GUZIK INTEGRATED TEST ENVIRONMENT (GITE)

I. INTRODUCTION.

GITE is designed to ease access to various functions of Guzik Read-Write Analyzers, reduce development time of new tests and help achieve upward compatibility of tests with future models of the RWA and other equipment.

GITE is developed under DESQview multitasking system. DESQview allows several tasks (in DESQview terminology also called applications) to run at the same time. All applications reside in conventional and extended memory, and control can be quickly switched from one to another. On the screen, each application has its own window, which can overlap and partially or completely hide windows of other applications. Operator keystrokes are passed to the application which is currently selected as foreground. With some limitations, DESQview allows such selection, invocation of new applications, and deletion of running ones, to be done from any running application as well as directly from the keyboard. An application can also put itself into a "sleeping" mode, waiting for environment events without using the CPU, which is a shared resource.

GITE uses separate applications to handle different instruments in the test configuration, including RWA; and to allow a test program, running as a yet another application, the use of various functions provided by the handlers. Handler applications could have been called drivers of their respective hardware devices but for the fact that the level of functions they perform is significantly higher than is customary for device drivers; which in turn increases their size and, consequently, pulls in a multitasking environment like DESQview. The advantages of this approach include:

- the environment is easily customized for a particular test hardware configuration by simply loading appropriate handlers;
- handlers help achieve compatibility between different models of hardware by providing an additional level at which the differences may be buffered;
- the high level of functions supported by the handlers makes test applications themselves simple and easy to modify.

However, because of their size, the handlers not only use up memory, but require a multitasking environment with intertask communication, which slows the operation down as compared to a single task.

The intertask communication capabilities, provided by DESQview through its Application Program Interface (API) library, are crucial for the operation of GITE. Extended memory requirements are also high: GITE requires that all active applications reside in memory, which calls for a 386-compatible machine with at least 2Mbytes RAM.

II. THE INTERPRETER.

To reduce dependence of applications upon one another, GITE introduces a special application called the Interpreter, which acts as a communication manager for the applications that choose to use its services. Through the Interpreter, any application can:

- declare its presence and inquire what other applications have done likewise;
- declare some of its variables read- or read/write-accessible to other applications, and access variables in other applications that have been likewise declared;
- declare some of its functions invocable by other applications and invoke functions in other applications that have been likewise declared, passing parameters and getting back return values, if any;
- load repeatedly used command sequences into the Interpreter and into other conforming applications, and then invoke them by a single command to speed up execution.

No multi-application protection is implemented: it is not possible to make functions or variables available to some applications but not others. The environment is not recursive: an application may not, directly or indirectly, use the Interpreter to access its own variables and invoke its own routines. Recursion may be introduced in future releases, however.

Applications communicate their declarations and requests to the Interpreter through GITE library routines (and, on a lower level, DESQview API routines). Requests coming to the application from the Interpreter, including variable read/write and function invocation requests, are processed by the GITE library. No user code is required to handle these requests.

The act of declaring a function or a variable accessible, through the Interpreter, to other applications, is called, in GITE terminology, "installing a symbol in the Interpreter". The information supplied to the Interpreter during installation includes (but is not limited to) a (unique within an application) name by which the symbol shall be known and accessed, symbol type (function or variable) and the address of the object in question.

The Interpreter maintains a table of installed symbols. When the Interpreter receives a request from an application to access a symbol (also called an Interpreter command), it searches through the table for a symbol of this name, and sends the request for an action to the application which installed the symbol. When the requested memory read/write or function execution is completed, the application notifies the Interpreter and sends it a return value, if any. The Interpreter then relays the return value to the requester, and tells it the operation is completed. The requester application may not issue new commands before this.

No new request, not even an abort request, is considered by an application while it is actively executing. The next request, if any, is processed when the application is in, or is preparing to switch itself into, wait state. An application switches itself into wait state when its processing of the last incoming request is completed, or when it is awaiting completion of a request of its own.

This Interpreter-assisted dynamic linking approach makes requests independent from particular addresses in applications, which may change when applications are modified.

Environment operations may be speeded up if a repeatedly used command sequence is communicated to the Interpreter in advance. The sequence (called a program) is stored inside the Interpreter in a partially compiled form, thus saving not only requester-Interpreter communications, but also most of the run-time table search during repeated invocations. If a program uses symbols of only one application, the efficiency may be further increased by advance downloading it into this application; in this case both table searches and intertask communications are completely eliminated at run-time and the time overhead is reduced to a minimum. The amount of memory taken by such programs is not significant; the principal limitation rather stems from the fact that, as implemented, Interpreter programs take no parameters, and give back no return values.

The Interpreter accepts a startup batch file with Interpreter commands. The file, if any, must be specified on the DOS command line when the Interpreter is invoked, in quotes and with a "-s" flag (example: '-s "intstrt.prg"'). (DESQview supports command line parameters for applications running under it.) The Interpreter executes commands in the startup batch file before it begins taking commands from other applications. The startup file is usually used to automatically activate certain applications and make foreground the application which functions as user interface.

III. REQUEST FORMATS.

Any application wishing to access a variable or invoke a function installed by another application uses a GITE library routine to send a request string to the Interpreter and fetch a return value, if expected. The requests include the symbol name with which the object had been installed by another application. If several applications had installed symbols with the same name, the request will be resolved to the object which had been installed first, unless the request specifies the desired application name in the form APPLICATIONNAME.SYMBOLNAME. A reference to a symbol that had not been installed creates an error condition or is ignored, according to the settings of the Interpreter (see "symchk" command).

The formats of the requests are:

SYMBOLNAME

To examine a variable. A return value is produced.

SYMBOLNAME newvalue

To change a variable. No return value.

SYMBOLNAME value1 value2 value3 keyword1:value4 keyword2:value5

To invoke a function. The values listed after SYMBOLNAME are sequentially assigned as actual values for parameters one, two, etc. of the function call. After the sequence, values preceded by keywords are assigned as actual values for parameters associated with the respective keywords. Both the sequential and the keyworded parts are optional. If a parameter is not assigned any value by either, the default

value is substituted by the Interpreter, or an error condition is declared, according to the settings of the Interpreter (see "parchk" command). If a value is assigned to a parameter more than once, the right-most assignment overrides previous ones. An unknown keyword, or a surplus of parameters in the sequence, however, constitute an error. The number of parameters, their associated keywords and defaults are among the information supplied to the Interpreter when a function symbol is being installed. A return value may or may not be produced, depending on the nature of the function.

Several commands may be given on a single line, separated by semicolons. If any parameter in a command is meant to be a string but contains semicolons, tabulations, spaces, or may be mistaken for a number, it must be quoted. If a return value is produced, it will be lost unless fetched by the application before it issues its next Interpreter command; an attempt to fetch a value where none is produced results in a timeout condition.

IV. VALIDATION.

Validation is an additional service provided by the Interpreter. For every request for modification of a variable, the new value is checked against certain rules; for every request for a function invocation, the number of parameters and actual values of each parameter are checked, as well as the function return value. Any request or return value which does not pass the check is rejected; it is not sent to the application it would otherwise be sent to, and the Interpreter declares an environment error condition.

Rules used in validation are specