIZE_TEST);
           if (!gage_driver_initialize(gage_board_location, GAGE_MEMORY_SIZE_TEST))  {
                      uInt16    segment, index, board_type, status;
                      gage_get_records (gage_board_location, 0, &segment, &index, &board_type, &status);
                      printf ("No CompuScope boards found, Error code = %02x.\n", status);
                      return (1);
           }
           gage_get_driver_info (&gdi);
           current_board_type = B_T_DOUBLE_UP_BITS(gdi.board_type);
           current_memory_size = gdi.max_memory;
           printf ("CompuScope %s board found.\n", board_type_and_size_to_text (gdi.board_type,
                      gdi.max_memory));
           if (gdi.board_type & (GAGE_ASSUME_CS1012 | GAGE_ASSUME_CS6012))  {
                      sample_offset = 0;
                      sample_resolution = 2048;
           }else{
                      sample_offset = 128;
                      sample_resolution = 128;
           }
           gage_select_board(1);
           return (0);
}          /*  End of init_driver_hardware ().  */
```

Preparing the CompuScope card for data capture can be done at any time prior to starting a data acquisition sequence. An attempt to change the board settings after data capture has commenced can result in a poor and unpredictable acquisition. With that said, there could be a set of circumstances that will require changing these settings "on the fly". There are three routines that are used to set the board parameters.

The first is the **gage_capture_mode** routine that sets the CompuScope operating mode. The single channel operation mode is used when the full memory depth of the CompuScope is required or when the maximum sample rate is required. If these two requirements are not required it is suggested that the dual channel operation mode be used to simplify data transfer. The sample rate is also set using this routine. There are two parameters used when setting the sample rate. The first is the base rate and the second is the multiplier. These values should use the pre-defined constants, as the **srtable** definition illustrates.

The second routine is the **gage_input_control** routine which initializes the input parameters for each channel. This routine must be called once for each channel that needs to have its input parameters updated. If the settings for channel A need changing and the single channel operation mode is being used then channel B must be initialized to the same settings.

The last routine, **gage_trigger_control**, is used to initialize the trigger circuitry. The external trigger source is configured with this routine along with the source, slope and level of the trigger event. The number of samples to acquire after the trigger event is also set by this routine.

```
void	prepare_for_capture (boarddef *board)
{
	gage_capture_mode (board->opmode, srtable[board->srindex].rate,
		srtable[board->srindex].mult);
	gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board->couple_a,
		board->range_a);
	if (board->opmode == GAGE_SINGLE_CHAN)
		gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
			board->couple_a,board->range_a);
	else
		gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
			board->couple_b,board->range_b);
	gage_trigger_control (board->source, board->couple_e, board->range_e, board->slope,
		board->level, board->depth);
}	/*  End of prepare_for_capture ().  */
```

The board settings will remain in effect between acquisitions, therefore it is only necessary to call these routines again when changes to the board parameters are required.

Starting data acquisitions with each channel synchronized to the starting impulse. The **gage_start_capture** routine is used for this purpose. This routine takes a parameter that is used to automatically start the data acquisition when the trigger source has been set to trigger under software control. The software trigger may also be issued by calling **gage_software_trigger** after of this routine, when the passed parameter is zero.

```
	gage_start_capture (board.source == GAGE_SOFTWARE);
```

When this routine completes, the board has started it next data acquisition sequence. This routine is a good candidate for optimizations, in time critical measurements and interrupt service routines, since it does a lot of extra work ensuring that the board/system is in a known state before trying to start the acquisition.

When is the acquisition complete after data capture has started? The program must determine when it will stop, and if it is not coming to a timely finish, what to do regain control of the CompuScope card. There are two schools of thought on the issue of time-out in data acquisition.

One states that the data acquisition system will wait indefinitely for the trigger event to occur. This is useful when the signal to be captured will cause the trigger when it is in error. Causing a time-out to occur in this case could cause the program to miss the actual event that may occur when the board is busy being read by the application program. The second case states that reading the acquired data to determine what the input signal is doing at any given time is of more importance than capturing every error event.

Often a combination of these two approaches is required in a data acquisition system. Two driver routines **gage_triggered** and **gage_busy** are used to monitor the progress of the trigger event and the completion of the current data acquisition respectfully. Two other routines will control the data acquisition when a time-out is to occur. Using **gage_forced_trigger_capture** will force a trigger event to capture the current data. Use **gage_abort_capture** when the current acquisition is to be terminated prematurely.

```
time_out = biostime (0, 0L) + TRIG_TIMEOUT;
while (!gage_triggered ())
        if (biostime (0, 0L) > time_out)
                gage_forced_trigger_capture (board.source);
time_out = biostime (0, 0L) + BUSY_TIMEOUT;
while (gage_busy ())
        if (biostime (0, 0L) > time_out)
                gage_abort_capture (board.source);
```

Examining the CompuScope data after a call to **gage_forced_trigger_capture** will reveal that the expected trigger event was not encountered. Similarly, when **gage_abort_capture** is called, the data will often contain a discontinuity where the acquisition was not allowed to complete normally.

Getting to the CompuScope data can be done when the board is no longer “busy”. First, the location of the sampled data in the CompuScope memory is determined with a **gage_calculate_addresses** call. This routine returns the starting, trigger and ending addresses. The use of this routine to calculate these addresses greatly simplifies access to the data. The CompuScope memory is organized as a circular buffer, so direct comparisons of the addresses should be avoided. However, **gage_normalize_address** can be used for these comparisons and also to determine the actual amount of valid pre and post trigger data. Next, the CompuScope memory must be connected to the PC with a call to the **gage_need_ram** routine. There are two methods of transferring data from the CompuScope cards. The first is to use the **gage_mem_read_xxxxxx** routines which when passed the desired sample location will return the corresponding data value. The second method uses either one of the built-in block transfer routines such as **gage_trigger_view_transfer** or **gage_32k_to_buffer** or a specialized derivative based on the source code drivers. Both methods have advantages and disadvantages. Although the **gage_mem_read_xxxxxx** routines return one point at a time, internally they still transfer a full four kilosample block of memory. This speeds up the access to the CompuScope memory dramatically for data that is accessed sequentially. This is the preferred method for most applications that need to use discrete data points since the data is corrected by the driver for any idiosyncrasies in memory layout and polarity (the data is always returned as an integer with the smallest value representing the largest voltage and the largest data sample value representing the smallest voltage). If the application has demanding requirements for data through-put, the block transfer routines may be used to access the sampled data. The memory organization splits the RAM between channel A and B. When the single channel mode is used, the two areas of RAM are interleaved on the CompuScope card and the PC has to access the memory in two distinct areas. The application must then access the samples alternatively from the data space for channel A and the data space for channel B.

Using the acquired data our sample program grabs 512 points from the data buffer to calculate the mean and RMS values of the captured data. The two calculations are done at the same time.

```
int16     calculate_mean_and_rms (long nsamples, double *mean, double *rms, boarddef *board)
{
          int16     (*read_data) (int32 index);
          int       i;
          int32     trigger, start, end, address;
          double    value, input_range;
          *mean = *rms = 0.0;
          gage_get_driver_info (&gdi);
          input_range = calc_input_range (board->range_a);
          if (board->opmode == GAGE_SINGLE_CHAN)
                    read_data = gage_mem_read_single;
          else
                    read_data = gage_mem_read_chan_a;
          gage_calculate_addresses (GAGE_CHAN_A, board->opmode, srtable[board->srindex].sr_calc,
                    &trigger, &start, &end);
          if (gage_normalize_address (trigger, end, gdi.max_available_memory) < nsamples)
                    return (0);
          gage_need_ram (TRUE);
          for (i = 0, address = trigger ; i < nsamples ; i++)  {
                    value = (double)(sample_offset - read_data (address)) / (double)(sample_resolution) *
                              input_range;
                    address = (address + 1) & (gdi.max_available_memory - 1);
                    *mean += value;
                    *rms += (value * value);
          }
          *mean /= i;
          *rms = sqrt (*rms / i);
          gage_need_ram (FALSE);
          return (1);
}         /*  End of calculate_mean_and_rms ().  */
```

What to do next? The sample program will acquire once, calculate and display its results and stop. Generally, either the number acquisitions is fixed and therefore a "counting loop" is required or the data acquisitions are to continue indefinitely. Modifying the sample program to loop is easily accomplished when two facts are known. One, if the board settings are to be changed then continue by jumping to the **prepare_for_capture** routine. Two, if another acquisition is to be performed then jump to the **gage_start_capture** routine. Otherwise when no more acquisitions are required perform any required house keeping and return to DOS. Our sample program simply exits to DOS returning either a zero or a one (okay or error respectfully) for the ERRORLEVEL batch commands.

The source code for the sample program is contained in the files **AMEANRMS.C** and **DRV_SUPP.C** on the CompuScope driver distribution disks. The project files to build the program are called MEANRMSB.PRJ (for Borland compilers), MEANRMSM (for Micorsoft) and MEANRMSW (for Watcom). Similar programs can be found as **AMEANRMS.PAS** and **AMEANRMS.BAS** for Pascal and Basic respectively.

Global Routines: Group One.

This group of routines comprise the core of the callable functions that control the CompuScope series of data acquisition cards. These routines are all that a programmer needs use to get the board up and running as given by the example programs and the code segment examples in this section. The next section dealing with group two global routines will discuss some optimizations that can be performed when using the routines in that group. However, to get started these routines are all that is needed when used in a manner consistent with the examples in the routine descriptions.

The following routines are for application program use and are subject to change. However, in future releases of the drivers any changes will be documented appropriately. If a name change occurs or parameter list changes are required then these routines, as they exist, will be maintained either as macros or complete routines to avoid future compatibility problems. These changes, as they occur, will be listed in the file "RELEASE.DOC" on the distribution disk for the drivers. Please check the distribution disk for any additional text files that outline other pertinent information relating to the CompuScope drivers.

gage_32k_to_buffer

Syntax

C:
```
#include <gage_drv.h>
void gage_32k_to_buffer (uInt8 far *buffer, int16 high_half, int16 start_block, int16 far
*blocks_transfered);
```

Visual Basic:
```
Sub gage_32k_to_buffer (buffer As Integer, ByVal high_half As Integer, ByVal start_block As Integer,
                        blocks_transfered As Integer)
```
Quick Basic:
```
Sub gage32ktobuffer (SEG buffer As Any, ByVal highhalf As Integer, ByVal startblock As
                        Integer, SEG blockstransfered As Integer)
```

Pascal:
```
procedure gage_32k_to_buffer (var buffer: uInt8; high_half, start_block: int16;
                        var blocks_transfered: int16);
```

Remarks

gage_32k_to_buffer is used to copy the CompuScope memory space to the supplied buffer 32 kilobytes at a time. This routine works correctly with all CompuScope boards regardless of the size of memory installed. **buffer** is a byte array that must be at least 32 kilobytes in length. **high_half** is a flag used to select either the low half of memory (channel A's data) or the high half of memory (channel B's data). When high_half is 0 then the data associated with channel A is used and when high_half is 1 the data from channel B is moved to the buffer. **start_block** is the block at which the data transfer is to start and is used to offset the start of the CompuScope data buffers for transferring more than 32 kilobytes of data. **blocks_transfered** is the actual number of 4K blocks of data transferred. The number **blocks_transfered** can be used to update **start_block** for the next block transfer. If the **blocks_transfered** parameter is not equal to 8, the end of data for transfer has been reached. Please note that the CS6012 and CS1012 transfer 4 8K blocks (2 bytes per sample) rather than 8 4K blocks. Note that all boards transfer 4K samples, the CS250, CS225, CS220 and CSLITE have a sample size of 1 byte while the CS6012 and CS1012 have a sample size of 2 bytes (12 bits sign extended to 16 bits).

Return Value

None.

See also

gage_trigger_view_transfer. Sample Routines: write_cs_data.

Examples

C:
```
gage_32k_to_buffer (buffer, 0, start_block,   &blocks_transfered);
```

Visual Basic:

```
Call gage_32k_to_buffer (buffer(0), 0, start_block, blocks_transfered)
```

Quick Basic:

```
CALL gage32ktobuffer (buffer(0), 0, startblock, blockstransfered)
```

Pascal:

```
gage_32k_to_buffer (buffer, 0, start_block, @blocks_transfered);
```

Note: Because Visual Basic has no variables of type byte, the buffer must be an integer array. This is no problem when dealing with the CS6012 or CS1012, which have a sample size of 2 bytes (12 bits sign extended to 16 bits). With the CS250, CS225, CS220 and CSLITE, which have a sample size of 1 byte, the high and low bytes of the integer must be extracted. This can be done as follows:

```
j = 0
For i = 0 To (Size Of Buffer \ 2) - 1
    temp = CLng (65535) And CLng (buffer(i))
    temp_buffer(j) = temp And  255   ' extract the high byte and store as an integer
    temp_buffer(j+1) = (temp \ 256) And 255   ' extract the low byte and store as an integer
    j = j + 2
Next
```

In Quick Basic, you can use the DEF SEG and PEEK functions to extract a byte.

gage_abort_capture

Syntax

C:
#include <gage_drv.h>
void gage_abort_capture (int16 reset_trigger_source);

Visual Basic:
sub gage_abort_capture (ByVal reset_trigger_source As Integer)

Quick Basic:
sub gageabortcapture (ByVal resettriggersource As Integer)

Pascal:
procedure gage_abort_capture (reset_trigger_source: int16);

Remarks

gage_abort_capture is used to regain control of the CompuScope board(s), primarily in the event that a trigger event never occurs. This routine forces the board(s) to a not busy state, thus allowing the board(s) to be re-configured, rearmed and/or the memory to be accessed. The **reset_trigger_source** parameter is used to re-initialize the trigger source after the hardware has been aborted. This is normally set to the initial trigger source used. The **gage_abort_capture** routine should be used instead of **gage_abort** as it works for multiple board installations.

Return Value

None.

See also

gage_abort, gage_busy and gage_forced_trigger_capture. Sample Routines: acquire_check.

Examples

C:
gage_abort_capture (board.source);

Visual Basic:
Call gage_abort_capture (board.source)

Quick Basic:
CALL gageabortcapture (board.source)

Pascal:
gage_abort_capture (board.source);

gage_busy

Syntax

C:
```
#include <gage_drv.h>
int16 gage_busy (void);
```

Visual Basic:
```
function gage_busy () As Integer
```

Quick Basic:
```
function gagebusy% ()
```

Pascal:
```
function gage_busy: int16;
```

Remarks

gage_busy determines if CompuScope is busy capturing data.

Return Value

A non zero or true value is returned, if the board is busy, otherwise a false value is returned.

See also

gage_ram_full and gage_triggered. Sample Routines: acquire_check.

Examples

C:
```
is_busy = gage_busy ();
```

Visual Basic:
```
is_busy = gage_busy
```

Quick Basic:
```
isbusy % = gagebusy%
```

Pascal:
```
is_busy := gage_busy ;
```

gage_calculate_addresses

Syntax

C:

```
#include <gage_drv.h>
void gage_calculate_addresses (int16 chan, int16 op_mode, float tbs, int32 far *trig, int32 far *start,
                               int32 far *end);
```

Visual Basic:

```
Sub gage_calculate_addresses (ByVal chan As Integer, ByVal op_mode As Integer, ByVal tbs As Single,
                               trig As Long, start As Long, ending As Long)
```

Quick Basic:

```
Sub gagecalculateaddresses (ByVal chan As Integer, ByVal opmode As Integer, ByVal tbs As Single,
                               SEG trigaddr As Long, SEG startaddr As Long, SEG endaddr As Long)
```

Pascal:

```
procedure gage_calculate_addresses (chan, op_mode: int16; tbs: single; var trig: int32;
                               var start: int32; var endaddr: int32);
```

Remarks

gage_calculate_addresses returns the three important addresses for the specified channel of the current board to the calling routine. Remember to set the current board, using **gage_select_board**, prior to calling this routine so that the addresses for the proper CompuScope card are retrieved. The CompuScope memory is organized as a circular buffer, therefore use the provided "**gage_normalize_address**" function when comparing addresses to be certain that the address is logically presented and not physically presented which will generate erroneous operation of the code. The **chan** parameter controls which channel's addresses on the current board are calculated and returned (either **GAGE_CHAN_A** or **GAGE_CHAN_B**). **op_mode** is either of the constants **GAGE_SINGLE_CHAN** or **GAGE_DUAL_CHAN**. Use the value that controlled the most recent data capture. **tbs** is the "Time Between Samples" in nanoseconds for the captured signal, for example, if sampling data at 10 MHz then this value would be 100.0, similarly sampled data at 1 KHz would require this value to be set to 1000000.0. ***trig** will be the adjusted trigger address for the type of hardware being used. ***start** will be the first valid address of the most recent data capture. ***end** will be the last valid address of the most recent data capture. This routine is the best method to determine these addresses since it will work with any board.

Return Value

None.

See also

gage_normalize_address. Sample Routines: InitMemPtrs.

Examples

C:

```
gage_calculate_addresses (GAGE_CHAN_A, gdi.mode, tbs, &trig_addr, &start_addr, &end_addr);
```

Visual Basic:

```
Call gage_calculate_addresses(GAGE_CHAN_A, board.opmode, srtable(board.srindex).calc,
                    trigger_address(0), starting_address(0), ending_address(0))
```

Quick Basic:

```
Call gagecalculateaddresses(GAGECHANA, board.opmode, srtable(board.srindex).calc,
          triggeraddress(0), startingaddress(0), endingaddress(0))
```

Pascal:

```
gage_calculate_addresses(GAGE_CHAN_A, board.opmode, srtable[board.srindex].calc,
          trigger_address[0], starting_address[0], ending_address[0]);
```

gage_calculate_mr_addresses

Syntax

C:
#include <gage_drv.h>
int32 gage_calculate_mr_addresses (int32 group, int16 board_type, int32 depth, int32 memory, int16 chan, int16 op_mode, float tbs, int32 far *trig, int32 far *start, int32 far *end);

Visual Basic:
Function gage_calculate_mr_addresses (ByVal group As Long, ByVal board_type As Integer, ByVal depth As Long, ByVal memory As Long, ByVal chan As Integer, ByVal op_mode As Integer, ByVal float As Single, trig As Long, startaddr As Long, endaddr As Long) As Long

Quick Basic:
Function gagecalculatemraddresses& (ByVal group As Long, ByVal boardtype As Integer, ByVal depth As Long, ByVal memory As Long, ByVal chan As Integer, ByVal opmode As Integer, ByVal tbs As Single, SEG trigaddr As Long, SEG startaddr As Long, SEG endaddr As Long)

Pascal:
function gage_calculate_mr_addresses (group: int32; board_type: int16, depth, memory: int32; chan, op_mode: int16; tbs: single; var trig: int32; var start: int32; var endaddr: int32): int32;

Remarks

gage_calculate_mr_addresses returns the three important addresses for the specified channel of the current board to the calling routine. The board must be set to multiple record mode with a call to **gage_multiple_record**. Remember to set the current board, using **gage_select_board**, prior to calling **gage_calculate_mr_addresses** so that the addresses for the proper CompuScope card are retrieved. The CompuScope memory is organized as a circular buffer, therefore use the provided **gage_normalize_address** function when comparing addresses to be certain that the address is logically presented and not physically presented which will generate erroneous operation of the code. The **group** parameter is the current multiple record group, with 1 being the first multiple record group, -1 the last and n the nth multiple record group. A 0 will return the addresses set to all valid samples without regard to groups. If the group is out of range then the last multiple record group is returned. **board_type** is the current board and one of the predefined constants, **GAGE_ASSUME_CSLITE**, **GAGE_ASSUME_CSLITE15**, **GAGE_ASSUME_CS25016**, **GAGE_ASSUME_CS220**, **GAGE_ASSUME_CS225**, **GAGE_ASSUME_CS250**, **GAGE_ASSUME_CS1012** or **GAGE_ASSUME_CS6012**, should be used. The **depth** parameter is the requested depth for each acquisition of the multiple record capture and is the same value as the current trigger depth. **memory** is the total amount of memory available during the multiple record capture and is the full depth of the channel, depending on which mode the card is currently in. This value can be obtained from the **gage_driver_info_type** structure in the **max_available_memory** field.. The **chan** parameter controls whether a hardware of software multiple record is done. If **chan** equals 0, a hardware multiple record address calculation is performed. If **chan** equals -1, a software multiple record address calculation is done. See the C sample program, SWMULREC.C for an example of how to do software multiple record. **op_mode** is either of the constants **GAGE_SINGLE_CHAN** or **GAGE_DUAL_CHAN**. Use the value that controlled the most recent data capture. **tbs** is the "Time Between Samples" in nanoseconds for the

captured signal, for example, if sampling data at 10 MHz then this value would be 100.0, similarly sampled data at 1 KHz would require this value to be set to 1000000.0. ***trig** will be the adjusted trigger address for the type of hardware being used. ***start** will be the first valid address of the group requested for the most recent data capture. ***end** will be the last valid address of the group requested for the most recent data capture. This function assumes that the multiple record option is available on the current board. If it is called with a CompuScope LITE, ***start** and ***trig** are set to zero, and ***end** is set to max_available_memory - 1.
Note: Multiple record is standard on some boards and a hardware option on others. It is not available on the CompuScope LITE.

Return Value

The return value is equal to the current group. The group should equal the return value. When it doesn't the maximum multiple record group is returned.

See also

gage_detect_multiple_record and gage_multiple_record. Sample Routines: do_multiple_record.

Examples

C:
```
group = gage_calculate_mr_addresses (0L, gage_driver_info.board_type,
            gage_driver_info.trigger_depth, gage_driver_info.max_available_memory, 0,
            GAGE_DUAL_CHAN, srtable[board.srindex].sr_calc, *trig, *start, *end);
```

Visual Basic:
```
group = gage_calculate_mr_addresses(0, gage_driver_info.board_type,
        gage_driver_info.trigger_depth, gage_driver_info.max_available_memory, 0,
        GAGE_DUAL_CHAN, srtable(board.srindex).sr_calc, trigaddr(0), startaddr(0),
        endaddr(0))
```

Quick Basic:
```
group& = gagecalculatemraddresses& (0, gagedriverinfo.boardtype, gagedriverinfo.triggerdepth,
            gagedriverinfo.maxavailablememory, 0, GAGEDUALCHAN,
        srtable(board.srindex).calc, triggeraddress(0), startingaddress(0), endingaddress(0))
```

Pascal:
```
group := gage_calculate_mr_addresses (0, gage_driver_info.board_type,
            gage_driver_info.trigger_depth, gage_driver_info.max_available_memory, 0,
            GAGE_DUAL_CHAN, srtable[board.srindex].calc, trigger_address[0],
            starting_address[0], ending_address[0]);
```

gage_capture_mode

Syntax

C:
#include <gage_drv.h>
int16 gage_capture_mode (int16 mode, int16 rate, int16 multiplier);

Visual Basic:
function gage_capture_mode (ByVal mode As Integer, ByVal rate1 As Integer,
ByVal multiplier As Integer): As Integer

Quick Basic:
function gagecapturemode% (ByVal mode As Integer, ByVal rate1 As Integer,
ByVal multiplier As Integer)

Pascal:
function gage_capture_mode (mode, rate, multiplier: int16): int16;

Remarks

gage_capture_mode sets the data capture mode **mode**, by setting the number of active channels with the constants **GAGE_SINGLE_CHAN** or **GAGE_DUAL_CHAN** and the sample rate with **rate** and **multiplier**. The following table lists the available sample rates for the CompuScope boards.

Sample Rate	CS6012	CS1012	CS250	CS225	CS220	CSLITE
1 Hz to 1 MHz	S, D	S, D	S, D	S, D	S, D	S, D
2 MHz	S, D	S, D	S, D	S	S, D	S, D
5 MHz	S, D	S, D	S, D	S, D	S, D	S, D
10 MHz	S, D	S, D	S, D	S	S, D	S, D
12.5 MHz	No	No	No	D *	No	No
20 MHz	S	S	No	No	S, D	S, D
25 MHz	No	No	S, D	S, D	No	No
30 MHz	S, D	No	No	No	No	No
40 MHz	No	No	No	No	S	S
50 MHz	No	No	S, D	S	No	No
60 MHz	S	No	No	No	No	No
100 MHz	No	No	S	No	No	No

D = Dual mode, S = Single mode.

* Note that 12.5 MHz is obtained by using GAGE_RATE_12500 and GAGE_KHZ.

CompuScope sample rate **rate** values.

```
GAGE_RATE_1
GAGE_RATE_2
GAGE_RATE_4
GAGE_RATE_5
GAGE_RATE_10
GAGE_RATE_20
GAGE_RATE_25        /*  Only used with GAGE_MHZ on the CompuScope 250.      */
GAGE_RATE_30        /*  Only used with GAGE_MHZ and CompuScope 6012.        */
GAGE_RATE_40        /*  Only used with GAGE_MHZ on CompuScope 220 and LITE.  */
GAGE_RATE_50
GAGE_RATE_60        /* Only used with GAGE_MHZ and CompuScope 6012.         */
GAGE_RATE_100
GAGE_RATE_120       /* Used only with GAGE_MHZ for future use.              */
GAGE_RATE_125       /* Used only with GAGE_MHZ and CompuScope 2125.         */
GAGE_RATE_150       /* Used only with GAGE_MHZ for future use.              */
GAGE_RATE_200
GAGE_RATE_250       /* Used only with GAGE_MHZ and CompuScope 2125.         */
GAGE_RATE_300       /* Used only with GAGE_MHZ for future use.              */
GAGE_RATE_500
GAGE_RATE_12500     /* Used only with GAGE_KHZ and CompuScope 225 for 12.5 MHz.*/
```

CompuScope sample rate **multiplier** values.

```
GAGE_HZ
GAGE_KHZ
GAGE_MHZ
GAGE_GHZ
GAGE_EXT_CLK
GAGE_SW_CLK
```

The **multiplier** value **GAGE_EXT_CLK** is used to set the CompuScope external clock mode. This mode may or may not be available as a software selection, as it is available only through a hardware jumper on some versions of the board, but this mode should be set so as to stop the internal clock and to maintain software compatibility with future versions of the hardware. The **GAGE_SW_CLK** constant for **multiplier** sets the hardware to respond to a clock pulse generated **gage_software_clock** under software control for special applications (if the hardware is so equipped).

Return Value

If the function is successful then a one is returned and the affected parameters are set. If the function fails then the routine will return a zero and no CompuScope settings are affected and **gage_get_error_code** may be called to obtain the error code.

See Also

gage_input_control and gage_trigger_control. Sample Routines: prepare_for_capture.

Examples

C:

```
gage_capture_mode (GAGE_DUAL_CHAN, GAGE_RATE_10, GAGE_KHZ);
```

Visual Basic:

```
dummy = gage_capture_mode (GAGE_DUAL_CHAN, GAGE_RATE_10, GAGE_KHZ)
```

Quick Basic:

```
dummy% = gagecapturemode% (GAGEDUALCHAN, GAGERATE10, GAGEKHZ)
```

Pascal:

```
gage_capture_mode (GAGE_DUAL_CHAN, GAGE_RATE_10, GAGE_KHZ);
```

gage_detect_multiple_record

Syntax:

C:
#include <gage_drv.h>
int16 gage_detect_multiple_record (void);

Visual Basic:
function gage_detect_multiple_record () As Integer

Quick Basic:
function gagedetectmultiplerecord% ()

Pascal:
function gage_detect_multiple_record : int16;

Remarks

gage_detect_multiple_record is used to determine if a CompuScope board has multiple record capability. As this feature is a special order item from the factory, it is not standard equipment. Multiple record is also currently not available on the CompuScope LITE.

Return Value

A TRUE (non-zero) value is returned if the CompuScope hardware has multiple record capability and a FALSE (zero) otherwise.

See also

gage_multiple_record. Sample Routines: init_driver_hardware.

Examples

C:
mr_available = gage_detect_multiple_record

Visual Basic:
mr_available = gage_detect_multiple_record

Quick Basic:
mravailable% = gagedetectmultiplerecord%

Pascal:
mr_available := gage_detect_multiple_record

gage_driver_initialize

Syntax

C:
#include <gage_drv.h>
int16 gage_driver_initialize (uInt16 far *records, uInt16 memory);

Visual Basic:
function gage_driver_initialize (records As Integer, ByVal memory As Integer) As Integer

Quick Basic:
function gagedriverinitialize% (SEG records As Integer, ByVal memory As Integer)

Pascal:
gage_b_l_array: array [0..GAGE_B_L_BUFFER_SIZE] of word;
function gage_driver_initialize (var records: gage_b_l_array; memory: uInt16): int16;

Remarks

The **gage_driver_initialize** routine will fully configure each board found in the system. From then on a call to **gage_select_board** will be required to access any board .

The **records** parameter is assumed to be an uninitialized word array. The array **gage_board_location** has been created for this purpose in the driver (also available externally) and is **GAGE_B_L_BUFFER_SIZE** words long. The format of the array is that the first **GAGE_B_L_STATUS_START** words are for the board segment and index values, each pair occupies **GAGE_B_L_ELEMENT_SIZE** words, for each of the possible **GAGE_B_L_MAX_CARDS** boards.

A status field is provided for each potential board location which is **GAGE_B_L_STATUS_SIZE** words in length. The values for the status field are constants that correspond to bit positions in the status field and must be masked to determine which errors occurred when initializing the board. The low nibble is for problems with the segment and index. **GAGE_BAD_LSB_SEGMENT** means that the low order byte of the segment was not equal to zero, **GAGE_BAD_MSB_SEGMENT** is used when the segment is either less then A000 hex or greater then DF00 hex (the valid area in the memory map reserved for slot resources is 0A0000 hex to 0DFFFF hex), **GAGE_BAD_LSB_INDEX** is set when the least significant bit of the index is not zero and **GAGE_BAD_MSB_INDEX** is used when the high order byte of the index is either equal to 00 hex or greater then 03 hex (the valid area in the I/O map reserved for slot resources is 0100 hex to 03ff hex).

If the status bits are zero after the above test then the board initialization continues. If the expected board does not properly respond, then **GAGE_DETECT_FAILED** bit is set and initialization stops. If the board is found then the board's memory is checked and sized. If any byte of memory does not match its preset value then the **GAGE_MEMORY_FAILED** bit is set and initialization of this board stops.

The most significant byte contains the board type found during initialization. This byte is set after the driver tries to detect the hardware. Therefore, this byte could not be set and the board was not initialized due to a memory error. There are a number of constants available that can be masked with the status word to determine the board type. These are **GAGE_ASSUME_CSLITE**, **GAGE_ASSUME_CS220, GAGE_ASSUME CS225, GAGE_ASSUME_CS250, GAGE_ASSUME_CSLITE15, GAGE_ASSUME_CS2125, GAGE_ASSUME_CS1012** and **GAGE_ASSUME_CS6012.**.

The **memory** parameter allows the memory self test to be disabled by supplying the size of the memory in kilobytes of the installed board(s). The constant **GAGE_MEMORY_SIZE_TEST** will force the memory test to be performed. Note that the memory size for all installed boards must be the same or a conflict can occur and the data returned may be invalid for the boards with the incorrect memory size assigned.

Several memory size constants **GAGE_MEMORY_SIZE_016K**, **GAGE_MEMORY_SIZE_032K**, **GAGE_MEMORY_SIZE_064K**, **GAGE_MEMORY_SIZE_128K**, **GAGE_MEMORY_SIZE_256K**, **GAGE_MEMORY_SIZE_512K, GAGE_MEMORY_SIZE_001M**,**GAGE_MEMORY_SIZE_002M**, **GAGE_MEMORY_SIZE_004M** and **GAGE_MEMORY_SIZE_008M** are available for specifying default memory size for any of the current CompuScope cards.

The constant **GAGE_MEMORY_SIZE_TEST** will force the memory test to be performed. If an incorrect memory size is found then the status field for the segment and index record in question will have the **GAGE_BAD_MEMORY_SIZE** bit set. If the status is zero and the corresponding segment and index record are non-zero then this particular board was properly initialized. If, however, the status is zero and the segment and index record are also zero then the "board" is the premature end of the **records** array. By default the first board found will be selected.

A local data structure, in the driver, is created for each board. To allow access to this structure the routine **gage_get_driver_info** has been implemented that queries the driver about information on the current selected board. It is advised that this method be maintained for compatibility with future releases of the CompuScope drivers and not using the internal driver variables directly.

The last area of the status word returns the type of CompuScope board found. These maskable bits can be detected by using the **GAGE_ASSUME_CS6012**, **GAGE_ASSUME_CS1012**, **GAGE_ASSUME_CS2125**, **GAGE_ASSUME_CS250**, **GAGE_ASSUME_CS225**, **GAGE_ASSUME_CS220**, **GAGE_ASSUME_CSLITE** and **GAGE_ASSUME_CSLITE15** constants. This is useful in case the driver could not detect the memory size but was able to detect the CompuScope board type.

Return Value

The value returned is the number of active CompuScope boards found. If one or more of the boards was expected but not found the number of found boards will be returned as a negative number with the absolute value equal to the number of boards actually initialized. A return of zero (0) indicated that either no boards are found or there is a problem(s) with the array that has been passed to this routine that prevents initialization.

See also

gage_read_config_file, gage_get_driver_info and gage_select_board.
Sample Routines: init_driver_hardware.

Examples

C:
```
gage_driver_initialize((uInt16 far *)gage_board_location, GAGE_MEMORY_SIZE_TEST);
```

Visual Basic:
```
boards =  gage_driver_initialize(gage_board_location(0), GAGE_MEMORY_SIZE_TEST)
```

Quick Basic:

```
boards% =  gagedriverinitialize% (SEG gageboardlocation(1), GAGE_MEMORY_SIZE_TEST)
```

Pascal:

```
gage_driver_initialize(gage_board_location, GAGE_MEMORY_SIZE_TEST);
```

gage_driver_remove

Syntax

C:
#include <gage_drv.h>
void gage_driver_remove (void);

Visual Basic:
Sub gage_driver_remove ()

Quick Basic:
Sub gagedriverremove ()

Pascal:
procedure gage_driver_remove;

Remarks

gage_driver_remove is used to remove from memory all data structures that were created by the drivers or the DLL. This routine should typically be used upon exit from the application program.

Return Value

None.

See also

gage_driver_initialize

Examples
C:
gage_driver_remove ();

Visual Basic:
Call gage_driver_remove ();

Quick Basic:
CALL gagedriverremove ();

Pascal:
gage_driver_remove;

gage_forced_trigger_capture

Syntax

C:
```
#include <gage_drv.h>
void gage_forced_trigger_capture (int16 reset_trigger_source);
```

Visual Basic:
```
Sub gage_forced_trigger_capture (ByVal reset_trigger_source As Integer)
```

Quick Basic:
```
Sub gageforcedtriggercapture (ByVal resettriggersource As Integer)
```

Pascal:
```
procedure gage_forced_trigger_capture (reset_trigger_source: int16);
```

Remarks

gage_forced_trigger_capture is used to force the capture of data by the CompuScope board(s), primarily in the event that a trigger event never occurs. This routine forces the board(s) to accept a trigger event not previously received. The **reset_trigger_source** parameter is used to re-initialize the trigger source after the hardware has been aborted. This is normally set to the initial trigger source used. The **gage_forced_trigger_capture** routine should be used instead of **gage_set_trigger_source** and **gage_software_trigger** routines since it works for multiple board installations and does not leave the driver in an unpredictable state.

Return Value

None.

See also

gage_abort_capture and gage_triggered. Sample Routines: acquire_check.

Examples

C:
```
gage_forced_trigger_capture (board.source);
```

Visual Basic:
```
Call gage_forced_trigger_capture (board.source)
```

Quick Basic:
```
CALL gageforcedtriggercapture (board.source)
```

Pascal:
```
gage_forced_trigger_capture(board.source);
```

gage_get_boards_found

Syntax

C:
```
#include <gage_drv.h>
int16 gage_get_boards_found (void);
```

Visual Basic:
```
Function gage_get_boards_found () As Integer
```

Quick Basic:
```
Function gagegetboardsfound% ()
```

Pascal:
```
function gage_get_boards_found: int16;
```

Remarks

gage_get_boards_found determines how many CompuScope boards were encountered when the last call to the **gage_driver_initialize** routine was performed.

Return Value

The number of boards currently installed by the driver.

See also

gage_driver_initialize. Sample Routines: prepare_for_capture, transfer_data.

Examples

C:
```
boards_found = gage_get_boards_found ();
```

Visual Basic:
```
 boards = gage_get_boards_found
```

Quick Basic:
```
 boards% = gagegetboardsfound%
```

Pascal:
```
boards_found := gage_get_boards_found;
```

gage_get_config_filename

Syntax

C:
#include <gage_drv.h>
int16 gage_get_config_filename (LPSTR cfgfn);

Visual Basic:
function gage_get_config_filename (ByVal cfgfn As String) As Integer

Quick Basic:
function gagegetconfigfilename% (ByVal segfilename As Integer, ByVal offfilename As Integer)

Pascal:
function gage_get_config_filename (cfgfn: pchar): int16;

Remarks

gage_get_config_filename determines the complete path to the configuration file that contains the board location data created by either the DOS based GSINST.EXE program or the Windows based GSWINST.EXE program. The configuration file created by them is called GAGESCOP.INC and must be in the Windows directory if you are using the Windows DLL, or in your current directory if you are using the DOS drivers. **cfgfn** is a string variable that must be long enough to hold the returned path. Note the different parameters used under Quick Basic. This is due to differences in how strings are handled and represented under C and Quick Basic. See the example program SIMPLE.BAS to
see how to manipulate the string to use it with this routine. Under protected mode Pascal, this routine is contained in the GAGE_DRV.PAS file.

Return Value

A true value is returned, if the routine successfully returned the configuration filename.

See also

gage_read_config_file. Sample Routines: init_driver_hardware.

Examples

C:
ret = gage_get_config_filename ((LPSTR)(board_loc_file));

Visual Basic:
i = gage_get_config_filename (board_loc_file)

Quick Basic:
result% = gagegetconfigfilename% (VARSEG(configname$), SADD(configname$))

Pascal:
i := gage_get_config_filename (board_loc_file);

gage_get_current_drv_structure

Syntax

C:

```
#include <gage_drv.h>
#include <gage_low.h>
void gage_get_current_drv_structure (gage_board_type far *gbm, int32 far *gbt_size,
                                      int32 far *gbt_routines);
```

Visual Basic:

This routine is currently not supported under Visual Basic.

Quick Basic:

This routine is currently not supported under Quick Basic.

Pascal:

This routine is currently not supported under Pascal.

Remarks

gage_get_current_drv_structure is used to get information and access to the internal gage_board_type structure used by the driver. The header file gage_low.h must be included by the calling program so this structure will be defined. The parameter **gbm** is the structure for the master board, which is the first board in a multiple board system. **gbt_size** is the size of this structure and **gbt_routines** is a pointer to the routines that are contained in the structure. Use of this structure and routines can in some cases be used to optimize the application program. It is recommended that program development be done without the use of this routine. After the program is running, if there is a need to increase performance then this structure can be used. This routine is meant mainly for Windows applications, as DOS applications written in C already have access to the gage_board_type structure by including the header file gage_low.h in their program.

Return Value

None.

See also

gage_select_current_board. Sample Routines: driver_support.

Examples

C:

```
gage_get_current_drv_structure ((gage_board_type far *)(&gbm); (int32 far *)(&gbt_size),
                                      (int32 far *)(&gbt_routines));
```

Note: The typecasts are unnecessary if your program is compiled in the large model.

gage_get_driver_info

Syntax

C:
#include <gage_drv.h>
void gage_get_driver_info ((gage_driver_info_type far *)(driver_info));

Visual Basic:
Sub gage_get_driver_info (driver_info As gage_driver_info_type)

Quick Basic:
Sub gagegetdriverinfo (SEG driverinfo As gagedriverinfotype)

Pascal:
procedure gage_get_driver_info (var driver_info: gage_driver_info_type);

Remarks

gage_get_driver_info fills a structure or record with the relevant information from the driver variables as to the current settings in the current CompuScope . The structure **driver_info** is a subset of the record used by the driver and includes only those values that have meaning to the control program. For further information about the structure, refer to the section titled CompuScope Driver: types, structures and definitions.

Return Value

None, but the structure or record is filled with the proper information from the driver's structure.

See Also

gage_get_driver_info_structure and gage_update_driver_info.
Sample Routines: init_driver_hardware, prepare_for_capture.

Examples

C:
gage_get_driver_info ((gage_driver_info_type far*)(&driver_info));

Visual Basic:
Call gage_get_driver_info (driver_info)

Quick Basic:
CALL gagegetdriverinfo (SEG driver_info)

Pascal:
gage_get_driver_info (driver_info);

gage_get_driver_info_structure

Syntax

C:

```
#include <gage_drv.h>
void gage_get_driver_info (uInt16 far *major_version, uInt16 far *minor_version, uInt16 far
                    *board_support,   gage_driver_info_type far * far * gdi, int32 far *gdi_size);
```

Visual Basic:

```
Sub gage_get_driver_info (major_version As Integer, minor_version As Integer, board_support As
                              Integer, gdi As gage_driver_info_type, gdi_size As Long)
```

Quick Basic:

```
Sub gagegetdriverinfo (SEG majorversion%, SEG minorversion%, SEG boardtype%, SEG gdi As
                    gagedriverinfotype, SEG gdisize&)
```

Pascal:

```
procedure gage_get_driver_info (major_version: puInt16; minor_version: puInt16; board_support:
                              puInt16; gdi: gdiPtrPtr; gdi_size: pint32);
```

Remarks

gage_get_driver_info_structure returns relevant information about the main structure used by the driver. The current driver version is returned in the **major_version** and **minor_version** parameters. **board_support** returns the CompuScope boards that are supported by the driver. This will be some combination of the "Orable" predefined constants **GAGE_ASSUME_CSLITE**, **GAGE_ASSUME_CSLITE15**, **GAGE_ASSUME_CS25016**, **GAGE_ASSUME_CS220**, **GAGE_ASSUME_CS225**, **GAGE_ASSUME_CS250**, **GAGE_ASSUME_CS2125**, **GAGE_ASSUME_CS1012** and **GAGE_ASSUME_CS6012**. The **gdi** parameter returns a pointer to the gage_driver_info_type structure. This structure is a subset of the record used by the driver and includes only those values that have meaning to the control program. For further information about the structure, refer to the section titled CompuScope Driver: types, structures and definitions. **gdi_size** is the current size of this structure in this version of the driver. This parameter can be used to allocate the proper amount of memory to hold the new version of the gage_driver_info_structure if it has changed. This way the new driver can still be used without having to recompile the application program. Note that any new fields will be unavailable to the application program until it is recompiled with the new header files. It is recommended that this routine be called before calling either **gage_get_driver_info** or **gage_update_driver_info** to maintain compatibility with future versions of the DLL or drivers.

Return Value

None, but the parameters are filled with the proper information about the driver's structure.

See Also

gage_get_driver_info and gage_update_driver_info. Sample Routines: driver_support.

Examples

C:

```
gage_get_driver_info_structure ((uInt16 far *)(&major_version), (uInt16 far *)(&minor_version),
(uInt16 far *)(&board_support),(gage_driver_info_type far * far *)(&gdi), (int32 far *)(&gdi_size));
```

Visual Basic:

```
Call gage_get_driver_info_structure(major_version, minor_version, board_support, gdi, gdi_size)
```

Quick Basic:

```
CALL gagegetdriverinfostructure (SEG majorversion, SEG minorversion, SEG boardsupport, SEG gdi,
                                SEG gdisize)
```

Pascal:

```
gage_get_driver_info_structure (@major_version, @minor_version, @board_support, @gditemp,
                    @gdi_size);
```

gage_get_error_code

Syntax

C:
```
#include <gage_drv.h>
int16 gage_get_error_code (void);
```

Visual Basic:
```
Function gage_get_error_code () As Integer
```

Quick Basic:
```
Function gagegeterrorcode% ()
```

Pascal:
```
function gage_get_error_code: int16;
```

Remarks

gage_get_error_code returns the error code associated with the last call to the CompuScope driver.

Return Value

The error that occurred and the board that caused the error. This function returns a value which is encoded with the high byte containing the board in error and the low byte is set equal to the defined error constant. These constants are listed below.

GAGE_NO_ERROR
GAGE_NO_SUCH_BOARD
GAGE_NO_SUCH_MODE
GAGE_NO_SUCH_INPUT
GAGE_INVALID_SAMPLE_RATE
GAGE_NO_SUCH_COUPLING
GAGE_NO_SUCH_CHANNEL
GAGE_NO_SUCH_GAIN
GAGE_NO_SUCH_TRIG_DEPTH
GAGE_NO_SUCH_TRIG_POINT
GAGE_NO_SUCH_TRIG_SLOPE
GAGE_NO_SUCH_TRIG_SOURCE
GAGE_MISC_ERROR

See also

Nothing.

Examples

C:
```
error = gage_get_error_code ();
```

Visual Basic:
```
ret = gage_get_error_code()
```

Quick Basic:
ret% = gagegeterror_code% ()

Pascal:
error_code := gage_get_error_code;

gage_get_interpolate_trigger

Syntax

C:
#include <gage_drv.h>
void gage_get_interpolate_trigger (int16 action, int16 chan, int32 far *trig, int32 start, int32 end);

Visual Basic:
Sub gage_get_interpolate_trigger (ByVal action As Integer, ByVal chan As Integer, trig As Long, ByVal start As Integer, ByVal endaddr As Long)

Quick Basic:
SUB gagegetinterpolatetrigger (ByVal action As Integer, ByVal chan As Integer, SEG trigaddr As Long, ByVal startaddr As Long, ByVal endaddr As Long)

Pascal:
procedure gage_get_interpolate_trigger (action, chan: int16; var trig: int32; start, endaddr: int32);

Remarks

gage_get_interpolate_trigger calculates an interpolated trigger address for the specified channel. If **action** is 1, the **channel** specified becomes the interpolated trigger address channel and the interpolated trigger address is calculated for that **channel**. The routine should then be called with **action** set to 0 so the trigger addresses can be calculated for the other channels based on the interpolated trigger address of the interpolated trigger channel. **trig** is the new interpolated trigger address. **start** and **end** are the starting and ending addresses, which should be obtained by calling **gage_calculate_addresses**.

Return Value

None

See also

gage_reset_interpolate_trigger. Sample Routines: init_memory_pointers, init_chan_memory_pointers.

Examples

C:
gage_get_interpolate_trigger (1, interpolate_trigger_channel & 1, &trig, start, end);

Visual Basic:
Call gage_get_interpolate_trigger (1, interpolate_trigger_channel AND 1, trig, start, endaddr)

Quick Basic:
CALL gagegetinterpolatetrigger (1, interpolatetriggerchannel AND 1, SEG trig, start, endaddr)

Pascal:
gage_get_interpolate_trigger (0, chan AND 1, trig, start, endaddr);

gage_get_records

Syntax

C:
#include <gage_drv.h>
int16 gage_get_records (uInt16 far *records, int16 record, uInt16 far *segment, uInt16 far *index, uInt16 far *typename, uInt16 far *status);

Visual Basic:
Function gage_get_records (records As Integer, ByVal record As Integer, segment As Integer, index As Integer, typename As Integer, status As Integer) As Integer

Quick Basic:
Function gagegetrecords% (SEG records As Integer, ByVal record As Integer, SEG segment As Integer, SEG index As Integer, SEG typename As Integer, SEG status As Integer)
Pascal:
function   gage_get_records(var records: gage_b_l_array; rec: int16; var segment: uInt16; var index: uInt16; var typename: uInt16; var status: uInt16): int16;
Note: the type gage_b_l_array is defined in GAGE_DRV.PAS

Remarks

gage_get_records can be used to read the **gage_board_location** array. The array **gage_board_location** has been created by the driver (also available externally) and can hold the initialization for **GAGE_B_L_MAX_CARDS** boards. The array is initialized by using a call to **gage_set_records**. The **record** parameter is the board number of the record to be read, beginning with 0. The **segment**, **index, typename** and **status** parameters are the values in the **gage_board_location** array for that particular board. The **segment** and **index** are the values that were set by the GSINST.EXE configuration program. The **typename** and **status** parameters return the board type and the initialization status of the board as one of the predefined constants (**GAGE_ASSUME_CSLITE,** etc.). The **status** will hold any errors that occurred during board initialization.

Return Value

The return value is one if the **record** parameter is in range (0 to **GAGE_B_L_MAX_CARDS** - 1) and zero otherwise. If the return value is one, the **segment**, **index**, **typename** and **status** parameters will contain valid values for the specified **record**.

See also

gage_set_records and gage_driver_initialize. Sample Routines: init_driver_hardware.

Examples

C:
gage_get_records ((uInt16 far *)gage_board_location), 0, (uInt16 far *)(&segment), (uInt16 far *)(&index), (uInt16 far *)(&typename), (uInt16 far *)(status));
Note: the typecasts may not be necessary, depending on the memory model used.
Visual Basic:

```
ret = gage_get_records (gage_board_location (0), 0, segment, index, typename, status)
```

Quick Basic:

```
ret% = gagegetrecords% (SEG gage_board_location(1), 0, SEG segment, SEG index, SEG typename,
                        SEG status)
```

Pascal:

```
ret := gage_get_records (gage_board_location,  0, segment, index, typename, status);
```

gage_get_trigger_view_offset

Syntax

C:
#include <gage_drv.h>
void get_trigger_view_offset (int32 far *offset)

Visual Basic:
Sub get_trigger_view_offset (offset As Long)

Quick Basic:
Sub get_trigger_view_offset (SEG offset As Long)

Pascal:
procedure get_trigger_view_offset (var offset: int32);

Remarks

gage_get_trigger_view_offset can be used to get the current trigger view offset when using the routine **gage_trigger_view_transfer**. The default value of the offset is 0, which means that capture will start at the trigger address. This value can be changed, to capture pre-trigger data for example, by using the routine **gage_set_trigger_view_offset**.

Return Value

None.

See also

gage_set_trigger_view_offset and gage_trigger_view_transfer.

Examples

C:
gage_get_trigger_view_offset ((int32 far *)offset);
Note: the typecast is unnecessary if the application program is large model

Visual Basic:
Call gage_get_trigger_view_offset (offset)

Quick Basic:
CALL gage_get_trigger_view_offset (SEG offset)

Pascal:
gage_get_trigger_view_offset (offset);

gage_initialize_start_capture

Syntax

C:
```
#include <gage_drv.h>
void gage_initialize_start_capture (int16 hardware_to_use)
```

Visual Basic:
```
Sub gage_initialize_start_capture (ByVal hardware_to_use As Integer)
```

Quick Basic:
```
Sub gageinitializestartcapture (ByVal hardwaretouse As Integer)
```

Pascal:
```
procedure gage_initialize_start_capture (hardware_to_use: int16);
```

Remarks

gage_initialize_start_capture is used to set up the **gage_start_capture** routine. The **gage_start_capture** routine is actually a pointer to the proper routine. The driver sets the pointer to use the start capture routine that controls the current boards. If, when you use GageScope you need to specify the OLDHW command line parameter when using old version of the CompuScope 220 and CompuScope LITE in master/slave operation then this routine must be called with the **hardware_to_use** parameter set to zero, otherwise this routine should not be called by the application program.

Return Value

None.

See also

gage_driver_initialize and gage_start_capture.

Examples

C:
```
 gage_initialize_start_capture (1);
```

Visual Basic:
```
 Call gage_initialize_start_capture (1)
```

Quick Basic:
```
 Call gageinitializestartcapture (1)
```

Pascal:
```
gage_initialize_start_capture (1);
```

gage_input_control

Syntax

C:
#include <gage_drv.h>
int16 gage_input_control (int16 channel, int16 enable, int16 coupling, int16 gain);

Visual Basic:
Function gage_input_control (ByVal channel As Integer, ByVal enable As Integer,
ByVal coupling As Integer, ByVal gain As Integer) As Integer

Quick Basic:
Function gageinputcontrol% (ByVal channel As Integer, ByVal enable As integer, ByVal coupling As Integer, ByVal gain As Integer)

Pascal:
function gage_input_control (channel, enable, coupling, gain: int16): int16;

Remarks

gage_input_control is used to set up the input channels of the CompuScope cards. The **channel** can be either channel A or B and the constants **GAGE_CHAN_A** and **GAGE_CHAN_B** should be used to indicate their respective channels. The **enable** parameter, using the constants **GAGE_INPUT_ENABLE** and **GAGE_INPUT_DISABLE**, either enables a channel to be captured or disables the data capture on that channel (on the CompuScope LITE it enables the internally routed test signal). Note, however, in the single channel mode both of the channels must be enabled for the board to capture data correctly. This will be done automatically if this routine is called after setting the capture mode with **gage_capture_mode**. The **coupling** can be either DC or AC as indicated by the constants **GAGE_DC** and **GAGE_AC** respectively.

There are six different **gain** values for each channel from divide by 5 to times ten. The constants **GAGE_PM_5_V**, **GAGE_PM_2_V, GAGE_PM_1_V**, **GAGE_PM_500_MV**, **GAGE_PM_200_MV** and **GAGE_PM_100_MV** cover the range of values. Note that these are dependent on the exact CompuScope being used :

Constant	Range	CS2125	CS6012	CS1012	CS250	CS225	CS220	CSLITE
GAGE_PM_5_V	+/ 5 Volts	Yes	Yes	Yes	Yes	Yes	Yes	No
GAGE_PM_2_V	+/ 2 Volts	Yes	Yes	Yes	No	No	Yes	No
GAGE_PM_1_V	+/ 1 Volt	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GAGE_PM_500_MV	+/ 500 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	No
GAGE_PM_200_MV	+/ 200 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GAGE_PM_100_MV	+/ 100 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	No

A seventh **gain** value, **GAGE_PM_10_V,** is reserved for future use.

Note that if you are operating in single channel mode, you should set the gain for channel A and channel B to be the same value.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and **gage_get_error_code** may be called to obtain the error code.

See also

gage_trigger_control and gage_capture_mode. Sample Routines: prepare_for_capture.

Examples

C:
```
gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, GAGE_DC, GAGE_PM_1_V);
```

Visual Basic:
```
dummy = gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, GAGE_DC,
                                        GAGE_PM_1_V)
```

Quick Basic:
```
dummy% = gageinputcontrol% (GAGECHANA, GAGEINPUTENABLE, GAGEDC, GAGEPM1V)
```

Pascal:
```
gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, GAGE_DC, GAGE_PM_1_V);
```

gage_make_error_code

Syntax

C:
#include <gage_drv.h>
void gage_make_error_code (uInt16 board, uInt16 error);

Visual Basic:
Sub gage_make_error_code (ByVal board As Integer, ByVal errno As Integer)

Quick Basic:
Sub gagemakeerrorcode (ByVal board As Integer, ByVal errno As Integer)

Pascal:
procedure gage_make_error_code (board, error: uInt16);

Remarks

gage_make_error_code is used by the driver to create the error codes returned by the **gage_get_error_code** routine.

Return Value

None.

See also

gage_get_error_code.

Examples

C:
gage_make_error_code (board, GAGE_NO_SUCH_BOARD);

Visual Basic:
Call gage_make_error_code (board, GAGE_NO_SUCH_BOARD)

Quick Basic:
Call gagemakeerrorcode (board, GAGE_NO_SUCH_BOARD)

Pascal:
gage_make_error_code (board, GAGE_NO_SUCH_BOARD);

gage_mem_read_chan_a

Syntax

C:
```
#include <gage_drv.h>
int16 gage_mem_read_chan_a (int32 location);
```

Visual Basic:
```
Function gage_mem_read_chan_a (ByVal location As Long) As Integer
```

Quick Basic:
```
Function gagememreadchana% (ByVal location As Long)
```

Pascal:
```
function gage_mem_read_chan_a (location: int32): int16;
```

Remarks

gage_mem_read_chan_a reads one byte (one signed integer for the CS6012 and CS1012) out of CompuScope memory for channel A in dual channel mode, and returns it as an integer. The **location** required starts at the trigger address and can precede and/or follow the trigger address as set by the current trigger depth and as indicated by **gage_ram_full**. The **gage_calculate_addresses** routine is very useful for determining the valid addresses and should be used instead of directly invoking the **gage_trigger_address** and **gage_ram_full** routines for future compatibility with the GageScope drivers.

Return Value

The byte (or integer) corresponding to the specified location for channel A returned as an unsigned integer (0 - 255) for eight bit boards and as a signed 12-bit integer (-2048 - 2047) for the CS6012 and CS1012.

See also

gage_trigger_address, gage_mem_read_chan_b, gage_mem_read_dual, gage_mem_read_single and gage_ram_full. Sample Routines: transfer_data.

Examples

C:
```
data = gage_mem_read_chan_a (address);
```

Visual Basic:
```
data = gage_mem_read_chan_a (address)
```

Quick Basic:
```
data %= gagememreadchana% (address)
```

Pascal:
```
data := gage_mem_read_chan_a (address);
```

gage_mem_read_chan_b

Syntax

C:
#include <gage_drv.h>
int16 gage_mem_read_chan_b (int32 location);

Visual Basic:
Function gage_mem_read_chan_b (ByVal location As Long) As Integer

Quick Basic:
Function gagememreadchanb% (ByVal location As Long)

Pascal:
function gage_mem_read_chan_b (location: int32): int16;

Remarks

gage_mem_read_chan_b reads one byte (one signed integer for the CS6012 and CS1012) out of CompuScope memory for channel B in dual channel mode, and returns it as an integer. The **location** required starts at the trigger address and can precede and/or follow the trigger address as set by the current trigger depth and as indicated by **gage_ram_full**. The **gage_calculate_addresses** routine is very useful for determining the valid addresses and should be used instead of directly invoking the **gage_trigger_address** and **gage_ram_full** routines for future compatibility with the GageScope drivers.

Return Value

The byte (or integer) corresponding to the specified location for channel B returned as an unsigned integer (0 - 255) for eight bit boards and as a signed 12-bit integer (-2048 - 2047) for the CS6012 and CS1012.

See also

gage_trigger_address, gage_mem_read_chan_a, gage_mem_read_dual, gage_mem_read_single and gage_ram_full. Sample Routines: transfer_data.

Examples

C:
data = gage_mem_read_chan_b (address);

Visual Basic:
data = gage_mem_read_chan_b (address)

Quick Basic:
data% = gagememreadchanb% (address)

Pascal:
data := gage_mem_read_chan_b (address);

gage_mem_read_dual

Syntax

C:
```
#include <gage_drv.h>
int16 gage_mem_read_dual (int16 channel, int32 location);
```

Visual Basic:
```
Function gage_mem_read_dual (ByVal channel As Integer, ByVal location As Long) As Integer
```

Quick Basic:
```
Function gagememreaddual% (ByVal channel As Integer, ByVal location As Long)
```

Pascal:
```
function gage_mem_read_dual (channel: int16; location: int32): int16;
```

Remarks

gage_mem_read_dual reads one byte (one signed integer for the CS6012 and CS1012) out of CompuScope memory in dual channel mode and returns it as an integer. **channel** can be one of the constants **GAGE_CHAN_A** or **GAGE_CHAN_B**. The **location** required starts at the trigger address and can precede and/or follow the trigger address as set by the current trigger depth and as indicated by **gage_ram_full**. The **gage_calculate_addresses** routine is very useful for determining the valid addresses and should be used instead of directly invoking the **gage_trigger_address** and **gage_ram_full** routines for future compatibility with the GageScope drivers.

Return Value

The byte (or integer) corresponding to the specified location for channel A or channel B returned as an unsigned integer (0 - 255) for eight bit boards and as a signed 12-bit integer (-2048 - 2047) for the CS6012 and CS1012.

See also

gage_trigger_address, gage_mem_read_single and gage_ram_full.

Examples

C:
```
data = gage_mem_read_dual (GAGE_CHAN_A, address);
```

Visual Basic:
```
data = gage_mem_read_dual (GAGE_CHAN_A, address)
```

Quick Basic:
```
data% = gagememreaddual% (GAGE_CHAN_A, address)
```

Pascal:
```
data := gage_mem_read_dual (GAGE_CHAN_A, address);
```

gage_mem_read_single

Syntax

C:
#include <gage_drv.h>
int16 gage_mem_read_single (int32 location);

Visual Basic:
Function gage_mem_read_single (ByVal location As Long) As Integer

Quick Basic:
Function gagememreadsingle% (ByVal location As Long)

Pascal:
function gage_mem_read_single (location: int32): int16;

Remarks

gage_mem_read_single reads one byte (one signed integer for the CS6012 or CS1012) out of CompuScope memory in single channel mode and returns it as an integer. The **location** required starts at the trigger address and can precede and/or follow the trigger address as set by the current trigger depth and as indicated by **gage_ram_full**. The **gage_calculate_addresses** routine is very useful for determining the valid addresses and should be used instead of directly invoking the **gage_trigger_address** and **gage_ram_full** routines for future compatibility with the GageScope drivers.

Return Value

The byte (or integer) corresponding to the specified location for channel A returned as an unsigned integer (0 - 255) for eight bit boards and as a signed 12-bit integer (-2048 - 2047) for the CS6012 or CS1012.

See also

gage_mem_read_dual, gage_trigger_address and gage_ram_full. Sample Routines: transfer_data.

Examples

C:
data = gage_mem_read_single (address);

Visual Basic:
data = gage_mem_read_single (address)

Quick Basic:
data% = gagememreadsingle% (address)

Pascal:
data := gage_mem_read_single (address);

gage_multiple_record

Syntax

C:
#include <gage_drv.h>
void gage_multiple_record (uInt16 mode);

Visual Basic:
Sub gage_multiple_record (ByVal mode As Integer)

Quick Basic:
Sub gagemultiplerecord (ByVal mode As Integer)

Pascal:
procedure gage_multiple_record (mode: uInt16);

Remarks:

gage_multiple_record allows the application program to force the CompuScope hardware to capture only the depth specified when the trigger event occurs. The card then re-arms itself to accept another trigger event and record that data immediately following the first capture. This process is repeated until the memory has been completely overwritten. During this operation the busy signal does not come down until the entire buffer has been written. The trigger signal can be monitored by using the **gage_triggered** routine to determine the current status of the CompuScope hardware. The board can not be busy when accessing the RAM. The busy status must be checked with **gage_busy** to wait until all the data captures have been completed. The mode parameter must be set to one to enable this mode, and cleared when this mode is not required (the default setting by the driver). This feature is a special order item from the factory for some CompuScopes, and is standard equipment on others. It is not available on the CompuScope LITE.

VERY IMPORTANT: When the multiple record feature is to be used, the **gage_capture_mode**, **gage_input_control** and **gage_trigger_control** routines must be called (to set internal variables) after calling **gage_multiple_record** regardless of whether the other parameters need to be changed.

Return Value

None

See also

gage_start_capture, gage_busy and gage_triggered. Sample Routines: prepare_for_capture.

Examples

C:
gage_multiple_record (1);

Visual Basic:
Call gage_multiple_record (1)
Quick Basic:
Call gagemultiplerecord (1)

Pascal:

```
gage_multiple_record (1);
```

gage_need_ram

Syntax

C:
```
#include <gage_drv.h>
void gage_need_ram (int16 need);
```

Visual Basic:
```
Sub gage_need_ram (ByVal need As Integer)
```

Quick Basic:
```
Sub gageneedram (ByVal need As Integer)
```

Pascal:
```
procedure gage_need_ram (need: int16);
```

Remarks

gage_need_ram allows the application program to access the CompuScope hardware's RAM buffers to examine the data that has been captured by the board. The board can not be busy when accessing the RAM. The busy status can checked with **gage_busy**.

Return Value

None.

See also

gage_mem_read_single, gage_mem_read_dual and gage_busy. Sample Routines: transfer_data.

Examples

C:
```
gage_need_ram (TRUE);
```

Visual Basic:
```
Call gage_need_ram (TRUE)
```

Quick Basic:
```
CALL gageneedram (TRUE)
```

Pascal:
```
gage_need_ram (True);
```

gage_normalize_address

Syntax

C:
#include <gage_drv.h>
int32 gage_normalize_address (int32 start, int32 address, int32 ram_size);

Visual Basic:
Function gage_normalize_address (ByVal start As Long, ByVal address As Long,
ByVal ram_size As Long) As Long

Quick Basic:
Function gagenormalizeaddress& (ByVal start As Long, ByVal address As Long,
ByVal ramsize As Long)

Pascal:
function gage_normalize_address (start, address, ram_size: int32): int32;

Remarks

gage_normalize_address calculates the logical address for the CompuScope hardware to allow the application programmer to compare different addresses obtained from the **gage_calculate_addresses** function. This routine manipulates the addresses obtained from **gage_calculate_addresses** to allow the application program to compare their values without concern for the memory rollover that occurs with the circular buffer architecture used by all versions of the CompuScope hardware. **start** is usually the starting address returned from **gage_calculate_addresses**. **address** is any address that is of interest in the CompuScope memory. **ram_size** is the size of the CompuScope memory on the card in question. This size will need to be determined with a call to the **gage_get_driver_info** routine.

Return Value

The difference between the **address** and the **start** including any memory buffer roll over that may have occurred as determined by **ram_size** (ie. the logical address).

See also

gage_calculate_addresses and gage_get_driver_info. Sample Routines: do_multiple_record.

Examples
C:
address = gage_normalize_address (trigger_address, end_address, max_available_memory);

Visual Basic:
address = gage_normalize_address (trigger_address, end_address , max_available_memory)

Quick Basic:
address& = gagenormalizeaddress& (triggeraddress, endaddress , maxavailablememory)

Pascal:
address := gage_normalize_address (trigger_address, end_address, max_available_memory);

gage_read_config_file

Syntax

C:
```
#include <gage_drv.h>
int16 gage_read_config_file (char far *filename, uInt16 far *records);
```

Visual Basic:
```
Function gage_read_config_file (ByVal filename As String, records As Integer) As Integer
```

Quick Basic:
```
Function gagereadconfigfile% (ByVal segfilename As Integer, ByVal offfilename As Integer, SEG records
                                    As Integer)
```

Pascal:
```
gage_b_l_array = array [0..GAGE_B_L_BUFFER_SIZE] of uInt16;
function gage_read_config_file (filename: pchar; var records: gage_b_l_array): int16;
```

Remarks

gage_read_config_file reads a file and stores the data in an array of words. The parameter **filename** is a text string that tells the routine the name of the file that contains the board indexes and starting segment values for each of the installed CompuScope boards. In Quick Basic, the driver is passed the offset and segment of the string.

This file is created with the utility **GSINST.EXE** or **GSWINST.EXE** which creates or modifies the file **GAGESCOPE.INC**. This file can be renamed and the desired name can be passed to this routine.

The **records** parameter is assumed to be an uninitialized word array. The array is then initialized with the values found in the file. The array **gage_board_location** has been created for this purpose in the DLL (also available externally) and is **GAGE_B_L_BUFFER_SIZE** words long. The format of the array is that the first **GAGE_B_L_STATUS_START** words are for the board segment and index values, each pair occupies **GAGE_B_L_ELEMENT_SIZE** words, for each of the possible **GAGE_B_L_MAX_CARDS** boards. A status field is provided for each potential board location which is **GAGE_B_L_STATUS_SIZE** words in length.

The values for the status field are discussed under **gage_driver_initialize**. This mechanism for initializing the boards removes the requirement of needing a disk file to run the application, since this array can be loaded (initialized) by the application program using another method. The **gage_set_records** routine was created for just this purpose.

Return Value

the return value represents the number of records initialized if the return value is greater than 0. if the number is less than zero then the number corresponds to the error encountered. The errors are: -1, file does not exist; -2, file cannot be opened; -3, file size cannot be determined; -4, file size modulo four is not zero; -5, file size indicates that more boards than the driver supports are present; -6, file cannot be read successfully; -7, file cannot be closed; 0, reserved for future use.

See also

gage_driver_initialize and gage_get_config_filename. Sample Routines: init_driver_hardware.

Examples

C:
```
expected = gage_read_config_file ((char far*)("GAGESCOP.INC"), (uInt16 far*)(&gage_board_location));
```

Visual Basic:
```
expected = gage_read_config_file (board_loc_file, gage_board_location (0))
```

Quick Basic:
```
expected% = gagereadconfigfile% (VARSEG(boardlocation$), SADD(boardlocation$),
                                        SEG boardlocation(1))
```

Pascal:
```
expected := gage_read_config_file (board_loc_file, gage_board_location);
```

gage_read_master_status

Syntax

C:
```
#include <gage_drv.h>
int16 gage_read_master_status (void);
```

Visual Basic:
```
Function gage_read_master_status () As Integer
```

Quick Basic:
```
Function gagereadmasterstatus% ()
```

Pascal:
```
function gage_read_master_status: int16;
```

Remarks

gage_read_master_status reads the status bits from the master card, regardless of how many CompuScope cards are in the system.

Return Value

The integer returned equals the status bits from the master board status register. This value will be an integer from 0 to 7, with bit 0 representing the busy bit, bit 1 representing the ram full bit and bit 2 representing the trigger bit. These bit patterns may or may not match the hardware status register pattern.

See also

gage_busy, gage_ram_full, gage_triggered and Appendix describing the register functionality for your particular CompuScope board.

Examples

C:
```
status = gage_read_master_status ();
```

Visual Basic:
```
status = gage_read_master_status
```

Quick Basic:
```
status% = gagereadmasterstatus%
```

Pascal:
```
status := gage_read_master_status;
```

gage_reset_interpolate_trigger

Syntax

C:
```
#include <gage_drv.h>
void gage_reset_interpolate_trigger (int16 board, int16 channel, int16 auto_mode, int16 level,
                              int16 slope);
```

Visual Basic:
```
Sub gage_reset_interpolate_trigger (ByVal board As Integer, ByVal channel As Integer, ByVal
                              auto_mode As Integer, ByVal level As Integer, ByVal slope As Integer)
```

Quick Basic:
```
Sub gageresetinterpolatetrigger (ByVal board As Integer, ByVal channel As Integer, ByVal automode As
                              Integer, ByVal level As Integer, ByVal slope As Integer)
```

Pascal:
```
procedure gage_reset_interpolate_trigger (board, channel, auto_mode, level, slope: int16);
```

Remarks

gage_reset_interpolate_trigger sets or resets the interpolated trigger values. This routine should be called once before calling **gage_get_interpolate_trigger** and again after any of the input parameters have been changed by calling **gage_input_control** and / or **gage_trigger_control.** The **board** and **channel** parameters are the board and channel for which the interpolated trigger is requested. **auto_mode** should be set to 1 if the trigger source is the same as the **channel** parameter, 0 otherwise. **level** and **slope** are the current trigger level and trigger slope respectively.

Return Value
None

See also

gage_get_interpolate_trigger. Sample Routines: init_interpolated_trigger.

Examples

C:
```
gage_reset_interpolate_trigger (board, (interpolate_trigger_channel & 1) + GAGE_CHAN_A, 1,
                              trigger_level, trigger_slope);
```

Visual Basic:
```
Call gage_reset_interpolate_trigger(board, (interpolate_trigger_channel And 1) + GAGE_CHAN_A, 1,
                                      trigger_level, trigger_slope)
```

Quick Basic:
```
Call gageresetinterpolatetrigger(board, (interpolatetriggerchannel And 1) + GAGECHANA, 1,
                                      triggerlevel, triggerslope)
```

Pascal:
```
gage_reset_interpolate_trigger (board, (interpolate_trigger_channel AND 1) + GAGE_CHAN_A,
                                     1, trigger_level, trigger_slope);
```

gage_select_board

Syntax

C:
#include <gage_drv.h>
int16 gage_select_board (int16 board);

Visual Basic:
Function gage_select_board (ByVal board As Integer) As Integer

Quick Basic:
Function gageselectboard% (ByVal boards As Integer)

Pascal:
function gage_select_board (board: int16): int16;

Remarks

gage_select_board sets the driver so that the current board is the board identified by the number passed to the routine, if possible.

Return Value

The integer returned equals the value passed to the function as **board**. If an error occurs or the value passed to the function exceeds the number of boards installed in the system then the return value does not equal **board** and **gage_get_error_code** may be called to obtain the error code.

See also

gage_driver_initialize. Sample Routines: init_driver_hardware, prepare_for_capture.

Examples

C:
current_board = gage_select_board (i);

Visual Basic:
current_board = gage_select_board (i)

Quick Basic:
currentboard% = gageselectboard% (i)

Pascal:
current_board := gage_select_board (i);

gage_select_current_board

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_select_current_board (int16 board, gage_board_type far *gcc);
```

Visual Basic:
This routine is currently not suitable for Visual Basic.

Quick Basic:
This routine is currently not suitable for Quick Basic.

Pascal:
This routine is currently not suitable for Pascal.

Remarks

gage_select_current_card is used to gain access to the internal driver structure, gage_board_type, for the specified CompuScope card. The header file gage_low.h must be included to define the gage_board_type structure. **board** is the specified CompuScope card. **gcc** is a pointer to the internal structure for this board. In some cases, an application program can be optimized by using this structure directly. It is recommended that program development be done without using this routine. Once the program is running, then this function can be used to optimize the program. **gage_select_current_card** is meant mainly for Windows applications, as DOS C applications already have access to the gage_board_type structure by including gage_low.h in the program. Please see the header file, gage_low.h, for more information about the gage_board_type structure.

Return value

A one is returned if the correct board was found, otherwise a zero is returned.

See also

gage_get_current_drv_structure.

Examples

C:
```
gage_select_current_board (1, (gage_board_type far *)(&gage_current_card));
```

gage_set_ext_clock_variables

Syntax

C:
#include "gage_drv.h"
void gage_set_ext_clock_variables (uInt16 external_clock_delay, float external_clock_rate);

Visual Basic:
Sub gage_set_ext_clock_variables (ByVal external_clock_delay As Integer, ByVal
external_clock_rate As Single)

Quick Basic:
Sub gagesetextclockvariables (ByVal externalclockdelay As Integer, ByVal
externalclockrate As Integer)

Pascal:
procedure gage_set_ext_clock_variables (external_clock_rate: uInt16, external_clock_rate: single);

Remarks

gage_set_ext_clock_variables is used to set the needed variables when using an external clock. **external_clock_rate** is the requested clock rate. This value will be the sample rate in single channel mode and twice the sample rate in dual channel mode. The **external_clock_delay** parameter sets the clock delay to synchronize the external clock to the CompuScope card. Using a value of -1 will let the drivers calculate this value. Use another value to override this option. The values calculated by the driver will be 0 if the sample rate is greater than 5 MHz or the maximum of 1 and 10,000 / external_clock_rate otherwise.

Return value

None

See also

Nothing.

Examples

C:
gage_set_ext_clock_variables (-1, 10000000.0);

Visual Basic:
Call gage_set_ext_clock_variables (-1, 10000000.0)

Visual Basic:
CALL gagesetextclockvariables (-1, 10000000.0)

Pascal:
gage_set_ext_clock_variables (-1, 10000000.0);

gage_set_records

Syntax

C:
```
#include <gage_drv.h>
int16 gage_set_records (uInt16 far *records, int16 record, uInt16 segment, uInt16 index,
        uInt16 typename, uInt16 status);
```

Visual Basic:
```
Function gage_set_records (records As Integer, ByVal record As Integer, ByVal segment As Integer,
        ByVal index As Integer, ByVal typename As Integer, ByVal status As Integer) As Integer
```

Quick Basic:
```
Function gagesetrecords% (SEG records As Integer, ByVal record As Integer, ByVal segment As Integer,
        ByVal index As Integer, ByVal typename As Integer, ByVal status As Integer)
```

Pascal:
```
gage_b_l_array: array [0..GAGE_B_L_BUFFER_SIZE] of word;
function gage_set_records (var records: gage_b_l_array; rec: int16; segment, index, typename,
        status: uInt16): int16;
```

Remarks

gage_set_records is used to initialize the **gage_board_location** array when either the configuration file is bad or not used. Using this routine is preferred because the format of the **gage_board_location** file is subject to change. The array **gage_board_location** has been created by the driver (also available externally) and can hold the initialization for **GAGE_B_L_MAX_CARDS** boards. The **record** parameter is the number of the board to be initialized, the **segment**, **index, typename** and **status** parameters are the desired values for that particular board. The segment and index are the values that are used by the GSINST.EXE configuration program. The **typename** and **status** should be zero in a call to **gage_set_records**. Upon return, the appropriate locations of the **gage_board_location** array will be filled with the segment, index, type of board (**GAGE_ASSUME_CSLITE**, etc.) and initialization errors, if any.

Return Value

The return value is one if the **record** parameter is in range (0 to GAGE_B_L_MAX_CARDS - 1) and zero otherwise.

See also

gage_driver_initialize and gage_get_records. Sample Routines: init_driver_hardware.

Examples

C:
```
gage_set_records ((uInt16 far *)gage_board_location, 0, seg, ind, , 0, 0);
```

Visual Basic:
```
dummy = gage_set_records (gage_board_location (0), 0, seg, ind, , 0, 0)
```

Quick Basic:
```
dummy% = gage_set_records% (gageboardlocation (1), 0, seg, ind, , 0, 0)
```

Pascal:

```
gage_set_records (gage_board_location, 0, seg, ind, 0, 0);
```

gage_set_trigger_view_offset

Syntax

C:
#include <gage_drv.h>
void gage_set_trigger_view_offset (int32 offset);

Visual Basic:
Sub gage_set_trigger_view_offset (ByVal offset As Long)

Quick Basic:
Sub gagesettriggerviewoffset (ByVal offset As Long)

Pascal:
procedure gage_set_trigger_view_offset: int32;

Remarks

gage_set_trigger_view_offset allows the application program to change where data capture begins when using **gage_trigger_view_transfer** by offsetting the trigger address. A negative offset can be used to capture pre-trigger data. Care should be taken when using a negative number so that you don't get data from before the start address. The valid start, trigger and ending addresses can be obtained from a call to **gage_calculate_addresses**.

Return Value

None.

See also

gage_get_trigger_view_offset, gage_trigger_view_transfer and gage_calculate_addresses.
Sample Routines: do_multiple_record.

Examples

C:
gage_set_trigger_view_offset (offset);

Visual Basic:
Call gage_set_trigger_view_offset (offset)

Quick Basic:
CALL gagesettriggerviewoffset (offset)

Pascal:
gage_set_trigger_view_offset (offset);

gage_software_clock

Syntax

C:
#include <gage_drv.h>
void gage_software_clock (void);

Visual Basic:
Sub gage_software_clock ()

Quick Basic:
Sub gagesoftwareclock ()

Pascal:
procedure gage_software_clock;

Remarks

gage_software_clock will cause the hardware to sample data when the mode clock rate has been set to **GAGE_SW_CLK**. Note that this feature is not available on all versions of the CompuScope boards as a standard feature. Please contact the factory if this option was not explicitly ordered.

Return Value

None.

See also

gage_capture_mode.

Examples

C:
gage_software_clock ();

Visual Basic:
Call gage_software_clock ()

Quick Basic:
CALL gagesoftwareclock ()

Pascal:
gage_software_clock;

gage_software_trigger

Syntax

C:
#include <gage_drv.h>
void gage_software_trigger (void);

Visual Basic:
Sub gage_software_trigger ()

Quick Basic:
Sub gagesoftwaretrigger ()

Pascal:
procedure gage_software_trigger;

Remarks

gage_software_trigger will trigger the hardware when the trigger source has been set to **GAGE_SOFTWARE**. The gage_start_capture routine can perform this task automatically, if so desired, if the parameter passed to that routine is non-zero.

Return Value

None.

See also

gage_trigger_control and gage_start_capture.

Examples

C:
gage_software_trigger ();

Visual Basic:
Call gage_software_trigger ()

Quick Basic:
CALL gagesoftwaretrigger ()

Pascal:
gage_software_trigger;

gage_start_capture

Syntax

C:
```
#include <gage_drv.h>
void gage_start_capture (int16 auto_trigger);
```

Visual Basic:
```
Sub gage_start_capture (ByVal auto_trigger As Integer)
```

Quick Basic:
```
Sub gagestartcapture (ByVal autotrigger As Integer)
```

Pascal:
```
procedure gage_start_capture (auto_trigger: int16);
```

Remarks

gage_start_capture is used to prepare the CompuScope hardware for data acquisition. The hardware must be previously configured for the operation mode, input ranges and trigger conditions with calls to **gage_capture_mode**, **gage_input_control** and **gage_trigger_control**. This routine is actually a pointer to the routine that performs this task as outlined in the description for the **gage_initialize_start_capture** routine. This routine should be used instead of the **gage_get_data** routine to maintain compatibility with future versions of CompuScope hardware, master/slave operation and the CompuScope drivers. The auto_trigger parameter, when used with a non-zero value, allows this routine to immediately trigger the CompuScope hardware when starting the data capture. The trigger source must have been previously set to **GAGE_SOFTWARE** for this parameter to have a predictable effect with all versions of CompuScope boards.

Return Value

None.

See also

gage_initialize_start_capture. Sample Routines: acquire_check.

Examples

C:
```
gage_start_capture (board.source == GAGE_SOFTWARE);
```

Visual Basic:
```
 Call gage_start_capture (auto_trigger)
```

Quick Basic:
```
 CALL gagestartcapture (auto_trigger)
```

Pascal:
```
gage_start_capture (auto_trigger);
```

gage_trigger_control

Syntax

C:
#include <gage_drv.h>
int16 gage_trigger_control (int16 source, int16 ext_coupling, int16 ext_gain, int16 slope, int16 level,
    int32 depth);

Visual Basic:
Function gage_trigger_control (ByVal source As Integer, ByVal ext_coupling As Integer, ByVal ext_gain
    As Integer, ByVal slope As Integer, ByVal level As Integer, ByVal depth As Long) As Integer

Quick Basic:
Function gagetriggercontrol% (ByVal source As Integer, ByVal extcoupling As Integer, ByVal extgain As
    Integer, ByVal slope As Integer, ByVal level As Integer, ByVal depth As Long)

Pascal:
function gage_trigger_control (source, ext_coupling, ext_gain, slope: int16; level: int16;
    depth: int32): int16;

Remarks

gage_trigger_control is used to set up the trigger parameters of the CompuScope . The **source** can be either channel A, B, EXTERNAL or SOFTWARE and the constants **GAGE_CHAN_A**, **GAGE_CHAN_B**, **GAGE_EXTERNAL** and **GAGE_SOFTWARE** should be used to indicate their respective sources.

The **ext_coupling** can be either DC or AC (except for the CompuScope LITE, which only has DC coupled external triggering) for the external trigger input as indicated by the constants **GAGE_DC** and **GAGE_AC** respectively (please note that the CompuScope LITE external trigger input is DC coupled only).

There are two different **ext_gain** values for the external trigger input. These constants are **GAGE_PM_5_V** and **GAGE_PM_1_V** (please note that the CompuScope LITE external trigger input has only the + / - 1 Volt range).

The trigger **slope** can be either positive or negative, using constants **GAGE_POSITIVE** and **GAGE_NEGATIVE**, and affects all sources except the software trigger.

The **level** can be any value between 0 and 255. The values are scaled internally to match the trigger input range selected. The minimum input level is 0 while the maximum trigger level is 255 (these correspond to -1 volt and +1 volt respectively for the times one input gain).

The **depth** parameter sets the trigger depth, that is the number of samples captured after the trigger event. The minimum value is 0. The resolution of this parameter is :

64 bytes for	CompuScope 1012, 6012	(e.g. 256, 320, 384, ... , 8128, 8192, ..)
64 bytes for	CompuScope 250, 225	(e.g. 256, 320, 384, ... , 8128, 8192, ...)
power of 2 for	CompuScope 220	(e.g. 256, 512, 1024, ... , 131072, 262144, ...)

16 bytes for CompuScope LITE (e.g. 256, 272, 288, 304, 320, ... , 8176, 8192 ...)

This can help reduce the capturing overhead when only a few points of data are required. Several defined constants are available to the application programmer to illustrate the use of this parameter. The constants **GAGE_POST_0K**, **GAGE_POST_1K**, **GAGE_POST_2K**, **GAGE_POST_4K**, **GAGE_POST_8K**, **GAGE_POST_16K**, **GAGE_POST_32K**, **GAGE_POST_64K** and **GAGE_POST_128K** , **GAGE_POST_256K**, **GAGE_POST_512K**, **GAGE_POST_1M**, **GAGE_POST_2M**, **GAGE_POST_4M** and **GAGE_POST_8M** are for the trigger depths of 0, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304 and 8388608 bytes respectively.

Note: Prior to version 2.40 of the drivers, the trigger level was represented as a byte.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and **gage_get_error_code** may be called to obtain the error code.

See also

gage_capture_mode and gage_input_control. Sample Routines: prepare_for_capture.

Examples

C:
```
gage_trigger_control (GAGE_CHAN_A, GAGE_DC, GAGE_PM_1_V, GAGE_POSITIVE, 160,
                        GAGE_POST_16K);
```

Visual Basic:
```
dummy = gage_trigger_control (GAGE_CHAN_A, GAGE_DC, GAGE_PM_1_V, GAGE_POSITIVE,
                        160, GAGE_POST_16K)
```

Quick Basic:
```
dummy% = gagetriggercontrol% (GAGECHANA, GAGEDC, GAGEPM1V, GAGEPOSITIVE,
                        160, GAGEPOST16K)
```

Pascal:
```
gage_trigger_control (GAGE_CHAN_A, GAGE_DC, GAGE_PM_1_V, GAGE_POSITIVE, 160,
                        GAGE_POST_16K);
```

gage_trigger_view_transfer

Syntax

C:
```
#include <gage_drv.h>
void gage_trigger_view_transfer (int16 channel, uInt8 far *buffer, int16 nsamples);
```

Visual Basic:
```
Sub gage_trigger_view_transfer (ByVal channel As Integer, buffer(0) As Integer,
                                ByVal nsamples As Integer)
```

Quick Basic:
```
Sub gagetriggerviewtransfer (ByVal channel As Integer,  SEG buffer As Any,
                             ByVal nsamples As Integer)
```

Pascal:
```
procedure gage_trigger_view_transfer (channel: int16, var buffer: uInt8; nsamples: int16);
```

Remarks

gage_trigger_view_transfer is used to copy **nsample** points of the CompuScope memory space**,** from the channel specified in the parameter **channel**, to the supplied buffer array **buffer** starting with the trigger address of the most recent data capture. A maximum of 4K samples may be transferred at a time. Please note that for the CS250, CS225, CS220 and CSLITE, sample size is a byte so 4K samples equals 4K bytes. The CS6012 and CS1012 use integer samples therefore 4K samples equals 4K integers. The GageScope program makes use of this routine to improve the transfer rate with the trigger view display mode and the X - Y mode.

Note: This routine is not as useful in Visual Basic. Because there is no byte type in Basic, the data is returned as integers. This is no problem with the 12-bit CompuScope cards (CS6012 and CS1012). With the 8-bit cards, any speed improvements gained by using this routine may be lost extracting the data bytes out of the integers returned. When using Quick Basic, the buffer can be read using the DEF SEG and PEEK functions, which will return a byte represented as an integer.

Return Value

None.

See also

gage_32k_to_buffer. Sample Routines: transfer_data.

Examples

C:
```
gage_trigger_view_transfer (GAGE_CHAN_A, (uInt8 far *)(buffer), 512)
```

Visual Basic:
```
Call gage_trigger_view_transfer (GAGE_CHAN_A, buffer(0), 512)
```

Quick Basic:

```
CALL gagetriggerviewtransfer (GAGECHANA, SEG buffer(0), 512)
```

Pascal:

```
gage_trigger_view_transfer (GAGE_CHAN_A,  (byte) buffer, 512)
```

Note: Because Visual Basic has no variables of type byte, the buffer must be an integer array. This is no problem when dealing with the CS6012 or CS1012, which have a sample size of 2 bytes (12 bits sign extended to 16 bits). With the CS250, CS225, CS220 and CSLITE, which have a sample size of 1 byte, the high and low bytes of the integer must be extracted. This can be done as follows:

```
j = 0
For i = 0 To (Size Of Buffer \ 2) - 1
    temp = CLng (65535) And CLng (buffer(i))
    temp_buffer(j) = temp And 255   ' extract the high byte and store as an integer
    temp_buffer(j+1) = (temp \ 256) And 255   ' extract the low byte and store as an integer
    j = j + 2
Next
```

In Quick Basic, you can use the DEF SEG and PEEK functions to extract a byte.

gage_triggered

Syntax

C:
#include <gage_drv.h>
int16 gage_triggered (void);

Visual Basic:
Function gage_triggered () As Integer

Quick Basic:
Function gagetriggered% ()

Pascal:
function gage_triggered: int16;

Remarks

gage_triggered determines if the CompuScope hardware has encountered a trigger event.

Return Value

A non zero or true value is returned if the board has triggered, otherwise a false value is returned.

See also

gage_busy and gage_ram_full. Sample Routines: acquire_check.

Examples

C:
triggered = gage_triggered ();

Visual Basic:
triggered = gage_triggered

Quick Basic:
triggered% = gagetriggered%

Pascal:
triggered := gage_triggered;

gage_update_driver_info

Syntax

C:
```
#include <gage_drv.h>
void gage_update_driver_info (void);
```

Visual Basic:
```
Sub gage_update_driver_info ()
```

Quick Basic:
```
Sub gageupdatedriverinfo ()
```

Pascal:
```
procedure gage_update_driver_info;
```

Remarks

This routine updates the structure obtained from **gage_get_driver_info_structure** so it contains the values contained in the driver. It is recommended that application programs use **gage_get_driver_info_structure** and then either this routine or **gage_get_driver_info** to maintain compatibility with future versions of the drivers or DLL.

Return Value

Nothing.

See also

gage_get_driver_info_structure and gage_get_driver_info.

Examples

C:
```
gage_update_driver_info ();
```

Visual Basic:
```
Call gage_update_driver_info ()
```

Quick Basic:
```
CALL gageupdatedriverinfo ()
```

Pascal:
```
gage_update_driver_info;
```

Global Routines: Group Two.

This section of routines offer some optimizations that can be performed when using the routines in this group. Most of the optimizations that could be performed will have very little effect on the speed of the application program since most routines to control the hardware are used for initialization and control configuration and as such are typically called infrequently.

The only place that true optimization can be realized is during the retrieval of the data from the CompuScope hardware. The eight group one routines gage_32k_to_buffer, gage_trigger_view_transfer, gage_mem_read_dual, gage_mem_read_chan_a, gage_mem_read_chan_b, gage_mem_read_single gage_calculate_addresses and gage_trigger_address are used for this purpose. gage_trigger_address is a simple routine that is called only once per capture and therefore optimization, even in assembly language, will not yield much if any improvement. However, if the functionality of the read back functions gage_mem_read_dual, gage_mem_read_chan_a, gage_mem_read_chan_b and gage_mem_read_single are incorporated into your display routine, for example, then an increase of display speed can be realized.

The example used for the gage_set_block_number illustrates this point by loading a buffer, however, it ignores the starting address that would be determined with a call to gage_trigger_address. To incorporate the trigger address the routine would have to determine which block the trigger address is in and which byte within that block the data capture started. The code that illustrates this can be found in the gage_mem_read_dual, gage_mem_read_chan_a, gage_mem_read_chan_b and gage_mem_read_single routines for dual channel, channel a only, channel b only and single channel operation respectfully and is not repeated here. These routines can be found in the driver source code for each particular CompuScope board.

The acquisition to disk example program also illustrates this concept and does use the trigger address and is the best tutorial on improving transfer rates.

All of the routines in this section are called by the routines in group one to carry out their assigned tasks. Most of the internal error checking that is done when these routines are called by the group one routines is not performed when these routines are called directly. Most of these routines need never be called by the application program and when they are used directly care should be exercised to follow the examples provided in the accompanying descriptions.

gage_abort

Syntax

C:
#include <gage_drv.h>
void gage_abort (void);

Visual Basic:
sub gage_abort ()

Quick Basic:
sub gageabort()

Pascal:
procedure gage_abort;

Remarks

gage_abort is used to regain control of the CompuScope board, primarily in the event that a trigger event never occurs. This routine forces the board to a not busy state, thus allowing the board to be re-configured, rearmed and/or the memory to be accessed. The **gage_abort_capture** routine should usually be used instead as it works for multiple board configurations.

Return Value

None.

See also

gage_busy and gage_abort_capture. Sample Routines: low_level_capture.

Examples

C:
gage_abort ();

Visual Basic:
Call gage_abort ()

Quick Basic:
Call gageabort ()

Pascal:
gage_abort;

gage_fast_set_block_number

Syntax

C:
#include <gage_drv.h>
void gage_fast_set_block_number (uInt16 block);

Visual Basic:
sub gage_fast_set_block_number (ByVal block As Integer)

Quick Basic:
sub gagefastsetblocknumber (ByVal block As Integer)

Pascal:
procedure gage_fast_set_block_number (block : uInt16);

Remarks

gage_fast_set_block_number is used by the **gage_32k_to_buffer** and **gage_trigger_view_transfer** routines to quickly change the block number to connect the required block of the CompuScope hardware's internal buffer to the PC bus. This routine assumes that **gage_set_block_number** routine was the previous driver routine called. If it is not, then operation of the CompuScope hardware will be corrupted. A routine that needs more than one byte of data at a time can use this routine directly to speed up certain types of data retrieval transfer rates. However extra care must be taken to connect the appropriate block and to start in the right place (referenced from the trigger address). The block variable sets the internal memory block number. The memory blocks are organized with all of channel A's data occupying the first set of four kilosample blocks up to the total size of one channel's maximum memory size. The same number of blocks are available for channel B. This bank offset value can be queried from the driver by using the **gage_get_driver_info** routine. In the single channel mode the memory is organized with the first byte being in the data space for channel A and then the next byte in the address space for channel B. The data continues to alternate in this fashion throughout the memory depth. See the source code for these routines to gain extra information on this subject.

Return Value
None.

See also
gage_get_driver_info, gage_mem_read_chan_a, gage_mem_read_chan_b, gage_mem_read_dual, gage_mem_read_single and gage_set_block_number.

Examples
C:
gage_fast_set_block_number (block);

Visual Basic:
Call gage_fast_set_block_number (block);

Quick Basic:
CALL gagefastsetblocknumber (block);

Pascal:
gage_fast_set_block_number (block);

gage_get_data

Syntax

C:
#include <gage_drv.h>
void gage_get_data (void);

Visual Basic:
Sub gage_get_data ()

Quick Basic:
Sub gagegetdata ()

Pascal:
procedure gage_get_data;

Remarks

gage_get_data sets the CompuScope in its capture mode. After this call the board is digitizing the input data and placing the values into internal RAM while waiting for a trigger event, at which time "sample depth" more samples will be taken before the board will stop capturing the input data. The **gage_start_capture** routine should now be used to fully support the available hardware and more importantly when using multiple boards.

Return Value

None.

See also

gage_abort and gage_trigger_control.

Examples

C:
gage_get_data ();

Visual Basic:
gage_get_data

Quick Basic:
CALL gagegetdata ()

Pascal:
gage_get_data;

gage_get_data_high

Syntax

C:
#include “gage_drv.h”
#include “gage_low.h”
void gage_get_data_high (void);

Visual Basic:
Sub gage_get_data_high ()

Quick Basic:
REM $INCLUDE: 'GAGE_LOW.BAS'
Sub gagegetdatahigh ()

Pascal:
procedure gage_get_data_high;

Remarks

gage_get_data_high and **gage_get_data_low** are used for MASTER/SLAVE CompuScope applications. These routines are used to set the MASTER CompuScope in its capture mode, while **gage_get_data** is used to set the SLAVE boards in their capture mode. The procedure to start data capture in the MASTER/SLAVE mode is to call **gage_get_data_low** for the MASTER board, then call **gage_get_data** for all the SLAVE boards that are installed (in reverse order), and finally **gage_get_data_high** for the MASTER board. After these calls the boards are digitizing the input data and placing the values into internal RAM while waiting for a trigger event to occur on the MASTER board, at which time "sample depth" more samples will be taken before the board will stop capturing the input data. The driver routine **gage_start_capture** should now be used instead to fully support the available hardware and multiple boards. These routines are defined in GAGE_LOW.H and GAGE_LOW.BAS for C and Quick Basic.

Return Value
None.

See also

gage_get_data, gage_get_data_low and gage_start_capture. Sample Routines: low_level_transfer.

Examples
C:
gage_get_data_high ();

Visual Basic:
Call gage_get_data_high ()

Quick Basic:
Call gagegetdatahigh ()

Pascal:
gage_get_data_high;

gage_get_data_low

Syntax

C:
#include “gage_drv.h”
#include “gage_low.h”
void gage_get_data_low (void);

Visual Basic:
Sub gage_get_data_low ()

Quick Basic:
REM $INCLUDE: 'GAGE_LOW.BAS'
Sub gagegetdatalow ()

Pascal:
procedure gage_get_data_low;

Remarks

gage_get_data_low and **gage_get_data_high** are used for MASTER/SLAVE CompuScope applications. These routines are used to set the MASTER CompuScope in its capture mode, while **gage_get_data** is used to set the SLAVE boards in their capture mode. The procedure to start data capture in the MASTER/SLAVE mode is call **gage_get_data_low** for the MASTER board, then call **gage_get_data** for all the SLAVE boards that are installed (in reverse order), and finally **gage_get_data_high** for the MASTER board. After these calls the boards are digitizing the input data and placing the values into internal RAM while waiting for a trigger event to occur on the MASTER board, at which time "sample depth" more samples will be taken before the board will stop capturing the input data. The driver routine **gage_start_capture** should now be used instead to fully support the available hardware and multiple boards. These routines are defined in GAGE_LOW.H and GAGE_LOW.BAS for C and Quick Basic respectively.

Return Value
None.

See also

gage_get_data, gage_get_data_high and gage_start_capture. Sample Routines: low_level_transfer.

Examples
C:
gage_get_data_low ();

Visual Basic:
Call gage_get_data_low ()

Quick Basic:
CALL gage_get_data_low ()

Pascal:
gage_get_data_low ;

gage_init_clock

Syntax

C:
#include <gage_drv.h>
void gage_init_clock (void);

Visual Basic:
Sub init_clock ()

Quick Basic:
Sub initclock ()
Pascal:
procedure init_clock;

Remarks

gage_init_clock is used to initialize the CompuScope clock circuitry initially handled by the driver and also required when in the MASTER/SLAVE mode of operation. The gage_start_capture routine should be used instead of using this routine directly.

Return Value

None.

See also

gage_get_data_high and gage_get_data_low. Sample Routines: low_level_capture.

Examples

C:
gage_init_clock ();

Visual Basic:
Call gage_init_clock ()

Quick Basic:
Call gageinitclock ()

Pascal:
gage_init_clock;

gage_ram_full

Syntax

C:
#include <gage_drv.h>
int16 gage_ram_full (void);

Visual Basic:
Function gage_ram_full () As Integer

Quick Basic:
Function gageramfull% ()

Pascal:
function gage_ram_full: int16;

Remarks

gage_ram_full determines if CompuScope RAM was filled up (entirely overwritten with new data) during data capture. A return value equal to zero means only the data from location zero to location trigger address is valid pre-trigger data. The use of the **gage_calculate_addresses** routine is strongly recommended for determining the valid size of the memory captured.

Return Value

A non zero or true value is returned if the board's ram buffers are full, otherwise a false value is returned.

See also

gage_busy and gage_triggered.

Examples

C:
is_ram_full = gage_ram_full ()

Visual Basic:
is_ram_full = gage_ram_full

Quick Basic:
isramfull% = gageramfull%

Pascal:
is_ram_full = gage_ram_full

gage_set_block_number

Syntax

C:
#include "gage_drv.h"
void gage_set_block_number (uInt16 block);

Visual Basic:
Sub gage_set_block_number (ByVal block As Integer)

Quick Basic:
Sub gagesetblocknumber (ByVal block As Integer)

Pascal:
procedure gage_set_block_number (block: uInt16);

Remarks

gage_set_block_number is used by all of the memory read routines to connect the required block of the CompuScope hardware's internal buffer to the PC bus. Because the CompuScope drivers transfer memory from the card in 4 kilosample blocks, a routine that needs more than one byte of data at a time can use this routine directly to speed up certain types of data retrieval transfer rates. However, extra care must be taken to connect the appropriate block and to start in the right place (referenced from the trigger address). The block variable sets the internal memory block number. The memory blocks are organized with all of channel A's data occupying the first set of four kilosamples up to the total size of one channel's maximum sample memory size. The same number of blocks are available for channel B. For example, on a 16K CSLITE, there are 16K / 4 kilosamples = 4 blocks. Channel A occupies blocks 0 and 1, channel B occupies blocks 0 + bank_offset_value and 1+ bank_offset_value. This bank_offset_value can be queried from the driver using the **gage_get_driver_info** routine. In the single channel mode, the memory is organized with the first byte being in the data space for channel A and the next byte in the address space for channel B. The data continues to alternate in this fashion throughout the memory depth. See the source code for these routines to gain extra information on this subject. Please note that for the CompuScope CS220 - 32k and CS220 - 256k, memory cannot be written to. The CS250, CS225, CS220 and CSLITE sample size is 1 byte while the CS6012 and CS1012 sample size is 2 bytes (12 bits sign extended to 16 bits).

Return value

None

See also

gage_get_driver_info, gage_mem_read_chan_a, gage_mem_read_chan_b, gage_mem_read_dual and gage_mem_read_single.

Examples

C:
gage_set_block_number (block);
Visual Basic:
Call gage_set_block_number (block)

Quick Basic:
Call gagesetblocknumber (block)

Pascal:
gage_set_block_number (block);

gage_trigger_address

Syntax

C:
```
#include <gage_drv.h>
int32 gage_trigger_address (void);
```

Visual Basic:
```
Function gage_trigger_address () As Long
```

Quick Basic:
```
Function gagetriggeraddress& ()
```

Pascal:
```
function gage_trigger_address: int32;
```

Remarks

gage_trigger_address finds the address in CompuScope memory where the trigger event occurred.

Return Value

The long integer returned is the trigger address.

See also

gage_mem_read_dual and gage_mem_read_single.

Examples

C:
```
address = gage_trigger_address ();
```

Visual Basic:
```
address = gage_trigger_address
```

Quick Basic:
```
address& = gagetriggeraddress&
```

Pascal:
```
address := gage_trigger_address;
```

gage_triggered_aux

Syntax

C:
```
#include <gage_drv.h>
int16 gage_triggered_aux (void);
```

Visual Basic:
```
Function gage_triggered_aux () As Integer
```

Quick Basic:
```
Function gagetriggeredaux% ()
```

Pascal:
```
function gage_triggered_aux: int16;
```

Remarks

gage_triggered_aux determines if CompuScope has encountered an auxiliary trigger event. This status feature is only available on certain versions of the CompuScope hardware and is not standard on all boards.

Return Value

A non zero or true value is returned if the board's auxiliary trigger is active, otherwise a false value is returned.

See also

gage_triggered.

Examples

C:
```
aux_triggered    = gage_triggered_aux ()
```

Visual Basic:
```
aux_triggered = gage_triggered_aux
```

Quick Basic:
```
aux_triggered% = gagetriggeredaux%
```

Pascal:
```
aux_triggered = gage_triggered_aux;
```

Global Routines: Group Three. Dos Specific Routines

gage_channel_enable

Syntax

C:

```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_channel_enable (int16 source, int16 enable);
```

Quick Basic:

```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function% gagechannelenable (ByVal source As Integer, ByVal enable As Integer)
```

Remarks

gage_channel_enable can be used to prevent data capture on either channel A or channel B. With each call to **gage_capture_mode** both of the channels are re-enabled. In single channel mode both channels must be enabled in order to capture valid data. The channel can be either channel A or B and the constants **GAGE_CHAN_A** and **GAGE_CHAN_B** should be used to indicate their respective channels. The **enable** parameter, using the constants **GAGE_INPUT_ENABLE** and **GAGE_INPUT_DISABLE**, either enables a channel to be captured or disables the data capture on that channel (on the CompuScope LITE the **GAGE_INPUT_DISABLE** constants connects the input of both channels to the internal test signal). Again, note that in the single channel mode both of the channels must be enabled for the board to capture data correctly.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_input_control and gage_capture_mode.

Examples

C:

```
gage_channel_enable (GAGE_CHAN_A, GAGE_INPUT_DISABLE);
```

Quick Basic:

```
ret% = gagechannelenable% (GAGECHANA, GAGEINPUTDISABLE)
```

gage_set_coupling

Syntax

C:

```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_coupling (int16 channel, int16 coupling);
```

Quick Basic:

```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesetcoupling% (ByVal channel As Integer, ByVal coupling As Integer)
```

Remarks

gage_set_coupling is used to set up the input coupling for the CompuScope card. The channel can be either channel A, B or the external trigger and the constants **GAGE_CHAN_A**, **GAGE_CHAN_B** and **GAGE_EXTERNAL** should be used for the desired input source. The coupling can be either DC or AC as indicated by the constants **GAGE_DC** and **GAGE_AC** respectfully. The CompuScope LITE card does not support an external trigger with AC coupling.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_input_control.

Examples

C:

```
gage_set_coupling (GAGE_CHAN_A, GAGE_AC);  /*  Coupling is now AC.  */
```

Quick Basic:

```
ret% = gagesetcoupling% (GAGECHANA, GAGEAC)
```

gage_set_gain

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_gain (int16 channel, int16 gain);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesetgain% (ByVal channel As Integer, ByVal gain As Integer)
```

Remarks

gage_set_gain is used to set up the input gain for the CompuScope hardware. The channel can be either channel A, B or the external trigger and the constants **GAGE_CHAN_A**, **GAGE_CHAN_B** and **GAGE_EXTERNAL** should be used for the desired input source. There are six different gain values for each channel from divide by five to times ten. The constants and the hardware that supports them are listed in the following table.

Constant	Range	CS2125	CS6012	CS1012	CS250	CS225	CS220	CSLITE
GAGE_PM_5_V	+/ 5 Volts	Yes	Yes	Yes	Yes	Yes	Yes	No
GAGE_PM_2_V	+/ 2 Volts	Yes	Yes	Yes	No	No	Yes	No
GAGE_PM_1_V	+/ 1 Volt	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GAGE_PM_500_MV	+/ 500 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	No
GAGE_PM_200_MV	+/ 200 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GAGE_PM_100_MV	+/ 100 Millivolts	Yes	Yes	Yes	Yes	Yes	Yes	No

The external trigger input is limited to only two gain values, **GAGE_PM_5_V** and **GAGE_PM_1_V**. The CompuScope LITE does not support the plus / minus five volt input range.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_input_control and gage_trigger_control.

Examples

C:
```
gage_set_gain (GAGE_CHAN_A, GAGE_PM_100_MV);  /*  Channel A gain is now + / - 100 mv.  */
```

Quick Basic:
```
ret% = gagesetgain% (GAGECHANA, GAGEPM100MV)
```

gage_set_segment

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
void gage_set_segment (uInt16 segment);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Sub gagesetsegment (ByVal segment As Integer)
```

Remarks

gage_set_segment can be used to change the memory transfer segment for the CompuScope card's internal buffer. The high byte of the **segment** must be between A0 hex and EF hex and the low byte must be zero. Note that on the CompuScope 6012 and 1012 the memory segments must be even, for example C800 and D000 are valid segments but D700 is not valid since the block size of the CS6012 / 1012 is 8 kilobytes (4K 16-bit samples).

Return Value

None.

See also

gage_driver_initialize.

Examples

C:
```
gage_set_segment (0xde00);  /*  Move the default memory buffer transfer area.  */
```

Quick Basic:
```
CALL gagesetsegment (&hde00)
```

gage_set_trigger_depth

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_trigger_depth (int32 depth);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesettriggerdepth% (ByVal depth As Long)
```

Remarks

gage_set_trigger_depth is used to set up one of the trigger parameters for the CompuScope card. The **depth** parameter sets the trigger depth, that is the number of samples captured after the trigger event. The minimum value is 0. The resolution of this parameter is 64 bytes for the CompuScope 6012, 1012 250 and 225, 16 bytes for the CompuScope LITE and a power of two for the CompuScope 220. Setting the trigger depth can help reduce the capturing overhead when only a screens worth of data is required. Several defined constants are available to the application programmer to illustrate the use of this parameter. The constants **GAGE_POST_0K**, **GAGE_POST_1K**, **GAGE_POST_2K**, **GAGE_POST_4K**, **GAGE_POST_8K**, **GAGE_POST_16K**, **GAGE_POST_32K**, **GAGE_POST_64K**, **GAGE_POST_128K**, **GAGE_POST_256K**, **GAGE_POST_512K**, **GAGE_POST_1M**, **GAGE_POST_2M**, **GAGE_POST_4M** and **GAGE_POST_8M** are for the trigger depths of 0, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304 and 8388608 bytes respectfully.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_trigger_control.

Examples

C:
```
gage_set_trigger_depth (512);  /*  Capture a point for a 512 pixel window on the display screen.  */
```

Quick Basic:
```
ret = gagesettriggerdepth %(512)
```

gage_set_trigger_level

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_trigger_level (int16 level);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesettriggerlevel% (ByVal level As Integer)
```

Remarks

gage_set_trigger_level is used to set up one of the trigger parameters for the CompuScope card. The level can be any value between 0 and 255. The values are scaled internally to match the input range selected. The minimum input level is 0 while the maximum trigger level is 255 (corresponding to -1 volt and +1 volt in the times one gain range). Important note, if the trigger source is set to software triggering and the CompuScope 250 card is being used then this routine should not be called since if the wrong value is set with this routine the software trigger function **gage_software_trigger** may not cause the hardware to trigger.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_trigger_control.

Examples

C:
```
gage_set_trigger_level (192);  /*  Level midway between 0 volts and full voltage.  */
```

Quick Basic:
```
ret% = gagesettriggerlevel% (192)
```

* Prior to version 2.40 of the C driver, level was represented as a byte.

gage_set_trigger_slope

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_trigger_slope (int16 slope);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesettriggerslope% (ByVal slope As Integer)
```

Remarks

gage_set_trigger_slope is used to set up one of the trigger parameters for the CompuScope card. The trigger slope can be either positive or negative, using constants **GAGE_POSITIVE** and **GAGE_NEGATIVE**, and affects all sources except the software trigger. Important note, if the trigger source is set to software triggering and the CompuScope 250 card is being used then this routine should not be called since if the wrong value is set with this routine the software trigger function **gage_software_trigger** may not cause the hardware to trigger.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_trigger_control.

Examples

C:
```
gage_set_trigger_slope (GAGE_NEGATIVE);  /*  Make the trigger slope negative.  */
```

Quick Basic:
```
ret% = gagsettriggerslope% (GAGENEGATIVE)
```

gage_set_trigger_source

Syntax

C:
```
#include "gage_drv.h"
#include “gage_low.h”
int16 gage_set_trigger_source (int16 source);
```

Quick Basic:
```
REM $INCLUDE: 'GAGE_LOW.BAS'
Function gagesettriggersource% (ByVal source As Integer)
```

Remarks

gage_set_trigger_source is used to set up one of the trigger parameters for the CompuScope card. The source can be either channel A, B, EXTERNAL or SOFTWARE and the constants **GAGE_CHAN_A**, **GAGE_CHAN_B**, **GAGE_EXTERNAL** and **GAGE_SOFTWARE** should be used to indicate their respective sources. Important note, if the trigger source is set to software triggering and the CompuScope 250 card is being used then the **gage_set_trigger_slope** and **gage_set_trigger_level** routines should not be called since if the wrong value for trigger slope or trigger level are set then the software trigger function **gage_software_trigger** may not cause the hardware to trigger.

Return Value

A one is returned if successful and a zero is returned when the routine fails, and gage_error_code contains the error code.

See also

gage_trigger_control.

Examples

C:
```
gage_set_trigger_source (GAGE_SOFTWARE);  /*  Software triggering enabled.  */
```

Quick Basic:
```
ret% = gagesettriggersource% (GAGESOFTWARE)
```

Global Routines: Group Four.Board Specific Routines

The following board specific routines may be utilized to make use of the unique characteristics of each of the CompuScope boards. These routines are designed to take advantage of the hardware of a particular board and cannot be used with another CompuScope card. The CS1012 functions currently in this section take advantage of the ability of the CompuScope 6012 and the CompuScope 1012 to have different trigger levels simultaneously and to adjust the input offset to ground. There also routines to allow testing of the memory on the CS6012 or CS1012, which due to their different architecture, cannot be accessed directly.

The CompuScope 250 specific routines require the hardware to have equivalent time sampling capability. This allows the CS250 to capture up to 255 samples at a sample rate of 4 Ghz.

At this time, there are no board specific routines available for the other CompuScope boards.

CompuScope 6012 and 1012

cs1012_enable_test_memory

cs1012_test_memory_chan

Syntax

C:
#include “gage_drv.h”
#include “cs112drv.h”
void cs1012_enable_test_memory (int16 test);
void cs1012_test_memory_chan (int16 channel)

Basic:
not currently suitable for use with Visual Basic or Quick Basic.

Pascal:
procedure cs1012_enable_test_memory (test: int16);
procedure cs1012_test_memory_channel (channel: int16);

Remarks

cs1012_enable_test_memory enables the CPU to write to the ram buffers of the CompuScope 6012 or 1012 for the purpose of testing memory. The parameter test should be set to TRUE (non-zero) before writing to or reading from the CS6012 / 1012 and set to false when finished. **cs1012_test_memory_chan** connects either the memory of either channel A or channel B to the PC bus. The channel can be either GAGE_CHAN_A or GAGE_CHAN_B. Note that a write to channel A is 16 bits wide and a write to channel B is 8 bits wide.
NOTE: These routines are used mainly for internal development and their future inclusion in any drivers is not guaranteed. Application programs should generally have no need to use these routines.

Returns

None.

Examples

C:
cs1012_enable_test_memory (TRUE);
cs1012_test_memory_chan (GAGE_CHAN_A);

Pascal:
cs1012_enable_test_memory (TRUE);
cs1012_test_memory_chan (GAGE_CHAN_A);

cs1012_offset_adjust

Syntax

C:
```
#include “gage_drv.h”
#include “cs112drv.h”
void cs1012_offset_adjust (int16 channel, int16 offset_adjust);
```

Visual Basic:
```
Sub cs1012_offset_adjust (ByVal channel As Integer, ByVal offset_adjust As Integer)
```

Quick Basic:
```
Sub cs1012offsetadjust (ByVal channel As Integer, ByVal offsetadjust As Integer)
```

Pascal:
```
procedure cs1012_offset_adjust (channel, offset_adjust : int16);
```

Remarks

cs_1012_offset_adjust is used to adjust the DC offset on the currently selected CompuScope 6012 or 1012 card. The channel parameter can be either **GAGE_CHAN_A** or **GAGE_CHAN_B**. The offset_adjust can be any integer between -2047 to +2047, where -2047 is the most negative offset of the current amplitude range and 2047 the most positive. If the offset_adjust is 0, the routine performs an automatic calibration of the CS6012 / 1012. In this case, the inputs of the board must be connected to the ground of the circuit under test. Note that this routine should be called whenever the input range is called or when switching from dual channel mode to single channel mode or vice versa. Note that C programs must include the cs112drv.h header file.

Return value:

None.

Examples

C:
```
cs1012_offset_adjust (GAGE_CHAN_A);
```

Visual Basic:
```
Call cs1012_offset_adjust (GAGE_CHAN_A)
```

Quick Basic:
```
Call cs1012offsetadjust (GAGECHANA)
```

Pascal:
```
cs1012_offset_adjust (GAGE_CHAN_A);
```

cs1012_trigger_control_2

Syntax

C:
#include “gage_drv.h”
#include “cs112drv.h”
int16 cs1012_trigger_control_2 (int16 source_1, int16 source_2, int16 ext_coupling, int16 ext_gain, int16 slope_1, int16 slope_2, int16 level_1, int16 level_2, int32 depth, int16 trigger_from_bus, int16 trigger_to_bus)

Visual Basic:
Function cs1012_trigger_control_2 (ByVal source_1 As Integer, ByVal source_2 As Integer, ByVal ext_coupling As Integer, ByVal ext_gain As Integer, ByVal slope_1 As Integer, ByVal slope_2 As Integer, ByVal level_1 As Integer, ByVal level_2 As Integer, ByVal depth As Long, ByVal trigger_from_bus As Integer, ByVal trigger_to_bus As Integer) As Integer

Quick Basic:
Function cs1012triggercontrol2% (ByVal source1 As Integer, ByVal source2 As Integer, ByVal extcoupling As Integer, ByVal extgain As Integer, ByVal slope1 As Integer, ByVal slope2 As Integer, ByVal level1 As Integer, ByVal level2 As Integer, ByVal depth As Long, ByVal triggerfrombus As Integer, ByVal triggertobus As Integer)

Pascal:
function cs1012_trigger_control_2 (source_1, source_2, ext_coupling, ext_gain, slope_1, slope_2, level_1, level_2 : int16; depth : int32; trigger_from_bus, trigger_to_bus : int16) : int16;

Remarks

cs1012_trigger_control_2 is a CompuScope 6012 / 1012 specific routine to take advantage of it's ability to have 2 different trigger levels and is used to set up the trigger parameters of the CompuScope . The parameters **source_1** and **source_2** can be either channel A, B, EXTERNAL or SOFTWARE and the constants **GAGE_CHAN_A**, **GAGE_CHAN_B**, **GAGE_EXTERNAL** and **GAGE_SOFTWARE** should be used to indicate their respective sources. The two sources do not have to be the same. To disable one of the triggers, it should be set to **GAGE_SOFTWARE**. If a software trigger is desired, then both trigger sources should be set to **GAGE_SOFTWARE**.

The **ext_coupling** can be either DC or AC for the external trigger input as indicated by the constants **GAGE_DC** and **GAGE_AC** respectfully .

There are two different **ext_gain** values for the external trigger input. These constants are **GAGE_PM_5_V** and **GAGE_PM_1_V**.

The trigger **slope_1** and **slope_2** can be either positive or negative, using constants **GAGE_POSITIVE** and **GAGE_NEGATIVE**, and affects all sources except the software trigger.

The **level_1** and **level_2** parameters can be any value between 0 and 255. The values are scaled internally to match the trigger input range selected. The minimum input level is 0 while the maximum trigger level is 255.

The **depth** parameter sets the trigger depth, that is the number of samples captured after the trigger event. The minimum value is 0. The resolution of this parameter for the CompuScope 6012 and 1012 is 64 bytes (e.g. 256, 320, 384, ..., 8128, 8192, ...).This can help reduce the capturing overhead when only a few points of data are required. Several defined constants are available to the application programmer to illustrate the use of this parameter. The constants **GAGE_POST_0K**, **GAGE_POST_1K**, **GAGE_POST_2K**, **GAGE_POST_4K**, **GAGE_POST_8K**, **GAGE_POST_16K**, **GAGE_POST_32K**, **GAGE_POST_64K** and **GAGE_POST_128K** , **GAGE_POST_256K**, **GAGE_POST_512K**, **GAGE_POST_1M**, **GAGE_POST_2M**, **GAGE_POST_4M** and **GAGE_POST_8M** are for the trigger depths of 0, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304 and 8388608 bytes respectively.

The **trigger_from_bus** parameter should always be set to TRUE (non-zero). In a multiple card system, **trigger_to_bus** parameter should be set to TRUE (non-zero) for the master and FALSE (zero) for the slave(s).

Note that C programs must include the cs112drv.h header file.

Return value

A one is returned if successful and a zero is returned when the routine fails, and **gage_get_error_code** may be called to obtain the error code.

See also

gage_trigger_control, gage_capture_mode and gage_input_control.

Examples:

C:
```
cs1012_trigger_control_2 (GAGE_CHAN_A, GAGE_EXTERNAL, GAGE_DC, GAGE_PM_1_V,
        GAGE_POSITIVE, GAGE_NEGATIVE, 160, 100, GAGE_POST_16K, TRUE, TRUE );
```

Visual Basic:
```
dummy = cs1012_trigger_control_2 (GAGE_CHAN_A, GAGE_EXTERNAL, GAGE_DC,
        GAGE_PM_1_V, GAGE_POSITIVE, GAGE_NEGATIVE, 160, 100, GAGE_POST_16K,
        TRUE,   TRUE )
```

Quick Basic:
```
dummy% = cs1012triggercontrol2% (GAGECHANA, GAGEEXTERNAL, GAGEDC, GAGEPM1V,
        GAGEPOSITIVE, GAGENEGATIVE, 160, 100, GAGEPOST16K, TRUE, TRUE )
```

Pascal:
```
cs1012_trigger_control_2 (GAGE_CHAN_A, GAGE_EXTERNAL, GAGE_DC, GAGE_PM_1_V,
        GAGE_POSITIVE, GAGE_NEGATIVE, 160, 100, GAGE_POST_16K, TRUE, TRUE );
```

CompuScope 250

cs250_enable_ets

Syntax

C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_enable_ets (int16 ets);

Visual Basic:
Sub cs250_enable_ets (ByVal ets As Integer)

Quick Basic:
Sub cs250enableets (ByVal ets As Integer)

Pascal:
procedure cs250_enable_ets (ets: int16);

Remarks

cs250_enable_ets must be called to enable the equivalent time sampling hardware on a specially equipped CompuScope 250. If **ets** is 1, the equivalent time sampling hardware is enabled. If the parameter is a 0, the hardware is disabled. The power-on default is disabled. For equivalent time sampling, the trigger depth (set by calling **gage_trigger_control**) should be no greater then 64 points. Note that the CS250 must be set to 50 Mhz, dual channel mode for equivalent time sampling..

Return value

None.

See Also

Sample program CS250ETS.EXE

Examples

C:
cs250_enable_ets (1);

Visual Basic:
Call cs250_enable_ets (1)

Quick Basic:
Call cs250enableets (1)

Pascal:
cs250_enable_ets (1);

cs250_ets_average_capture

Syntax

C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_ets_average_capture (int16 auto_trigger, int32 timeout, int16 avgs);

Visual Basic:
Sub cs250_ets_average_capture (ByVal auto_trigger As Integer, ByVal timeout As Long, ByVal avgs As Integer)

Quick Basic:
Sub cs250etsaveragecapture (ByVal autotrigger As Integer, ByVal timeout As Long, ByVal avgs As Integer)

Pascal:
procedure cs250_ets_average_capture (auto_trigger: int16; timeout: int32; avgs: int16);

Remarks

cs250_ets_average_capture performs an equivalent time sampling capture similar to the routine **cs250_ets_capture** except it captures the same equivalent time sampling delay sample an **avgs** number of times, discards the minimum and maximum values and averages the remaining data points. **avgs** must be a minimum of 3 and a maximum of 22. Setting the **auto_trigger** parameter to 1 will cause a software trigger to occur immediately. Note that the trigger source must be set to **GAGE_SOFTWARE** via a call to **gage_trigger_control** for this parameter to have an effect. The **timeout** parameter can be used to set the trigger timeout period in the case of a hardware trigger. The **timeout** value is in 10ths of milliseconds and is set on a per acquisition basis. If the timeout period is exceeded without a trigger event occurring, the driver will force a trigger. The **cs250_enable_ets** routine must be called prior to this routine in order to enable the equivalent time sampling mode. The driver routine **gage_select_board** must be called with a parameter of 1 before that. The equivalent time sampling mode must be disabled by calling **cs250_enable_ets** with a parameter of 0 before the CompuScope 250 can be used in a normal mode again. The data is returned in the internal buffers of the driver and can be accessed using the appropriate **gage_mem_read_xxxx** routine.

Return value

None.

See Also

cs250_ets_capture, cs250_enable_ets, gage_select_board and Sample program CS250ETS.EXE

Examples

C:
cs250_ets_average_capture (1, 10, 20);

Visual Basic:
Call cs250_ets_average_capture (1, 10, 20)

Quick Basic:
Call cs250etsaveragecapture (1, 10, 20)

Pascal:
cs250_ets_average_capture (1, 10, 20);

Note: Because Visual Basic or Quick Basic have no variables of type byte, the buffer must be an integer array. Because the CS250 has a sample size off 1 byte, the high and low bytes of the integer must be extracted. This can be done as follows:

```
j = 0
For i = 0 To (Size Of Buffer \ 2) - 1
    temp = Clng (65535) And Clng (buffer(i))
    temp_buffer(j) = temp And 255   ' extract the high byte and store as an integer
    temp_buffer(j+1) = (temp \ 256) And 255  ' extract the low byte and store as an integer
    j = j + 2
Next
```

cs250_ets_average_capture_2

Syntax

C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_ets_average_capture_2 (int16 auto_trigger, int32 timeout, int 16 avgs, uInt8 far *a_buffer, uInt8 far *b_buffer);

Visual Basic:
Sub cs250_ets_average_capture_2 (ByVal auto_trigger As Integer, ByVal timeout As Long, ByVal avgs As Integer, a_buffer As Integer, b_buffer As Integer)

Quick Basic:
Sub cs250etsaveragecapture2 (ByVal autotrigger As Integer, ByVal timeout As Long, ByVal avgs As Integer, SEG abuffer As Integer, SEG bbuffer As Integer)

Pascal:
procedure cs250_ets_average_capture_2 (auto_trigger: int16; timeout: int32; avgs: int16; var a_buffer: bytearray; var b_buffer: bytearray);

Remarks

cs250_ets_average_capture_2 performs an equivalent time sampling capture similar to the routine **cs250_ets_capture_2** except it captures the same equivalent time sampling delay sample an **avgs** number of times, discards the minimum and maximum values and averages the remaining data points. **avgs** must be a minimum of 3 and a maximum of 22. Setting the **auto_trigger** parameter to 1 will cause a software trigger to occur immediately. Note that the trigger source must be set to **GAGE_SOFTWARE** via a call to **gage_trigger_control** for this parameter to have an effect. The **timeout** parameter can be used to set the trigger timeout period in the case of a hardware trigger. The **timeout** value is in 10ths of milliseconds and is set on a per acquisition basis. If the timeout period is exceeded without a trigger event occurring, the driver will force a trigger. The **cs250_enable_ets** routine must be called prior to this routine in order to enable the equivalent time sampling mode. The driver routine **gage_select_board** must be called with a parameter of 1 before that. The equivalent time sampling mode must be disabled by calling **cs250_enable_ets** with a parameter of 0 before the CompuScope 250 can be used in a normal mode again. The samples are returned in the supplied buffers. Channel A’s data will be returned in the first buffer and channel B’s data in the second 256. In Visual Basic or Quick Basic, the buffer should be an integer array. The bytes will then have to be extracted from the high and low bytes of each integer.

Return value

None.

See Also

cs250_ets_capture_2, cs250_enable_ets, gage_select_board and Sample program CS250ETS.EXE

Examples

C:
```
cs250_ets_average_capture_2 (1, 10, 20, (uInt8 far *)a_buffer, (uInt8 far *)b_buffer);
```

Visual Basic:
```
Call cs250_ets_average_capture_2 (1, 10, 20, a_buffer (0), b_buffer (0))
```

Quick Basic:
```
Call cs250etsaveragecapture2 (1, 10, 20, SEG abuffer(0), SEG bbuffer(0))
```

Pascal:
```
cs250_ets_average_capture_2 (1, 10, 20, a_buffer, b_buffer);
```

Note: Because Visual Basic or Quick Basic have no variables of type byte, the buffer must be an integer array. Because the CS250 has a sample size off 1 byte, the high and low bytes of the integer must be extracted. This can be done as follows:

```
j = 0
For i = 0 To (Size Of Buffer \ 2) - 1
    temp = Clng (65535) And Clng (buffer(i))
    temp_buffer(j) = temp And 255   ' extract the high byte and store as an integer
    temp_buffer(j+1) = (temp \ 256) And 255  ' extract the low byte and store as an integer
    j = j + 2
Next
```

cs250_ets_capture

Syntax
C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_ets_capture (int16 auto_trigger, int32 timeout);

Visual Basic:
Sub cs250_ets_capture (ByVal auto_trigger As Integer, ByVal timeout As Long)

Quick Basic:
Sub cs250etscapture (ByVal autotrigger As Integer, ByVal timeout As Long)

Pascal:
procedure cs250_ets_capture (auto_trigger: int16, timeout As int32);

Remarks
cs250_ets_capture performs an equivalent time sampling capture. Equivalent time sampling can currently only be done in dual channel mode. The **cs250_enable_ets** routine must be called prior to this routine in order to enable the equivalent time sampling mode. The driver routine **gage_select_board** must be called with a parameter of 1 before that. The equivalent time sampling mode must be disabled by calling **cs250_enable_ets** with a parameter of 0 before the CompuScope 250 can be used in a normal mode again. Setting the **auto_trigger** parameter to 1 will cause a software trigger to occur immediately. Note that the trigger source must be set to **GAGE_SOFTWARE** via a call to **gage_trigger_control** for this parameter to have an effect. The **timeout** parameter can be used to set the trigger timeout period in the case of a hardware trigger. The **timeout** value is in 10ths of milliseconds and is set on a per acquisition basis. If the timeout period is exceeded without a trigger event occurring, the driver will force a trigger. The data is returned in the internal buffers of the driver and can be accessed using the appropriate **gage_mem_read_xxxx** routine.

Return value
None.

See Also
cs250_enable_ets, gage_select_board and Sample program CS250ETS.EXE

Examples

C:
cs250_ets_capture (1, 10);

Visual Basic:
Call cs250_ets_capture (1, 10)

Quick Basic:
Call cs250etscapture (1, 10)

Pascal:
cs250_ets_capture (1, 10);

cs250_ets_capture_2

Syntax
C:
#include "gage_drv.h"
#include "cs250drv.h"
void cs250_ets_capture_2 (int16 auto_trigger, int32 timeout, uInt8 far *a_buffer, uInt8 far *b_buffer);

Visual Basic:
Sub cs250_ets_capture_2 (ByVal auto_trigger As Integer, ByVal timeout As Long, a_buffer As Integer, b_buffer As Integer)
Quick Basic:
Sub cs250etscapture2 (ByVal autotrigger As Integer, ByVal timeout As Long, SEG abuffer As Integer, SEG bbuffer As Integer)
Pascal:
procedure cs250_ets_capture_2 (auto_trigger: int16; timeout: int32; var a_buffer: bytearray; var b_buffer: bytearray);

Remarks
cs250_ets_capture_2 performs an equivalent time sampling capture. Equivalent time sampling can currently only be done in dual channel mode. The **cs250_enable_ets** routine must be called prior to this routine in order to enable the equivalent time sampling mode. The driver routine **gage_select_board** must be called with a parameter of 1 before that. The equivalent time sampling mode must be disabled by calling **cs250_enable_ets** with a parameter of 0 before the CompuScope 250 can be used in a normal mode again. Setting the **auto_trigger** parameter to 1 will cause a software trigger to occur immediately. Note that the trigger source must be set to **GAGE_SOFTWARE** via a call to **gage_trigger_control** for this parameter to have an effect. The **timeout** parameter can be used to set the trigger timeout period in the case of a hardware trigger. The **timeout** value is in 10ths of milliseconds and is set on a per acquisition basis. If the timeout period is exceeded without a trigger event occurring, the driver will force a trigger. The samples are returned via the supplied buffers which must be large enough to hold 256 samples each. Channel A's data will be returned in the first buffer and channel B's data in the next. For Visual Basic and Quick Basic, the supplied buffer should hold 256 integers. The samples will then have to be extracted, similar to how it is done for the block transfer routines.

See Also
cs250_enable_ets, gage_select_board and Sample program CS250ETS.EXE

Examples
C:
cs250_ets_capture (0, 10, (uInt8 far *)a_buffer, (uInt8 far *)b_buffer);

Visual Basic:
Call cs250_ets_capture (0, 10, a_buffer(0), b_buffer(0))

Quick Basic:
Call cs250etscapture (0, 10, SEG abuffer(0), SEG bbuffer(0))

Pascal:
cs250_ets_capture (0, 10, a_buffer, b_buffer);

cs250_ets_detect

Synatx:

C:
```
#include “gage_drv.h”
#include “cs250drv.h”
int16 cs250_ets_detect (void);
```

Visual Basic:
```
Function cs250_ets_detect () As Integer
```

Quick Basic:
```
Function cs250etsdetect%
```

Pascal:
```
function cs250_ets_detect: int16;
```

Remarks:

cs250_ets_detect will detect if the equivalent time sampling option is available on the current installed CompuScope 250.

Return Value:

A one is returned if equivalent time sampling hardware is found, otherwise a zero is returned.

See Also:

cs250_ets_enable.

Examples:

C:
```
found = cs250_ets_detect ();
```

Visual Basic:
```
found = cs250_ets_detect
```

Quick Basic:
```
found = cs250etsdetect
```

Pascal:
```
found := cs250_ets_detect;
```

cs250_mem_read_ets_data

Syntax

C:
```
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_mem_read_ets_data (int32 location, uInt8 far *abuf; uInt8 far *bbuf);
```

Visual Basic:
```
Sub cs250_mem_read_ets_data (ByVal location As Long, abuf As Integer, bbuf As Integer)
```

Quick Basic:
```
Sub cs250memreadetsdata (ByVal location As Long, SEG abuf As Integer, SEG bbuf As Integer)
```

Pascal:
```
procedure cs250_mem_read_ets_data (location: int32; var abuf: bytearray; var bbuf: bytearray);
```

Remarks

cs250_mem_read_ets_data gets one point out of each of buffer at the offset specified by **location** and places channel A’s value in **abuf** and channel B’s value in **bbuf**. **location** is the raw trigger address plus 10. The raw trigger address can be obtained by calling **gage_trigger_address**.

Return value

None.

See Also

gage_trigger_address, cs250_ets_capture and Sample program CS250ETS.EXE

Examples

C:
```
cs250_mem_read_ets_data (address, (uInt8 far *)abuf, (uInt8 far *)bbuf);
```

Visual Basic:
```
Call cs250_mem_read_ets_data (address, abuf(0), bbuf(0))
```

Quick Basic:
```
Call cs250memreadetsdata (address, SEG abuf(0), SEG bbuf(0))
```

Pascal:
```
cs250_mem_read_ets_data (address, abuf, bbuf);
```

cs250_set_ets_delay

Syntax

C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_set_ets_delay (int16 ets_delay);

Visual Basic:
Sub cs250_set_ets_delay (ByVal ets_delay As Integer)

Quick Basic:
Sub cs250setetsdelay (ByVal ets_delay As Integer)

Pascal:
procedure cs250_set_ets_delay (ets_delay : int16);

Remarks

cs250_set_ets_delay is used to set the delay between equivalent time samples on a specially equipped CompuScope 250. The parameter **ets_delay**, which must be a number between 0 and 255, sets the delay for each sample to be taken. The resolution of the delay is dependent on what was set in the call to **cs250_set_ets_rate**. Therefore, if the resolution was set in **cs250_set_ets_rate** to be 250 picoseconds, calling **cs_250_set_ets_delay** with a 0 will capture with no delay, with a one will cause a 250 ps delay, with two will cause a 500 ps delay, etc.

Return value

None.

See Also

cs250_set_ets_rate and Sample program CS250ETS.EXE

Examples

C:
cs250_set_ets_delay (0);

Visual Basic:
Call cs250_set_ets_delay (0)

Quick Basic:
Call cs250setetsdelay (0)

Pascal:
cs250_set_ets_delay (0);

cs250_set_ets_rate

Syntax

C:
#include “gage_drv.h”
#include “cs250drv.h”
void cs250_set_ets_rate (int16 ets_rate);

Visual Basic:
Sub cs250_set_ets_rate (ByVal ets_rate As Integer)

Quick Basic:
Sub cs250setetsrate (ByVal ets_rate As Integer)

Pascal:
procedure cs250_set_ets_rate (ets_rate : int16);

Remarks

cs250_set_ets_rate is used to set the resolution of the delay for equivalent time sampling on a specially equipped CompuScope 250. Equivalent time sampling must first be enabled with **cs250_enable_ets** and the CS250 must be in 50 Mhz dual channel mode. The parameter **ets_rate** must be an integer between 0 and 3. A zero will cause no delay, a one will produce a 1000 picosecond delay for a 1 GHz sample rate, a two will cause a 500 picosecond delay for a 2 GHz sample rate and a three gives a 250 picosecond delay which produces a 4 GHz sample rate. Invalid values are ignored.

Return value

None.

See Also

cs250_set_ets_delay, cs250_enable_ets and Sample program CS250ETS.EXE

Examples

C:
cs250_set_ets_rate (3);

Visual Basic:
Call cs250_set_ets_rate (3)

Quick Basic:
Call cs250setetsrate (3)

Pascal:
cs250_set_ets_rate (3);

CompuScope 225

There are currently no board specific routines available for the CompuScope 225.

CompuScope 220

There are currently no board specific routines available for the CompuScope 220.

CompuScope Lite

There are currently no board specific routines available for the CompuScope Lite.

Sample Routines

The following sections are designed to show the most important CompuScope driver function calls in the context of an actual routine. These routines are not meant to be a complete program by themselves, although they can be included in your own programs. The three sections that follow show example routines for initialization, preparation, getting the important addresses, data capture and data transfer in C, Pascal and Basic. Except for the obvious differences between Windows and DOS in terms of display output and memory management, the routines should be usable under both.

The Basic routines have been written using the syntax of Visual Basic. In Quick Basic, underscores are not allowed in variable or subroutine names. Also, the syntax of some of the driver functions are different for Quick Basic and Visual Basic, most notably the use of the SEG keyword and differences in how strings are handled in Quick Basic. Check the file, GAGE_DRV.BAS, on your distribution diskette for the proper syntax.

Many of the sample routines included in this section are taken from the sample programs on the distribution diskette. Although every effort has been made to keep them error-free, the sample programs often get updated after the manual has been completed. If any discrepancy occurs between the sample routines, driver calls or explanations and the actual sample programs or driver code on the distribution diskette, then the version on the diskette will be the correct one.

Variable Definitions for Examples: C.

This section lists the definitions and variables assumed to be present for the various sample Routines. See the section on structures, types and definitions for an explanation of the user defined memory types.

```
#define  POINTS                 4096
#define  DEFAULT_SEGMENT   0xeD000
#define DEFAULT_INDEX          0x0200

int16                          sample_offset;
int16                          sample_resolution;
int16                          multiple_record_available;
int                            use_multiple_record;
int32                          address;
int                            use_trigger_view_transfer;
uInt8                          a_buffer[POINTS];
uInt8                          b_buffer[POINTS >> 1];
long                           buffer_mask;      /* equal to max_available_memory  */
int32                          current_memory_size;
int32                          channel_ram_size;
```

The following definitions are used for some of the examples which have their roots in the C sample programs. The additional "types" are defined and initialized in the sample programs or their support files on the distribution diskette..

```
#define  INTERPOLATE_TRIGGER_DISABLE   32

boarddef            board  = {
                           GAGE_DUAL_CHAN,
                           21,
                           GAGE_PM_1_V,
                           GAGE_DC,
                           GAGE_PM_1_V,
                           GAGE_DC,
                           GAGE_CHAN_A,
                           GAGE_POSITIVE,
                           GAGE_POST_8K,
                           128,
                           GAGE_PM_1_V,
                           GAGE_DC,
                    };

typedef struct {
          int16              rate;
          int16              mult;
          uInt32             sr_flag;
          float              sr_calc;
          char               sr_text;
} srtype;

gage_board_type              gage_current_card;
char                         board_loc_file[260];
srtype                       sample_rate_table[40];
```

```
gage_driver_info_type    gage_driver_info;
uInt16                   gage_board_location[GAGE_B_L_BUFFER_SIZE];
int32                    trigger_address[GAGE_B_L_MAX_CARDS * 2];
int32                    starting_address[GAGE_B_L_MAX_CARDS * 2];
int32                    ending_address[GAGE_B_L_MAX_CARDS * 2];
int                      interpolate_trigger;
int16                    interpolate_trigger_channel;
```

Sample Routines: C.

```
int         init_driver_hardware (void)

/*          Initializing the CompuScope driver and hardware.      */
{
            int16                 boards_in_system;

            gage_get_config_filename (board_loc_file);
            if (gage_read_config_file (board_loc_file, (uInt16 far *)gage_board_location) < 0)  {
                        printf ("Errors in GAGESCOP.INC, defaults used.\n");
                        gage_set_records ((uInt16 far *)gage_board_location, 0, DEFAULT_SEGMENT,
                                                                        DEFAULT_INDEX, 0, 0);
                        for (j = 1; j < GAGE_B_L_MAX_CARDS;  j ++)
                                    gage_set_records (uInt16 far )gage_board_location, j, 0, 0, 0, 0);
            }
            gage_driver_initialize((uInt16 far *)gage_board_location, GAGE_MEMORY_SIZE_TEST);
            if ((boards_in_system = gage_driver_initialize((uInt16 far *)gage_board_location,
                                                              GAGE_MEMORY_SIZE_TEST)) == 0) {
                        uInt16    segment, index, board_type, status;

                        gage_get_records ((uInt16 far *)gage_board_location, 0, &segment, &index,
                                                              &board_type, &status);
                        printf ("No CompuScope boards found, Error code = %02x.\n", status);
                        return (0);
            }else if (boards_in_system < 0)  {
                        boards_in_system = -boards_in_system;
                        printf ("Not all CompuScope boards found\n");
            }
            gage_select_board(1);
            gage_get_driver_info (&gage_driver_info);
            current_board_type = B_T_DOUBLE_UP_BITS (gage_driver_info.board_type);
            /* B_T_DOUBLE_UP_BITS is a macro defined in the support file structs.h. */
            current_memory_size = gage_driver_info.max_memory;
            printf ("CompuScope %s board found.\n", board_type_and_size_to_text
                                  (gage_driver_info.board_type, gage_driver_info.max_memory));
            /* board_type_and_size_to_text is a support routine that determines the name and
               memory size of board_type. It is available in driv_supp.c.   */
            sample_offset = gage_driver_info.sample_offset;
            sample_resolution = gage_driver_info.sample_resolution;

/*Detect if multiple recording option is present on the installed CompuScope hardware.    */

            if ((multiple_record_available = gage_detect_multiple_record ()) == 0)
                        use_multiple_record = 0;
            return (boards_in_system);
}/* End of init_driver_hardware. */
```

```
void    prepare_for_capture (boarddef  *board)

/*      Preparing the CompuScope card for data capture.
        Initialize the hardware with the new values.
*/
{
        int16           j, boards_in_system;

        boards_in_system = gage_get_boards_found ();
        for ( j = 1; j <= boards_in_system; j++) {
                gage_select_board (j);
                gage_multiple_record (0); /* Set to 1 to enable multiple record if available. */
        }
        gage_select_board (1);          /*  Master.  */
        gage_get_driver_info (&gage_driver_info);
        gage_capture_mode (board->opmode, sample_rate_table[board->srindex].rate,
                        sample_rate_table[board->srindex].mult);
        gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board->couple_a,
                        board->range_a);
        if (board->opmode == GAGE_SINGLE_CHAN)
                gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                board->couple_a,board->range_a);
        else
                gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                board->couple_b,board->range_b);

        for (j = 2 ; j <= boards_in_system ; j++)  {
                gage_select_board (j);  /*  Slaves.  */
                If ((gage_driver_info.board_type & (GAGE_ASSUME_CS1012 |
                        GAGE_ASSUME_CS6012)) && (board->srindex != 40))
                /* External clock is not in use. */
                        gage_capture_mode (board->opmode,
                                sample_rate_table[board->srindex].rate,
                        sample_rate_table[board->srindex].mult);
                else
                        gage_capture_mode (board->opmode, 0, GAGE_EXT_CLK);
                if (board->opmode == GAGE_SINGLE_CHAN)
                        gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                        board->couple_a,board->range_a);
                else
                        gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                        board->couple_b,board->range_b);
                gage_trigger_control (GAGE_SOFTWARE, GAGE_DC, GAGE_PM_1_V,
                                        GAGE_POSITIVE, 0x7f, board->depth);
        }
        gage_select_board (1);  /*  Master.  */
        gage_trigger_control (board->source, board->couple_e, board->range_e, board->slope,
                                        board->level, board->depth);
        gage_get_driver_info (&gage_driver_info);
        channel_ram_size = gage_driver_info.max_available_memory;
        timing (100);   /*  Allow relays to settle.  */

}       /*      End of prepare_for_capture ().      */
```

```
void        driver_support (void)
{           /* Getting the CompuScope driver version and supported board types from the driver. */

            char                        str[80];
            int                         types = 0;
            uInt16                      major_version, minor_version, board_support;
            int32                       gdi_size, gbt_size, gbt_routines;
            gage_driver_info_type       *gdi;
            gage_board_type             gbm;

            gage_get_driver_info_structure (&major_version, &minor_version, &board_support,
                        &gdi, &gdi_size);
            gage_get_current_drv_structure (&gbm, &gbt_size, &gbt_routines);

            printf ("CompuScope Driver Version %d.%02d.\n", major_version, minor_version);

            types = (((board_support & GAGE_ASSUME_CS2125) != 0)
                                    +       ((board_support & GAGE_ASSUME_CS6012) != 0)
                                    +       ((board_support & GAGE_ASSUME_CS1012) != 0)
                                    +       ((board_support & GAGE_ASSUME_CS250) != 0)
                                    +       ((board_support & GAGE_ASSUME_CS225) != 0)
                                    +       ((board_support & GAGE_ASSUME_CS220) != 0)
                                    +       ((board_support & (GAGE_ASSUME_CSLITE |
                                                    GAGE_ASSUME_CSLITE15)) != 0));

            if (types == 0)  {
                        printf ("Driver supports no boards. ");
                        return;
            }
            strcpy (str, "Driver support for ");
            if (board_support & GAGE_ASSUME_CS2125)  {
                        strcat (str, "CS2125");
                        types--;
                        if (types > 1)
                                    strcat (str, ", ");
                        else if (types > 0)
                                    strcat (str, " and ");
            }
            if (board_support & GAGE_ASSUME_CS6012)  {
                        strcat (str, "CS6012");
                        types--;
                        if (types > 1)
                                    strcat (str, ", ");
                        else if (types > 0)
                                    strcat (str, " and ");
            }
            if (board_support & GAGE_ASSUME_CS1012)  {
                        strcat (str, "CS1012");
                        types--;
                        if (types > 1)
                                    strcat (str, ", ");
                        else if (types > 0)
                        strcat (str, " and ");
        }
        if (board_support & GAGE_ASSUME_CS250)  {
                strcat (str, "CS250");
                types--;
                if (types > 1)
                        strcat (str, ", ");
                else if (types > 0)
                        strcat (str, " and ");
        }
        if (board_support & GAGE_ASSUME_CS225)  {
                strcat (str, "CS225");
                types--;
                if (types > 1)
                        strcat (str, ", ");
                else if (types > 0)
                        strcat (str, " and ");
        }
        if (board_support & GAGE_ASSUME_CS220)  {
                strcat (str, "CS220");
                types--;
                if (types > 1)
                        strcat (str, ", ");
                else if (types > 0)
                        strcat (str, " and ");
        }
        if (board_support & (GAGE_ASSUME_CSLITE | GAGE_ASSUME_CSLITE15))  {
                strcat (str, "CSLITE");
                types--;
                if (types > 1)
                        strcat (str, ", ");
                else if (types > 0)
                        strcat (str, " and ");
        }
        strcat (str, ".\n");
        printf (str);
}       /*      End of driver_support ().   */



void    InitMemPtrs (void)
{
        int             i;
        int16           channel;

if (USE_INTERPOLATED_TRIGGER_CODE) {
        /*  See the sample programs on the distribution diskette to see how to
            use the support routines for interpolated trigger.          */
        init_memory_pointers (1, board.opmode, sample_rate_table[board.srindex].sr_calc,
                        trigger_address, starting_address, ending_address);
}else{
        for (i = 0; i < (boards_in_system * 2); i += (1 + gage_current_card->one_channel_active)) {
                gage_select_board ((i >> 1) + 1);
                channel = (i & 1) + GAGE_CHAN_A;
        /* When not using interpolated trigger, use this line. */
                gage_calculate_addresses (channel, board.opmode,
                        sample_rate_table[board.srindex].sr_calc,
                        &trigger_address[i], &starting_address[i],
                        &ending_address[i]);
        }
}       /*  End of InitMemPtrs ().  */


void    AcquireCheck (void)
{
        int         check;

        gage_get_driver_info (&gage_driver_info);
        gage_start_capture (gage_driver_info.trigger_source == GAGE_SOFTWARE);

        for (check = 0 ; !gage_triggered () && check <= 10000 ; check++)
                        ;  /* Wait until board triggers or a time-out occurs.     */
        if (check > 10000)
                gage_forced_trigger_capture (gage_driver_info.trigger_source);
        for (check = 0 ; gage_busy () && check <= 10000 ; check++)
                ;           /* Wait until board is not busy or a time-out occurs.   */
        if (check > 10000)
                gage_abort_capture (gage_driver_info.trigger_source);
}



void        transfer_data (void)
{
        int16       *a_buf_16, *b_buf_16;
        a_buf_16 = (int16 *)(a_buffer);
        b_buf_16 = (int16 *)(b_buffer);
        word        sample_size;
        int32       index;

        boards_in_system = gage_get_boards_found ();
        for (j = 0; j < boards_in_system; j++) {
                gage_select_board (j + 1);
                gage_get_driver_info (&gage_driver_info);
                if !use_trigger_view_transfer {
                        gage_need_ram (TRUE);
                        if (board.opmode == GAGE_SINGLE_CHAN)  {
                        /*  Do Channel A, single channel mode.  */
                                index = trigger_address[j << 1];
                                for ( k = 0; k < POINTS; k++)
                                    a_buffer[k] = gage_mem_read_single (index = (index + 1)
                                                        % buffer_mask);
                        }else{
                        /*  Do Channel A and B, dual channel mode.  */
                                index = trigger_address[j];
                                for ( k = 0; k < POINTS; k++)
                                    a_buffer[k] = gage_mem_read_chan_a (index = (index + 1)
                                                        % buffer_mask);
```

```
                              index = trigger_address[j + 1];
                              for (k = 0; k < POINTS; k++)
                                   b_buffer[k] = gage_mem_read_chan_b (index = (index + 1)
                                                    % buffer_mask);
                         }
                         gage_need_ram (FALSE);
               }else {   /* Use trigger_view_transfer. */
                         if (gage_driver_info.mode == GAGE_SINGLE_CHAN) {
                         /* In single channel mode, the data samples will be interleaved between
                            channel A and channel B (ie. the first point will be in channel A, the
                          second in B, the third in A, etc.  */
                          gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, POINTS >> 1);
                          gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, POINTS >> 1);
                          sample_size = (gage_driver_info.board_type ==
                         (GAGE_ASSUME_CS1012 |GAGE_ASSUME_CS6012))
                              /*Uninterleave the data.     */
                          if (sample_size) {
                              for (k = POINTS - 1; m = (POINTS >> 1) -1; m > 0) {
                                   a_buf_16[k--] = b_buf_16[m];
                                   a_buf_16[k--] = a_buf_16[m--];
                              }
                              a_buf_16[1] = b_buf_16[0];
                               }else {
                                   for (k = POINTS - 1; m = (POINTS >> 1) -1; m > 0) {
                                             a_buffer[k--] = b_buffer[m];
                                             a_buffer[k--] = a_buffer[m--];
                                   }
                                   a_buffer[1] = b_buffer[0];
                               }/* Data ends up in a_buffer. */
                         }else {
                              gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, POINTS);
                              gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, POINTS);
                         }
               }
               /* Display, save or otherwise manipulate the captured data.      */
     }
}



int       write_cs_data (char *filename, int chan)  /*  chan equals GAGE_CHAN_A or
                                                        GAGE_CHAN_B.  */
{
          FILE    *fp;
          int16        start_block, blocks_transfered, res_12_bit;
          gage_driver_info_type  gage_driver_info;

          gage_get_driver_info (&gage_driver_info);
          res_12_bit = (gage_driver_info.board_type == (GAGE_ASSUME_CS1012 ||
                                                   GAGE_ASSUME_CS6012));
          if ((fp = fopen (filename, "wb")) == NULL)  /*  Open file for binary output.  */
                    return (1);

  /*If a CS6012 or CS1012 is being used buffer should be an integer array, if  a CS250, CS225, CS220*/
  /*or CSLITE is being used, buffer should be a byte array. For a combination of both, buffer should   */
  /*  be a byte array and then be cast as an integer array when dealing with the CS6012 or CS1012.
        */

        if (gage_driver_info.mode == GAGE_SINGLE_CHAN)  {
                start_block = blocks_transfered = 0;
                do{
                    gage_32k_to_buffer (buffer, 0, start_block +=
                            (blocks_transfered >> res_12_bit), &blocks_transfered);
                 if (fwrite (buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW, 1, fp) != 1)
         {
                            fclose (fp);
                            return (2);
                  }
                }while (blocks_transfered == 8);
                start_block = blocks_transfered = 0;
                do{
                    gage_32k_to_buffer (buffer, 1, start_block += (blocks_transfered >> res_12_bit),
                                    &blocks_transfered);
                     if (fwrite (buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW,
                                                                1, fp) != 1)
{
                            fclose (fp);
                            return (2);
                      }
                }while (blocks_transfered == 8);
        }else{
                start_block = blocks_transfered = 0;
                do{
                    gage_32k_to_buffer (buffer, chan - 1, start_block +=
                                    (blocks_transfered >> res_12_bit), &blocks_transfered);
                    if (fwrite (buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW,
                                                                1, fp) != 1)
{
                            fclose (fp);
                            return (2);
                     }
                }while (blocks_transfered == 8);
        }
        fclose (fp);
        return (0);
}   /*  End of write_cs_data ().  */



void do_multiple_record (int32 startpoint, int32 points)
{
        int16     j, boards_in_system;
        int32     trig, start, end, group, size;

        boards_in_system = gage_get_boards_found ();
        for (j = 1; j <= boards_in_system; j++) {
            gage_select_board (j);
```

```
                gage_get_driver_info (&gage_drver_info);
                group = 1;
                while (group == gage_calculate_mr_addresses (group,
                        gage_driver_info.board_type,
                        gage_driver_info.trigger_depth,
                        gage_driver_info.max_available_memory,
                        0, gage_driver_info.mode,
                        sample_rate_table[board.srindex].sr_calc, &trig, &start, &end));
                 {
                  size = gage_normalize_address (trig, end, gage_driver_info.max_available_memory + 1);
                   if (startpoint < 0L)
                        startpoint = 0L;
                   if (points < 0) {          /* If points is -1, calculate the best number. */
                        if (size < = GAGE_MAX_RAM_WINDOW)
                                points = size;
                        else
                                points = GAGE_MAX_RAM_WINDOW;
                   }
                   /* Constrain points to 4K because we're using  trigger_view_transfer */
                  if (points > GAGE_MAX_RAM_WINDOW)
                        points = GAGE_MAX_RAM_WINDOW;
                  if (startpoint > (size - points))
                        startpoint = (size - points);
                  gage_set_trigger_view_offset (startpoint + trig);
                  if (gage_driver_info.mode == GAGE_SINGLE_CHAN) {
                        gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, points >> 1);
                        gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, points >> 1);
                        {/* Uninterleave the data from single channel mode. */
                                register k, m;
                                for (k = points - 1; m = (points >> 1) - 1; j > 0) {
                                        a_buffer[i--] = b_buffer[j];
                                        a_buffer[i--] = b_buffer[j--];
                                }
                                a_buffer[1] = b_buffer[0];
                        }
                 }else{  /* Dual channel mode. */
                        gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, points);
                        gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, points);
                 }
                   group ++;
                }       /* end while. */
        /*  Display, save or manipulate data.               */
                do_stuff (gage_driver_info.mode, a_buffer,
                        (gage_driver_info.mode == GAGE_DUAL_CHAN) ?b_buffer :
                        NULL), trig, start, end);
        }           /* end for */
}
```

```
void	init_chan_memory_pointers (int16 it, int16 board, int16 chan, int16 operation_mode, float tbs,
				int32 *trig, int32 *start, int32 *end)
{
	gage_select_board (board);
	gage_calculate_addresses (
		chan & 1,
		operation_mode,
		tbs,
		trig,
		start,
		end);
	if (it)
		gage_get_interpolate_trigger (
			0,
			chan & 1,
			trig,
			*start,
			*end);
}	/*  End of init_chan_memory_pointers ().  */



void	init_memory_pointers (int16 board, int16 operation_mode, float tbs, int32 *trig, int32 *start,
				int32 *end)
{
	int			it = 0, i, a_board;

	if (interpolate_trigger_channel < INTERPOLATE_TRIGGER_DISABLE)  {
		int32	trig, start, end;

		gage_select_board (board);
		gage_calculate_addresses (
			interpolate_trigger_channel & 1,
			operation_mode,
			tbs,
			&trig,
			&start,
			&end);

		gage_get_interpolate_trigger (
			1,
			interpolate_trigger_channel & 1,
			&trig,
			start,
			end);
		it = 1;
	}
	for (i = 0 ; i < (boards_in_system * 2) ; i += (1 + gage_current_card->one_channel_active))  {
				a_board = ((i >> 1) + 1);
		init_chan_memory_pointers (it, a_board, i, operation_mode, tbs, &trig[i], &start[i],
				&end[i]);
	}
}	/*  End of init_memory_pointers ().  */
```

```
void    init_interpolate_trigger (int board, int trigger_source, int operation_mode, int trigger_level,
                                                                        int trigger_slope)
{
        int     auto_mode;
        if (interpolate_trigger)  {
          if (board)  {
                auto_mode = ((trigger_source == GAGE_CHAN_A)
                || ((trigger_source == GAGE_CHAN_B) &&
                (operation_mode == GAGE_DUAL_CHAN)));
                if (auto_mode)
                   gage_reset_interpolate_trigger (board, (interpolate_trigger_channel & 1) +
                        GAGE_CHAN_A, 1, trigger_level, trigger_slope);
                        /*  Enable interpolated triggering.   */
                else
                   gage_reset_interpolate_trigger (board, (interpolate_trigger_channel & 1) +
                        GAGE_CHAN_A, 0, 0, 0);
                        /*  Enable interpolated triggering.   */
           } /* end of if board. */
        }else{
           gage_reset_interpolate_trigger (0, GAGE_CHAN_A, 0, 0, 0);
                /*   Disable interpolated triggering.  */
        }
}       /*      End of init_interpolate_trigger ().    */


void low_level_capture (int boards)
{
/* Note: this is not the recommended way of capturing data, but may be used to optimize single board
systems. */
        int16   i;

        gage_select_board (1);              /*  Select MASTER.  */
        gage_abort ();                      /*  Clear MASTER operation.  */
        gage_init_clock ();                 /        *  Start MASTER clock.  */
        gage_get_data_low ();               /*  Begin data capture sequence for MASTER.  */
        for (i = 2 ; i <= boards ; i++)  {
                gage_select_board (i);      /*  Select SLAVE "i".  */
                gage_abort ();              /*  Clear SLAVE "i" operation.  */
                gage_get_data_low ();       /*  Start data capture for SLAVE "i".  */
        }
        for (i = boards ; i > 1 ; i--)  {
                gage_select_board (i);      /*  Select SLAVE "i".  */
                gage_get_data_high ();     /*  Wait for data capture acknowledge for SLAVE "i".  */
        }
        gage_select_board (1);              /*  Reselect MASTER.  */
        gage_get_data_high ();              /*  Wait for MASTER data capture acknowledge to finish
                                                    sequence.  */
        while (!gage_triggered ())
           ;
        while (gage_busy ())
           ;
        /*  Data has now been received.  */
}
```

Variable Definitions for Examples: Pascal.

This section lists the definitions, types and variables assumed to be present for the various sample routines. See the section on driver types, structures and defintions for an explanation of the user defined types.

```
const
        POINTS = 4096
        DEFAULT_INDEX = $0200
        DEFAULT_SEGMENT = $D000

var
        buffer:                         array [0..POINTS] of uInt8;
        a_buffer:                       array [0..POINTS shr 1] of uInt8;
        b_buffer:                       array [0..POINTS shr 1] of uInt8;
        buf_16:                         array [0..POINTS] of int16;
        a_buf_16:                       array [0..POINTS shr 1] of int16;
        b_buf_16:                       array [0..POINTS shr 1] of int16;
        str:                            array [0..255] of char;
        sample_offset:                  int16;
        sample_resolution:              int16;
        multiple_record_available:int16;
        use_multiple_record:            integer;
        current_board_type:             uInt16;
        current_memory_size:            int32;
        buffer_mask:                    int32; { equal to max_available_memory }
        channel_ram_size:               int32;
        use_trigger_view_transfer:integer;
```

The following definitions, types and variables are used for some of the examples which have their roots in the PASCAL sample programs. The additional "types" are defined and initialized in the sample programs or their support files on the distribution diskette..

```
const
        INTERPOLATE_TRIGGER_DISABLE = 32
type
        address = array[0..GAGE_B_L_MAX_CARDS * 2 - 1] of int32;
        gage_b_l_array:  array [0..GAGE_B_L_BUFFER_SIZE] of uInt16;

        srtype = record
                rate:           int16;
                mult:           int16;
                sr_flag:        int32;
                sr_calc:        single;
                text:           pchar;
        end;

        boarddef = record
                opmode:         int16;
                srindex:        int16;
                range_a:        int16;
                couple_a:       int16;
```

```
            range_b:            int16;
            couple_b:           int16;
            source:             int16;
            slope:              int16;
            depth:              int32;
            level:              int16;
            range_e:            int16;
            couple_e:           int16;
        end;

{ Initialization of the board structure. The board structure is of type boarddef.    }

        board.opmode      :=  GAGE_DUAL_CHAN:
        board.srindex     :=  21:
        board.range_a     :=  GAGE_PM_1_V:
        board.couple_a    :=  GAGE_DC;
        board.range_b     :=  GAGE_PM_1_V;
        board.couple_b    :=  GAGE_DC;
        board.source      :=  GAGE_CHAN_A;
        board.slope       :=  GAGE_POSITIVE;
        board.depth       :=  GAGE_POST_8K;
        board.level       :=  128;
        board.range_e     :=  GAGE_PM_1_V;
        board.couple_e    :=  GAGE_DC;

        interpolate_trigger:        integer:
        gage_driver_info:gage_driver_info_type;
        gage_board_location:        gage_b_l_array;
        board_loc_file:             array [0..260] of char;
        board:                      boarddef;
        srtable:                    array[0..40] of srtype;
        trigger_address:            address;
        starting_address:           address;
        ending_address:             address;
```

Sample Routines: Pascal.

```
function init_driver_hardware: integer;

{          Initializing the CompuScope driver and hardware.     }
var
           boards_in_system:                   int16;
           segment, index, board_type, status: uInt16;
begin
           gage_get_config_filename (board_loc_file);
           if (gage_read_config_file (board_loc_file, gage_board_location) < 0) then begin
               writeln (‘Errors in GAGESCOP.INC, defaults used’);
               gage_set_records (gage_board_lcation,0,DEFAULT_SEGMENT,DEFAULT_INDEX,0,0):
               for j := 1 to GAGE_B_L_MAX_CARDS - 1 do
                      gage_set_records (gage_board_location, j, 0, 0, 0, 0);
           end;
           gage_driver_initialize (gage_board_location, GAGE_MEMORY_SIZE_TEST);
           boards_in_system := gage_driver_initialize (gage_board_location,
                                             GAGE_MEMORY_SIZE_TEST)
           if boards_in_system = 0 then begin
               gage_get_records (gage_board_location, 0, segment, index, board_type, status);
               writeln (‘No CompuScope boards found, Error code =  ‘, status);
               exit
           end
           else if boards_in_system < 0 then begin
               boards_in_system := -boards_in_system;
               writeln (‘Not all CompuScope boards found.’);
           end;
           gage_select_board (1);
           gage_get_driver_info (gage_driver_info);
           current_board_type := B_T_DOUBLE_UP_BITS (gage_driver_info.board_type);
           { B_T_DOUBLE_UP_BITS is a function found in START.PAS }
           current_memory_size :=  gage_driver_info.max_memory;
           sample_offset := gage_driver_info.sample_offset;
           sample_resolution = gage_driver_info.sample_resolution;
           { Detect if multiple recording option is present on installed CompuScope hardware. }
           multiple_record_available := gage_detect_multiple_record;
           if multiple_record_available = 0 then
                      use_multiple_record := TRUE;
           else
                      use_multiple_record := FALSE;
           init_driver_hardware := boards_in_system;
end;



procedure prepare_for_capture (boards_in_system : integer; var board : boarddef);
{
           Preparing the CompuScope card for data capture.
           Initialize the hardware with the new values.
}
```

```
var
          j, boards_in_system          : int16;
begin
          boards_in_system := gage_get_boards_found;
          for j := 1 to boards_in_system do begin
             gage_select_board (j);
             gage_multiple_record (0); {Set to 1 to enable multiple record if available. }
          end
          gage_select_board (1);      {Master. }
          gage_get_driver_info (gage_driver_info);
          gage_capture_mode (board.opmode, srtable[board.srindex].rate, srtable[board.srindex].mult);
          gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board.couple_a,
                                        board.range_a);
          if board.opmode = GAGE_SINGLE_CHAN then
             gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE, board.couple_a,
                                board.range_a)
          else
             gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE, board.couple_b,
                                board.range_b);
          for i := 2 to boards_in_system do
          begin
            gage_select_board (i);    {  Slaves.  }
            if (((gage_driver_info.board_type = GAGE_ASSUME_CS1012) or
                   (gage_driver_info_type = GAGE_ASSUME_CS6012)) and
                   (board.srindex <> 40)) then
                   { External clock not in use with 1012 or 6012. }
                   gage_capture_mode (board.opmode, srtable[board.srindex].rate,
                                        srtable[board.srindex].mult)
            else
                   gage_capture_mode (board.opmode, 0, GAGE_EXT_CLK);
            gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board.couple_a,
                                        board.range_a);
            if board.opmode = GAGE_SINGLE_CHAN then
                   gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                            board.couple_a, board.range_a)
            else
                   gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                            board.couple_b, board.range_b);
            gage_trigger_control (GAGE_SOFTWARE, GAGE_DC, GAGE_PM_1_V,
                            GAGE_POSITIVE, $7f, board.depth);
          end;
          gage_select_board (1);      {  Master.  }
          gage_trigger_control (board.source, board.couple_e, board.range_e, board.slope, board.level,
                                                  board.depth);
          gage_get_driver_info (gage_driver_info);
          channel_ram_size := gage_driver_info.max_available_memory;
          delay (100);        {  Allow relays to settle.  }
end;      {          End of prepare_for_capture.          }
```

```
procedure InitMemPtrs;
var
        i, channel:         int16;
begin
        if USE_INTERPOLATED_TRIGGER_CODE then
        { See the sample programs on the distribution diskette to see how to use the support routines
                        for interpolated trigger.}
                init_memory_pointers (1, board.opmode, srtable[board.srindex].sr_calc,
                                trigger_address, starting_address, ending_address);
        else begin
                i := 0;
                while i < (boards_in_system * 2) do begin
                        gage_select_board ((i shr 1) + 1);
                        channel := (i AND 1) + GAGE_CHAN_A;
        { When not using interpolated trigger, use this line. }
                        gage_calculate_addresses(channel, board.opmode,
                        srtable[board.srindex].sr_calc, trigger_address[i],
                                starting_address[i], ending_address[i]);
                        if board.opmode = GAGE_SINGLE_CHAN then
                                i := i + 2
                        else
                                i := i + 1;
                end;
        end;
end;


procedure AcquireCheck:
vars
        check, auto_trigger:        int16;
begin
        gage_get_driver_info (gage_driver_info);
        if gage_driver_info.trigger_source = GAGE_SOFTWARE then
                auto_trigger := 1
        else
                auto_trigger := 0;
        gage_start_capture (auto_trigger);
        check := 0;
        while (gage_triggered = 0) and (check <= 10000) do
                        check := check + 1; { Wait until board triggers or a time-out occurs. }
        if check > 10000 then
                gage_forced_trigger_capture (gage_driver_info.trigger_source);
        check := 0;
        while (gage_busy <> 0) and (check <= 10000) do
                check := check + 1;         { Wait until board is not busy or a time-out occurs. }
        if check > 10000 then
                gage_abort_capture (gage_driver_info.trigger_source);
end;
```

```
procedure transfer_data:
var
        j, k, m:   sample_size: int16;
        index:     int32;
begin
        boards_in_system := gage_get_boards_found:
        for j := 1 to boards_in_system do begin
                gage_select_board (j);
                gage_get_driver_info (gage_driver_info);
                if not use_trigger_view_transfer then begin
                   gage_need_ram (TRUE);
                   if gage_driver_info.mode = GAGE_SINGLE_CHAN then begin
                        { Do channel A, single channel mode. }
                        index := trigger_address[((j -1) shl 1)];
                        k := 0;
                        while k < POINTS do begin
                           buffer[k] := gage_mem_read_single (index mod buffer_mask);
                           index := index + 1;
                           k := k + 1;
                        end;
                   end
                   else begin
                        { Do channel A and B, dual channel mode. }
                        index := trigger_address[j];
                        while k < POINTS do begin
                           a_buffer[k] = gage_mem_read_chan_a (index mod buffer_mask);
                           index := index + 1;
                           k := k + 1;
                        end;
                        index := trigger_address[j + 1];
                        while k < POINTS do begin
                           b_buffer[k] = gage_mem_read_chan_b (index mod buffer_mask);
                           index := index + 1;
                           k := k + 1;
                        end;
                   end;
                   gage_need_ram (FALSE);
                end
                else begin          { Use trigger_view_transfer. }
                   if gage_driver_info.mode = GAGE_SINGLE_CHAN then begin
                        { In single channel mode, the data samples will be interleaved between
                channel  A and channel B (ie. the first point will be in channel A, the
                second in           B, the third in A, etc.}
                        gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, POINTS >> 1);
                        gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, POINTS >> 1);
                        if (gage_driver_info.board_type = GAGE_ASSUME_CS1012) OR
                        (gage_driver_info.board_type = GAGE_ASSUME_CS6012) then
                                sample_size := 1
                        else
                                sample_size := 0;
                        k := POINTS - 1;
                        m := (POINTS shr 1) - 1;
                        if sample_size = 1 then begin  { 1012 or 6012 needs an integer buffer }
                           while m > 0 do begin { Uninterleave the data. }
```

```
                                        a_buf_16[k] = b_buf_16[m];
                                        k := k - 1;
                                        a_buf_16[k] := a_buf_16[m];
                                        k := k - 1;
                                        m := m - 1
                                    end
                                  a_buf_16[1] := b_buf_16[0];
                                  else begin
                                     while m > 0 do begin { Uninterleave the data. }
                                        a_buffer[k] = b_buffer[m];
                                        k := k - 1;
                                        a_buffer[k] := a_buffer[m];
                                        k := k - 1;
                                        m := m - 1
                                     end
                                     a_buffer[1] := b_buffer[0];
                                  end
                              end
                              else begin
                                  gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, POINTS);
                                  gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, POINTS);
                              end;
                            end;
      { Display, save or otherwise manipulate the captured data. }
      end;  { for loop. }
end;



function write_cs_data (var filename: pchar; chan: integer);  { chan equals GAGE_CHAN_A or }
                                                               {GAGE_CHAN_B        }
var
      file:       fp;
      start_blocks, blocks_transfered,  res_12_bit:    int16;
      gage_driver_info:  gage_driver_info_type;
begin
      gage_get_driver_info (gage_driver_info);
      if (gage_driver_info.board_type = GAGE_ASSUME_CS1012) OR
                  (gage_driver_info.board_type = GAGE_ASSUME_CS6012) then
             res_12_bit := 1
      else
            res_12_bit := 0;
      fp := _lopen (filename, ReadWrite);
      if fp = nil then
            exit;

  {   If only a CS6012 or CS1012 is being used buffer should be an integer array, if a CS250, CS225,
      CS220 or  CSLITE is being used, buffer should be a byte array. For a combination of both,
      buffer should  be a byte array and then be cast as an integer array when dealing with the
      CS6012 or CS1012.}

      if gage_driver_info.mode = GAGE_SINGLE_CHAN then begin
            start_block := 0;
            blocks_transfered := 0;
```

```
                    repeat
                       gage_32k_to_buffer (buffer, 0, start_block, @blocks_transfered);
                       start_block = start_block + (blocks_transfered shr  res_12_bit);
                       if _lwrite (fp, buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW) = -1
                                                       then  begin
                                 _lclose (fp);
                                 exit;
                        end;
                    until blocks_transfered <> 8;
                    start_block := 0;
                    blocks_transfered := 0;
                    repeat
                       gage_32k_to_buffer (buffer, 1, start_block, @blocks_transfered);
                       start_block = start_block + (blocks_transfered shr res_12_bit);
                       if _lwrite (fp, buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW) = -1
                                                                                  then
                       begin
                                  _lclose (fp);
                                 exit;
                       end;
                    until blocks_transfered <> 8;
              end
              else begin
                    start_blocks := 0;
                    blocks_transfered := 0;
                    repeat
                       gage_32k_to_buffer (buffer, chan - 1, start_block, @blocks_transfered);
                       start_block = start_block + (blocks_transfered shr res_12_bit);
                       if _lwrite (fp, buffer, blocks_transfered * GAGE_MAX_RAM_WINDOW) = -1
                                                                           then
                       begin
                                  _lclose (fp);
                                 exit;
                       end;
                    until blocks_transfered <> 8;
              end;
              fclose (fp);
end;



procedure do_multiple_record (startpoint: int32; points: int32);
var
              j, group:                      int16;
              trig, start, endaddr:          int32;
begin
              boards_in_system := gage_get_boards_found;
              for j := 1 to boards_in_system do begin
                   gage_select_board (j);
                   gage_get_driver_info (gage_driver_info);
                   group := 1;
```

```
            while (group = gage_calculate_mr_addresses (group,
                  gage_driver_info.board_type,
                  gage_driver_info.trigger_depth;
                           gage_driver_info.max_available_memory,
                           0, gage_driver_info.mode,
                           srtable[board.srindex].sr_calc,
                           trig, start, endaddr)) do
            begin
                  size := gage_normalize_address (trig, endaddr,
                                    gage_driver_info.max_available_memory + 1);
                  if startpoint < 0 then
                           startpoint := 0;
                  if points < 0 then begin { If points is -1, calculate the best number. }
                           if size <= GAGE_MAX_RAM_WINDOW then
                                    points := size
                           else
                                    points := GAGE_MAX_RAM_WINDOW;
                  end;
                  { Constrain points to 4K because we're using trigger_view_transfer. }
                  if points > GAGE_MAX_RAM_WINDOW then
                           points = GAGE_MAX_RAM_WINDOW;
                  if startpoint > (size - points) then
                           startpoint = (size - points);
                  gage_set_trigger_view_offset (startpoint + trig);
                  if gage_driver_info.mode = GAGE_SINGLE_CHAN then begin
                           gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, points >> 1;
                           gage_trigger_view_transfer  (GAGE_CHAN_B, b_buffer, points >> 1);
                           k := points - 1;
                           m := (points shr 1) - 1;
                           while m > 0 do begin { Uninterleave the data. }
                                    a_buffer[k] = b_buffer[m];
                                    k := k - 1;
                                    a_buffer[k] := a_buffer[m];
                                    k := k - 1;
                                    m := m - 1
                           end
                           a_buffer[1] := b_buffer[0];
                  end
                  else begin
                           gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, points);
                           gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, points);
                  end
                  group := group + 1;
            end; {while}
            { Display, save or manipulate data. }
            do_stuff (board.opmode, a_buffer, b_buffer, trig, start, end);
         end;  {for}
end;
```

```
procedure init_chan_memory_pointers (board, chan, operation_mode: int16; tbs: single;
                                     var trig:  int32; start: int32; var endaddr: int32);
begin
        gage_select_board (board);
        gage_calculate_addresses (chan AND 1, operation_mode, tbs, trig, start, endaddr);
        gage_get_interpolate_trigger (0, chan AND 1, trig, start, endaddr);
end;




procedure init_memory_pointers (board, operation_mode: int16; tbs: single; var trig: address;
                                                        var start: address; var endaddr: address);
var
        i, a_board:                     integer;
        trigr, startr, endaddrr:        int32;
begin
        if interpolate_trigger_channel < INTERPOLATE_TRIGGER_DISABLE then begin
           gage_select_board (board);
           gage_calculate_addresses (interpolate_trigger_channel AND 1, operation_mode, tbs, trigr,
                                                              startr, endaddrr);
           gage_get_interpolate_trigger (1, interpolate_trigger_channel AND 1, trigr, startr,
                                                        endaddrr);
        end;
        boards_in_system := gage_get_boards_found;
        i := 0;
        while i < boards_in_system * 2 do begin
           a_board := (i shr 1) + 1;
           init_chan_memory_pointers (a_board, i, operation_mode, tbs, trig[i], start[i], endaddr[i]);
           if operation_mode = GAGE_DUAL_CHAN then
                i := i + 1
           else
                i := i + 2;
           end;
        end;
end;




procedure init_interpolate_trigger (board, trigger_source, operation_mode, trigger_level,
                                                        trigger_slope: int16);
var
        auto_mode: boolean;
begin
    if interpolate_trigger = 1 then begin
        if board <> 0 then begin
           if ((trigger_source = GAGE_CHAN_A) OR ((trigger_source = GAGE_CHAN_B) AND
                                        (operation_mode = GAGE_DUAL_CHAN))) then
                auto_mode := true
           else
                auto_mode := false;
           if auto_mode then
```

```
                gage_reset_interpolate_trigger (board, (interpolate_trigger_channel AND 1) +
                GAGE_CHAN_A, 1, trigger_level, trigger_slope)
                        {Enable interpolated triggering.}
           else
                gage_reset_interpolate_trigger (board, (interpolate_trigger_channel AND 1) +
                GAGE_CHAN_A, 0, 0, 0) {Enable interpolated triggering.}
        end
    end
    else
        gage_reset_interpolate_trigger (0, GAGE_CHAN_A, 0, 0, 0); {Disable interpolated trigger.}
end;



procedure low_level_capture (boards: integer)
{ Note: this is not the recommended way to capture data.  This method may be used to  }
{ optimize single board systems. }
var
        i:          int16;
begin
        gage_select_board (1);              { Select MASTER }
        gage_abort;                         { Clear MASTER operation }
        gage_init_clock;                    { Start MASTER clock }
        gage_get_data_low:                  { Begin data capture sequence for MASTER }
        for i:= 2 to boards - 1 do
        begin
          gage_select_board (i);            { Select SLAVE }
          gage_abort;                       { Clear SLAVE "i" operation }
          gage_get_data_low;                { Start data capture for SLAVE "i" }
        end
        i := boards;
        while i > 1 do
        begin
          gage_select_board (i);            { Select SLAVE "i" }
          gage_get_data_high;               { Wait for data capture acknowledge for SLAVE "i" }
          i := i - 1;
        end;
        gage_select_board (1);              { Re-select MASTER }
        gage_get_data_high;                 { Wait for MASTER data capture acknowledge to finish
                                                    sequence }
        while gage_triggered = 0 do;
        while gage_busy <> 0 do;            { Data has now been received }
end;
```

Variable Definitions for Examples: Basic.

This section lists the definitions, types and variables assumed to be present for the various sample routines. Note that there are syntax differences between Visual and Quick Basic, and differences in how strings are passed to the drivers, most notably in **gage_get_config_filename** and **gage_read_config_file**.
Check the sample programs on the distribution diskette for further information.

```
        Global Const POINTS = 4096
        Global Const DEFAULT_SEGMENT = &HD000
        Global Const DEFAULT_INDEX = &H0200


        Global address                      As Long
        Global starting_address             As Long
        Global trigger_address              As Long
        Global ending_address               As Long


        Global ret                          As Integer
        Global a_buffer (POINTS)            As Integer
        Global b_buffer (POINTS)            As Integer
        Global str                          As String * 250
        Global multiple_record_available    As Integer
        Global use_multiple_record          As Integer
        Global buffer_mask                  As Long ‘equal to max_available_memory
        channel_ram_size            As Long
        use_trigger_view_transfer           As Integer
```

The following definitions are used for some of the examples which have their roots in the BASIC sample programs. The additional "types" and variables are defined and initialized in the sample programs or their support files on the distribution diskette.

```
const INTERPOLATE_TRIGGER_DISABLE = 32

Type Srtype
        rate As Integer
        mult As Integer
        sr_flag As Long
        sr_calc As Single
        srtext As String
End Type


Type boarddef
        opmode As Integer
        srindex As Integer
        range_a As Integer
        couple_a As Integer
        range_b As Integer
        couple_b As Integer
        source As Integer
        slope As Integer
        depth As Integer
        level As Integer
        range_e As Integer
        couple_e As Integer
End Type


' Initialization of the board structure. The board structure is of type boarddef.

        board.opmode     =  GAGE_DUAL_CHAN
        board.srindex    =  21
        board.range_a    =  GAGE_PM_1_V
        board.couple_a   =  GAGE_DC
        board.range_b    =  GAGE_PM_1_V
        board.couple_b   =  GAGE_DC
        board.source     =  GAGE_CHAN_A
        board.slope      =  GAGE_POSITIVE
        board.depth      =  GAGE_POST_8K
        board.level      =  128
        board.range_e    =  GAGE_PM_1_V
        board.couple_e   =  GAGE_DC

        Global driver_info                  As gage_driver_info_type
        Global gage_driver_info             As gage_driver_info_type
        Global board_loc_file               As String * 260
        Global board                        As boarddef;
        Global srtable (40)                 As srtype
        Global trigger_address(32)          As Long
        Global starting_address(32)         As Long
        Global ending_address(32)           As Long
        Global interpolate_trigger  As Integer
        Global interpolate_trigger_channel As Integer
        Global gage_board_location(GAGE_B_L_BUFFER_SIZE)     As Integer
        Global USE_INTERPOLATED_TRIGGER_CODE    As Integer
```

Sample Routines: Basic.

```
Function init_driver_hardware%
‘ Initializing the CompuScope driver and hardware

DIM boards_in_system As Integer
DIM segment As Integer
DIM index As Integer
DIM board_type As Integer
DIM status As Integer
Dim sample_offset As Integer
Dim sample_resolution As Integer

ret = gage_get_config_filename(board_loc_file)
ret = gage_read_config_file(board_loc_file, gage_board_location(0))
IF ret < 0 THEN
        PRINT “Errors in GAGESCOP.INC, defaults used”
        temp = gage_set_records (gage_board_location(0), 0, DEFAULT_SEGMENT,
                                                DEFAULT_INDEX, 0, 0)
        FOR j = 1 TO GAGE_B_L_MAX_CARDS - 1
                temp = gage_set_records (gage_board_location(0), j, 0, 0, 0, 0)
        NEXT
END IF
temp = gage_driver_initialize(gage_board_location(0), GAGE_MEMORY_SIZE_TEST)
boards_in_system = gage_driver_initialize(gage_board_location(0), GAGE_MEMORY_SIZE_TEST)
IF boards_in_system = 0 THEN
        temp = gage_get_records (gage_board_location(0), 0, segment, index, board_type, status)
        PRINT “No boards found, error = “; status
        END             ' If no board is found, end program.
ELSEIF boards_in_system < 0 THEN
        boards_in_system = -boards_in_system
        PRINT “Not all CompuScope boards found”
END IF
temp = gage_select_board (1)
CALL gage_get_driver_info (gage_driver_info)
current_board_type = gage_driver_info.board_type
current_memory_size = gage_driver_info.max_memory
sample_offset = gage_driver_info.sample_offset
sample_resolution = gage_driver_info.sample_resolution
‘       Detect if multiple recording option is available on installed CompuScope hardware
multiple_record_available = gage_detect_multiple_record ()
init_driver_hardware = boards_in_system
END FUNCTION



SUB prepare_for_capture (board AS boarddef)

boards_in_system = gage_get_boards_found ()
FOR j = 1 TO boards_in_system
        temp% = gage_select_board (j)
```

```
            CALL gage_multiple_record (0)     ‘ Set to 1 to enable multiple record if available.
NEXT
temp% = gage_select_board (1)        'Master.
CALL gage_get_driver_info (gage_driver_info)
temp% = gage_capture_mode (board.opmode, srtable(board.srindex).rate, srtable(board.srindex).mult)
temp% = gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board.couple_a,
                                                        board.range_a)
IF board.opmode = GAGE_SINGLE_CHAN THEN
            temp% = gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE, board.couple_a,
                                                        board.range_a)
ELSE
            temp% = gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE, board.couple_b,
                                                        board.range_b)
END IF
FOR j = 2 TO boards_in_system
            temp% = gage_select_board (j)   ' Slaves.
            IF (((gage_driver_info.board_type = GAGE_ASSUME_CS1012) OR
            (gage_driver_info.board_type = GAGE_ASSUME_CS6012)) AND
                                                                    board.srindex <> 40) THEN
                        ‘ External clock not in use for 1012 or 6012.
                        temp% = gage_capture_mode (board.opmode, srtable(board.srindex).rate,
                                                srtable(board.srindex).mult)
            ELSE
                        temp% = gage_capture_mode (board.opmode, 0, GAGE_EXT_CLK)
            END IF
            temp% = gage_input_control (GAGE_CHAN_A, GAGE_INPUT_ENABLE, board.couple_a,
                                                        board.range_a)
            IF board.opmode = GAGE_SINGLE_CHAN THEN
                        temp% = gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                                board.couple_a, board.range_a)
            ELSE
                        temp% = gage_input_control (GAGE_CHAN_B, GAGE_INPUT_ENABLE,
                                                board.couple_b, board.range_b)
            END IF
            temp% = gage_trigger_control (GAGE_SOFTWARE, GAGE_DC, GAGE_PM_1_V,
                                                GAGE_POSITIVE, 128, board.depth)
NEXT
temp% = gage_select_board (1)  ' Master
temp% = gage_trigger_control (board.source, board.couple_e, board.range_e, board.slope, board.level,
                                                                            board.depth)
CALL gage_get_driver_info (gage_driver_info)
channel_ram_size& = gage_driver_info.max_available_memory
SLEEP 1     '  Allow relays to settle.  This command is available only under Quick Basic.
END SUB



SUB InitMemPtrs ()
DIM  dummy As Integer
DIM i As Integer
DIM j As Integer

IF USE_INTERPOLATED_TRIGGER_CODE = 1 THEN
```

```
        Call init_memory_pointers (1, board.opmode, srtable(board.srindex).sr_calc, trigger_address(),
                                                        starting_address(), ending_address())
ELSE
        ' Always initialize Channel A
        j = 0
        WHILE j <= (boards_in_system * 2) - 1
                dummy = gage_select_board((j / 2) + 1)
                 k = j And 1
                channel = k + GAGE_CHAN_A
                Call gage_calculate_addresses(channel, board.opmode,
                        srtable(board.srindex).sr_calc, trigger_address(j), starting_address(j),
                        ending_address(j))
                IF board_info.opmode = GAGE_SINGLE_CHAN THEN
                        j = j + 2
                ELSE
                         j = j + 1
                END IF
        WEND
END IF
END SUB



SUB AcquireCheck ()
DIM check As Integer
DIM auto_trigger As Integer

CALL gage_get_driver_info (gage_driver_info)
IF gage_driver_info.trigger_source = GAGE_SOFTWARE THEN
        auto_trigger = 1
ELSE
        auto_trigger = 0
END IF
CALL gage_start_capture (auto_trigger)
check = 0
done = FALSE
WHILE done = FALSE
        IF gage_triggered THEN
                done = TRUE
        END IF
        IF check > 10000 THEN  '  Check for trigger time out
                done = TRUE
                CALL gage_forced_trigger_capture (gage_driver_info.trigger_source)
        END IF
        check = check + 1
WEND
done = FALSE
check = 0
WHILE done = FALSE
        IF NOT gage_busy THEN
                done = TRUE
        END IF
        IF check > 30000 THEN  '  Check for capture time out
                done = TRUE
                CALL gage_abort_capture (gage_driver_info.trigger_source)
        END IF
        check = check + 1
WEND
END SUB



SUB transfer_data ()
DIM j As Integer
DIM k As Integer
DIM m As Integer
DIM index As Long

boards_in_system =  gage_get_boards_found
FOR j = 1 to boards_in_system
        temp% = gage_select_board (j)
        CALL gage_get_driver_info (gage_driver_info)
        IF NOT use_trigger_view_transfer THEN
           CALL gage_need_ram (1)
           IF gage_driver_info.mode = GAGE_SINGLE_CHAN THEN
                ‘ Do channel A, single channel mode.
                index = trigger_address(((j - 1) * 2))
                k = 0
                WHILE k < POINTS
                   buffer(k) = gage_mem_read_single (index MOD buffer_mask)
                   index = index + 1
                   k = k + 1
                WEND
           ELSE
                ‘ Do channel A and B, dual channel mode
                index = trigger_address(j)
                WHILE k < POINTS
                   a_buffer(k) = gage_mem_read_chan_a (index MOD buffer_mask)
                   index = index + 1
                   k = k + 1
                WEND
                index = trigger_address (j + 1)
                WHILE k < POINTS
                   b_buffer(k) = gage_mem_read_chan_b (index MOD buffer_mask)
                   index = index + 1
                   k = k + 1
                WEND
           END IF
        ELSE    ‘ Use trigger_view_transfer
           IF gage_driver_info.mode = GAGE_SINGLE_CHAN THEN
        ‘ In single channel mode, the data will be interleaved between channel A and channel B
        ‘ (ie. the first point will be in channel A, the second in channel B, the third in A, etc)
                CALL gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer (0), POINTS \ 2)
                CALL gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer (0), POINTS \ 2)
                WHILE m > 0     ‘ Uninterleave the data.
                        a_buffer(k) = b_buffer(m)
                        k = k - 1
                        a_buffer(k) = a_buffer(m)
                        k = k - 1
                        m = m - 1
                WEND
                a_buffer (1) = b_buffer (0)
            ELSE
                CALL gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer, POINTS)
                CALL gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer, POINTS)
            END IF
        END IF
         ‘Display, save or otherwise manipulate the data.  If the board is a 12 bit board (CS1012 or
        ‘CS6012), the data can be used as is.  If the board is an 8 bit board, the Basic buffers are
        ‘integer buffers and the data has to be further manipulated.  The following excerpt will extract
        ‘the individual bytes from each integer in the original buffer and place them into a new buffer.
        ‘j = 0
        ‘FOR i = 0 TO (POINTS \ 2) - 1
        ‘       temp = CLng(65535) And CLng(buffer(i))
         ‘      temp_buffer(j) = temp And 255
        ‘       temp_buffer(j + 1) = (temp \ 256) And 255
        ‘       j = j + 2
        ‘NEXT
        ‘ If you are using Quick Basic, you can used the VARSEG and VARPTR to get the address of
        ‘ the buffer and PEEK to get the byte values.
        ‘ addr = VARPTR (buffer(0))
        ‘ FOR j = 0 TO POINTS
        ‘       newbuffer(j) = PEEK (addr + j)
NEXT



FUNCTION cs_write_data (filename As String, chan As Integer)
  ' chan refers to either GAGE_CHAN_A or GAGE_CHAN_B
Dim start_block As Integer
Dim blocks_transfered As Integer
Dim fnum As Integer
Dim res_12_bit As Integer

CALL gage_get_driver_info (driver_info)
IF  (driver_info.board_type = GAGE_ASSUME_CS1012) OR (driver_info.board_type =
                                                  GAGE_ASSUME_CS6012) THEN
        res_12_bit = 2
ELSE
        res_12_bit = 1
END IF
fnum = FreeFile
OPEN filename FOR Binary As fnum
IF driver_info.mode = GAGE_SINGLE_CHAN THEN
        start_block = 0
        blocks_transfered = 0
        DO
          CALL gage_32k_to_buffer (buffer(0), 0, start_block, blocks_transfered)
          PUT #fnum, ,  buffer
          ' User should put On Error statement here to trap file errors
          start_block = start_block + (blocks_transfered \ res_12_bit)
        LOOP UNTIL blocks_transfered <> 8
        start_blocks = 0
        blocks_transfered = 0
        DO
          CALL gage_32k_to_buffer (buffer(0), 1, start_block, blocks_transfered)
          PUT #fnum, , buffer
          ‘ User should put On Error statement here to trap file errors
          start_block = start_block + (blocks_transfered \ res_12_bit)
        LOOP UNTIL blocks_transfered <> 8
ELSE
        start_block = 0
        blocks_transfered = 0
        DO
          CALL gage_32k_to_buffer (buffer(0), chan - 1, start_block, blocks_transfered)
          PUT #fnum, , buffer
          ' User should put On Error statement here to trap file errors
          start_block = start_block + (blocks_transfered \ res_12_bit)
        LOOP UNTIL blocks_transfered <> 8
END IF
CLOSE
END FUNCTION
```

Note: Because Basic has no variables of type byte, the buffer must be an integer array. This is no problem when dealing with the CS6012 or CS1012, which has a sample size of 2 bytes (12 bits sign extended to 16 bits). With the CS250, CS225, CS220 and CSLITE, which have a sample size of 1 byte, the high and low bytes of the integer must be extracted. This can be done as follows:

```
        ‘j = 0
        ‘FOR i = 0 TO (POINTS \ 2) - 1
        ‘       temp = CLng(65535) And CLng(buffer(i))
        ‘       temp_buffer(j) = temp And 255
        ‘       temp_buffer(j + 1) = (temp \ 256) And 255
        ‘       j = j + 2
        ‘NEXT
```

If you are using Quick Basic, you can also use the VARSEG and VARPTR functions to get the address of the buffer. You can then access the individual bytes using the PEEK function.

```
SUB do_multiple_record (ByVal startpoint As Long, ByVal points As Long)
DIM j As Integer
DIM trig As Long
DIM startaddr As Long
DIM endaddr As Long
DIM group As Integer

boards_in_system = gage_get_boards_found
FOR j = 1 TO boards_in_system
        temp% = gage_select_board (j)
        CALL gage_get_driver_info (gage_driver_info)
        group = 1
        WHILE (group = gage_calculate_mr_addresses (group,
                                    gage_driver_info.board_type,
                                    gage_driver_info.trigger_depth,
                                    gage_driver_info.max_available_memory, 0,
                                    gage_driver_info.mode, srtable(board.srindex).sr_calc, trig,
                                    startaddr, endaddr))
                    size = gage_normalize_address (trig, endaddr,
                                    gage_driver_info.max_available_memory + 1)
                    IF startpoint < 0 THEN
                            startpoint = 0
                    END IF
                    IF points < 0 THEN
                            IF size <= GAGE_MAX_RAM_WINDOW THEN
                                    points = size
                            ELSE
                                    points = GAGE_MAX_RAM_WINDOW
                            END IF
                    END IF
                    ‘ Constrain points to 4K because we’re using trigger_view_transfer
                    IF points > GAGE_MAX_RAM_WINDOW THEN
                            points = GAGE_MAX_RAM_WINDOW
                    END IF
                    IF startpoint > (size - points) THEN
                            startpoint = (size - points)
                    END IF
                    CALL gage_set_trigger_view_offset (startpoint + trig)
                    IF driver_info.mode = GAGE_SINGLE_CHAN THEN
                       CALL gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer(0), points \ 2)
                       CALL gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer(0), points \ 2)
                       k = points - 1
                       m = (points \ 2) - 1
                       WHILE m > 0              ‘ Uninterleave the data.
                            a_buffer(k) = b_buffer(m)
                            k = k -1
                            a_buffer (k) = a_buffer(m)
                            k = k - 1
                            m = m -1
                       WEND
                       a_buffer(1) = b_buffer(0)
                    ELSE
                       CALL gage_trigger_view_transfer (GAGE_CHAN_A, a_buffer(0), points)
                       CALL gage_trigger_view_transfer (GAGE_CHAN_B, b_buffer(0), points)
                    END IF
                    group = group + 1
        WEND
‘ Display , save or manipulate data. See previous routines for extracting the bytes from an integer
‘ buffer in Basic.
        do_stuff (driver_info.mode, a_buffer, b_buffer, trig, start, endaddr)
NEXT
END SUB
```

```
SUB init_chan_memory_pointers (ByVal board As Integer, ByVal chan As Integer, ByVal
                              operation_mode As Integer, ByVal tbs As Single, trig As Long,
                              start As Long, endaddr As Long)

DIM dummy As Integer

dummy = gage_select_board (board)
CALL gage_calculate_addresses (chan And 1, operation_mode, tbs, trig, start, endaddr)
CALL gage_get_interpolate_trigger(0, chan And 1, trig, start, endaddr)

END SUB



SUB init_memory_pointer (ByVal board As Integer, ByVal operation_mode As Integer,
                              ByVal tbs As Single, trig() As Long, start() As Long, endaddr() As Long)

DIM i As Integer, a_board As Integer, dummy As Integer, boards_in_system As Integer
DIM temptrig As Long
DIM tempstart As Long
DIM tempend As Long

IF interpolate_trigger_channel < INTERPOLATE_TRIGGER_DISABLE THEN
          dummy = gage_select_board (board)
          CALL gage_calculate_addresses(interpolate_trigger_channel And 1, operation_mode, tbs,
                                                  temptrig, tempstart, tempend)
          CALL gage_get_interpolate_trigger(1, interpolate_trigger_channel And 1, temptrig,
                                        tempstart, tempend)
END IF
boards_in_system = gage_get_boards_found ()
i = 0
WHILE i < (boards_in_system * 2)
          a_board = (i \ 2) + 1
          CALL init_chan_memory_pointers(a_board, i, operation_mode, tbs, trig(i), start(i), endaddr(i))
          IF operation_mode = GAGE_DUAL_CHAN THEN
                    i = i + 1
          ELSE
                    i = i + 2
          END IF
WEND
END SUB


SUB init_interpolate_trigger (ByVal board As Integer, ByVal trigger_source As Integer, ByVal
          operation_mode As Integer, ByVal trigger_level As Integer, ByVal trigger_slope As Integer)
DIM auto_mode As Integer

IF interpolate_trigger = 1 THEN
     IF board <> 0 THEN
          IF ((trigger_source = GAGE_CHAN_A) OR ((trigger_source = GAGE_CHAN_B) AND
                              (operation_mode = GAGE_DUAL_CHAN))) THEN
             auto_mode = TRUE
          ELSE
```

```
               auto_mode = FALSE
            END IF
            IF auto_mode THEN   'Enable interpolated triggering.
               CALL gage_reset_interpolate_trigger(board, (interpolate_trigger_channel And 1) +
                                            GAGE_CHAN_A, 1, trigger_level, trigger_slope)
            ELSE
               CALL gage_reset_interpolate_trigger(board, (interpolate_trigger_channel And 1) +
                                                       GAGE_CHAN_A, 0, 0, 0)
            END IF
        ELSE
            CALL gage_reset_interpolate_trigger(0, GAGE_CHAN_A, 0, 0, 0)
                                 'Disable interpolated triggering.
        END IF
END IF
END SUB


SUB low_level_capture (ByVal boards As Integer)
‘  Note:  This is not the recommended way of doing data capture.
‘  It may be used to optimize single board systems.
i := gage_select_board (1)              ' Select MASTER.
CALL gage_abort ()                      ' Clear MASTER operation.
CALL gage_init_clock ()                 ' Start MASTER clock.
CALL gage_get_data_low ()                           ' Begin data capture sequence for MASTER.
FOR i = 2 TO boards - 1
            dummy = gage_select_board (i)       ' Select SLAVE "i".
            CALL gage_abort ()                  ' Clear SLAVE "i" operation.
            CALL gage_get_data_low ()           ' Start data capture for SLAVE "i" .
NEXT
FOR i = boards TO 2 STEP -1
            dummy := gage_select_board (i)      ' Select SLAVE "i".
            CALL gage_get_data_high ()          ' Wait for data capture acknowledge for SLAVE "i".
NEXT
dummy = gage_select_board (i)           ' Re-select MASTER.
CALL gage_get_data_high ()              ' Wait for MASTER data capture acknowledge to finish
                                        ‘           sequence.
DO
            i = gage_triggered ()
LOOP UNTIL i <> 0
DO
            i = gage_busy ()
LOOP UNTIL i = 0                                    ' Data has now been received.
END SUB
```

Sample Programs.

The following section describes some of the sample programs provided on the distribution diskettes for the CompuScope drivers. These programs are meant to be examples of how to write application programs using the CompuScope drivers and can be used as a starting point for your own programs. The descriptions focus on the routines used by the drivers. It is assumed the programmer has some knowledge of Windows and / or DOS programming in the language of choice.

As more sample programs become available, they will be provided on future releases of the driver and uploaded to the Gage BBS at 514-337-4317.

The DOS sample programs in C have one set of source files that are compilable under either Borland, Microsoft or Watcom C. Project or makefiles for the sample programs are provided on your distribution diskettes for each of these compilers. These files have names that end in B (for Borland), M (for Microsoft) or W (for Watcom).

The sample programs provided are written as generally as possible and will work with any CompuScope board (s), either in single card or master / slave setups.

Every effort has been made to keep these descriptions as accurate as possible. However, the sample programs often get updated after the manual has been completed. If any discrepancies exist between the documentation and the sample programs or driver code on the diskette, the version on the diskette should be considered the correct version.

Sample Programs for C.

The sample programs included with the CompuScope C drivers offer the illustration of two common applications of the CompuScope hardware. Capturing signals and displaying them and capturing a signal and transferring the data to a disk file.

GAGETSTx.EXE programs.

These three programs use the same source code. **GAGETSTB.EXE** is compiled using Borland C++ Version 3.1, **GAGETSTM.EXE** is compiled using Microsoft C++ Version 7.0 and **GAGETSTW.EXE** using Watcom C 9.0.

This sample program captures data and then displays the information on the display. Besides the driver files, the additional source files required are the **STRUCTS.H**, **STRUCTS.C**, **SCREENS.H**, **SCREENS.C** and **TEST.C** files.

TEST.C

The TEST.C file contains all of the initialization code along with examples of the code required to set different parameters governing the capture of data. The following is a discussion of what each routine does and why it is required.

prepare_for_capture

This routine calls the CompuScope C driver routines in the proper order for both single board systems and master/slave configurations. The parameters used are taken from the **boarddef** structure **board** described earlier. The variables can be changed at any time prior to calling this routine.

exercise_compuscopes

This routine calls the screen handling routines from the SCREENS.C file and establishes the values for the required parameters in the **boarddef** structure **board** and then calls the prepare_for_capture routine.

main

The main routine initializes the **gage_board_location** array by calling the **gage_read_config_file** routine to read the **GAGESCOP.INC** configuration file. If there was a problem with the configuration file then the **gage_board_location** array is initialized with calls to the gage_set_records routine to load the default values. The **gage_board_location** array is then passed to the **gage_driver_initialize** routine to initialize the hardware. Note that **gage_driver_initialize** is called twice, once to pre-initialize the hardware and the second to actually initialize the CompuScope C driver. This is required primarily for master/slave operation where multiple cards are present in the data acquisition system. Once the return value is verified then the **exercise_compuscopes** routine is called to control the capture of data.

SCREENS.C

The SCREENS.C file has all the support routines for displaying the data on the screen and one routine which controls the CompuScope hardware in a continuous capture and display mode. Most of the support code will not be discussed since it is not required for the acquisition of data, only for the display of data. The **continuous_display_trace** routine will be discussed in detail since it is the routine of the most interest.

continuous_display_trace

This routine starts by taking care of some housekeeping tasks required by the operation mode and/or the type of hardware in use. The single channel data capture is displayed slightly differently and the 12-bit data acquisition boards also store their data differently than the 8-bit CompuScope hardware. Some display overhead is performed next prior to starting the main loop of the routine. After the keyboard is checked for activity the current data capture is started by calling the **gage_start_capture** routine. The routine then waits for the trigger event to occur. After either the trigger occurs or a trigger is forced by calling **gage_forced_trigger_capture** the routine waits for the data to be captured by monitoring the **gage_busy** routine. After the data is captured the routine sets the required memory addresses by calling the **gage_calculate_addresses** routine. The CompuScope memory is made visible to the CPU by calling the **gage_need_ram** routine and either the **gage_mem_read_single** or **gage_mem_read_dual** routines are used to extract the data points from the CompuScope RAM buffers. The data is then converted into a displayable form and placed on the screen. This is repeated for each available channel and then the next data acquisition is started completing the cycle.

STRUCTS.H and **STRUCTS.C**

These files define the **boarddef**, **srtype** and **irtype** structures and initialize the variables based on these structures. A more complete discussion of these files was presented in the **A simple application using the CompuScope C drivers** section of this manual.

GAGEA2Dx.EXE sample programs.

These three programs use the same source code. **GAGEA2DB.EXE** is compiled using Borland C++ Version 3.1, **GAGEA2DM.EXE** is compiled using Microsoft C++ Version 7.0 and **GAGEA2DW.EXE** using Watcom C Version 9.0.

This sample program captures data and then stores the captured information to disk files using the GAGESCOPE binary signal file format. Besides the driver files, the additional source files required are the **STRUCTS.H**, **STRUCTS.C**, **DISKFILE.H**, **DISKFILE.C** and **ACQ2DISK.C** files.

ACQ2DISK.C

The ACQ2DISK.C file contains all of the initialization code along with an example of the code required to set different parameters governing the capture of data. The following is a discussion of what each routine does and why it is required.

readkey

Reads the keyboard taking into account "special" keys.

prepare_for_capture

This routine calls the CompuScope C driver routines in the proper order for both single board systems and master/slave configurations. The parameters used can be made into variables that will contain valid data at the time of the calls to the driver routines.

transfer_cs_2_file

This routine sets up the block number of the trigger address in memory to correctly access the desired portion of the sampled signal. The CompuScope buffers are then made available to the CPU by calling the gage_need_ram routine. The block number is initialized and the data is transferred directly from the CompuScope buffer to the disk file by using the fwrite library command. This process is repeated until all of the data to be transferred has been moved to the disk file.

acquire_2_disk_file

This routine controls the data acquisition process. The parameter tbsins is the time between samples in nanoseconds and the trigger_source parameter is self explanatory. The data acquisition is started by a call to gage_start_capture. If required calls to gage_forced_trigger_capture and/or gage_abort_capture may be necessary if the time-out criteria occur. Next for each board in the system and each active channel on each card a unique filename is created and the file is opened. The memory addresses are calculated and the open file and these addresses plus the number of samples of data to be transferred are passed to the transfer_cs_2_file for the actual transfer to the disk file. Then the disk file is closed and process is repeated for the remaining boards.

perform_acquisition

This routine simply calls the prepare_for_capture and acquire_2_disk_file routines.

main

The main routine initializes the gage_board_location array by calling the gage_read_config_file routine and then passes the array to the gage_driver_initialize routine to initialize the hardware. Note that gage_driver_initialize is called twice, once to pre-initialize the hardware and the second to actually initialize the CompuScope C driver. This is required primarily for master/slave operation where multiple cards are present in the data acquisition system. Once the return value is verified then the perform_acquisition routine is called to control the capture of data.

SOFTWARE MULTIPLE RECORD.

This document describes the new SOFT_MRx and TRA2DSKx programs. The x is either a B, M or a W for Borland, Microsoft or Watcom respectfully. These two programs are very similar when comparing the results generated but work in much different ways to provide solutions to two common applications.

The SOFT_MRx program performs a software multiple record function that optionally allows trigger interpolation and will work with any version of CompuScope hardware. The program can acquire approximately 1500 256 byte samples every second when the hardware is running at 10 MSPS. The hardware multiple record option, for comparison, can capture approximately 38000 256 byte samples every second. The number of samples that can be taken is limited by the amount of extended memory available to the application. The data sets are acquired and transferred into extended memory until all the samples are taken. The data is then written to the disk with a GageScope header and the multiple record field of the header set to two. This value can be used by GageScope to read the file as a software multiple record file so the acquired data can be viewed. The program will work either in single or dual channel mode.

The TRA2DSKx program works quite differently. The file format is the same (Software Multiple Record), however, the data stored is different. The data is sampled as before but both channel’s data is transferred to the output file immediately after acquisition, before the next set of data is sampled. The advantage that this program has is that the size of the total acquisition is only limited by hard disk space. If the data is captured in the single channel operation mode then the data is de-multiplexed before saving the data. The data then appears to be dual channel data. This operation must be done since it is conceivable that at run time the number of acquisitions may not be known (assuming a programmer using these drivers makes the necessary changes). The CompuScope 1012 driver code has a routine called cs1012_direct_disk_transfer which saves data to either a file pointer or a file handle (established by conditional compile directives). This routine will increase performance by about 2.9% and 8.3% respectfully. The calling code will need to be changed slightly to allow the use of this routine.

Sample Program for Protected Mode Pascal

The sample program included with the GageScope protected-mode DLL shows the basics of operating the CompuScope hardware and displaying the captured signal.

GAGETSTP.EXE sample program.

This sample program shows the proper order in which to call the DLL functions in order to capture data on your CompuScope hardware. The program consists of four files, GAGETSTP.PAS, SCREENS.PAS, GAGE_DRV.PAS and START.PAS.

GAGETSTP.PAS

The GAGETSTP.PAS file contains all the code to initialize the CompuScope hardware along with examples of code required to set different parameters governing the capture of data. The following is a discussion of what each routine does and why it is required.

prepare_for_capture

This routine calls the GageScope DLL routines in the proper order for both single board systems and master/slave configurations. Note that the routines work on the currently selected board. The parameters used can be made into variables that will contain valid data at the time of the calls to the DLL routines.

exercise_compuscopes

This routine calls the screen handling routines from the SCREENS.PAS file and establishes the values for the required parameters to the prepare_for_capture routine.

The main program body of GAGETSTP.PAS initializes the gage_board_location array by calling the **gage_read_config_file** function and the passes the array to the **gage_driver_initialize** routine to initialize the hardware. If the configuration file, GAGESCOP.INC, is not found, then **gage_set_records** is called with the default values of $D000 for the segment and $0200 for the index. Note that the **gage_driver_initialize** routine is called twice, once to pre-initialize the hardware and the second time to actually initialize the DLL. This is required primarily for master/slave operation where multiple cards are present in the data acquisition system. Once the return value is verified, the exercise_compuscopes routine is called to control the capture of data.

SCREENS.PAS

The SCREENS.PAS file has all the support routines for displaying the data on the screen and one routine which controls the CompuScope hardware in a continuous capture and display mode. Most of the support code will not be discussed since it is not required for the acquisition of data, only for the display of data. The continuous_display_trace routine will be discussed in detail since it is the routine of most interest.

continuous_display_trace

This routine starts by taking care of some housekeeping tasks required by the operation mode and/or the type of hardware in use. The single channel data capture is displayed slightly differently then the dual channel data capture and the 12-bit data acquisition boards store their data differently than the 8-bit CompuScope hardware. The data is returned from an 8-bit board as an unsigned byte between 0 and 255, with 0 representing the maximum voltage and 255 representing the minimum. For a 12-bit board the data is returned as an integer, with -2048 representing the maximum voltage and +2047 representing the minimum. Some display overhead is performed prior to starting the main loop of the routine. After the keyboard is checked for activity, the current data capture is started by calling the **gage_start_capture** routine. The routine then waits for the trigger event to occur by checking the **gage_triggered** routine. After either the trigger occurs or a trigger is forced by calling **gage_forced_trigger_capture**, the routine waits for the data to be captured by monitoring the **gage_busy** routine. Once the data is captured the routine sets the required memory addresses by calling the **gage_calculate_addresses** routine. The CompuScope memory is made visible to the CPU by calling the **gage_need_ram** routine and either the **gage_mem_read_single** or **gage_mem_read_dual** routines are used to extract the data points from the CompuScope RAM buffers. The data is then converted into displayable form and placed on the screen. This is repeated for each available channel and then the next data acquisition is started. completing the cycle

GAGE_DRV.PAS

The GAGE_DRV.PAS file contains the constants, functions and procedures used to access the GAGE_DRV.DLL. It's operation is exactly the same as its Windows counterpart, with the exceptions noted in the preceding section. When a DLL routine is required, the name of the routine is used just as in a regular Pascal subroutine call.

START.PAS

The START unit contains some structures and variables used by the sample program. They are initialized from the main program with a call to init_variables, which is contained in START.PAS.

Sample Programs for Quick Basic

SIMPLE.BAS

The sample program is broken down into several sections that together can form the basis for a new application tailored to specific requirements. Each section will be defined by the line numbers of the original source file, SIMPLE.BAS.

MAIN PROGRAM

Line 8: Include the driver definitions file GAGE_DRV.BAS.

Lines 12 to 17: Declare the function and subroutines that are used by the sample program and are declared in the SIMPLE.BAS source file. The code for these functions and subroutines will be described later in this section.

Line 21: Define a constant which will limit the maximum number of points to be displayed by the "outputsymbolpoints" subroutine.

Lines 23 to 34: Define various variables that are used by the sample program, each variables use is described along with is definition.

Line 36: Define the "gageboardlocation" array, this array is used to store the desired locations of the installed hardware and is passed to the "gagereadconfigfile" and/or "gagesetrecords" functions for initialization and then to the "gagedriverinitialize" function to initialize the hardware.

Line 38: Define the "gdi" structure which is based on the user defined type called "gagedriverinfotype" and is defined in the GAGE_DRV.BAS file.

Lines 45 to 52: Initialize the "gageboardlocation" array and report any errors.

Lines 54 to 72: Initialize the driver and the hardware and report any errors. Note the two calls to "gagedriverinitialize". The first call pre-initializes the hardware and the second call actually performs the driver initialization. Also of note is the fact that if "gagedriverinitialize" returns a value less than one then not all boards were found during initialization. The missing boards can be determined by comparing the driver values for the board segment and index as reported by "gagesetdriverinfo" (see lines 68 to 72) and the "gageboardlocation" array.

Lines 74 to 82: This section establishes the installed CompuScope boards resolution and sets the two parameters the govern the display of the two different resolutions.

Lines 86 to 91: This section initializes several variables that will control the data acquisition and display of the acquired data.

Lines 93 to 118: This section initializes the hardware prior to data acquisition and reports any errors or the new settings for the board. This section would require expanding if more boards are to be installed.

Lines 122 to 125: This section is where the actual data acquisition occurs. "gagestartcapture" is used to begin the acquisition and "waitgageready" is used to wait until the acquisition is complete and the call to "gagesetdriverinfo" is used to return the maximum memory for each channel.

Line 129: The "outputsymbolpoints" routine is used to display the acquired data as voltage levels based on the gain used during capture.

Line 134: This line removes the driver from memory immediately prior to exiting.

FUNCTIONS AND SUBROUTINES

outputsymbol: Displays the voltage level of the passed variable. This routine uses the "offset8bit" and "sampleres" variables to scale the data based on whether an 8 bit or 12 bit board was initialized. Also used is the "gnf" or gain factor variable. The gain factor variable must be initialized prior to calling this routine otherwise unpredictable results may occur. See the code for the "outputsymbolpoints" routine for examples of the proper calling procedure.

outputsymbolab: Displays the voltage levels of the passed variables. This routine uses the "offset8bit" and "sampleres" variables to scale the data based on whether an 8 bit or 12 bit board was initialized. Also used are the "gnfa" and "gnfb" or gain factor variables for channel A and channel B. The gain factor variable must be initialized prior to calling this routine otherwise unpredictable results may occur. See the code for the "outputsymbolpoints" routine for examples of the proper calling procedure.

outputsymbolpoints: This routine can form the basis of any display routine. The first task this subroutine performs is the initialization of the current memory pointers by calling the "gagecalculateaddresses" subroutine in the driver. The values returned are then used to set the starting point of the displayed data and the number of points to display (count& and max& respectfully, note also that these variables and the address variables are long integers). The memory in the CompuScope hardware is organized as a circular buffer, therefore the driver routine "gagenormalizeaddress" is used to make logical comparisons of the address variables. Calling the "gageneedram" routine with a true value will connect the CompuScope memory to the PC bus so that the samples may be extracted from the hardware. The channels parameter is now checked to see what display format is required, if equal to zero the display single channel data, if equal to two display dual channel data, if equal to three display channel A data only while in dual channel mode and if equal to four display channel B data only while in dual channel mode.

paused: This function is called to check if the user would like to pause the display of the data. If a key has been pressed then the routine reads it and waits for another. If the next key pressed is a space then display continues, otherwise the display is halted.

printgbltable: This routine prints the gageboardlocation table and illustrates the organization of this array. The first element is the segment for the first board, the second element is the index for the first board and the thirty-third element is the status of the first board (which is set by the "gagedriverinitialize" function). The second board uses elements 3, 4 and 34 respectfully and so on for a maximum of sixteen boards.

gagewaitready: This routine wait for the hardware to encounter a trigger or to time-out waiting and then force a trigger event to occur by calling "gageforcedtriggercapture". Next the routine waits for the acquisition to finish or to time-out waiting which will cause the current acquisition to be aborted by calling "gageabortcapture".

TEST.BAS

This sample program included with the GageScope Quick Basic driver shows the basics of operating the CompuScope hardware and displaying the captured signal.

TEST.BAS

The TEST.BAS file contains all the code to initialize the CompuScope hardware along with examples of code required to set different parameters governing the capture of data. The following is a discussion of what each routine does and why it is required.

prepareforcapture

This routine calls the GageScope driver routines in the proper order for both single board systems and master/slave configurations. Note that the routines work on the currently selected board. The parameters used can be made into variables that will contain valid data at the time of the calls to the driver routines.

exercisecompuscopes

This routine calls the screen handling routines and establishes the values for the required parameters to the prepareforcapture routine.

The main program body of TEST.BAS initializes the gageboardlocation array by calling the **gagereadconfig_file** function and the passes the array to the **gagedriverinitialize** routine to initialize the hardware. If the configuration file, GAGESCOP.INC, is not found, then **gagesetrecords** is called with the default values of &HD000 for the segment and &H0200 for the index. Note that the **gagedriverinitialize** routine is called twice, once to pre-initialize the hardware and the second time to actually initialize the driver. This is required primarily for master/slave operation where multiple cards are present in the data acquisition system. Once the return value is verified, the exercisecompuscopes routine is called to control the capture of data.

Most of the support code will not be discussed since it is not required for the acquisition of data, only for the display of data. The continuousdisplaytrace routine will be discussed in detail since it is the routine of most interest.

continuousdisplaytrace

This routine starts by taking care of some housekeeping tasks required by the operation mode and/or the type of hardware in use. The single channel data capture is displayed slightly differently then the dual channel data capture and the 12-bit data acquisition boards store their data differently than the 8-bit CompuScope hardware. The data is returned from an 8-bit board as an unsigned byte between 0 and 255, with 0 representing the maximum voltage and 255 representing the minimum. For a 12-bit board the data is returned as an integer, with -2048 representing the maximum voltage and +2047 representing the minimum. Some display overhead is performed prior to starting the main loop of the routine. After the keyboard is checked for activity, the current data capture is started by calling the **gagestartcapture** routine. The routine then waits for the trigger event to occur by checking the **gagetriggered** routine. After either the trigger occurs or a trigger is forced by calling **gageforcedtriggercapture**, the routine waits for the data to be captured by monitoring the **gagebusy** routine. Once the data is captured the routine sets the required memory addresses by calling the **gagecalculateaddresses** routine. The CompuScope memory is made visible to the CPU by calling the **gageneedram** routine and either the **gagememreadsingle** or **gagememreaddual** routines are used to extract the data points from the CompuScope RAM buffers. The data is then converted into displayable form and placed on the screen. This is repeated for each available channel and then the next data acquisition is started. completing the cycle

GAGE_DRV.BAS

The GAGE_DRV.BAS file contains the constants, functions and procedures used to access the GageScope driver. When a driver routine is required, the name of the routine is used just as in a regular Basic subroutine call.

INIT.BAS

The INIT.BAS file contains some structures and variables used by the sample program. They are initialized in the main program with a call to initvariables.

The Sample Program for Windows: C.

The basic sample program, GSDLLDEM.C, was created using the design tools that are bundled with the Microsoft Quick C package. It features discrete and continuous data capture with a re-sizable window that continues to be updated even when the control focus shifts to other applications. Several routines and data structures have been added to the sample program that will allow the programmer to get started quickly.

Data Structures.

The **boarddef** structure defines all of the settings that may be changed on the CompuScope card. The **srtable** structure defines all of the possible sample rate settings that any of the CompuScope cards may use. The top 16 bits of the flag is used to specify if the sample rate is available single channel and the bottom 16 bits are used to specify that the sample rate is available when acquiring data in the dual channel mode. The **irtype** structure similarly defines which input ranges are available for the installed CompuScope hardware. The structure has a flag field whose operation is similar to the flag used by the **srtype** structure. The structures are defined in the file STRUCTS.H and initialized in STRUCTS.C.

The first data structure defines variables that are used prior to setting the hardware parameters. An entry for each of the relevant controls is present. The data structure is used by the "SetBoard" routine, which sends the value of each parameter to the hardware.

```
typedef struct {              /*  All constants mentioned can be found in the file GAGE_DRV.H. */
        int16   opmode;       /*  Operation Mode.  Use the GAGE_SINGLE_CHAN or
                                  GAGE_DUAL_CHAN constants.  */
        int16   srindex;      /*  Sample Rate INDEX is the index to the sample rate table.  */
        int16   range_a;      /*  The input range for channel A.  Use one of  the following defined
                                  constants: GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V,
                                  GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV.
                                  Remember that not all constants are available for all versions of
                                  hardware.  See the table in GAGE_DRV.H where these constants are
                                  defined for an up to date listing.  */
        int16   couple_a;     /*  The input coupling for channel A.  Use one of  the following
                                  defined constants: GAGE_DC or GAGE_AC.  */
        int16   range_b;      /*  The input range for channel B.  Use one of  the following defined
                                  constants: GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V,
                                  GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV.
                                  Remember that not all constants are available for all versions of
                                  hardware.  See the table in GAGE_DRV.H where these constants are
                                  defined for an up to date listing.  */
        int16   couple_b;     /*  The input coupling for channel B.  Use one of  the following
                                  defined constants: GAGE_DC or GAGE_AC.  */
        int16   source;       /*  The trigger source for the next data acquisition.  Use one of  the
                                  following defined constants: GAGE_CHAN_A, GAGE_CHAN_B,
                                  GAGE_EXTERNAL or GAGE_SOFTWARE.  */
        int16   slope;        /*  The trigger slope for the next data acquisition.  Use one of  the
                                  following defined constants: GAGE_POSITIVE or
                                  GAGE_NEGATIVE.  */
        int32   depth;        /*  The trigger or sample depth for the next data acquisition.  Use one
                                  of  the following defined constants: GAGE_POST_0K,
                                  GAGE_POST_128, GAGE_POST_256, GAGE_POST_512,
                                  GAGE_POST_1K, GAGE_POST_2K, GAGE_POST_4K,
                                  GAGE_POST_8K, GAGE_POST_16K, GAGE_POST_32K,
                                  GAGE_POST_64K, GAGE_POST_128K, GAGE_POST_256K,
                                  GAGE_POST_512K, GAGE_POST_1M, GAGE_POST_2M,
                                  GAGE_POST_4M or GAGE_POST_8M.  These constants are the
                                  only available sample depths for the CompuScope 220 series of
                                  boards.  The CompuScope LITE can have the sample depth set to any
                                  modulo 16 value (0, 16, 32, 48, ..., max RAM).  The CompuScope 250
                                  and the CompuScope 1012 can have the sample depth set to any
                                  modulo 64 value (0, 64, 128, 192, ... , max RAM).  */
        int16   level;          /*  The trigger level for the external trigger input.  This value ranges
                                  from 0 to 255.  0 volts trigger is actually 128 and all other values for
                                  the voltage range used are scaled accordingly.  */
        int16   range_e;        /*  The input range for the external trigger input.  Use one of  the
                                  following defined constants: GAGE_PM_5_V or GAGE_PM_1_V.
                                  NOTE: GAGE_PM_5_V is not available on the CompuScope LITE.
                                  */
        int16   couple_e;       /*  The input coupling for the external trigger input.  Use one of  the
                                  following defined constants: GAGE_DC or GAGE_AC.  NOTE:
                                  GAGE_AC is not available on the CompuScope LITE.  */
} boarddef;
```

The irtype structure is not currently used by the sample program, but is included for future use.

```
typedef struct {
        int16           constant;       /*Value of the driver constant for the input range.      */
        uInt32          gf_flag;        /*Flag indicates which board supports which gain.        */
        double          gf_calc;        /*Multiplier used for display voltage amplitude.         */
        char            *gf_text;       /*Text associated with the input range.          */
} irtype;
```

The sample rate table structure is not needed, however it is included and initialized in the GSDLLDEM.C file.

```
typedef struct {
        int16           rate;           /*  Rate used for setting the CompuScope.  */
        int16           mult;           /*  Multiplier used for setting the CompuScope.  */
        uInt32          sr_flag;        /*  Flag to indicate which board supports which sample rate
                                        in which mode it is supported.  */
        float           sr_calc;        /*  Time between samples (in ns).  Used in calculating the
                                        number of points.  */
        char            *text;          /*  Text associated with the current settings of the
                                        CompuScope.  */
} srtype;
```

Function Prototypes.

The following functions have been written to help speed up the development process for the programmer using the CompuScope DLL and the CompuScope Series of high speed data acquisition cards.
**void	SetDefaultBoardLocation (uInt16 seg, uInt16 ind);**

This routine is called if the DLL routine "gage_read_config_file" returns false indicating the board location configuration file is corrupted or missing. A default segment and index are passed to this routine and the global data structure "gage_board_location" is updated prior to calling the DLL routine "gage_driver_initialize.

**char	*BoardTypeSizeToText (int16 board_type, int32 max_memory);**

This simple routine just converts the two constants, defined in "GAGE_DRV.H" passed to it to the text equivalent. For example, consider the following code fragment:

```
int16	board_type = GAGE_ASSUME_CS220;
int32	max_memory = GAGE_MEMORY_SIZE_256K;
char	*str;

str = BoardTypeSizeToText (board_type, max_memory);
```

"str" would now point to a buffer containing the string "220 256K".

**int	InitBoard (HWND hWnd);**

This routine does all the necessary calls in the proper order to initialize the driver. Some modification to this routine will be necessary if the programmer does not want the progression messages displayed.

**void	SetBoard (HWND hWnd);**

This routine uses the "boarddef" structure named "board" which was previously loaded with the desired capture parameters. The sample program initializes these parameters statically, it is the programmers responsibility to perform the same task prior to capturing data on the CompuScope card. Note the order that these routines are called as the ordering is important for the proper operation of the CompuScope hardware and for the CompuScope DLL driver to maintain correct information as to the size of memory available to each channel etceteras.

**void	InitMemPtrs (void);**

This routine is used to create the appropriate addresses of the signal just captured. It should help the programmer understand how data is stored in the CompuScope cards and how to retrieve it. Also, it will require slight adjustments if multiple boards are to be supported.

**void	AcquireStart (HWND hWnd);**
**void	AcquireCheck (HWND hWnd);**
**void	Abort (HWND hWnd);**

These three routines are closely coupled to perform the basic data capture sequences. Study these routines to determine how they can be modified to perform the desired task for the application under development.
The "AcquireStart" routine initializes the hardware and starts the data acquisition process and issues a software trigger if required.
The "AcquireCheck" routine checks the status bits in the hardware to determine when the trigger has been received and the data acquisition has been completed, the memory pointers are then initialized and the re-

display is requested. If the ContinuousMode flag was set then the "AcquireStart" routine is called again to repeat the cycle.
The "Abort" routine disables the ContinuousMode, if it is set, which will cause the hardware to stop capturing data the next time data acquisition has been completed. If the ContinuousMode was not set then the hardware is aborted directly.

**void	Display (HWND hWnd);**

This simple routine invalidates the window where the scope data is being sent, via a call to "InvalidateRect(hWnd, NULL, FALSE)" in order to re-paint the screen when the "UpdateWindow(hWnd)" command is processed.

The Sample Program: Turbo Pascal for Windows.

The basic sample program, GSDLLDEM.PAS , was created using the design tools and Object Windows Library that are bundled with the Borland Turbo Pascal for Windows package. The program has been tested under both Turbo Pascal for Windows 1.0 and Borland Pascal 7.0. It features discrete and continuous data capture with a re-sizable window that continues to be updated even when the control focus shifts to other applications. Several routines and data structures have been added to the sample program that will allow the programmer to get started quickly.

Data Structures.

The **boarddef** structure defines all of the settings that may be changed on the CompuScope card. The **srtable** structure defines all of the possible sample rate settings that any of the CompuScope cards may use. The top 16 bits of the flag is used to specify if the sample rate is available single channel and the bottom 16 bits are used to specify that the sample rate is available when acquiring data in the dual channel mode. The **irtype** structure similarly defines which input ranges are available for the installed CompuScope hardware. The structure has a flag field whose operation is similar to the flag used by the **srtype** structure. The structures are defined and initialize in the file START.PAS.

The first data structure defines variables that are used prior to setting the hardware parameters. An entry for each of the relevant controls is present. The data structure is used by the "SetBoard" routine, which sends the value of each parameter to the hardware.

```
type
boarddef = record                 /*  All constants mentioned can be found in the file GAGE_DRV.PAS.
                                  */
        opmode:     int16;        /*  Operation Mode.  Use the GAGE_SINGLE_CHAN or
                                  GAGE_DUAL_CHAN constants.  */
        srindex:    int16;        /*  Sample Rate INDEX is the index to the sample rate table.  */
        range_a:    int16;        /*  The input range for channel A.  Use one of  the following defined
                                  constants : GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V,
                                  GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV.
                                  Remember that not all constants are available for all versions of
                                  hardware.  See the table in GAGE_DRV.PAS where these constants are
                                  defined for an up to date listing.*/
        couple_a:   int16;        /*  The input coupling for channel A.  Use one of  the following defined
                                  constants: GAGE_DC or GAGE_AC.  */
        range_b:    int16;        /*  The input range for channel B.  Use one of  the following defined
                                  constants: GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V,
                                  GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV.
                                  Remember that not all constants are available for all versions of
                                  hardware.  See the table in GAGE_DRV.PAS where these constants are
                                  defined for an up to date listing.*/
        couple_b:   int16;        /*  The input coupling for channel B.  Use one of  the following defined
                                  constants: GAGE_DC or GAGE_AC.  */
        source:     int16;        /*  The trigger source for the next data acquisition.  Use one of  the
                                  following defined constants: GAGE_CHAN_A, GAGE_CHAN_B,
                                  GAGE_EXTERNAL or GAGE_SOFTWARE.  */
        slope:      int16;        /*  The trigger slope for the next data acquisition.  Use one of  the
                                  following defined constants: GAGE_POSITIVE or
                                  GAGE_NEGATIVE.  */
        depth:      int32;        /*  The trigger or sample depth for the next data acquisition.  Use one of
                                  the following defined constants: GAGE_POST_0K,
                                  GAGE_POST_128, GAGE_POST_256, GAGE_POST_512,
                                  GAGE_POST_1K, GAGE_POST_2K, GAGE_POST_4K,
                                  GAGE_POST_8K, GAGE_POST_16K, GAGE_POST_32K,
                                  GAGE_POST_64K, GAGE_POST_128K, GAGE_POST_256K,
                                  GAGE_POST_512K, GAGE_POST_1M, GAGE_POST_2M,
                                  GAGE_POST_4M or GAGE_POST_8M.  These constants are the only
                                  available sample depths for the CompuScope 220 series of boards.  The
                                  CompuScope LITE can have the sample depth set to any modulo 16
                                  value (0, 16, 32, 48, ..., max RAM).  The CompuScope 250 can have
                                  the sample depth set to any modulo 64 value (0, 64, 128, 192, ... , max
                                  RAM).  */
        level:       int16;        /*  The trigger level for the external trigger input.  This value ranges
                                  from 0 to 255.  0 volts trigger is actually 128 and all other values for the
                                  voltage range used are scaled accordingly.  */
        range_e:     int16;        /*  The input range for the external trigger input.  Use one of  the
                                  following defined constants: GAGE_PM_5_V or GAGE_PM_1_V.
                                  NOTE: GAGE_PM_5_V is not available on the CompuScope LITE.*/
        couple_e:    int16;        /*  The input coupling for the external trigger input.  Use one of  the
                                  following defined constants: GAGE_DC or GAGE_AC.  NOTE:
                                  GAGE_AC is not available on the CompuScope LITE.  */
end;
```

The following irtype structure is not used by the sample program but is provided for future use.

```
irtype = record
     constant:    int16;        { Value of the driver constant for the input range.        }
     gf_flag:     int32;        { Flag indicates which board supports which gain.          }
     gf_calc:     double;       { Multiplier used for display voltage amplitude.           }
     gf_text:     pchar;        { Text associated with the input range.                    }
end;
```

The sample rate table structure is not needed, however it is included and initialized in the START.PAS unit file.

```
type
srtype = record
     rate:    int16;    {  Rate used for setting the CompuScope.                         }
     mult:    int16;    {  Multiplier used for setting the CompuScope.                   }
     sr_flag: int32;    {  Flag to indicate which board supports which sample rate.      }
                        {  and in which mode it is supported.                            }
     sr_calc: single;   {  Time between samples (in ns).  Used in calculating the        }
                        {  number of points.                                             }
     text:    pchar;    {  Text associated with the current settings of the  CompuScope. }
end;

srtable: array[0..40] of srtype;
```

Function Prototypes.

The following functions have been written to help speed up the development process for the programmer using the CompuScope DLL and the CompuScope Series of high speed data acquisition cards.

procedure SetDefaultBoardLocation (seg, ind : word);

This routine is called if the DLL routine "gage_read_config_file" returns false indicating the board location configuration file is corrupted or missing. A default segment and index are passed to this routine and the global data structure "gage_board_location" is updated prior to calling the DLL routine "gage_driver_initialize.

function BoardTypeSizeToText (board_type : integer, max_memory : longint) : pchar;

This simple routine just converts the two constants, defined in "GAGE_DRV.PAS", passed to it and returns the text equivalent. For example, consider the following code fragment:

tempstr: array[0..15] of char;
board_type := GAGE_ASSUME_CS220;
max_memory := GAGE_MEMORY_SIZE_256K;

StrCopy (tempstr, BoardTypeSizeToText (board_type, max_memory));

"tempstr" would now contain the string "220 256K".

function InitBoard : integer;

This routine does all the necessary calls in the proper order to initialize the driver. Some modification to this routine will be necessary if the programmer does not want the progression messages displayed. As written, the sample program will abort if no CompuScope board is found or if "gage_select_board" fails. The probable cause of this is an incorrect segment or index value in the configuration file. The problem can be corrected by running the "GSINST" utility.

procedure SetBoard ;

This routine uses the "boarddef" structure named "board" which was previously loaded with the desired capture parameters. The sample program initializes these parameters statically, it is the programmers responsibility to perform the same task prior to capturing data on the CompuScope card. Note the order that these routines are called as the ordering is important for the proper operation of the CompuScope hardware and for the CompuScope DLL driver to maintain correct information as to the size of memory available to each channel etceteras.

procedure InitMemPtrs ;

This routine is used to create the appropriate addresses of the signal just captured. It should help the programmer understand how data is stored in the CompuScope cards and how to retrieve it. Also, it will require slight adjustments if multiple boards are to be supported.

procedure AcquireStart ;
procedure AcquireCheck ;
procedure Abort ;

These three routines are closely coupled to perform the basic data capture sequences. Study these routines to determine how they can be modified to perform the desired task for the application under development. The "AcquireStart" routine initializes the hardware and starts the data acquisition process and issues a software trigger if required.

The "AcquireCheck" routine checks the status bits in the hardware to determine when the trigger has been received and the data acquisition has been completed, the memory pointers are then initialized and the re-display is requested. If the ContinuousMode flag was set then the "AcquireStart" routine is called again to repeat the cycle. Note that in the sample program, "AcquireCheck " is called whenever a timer event occurs. The C sample program, GSDLLDEM.C, demonstrates the different technique of placing "AcquireCheck" in the windows message loop.
The "Abort" routine disables the ContinuousMode, if it is set, which will cause the hardware to stop capturing data the next time data acquisition has been completed. If the ContinuousMode was not set then the hardware is aborted directly.

procedure Display ;

This simple routine invalidates the window where the scope data is being sent, via a call to "InvalidateRect(HWindow, nil, false)" in order to re-paint the screen when the "UpdateWindow(HWindow)" command is processed.

The Sample Program: Visual Basic.

The Visual Basic sample program, GSDLLDEM.EXE , was created using the design tools that are available in the Microsoft Visual Basic 3.0 package. It features discrete and continuous data capture with a re-sizable window that continues to be updated even when the control focus shifts to other applications. Several routines and data structures have been added to the sample program that will allow the programmer to get started quickly.

Data Structures.

The **boarddef** structure defines all of the settings that may be changed on the CompuScope card. The **srtable** structure defines all of the possible sample rate settings that any of the CompuScope cards may use. The top 16 bits of the flag is used to specify if the sample rate is available single channel and the bottom 16 bits are used to specify that the sample rate is available when acquiring data in the dual channel mode. The **irtype** structure similarly defines which input ranges are available for the installed CompuScope hardware. The structure has a flag field whose operation is similar to the flag used by the **srtype** structure. The structures are defined and initialized in the file START.BAS.

The first data structure defines variables that are used prior to setting the hardware parameters. An entry for each of the relevant controls is present. The data structure is used by the "SetBoard" routine, which sends the value of each parameter to the hardware.

Type boarddef			/* All constants mentioned can be found in the file GAGE_DRV.BAS. */
	opmode	**As Integer**	**/*** Operation Mode. Use the GAGE_SINGLE_CHAN or GAGE_DUAL_CHAN constants. */
	srindex	**As Integer**	/* Sample Rate INDEX is the index to the sample rate table.
	range_a	**As Integer**	/* The input range for channel A. Use one of the following defined constants: GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V, GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV. Remember that not all constants are available for all versions of hardware. See the table in GAGE_DRV.PAS where these constants are defined for an up to date listing. */
	couple_a	**As Integer**	/* The input coupling for channel A. Use one of the following defined constants: GAGE_DC or GAGE_AC. */
	range_b	**As Integer**	/* The input range for channel B. Use one of the following defined constants: GAGE_PM_5_V, GAGE_PM_2_V, GAGE_PM_1_V, GAGE_PM_500_MV, GAGE_PM_200_MV or GAGE_PM_100_MV. Remember that not all constants are available for all versions of hardware. See the table in GAGE_DRV.PAS where these constants are defined for an up to date listing. */
	couple_b	**As Integer**	/* The input coupling for channel B. Use one of the following defined ‘ constants: GAGE_DC or GAGE_AC. */
	source	**As Integer**	/* The trigger source for the next data acquisition. Use one of the following defined constants: GAGE_CHAN_A, GAGE_CHAN_B, GAGE_EXTERNAL or GAGE_SOFTWARE. */
	slope	**As Integer**	/* The trigger slope for the next data acquisition. Use one of the following defined constants: GAGE_POSITIVE or GAGE_NEGATIVE. */
	depth	**As Long**	/* The trigger or sample depth for the next data acquisition. Use one of the following defined constants: GAGE_POST_0K, GAGE_POST_128, GAGE_POST_256, GAGE_POST_512,

```
                          GAGE_POST_1K, GAGE_POST_2K, GAGE_POST_4K,
                          GAGE_POST_8K, GAGE_POST_16K, GAGE_POST_32K,
                          GAGE_POST_64K, GAGE_POST_128K, GAGE_POST_256K,
                          GAGE_POST_512K, GAGE_POST_1M, GAGE_POST_2M,
                          GAGE_POST_4M or GAGE_POST_8M.  These constants are the only
                          available sample depths for the CompuScope 220 series of boards.  The
                          CompuScope LITE can have the sample depth set to any modulo 16
                          value (0, 16, 32, 48, ..., max RAM).  The CompuScope 250 can have
                          the sample depth set to any modulo 64 value (0, 64, 128, 192, ... , max
                          RAM).  */
    level      As Integer /*  The trigger level for the external trigger input.  This value ranges
                          from 0 to 255.  0 volts trigger is actually 128 and all other values for the
                          voltage range used are scaled accordingly.  */
    range_e    As Integer /*  The input range for the external trigger input.  Use one of  the
                          following defined constants: GAGE_PM_5_V or GAGE_PM_1_V.
                          NOTE: GAGE_PM_5_V is not available on the CompuScope LITE.*/
    couple_e   As Integer /*  The input coupling for the external trigger input.  Use one of  the
                          following defined constants: GAGE_DC or GAGE_AC.  NOTE:
                          GAGE_AC is not available on the CompuScope LITE.  */
End Type
```

The irtype structure is not currently used by the sample program, but is there for future use.

```
Type Irtype
    constant   As Integer ’Value of the driver constant for the input range.
    gf_flag    As Long    ‘Flag indicates which boards support which gain.
    gf_calc    As Double  ‘Multipliers used for display voltage amplitude.
    gf_text    As String  ‘Text associated with the input range.
End Type
```

The sample rate table structure is not needed, however it is included in the "GAGE_DRV.BAS" file and initialized in the "Initialize" subroutine.

```
Type srtype
    rate       As Integer ‘ Rate used for setting the CompuScope.
    mult       As Integer ‘ Multiplier used for setting the CompuScope.
    sr_flag    As Integer ‘ Flag to indicate which board supports which sample rate.
    sr_calc    As Single  ‘ Time between samples (in ns).  Used in calculating the number of
                          ‘ points.
    text       As String  ‘ Text associated with the current settings of the CompuScope.
End Type
```

Global gage_board_location(GAGE_B_L_BUFFER_SIZE) As Integer

Global srtable(40) As Srtype

Function Prototypes.

The following functions have been written to help speed up the development process for the programmer using the CompuScope DLL and the CompuScope Series of high speed data acquisition cards.

Sub SetDefaultBoardLocation (ByVal seg As Integer, ByVal ind As Integer)

This routine is called if the DLL routine "gage_read_config_file" returns false indicating the board location configuration file is corrupted or missing. A default segment and index are passed to this routine and the global data structure "gage_board_location" is updated prior to calling the DLL routine "gage_driver_initialize".

function InitBoard () As Integer

This routine does all the necessary calls in the proper order to initialize the driver. Some modification to this routine will be necessary if the programmer does not want the progression messages displayed. As written, the sample program will abort if no CompuScope board is found or if "gage_select_board" fails. The probable cause of this is an incorrect segment or index value in the configuration file. The problem can be corrected by running the "GSINST" utility. The routine also converts two constants, defined in "GAGE_DRV.BAS", which represent the name and memory size of the CompuScope board into their text equivalents.

Sub SetBoard ()

This routine uses the "boarddef" structure named "board" which was previously loaded with the desired capture parameters. The sample program initializes these parameters statically, it is the programmers responsibility to perform the same task prior to capturing data on the CompuScope card. Note the order that these routines are called as the ordering is important for the proper operation of the CompuScope hardware and for the CompuScope DLL driver to maintain correct information as to the size of memory available to each channel etceteras. Any errors that occur while setting the capture mode or input control are reported from this routine.

Sub InitMemPtrs

This routine is used to create the appropriate addresses of the signal just captured. It should help the programmer understand how data is stored in the CompuScope cards and how to retrieve it. Also, it will require slight adjustments if multiple boards are to be supported.

Sub AcquireStart
Sub AcquireCheck
Sub Abort

These three routines are closely coupled to perform the basic data capture sequences. Study these routines to determine how they can be modified to perform the desired task for the application under development.
The "AcquireStart" routine initializes the hardware and starts the data acquisition process and issues a software trigger if required.
The "AcquireCheck" routine checks the status bits in the hardware to determine when the trigger has been received and the data acquisition has been completed, the memory pointers are then initialized and the re-display is requested. If the ContinuousMode flag was set then the "AcquireStart" routine is called again to repeat the cycle. Note that in the sample program, "AcquireCheck " is called whenever a timer event occurs. The C sample program, GSDLLDEM.C, demonstrates the different technique of placing "AcquireCheck" in the windows message loop.
The "Abort" routine disables the ContinuousMode, if it is set, which will cause the hardware to stop capturing data the next time data acquisition has been completed. If the ContinuousMode was not set then the hardware is aborted directly.

Extra Support.

Interpolated Trigger support routines.

Included with the sample programs on the driver diskettes are several support routines for interpolated trigger. These routines are designed to be used with the driver calls **gage_get_interpolate_trigger** and **gage_reset_interpolate_trigger** and are provided to help in the use of these routines. They can be found in the files **INTERPOL.C** and **INTERPOL.H** for the C sample programs, **INTERPOL.PAS** for the Pascal programs, the module **INTERPOL.BAS** for Visual Basic, and as subroutines in the Quick Basic program **TEST.BAS**.

init_interpolate_trigger_channel

This routine initiates the interpolated trigger and checks that the right interpolated trigger channel is set. If the trigger source is GAGE_SOFTWARE, interpolated trigger is turned off. It also sets a default setting for the interpolated trigger channel.

verify_interpolate_trigger

This routine verifies that there are no invalid settings for the interpolated trigger. It checks to see that there the number of boards and the trigger source are valid values and sets the interpolate trigger flag and interpolate trigger channel appropriately.

init_interpolate_trigger

This routine sets the interpolated trigger channel according to the current parameters. If the interpolate_trigger flag is not set, interpolated trigger is turned off. Otherwise, if the trigger source is channel A or channel B in dual channel mode, then this becomes the interpolated trigger channel. If the trigger source is external, the interpolated trigger channel must be chosen to be either channel A or channel B. Interpolated trigger is turned on or off by a call to **gage_reset_interpolate_trigger**.

init_memory_pointers

This routine gets the address pointers for the interpolated trigger channel by calling the driver routine **gage_calculate_addresses**. It then calls **gage_get_interpolate_trigger** with the first parameter set to 1 to calculate the ideal interpolated address for that channel. **init_chan_memory_pointers** is then called to find the proper interpolated trigger address for the rest of the active channels.

init_chan_memory_pointers

The interpolated trigger address for the rest of the active hardware channels is determined by this routine. First **gage_calculate_addresses** is called to find the address pointers for the channel and then **gage_get_interpolate_trigger** is called with its first parameter set to 0 to find the interpolated trigger address based on the chosen interpolate trigger channel.

Note that these routines were written to support multiple boards. If you have only one CompuScope card and therefore only two hardware channels, the driver routines **gage_calculate_addresses** and **gage_get_interpolate_trigger** (with parameter 1 set to 1) can be called with the interpolated trigger channel and then called again (with the first parameter of **gage_get_interpolate_trigger** set to 0) for the other channel. See the example programs on the distribution diskette for the proper use of these routines.

The DISKFILE.H GageScope signal file header definition.

Included with the GageScope driver disk is the DISKFILE.H header file that defines the contents of the 512 byte block of information that starts each GageScope data file. The header file can easily be ported to Pascal or Basic. This will allow the application programmer to quickly create a program that can extract information from GageScope data files for special purposes. Also, when data is stored in the GageScope data file format the signal can be retrieved by the GageScope program. Remember that the GageScope data file must have a multiple of 4096 bytes of data for dual channel acquisitions and 8192 bytes for single channel files (8192 and 16384 bytes for the 12/16 bit boards) plus the 512 byte header. The DISKFILE.H header file is reproduced here for reference only. This file is subject to change and the most recent version will be on the distribution disk for the CompuScope drivers. Note, the file version should be checked when reading GageScope data files to ensure that your application program can support the current data file format. All changes to this file are added to the end, thereby reducing possible future compatibility issues.

```
/************************************\
*    DISKFILE.H                Version 2.80.  *
*    WRITTEN FOR BC31, MSC70, WC90. *
*    LAST UPDATE:                    94/11/03. *
\*************************************/

/**************************************************************************\
*    DISK FILE HEADER FOR ROUTINES TO TRANSFER TRACES TO / FROM DISK.*
\**************************************************************************/

#define          DISK_FILE_HEADER_SIZE      512
#define          DISK_FILE_FILEVERSIZE      14
#define          DISK_FILE_CHANNAMESIZE     9         /* 8 + NULL.  */
#define          DISK_FILE_COMMENT_SIZE     256
#define          DISK_FILE_MISC_SIZE        8
#define          DISK_FILE_SYSTEM_SIZE      26        /*  Note: data split into three areas.*/
#define          DISK_FILE_CHANNEL_SIZE     30        /*  Note: data split into three areas.  */
#define          DISK_FILE_DISPLAY_SIZE     12        /*  Note: data split into two areas.  */
#define          DISK_FILE_HEADER_PAD       (DISK_FILE_HEADER_SIZE -
DISK_FILE_FILEVERSIZE + DISK_FILE_CHANNAMESIZE + DISK_FILE_COMMENT_SIZE +
DISK_FILE_MISC_SIZE + DISK_FILE_SYSTEM_SIZE + DISK_FILE_CHANNEL_SIZE +
DISK_FILE_DISPLAY_SIZE))

typedef struct  {
    char         file_version[DISK_FILE_FILEVERSIZE];
    int16        crlf1;                                   /*DISK_FILE_MISC_SIZE*/
    char         name[DISK_FILE_CHANNAMESIZE];
    int16        crlf2;                                   /*DISK_FILE_MISC_SIZE*/
    char         comment[DISK_FILE_COMMENT_SIZE];
    int16        crlf3;                                   /*DISK_FILE_MISC_SIZE*/
    int16        control_z;                               /*DISK_FILE_MISC_SIZE*/
    int16        sample_rate_index;                       /*DISK_FILE_SYSTEM_SIZE*/
    int16        operation_mode;                          /*DISK_FILE_SYSTEM_SIZE*/
    int32        trigger_depth;                           /*DISK_FILE_SYSTEM_SIZE*/
    int16        trigger_slope;                           /*DISK_FILE_SYSTEM_SIZE*/
    int16        trigger_source;                          /*DISK_FILE_SYSTEM_SIZE*/
    int16        trigger_level;                           /*DISK_FILE_SYSTEM_SIZE*/
    int32        sample_depth;                            /*DISK_FILE_SYSTEM_SIZE*/
    int16        captured_gain;                  /*DISK_FILE_CHANNEL_SIZE   */
    int16        captured_coupling;              /*DISK_FILE_CHANNEL_SIZE   */
```

```
        int32        current_mem_ptr;                  /*DISK_FILE_CHANNEL_SIZE   */
        int32        starting_address;                 /*DISK_FILE_CHANNEL_SIZE   */
        int32        trigger_address;                  /*DISK_FILE_CHANNEL_SIZE   */
        int32        ending_address;                   /*DISK_FILE_CHANNEL_SIZE   */
        uInt16       trigger_time;                     /*DISK_FILE_CHANNEL_SIZE   */
        uInt16       trigger_date;                     /*DISK_FILE_CHANNEL_SIZE   */
        int16        trigger_coupling;                 /*DISK_FILE_SYSTEM_SIZE    */
        int16        trigger_gain;                     /*DISK_FILE_SYSTEM_SIZE    */
        int16        probe;                            /*DISK_FILE_CHANNEL_SIZE   */
        int16        inverted_data;                    /*DISK_FILE_DISPLAY_SIZE   */
        uInt16       board_type;                       /*DISK_FILE_DISPLAY_SIZE   */
        int16        resolution_12_bits;               /*DISK_FILE_DISPLAY_SIZE   */
                                                       /*Used when saving 12-bit data.  */
        int16        multiple_record;                  /*DISK_FILE_SYSTEM_SIZE    */
        int16        trigger_probe;                    /*DISK_FILE_SYSTEM_SIZE    */
        int16        sample_offset;                    /*DISK_FILE_DISPLAY_SIZE   */
        int16        sample_resolution;                /*DISK_FILE_DISPLAY_SIZE   */
        int16        sample_bits;                      /*DISK_FILE_DISPLAY_SIZE   */
        uInt32       extended_trigger_time;            /*DISK_FILE_CHANNEL_SIZE   */
        uInt8        padding[DISK_FILE_HEADER_PAD];
} disk_file_header;

/*    End of DISKFILE.H.  */
```

Common Problems

Problem:

The sample programs provided on the diskette report "No CompuScope boards found.".

Answer:

The DLL looks for a configuration file, GAGESCOP.INC, in your Windows directory (usually C:\WINDOWS). The sample programs that use the DOS drivers expect the configuration file to be in the current directory. A common problem is using the configuration utility, GSINST.EXE, to write the configuration file and not copying it to your windows directory. This can be solved by copying it to the appropriate directory, by using GSINST.EXE with the command line parameter (GSINST -Fc:\windows\gagescop.inc) or using the windows based utility GSWINST.EXE. There may also be a memory conflict with a memory manager or another piece of hardware in your system. To check this, try running GSINST.EXE or GSWINST.EXE. If they do not report any errors, the sample programs should run. If they do, see Appendix 0, Resolving Memory Conflicts.

Problem:

The DLL sample programs find my CompuScope board, but I get a message saying memory failed.

Answer:

The first thing you should do is determine if the board operates under DOS by running either the GSINST utility and testing the memory of the board, or running GAGESCOP.EXE. It is possible that some memory segments will work with DOS but cause a conflict under Windows. Try changing the memory segment and / or index with GSINST.EXE. Using memory segment B000 will often cause this message if you are running in a DOS session under Windows because this segment is being used by Windows.

Problem:

Multiple record is behaving erratically in my program.

Answer:

The order in which the driver routines are called is important. **gage_multiple_record** should be called before **gage_capture_mode**, **gage_input_control** or **gage_trigger_control**. Also, if you change multiple record, **gage_trigger_control** must be called again.

Problem:

The program runs, but data capture is not correct.

Answer:

The order in which routines called for data capture is important. **gage_trigger_control** should be called after **gage_input_control**, which should be called after **gage_capture_mode**.

Problem:

I get linker errors when I try to compile a C++ program with the GAGE_DRV.H file.

Answer:

You may be using an older version of the GAGE_DRV.H file. If so, it can still be used by putting the following #ifdef around the function prototypes in GAGE_DRV.H.

```
#ifdef _cplusplus
        extern  "C" {
#endif
        function prototypes
                ...
                ...
#ifdef __cplusplus
        }
#endif
```

Problem:

I have the DLL source code, but it won't compile with my compiler.

Answer:

The DLL code has been compiled in the large memory model using Microsoft Quick C for Windows, Borland C 3.1, Borland C 4.0 and Microsoft Visual C++. When using the Borland compilers, turn off case sensitive link. When using Microsoft Visual C++, turn on the generate for __far functions in the Windows Prolog / Epilog part of the compiler options. Optimizations should be turned off. The DLL source should compile under any compiler capable of compiling Windows programs and dynamic link libraries. Also, if possible, you should use the project or make files supplied on the distribution diskettes, as these have all the right compiler options set.

Problem:

My program compiled and ran with a previous version of the driver, but won’t with the new version.

Answer:

You should recompile your program using the new definition file (GAGE_DRV.H for C, GAGE_DRV.PAS for Pascal or GAGE_DRV.BAS for Basic). These header files will reflect any changes to data structures or functions in the DLL or drivers which may affect your program. The definition files may also contain new values for the predefined constants your program uses. These predefined constants should be used in your program rather then using their numeric values to maintain compatibility with newer versions of the driver. Also note that the definition files for Pascal and Visual Basic have been renamed (to GAGE_DRV.PAS from GAGE.PAS and GAGE_DRV.BAS from GLOBAL.BAS). Any references to the old names, for example in the USES statement in Pascal, should be updated.

Problem:

My program sets **gage_start_capture** with the auto_trigger parameter set to TRUE but data capture doesn’t happen immediately.

Answer:

The trigger source must be set to GAGE_SOFTWARE, through a call to gage_trigger_control, for this parameter to have a predictable effect.

Problem:

I have the DLL source code and a CompuScope 220 and want to recompile the DLL using only the CS220 drivers to reduce the size.

Answer:

A header file, whichdrv.h, is provided on the diskette for this purpose. This file consists of several defines, (ie. #ifdef ALLOW_CS220_CODE). By default, the defines cause compilation of all the driver files. If you only want to compile with the CS220DRV.C file, change the defines for the other boards. Your project file will then consist of GAGE_DRV.C, GSWINDLL.C, GSWINSEL.OBJ, CS220DRV.C and GAGE_DRV.DEF and their associated header files. The same procedure is followed to compile for the other boards or to recompile the DOS drivers. Note that the project can contain the other source files for the drivers since the code for each “undefined” driver is not compiled.

Problem:

I’m using trigger view transfer in single channel mode. When I examine the captured data, I seem to be missing every other point.

Answer:

In single channel mode, the data is interleaved between the two channel of the CompuScope card. Therefore, trigger view transfer must be called once with each channel to transfer all the data. Then the first sample will be in channel A, the second in channel B, the third in A, etc.

See the section on Memory Organization of the CompuScope for more information. The sample routine, transfer_data, in the Sample Routines section shows one method of demultiplexing the captured data.

Problem:

I am having trouble reading a file saved in the GageScope signal file format from your GAGESCOP.EXE program.

Answer:

The signal section of a GageScope file is saved as a binary image of the CompuScope card's ram buffers. Reading the data is similar to reading a buffer returned from a call to gage_trigger_view_transfer. If the file was saved in single channel mode, the data will be interleaved between channel A and channel B. The usual method of reading a file is to get the trigger address, trigger depth and sample depth from the header section of the file. Note that you must take into account the current available memory depth of the card to avoid wrapping around the circular buffer if the file was saved using the Save All option in GAGESCOPE. This value is stored as the sample depth field in the signal file header. If the file was saved using the Normal or User Defined options, the signal section of the file will be normalized. You can tell if this was the case by the following method. If the starting address < = trigger address < = ending address, the file is normalized. See the sections on Memory Organization of the CompuScope, Extra support and the GageScope signal file format for more information on reading GageScope signal files.

Problem:

I am using trigger view transfer and want to transfer data beginning at a point other then the trigger point.

Answer:

You can use the function gage_set_trigger_view_offset to set an offset from the trigger address at which you wish to begin transferring data. A negative value can be used to include pre-trigger data. In this case, care must be taken not to go beyond the starting address or the data will be invalid. This typically manifests itself as a discontinuity in the captured waveform.

Problem:

I cannot run my application program in the Quick Basic environment.

Answer:

You can increase the memory available to application programs running under Quick Basic by using the SETMEM function. See the sample program, SIMPLE.BAS, and the section on using the Quick Basic drivers to see how this is done.

Appendix A: Recompiling the CompuScope Drivers.

The C source code for the CompuScope drivers and DLL is provided both as a source of information and for those who wish to modify the drivers to suit their own needs. The same set of core source files is used to build all the drivers. Any differences in rebuilding specific drivers will be noted in this section. The drivers have been tested under Microsoft C 7.0, Visual C++ 1.0, Visual C++ 1.5, Borland C 3.1 and Borland C 4.0. They have also been compiled using the Watcom 32 bit C compiler, version 9.0 and 10.0. The source code and the header file, WHICHDRV.H, contain various conditional compiler defines which take effect depending on which drivers you are using and whether you are compiling for DOS or Windows. The drivers for all languages are compiled using the large model, byte-aligned and using the Pascal calling convention.

Note: You should set the identifier length to be at least 43. In Borland C, this options is located in the Options | Compiler | Source dialog box.

Because the C drivers are in the form of object files that are linked in with your application program, recompiling them is simply a matter of rebuilding your project or makefile. Programs using the CompuScope C drivers should generally be compiled in the large model. The compiler should be set to make structures byte-aligned. The files that make up the core of the C drivers are:

GAGE_DRV.C
CS112DRV.C
CS250DRV.C
CS220DRV.C
CSLITDRV.C
TIMERS.ASM (For the Watcom compiler, use TIMERS.C)

and their associated header files. Project files (for Borland C) and batch files to rebuild the sample programs are provided on the distribution diskette. Note that some of the path and executable file names in these files may have to be changed to conform to your particular environment.

To compile the drivers into a Quick Basic library, you must be using Microsoft C 5.1. A batch file, GAGEQLIB.BAT, is provided on the distribution diskette to aid in recompiling the drivers. Some of the path names in the batch file may have to be changed to reflect your own environment. The batch file will create both the GAGE_DRV.QLB and the GAGE_DRV.LIB files. The GAGE_DRV.QLB file is needed to create programs using the Quick Basic environment and the GAGE_DRV.LIB file is needed to use the command line compiler. The files needed to recreate the Quick Basic library are:

GAGE_DRV.C
CS112DRV.C
CS250DRV.C
CS220DRV.C
CSLITDRV.C
TIMERS.OBJ
GAGE_FIX.LIB

and their associated header files. GAGE_FIX.LIB is a fix-up library that must be created from the Microsoft C 5.1 compiler run-time libraries. Two batch files, XTRACFIX.BAT and FIXUPLIB.BAT, are provided for this purpose.

To recompile the DLL, you can use the project or makefile that is on the distribution diskette. The DLL should be compiled in the large model, with one byte structure alignment. The compiler will detect if you are compiling under Windows or DOS and turn on the appropriate conditional defines to compile the source code either for Windows or DOS. This compiler directive is defined in the file WHICHDRV.H. If you are using the Microsoft compilers, you should turn on the Generate for _far functions option in the Windows Prolog / Epilog section of the compiler options. The files needed to recreate the DLL are:

GAGE_DRV.C	(Driver supervisor file)
CS112DRV.C	(Support for CS6012 and CS1012)
CS250DRV.C	(Support for CS250 and CS225)
CS220DRV.C	(Support for CS220)
CSLITDRV.C	(Support for CSLITE)
TIMERS.ASM	
GSWINSEL.ASM	
GSWINDLL.C	
GAGE_DRV.DEF	

and their associated header files. A Microsoft Visual C makefile, GAGE_DRV.MAK, and a Borland C project file, GAGE_DRV.PRJ, are provided for your convenience. The object files are provided for those who do not have an assembler.

If you are using Microsoft Visual C and the Visual Workbench rescans the include dependencies for the project (for example, if you change the directory in which the files are located), you will get compile errors such as cannot find file \gage\gscope\gage_drv.h, etc. This because of some conditional compiler directives that are at the top of the source files. Because Visual C scans the include dependencies before compilation, it doesn’t know not to include them. This can be remedied by either editing the resulting makefile by hand and removing all references to include files located in \gage\gscope\, or by removing the relevant #ifdef sections from the top of the driver source files.

The DLL built for Protected Mode Pascal uses a modified version of the Borland runtime library to prevent the loading of the WINEM87.DLL if the program is running in protected mode and allow it if the program is running under Windows. This can be done by assembling the modified version of the file FPINIT.ASM on the distribution diskettes and using TLIB.EXE to replace the existing FPINIT.OBJ located in the CWL.LIB file with the newer version. An alternative is to rebuild the Borland Windows run-time libraries using the new version of FPINIT.ASM. The DLL can then be recompiled using the new runtime library file. You should back up the original version of the library first. Note that this has only been tested using Borland C version 3.1. The modified version of the DLL will work with either Windows or protected mode applications. See More information on rebuilding the library to make the modified DLL and the files needed to do so can be found on the distribution diskette. See PC MAGAZINE, Volume 13, Number 4 (February 22, 1994) for a more complete discussion of this technique.

Appendix B: Customizing the CompuScope Drivers.

The design of the CompuScope drivers allows the application programmer the flexibility of supporting more than one type of CompuScope hardware by specifying which CompuScope driver modules are to be compiled and included in your project. All of the driver files should be located in the same directory.

The WHICHDRV.H file needs to be edited to allow the compiler to correctly combine the different driver modules. The contents of this file specify which drivers are to be included in the project. The CompuScope Drivers are shipped with all CompuScope drivers enabled as illustrated in the copy of the WHICHDRV.H file listed below.

WHICHDRV.H for all versions CompuScope hardware.

```
/*****************************************\
*   WHICHDRV.H          VERSION 2.70.          *
*   WRITTEN FOR BORLAND C++ V.3.1              *
*   COPYRIGHT (C) GAGE APPLIED                 *
*   SCIENCES INC.     JUNE, 1994.              *
*   LAST UPDATE:                94/06/22.      *
\*****************************************/

#define  ALLOW_CSLITE_CODE

#define  ALLOW_CS250_CODE

#define  ALLOW_CS220_CODE

#define  ALLOW_CS1012_CODE

#define  nALLOW_CSDEMO_CODE

/*         End of WHICHDRV.H.  */
```

The file can be easily changed to allow only the CompuScope LITE board as shown below.

WHICHDRV.H for only the CompuScope LITE hardware.

```
/*****************************************\
*   WHICHDRV.H          VERSION 2.70.          *
*   WRITTEN FOR BORLAND C++ V.3.1              *
*   COPYRIGHT (C) GAGE APPLIED                 *
*   SCIENCES INC.     JUNE, 1994.              *
*   LAST UPDATE:                94/06/22.      *
\*****************************************/

#define  ALLOW_CSLITE_CODE

#define  nALLOW_CS250_CODE

#define  nALLOW_CS220_CODE

#define  nALLOW_CS1012_CODE
```

```
#define  nALLOW_CSDEMO_CODE

/*       End of WHICHDRV.H.  */
```

Similarly, the file could be changed to allow only the driver code for two boards to be included in the project by “allowing” the required boards. The following example will generate code to support the CompuScope 250 and CompuScope 6012/1012 boards.

WHICHDRV.H for the CompuScope 250 and 6012/1012.

```
/******************************************\
*  WHICHDRV.H          VERSION 2.70.        *
*  WRITTEN FOR BORLAND C++ V.3.1            *
*  COPYRIGHT (C) GAGE APPLIED               *
*  SCIENCES INC.    JUNE, 1994.             *
*  LAST UPDATE:              94/06/22.      *
\******************************************/

#define  nALLOW_CSLITE_CODE

#define  ALLOW_CS250_CODE

#define  nALLOW_CS220_CODE

#define  ALLOW_CS1012_CODE

#define  nALLOW_CSDEMO_CODE

/*       End of WHICHDRV.H.  */
```

The distrubution disk contains several batch files that automate some of these steps. The batch files LITE.BAT, 250.BAT, 220.BAT, 1012.BAT and GAGE.BAT will copy the WHICHDRV.LIT, WHICHDRV.250, WHICHDRV.220, WHICHDRV.112 and WHICHDRV.GAG files to the WHICHDRV.H file and update the file time stamp so that the driver will be correctly compiled.

The project files do not need to be modified since the code for the other boards does not exist (conditionally not compiled). The project does include all of the source files but the resulting “non compiled” object files are so small that excluding them does not seem to be worth the effort.

Appendix C: Converting Borland C++ to Microsoft C.

GAGE_DRV.C and GAGE_DRV.H

The CompuScope C drivers have been written to as closely follow the ANSI C specification for the C language. The only change required for the driver to compile with the Microsoft compiler is the name of the memory allocation header file, MALLOC.H instead of ALLOC.H. The ioport routines have been named using the Microsoft convention and the names are substituted via a macro in the file header IO.H. The Borland C++ compiler defines a constant "__BORLANDC__" that is used to detect the Borland compiler and include the correct header file. The absence of the constant causes the driver files and the example programs to be compiled for the Microsoft header file names and graphics library.

The demo programs have been written to work with either the Borland or the Microsoft C compilers. All compiler dependent function calls are contained in another source file. The inclusion of the appropriate graphics header file and in the case of the Microsoft compiler, a set of function wrappers that change the names of the graphics routines to Borland compatible names use the "__BORLANDC__" constant definition to direct the compilation process. The graphics display functions were originally written to use the Borland BGI graphics library. The conversion from the Borland graphics to the Microsoft graphics is accomplished by including the file BC2MSC_G.H. The contents of this file, listed below, converts the Borland BGI names used in the sample programs to the Microsoft naming convention.

```
#define  CGA           _CGA
#define  CGAHI         _HRESBW
#define  MCGA          _MCGA
#define  MCGAHI        _VRES2COLOR
#define  HERCMONO      _HGC
#define  HERCMONOHI    _HERCMONO
#define  EGA           _EGA
#define  EGAHI         _ERESCOLOR
#define  VGA           _VGA
#define  VGAHI         _VRES16COLOR

#define  getcolor      _getcolor
#define  setcolor      _setcolor
#define  moveto        _moveto
#define  lineto        _lineto

#define  line(x1,y1,x2,y2)        {_moveto((x1),(y1));_lineto((x2),(y2))}
```

The two routines, start_graphics and stop_graphics, require changing the way the program initializes the graphics hardware (done automatically by the presence/absence of "__BORLANDC__"). The header file, BC2MSC_G.H, handles the remaining incompatibilities.

Appendix D: Converting Borland C++ to Turbo C.

The conversion to the older Borland Turbo C dialect is quite simple. The project files between these two versions of the compiler are not compatible, therefore the ASCII project files listed here will correctly compile the sample programs. Note that in both cases the header files have been omitted and all of the drivers are listed. It is up to the programmer to add the header files desired and to omit the drivers files that are not required.

GAGE_TST.PRJ for Borland Turbo C version 2.0.

```
TEST.C
SCREENS.C
CS112DRV.C
CS250DRV.C
CS220DRV.C
CSLITDRV.C
GAGE_DRV.C
TIMERS.OBJ
CGA.OBJ
EGAVGA.OBJ
HERC.OBJ
```

GAGE_A2D.PRJ for Borland Turbo C version 2.0.

```
ACQ2DISK.C
CS112DRV.C
CS250DRV.C
CS220DRV.C
CSLITDRV.C
GAGE_DRV.C
TIMERS.OBJ
```

Appendix E: CompuScope LabWindows / CVI Driver.

Preface.

The GageScope LabWindows / CVI driver consists of an instrument file, GAGE_DRV.FP, which allows C programs written in the National Instruments LabWindows / CVI environment to communicate with the GAGE_DRV.DLL distributed by Gage Applied Sciences. The driver supports multiple boards and board types, however from the programmer's point of view only one board is accessible at any one given time. The initialization routine reads a specially formatted array to determine where the user has installed the CompuScope card(s). It then tries to initialize each board, determine that it is indeed present and tests and sizes the memory on each board it finds. Another routine will read a binary disk file and initialize the special array or the user can create the array with the proper format and pass it to the initialization routine. A routine is provided to select the desired active board and then all subsequent operations are applied to the currently active board, from data capture to configuration set up.   All the gage driver routines and functions described in other sections of this are available under CVI. They can be accessed through the file GAGE_DRV.FP, which includes help documentation for each function along with its parameters. More complete documentation on each of these routines is available in other sections of this manual. Note that due to limitations in name length in LabWindows CVI, the routine **gage_calculate_multiple_record_addresses** is referred to in CVI as **gage_calculate_mr_addresses**. The driver also comes with two sample programs, which can be used as standalone applications or as a starting point for your own programs.

It is suggested you read the sections of this manual titled Memory Organization of the CompuScope and Application Development to understand basics of the CompuScope boards.

The drivers will work with all CompuScope boards. The sample project GAGE.PRJ will similarly work with any CompuScope board. The sample project GAGEMULT.PRJ will work with any CompuScope card which has multiple record ability. Note that multiple record is standard on some CompuScopes and a hardware option on others. It is not available for the CompuScope LITE.

To run any program using the Gage LabWindows / CVI driver, the files GAGE_DRV.DLL and GAGESCOP.INC (a configuration file created with the DOS based GSINST.EXE or windows based GSWINST.EXE utilities) must be located in your windows directory (usually C:\WINDOWS).

Sample Programs.

GAGE_DRV.PRJ

The sample project GAGE_DRV.PRJ is an instrument example that uses the GAGE_DRV.DLL. This program will work with any CompuScope board in single or master / slave configurations and features user changeable sample rate, mode, input and trigger controls. The project consists of the files:

GAGESCOP.C
GAGE_AUX.C
INTERPOL.C
STRUCTS.C
GAGE_DRV.FP
GAGESCOP.H
GAGE.UIR

The routines in GAGE_AUX.C will be discussed first as they are the ones that communicate with the GAGE_DRV.DLL.

AcquireStart	sets up the CompuScope card to start capturing data according to the settings in the user interface panel.
AcquireCheck	checks the state of the CompuScope card when it is acquiring data to see if it has triggered. If no trigger event has occurred within a specified time-out period, a trigger is forced with a call to **gage_forced_trigger_capture**. The routine then checks to see if the board is busy. When the board is no longer busy, data capture is complete.
Abort	this routine is called to abort the data capture if the board remains busy beyond a specified time-out period.
transfer_data	the transfer_data routine gets the valid addresses for each CompuScope board through a call to **gage_calculate_addresses**, transfers the data from the boards and displays it.
gage_read_single	gets one sample from the CompuScope board in single channel mode and calculates the voltage according to the selected input range.
gage_read_dual_a	gets one sample from channel A of the CompuScope in dual channel mode and calculates the voltage according to the selected input range.
gage_read_dual_b	gets one sample from channel B of the CompuScope in dual channel mode and calculates the voltage according to the selected input range.
ShowData	plots the samples acquired from the CompuScope board.
SetDefaultBoardLocation	sets up the initialization array 'gage_board_location' with the default values of segment 0xD000 and index 0x0200 if the configuration file GAGESCOP.INC is not found.
BoardTypeSizeToText	converts the GAGE size and type constants to text.

InitBoard	finds and reads the GAGESCOP.INC configuration file, initializes the driver and prepares the boards to start data capture.
SetBoard	sets the board(s) current capture, input and trigger parameters.
InitMemPtrs	is used by transfer_data to get the valid addresses of the most recent data acquisition.

The file GAGESCOP.C contains the routines that interact with the user interface panel file GAGE.UIR.

DataFunc	handles all input from the user interface.
UpdateBoardSettings	validates all the settings in the struct 'NewBoard' and stores the accepted data in the struct 'board'.
SetPanel	sets all the user interface controls according to the values in the current 'board'.
ReadPanel	reads all the user interface controls and stores the new values in a copy of the 'board' struct called NewBoard. These settings are later validated by a call to UpdateBoardSettings.
InitDepthTable	initializes the user interface depth table. This table shows the available trigger depths for the installed CompuScope board(s).
InitRateTable	initializes the user interface rate table, which shows the allowable sample rates for the installed CompuScope board(s). If an invalid sample rate is chosen, depending on the operating mode, the maximum setting allowed will be used.

The file INTERPOL.C has some support routines for interpolated trigger. These routines are described in the section of the driver manual entitled Extra Support. The STRUCTS.C file contains some structures and their initialization that are used by the sample programs.

GAGEMULT.PRJ

The sample project GAGE_MULT.PRJ makes use of the multiple record feature available on some CompuScope boards. The front panel is similar to that of the GAGE.PRJ sample program. A new control is added which lets the user choose the size of the multiple record group. The number of such groups available is dependent on the memory depth of the installed CompuScope board(s). Since most of the routines and variables used in the GAGE.PRJ program are also used here, only the routines that differ will be discussed. They can be found in the file GAGEMULT.C.

AcquireCheckMulRec	polls the CompuScope cards for multiple record acquisition. It checks to see if anything on the front panel has been changed. The routine also checks the state of the CompuScope to see if it has triggered. If not it forces a trigger with a call to **gage_forced_trigger_capture.** The driver routine **gage_busy** is also monitored to see when data acquisition is complete. The routine also loops through all the acquired groups, calling transfer_multiple_record_data.

transfer_multiple_records_data gets the valid addresses of the captured data for each record by calling **gage_calculate_mr_addresses**, transfers the data from the CompuScope board and calls ShowMultipleRecordsData to display it.

ShowMultipleRecordsData plots the samples acquired from the current CompuScope board one record at a time.

SetBoardMultiple sets the current capture, input and trigger parameters for each board in multiple record mode.

Rebuilding the Driver.

To rebuild the GAGE_DRV.FP instrument, you need to use the file, GAGE_DRV.H, provided on your distribution diskette. You will also require the Watcom 32 bit C compiler version 9.0 or higher. From the GAGE_DRV.H file, LabWindows / CVI can generate the “glue code” needed to use the 16 bit DLL (GAGE_DRV.DLL) in the 32 bit LabWindows environment. This special version of the GAGE_DRV.H file differs from the one used to build the 16 bit GAGE_DRV.DLL in the following ways:

> LPSTR is not recognized under LabWindows / CVI and must be declared as unsigned char *.
>
> The keywords far and FAR are not recognized in the LabWindows environment.
>
> Type ‘int’ is a 16 bit integer to the DLL and should be defined as either int16 or short to avoid conflicts with the 32 bit integers used in LabWindows.
>
> Pointers to structs are not allowed in the “glue code”. To fix this, use void *. See the prototypes for the functions **gage_get_driver_info** and **gage_get_driver_info_structure** in GAGE_DRV.H for and example of this.

If you then choose the menu option “Generate Glue Code”, LabWindows / CVI will then generate the file GAGE_DRV.C. This file should then be compiled with the Watcom compiler using the following command line, which is also available as a batch file (watcomp.bat) on the distribution diskettes.

```
wcc386 -zw -s -4s -j -fpi87 -d0 -of gage_drv.c
```

This will create the object file, GAGE_DRV.OBJ, which will be associated with the file GAGE_DRV.DLL.

To recreate the instrument file, GAGE_DRV.FP, add GAGE_DRV.OBJ to the project files. Create a new “.FP”, calling it GAGE_DRV.FP. LabWindows / CVI will then associate it with GAGE_DRV.OBJ. Note that some pointer types, such as long * or struct *, are not available for specification in the FP definition utility, so these pointers must be defined as void *. In the instrument file on your distribution diskette, whenever a pointer is defined as void *, the help text indicates the specific type of pointer.

Appendix F: CS6012 / 1012 register functionality.

LIST OF I/O PORTS USED BY THE COMPUSCOPE 6012 / 1012

	<u>SYMBOL KEY</u>
+	Logic high level active.
-	Logic low level active.
(+)	Active on rising edge pulse.
(-)	Active on falling edge pulse.
(R)	Controls at least one relay with this signal Relays used take approximately 1 ms to activate.

Register 0: **CS1012_STATUS**

Bit 0:	BUSY: 1 = the card is converting data, 0 = the onboard memory buffers may be accessed.
Bit 1:	LSB Single Channel Trigger Address Bit +.
Bit 2:	TRIGGER: 1 = a trigger has occurred, 0 = Hardware is waiting for a trigger event.
Bit 3:	RAM_FULL: 1 = all of the on board RAM has been overwritten by the current data capture, 0 = some of the RAM contains invalid data.
Bit 4:	60 / 20 Mhz feedback: 0 = 20 Mhz clock set, 1 = 60 Mhz clock set. (See CS1012_INTERRUPT_CTRL, Bit 3.)
Bit 5:	INTERRUPT PENDING: 1 = an interrupt is pending or in progress, 0 = the interrupt control is idle.
Bit 6:	DATA_READY: 1 = data is ready to be read directly from the A/D converter..
Bit 7:	Trigger Address Bit 0 +.

Register 1: **CS1012_ENABLE_CONTROL**

Bit 0:	Channel A Ram Bank Enable: 1 = Capture, 0 = Protect Ram.
Bit 1:	Channel B Ram Bank Enable: 1 = Capture, 0 = Protect Ram.
Bit 2:	ABORT: (+), Toggle Bit With 0-1-0 sequence to abort data capture. 1 = busy signal is forced to low, 0 = busy signal will go low by normal means, if possible.
Bit 3:	START Local Clock: (+), 0-1-0 Bit Sequence To Start Board, hold high for CPU accesses. Therefore: 1 = CPU access and 0 = CS1012 access.
Bit 4:	BUSY: Force the busy signal to a high state. 1 = busy signal is forced to high, 0 = busy signal will go high by normal means, if possible.
Bit 5:	MEMORY TEST READ/WRITE: 1= write allowed when testing the memory, 0 = read allowed when testing the memory and also for normal operation.
Bit 6:	MEMORY TEST ENABLE: 1= testing of the memory is allowed - set this bit for both read and write cycles, 0 = normal operation.
Bit 7:	RESDAT1: Set to 0 and RESCLK, then set to 1 and RESCLK. This signal works with RESDAT0 to clear the RESET circuitry and to enable reads. See also RESDAT0 and RESCLK.

Register 2: **CS1012_CLOCK_CTRL**

Bit 0: RESDAT0: Set to 1 and RESCLK, then set to 0 and RESCLK. This signal works with RESDAT0 to clear the RESET circuitry and to enable reads. See also RESDAT0 and RESCLK

Bit 1: DIRECT ADC READ ENABLE: 0 = memory read/write, 1 = read last latched data from the A/D converter.

Bit 2: CLOCK SELECT 0: Select the sample rate to be used. These settings also affect the

Bit 3: CLOCK SELECT 1: CLKAD signal and the CLKLATCH signal.

Bit 4: CLOCK SELECT 2:

CLKSEL0-2	SINGLE CHAN	DUAL CHAN	
000 =	20 MHZ	10 MHZ	
001 =	10 MHZ	5 MHZ	
010 =	5 MHZ	2.5 MHZ	
011 =	4 MHZ	2 MHZ	
100 =	>= 40 KHZ	>= 20 KHZ	TIMER 0
101 =	<= 20 KHZ	<= 10 KHZ	TIMER 0/1
110 =	FREEZE	FREEZE	
111 =	INTERNAL 20 MHZ OSCILLATOR (Used for calibration and initialization.)		

Bit 5: DOUBLE RATE: 1 = dual channel capture mode, 0 = "unauthorized" single channel capture mode.

Bit 6: DIRECT ADC READ CLOCK: 0 = disable read, 1 = read last latched data from the A/D converter.

Bit 7: SELECT CHANNEL: 0 = channel A, 1 = channel B.

Register 3: **CS1012_TRIGGER_CTRL**

Bit 0: TRIGGER 1 ENABLE: allows trigger events to be recognized from the first trigger circuit. 1 = enabled, 0 = disabled.

Bit 1: TRIGGER 2 ENABLE: allows trigger events to be recognized from the second trigger circuit. 1 = enabled, 0 = disabled.

Bit 2: TRIGGER 1 SLOPE: 1 = positive, 0 = negative.

Bit 3: TRIGGER 2 SLOPE: 1 = positive, 0 = negative.

Bit 4: TRIGGER CLEAR: 1 = clears trigger and prevents triggering, 0 = enables complete trigger circuitry. This is a "master enable".

Bit 5: TRIGGER TO BUS: 1 = enable sending the trigger signal from this card to the master/slave trigger bus, 0 = disable this feature.

Bit 6: TRIGGER FROM BUS: 1 = enable receiving the trigger signal for this card from the master/slave trigger bus, 0 = disable this feature.

Bit 7: SOFTWARE TRIGGER: there is no mode or source to set. (+), 0-1-0 bit sequence to start the trigger event immediately.

Register 4: **CS1012_INTERRUPT_CTRL**

Bit 0: MEMORY CLOCK ENABLE: interrupt enable for the memory clock.

Bit 1: BUSY ENABLE: interrupt enable for trapping the busy signal.

Bit 2: TRIGGER ENABLE: interrupt enable for trapping the trigger event signal.

Bit 3: 60 / 20 MHZ: 0 = 20 Mhz Clock, 1 = 60 Mhz Clock. (See CS1012_STATUS, Bit 4).

Bit 4: MEMORY FULL ENABLE: interrupt enable for when the memory becomes full.

Bit 5: INTERRUPT ENABLE: 1 = enable the generation of interrupts, 0 = disable interrupts.

Bit 6: INTERRUPT CLEAR: 1 = clear any pending interrupts, 0 = normal operation.

Bit 7: GET DATA: (-), 1-0-1 Bit Sequence To Capture.

Register 5:	**CS1012_INPUT_CONFIG**
Bit 0:	ATTENUATE A: 1= channel A input divided by 2.5, 0 = channel A input times 1 (R).
Bit 1:	ATTENUATE B: 1= channel B input divided by 2.5, 0 = channel B input times 1 (R).
Bit 2:	SELECT B DATA: 1 = B bank gets chan B data, 0 = B bank gets chan A data (R).
Bit 3:	EXTERNAL TRIGGER COUPLING: 1 = AC, 0 = DC (R).
Bit 4:	CHANNEL B COUPLING: 1 = AC, 0 = DC (R).
Bit 5:	CHANNEL B CALIBRATE: 1 = calibrate source input, 0 = BNC source input (R).
Bit 6:	CHANNEL A COUPLING: 1 = AC, 0 = DC (R).
Bit 7:	CHANNEL A CALIBRATE: 1 = calibrate source input, 0 = BNC source input (R).

Register 6:	**CS1012_DAC**
Bit 0:	! DAC FRAME synchronization bit.
Bit 1:	! DAC CLK DATA in latch on falling edge.
Bit 2:	DAC SERIAL DATA in MSB first (16 bit).
Bit 3:	DAC ADDRESS 0, must match data bit 3.
Bit 4:	!UPDATE ALL DAC simultaneously when low.
Bit 5:	!CLEAR (ZERO) ALL DAC when low.
Bit 6:	RESCLK: reset clock for RESDAT0/1 to reset the memory controls. See also RESDAT0 and RESDAT1.
Bit 7:	NEED_RAM connects the CPU to the memory. 1 = CPU access, 0 = CS1012 access.

Register 7:	**CS1012_TRIGGER_ADDRESS_0**
Bit 0:	Trigger Address Bit 1 +.
Bit 1:	Trigger Address Bit 2 +.
Bit 2:	Trigger Address Bit 3 +.
Bit 3:	Trigger Address Bit 4 +.
Bit 4:	Trigger Address Bit 5 +.
Bit 5:	Trigger Address Bit 6 +.
Bit 6:	Trigger Address Bit 7 +.
Bit 7:	Trigger Address Bit 8 +.

Register 8:	**CS1012_TRIGGER_ADDRESS_1**
Bit 0:	Trigger Address Bit 9 +.
Bit 1:	Trigger Address Bit 10 +.
Bit 2:	Trigger Address Bit 11 +.
Bit 3:	Trigger Address Bit 12 +.
Bit 4:	Trigger Address Bit 13 +.
Bit 5:	Trigger Address Bit 14 +.
Bit 6:	Trigger Address Bit 15 +.
Bit 7:	Trigger Address Bit 16 +.

Register 9:	**CS1012_TRIGGER_ADDRESS_2**
Bit 0:	Trigger Address Bit 17 +.
Bit 1:	Trigger Address Bit 18 +.
Bit 2:	Trigger Address Bit 19 +.
Bit 3:	Trigger Address Bit 20 +.
Bit 4:	Trigger Address Bit 21 +.
Bit 5:	Trigger Source 2: 1 = Trigger 2 triggered, 0 = Trigger 2 clear.
Bit 6:	Trigger Source 1: 1 = Trigger 1 triggered, 0 = Trigger 1 clear.
Bit 7:	Reserved for future use.
	NOTE: if both Trigger Source 1 and Trigger Source 2 are clear and a trigger has occurred then the trigger source is from the trigger bus.

Register A: **CS1012_SEGMENT_LOW**

Bit 0:	Memory Segment Bit 0, Memory Address Bit 12: This bit must always be set to 1.
Bit 1:	Memory Segment Bit 1, Memory Address Bit 13: +.
Bit 2:	Memory Segment Bit 2, Memory Address Bit 14: +.
Bit 3:	Memory Segment Bit 3, Memory Address Bit 15: +.
Bit 4:	TRIGGER 1 source selection bit 0. 00 = Channel A.
Bit 5:	TRIGGER 1 source selection bit 1. 01 = Channel B.
Bit 6:	TRIGGER 2 source selection bit 0. 10 = External Trigger - Times 1.
Bit 7:	TRIGGER 2 source selection bit 1. 11 = External Trigger - Divide by 5.

Register B: **CS1012_SEGMENT_HIGH**

Bit 0:	Memory Segment Bit 4, Memory Address Bit 16: +.
Bit 1:	Memory Segment Bit 5, Memory Address Bit 17: +.
Bit 2:	Memory Segment Bit 6, Memory Address Bit 18: +.
Bit 3:	Memory Segment Bit 7, Memory Address Bit 19: +.
Bit 4:	Memory Segment Bit 8, Memory Address Bit 20: +.
Bit 5:	Memory Segment Bit 9, Memory Address Bit 21: +.
Bit 6:	Memory Segment Bit 10, Memory Address Bit 22: +.
Bit 7:	Memory Segment Bit 11, Memory Address Bit 23: +.

Register C: **CS1012_BLOCK_LOW**

Bit 0:	Not Connected.
Bit 1:	Not Connected.
Bit 2:	Not Connected.
Bit 3:	Block Bit 0 +.
Bit 4:	Block Bit 1 +.
Bit 5:	Block Bit 2 +.
Bit 6:	Block Bit 3 +.
Bit 7:	Block Bit 4 +.

Register D: **CS1012_BLOCK_HIGH**

Bit 0:	Block Bit 5 +.
Bit 1:	Block Bit 6 +.
Bit 2:	Block Bit 7 +.
Bit 3:	Block Bit 8 +.
Bit 4:	Block Bit 9 +.
Bit 5:	Block Bit 10 + (Not Used!).
Bit 6:	Block Bit 11 + (Not Used!).
Bit 7:	Block Bit 12 + (Not Used!).

Register E: **CS1012_TIMER_0**

Bit 0:	D0 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 1:	D1 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 2:	D2 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 3:	D3 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 4:	D4 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 5:	D5 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 6:	D6 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 7:	D7 of 8254 Timer 0: Sample Rate Selection Divider +.

Register 1E:	**CS1012_TIMER_1**
Bit 0:	D0 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 1:	D1 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 2:	D2 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 3:	D3 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 4:	D4 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 5:	D5 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 6:	D6 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 7:	D7 of 8254 Timer 1: Sample Rate Selection Divider +.

Register 2E:	**CS1012_TIMER_2**
Bit 0:	D0 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 1:	D1 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 2:	D2 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 3:	D3 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 4:	D4 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 5:	D5 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 6:	D6 of 8254 Timer 2: Post Trigger Depth Selection +.
Bit 7:	D7 of 8254 Timer 2: Post Trigger Depth Selection +.

Register 3E:	**CS1012_TIMER_CONTROL**
Bit 0:	BCD: 0 = Binary Counter (BCD Mode Not Used).
Bit 1:	M0: Mode Selection: Only Modes 3 + 5 Required.
Bit 2:	M1: Mode 3: Square Wave (Timers 0 + 1).
Bit 3:	M2: Mode 5: Hardware Trigger Strobe (Timer 2).
Bit 4:	RW0: Read/Write Control: 11 = Write LSB First and MSB
Bit 5:	RW1: Second, Only Read/Write mode required.
Bit 6:	SC0: Select Counter: 00 = Timer 0, 01 = Timer 1,
Bit 7:	SC1: 10 = Timer 2.

Register 0F:	**CS1012_NOT_USED**
Bit 0:	No Connection.
Bit 1:	No Connection.
Bit 2:	No Connection.
Bit 3:	No Connection.
Bit 4:	No Connection.
Bit 5:	No Connection.
Bit 6:	No Connection.
Bit 7:	No Connection.

Register 40:	**CS1012 does not decode address register bits A6 and A7, however, A6 (register 40) is reserved for future use.**
Register 80:	**CS1012 does not decode address register bits A6 and A7, however, A7 (register 80) is currently used for the multiple record or trigger stacking enable bit.**

Registers 10 to 1D, 1F to 2D, 2F to 3D, 3F, 41 to 7F and 81 to FF

The CS1012 does not decode address register bits A6 and A7 and uses A4 and A5 as the 8254's A0 and A1. Therefore these registers are duplicates of the registers listed above and are not individually decoded and the use of these locations should be avoided to prevent strange operation of the CS1012 hardware.

Appendix G: CS250 / 225 register functionality.

LIST OF I/O PORTS USED BY THE COMPUSCOPE 250 / 225

	SYMBOL KEY
+	Logic high level active.
-	Logic low level active.
(+)	Active on rising edge pulse.
(-)	Active on falling edge pulse.
(R)	Controls at least one relay with this signal Relays used take approximately 1 ms to activate.

Register 0: **CS250_STATUS_REG**
Bit 0: Busy: 1 = Converting Data, 0 = Memory can be accessed.
Bit 1: Ram Full: 1 = Ram Is Full, 0 = Ram Is Not Full.
Bit 2: Trigger Received: 1 = Trigger Occurred, 0 = No Trigger.
Bit 3: Get Data Acknowledge: 0 = Acknowledged, 1 = Waiting.
Bit 4: No Connection.
Bit 5: No Connection.
Bit 6: Trigger Address Bit 0 + (A/B).
Bit 7: Trigger Address Bit 1 + (Bank).

Register 1: **CS250_TRIG_ADDR_0**
Bit 0: Trigger Address Bit 2 +.
Bit 1: Trigger Address Bit 3 +.
Bit 2: Trigger Address Bit 4 +.
Bit 3: Trigger Address Bit 5 +.
Bit 4: Trigger Address Bit 6 +.
Bit 5: Trigger Address Bit 7 +.
Bit 6: Trigger Address Bit 8 +.
Bit 7: Trigger Address Bit 9 +.

Register 2: **CS250_TRIG_ADDR_1**
Bit 0: Trigger Address Bit 10 +.
Bit 1: Trigger Address Bit 11 +.
Bit 2: Trigger Address Bit 12 +.
Bit 3: Trigger Address Bit 13 +.
Bit 4: Trigger Address Bit 14 +.
Bit 5: Trigger Address Bit 15 +.
Bit 6: Trigger Address Bit 16 +.
Bit 7: Trigger Address Bit 17 +.

Register 3: **CS250_TRIG_ADDR_2**
Bit 0: Trigger Address Bit 18 +.
Bit 1: Trigger Address Bit 19 +.
Bit 2: Trigger Address Bit 20 +.
Bit 3: Trigger Address Bit 21 +.
Bit 4: Trigger Address Bit 22 +.
Bit 5: Trigger Address Bit 23 +.
Bit 6: Trigger Address Bit 24 +.
Bit 7: Trigger Address Bit 25 +.

Register 4:	**CS250_CHANNEL_ENABLE**
	Bit 0: Channel A Ram Bank Enable: 1 = Capture, 0 = Protect Ram.
	Bit 1: Channel B Ram Bank Enable: 1 = Capture, 0 = Protect Ram.
	Bit 2: Enable Start: 1 = Enables the clock to be started, 0 = Allows the clock to run.
	Bit 3: Fake Trigger: 0 = Set Trigger Bit, 1 = Normal Operation (CS250 Version 1.6 +).
	Bit 4: No Connection.
	Bit 5: ETS Control 0: 00 = User ETS, 01 = 1 GHZ ETS.
	Bit 6: ETS Control 1: 10 = 2 GHZ ETS, 11 = 4 GHZ ETS.
	Bit 7: ETS Enable: 0 = Normal Operation, 1 = Equivalent Time Sampling mode.
Register 5:	**CS250_ETS_DELAY**
	Bit 0: Intersample Delay Bit 0+.
	Bit 1: Intersample Delay Bit 1+.
	Bit 2: Intersample Delay Bit 2+.
	Bit 3: Intersample Delay Bit 3+.
	Bit 4: Intersample Delay Bit 4+..
	Bit 5: Intersample Delay Bit 5+.
	Bit 6: Intersample Delay Bit 6+.
	Bit 7: Intersample Delay Bit 7+.
Register 6:	**CS250_SPARE_REG_2**
	Bit 0: No Connection.
	Bit 1: No Connection.
	Bit 2: No Connection.
	Bit 3: No Connection.
	Bit 4: No Connection.
	Bit 5: No Connection.
	Bit 6: No Connection.
	Bit 7: No Connection.
Register 7:	**CS250_SPARE_REG_3**
	Bit 0: No Connection.
	Bit 1: No Connection.
	Bit 2: No Connection.
	Bit 3: No Connection.
	Bit 4: No Connection.
	Bit 5: No Connection.
	Bit 6: No Connection.
	Bit 7: No Connection.
Register 8:	**CS250_TRIGGER_CTRL**
	Bit 0: Negative Slope: 1 = Negative Slope, 0 = Positive Slope.
	Bit 1: Get Data: (-), 1-0-1 Bit Sequence To Capture.
	Bit 2: Start: (-), 1-0-1 Bit Sequence To Start Board.
	Bit 3: Keyboard Trigger: (+), 0-1-0 Bit Sequence To Trigger.
	Bit 4: Multiple Record: 0 = Normal Operation, 1 = Multiple Record.
	Bit 5: Trigger Control 0: 00 = Ch A Trig, 01 = Ch B Trig,
	Bit 6: Trigger Control 1: 10 = Ext Trig, 11 = Software Trig.
	Bit 7: No Connection.

Register 9: **CS250_TRIGGER_LEVEL**

Bit 0: Trigger Level Bit 0 +.
Bit 1: Trigger Level Bit 1 +.
Bit 2: Trigger Level Bit 2 +.
Bit 3: Trigger Level Bit 3 +.
Bit 4: Trigger Level Bit 4 +.
Bit 5: Trigger Level Bit 5 +.
Bit 6: Trigger Level Bit 6 +.
Bit 7: Trigger Level Bit 7 +.

Register A: **CS250_SEGMENT_REG**

Bit 0: Segment Bit 0 +.
Bit 1: Segment Bit 1 +.
Bit 2: Segment Bit 2 +.
Bit 3: Segment Bit 3 +.
Bit 4: Segment Bit 4 +.
Bit 5: Segment Bit 5 +.
Bit 6: Segment Bit 6 +.
Bit 7: Segment Bit 7 +.

Register B: **CS250_CLOCK_CTRL**

Bit 0: External Trigger /10 Gain: 1 = /10, 0 = x1. (R)
Bit 1: Clock Prescaler: 0 = Dual Channel and certain Single Channel, divide by 2 prescaler, 1 = Single Channel except 100 MHz, no divide by 2 prescaler. ®
Bit 2:
Bit 3:
Bit 4:

	CS250	CS225
000	100 MHz, 50 MHz	N/A
001	25 MHz	50 MHz, 25 MHz (Dual)
010	10 MHz	12.5 MHz (Dual)
011	5 MHz	10 MHz (Single), 5 MHz
100	2 MHz	N/A
101	1 MHz - 100 Hz	2 MHz (Single) - 100 Hz
110	50 Hz - 1 Hz	50 Hz - 1 Hz
111	Ext / SW	Ext / SW

Bit 5: Single Channel: 0 = Only Channel A, 1 = Both A + B channels. (R)
Bit 6: RESDAT0: Activates Memory Access, 01 and a (+) on start,
Bit 7: RESDAT1: followed by 10 and a (+) on start.

Register C: **CS250_CHANNEL_CTRL**

Bit 0: Pre trigger Enable: 1 = Pre Trig, 0 = Post Trig Mode.
Bit 1: Zero Trigger: 1 = Clear Trigger, 0 = Normal Operation (CS250 Version 1.6 +).
Bit 2: Channel A Divide 10 Gain: 1 = /10, 0 = x1 (R).
Bit 3: Abort: (+), Toggle Bit With 0-1-0 Sequence To Abort.
Bit 4: Channel B Divide 10 Gain: 1 = /10, 0 = x1 (R).
Bit 5: DC Coupling A : 1 = DC Coupling, 0 = AC coupling (R).
Bit 6: DC Coupling B : 1 = DC Coupling, 0 = AC coupling (R).
Bit 7: DC Coupling EXT: 1 = DC Coupling, 0 = AC coupling (R).

Register D: **CS250_GAIN_CTRL**

Bit 0: Channel A Times 1 Gain: 1 = x1, 0 = Not Active (R).
Bit 1: Channel A Times 2 Gain: 1 = x2, 0 = Not Active (R).
Bit 2: Channel A Times 5 Gain: 1 = x5, 0 = Not Active (R).
Bit 3: Channel A Times 10 Gain: 1 = x10, 0 = Not Active (R).
Bit 4: Channel B Times 1 Gain: 1 = x1, 0 = Not Active (R).
Bit 5: Channel B Times 2 Gain: 1 = x2, 0 = Not Active (R).
Bit 6: Channel B Times 5 Gain: 1 = x5, 0 = Not Active (R).
Bit 7: Channel B Times 10 Gain: 1 = x10, 0 = Not Active (R).

Register E: **CS250_BLOCK_0**

Bit 0: Block Bit 0 +.
Bit 1: Block Bit 1 +.
Bit 2: Block Bit 2 +.
Bit 3: Block Bit 3 +.
Bit 4: Block Bit 4 +.
Bit 5: Block Bit 5 +.
Bit 6: Block Bit 6 +.
Bit 7: Block Bit 7 +.

Register F: **CS250_BLOCK_1**

Bit 0: Block Bit 8 +.
Bit 1: Block Bit 9 +.
Bit 2: Block Bit 10 +.
Bit 3: Block Bit 11 +.
Bit 4: Block Bit 12 +.
Bit 5: Block Bit 13 +.
Bit 6: Block Bit 14 +.
Bit 7: Block Bit 15 +.

Register 10: **CS250_TIMER_0**

Bit 0: D0 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 1: D1 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 2: D2 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 3: D3 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 4: D4 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 5: D5 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 6: D6 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 7: D7 of 8254 Timer 0: Sample Rate Selection Divider +.

Register 11: **CS250_TIMER_1**

Bit 0: D0 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 1: D1 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 2: D2 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 3: D3 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 4: D4 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 5: D5 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 6: D6 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 7: D7 of 8254 Timer 1: Sample Rate Selection Divider +.

Register 12:	**CS250_TIMER_2**
	Bit 0: D0 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 1: D1 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 2: D2 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 3: D3 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 4: D4 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 5: D5 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 6: D6 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 7: D7 of 8254 Timer 2: Post Trigger Depth Selection +.
Register 13:	**CS250_TIMER_CTRL**
	Bit 0: BCD: 0 = Binary Counter (BCD Mode Not Used).
	Bit 1: M0: Mode Selection: Only Modes 3 + 5 Required.
	Bit 2: M1: Mode 3: Square Wave (Timers 0 + 1).
	Bit 3: M2: Mode 5: Hardware Trigger Strobe (Timer 2).
	Bit 4: RW0: Read/Write Control: 11 = Write LSB First and MSB
	Bit 5: RW1: Second, Only Read/Write mode required.
	Bit 6: SC0: Select Counter: 00 = Timer 0, 01 = Timer 1,
	Bit 7: SC1: 10 = Timer 2.

Appendix H: CS220 register functionality.

LIST OF I/O PORTS USED BY THE COMPUSCOPE 220

	SYMBOL KEY
+	Logic high level active.
-	Logic low level active.
(+)	Active on rising edge pulse.
(-)	Active on falling edge pulse.
*	These bits are valid for 1M, 2M, 4M and 8M memory boards only, and are not decoded on the 32K and 256K memory boards.

Register 0: **CS220_TRIGGER_CTRL**

Bit 0:	Positve Slope: 1 = Positive Slope, 0 = Negative Slope.
Bit 1:	MULREC: 0 = Normal operation, 1 = Multiple recording (sample stacking).
Bit 2:	Full memory: 1 = 256K, 0 = All Other Memory Configurations.
Bit 3:	Keyboard Trigger (+): 0-1-0 Sequence to Issue a Trigger Event.
Bit 4:	Need Ram: 1 = CS220 Access, 0 = PC Access.
Bit 5:	Trigger Enable A: 1 = Trigger Source is Channel A.
Bit 6:	Trigger Enable B: 1 = Trigger Source is Channel B.
Bit 7:	Trigger Enable EXT: 1 = Trigger Source is External Input.

Register 1: **CS220_CHANNEL_CTRL**

Bit 0:	Channel A Ram Bank Enable: 1= Capture, 0 = Protect Ram.
Bit 1:	Channel B Ram Bank Enable: 1= Capture, 0 = Protect Ram. (Both High For Single Channel Operation.)
Bit 2:	Abort (+): 0-1-0 Sequence to Stop Capture.
Bit 3:	Start (+): 0-1-0 Sequence to Start Clock.
Bit 4:	Get Data (-): 1-0-1 Sequence to Start Capture.
Bit 5:	DC Coupling A: 1 = DC Coupling, 0 = AC Coupling.
Bit 6:	DC Coupling B: 1 = DC Coupling, 0 = AC Coupling.
Bit 7:	DC Coupling EXT: 1 = DC Coupling, 0 = AC Coupling.

Register 2: **CS220_SEGMENT_REG**

Bit 0:	Segment Bit 0 +.
Bit 1:	Segment Bit 1 +.
Bit 2:	Segment Bit 2 +.
Bit 3:	Segment Bit 3 +.
Bit 4:	Segment Bit 4 +.
Bit 5:	Segment Bit 5 +.
Bit 6:	Segment Bit 6 +.
Bit 7:	Segment Bit 7 +.

Register 3:	**CS220_BLOCK_0**
Bit 0:	Block Bit 0 +.
Bit 1:	Block Bit 1 +.
Bit 2:	Block Bit 2 +.
Bit 3:	Block Bit 3 +.
Bit 4:	Block Bit 4 +.
Bit 5:	Block Bit 5 +.
Bit 6:*	Block Bit 6 +.
Bit 7:*	Block Bit 7 +.

Register 4:	**CS220_CLOCK_0**
Bit 0:	24 bit clock register bit 0 +.
Bit 1:	24 bit clock register bit 1 +.
Bit 2:	24 bit clock register bit 2 +.
Bit 3:	24 bit clock register bit 3 +.
Bit 4:	24 bit clock register bit 4 +.
Bit 5:	24 bit clock register bit 5 +.
Bit 6:	24 bit clock register bit 6 +.
Bit 7:	24 bit clock register bit 7 +.

Register 5:	**CS220_CLOCK_1**
Bit 0:	24 bit clock register bit 8 +.
Bit 1:	24 bit clock register bit 9 +.
Bit 2:	24 bit clock register bit 10 +.
Bit 3:	24 bit clock register bit 11 +.
Bit 4:	24 bit clock register bit 12 +.
Bit 5:	24 bit clock register bit 13 +.
Bit 6:	24 bit clock register bit 14 +.
Bit 7:	24 bit clock register bit 15 +.

Register 6:	**CS220_CLOCK_2**
Bit 0:	24 bit clock register bit 16 +.
Bit 1:	24 bit clock register bit 17 +.
Bit 2:	24 bit clock register bit 18 +.
Bit 3:	24 bit clock register bit 19 +.
Bit 4:	24 bit clock register bit 20 +.
Bit 5:	24 bit clock register bit 21 +.
Bit 6:	24 bit clock register bit 22 +.
Bit 7:	24 bit clock register bit 23 +.

Register 7:	**CS220_SAMPLE_DEPTH**
Bit 0:	Pre trigger Enable +. 0 = Other, 1 = Pre trig only.
Bit 1:	Sample Depth Bit 0 +. Values shown are single channel/dual channel depths:
Bit 2:	Sample Depth Bit 1 +. 256/128=01000,512/256=01001,1K/512=01010,2K/1K=01011,
Bit 3:	Sample Depth Bit 2 +. 4K/2K=01100,8K/4K=01101,16K/8K=01110,32K/16K=01111,
Bit 4:	Sample Depth Bit 3 +. 64K/32K=01100,128K/64K=01101,256K/128K=01110, 512K/256K=01111,
Bit 5:	Sample Depth Bit 4 +. 1M/512K=10100,2M/1M=10101,4M/2M=10110,8M/4M=10111.
Bit 6:	Single Channel +. 0 = Dual Mode, 1 = Single Mode.
Bit 7:	Double Rate -. 1 = Dual Mode, 0 = Single Mode.

Register 8:		**CS220_STATUS_REG**
	Bit 0:	Busy: 1 = Converting Data, 0 = Memory can be accessed.
	Bit 1:	Ram Full: 1 = Ram Is Full, 0 = Ram Is Not Full.
	Bit 2:	Trigger Received: 1 = Trigger Has Occurred, 0 = No Trigger Received.
	Bit 3:	Auxilary Trigger Received: 1 = Trigger Has Occurred, 0 = No Trigger Received.
	Bit 4:	No Connection.
	Bit 5:	No Connection.
	Bit 6:	No Connection.
	Bit 7:	No Connection.

Register 9:		**CS220_TRIGGER_ADDR_0**
	Bit 0:	Trigger Address Bit 0 +.
	Bit 1:	Trigger Address Bit 1 +.
	Bit 2:	Trigger Address Bit 2 +.
	Bit 3:	Trigger Address Bit 3 +.
	Bit 4:	Trigger Address Bit 4 +.
	Bit 5:	Trigger Address Bit 5 +.
	Bit 6:	Trigger Address Bit 6 +.
	Bit 7:	Trigger Address Bit 7 +.

Register A:		**CS220_TRIGGER_ADDR_1**
	Bit 0:	Trigger Address Bit 8 +.
	Bit 1:	Trigger Address Bit 9 +.
	Bit 2:	Trigger Address Bit 10 +.
	Bit 3:	Trigger Address Bit 11 +.
	Bit 4:	Trigger Address Bit 12 +.
	Bit 5:	Trigger Address Bit 13 +.
	Bit 6:	Trigger Address Bit 14 +.
	Bit 7:	Trigger Address Bit 15 +.

Register B:		**CS220_GAIN_CTRL**
	Bit 0:	Channel A Gain Bit 0 +. 000 = /10, 001 = x1, 010 = /5,
	Bit 1:	Channel A Gain Bit 1 +. 011 = x2, 100 = /2, 101 = x5,
	Bit 2:	Channel A Gain Bit 2 +. 110 = NA, 111 = x10.
	Bit 3:	External Gain Bit +. 0 = x10, 1 = x1.
	Bit 4:	Channel B Gain Bit 0 +. 000 = /10, 001 = x1. 010 = /5,
	Bit 5:	Channel B Gain Bit 1 +. 011 = x2, 100 = /2, 101 = x5,
	Bit 6:	Channel B Gain Bit 2 +. 110 = NA, 111 = x10.
	Bit 7:	No Connection.

Register C:		**CS220_TRIGGER_LEVEL**
	Bit 0:	Trigger Level Bit 7 +.
	Bit 1:	Trigger Level Bit 6 +.
	Bit 2:	Trigger Level Bit 5 +.
	Bit 3:	Trigger Level Bit 4 +.
	Bit 4:	Trigger Level Bit 3 +.
	Bit 5:	Trigger Level Bit 2 +.
	Bit 6:	Trigger Level Bit 1 +.
	Bit 7:	Trigger Level Bit 0 +.

Register D:	**CS220_NOT_USED_0D**
Bit 0:*	No Connection.
Bit 1:*	No Connection.
Bit 2:*	No Connection.
Bit 3:*	No Connection.
Bit 4:*	No Connection.
Bit 5:*	No Connection.
Bit 6:*	No Connection.
Bit 7:*	No Connection.

Register E:	**CS220_BLOCK_1**	
Bit 0:*	Block Bit 8 +.	
Bit 1:*	Block Bit 9 +.	
Bit 2:*	Block Bit 10 +.	
Bit 3:*	Block Bit 11 +.	
Bit 4:*	Block Bit 12 +.	
Bit 5:*	Reset clock (+).	Set Reset Data 1 and Reset Data 0 to 10 and then issue Reset Clock.
Bit 6:*	Reset Data 0 +.	Set Reset Data 1 and Reset Data 0 to 01 and then issue Reset Clock.
Bit 7:*	Reset Data 1 +.	Set Reset Data 1 and Reset Data 0 to 11 and then issue Reset Clock.

Register F:	**CS220_TRIGGER_ADDR_2**
Bit 0:*	Trigger Address Bit 16 +.
Bit 1:*	Trigger Address Bit 17 +.
Bit 2:*	Trigger Address Bit 18 +.
Bit 3:*	Trigger Address Bit 19 +.
Bit 4:*	Trigger Address Bit 20 +.
Bit 5:*	Trigger Address Bit 21 +.
Bit 6:*	Trigger Address Bit 22 +.
Bit 7:*	Trigger Address Bit 23 +.

Appendix I: CSLITE register functionality.

LIST OF I/O PORTS USED BY THE COMPUSCOPE LITE

	SYMBOL KEY
+	Logic high level active.
-	Logic low level active.
(+)	Active on rising edge pulse.
(-)	Active on falling edge pulse.
(R)	Controls at least one relay with this signal Relays used take approximately 1 ms to activate.

Register 0: **CSLITE_TRIGGER_CTRL**

Bit 0: Negative Slope: 1 = Negative Slope, 0 = Positive Slope.
Bit 1: Get Data: (-), 1-0-1 Bit Sequence To Capture.
Bit 2: Reset Clock: (-), 1-0-1 Bit Sequence To Allow Memory Access.
Bit 3: Keyboard Trigger: (+), 0-1-0 Bit Sequence To Trigger.
Bit 4: Need ram: 1 = CSLITE Access, 0 = PC Access.
Bit 5: Trigger Control 0: 100 = Ch A Trigger, 101 = Ch B Trigger,
Bit 6: Trigger Control 1: 110 = External Trigger.
Bit 7: Trigger Control 2: 0XX = Software Trigger.

Register 1: **CSLITE_TRIGGER_LEVEL**

Bit 0: Trigger Level Bit 0 +.
Bit 1: Trigger Level Bit 1 +.
Bit 2: Trigger Level Bit 2 +.
Bit 3: Trigger Level Bit 3 +. Note: All bits in this register
Bit 4: Trigger Level Bit 4 +. must be inverted and reversed.
Bit 5: Trigger Level Bit 5 +.
Bit 6: Trigger Level Bit 6 +.
Bit 7: Trigger Level Bit 7 +.

Register 2: **CSLITE_SEGMENT_REG**

Bit 0: Segment Bit 0 +.
Bit 1: Segment Bit 1 +.
Bit 2: Segment Bit 2 +.
Bit 3: Segment Bit 3 +. Note: Most significant byte only,
Bit 4: Segment Bit 4 +. least significant byte is zero.
Bit 5: Segment Bit 5 +.
Bit 6: Segment Bit 6 +.
Bit 7: Segment Bit 7 +.

Register 3: **CSLITE_CLOCK_CTRL**

Bit 0: No Connection.
Bit 1: No Connection.
Bit 2: Clock Ctrl 1: 000 = 40 MHz, 001 = 20 MHz, 010 = 10 MHz,
Bit 3: Clock Ctrl 2: 011 = 5 MHz, 100 = 2 MHz, 101 = 1 MHz - 400 Hz,
Bit 4: Clock Ctrl 3: 110 = 400 Hz - 1 Hz, 111 = External clock.
Bit 5: Single Channel: 0 = Only Channel A, 1 = Both A + B channels. (R)
Bit 6: RESDAT0: 00 and (+) on Reset Clock, 01 and (+) on Reset Clock,
Bit 7: RESDAT1: 10 and (+) on Reset Clock, 11 and (+) on Reset Clock,

Register 4:	**CSLITE_SAMPLE_CTRL**
	Bit 0: Pre trigger Enable: 1 = Pre Trig, 0 = Post Trig Mode.
	Bit 1: Start: (+), 0-1-0 Bit Sequence To Start Board.
	Bit 2: No Connection
	Bit 3: Abort: (+), Toggle Bit With 0-1-0 Sequence To Abort.
	Bit 4: Block Bit 0 +.
	Bit 5: Block Bit 1 +.
	Bit 6: Block Bit 2 +.
	Bit 7: Channel: 0 = Channel A, 1 = Channel B.
Register 5:	**CSLITE_GAIN_CTRL**
	Bit 0: Test input: 0 = Channel Data, 1 = Test Signal (R).
	Bit 1: AC Coupling A : 1 = AC Coupling, 0 = DC coupling (R).
	Bit 2: Channel A Divide 5 Gain: 1 = /5, 0 = x1 (R).
	Bit 3: Channel A Times 5 Gain: 1 = x5, 0 = x1 (R).
	Bit 4: No Connection.
	Bit 5: AC Coupling B : 1 = AC Coupling, 0 = DC coupling (R).
	Bit 6: Channel B Divide 5 Gain: 1 = /5, 0 = x1 (R).
	Bit 7: Channel B Times 5 Gain: 1 = x5, 0 = x1 (R).
Register 6:	**CSLITE_SPARE_REG_1**
	Not Used.
Register 7:	**CSLITE_SPARE_REG_2**
	Not Used.
Register 8:	**CSLITE_STATUS_REG**
	Bit 0: Busy: 1 = Converting Data, 0 = Memory can be accessed.
	Bit 1: Ram Full: 1 = Ram Is Full, 0 = Ram Is Not Full.
	Bit 2: Trigger Received: 1 = Trigger Occurred, 0 = No Trigger.
	Bit 3: Local Board Trigger: 1 = Responded to Trigger Source, 0 = Did Not Cause Trigger.
	Bit 4: Never Available.
	Bit 5: Never Available.
	Bit 6: Never Available.
	Bit 7: Never Available.
Register 9:	**CSLITE_TRIG_ADDR_0**
	Bit 0: Trigger Address Bit 0 +.
	Bit 1: Trigger Address Bit 1 +.
	Bit 2: Trigger Address Bit 2 +.
	Bit 3: Trigger Address Bit 3 +.
	Bit 4: Trigger Address Bit 4 +.
	Bit 5: Trigger Address Bit 5 +.
	Bit 6: Trigger Address Bit 6 +.
	Bit 7: Trigger Address Bit 7 +.

Register A: **CSLITE_TRIG_ADDR_1**
Bit 0: Trigger Address Bit 8 +.
Bit 1: Trigger Address Bit 9 +.
Bit 2: Trigger Address Bit 10 +.
Bit 3: Trigger Address Bit 11 +.
Bit 4: Trigger Address Bit 12 +.
Bit 5: Trigger Address Bit 13 +.
Bit 6: Trigger Address Bit 14 +.
Bit 7: Trigger Address Bit 15 +.

Register B: **CSLITE_SPARE_REG_3**
Not Used.

Register C: **CSLITE_SPARE_REG_4**
Not Used.

Register D: **CSLITE_SPARE_REG_5**
Not Used.

Register E: **CSLITE_SPARE_REG_6**
Not Used.

Register F: **CSLITE_SPARE_REG_7**
Not Used.

Register 10: **CSLITE_TIMER_0**
Bit 0: D0 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 1: D1 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 2: D2 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 3: D3 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 4: D4 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 5: D5 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 6: D6 of 8254 Timer 0: Sample Rate Selection Divider +.
Bit 7: D7 of 8254 Timer 0: Sample Rate Selection Divider +.

Register 11: **CSLITE_TIMER_1**
Bit 0: D0 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 1: D1 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 2: D2 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 3: D3 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 4: D4 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 5: D5 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 6: D6 of 8254 Timer 1: Sample Rate Selection Divider +.
Bit 7: D7 of 8254 Timer 1: Sample Rate Selection Divider +.

Register 12:	**CSLITE_TIMER_2**
	Bit 0: D0 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 1: D1 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 2: D2 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 3: D3 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 4: D4 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 5: D5 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 6: D6 of 8254 Timer 2: Post Trigger Depth Selection +.
	Bit 7: D7 of 8254 Timer 2: Post Trigger Depth Selection +.

Register 13:	**CSLITE_TIMER_CTRL**
	Bit 0: BCD: 0 = Binary Counter (BCD Mode Not Used).
	Bit 1: M0: Mode Selection: Only Modes 3 + 5 Required.
	Bit 2: M1: Mode 3: Square Wave (Timers 0 + 1).
	Bit 3: M2: Mode 5: Hardware Trigger Strobe (Timer 2).
	Bit 4: RW0: Read/Write Control: 11 = Write LSB First and MSB
	Bit 5: RW1: Second, Only Read/Write mode required.
	Bit 6: SC0: Select Counter: 00 = Timer 0, 01 = Timer 1,
	Bit 7: SC1: 10 = Timer 2.

Appendix J: Hardware/Software Data Translation.

The following tables illustrate the methods required to convert the data from CompuScope data acquisition cards.

The data value tables represent the data read from the card after a successful acquisition. The “Logical” column represents data acquired by calling the **gage_mem_read_XXXXXX** routines from the CompuScope driver. The “hardware” column represents data as stored in the memory of the CompuScope card. The driver calls to **gage_32k_to_buffer** and **gage_trigger_view_transfer** return the data as it is stored on the card.

The trigger level value tables show how the trigger levels assigned convert into voltages that will match the data captured from the data acquisition card.

CompuScope 250 / 225 Data Values

Voltage	Logical	Hardware
+1 Volt	0	0
+0.5 Volt	64	64
0 Volt	128	128
-0.5 Volt	192	192
-1 Volt	255	255

CompuScope 220 Data Values

Voltage	Logical	Hardware
+1 Volt	0	255
+0.5 Volt	64	192
0 Volt	128	127
-0.5 Volt	192	64
-1 Volt	255	0

CompuScope LITE Data Values

Voltage	Logical	Hardware
+1 Volt	0	0
+0.5 Volt	64	64
0 Volt	128	128
-0.5 Volt	192	192
-1 Volt	255	255

CompuScope 6012 / 1012 Data Values

Voltage	Logical	Hardware
+1 Volt	-2048	-2048
+0.5 Volt	-1024	-1024
0 Volt	0	0
-0.5 Volt	1024	1024
-1 Volt	2047	2047

CompuScope 250 / 225 Trigger Level Values

Voltage	Logical	Sent toDAC
+1 Volt	255	0
+0.5 Volt	192	63
0 Volt	128	127
-0.5 Volt	64	191
-1 Volt	0	255

CompuScope 220 Trigger Level Values

Voltage	Logical	Sent to DAC
+1 Volt	255	255
+0.5 Volt	192	3
0 Volt	128	1
-0.5 Volt	64	2
-1 Volt	0	0

CompuScope Lite Trigger Level Values

Voltage	Logical	Sent to DAC
+1 Volt	255	0
+0.5 Volt	192	252
0 Volt	128	254
-0.5 Volt	64	253
-1 Volt	0	255

CompuScope 6012 / 1012 Trigger Level Values

Voltage	Logical	Sent to DAC
+1 Volt	255	16
+0.5 Volt	192	1024
0 Volt	128	2048
-0.5 Volt	64	3072
-1 Volt	0	4096

To convert from the logical data value to a voltage, the formula is:

((128 - level) / 128) * input gain * probe factor (for 8-bit cards.)
(level / (0 - 2048)) * input gain * probe factor (for 12-bit cards.)

To convert from the logical trigger value to a voltage, the formula is:

((level - 128) / 128) * trigger gain * probe factor

Appendix K: GageScope Binary file Format (.SIG).

File Index	Field Type	Field Size	Field Variable	Field Description
0	char	14	file_version	Either GS V.1.20, GS V.2.00, GS V.2.05, GS V.2.10, GS V.2.15, GS V.2.20, GS V.2.25, GS V.2.50, GS V.2.60, GS V.2.65, GS V.2.70, GS V.2.75 or GS V.2.80.
14	int	2	crlf1	A carriage return line feed pair.
16	char	9	name	The channel name when stored.
25	int	2	crlf2	A carriage return line feed pair.
27	char	256	comment	The channel comment when stored.
283	int	2	crlf3	A carriage return line feed pair.
285	int	2	control_z	A control Z, artificial end of file.
287	int	2	sample_rate_index	Index to the sample rate table. Note 1.
289	int	2	operation_mode	1 = single channel, 2 = dual channel.
291	long	4	trigger_depth	Number of samples after the trigger point.
295	int	2	trigger_slope	1 = positive slope, 2 = negative slope.
297	int	2	trigger_source	1 = chan A, 2 = chan B, 3 = external, 4 = automatic, 5 = keyboard.
299	int	2	trigger_level	Stored as an int, actually a byte with the same format as the data. Note 4.
301	long	4	sample_depth	Number of bytes stored in the signal section of the file.

305	int	2	captured_gain	Index to the input range table. Note 2.
307	int	2	captured_coupling	1 = DC, 2 = AC.
309	long	4	current_mem_ptr	Where display started when signal was stored.
313	long	4	starting_address	The first point in the data.
317	long	4	trigger_address	The point in the data where trigger occurred.
321	long	4	ending_address	The last point of the captured data.
325	word	2	trigger_time	The time when the trigger event occurred. Note 6.
327	word	2	trigger_date	The date on which the trigger event occurred.
329	int	2	trigger_coupling	1 = DC, 2 = AC. For the external trigger input.
331	int	2	trigger_gain	Index to the input range table. Note 2.
333	int	2	probe	Index to the probe table. Note 3.
335	int	2	inverted_data	0 = normal data, 1 = inverted data (CS220), 2 = inverted and flipped data (CS220).
337	word	2	board_type	The CompuScope board type on which the saved data was captured. Note 5.
339	int	2	resolution_12_bits	0 = 8 bit file format, 1 = 12/16 bit file format.
341	int	2	multiple_record	The mode that the saved data was captured in: 0 = normal mode, 1 = Hardware multiple record, 2 = Software multiple record.
343	int	2	trigger_probe	Index to the probe table. Note 3.
345	int	2	sample_offset	Used to offset the data for display and conversion to real voltages. Normally 128 for 8-bit CompuScopes and zero for 12-bit CompuScopes.

347	int	2	sample_resolution	Used to scale the data for display and conversion to real voltages. Normally 128 for 8-bit CompuScopes and 2048 for 12-bit CompuScopes.
349	int	2	sample_bits	Number of bits in the sampled data. Normally 8 for 8-bit CompuScopes and 12 for 12-bit CompuScopes.
351	lword	4	extended trigger time	The time when trigger event occurred. Note 6.
355	byte	157	padding	0 filled section to complete the 512 byte header.
512	byte / int see note	var	signal	RAM image of the CompuScope memory at the time the signal file was stored. This data is in two different formats. The first format is when "operation_mode" is equal to two. The data is stored contiguously as a binary image of the saved channel's signal storage space (one-half the memory depth). The second format is when "operation_mode" is equal to one. The data is interleaved as a binary image of the complete signal storage space for the single channel mode (full memory depth). Interleaved in this example means the data is in two sections after the header. The first section is the data that was stored in the CompuScope memory for channel A and the second section of data is the memory assigned to channel B. All even addresses are in the area for channel A while the odd addresses are in the same place as the next smaller address, but offset into the channel B data area. In both cases, the stored addresses for the signals are indexes into the data. When extracting data, care must be taken to "wrap the pointers around" at the end of the file. **NOTE:** if the "resolution_12_bits" flag equals zero then the data is stored as unsigned 8 bit bytes. Note 4. Otherwise, if the "resolution_12_bits" flag equals one then the data is in the 12/16 bit format which is stored as 16 bit signed integers (in the 12 bit mode the sampled data is sign extended to 16 bits).

Note 1: The "sample_rate_index" to the sample rate table.

For all file versions		Version 2.65 and below		Version 2.70 and above		Version 2.85 and above	
Index	**Sample Rate**	**Index**	**Sample Rate**	**Index**	**Sample Rate**	**Index**	**Sample Rate**
0	1 Hz	18	1 MHz	18	1 MHz	18	1 MHz
1	2 Hz	19	2 MHz	19	2 MHz	19	2 MHz
2	5 Hz	20	5 MHz	20	4 MHz	20	4 MHz
3	10 Hz	21	10 MHz	21	5 MHz	21	5 MHz
4	20 Hz	22	20 MHz	22	10 MHz	22	10 MHz
5	50 Hz	23	25 MHz	23	20 MHz	23	12.5 MHz
6	100 Hz	24	40 MHz	24	25 MHz	24	20 MHz
7	200 Hz	25	50 MHz	25	30 MHz	25	25 MHz
8	500 Hz	26	100 MHz	26	40 MHz	26	30 MHz
9	1 KHz			27	50 MHz	27	40 MHz
10	2 KHz			28	60 MHz	28	50 MHz
11	5 KHz			29	100 MHz	29	60 MHz
12	10 KHz			30	120 MHz	30	100 MHz
13	20 KHz			31	125 MHz	31	120 MHz
14	50 KHz			32	150 MHz	32	125 MHz
15	100 KHz			33	200 MHz	33	150 MHz
16	200 KHz			34	250 MHz	34	200 MHz
17	500 KHz			35	300 MHz	35	250 MHz
				36	500 MHz	36	300 MHz
				37	1 GHz	37	500 MHz
				38	2 GHz	38	1 GHz
				39	5 GHz	39	2 GHz
				40	External Clock	40	4 GHz
				39*	4 GHz	41	5 GHz
				40*	5 GHz	42	External Clock
				41*	External Clock		

***** For File Version 2.80 and above

Note 2: The "captured_gain" and "trigger_gain" input ranges.

If "file_version" equals "GS V.1.20"	**Index**	**Input Range**
	0	+/- 1v
	1	+/- 200mv

If "file_version" is greater or equal to "GS V.2.00"	**Index**	**Input Range**
	0	+/- 10v
	1	+/- 5v
	2	+/- 2v
	3	+/- 1v
	4	+/- 500mv
	5	+/- 200mv
	6	+/- 100mv

Note 3: The "probe" connected to the input channel at time of capture.

If "file_version" is less than or equal to "GS V.2.10"	**Index**	**Probe Multiplier**
	0	x1
	1	x10
	2	x20
	3	x100
	4	x200

If "file_version" is greater than or equal to "GS V.2.15"	**Index**	**Probe Multiplier**
	0	x1
	1	x10
	2	x20
	3	x50
	4	x100
	5	x200
	6	x500
	7	x1000

Note 4: The "trigger_level" and the "captured_data" 8 bit byte format.

The byte stored as the trigger level is in the same format as the data read back from the CompuScope hardware. The data is unsigned and the largest value (255) represents -1 volt and the smallest value (0) represents + 1 volt. The data is normalized into a signed floating-point representation. The input range and probe are then multiplied by the return value to produce the actual level either sent to the hardware as the trigger level or returned from it after capture.

Note 5: The "board_type" constants.

"board_type"	**CompuScope Hardware**
0x0000	Unknown (pre "file_version" GS V.2.25).
0x0040	CompuScope 225
0x0100	CompuScope LITE (pre Hardware Version v 1.5).
0x0200	CompuScope 220.
0x0400	CompuScope 250.
0x0800	CompuScope LITE (Hardware Version v 1.5 & up).
0x1000	CompuScope 1012.
0x2000	CompuScope 6012.
0x4000	CompuScope 2125
0x8000	Reserved by Gage.

Note 6: Extended trigger time.

GageScope now supports an extended time stamp for the trigger event time of day. The resolution of this timer is in the hundredths of a second. However, on some computer systems the real-time clock does not support complete accuracy at this resolution. If the normal trigger time in the signal file is zero, then the extended trigger time stamp is being used. The encoded long integer is hhhhhmmmmmmsssssddddddd, where dddddd is the hundredth of a second. Note also that the new time also supports full seconds.

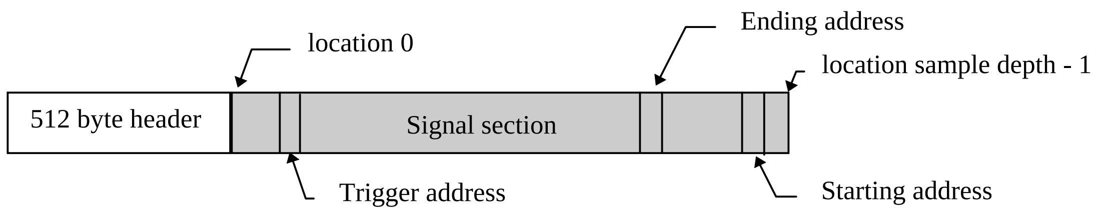
Reading GageScope Signal Files

Reading data from a GageScope signal file is similar to reading data returned from one of the block transfer routines, **gage_trigger_view_transfer** or **gage_32k_to_buffer**. For 8 bit cards, the samples are returned as a byte with a value between 0 (representing the most positive voltage) and 255 (representing the most negative voltage). For 12 bit cards, the samples are returned as a 12 bit integer, sign extended to 16 bits, between -2048 (the most positive voltage) and 2047 (the most negative voltage). The samples returned from the CS220 are always inverted (one's complement) and the least significant bit of the address is flipped under the following circumstances:

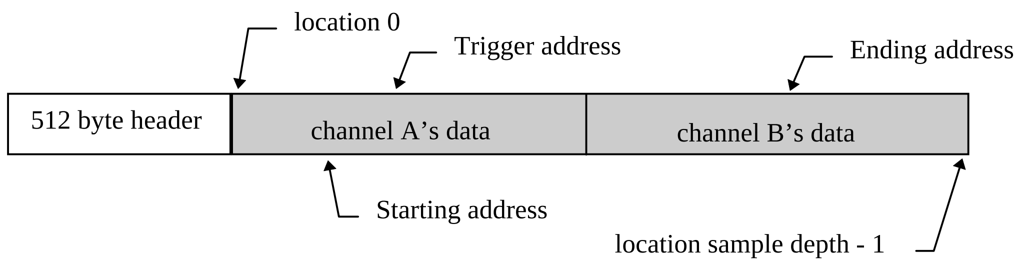
a) Single channel mode and board has 1M or greater, then the least significant bit of channel A's addresses are flipped.
b) Dual channel mode, the CompuScope 220 has less than 1M of memory and you are capturing channel B's data.

You can determine if this is the case by examining the inverted data field of the file header. If it is a one, the data is inverted. If the field has a two in it, the least significant bit of the address will be flipped. See the section titled Memory Organization of the CompuScope for more information.

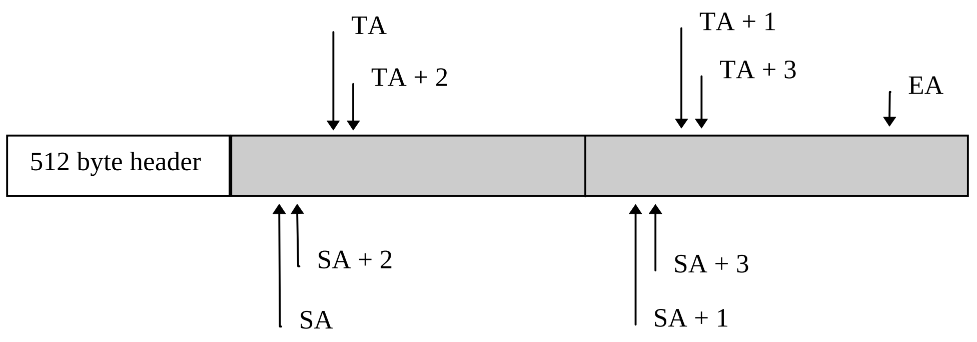
In versions 2.71 and later of GAGESCOP.EXE, files can be saved in three different ways. The Save All option saves a binary image of the complete memory depth of the card to disk. For example, with a 16K CSLITE, using the Save All option in dual channel mode to save channel A's data will result in a file of 8,704 bytes. The maximum available memory depth in dual channel mode is 8,192 bytes, with a 512 byte header. The signal section of the file starts at byte number 512. The starting, trigger and ending addresses can be obtained from the header, as well as the maximum available memory (sample depth). The usual method to read the data is to start at the trigger address and loop through the file, reading in one sample at a time. Because the file is saved as an image of memory, care must be taken to wrap around the circular buffer to avoid reading invalid data. If the file is saved as Normal or User Defined, only the actual sample points captured are saved to disk (padded to an 4K boundary). Because of the bit-flipping on the CS220, its data must start on an even address and end on an odd address, therefore its files may be slightly larger. In the case of both Normal or User Defined, the signal section of the file is normalized. If the starting address <= trigger address <= ending address, the file is normalized. In this case, you don't need to worry about wrapping around the circular buffer. The following diagram shows a signal file that was saved using the Save All option in dual channel mode. The gray area is the signal section and it is sample depth size long. The samples from the trigger address to the ending address are the valid post trigger points. The samples between the starting address and the trigger address - 1 are the valid pretrigger samples.



If the file is saved in single channel mode, the samples are interleaved between the data space of channel A and channel B. The first data point will be in channel A, the second in channel B, the third in A, etc. Therefore, after the header section, the first point will be at the trigger address, the second point at the trigger address + half the sample depth, the third point at the trigger address + 1, etc. The following diagram illustrates a signal file that was saved using the Save All option in single channel mode. If an address is even, it is in channel A's data space. If it is odd, it is in channel B's data space. For example, if the file was saved from a 16K CompuScope LITE, the sample depth in single channel mode is 16K. If the first sample read is from the trigger address, the next is in trigger address + ((sample depth / 2) - 1). The third point is in location trigger address + 1, etc.



For example, as shown below, the starting address (SA) is in channel A, the next point (SA + 1) is in channel B and so on. This is the case for single channel mode. If the file was saved with a CS220, then under certain circumstances (as described above) the least significant bit of channel A's address will be flipped.



Appendix L: GageScope Setup file Format (.SET).

GageScope stores the setup in an ASCII file with the following format:

Line	Type	Name	Description
1	char	file_version	Either GS V.1.20, GS V.2.00, GS V.2.05, GS V.2.10, GS V.2.15, GS V.2.20, GS V.2.25, GS V.2.50, GS V.2.60, GS V.2.65, GS V.2.70, GS V.2.75 or GS V.2.80.
2	char	name	The channel name.
3	char	comment	The Comment associated with the channel.
4	char	disk_filename	The file name used when enable is set to DISK.
5	int	enable	0 = off, 1 = on, 2 = test, 3 = disk, 4 = math.
6	int	amplitude	Index to the amplitude table. Note 1.
7	int	time_base	Index to the time base table. Note 2.
8	int	normal_display	0 = inverted, 1 = normal.
9	int	probe	Index to the probe table. Note 3.
10	int	gain	Index to the input range table. Note 4.
11	int	coupling	1 = DC, 2 = AC.
12	int	offset_0	8 channel display offset, internal use only.
13	int	offset_1	16 channel display offset, internal use only.
14	int	offset_2	32 channel display offset, internal use only.

2 - 12	are the entries for channel 1.
15 - 27	are the entries for channel 2.
28 - 40	are the entries for channel 3.
41 - 53	are the entries for channel 4.
54 - 66	are the entries for channel 5.
67 - 79	are the entries for channel 6.
80 - 92	are the entries for channel 7.
93 - 105	are the entries for channel 8.
106 - 118	are the entries for channel 9.
119 - 131	are the entries for channel 10.
132 - 144	are the entries for channel 11.
145 - 157	are the entries for channel 12.
158 - 170	are the entries for channel 13.
171 - 183	are the entries for channel 14.
184 - 196	are the entries for channel 15.
197 - 209	are the entries for channel 16.
210 - 222	are the entries for channel 17.
223 - 235	are the entries for channel 18.
236 - 248	are the entries for channel 19.
249 - 261	are the entries for channel 20.
262 - 274	are the entries for channel 21.
275 - 287	are the entries for channel 22.
288 - 300	are the entries for channel 23.
301 - 313	are the entries for channel 24.
314 - 326	are the entries for channel 25.
327 - 339	are the entries for channel 26.
340 - 352	are the entries for channel 27.

353 - 365		are the entries for channel 28.	
366 - 378		are the entries for channel 29.	
379 - 391		are the entries for channel 30.	
392 - 404		are the entries for channel 31.	
405 - 417		are the entries for channel 32.	
418 - 430		are the entries for master channel.	
431	int	sample_rate_index	Index to the sample rate table. Note 5.
432	int	operation_mode	1 = single channel, 2 = dual channel.
433	long	trigger_depth	Number of samples after the trigger point.
434	int	trigger_level	Stored as an int, actually a byte with the same format as the data. Note 6.
435	int	trigger_slope	1 = positive, 2 = negative.
436	int	trigger_source	1 = chan 1, 2 = chan 2, 3 = external, 4 = automatic or 5 = keyboard.
437	int	trigger_level_2	Stored as an int, actually a byte with the same format as the data (CS6012 / 1012 only). Note 6.
438	int	trigger_slope_2	1 = positive, 2 = negative (CS6012 / 1012 only).
439	int	trigger_source_2	1 = chan 1, 2 = chan 2, 3 = external or 4 = disabled (CS6012 / 1012 only).
440	int	trigger_coupling	1 = DC, 2 = AC.
441	int	trigger_gain	Index to the input range table. Note 4.
442	int	trigger_probe	Index to the probe table. Note 3.
434 - 442		are the entries for board 1.	
443 - 451		are the entries for board 2.	
452 - 460		are the entries for board 3.	
461 - 469		are the entries for board 4.	
470 - 478		are the entries for board 5.	
479 - 487		are the entries for board 6.	
488 - 496		are the entries for board 7.	
497 - 505		are the entries for board 8.	
506 - 514		are the entries for board 9.	
515 - 523		are the entries for board 10.	
524 - 532		are the entries for board 11.	
533 - 541		are the entries for board 12.	
542 - 550		are the entries for board 13.	
551 - 559		are the entries for board 14.	
560 - 568		are the entries for board 15.	
569 - 577		are the entries for board 16.	
578	long	continuous_trigger_timeout	Number of microseconds that GageScope waits before triggering automatically.
579	int	draw_signals_as_lines	0 = draw only the data points, 1 = connect the data points with lines, 2 = perform MEAN not MIN/MAX on densely displayed data.
580	int	display_grid	0 = Do not show the display grid, 1 = display the grid.
581	int	display_zero_ref	0 = no zero reference, 1 = zero reference drawn with signal.
582	int	display_data_point	0 = post-trigger data display, -1 = mid-trigger data display and -2 = pre-trigger data display.

583	int	channel_display_format	0 = display channels 1 - 8, 1 = display channels 9 - 16, 2 = display channels 17 - 24, 3 = display channels 25 - 32, 4 = display channels 1 - 16, 5 = display channels 17 - 32, 6 = display channels 1 - 32.
584	int	automatic_save_feature	0 = auto save is off, 1 = auto save is on.
585	int	automatic_save_max	Number of iterations that are to be stored by the auto save feature.
586	long	automatic_save_trigger_timeout	Number of microseconds that GageScope waits before triggering automatically while in the auto save mode.
587	long	automatic_save_i_s_delay	Number of microseconds that GageScope waits before triggering automatically while in the auto save mode.
588	char	automatic_save_file	The base filename to be used by the auto save feature.
589	int	printer_port	Which printer port is to be used. This variable when read is encoded in hex as 0x0DSP where D = the driver being used (0 = Epson FX, 1 = Epson MX, 2 = Epson LQ, 3 = Toshiba P, 4 = HP Laser Jet, 6 = HP Ink Jet), S = the size of the print out (0 = 11 x 8 1/2, 1 = 13 1/2 x 11) and P = the printer port in use (0 = FILE, 1 = LPT1, 2 = LPT2, 3 = LPT3, 4 = COM1, 5 = COM2).
590	int	printer_filename	What filename to use when printing to a file.
591	int	end_result	Arithmetic function end result.
592	int	operator_a	Arithmetic function first operator.
593	int	operand_1	Arithmetic function first operand.
594	int	operator_b	Arithmetic function second operator.
595	int	operand_2	Arithmetic function second operand.
596	int	operator_c	Arithmetic function third operator.
597	int	arith_enable	Arithmetic function enable.
598	int	fft_mode	0 = continuous operation, 1 = captured operation.
599	int	fft_channel_1 fft_channel_2	Number of the channel(s) for the FFT operation.
600	int	fft_points	Number of the points used from the source signal.
601	int	fft_window	Number of the window which is to"message" the input data.
602	float	fft_span	Frequency span of the FFT calculation.
603	float	fft_center_freq	Center frequency of the FFT calculation.
604	int	fft_start_time_sri	Index to the sample rate table for calculating the starting time to be applied to the FFT calculation. Note 5.
605	long	fft_start_time_points	Number of points from the trigger at the above sample rate used to calculate the starting time to be applied to the FFT calculation.
606	long	fft_db_max	Maximum display value for the dB vertical scale (2147483647 = Automatic scaling).
607	long	fft_db_min	Minimum display value for the dB vertical scale (2147483647 = Automatic scaling).
608	int	fft_calculate	Flag is set to perform the required FFT calculation.

609	int	xy_display_mode	0 = continuous operation, 1 = captured operation.
610	int	xy_display_x	Number of the channel for the X Axis.
611	int	xy_display_y	Number of the channel for the Y Axis.
612	int	xy_display_depth	Number of the points used from the source signals when in the continuous mode of operation.
613	int	xy_display_start_time_sri	Index to the sample rate table for calculating the starting time to be applied to the XY display in the captured mode of operation. Note 5.
614	long	xy_display_start_time_points	Number of points from the trigger at the above sample rate used to calculate the starting time to be applied to the XY display in the captured mode of operation.
615	int	xy_display_stop_time_sri	Index to the sample rate table for calculating the stoping time to be applied to the XY display in the captured mode of operation. Note 5.
616	long	xy_display_stop_time_points	Number of points from the trigger at the above sample rate used to calculate the stoping time to be applied to the XY display in the captured mode of operation.
617	int	xy_display_go	Flag is set to perform the required FFT calculation.
618	int	average_channel	The channel to be averaged. This channel is averaged and then displayed instead of the raw data captured from the hardware.
619	int	average_number_of_acquisitions	The number of samples to be averaged together to yield the resulting averaged waveform.
620	int	average_pre_points	The number of points before the trigger event, not including the trigger event (always less than or equal to zero).
621	int	average_post_points	The number of points after the trigger event, not including the trigger event (always greater than or equal to zero).
622	int	average_delta_points	The number of points to be averaged ((average post points - average pre points) + 1).
623	int	average_type	The mathimatical function to be appleid to the data when calculating the average (0 = normal average, 1 = RMS average).
624	int	average_enable	1 = average will be performed, 0 = no averaging.
625	long	average_depth	The actual trigger depth used by the hardware to ensure that there will be enough samples for average post points to be captured.
626	int	interpolate_trigger_channel	The flag to specify the use of the interpolated trigger (0 = normal operation, 1 = interpolated trigger).
627	char	data_directory_path	The directory path used by the program to save and restore data files (currently not used).
628	int	file_size_type	The file size type is set to: 0 when the data file is to be the smallest possible while saving all valid data; 1 when an image of the CompuScope memory is required; 2 when a specific pre and post amount of data is required.
629	long	file_size_max_pre_data	The maximum amount of data before the trigger event to store.
630	long	file_size_max_post_data	The maximum amount of data before the trigger event to store.

631	int	master_multiple_record	Set to a non-zero value when multiple record feature is enabled.
632	int	trigger_delay_time_sri	Index to the sample rate table for calculating the starting display point (in time) when the trigger delay is non-zero. Note 5.
633	long	trigger_delay_time_points	Number of points from the trigger at the above sample rate for calculating the starting display point (in samples) when the trigger delay is non-zero.
634	int	trigger_view_xy_timeout_flag	Determines if and how the trigger view display will timeout (0 = No time-out, 1 = Auto time-out, 2 = Timed time-out).

Note 1, the amplitude associated with each channel.

index	**amplitude**	**index**	**amplitude**
0	1000 V/d	9	1 V/d
1	500 V/d	10	500 mV/d
2	200 V/d	11	200 mV/d
3	100 V/d	12	100 mV/d
4	50 V/d	13	50 mV/d
5	20 V/d	14	20 mV/d
6	10 V/d	15	10 mV/d
7	5 V/d	16	5 mV/d
8	2 V/d		

Note 2, the time base associated with all channels.

index	**time base**	**index**	**time base**
0	10 nS/d	22	200 mS/d
1	20 nS/d	23	500 mS/d
2	50 nS/d	24	1 S/d
3	100 nS/d	25	2 S/d
4	200 nS/d	26	5 S/d
5	500 nS/d	27	10 S/d
6	1 uS/d	28	20 S/d
7	2 uS/d	29	50 S/d
8	5 uS/d	30	100 S/d
9	10 uS/d	31	200 S/d
10	20 uS/d	32	500 S/d
11	50 uS/d	33	1 KS/d
12	100 uS/d	34	2 KS/d
13	200 uS/d	35	5 KS/d
14	500 uS/d	36	10 KS/d
15	1 mS/d	37	20 KS/d
16	2 mS/d	38	50 KS/d
17	5 mS/d	39	100 KS/d
18	10 mS/d	40	200 KS/d
19	20 mS/d	41	500 KS/d
20	50 mS/d	42	1 MS/d
21	100 mS/d		

Note 3, the "probe" connected to the input channel and the external trigger.

if "file_version" is less than "GS V.2.15"

index	**probe multiplier**
0	x1
1	x10
2	x20
3	x100
4	x200

if "file_version" equals "GS V.2.15"

index	**probe multiplier**
0	x1
1	x10
2	x20
3	x50
4	x100
5	x200
6	x500
7	x1000

Note 4, the "gain" and "trigger_gain" input ranges.

if "file_version" equals "GS V.1.20"

index	**input range**
0	+/- 1v
1	+/- 200mv

if "file_version" equals "GS V.2.00"

index	**input range**
0	+/- 10v
1	+/- 5v
2	+/- 2v
3	+/- 1v
4	+/- 500mv
5	+/- 200mv
6	+/- 100mv

Note 5, the "sample_rate_index" to the sample rate table.

For all file versions		**Versions 2.65 and below**		**Versions 2.70 and above**		**Versions 2.85 and above**	
index	**sample rate**	**index**	**sample rate**	**index**	**sample rate**	**index**	**sample rate**
0	1 Hz	18	1 MHz	18	1 MHz	18	1 MHz
1	2 Hz	19	2 MHz	19	2 MHz	19	2 MHz
2	5 Hz	20	5 MHz	20	4 MHz	20	4 MHz
3	10 Hz	21	10 MHz21	5 MHz	21	5 MHz	
4	20 Hz	22	20 MHz22	10 MHz22	10 MHz		
5	50 Hz	23	25 MHz23	20 MHz23	12.5 MHz		
6	100 Hz	24	40 MHz24	25 MHz24	20 MHz		
7	200 Hz	25	50 MHz25	30 MHz25	25 MHz		
8	500 Hz	26	100 MHz	26	40 MHz26	30 MHz	
9	1 KHz			27	50 MHz27	40 MHz	
10	2 KHz			28	60 MHz28	50 MHz	
11	5 KHz			29	100 MHz	29	60 MHz
12	10 KHz			30	120 MHz	30	100 MHz
13	20 KHz			31	125 MHz	31	120 MHz
14	50 KHz			32	150 MHz	32	125 MHz
15	100 KHz			33	200 MHz	33	150 MHz
16	200 KHz			34	250 MHz	34	200 MHz
17	500 KHz			35	300 MHz	35	250 MHz
				36	500 MHz	36	300 MHz
				37	1 GHz	37	500 MHz
				38	2 GHz	38	1 GHz
				39	5 GHz	39	2 GHz
				40	External Clock	40	4 GHz
						41	5 GHz
						42	External Clock

File Version 2.80 and above

39	4 GHz
40	5 GHz
41	External Clock

Note 6, the "trigger_level" and the "captured_data" byte format.

The byte stored as the trigger level is in a different format from the data read back from the CompuScope hardware. The data is unsigned and the largest value (255) represents +1 volt and the smallest value (0) represents -1 volt. The input range and probe are then multiplied with the return value to produce the actual level either sent to the hardware as the trigger level or returned from it after capture.

Appendix M: CompuScope Data Transfer Benchmarks.

The following test results were obtained using a 66 MHz 80486DX2 IBM compatible computer configured as follows:

- ISA bus.
- AMIBIOS (1992) from American Megatrends.
- 128 Kb cache.
- 16 Megabytes of ram.
- 384 Kb of shadow ram.
- DOS 6.0
- Windows 3.1 running in Enhanced Mode.
- Smartdrive 4.0 with 2 Mb of cache under DOS and Windows.
- Qualitas 386Max version 7.0.

The tests were done with the 128Kb cache enabled and disabled, and with the bus operating at 8 MHz and at 16 MHz.

In dual channel mode, transfers from channel A and from channel B were measured separately and the results added together. The input signal was a 100 KHz sine wave and the CompuScope boards were operating at a 10 MHz sample rate. Each test was done three times and the average was used as the result.

The routine **gage_abort** was called just after entering and just prior to leaving the transfer routine. Timing measurements were made with an oscilloscope by monitoring the **abort** line of each of the boards.

The benchmark programs were written in C and compiled as large model for DOS and for Windows using both Borland C++ 3.1 and Microsoft Visual C++ 1.0. Two separate library routines, movedata and memcpy (fmemcpy for Windows), were also used.

CompuScope 1012

The board used was a 1M CompuScope CS1012 board, version 1.0. Measurements were taken at U18, pin 17. Measurements were taken for transfer of 25,000 samples from the CS1012 to a memory buffer in both dual and single channel modes.

Operating System	Compiler	Operating Mode	Bus Speed	Library Routine	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	memcpy	1,329,787	1,275,510
DOS	BC 3.1	DUAL	8 MHz	movedata	1,329,787	1,275,510
DOS	BC 3.1	DUAL	16 MHz	memcpy	Not Available	Not Available
DOS	BC 3.1	DUAL	16 MHz	movedata	Not Available	Not Available
DOS	BC 3.1	SINGLE	8 MHz	memcpy	1,315,789	1,275,510
DOS	BC 3.1	SINGLE	8 MHz	movedata	1,315,789	1,275,510
DOS	BC 3.1	SINGLE	16 MHz	memcpy	Not Available	Not Available
DOS	BC 3.1	SINGLE	16 MHz	movedata	Not Available	Not Available
DOS	VC 1.0	DUAL	8 MHz	memcpy	1,329,787	1,225,490
DOS	VC 1.0	DUAL	8 MHz	movedata	664,894	644,330
DOS	VC 1.0	DUAL	16 MHz	memcpy	Not Available	Not Available
DOS	VC 1.0	DUAL	16 MHz	movedata	Not Available	Not Available
DOS	VC 1.0	SINGLE	8 MHz	memcpy	1,315,789	1,219,512
DOS	VC 1.0	SINGLE	8 MHz	movedata	657,895	641,026
DOS	VC 1.0	SINGLE	16 MHz	memcpy	Not Available	Not Available
DOS	VC 1.0	SINGLE	16 MHz	movedata	Not Available	Not Available
WINDOWS	BC 3.1	DUAL	8 MHz	memcpy	1,201,923	1,041,667
WINDOWS	BC 3.1	DUAL	8 MHz	movedata	1,201,923	1,041,667
WINDOWS	BC 3.1	DUAL	16 MHz	memcpy	Not Available	Not Available
WINDOWS	BC 3.1	DUAL	16 MHz	movedata	Not Available	Not Available
WINDOWS	BC 3.1	SINGLE	8 MHz	memcpy	1,237,624	1,315,789
WINDOWS	BC 3.1	SINGLE	8 MHz	movedata	1,237,624	1,315,789
WINDOWS	BC 3.1	SINGLE	16 MHz	memcpy	Not Available	Not Available
WINDOWS	BC 3.1	SINGLE	16 MHz	movedata	Not Available	Not Available
WINDOWS	VC 1.0	DUAL	8 MHz	memcpy	1,201,923	1,041,667
WINDOWS	VC 1.0	DUAL	8 MHz	movedata	625,000	735,294
WINDOWS	VC 1.0	DUAL	16 MHz	memcpy	Not Available	Not Available
WINDOWS	VC 1.0	DUAL	16 MHz	movedata	Not Available	Not Available
WINDOWS	VC 1.0	SINGLE	8 MHz	memcpy	1,250,000	1,315,789
WINDOWS	VC 1.0	SINGLE	8 MHz	movedata	641,026	595,238
WINDOWS	VC 1.0	SINGLE	16 MHz	memcpy	Not Available	Not Available
WINDOWS	VC 1.0	SINGLE	16 MHz	movedata	Not Available	Not Available

CompuScope 250

The board used was a 2M CompuScope CS250 board, version 1.5. Measurements were taken at U37, pin 9. Measurements were taken for transfer of 50,000 samples from the CS250 to a memory buffer in both dual and single channel modes.

Operating System	Compiler	Operating Mode	Bus Speed	Library Routine	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	memcpy	1,063,830	1,041,666
DOS	BC 3.1	DUAL	8 MHz	movedata	1,063,830	1,041,666
DOS	BC 3.1	DUAL	16 MHz	memcpy	1,785,714	1,724,138
DOS	BC 3.1	DUAL	16 MHz	movedata	1,785,714	1,724,138
DOS	BC 3.1	SINGLE	8 MHz	memcpy	980,392	961,538
DOS	BC 3.1	SINGLE	8 MHz	movedata	980,392	961,538
DOS	BC 3.1	SINGLE	16 MHz	memcpy	1,639,344	1,612,903
DOS	BC 3.1	SINGLE	16 MHz	movedata	1,639,344	1,612,903
DOS	VC 1.0	DUAL	8 MHz	memcpy	1,041,666	1,020,408
DOS	VC 1.0	DUAL	8 MHz	movedata	892,857	862,069
DOS	VC 1.0	DUAL	16 MHz	memcpy	1,785,714	1,724,138
DOS	VC 1.0	DUAL	16 MHz	movedata	1,388,889	1,351,351
DOS	VC 1.0	SINGLE	8 MHz	memcpy	980,392	961,538
DOS	VC 1.0	SINGLE	8 MHz	movedata	877,193	862,069
DOS	VC 1.0	SINGLE	16 MHz	memcpy	1,623,377	1,587,302
DOS	VC 1.0	SINGLE	16 MHz	movedata	1,333,333	1,315,789
WINDOWS	BC 3.1	DUAL	8 MHz	fmemcpy	925,926	806,451
WINDOWS	BC 3.1	DUAL	8 MHz	movedata	925,926	806,451
WINDOWS	BC 3.1	DUAL	16 MHz	fmemcpy	1,488,095	1,219,512
WINDOWS	BC 3.1	DUAL	16 MHz	movedata	1,488,095	1,219,512
WINDOWS	BC 3.1	SINGLE	8 MHz	fmemcpy	909,090	833,333
WINDOWS	BC 3.1	SINGLE	8 MHz	movedata	909,090	833,333
WINDOWS	BC 3.1	SINGLE	16 MHz	fmemcpy	1,470,588	1,315,789
WINDOWS	BC 3.1	SINGLE	16 MHz	movedata	1,470,588	1,315,789
WINDOWS	VC 1.0	DUAL	8 MHz	fmemcpy	925,926	806,451
WINDOWS	VC 1.0	DUAL	8 MHz	movedata	793,651	714,286
WINDOWS	VC 1.0	DUAL	16 MHz	fmemcpy	1,488,095	1,219,512
WINDOWS	VC 1.0	DUAL	16 MHz	movedata	1,162,790	1,000,000
WINDOWS	VC 1.0	SINGLE	8 MHz	fmemcpy	909,090	819,672
WINDOWS	VC 1.0	SINGLE	8 MHz	movedata	825,082	746,269
WINDOWS	VC 1.0	SINGLE	16 MHz	fmemcpy	1,470,588	1,315,798
WINDOWS	VC 1.0	SINGLE	16 MHz	movedata	1,250,000	1,111,111

CompuScope 220

The board used was a 256K CompuScope CS220 board, version 1.5. Measurements were taken at U8, pin 6. Measurements were taken for transfer of 50,000 samples from the CS220 to a memory buffer in both dual and single channel modes.

Operating System	Compiler	Operating Mode	Bus Speed	Library Routine	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	memcpy	1,041,666	1,041,666
DOS	BC 3.1	DUAL	8 MHz	movedata	1,041,666	1,041,666
DOS	BC 3.1	DUAL	16 MHz	memcpy	1,785,714	1,736,111
DOS	BC 3.1	DUAL	16 MHz	movedata	1,785,714	1,736,111
DOS	BC 3.1	SINGLE	8 MHz	memcpy	1,063,830	1,041,666
DOS	BC 3.1	SINGLE	8 MHz	movedata	1,063,830	1,041,666
DOS	BC 3.1	SINGLE	16 MHz	memcpy	1,785,714	1,724,138
DOS	BC 3.1	SINGLE	16 MHz	movedata	1,785,714	1,724,138
DOS	VC 1.0	DUAL	8 MHz	memcpy	1,041,666	1,020,408
DOS	VC 1.0	DUAL	8 MHz	movedata	892,857	862,069
DOS	VC 1.0	DUAL	16 MHz	memcpy	1,785,714	1,689,189
DOS	VC 1.0	DUAL	16 MHz	movedata	1,358,696	1,302,083
DOS	VC 1.0	SINGLE	8 MHz	memcpy	1,063,830	1,041,666
DOS	VC 1.0	SINGLE	8 MHz	movedata	862,069	847,458
DOS	VC 1.0	SINGLE	16 MHz	memcpy	1,785,714	1,724,138
DOS	VC 1.0	SINGLE	16 MHz	movedata	1,351,351	1,315,789
WINDOWS	BC 3.1	DUAL	8 MHz	fmemcpy	815,660	707,214
WINDOWS	BC 3.1	DUAL	8 MHz	movedata	815,660	707,214
WINDOWS	BC 3.1	DUAL	16 MHz	fmemcpy	1,190,476	1,000,000
WINDOWS	BC 3.1	DUAL	16 MHz	movedata	1,190,476	1,000,000
WINDOWS	BC 3.1	SINGLE	8 MHz	fmemcpy	909,091	819,672
WINDOWS	BC 3.1	SINGLE	8 MHz	movedata	909,091	819,672
WINDOWS	BC 3.1	SINGLE	16 MHz	fmemcpy	1,428,571	1,240,694
WINDOWS	BC 3.1	SINGLE	16 MHz	movedata	1,428,571	1,240,694
WINDOWS	VC 1.0	DUAL	8 MHz	fmemcpy	823,723	714,286
WINDOWS	VC 1.0	DUAL	8 MHz	movedata	714,286	625,000
WINDOWS	VC 1.0	DUAL	16 MHz	fmemcpy	1,190,476	1,000,000
WINDOWS	VC 1.0	DUAL	16 MHz	movedata	974,659	833,333
WINDOWS	VC 1.0	SINGLE	8 MHz	fmemcpy	914,077	829,187
WINDOWS	VC 1.0	SINGLE	8 MHz	movedata	781,250	714,286
WINDOWS	VC 1.0	SINGLE	16 MHz	fmemcpy	1,428,571	1,240,694
WINDOWS	VC 1.0	SINGLE	16 MHz	movedata	1,118,568	994,036

CompuScope LITE

The board used was a 64K CompuScope LITE board, version 1.6. Measurements were taken at U7, pin 9. Measurements were taken for 50,000 samples from the CSLITE to a memory buffer in both single and dual channel modes.

Operating System	Compiler	Operating Mode	Bus Speed	Library Routine	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	memcpy	1,028,806	980,392
DOS	BC 3.1	DUAL	8 MHz	movedata	1,028,806	980,392
DOS	BC 3.1	DUAL	16 MHz	memcpy	1,689,189	1,633,987
DOS	BC 3.1	DUAL	16 MHz	movedata	1,689,189	1,633,987
DOS	BC 3.1	SINGLE	8 MHz	memcpy	1,052,631	1,052,631
DOS	BC 3.1	SINGLE	8 MHz	movedata	1,052,631	1,052,631
DOS	BC 3.1	SINGLE	16 MHz	memcpy	1,785,714	1,754,386
DOS	BC 3.1	SINGLE	16 MHz	movedata	1,785,714	1,785,385
DOS	VC 1.0	DUAL	8 MHz	memcpy	1,026,694	980,392
DOS	VC 1.0	DUAL	8 MHz	movedata	892,857	877,193
DOS	VC 1.0	DUAL	16 MHz	memcpy	1,724,138	1,602,564
DOS	VC 1.0	DUAL	16 MHz	movedata	1,381,215	1,315,789
DOS	VC 1.0	SINGLE	8 MHz	memcpy	1,048,218	1,041,667
DOS	VC 1.0	SINGLE	8 MHz	movedata	862,069	862,069
DOS	VC 1.0	SINGLE	16 MHz	memcpy	1,785,714	1,785,714
DOS	VC 1.0	SINGLE	16 MHz	movedata	1,351,351	1,315,789
WINDOWS	BC 3.1	DUAL	8 MHz	fmemcpy	1,041,667	1,000,000
WINDOWS	BC 3.1	DUAL	8 MHz	movedata	1,041,667	1,000,000
WINDOWS	BC 3.1	DUAL	16 MHz	fmemcpy	1,623,376	1,543,210
WINDOWS	BC 3.1	DUAL	16 MHz	movedata	1,623,376	1,543,210
WINDOWS	BC 3.1	SINGLE	8 MHz	fmemcpy	1,046,025	980,392
WINDOWS	BC 3.1	SINGLE	8 MHz	movedata	1,046,025	980,392
WINDOWS	BC 3.1	SINGLE	16 MHz	fmemcpy	1,754,386	1,666,666
WINDOWS	BC 3.1	SINGLE	16 MHz	movedata	1,754,386	1,666,666
WINDOWS	VC 1.0	DUAL	8 MHz	fmemcpy	1,041,666	1,000,000
WINDOWS	VC 1.0	DUAL	8 MHz	movedata	862,069	833,333
WINDOWS	VC 1.0	DUAL	16 MHz	fmemcpy	1,712,328	1,543,210
WINDOWS	VC 1.0	DUAL	16 MHz	movedata	1,329,787	1,275,510
WINDOWS	VC 1.0	SINGLE	8 MHz	fmemcpy	1,052,631	1,000,000
WINDOWS	VC 1.0	SINGLE	8 MHz	movedata	862,069	833,333
WINDOWS	VC 1.0	SINGLE	16 MHz	fmemcpy	1,754,386	1,666,666
WINDOWS	VC 1.0	SINGLE	16 MHz	movedata	1,333,333	1,250,000

Appendix N: Trigger View Transfer Benchmarks.

The following test results were obtained using a 66 MHz 80486DX2 IBM compatible computer configured as follows:

- ISA bus.
- AMIBIOS (1992) from American Megatrends.
- 128 Kb cache.
- 16 Megabytes of ram.
- 384 Kb of shadow ram.
- DOS 6.0
- Windows 3.1 running in Enhanced Mode.
- Smartdrive 4.0 with 2 Mb of cache under DOS and Windows.
- Qualitas 386Max version 7.0.

The tests were done with the 128Kb cache enabled and disabled, and with the bus operating at 8 MHz and at 16 Mhz (except for the CS1012).

Transfers were done using 512, 1024, 2048 and 4096 bytes in both dual and single channel modes. In dual channel mode, the transfers were done from both channel A and channel B. The resulting time was divided in half to get the results for one channel. In single channel mode, half the requested transfer was from channel A and the other half from channel B. For example, for a requested transfer of 512 bytes, in dual channel mode 512 bytes was transferred from channel A and 512 bytes from channel B. In single channel mode, 256 bytes were transferred from each channel. The input signal was a 100 KHz sine wave and the CompuScope boards were operating at a 10 MHz sample rate. Each test was done three times and the average was used as the result.

The routine **gage_abort** was called just after entering and just prior to leaving the transfer routine. Timing measurements were made with an oscilloscope by monitoring the **abort** line of each of the boards. On the CompuScope 1012, the routine **gage_multiple_record** was called instead and the **multiple_record** line was monitered.

The first set of tables show the transfer speed as samples per second, while the second set of tables show the time required to transfer the requested data.

The benchmark programs were written in C and compiled as large model for DOS and for Windows using both Borland C++ 3.1 and Microsoft Visual C++ 1.0.

CompuScope 1012

The board used was a 512K CompuScope CS1012 board, version 1.3. Measurements were taken at U4, pin 6. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS1012 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	512	1,137,778	731,429
DOS	BC 3.1	DUAL	8 MHz	1024	1,219,048	975,238
DOS	BC 3.1	DUAL	8 MHz	2048	1,280,000	1,107,027
DOS	BC 3.1	DUAL	8 MHz	4096	1,300,317	1,204,706
DOS	BC 3.1	SINGLE	8 MHz	512	984,615	501,961
DOS	BC 3.1	SINGLE	8 MHz	1024	1,137,778	731,429
DOS	BC 3.1	SINGLE	8 MHz	2048	1,219,048	975,238
DOS	BC 3.1	SINGLE	8 MHz	4096	1,280,000	1,107,027
DOS	VC 1.0	DUAL	8 MHz	512	1,024,000	533,333
DOS	VC 1.0	DUAL	8 MHz	1024	1,150,562	758,519
DOS	VC 1.0	DUAL	8 MHz	2048	1,241,212	975,238
DOS	VC 1.0	DUAL	8 MHz	4096	1,280,000	1,107,027
DOS	VC 1.0	SINGLE	8 MHz	512	825,806	328,205
DOS	VC 1.0	SINGLE	8 MHz	1024	1,024,000	533,333
DOS	VC 1.0	SINGLE	8 MHz	2048	1,150,562	758,519
DOS	VC 1.0	SINGLE	8 MHz	4096	1,241,212	975,238
WINDOWS	BC 3.1	DUAL	8 MHz	512	930,909	379,259
WINDOWS	BC 3.1	DUAL	8 MHz	1024	1,089,362	585,143
WINDOWS	BC 3.1	DUAL	8 MHz	2048	1,204,706	803,137
WINDOWS	BC 3.1	DUAL	8 MHz	4096	1,241,212	1,011,358
WINDOWS	BC 3.1	SINGLE	8 MHz	512	701,370	213,333
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	930,909	379,259
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	1,089,362	585,143
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	1,204,706	803,137
WINDOWS	VC 1.0	DUAL	8 MHz	512	914,286	379,259
WINDOWS	VC 1.0	DUAL	8 MHz	1024	1,089,362	585,143
WINDOWS	VC 1.0	DUAL	8 MHz	2048	1,204,706	803,137
WINDOWS	VC 1.0	DUAL	8 MHz	4096	1,241,212	999,024
WINDOWS	VC 1.0	SINGLE	8 MHz	512	701,370	213,333
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	914,286	365,714
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	1,089,362	585,143
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	1,204,706	803,137

CompuScope 250

The board used was a 32K CompuScope CS250 board, version 1.5. Measurements were taken at U37, pin 9. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS250 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	512	914,286	609,524
DOS	BC 3.1	DUAL	8 MHz	1024	975,238	758,519
DOS	BC 3.1	DUAL	8 MHz	2048	1,024,000	890,435
DOS	BC 3.1	DUAL	8 MHz	4096	1,024,000	963,765
DOS	BC 3.1	DUAL	16 MHz	512	1,462,857	769,925
DOS	BC 3.1	DUAL	16 MHz	1024	1,600,000	1,083,598
DOS	BC 3.1	DUAL	16 MHz	2048	1,706,667	1,365,333
DOS	BC 3.1	DUAL	16 MHz	4096	1,742,979	1,545,660
DOS	BC 3.1	SINGLE	8 MHz	512	800,000	426,667
DOS	BC 3.1	SINGLE	8 MHz	1024	775,756	609,524
DOS	BC 3.1	SINGLE	8 MHz	2048	975,238	772,830
DOS	BC 3.1	SINGLE	8 MHz	4096	1,024,000	890,435
DOS	BC 3.1	SINGLE	16 MHz	512	1,219,048	522,449
DOS	BC 3.1	SINGLE	16 MHz	1024	1,462,857	812,698
DOS	BC 3.1	SINGLE	16 MHz	2048	1,600,000	1,113,043
DOS	BC 3.1	SINGLE	16 MHz	4096	1,706,667	1,365,333
DOS	VC 1.0	DUAL	8 MHz	512	853,333	426,667
DOS	VC 1.0	DUAL	8 MHz	1024	930,909	602,353
DOS	VC 1.0	DUAL	8 MHz	2048	999,024	787,692
DOS	VC 1.0	DUAL	8 MHz	4096	1,024,000	890,435
DOS	VC 1.0	DUAL	16 MHz	512	1,280,000	538,947
DOS	VC 1.0	DUAL	16 MHz	1024	1,505,882	787,692
DOS	VC 1.0	DUAL	16 MHz	2048	1,683,400	1,107,027
DOS	VC 1.0	DUAL	16 MHz	4096	1,706,667	1,365,333
DOS	VC 1.0	SINGLE	8 MHz	512	691,892	275,269
DOS	VC 1.0	SINGLE	8 MHz	1024	853,333	439,485
DOS	VC 1.0	SINGLE	8 MHz	2048	930,909	607,715
DOS	VC 1.0	SINGLE	8 MHz	4096	999,024	787,692
DOS	VC 1.0	SINGLE	16 MHz	512	1,024,000	316,049
DOS	VC 1.0	SINGLE	16 MHz	1024	1,280,000	530,570
DOS	VC 1.0	SINGLE	16 MHz	2048	1,505,882	809,486
DOS	VC 1.0	SINGLE	16 MHz	4096	1,638,400	1,116,076

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
WINDOWS	BC 3.1	DUAL	8 MHz	512	752,941	320,000
WINDOWS	BC 3.1	DUAL	8 MHz	1024	890,435	499,512
WINDOWS	BC 3.1	DUAL	8 MHz	2048	975,238	671,475
WINDOWS	BC 3.1	DUAL	8 MHz	4096	1,024,000	819,200
WINDOWS	BC 3.1	DUAL	16 MHz	512	1,113,043	379,259
WINDOWS	BC 3.1	DUAL	16 MHz	1024	1,416,216	620,606
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1,593,774	930,909
WINDOWS	BC 3.1	DUAL	16 MHz	4096	1,696,066	1,187,246
WINDOWS	BC 3.1	SINGLE	8 MHz	512	595,349	189,630
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	752,941	330,323
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	890,435	499,512
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	975,238	682,667
WINDOWS	BC 3.1	SINGLE	16 MHz	512	787,692	213,333
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	1,113,043	379,259
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1,374,497	620,606
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	1,575,384	930,909
WINDOWS	VC 1.0	DUAL	8 MHz	512	752,941	320,000
WINDOWS	VC 1.0	DUAL	8 MHz	1024	890,435	499,512
WINDOWS	VC 1.0	DUAL	8 MHz	2048	975,238	682,667
WINDOWS	VC 1.0	DUAL	8 MHz	4096	999,024	819,200
WINDOWS	VC 1.0	DUAL	16 MHz	512	1,113,043	379,259
WINDOWS	VC 1.0	DUAL	16 MHz	1024	1,412,414	620,606
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1,575,385	930,909
WINDOWS	VC 1.0	DUAL	16 MHz	4096	1,682,136	1,187,246
WINDOWS	VC 1.0	SINGLE	8 MHz	512	595,349	189,630
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	775,758	320,000
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	890,435	499,512
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	975,238	682,667
WINDOWS	VC 1.0	SINGLE	16 MHz	512	812,698	213,333
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	1,121,577	379,259
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1,393,197	620,606
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	1,593,774	930,909

CompuScope 220

The board used was a 256K CompuScope CS220 board, version 1.5. Measurements were taken at U8, pin 6. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS220 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	512	914,286	568,889
DOS	BC 3.1	DUAL	8 MHz	1024	975,238	731,429
DOS	BC 3.1	DUAL	8 MHz	2048	1,024,000	871,489
DOS	BC 3.1	DUAL	8 MHz	4096	1,036,962	952,558
DOS	BC 3.1	DUAL	16 MHz	512	1,442,254	731,429
DOS	BC 3.1	DUAL	16 MHz	1024	1,600,000	1,044,898
DOS	BC 3.1	DUAL	16 MHz	2048	1,706,667	1,321,290
DOS	BC 3.1	DUAL	16 MHz	4096	1,742,979	1,517,037
DOS	BC 3.1	SINGLE	8 MHz	512	800,000	441,379
DOS	BC 3.1	SINGLE	8 MHz	1024	914,286	556,522
DOS	BC 3.1	SINGLE	8 MHz	2048	975,238	731,429
DOS	BC 3.1	SINGLE	8 MHz	4096	1,024,000	871,489
DOS	BC 3.1	SINGLE	16 MHz	512	1,190,698	457,143
DOS	BC 3.1	SINGLE	16 MHz	1024	1,442,254	731,429
DOS	BC 3.1	SINGLE	16 MHz	2048	1,600,000	1,044,898
DOS	BC 3.1	SINGLE	16 MHz	4096	1,706,667	1,321,290
DOS	VC 1.0	DUAL	8 MHz	512	839,344	445,217
DOS	VC 1.0	DUAL	8 MHz	1024	921,818	620,606
DOS	VC 1.0	DUAL	8 MHz	2048	999,024	787,692
DOS	VC 1.0	DUAL	8 MHz	4096	1,024,000	890,435
DOS	VC 1.0	DUAL	16 MHz	512	1,280,000	533,333
DOS	VC 1.0	DUAL	16 MHz	1024	1,505,882	819,200
DOS	VC 1.0	DUAL	16 MHz	2048	1,638,400	1,137,778
DOS	VC 1.0	DUAL	16 MHz	4096	1,706,667	1,365,333
DOS	VC 1.0	SINGLE	8 MHz	512	691,892	272,340
DOS	VC 1.0	SINGLE	8 MHz	1024	839,344	426,667
DOS	VC 1.0	SINGLE	8 MHz	2048	930,909	620,606
DOS	VC 1.0	SINGLE	8 MHz	4096	999,024	787,692
DOS	VC 1.0	SINGLE	16 MHz	512	984,615	312,195
DOS	VC 1.0	SINGLE	16 MHz	1024	1,264,198	533,333
DOS	VC 1.0	SINGLE	16 MHz	2048	1,505,882	819,200
DOS	VC 1.0	SINGLE	16 MHz	4096	1,638,400	1,137,778

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
WINDOWS	BC 3.1	DUAL	8 MHz	512	731,429	292,571
WINDOWS	BC 3.1	DUAL	8 MHz	1024	853,333	448,140
WINDOWS	BC 3.1	DUAL	8 MHz	2048	952,558	630,154
WINDOWS	BC 3.1	DUAL	8 MHz	4096	999,024	787,692
WINDOWS	BC 3.1	DUAL	16 MHz	512	1,066,667	341,333
WINDOWS	BC 3.1	DUAL	16 MHz	1024	1,347,368	564,187
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1,593,774	858,700
WINDOWS	BC 3.1	DUAL	16 MHz	4096	1,682,136	1,137,778
WINDOWS	BC 3.1	SINGLE	8 MHz	512	562,637	170,667
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	731,429	292,571
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	853,333	455,111
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	952,558	640,000
WINDOWS	BC 3.1	SINGLE	16 MHz	512	752,941	182,857
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	1,077,895	341,333
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1,347,368	558,038
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	1,575,385	853,333
WINDOWS	VC 1.0	DUAL	8 MHz	512	752,941	292,571
WINDOWS	VC 1.0	DUAL	8 MHz	1024	890,435	465,455
WINDOWS	VC 1.0	DUAL	8 MHz	2048	952,558	640,000
WINDOWS	VC 1.0	DUAL	8 MHz	4096	999,024	787,692
WINDOWS	VC 1.0	DUAL	16 MHz	512	1,066,667	341,333
WINDOWS	VC 1.0	DUAL	16 MHz	1024	1,347,368	558,038
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1,557,414	853,333
WINDOWS	VC 1.0	DUAL	16 MHz	4096	1,682,136	1,137,778
WINDOWS	VC 1.0	SINGLE	8 MHz	512	568,889	170,667
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	742,029	292,571
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	890,435	458,166
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	952,558	640,000
WINDOWS	VC 1.0	SINGLE	16 MHz	512	752,941	182,857
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	1,089,362	341,333
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1,347,368	558,038
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	1,534,082	835,918

CompuScope LITE

The board used was a 64K CompuScope LITE board, version 1.6. Measurements were taken at U7, pin 9. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CSLITE to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
DOS	BC 3.1	DUAL	8 MHz	512	957,009	726,241
DOS	BC 3.1	DUAL	8 MHz	1024	1,024,000	890,435
DOS	BC 3.1	DUAL	8 MHz	2048	999,024	910,222
DOS	BC 3.1	DUAL	8 MHz	4096	999,024	952,558
DOS	BC 3.1	DUAL	16 MHz	512	1,462,857	1,044,898
DOS	BC 3.1	DUAL	16 MHz	1024	1,505,882	1,312,821
DOS	BC 3.1	DUAL	16 MHz	2048	1,658,300	1,575,385
DOS	BC 3.1	DUAL	16 MHz	4096	1,731,924	1,575,385
DOS	BC 3.1	SINGLE	8 MHz	512	898,246	581,818
DOS	BC 3.1	SINGLE	8 MHz	1024	984,615	736,691
DOS	BC 3.1	SINGLE	8 MHz	2048	1,024,000	890,435
DOS	BC 3.1	SINGLE	8 MHz	4096	999,024	910,222
DOS	BC 3.1	SINGLE	16 MHz	512	1,383,784	752,941
DOS	BC 3.1	SINGLE	16 MHz	1024	1,505,882	1,066,667
DOS	BC 3.1	SINGLE	16 MHz	2048	1,551,515	1,248,780
DOS	BC 3.1	SINGLE	16 MHz	4096	1,575,385	1,412,414
DOS	VC 1.0	DUAL	8 MHz	512	930,909	591,908
DOS	VC 1.0	DUAL	8 MHz	1024	930,909	750,183
DOS	VC 1.0	DUAL	8 MHz	2048	975,238	853,333
DOS	VC 1.0	DUAL	8 MHz	4096	995,383	952,558
DOS	VC 1.0	DUAL	16 MHz	512	1,484,058	758,519
DOS	VC 1.0	DUAL	16 MHz	1024	1,551,515	1,066,667
DOS	VC 1.0	DUAL	16 MHz	2048	1,658,300	1,321,290
DOS	VC 1.0	DUAL	16 MHz	4096	1,638,400	1,444,797
DOS	VC 1.0	SINGLE	8 MHz	512	812,698	457,143
DOS	VC 1.0	SINGLE	8 MHz	1024	882,759	581,818
DOS	VC 1.0	SINGLE	8 MHz	2048	930,909	739,350
DOS	VC 1.0	SINGLE	8 MHz	4096	999,024	877,088
DOS	VC 1.0	SINGLE	16 MHz	512	1,163,636	512,000
DOS	VC 1.0	SINGLE	16 MHz	1024	1,365,333	769,925
DOS	VC 1.0	SINGLE	16 MHz	2048	1,517,037	1,061,140
DOS	VC 1.0	SINGLE	16 MHz	4096	1,658,300	1,280,000

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Samples/second	Cache Disabled Samples/second
WINDOWS	BC 3.1	DUAL	8 MHz	512	939,450	736,691
WINDOWS	BC 3.1	DUAL	8 MHz	1024	1,024,000	890,435
WINDOWS	BC 3.1	DUAL	8 MHz	2048	999,024	910,222
WINDOWS	BC 3.1	DUAL	8 MHz	4096	1,024,000	967,178
WINDOWS	BC 3.1	DUAL	16 MHz	512	1,575,385	1,024,000
WINDOWS	BC 3.1	DUAL	16 MHz	1024	1,651,613	1,219,048
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1,658,300	1,478,700
WINDOWS	BC 3.1	DUAL	16 MHz	4096	1,769,330	1,606,274
WINDOWS	BC 3.1	SINGLE	8 MHz	512	882,759	568,889
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	898,246	711,111
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	930,909	890,435
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	1,024,000	910,222
WINDOWS	BC 3.1	SINGLE	16 MHz	512	1,312,821	731,429
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	1,505,882	984,615
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1,551,515	1,280,000
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	1,638,400	1,447,350
WINDOWS	VC 1.0	DUAL	8 MHz	512	914,286	716,084
WINDOWS	VC 1.0	DUAL	8 MHz	1024	1,024,000	890,435
WINDOWS	VC 1.0	DUAL	8 MHz	2048	1,024,000	945,958
WINDOWS	VC 1.0	DUAL	8 MHz	4096	999,024	966,038
WINDOWS	VC 1.0	DUAL	16 MHz	512	1,462,857	984,615
WINDOWS	VC 1.0	DUAL	16 MHz	1024	1,721,008	1,575,385
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1,780,870	1,412,414
WINDOWS	VC 1.0	DUAL	16 MHz	4096	1,780,870	1,596,881
WINDOWS	VC 1.0	SINGLE	8 MHz	512	867,797	556,522
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	930,909	716,084
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	975,238	842,798
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	991,768	924,605
WINDOWS	VC 1.0	SINGLE	16 MHz	512	1,312,821	742,029
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	1,528,358	1,024,000
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1,517,037	1,280,000
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	1,780,870	1,517,037

CompuScope 1012

The board used was a 512K CompuScope CS1012 board, version 1.3. Measurements were taken at U4, pin 6. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS1012 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer. Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
DOS	BC 3.1	DUAL	8 MHz	512	0.45	0.70
DOS	BC 3.1	DUAL	8 MHz	1024	0.84	1.05
DOS	BC 3.1	DUAL	8 MHz	2048	1.60	1.85
DOS	BC 3.1	DUAL	8 MHz	4096	3.15	3.40
DOS	BC 3.1	SINGLE	8 MHz	512	0.52	1.02
DOS	BC 3.1	SINGLE	8 MHz	1024	0.90	1.40
DOS	BC 3.1	SINGLE	8 MHz	2048	1.68	2.10
DOS	BC 3.1	SINGLE	8 MHz	4096	3.20	3.70
DOS	VC 1.0	DUAL	8 MHz	512	0.50	0.96
DOS	VC 1.0	DUAL	8 MHz	1024	0.89	1.35
DOS	VC 1.0	DUAL	8 MHz	2048	1.65	2.10
DOS	VC 1.0	DUAL	8 MHz	4096	3.20	3.70
DOS	VC 1.0	SINGLE	8 MHz	512	0.62	1.56
DOS	VC 1.0	SINGLE	8 MHz	1024	1.00	1.92
DOS	VC 1.0	SINGLE	8 MHz	2048	1.78	2.70
DOS	VC 1.0	SINGLE	8 MHz	4096	3.30	4.20
WINDOWS	BC 3.1	DUAL	8 MHz	512	0.55	1.35
WINDOWS	BC 3.1	DUAL	8 MHz	1024	0.94	1.75
WINDOWS	BC 3.1	DUAL	8 MHz	2048	1.70	2.55
WINDOWS	BC 3.1	DUAL	8 MHz	4096	3.30	4.05
WINDOWS	BC 3.1	SINGLE	8 MHz	512	0.73	2.40
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	1.10	2.70
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	1.88	3.50
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	3.40	5.10
WINDOWS	VC 1.0	DUAL	8 MHz	512	0.56	1.35
WINDOWS	VC 1.0	DUAL	8 MHz	1024	0.94	1.75
WINDOWS	VC 1.0	DUAL	8 MHz	2048	1.70	2.55
WINDOWS	VC 1.0	DUAL	8 MHz	4096	3.30	4.10
WINDOWS	VC 1.0	SINGLE	8 MHz	512	0.73	2.40
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	1.12	2.80
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	1.88	3.50
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	3.40	5.10

CompuScope 250

The board used was a 32K CompuScope CS250 board, version 1.5. Measurements were taken at U37, pin 9. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS250 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
DOS	BC 3.1	DUAL	8 MHz	512	0.56	0.84
DOS	BC 3.1	DUAL	8 MHz	1024	1.05	1.35
DOS	BC 3.1	DUAL	8 MHz	2048	2.00	2.30
DOS	BC 3.1	DUAL	8 MHz	4096	4.00	4.25
DOS	BC 3.1	DUAL	16 MHz	512	0.35	0.67
DOS	BC 3.1	DUAL	16 MHz	1024	0.64	0.95
DOS	BC 3.1	DUAL	16 MHz	2048	1.20	1.50
DOS	BC 3.1	DUAL	16 MHz	4096	2.35	2.65
DOS	BC 3.1	SINGLE	8 MHz	512	0.64	1.20
DOS	BC 3.1	SINGLE	8 MHz	1024	1.32	1.68
DOS	BC 3.1	SINGLE	8 MHz	2048	2.10	2.65
DOS	BC 3.1	SINGLE	8 MHz	4096	4.00	4.60
DOS	BC 3.1	SINGLE	16 MHz	512	0.42	0.98
DOS	BC 3.1	SINGLE	16 MHz	1024	0.70	1.26
DOS	BC 3.1	SINGLE	16 MHz	2048	1.28	1.84
DOS	BC 3.1	SINGLE	16 MHz	4096	2.40	3.00
DOS	VC 1.0	DUAL	8 MHz	512	0.60	1.20
DOS	VC 1.0	DUAL	8 MHz	1024	1.10	1.70
DOS	VC 1.0	DUAL	8 MHz	2048	2.05	2.60
DOS	VC 1.0	DUAL	8 MHz	4096	4.00	4.60
DOS	VC 1.0	DUAL	16 MHz	512	0.40	0.95
DOS	VC 1.0	DUAL	16 MHz	1024	0.68	1.30
DOS	VC 1.0	DUAL	16 MHz	2048	1.25	1.85
DOS	VC 1.0	DUAL	16 MHz	4096	2.40	3.00
DOS	VC 1.0	SINGLE	8 MHz	512	0.74	1.86
DOS	VC 1.0	SINGLE	8 MHz	1024	1.20	2.33
DOS	VC 1.0	SINGLE	8 MHz	2048	2.20	3.37
DOS	VC 1.0	SINGLE	8 MHz	4096	4.10	5.20
DOS	VC 1.0	SINGLE	16 MHz	512	0.50	1.62
DOS	VC 1.0	SINGLE	16 MHz	1024	0.80	1.93
DOS	VC 1.0	SINGLE	16 MHz	2048	1.36	2.53
DOS	VC 1.0	SINGLE	16 MHz	4096	2.50	3.67

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
WINDOWS	BC 3.1	DUAL	8 MHz	512	0.68	1.60
WINDOWS	BC 3.1	DUAL	8 MHz	1024	1.15	2.05
WINDOWS	BC 3.1	DUAL	8 MHz	2048	2.10	3.05
WINDOWS	BC 3.1	DUAL	8 MHz	4096	4.00	5.00
WINDOWS	BC 3.1	DUAL	16 MHz	512	0.46	1.35
WINDOWS	BC 3.1	DUAL	16 MHz	1024	0.74	1.65
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1.29	2.20
WINDOWS	BC 3.1	DUAL	16 MHz	4096	2.42	3.45
WINDOWS	BC 3.1	SINGLE	8 MHz	512	0.86	2.70
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	1.36	3.10
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	2.30	4.10
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	4.20	6.00
WINDOWS	BC 3.1	SINGLE	16 MHz	512	0.65	2.40
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	0.92	2.70
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1.49	3.30
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	2.70	4.40
WINDOWS	VC 1.0	DUAL	8 MHz	512	0.68	1.60
WINDOWS	VC 1.0	DUAL	8 MHz	1024	1.15	2.05
WINDOWS	VC 1.0	DUAL	8 MHz	2048	2.10	3.00
WINDOWS	VC 1.0	DUAL	8 MHz	4096	4.10	5.00
WINDOWS	VC 1.0	DUAL	16 MHz	512	0.46	1.35
WINDOWS	VC 1.0	DUAL	16 MHz	1024	0.73	1.65
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1.30	2.20
WINDOWS	VC 1.0	DUAL	16 MHz	4096	2.44	3.45
WINDOWS	VC 1.0	SINGLE	8 MHz	512	0.86	2.70
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	1.32	3.20
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	2.30	4.10
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	4.20	6.00
WINDOWS	VC 1.0	SINGLE	16 MHz	512	0.63	2.40
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	0.92	2.70
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1.47	3.30
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	2.57	4.40

CompuScope 220

The board used was a 256K CompuScope CS220 board, version 1.5. Measurements were taken at U8, pin 6. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CS220 to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
DOS	BC 3.1	DUAL	8 MHz	512	0.56	0.90
DOS	BC 3.1	DUAL	8 MHz	1024	1.05	1.40
DOS	BC 3.1	DUAL	8 MHz	2048	2.00	2.35
DOS	BC 3.1	DUAL	8 MHz	4096	3.95	4.30
DOS	BC 3.1	DUAL	16 MHz	512	0.36	0.70
DOS	BC 3.1	DUAL	16 MHz	1024	0.64	0.98
DOS	BC 3.1	DUAL	16 MHz	2048	1.20	1.55
DOS	BC 3.1	DUAL	16 MHz	4096	2.35	2.70
DOS	BC 3.1	SINGLE	8 MHz	512	0.64	1.16
DOS	BC 3.1	SINGLE	8 MHz	1024	1.12	1.84
DOS	BC 3.1	SINGLE	8 MHz	2048	2.10	2.80
DOS	BC 3.1	SINGLE	8 MHz	4096	4.00	4.70
DOS	BC 3.1	SINGLE	16 MHz	512	0.43	1.12
DOS	BC 3.1	SINGLE	16 MHz	1024	0.71	1.40
DOS	BC 3.1	SINGLE	16 MHz	2048	1.28	1.96
DOS	BC 3.1	SINGLE	16 MHz	4096	2.40	3.10
DOS	VC 1.0	DUAL	8 MHz	512	0.61	1.15
DOS	VC 1.0	DUAL	8 MHz	1024	1.10	1.65
DOS	VC 1.0	DUAL	8 MHz	2048	2.05	2.60
DOS	VC 1.0	DUAL	8 MHz	4096	4.00	4.60
DOS	VC 1.0	DUAL	16 MHz	512	0.40	0.96
DOS	VC 1.0	DUAL	16 MHz	1024	0.68	1.25
DOS	VC 1.0	DUAL	16 MHz	2048	1.25	1.80
DOS	VC 1.0	DUAL	16 MHz	4096	2.40	3.00
DOS	VC 1.0	SINGLE	8 MHz	512	0.74	1.88
DOS	VC 1.0	SINGLE	8 MHz	1024	1.22	2.40
DOS	VC 1.0	SINGLE	8 MHz	2048	2.20	3.30
DOS	VC 1.0	SINGLE	8 MHz	4096	4.10	5.20
DOS	VC 1.0	SINGLE	16 MHz	512	0.52	1.64
DOS	VC 1.0	SINGLE	16 MHz	1024	0.81	1.92
DOS	VC 1.0	SINGLE	16 MHz	2048	1.36	2.50
DOS	VC 1.0	SINGLE	16 MHz	4096	2.50	3.60

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
WINDOWS	BC 3.1	DUAL	8 MHz	512	0.70	1.75
WINDOWS	BC 3.1	DUAL	8 MHz	1024	1.20	2.29
WINDOWS	BC 3.1	DUAL	8 MHz	2048	2.15	3.25
WINDOWS	BC 3.1	DUAL	8 MHz	4096	4.10	5.20
WINDOWS	BC 3.1	DUAL	16 MHz	512	0.48	1.50
WINDOWS	BC 3.1	DUAL	16 MHz	1024	0.76	1.82
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1.30	2.39
WINDOWS	BC 3.1	DUAL	16 MHz	4096	2.44	3.60
WINDOWS	BC 3.1	SINGLE	8 MHz	512	0.91	3.00
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	1.40	3.50
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	2.40	4.50
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	4.30	6.40
WINDOWS	BC 3.1	SINGLE	16 MHz	512	0.68	2.80
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	0.95	3.00
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1.52	3.67
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	2.60	4.80
WINDOWS	VC 1.0	DUAL	8 MHz	512	0.68	1.75
WINDOWS	VC 1.0	DUAL	8 MHz	1024	1.15	2.20
WINDOWS	VC 1.0	DUAL	8 MHz	2048	2.15	3.20
WINDOWS	VC 1.0	DUAL	8 MHz	4096	4.10	5.20
WINDOWS	VC 1.0	DUAL	16 MHz	512	0.48	1.50
WINDOWS	VC 1.0	DUAL	16 MHz	1024	0.76	1.84
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1.32	2.40
WINDOWS	VC 1.0	DUAL	16 MHz	4096	2.44	3.60
WINDOWS	VC 1.0	SINGLE	8 MHz	512	0.90	3.00
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	1.38	3.50
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	2.30	4.47
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	4.30	6.40
WINDOWS	VC 1.0	SINGLE	16 MHz	512	0.68	2.80
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	0.94	3.00
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1.52	3.67
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	2.67	4.90

CompuScope LITE

The board used was a 64K CompuScope LITE board, version 1.6. Measurements were taken at U7, pin 9. Measurements were taken for transfers of 512, 1024, 2048 and 4096 samples from the CSLITE to a memory buffer in both dual and single channel modes using **gage_trigger_view_transfer**.

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
DOS	BC 3.1	DUAL	8 MHz	512	0.54	0.72
DOS	BC 3.1	DUAL	8 MHz	1024	1.00	1.15
DOS	BC 3.1	DUAL	8 MHz	2048	2.05	2.25
DOS	BC 3.1	DUAL	8 MHz	4096	4.10	4.30
DOS	BC 3.1	DUAL	16 MHz	512	0.35	0.49
DOS	BC 3.1	DUAL	16 MHz	1024	0.68	0.78
DOS	BC 3.1	DUAL	16 MHz	2048	1.24	1.30
DOS	BC 3.1	DUAL	16 MHz	4096	2.37	2.60
DOS	BC 3.1	SINGLE	8 MHz	512	0.57	0.88
DOS	BC 3.1	SINGLE	8 MHz	1024	1.04	1.39
DOS	BC 3.1	SINGLE	8 MHz	2048	2.00	2.30
DOS	BC 3.1	SINGLE	8 MHz	4096	4.10	4.50
DOS	BC 3.1	SINGLE	16 MHz	512	0.37	0.68
DOS	BC 3.1	SINGLE	16 MHz	1024	0.68	0.96
DOS	BC 3.1	SINGLE	16 MHz	2048	1.32	1.64
DOS	BC 3.1	SINGLE	16 MHz	4096	2.60	2.90
DOS	VC 1.0	DUAL	8 MHz	512	0.55	0.87
DOS	VC 1.0	DUAL	8 MHz	1024	1.10	1.37
DOS	VC 1.0	DUAL	8 MHz	2048	2.10	2.40
DOS	VC 1.0	DUAL	8 MHz	4096	4.12	4.30
DOS	VC 1.0	DUAL	16 MHz	512	0.35	0.68
DOS	VC 1.0	DUAL	16 MHz	1024	0.66	0.96
DOS	VC 1.0	DUAL	16 MHz	2048	1.24	1.55
DOS	VC 1.0	DUAL	16 MHz	4096	2.50	2.84
DOS	VC 1.0	SINGLE	8 MHz	512	0.63	1.12
DOS	VC 1.0	SINGLE	8 MHz	1024	1.16	1.76
DOS	VC 1.0	SINGLE	8 MHz	2048	2.20	2.77
DOS	VC 1.0	SINGLE	8 MHz	4096	4.10	4.67
DOS	VC 1.0	SINGLE	16 MHz	512	0.44	1.00
DOS	VC 1.0	SINGLE	16 MHz	1024	0.75	1.33
DOS	VC 1.0	SINGLE	16 MHz	2048	1.35	1.93
DOS	VC 1.0	SINGLE	16 MHz	4096	2.47	3.20

Operating System	Compiler	Operating Mode	Bus Speed	Transfer Size	Cache Enabled Transfer Speed (milliseconds)	Cache Disabled Transfer Speed (milliseconds)
WINDOWS	BC 3.1	DUAL	8 MHz	512	0.55	0.70
WINDOWS	BC 3.1	DUAL	8 MHz	1024	1.00	1.15
WINDOWS	BC 3.1	DUAL	8 MHz	2048	2.05	2.25
WINDOWS	BC 3.1	DUAL	8 MHz	4096	4.00	4.24
WINDOWS	BC 3.1	DUAL	16 MHz	512	0.33	0.50
WINDOWS	BC 3.1	DUAL	16 MHz	1024	0.62	0.84
WINDOWS	BC 3.1	DUAL	16 MHz	2048	1.24	1.39
WINDOWS	BC 3.1	DUAL	16 MHz	4096	2.32	2.55
WINDOWS	BC 3.1	SINGLE	8 MHz	512	0.58	0.90
WINDOWS	BC 3.1	SINGLE	8 MHz	1024	1.14	1.44
WINDOWS	BC 3.1	SINGLE	8 MHz	2048	2.20	2.30
WINDOWS	BC 3.1	SINGLE	8 MHz	4096	4.00	4.50
WINDOWS	BC 3.1	SINGLE	16 MHz	512	0.39	0.70
WINDOWS	BC 3.1	SINGLE	16 MHz	1024	0.68	1.04
WINDOWS	BC 3.1	SINGLE	16 MHz	2048	1.32	1.60
WINDOWS	BC 3.1	SINGLE	16 MHz	4096	2.50	2.83
WINDOWS	VC 1.0	DUAL	8 MHz	512	0.56	0.72
WINDOWS	VC 1.0	DUAL	8 MHz	1024	1.00	1.15
WINDOWS	VC 1.0	DUAL	8 MHz	2048	2.00	2.17
WINDOWS	VC 1.0	DUAL	8 MHz	4096	4.10	4.24
WINDOWS	VC 1.0	DUAL	16 MHz	512	0.35	0.50
WINDOWS	VC 1.0	DUAL	16 MHz	1024	0.60	0.80
WINDOWS	VC 1.0	DUAL	16 MHz	2048	1.15	1.45
WINDOWS	VC 1.0	DUAL	16 MHz	4096	2.30	2.57
WINDOWS	VC 1.0	SINGLE	8 MHz	512	0.59	0.92
WINDOWS	VC 1.0	SINGLE	8 MHz	1024	1.10	1.43
WINDOWS	VC 1.0	SINGLE	8 MHz	2048	2.10	2.43
WINDOWS	VC 1.0	SINGLE	8 MHz	4096	4.13	4.43
WINDOWS	VC 1.0	SINGLE	16 MHz	512	0.39	0.69
WINDOWS	VC 1.0	SINGLE	16 MHz	1024	0.67	1.00
WINDOWS	VC 1.0	SINGLE	16 MHz	2048	1.35	1.60
WINDOWS	VC 1.0	SINGLE	16 MHz	4096	2.30	2.70

Appendix O: Resolving Memory Conflicts

You may need to choose alternate I/O and memory segment addresses for one or more board(s) if other devices and/or programs are using those specified as defaults. Also, both I/O and memory segment addresses must be different for each installed board, so for multiple board installations you will have to choose alternate addresses for the second and any subsequent boards.

You may also encounter memory segment address conflicts when using certain memory management programs. See the next section for tips on how to handle these.

Memory Conflicts with SVGA cards

In the past, some CompuScope users have had problems getting CompuScope boards to work properly in their 386 or 486 machines with SuperVGA graphics adapters. We encountered the same problem with an IBM-compatible 80386 PC AT machine.

A number of generic 80386 and 486 machines come with non-standard SuperVGA cards which use "high speed modes" to increase the speed of the graphics display. The "high speed mode", also known as "fast decode mode", forces the SuperVGA card to interfere with all memory decoding between 640K and 1M, thus causing a memory conflict with CompuScope cards which require a 4 KB window in this area of the memory.

In our machine it was the D000 address (the default memory segment address) that was in conflict, and one user had a conflict with B000.

Most manufacturers of Super VGA cards allow the user to disable the "high speed mode" using one of the DIP switches on the board. Consult the manual for your Super VGA card and set the DIP switches accordingly. If you do not have a manual or if the card does not allow this mode to be disabled, insert your Super VGA card in an 8 bit slot in your PC AT; this invariably disables all non-standard modes.

Memory Conflicts with EMM386

Once we got our CompuScope running in the 80386 machine, we decided to install a memory manager. The EMM386 expanded memory manager was automatically installed by the MEMMAKER program which came with MSDOS 6.0.

After installing EMM386, we noticed that the CompuScope board had stopped functioning properly again. This time the reason was that EMM386 had taken up all the upper memory blocks to install some of the Extended memory for running DOS drivers, TSRs, etc. under the guise of “optimizing” the computer.

MEMMAKER had changed the first line of the CONFIG.SYS file to:

```
DEVICE=C:\DOS\EMM386.EXE NOEMS
```

This means that all the upper memory blocks are automatically dedicated to loading TSRs and MSDOS devices. The drivers needs 4 kilobyte window for running CS250, CS225, CS220 and CSLITE boards, or an 8 kilobyte window for running the CS6012 and CS1012 boards, in this upper memory block area to communicate with the on-board RAM of each installed CompuScope board.

If all of the upper memory blocks are taken up by EMM386, CompuScope card(s) will not operate properly.

Again the solution is simple. To operate a CompuScope board at the D000 memory address, just change the first line of the CONFIG.SYS file to:

```
DEVICE=C:\DOS\EMM386.EXE NOEMS X=D000-D100     (for CS250, CS225, CS220 and CSLITE)
```

or

```
DEVICE=C:\DOS\EMM386.EXE NOEMS X=D000-D200     (for CS6012 and CS1012)
```

This tells the EMM386 driver to exclude the specified range of addresses (4 or 8 Kilobytes). That is, to leave it unused so the CompuScope board can use it. If you are trying to operate the CompuScope board at any other memory location that does not work, specify that range in place of D000-D100.

Note that, in some computers, you may not be able to load EMM386 after having excluded the D000-D100 or D000-D200 area, as that is the page frame address for EMM386. In such cases, it is advisable to try and operate the CompuScope card at a different address and to exclude this other address in the CONFIG.SYS file.

Memory Conflicts with 386MAX

Yet another problem was faced by our software engineering team when they first loaded their machines with 386MAX, a sophisticated extended and expanded memory manager popular with software engineers due to its superior performance.

The problem was the same: 386MAX had taken up all the upper memory blocks for its own use, leaving no memory for the CompuScope cards to use. The solution was slightly different: 386MAX uses a file called 386MAX.PRO to store its profile and the addition of the following line was sufficient to instruct it to leave free the 4 kilobyte area starting at D000:0000 :

```
RAM = D000-D100                                    (for CS250, CS225, CS220 and CSLITE)
```

or

```
RAM = D000-D200                                    (for CS6012 and CS1012).
```

Memory Conflicts with QEMM386

QEMM386 is a memory manager supplied by Quarterdeck Office Systems. It too takes up all the upper memory blocks for DOS drivers, TSRs etc. under the guise of "optimizing" the system.

This causes exactly the same symptoms as the ones described above for EMM386, and the solution is also exactly the same. Just change the first line in CONFIG.SYS from

```
DEVICE = QEMM386.SYS RAM
```

to

```
DEVICE = QEMM386.SYS RAM X=D000-D100     (for CS250, CS225, CS220 and CSLITE)
```

or to

```
DEVICE = QEMM386.SYS RAM X=D000-D200     (for CS6012 and CS1012)
```

Memory Conflicts with motherboard chipsets

Some motherboard chipsets can also cause a conflict with memory between 640K and 1M. These conflicts are caused by non-standard timing and gating of signals in the chipset and can always be fixed by disabling the appropriate options in the Advanced CMOS Setup of the PC.

Some of the conflicts we have seen are : Cyrix Coprocessor Cache Enable, Fast Gate A20 Option, Decoupled Refresh Option etc. **It must be noted that these options do not always cause a conflict**.

In general, if you are having trouble running a CompuScope board, try the following:

- Remove all other boards from your computer, leaving only the motherboard, the VGA card and the hard disk controller
- Disable any "high speed modes" on your Super VGA card
- Disable any memory managers running in the system
- If the board still does not work, change the memory segment address using GSINST
- If the board still does not work, disable all options in the Advanced CMOS Setup of the BIOS in your PC.
- Once you get the board working without the memory manager, re-install the manager, but with the “X” parameter, specifying the memory address range taken up by the board or boards.
- Insert all other boards in the computer, one at a time, and check for any conflicts using GSINST.

Technical Support.

Gage Applied Science Inc. offers free technical support for all its drivers. Technical support is available by phone at 514-337-6893 from 9:00 A.M. to 5:00 P.M. Eastern Standard Time, Monday to Friday. Support is also available by fax at 514-337-8411 or through our bulletin board service at 514-337-4317.
The BBS operates at speeds of up to 14.4 K baud, with 8 data bits, 1 stop bit and no parity.

When calling for support we ask that you have the following information available:

Version and type of your CompuScope drivers.
    The version number can be obtained at the top of any of the driver source files or on the distribution diskette.

Type, version and memory depth of your CompuScope card.

Type and version of your operating system.

Type and speed of your computer and bus (ie. ISA, EISA, LOCAL BUS)

Contents of your CONFIG.SYS and AUTOEXEC.BAT files.

Any extra hardware peripherals (ie. CD-ROM, joystick, network card, etc.)

Were you able to reproduce the problem with GAGESCOP.EXE.?

If the problem is with an application program you are writing, the simplest approach is often to either send us some of the code you are having problems with, along with other details such as sample rate, trigger source, input signal, etc., either by fax or through our bulletin board. This way, we can try to reproduce the problem.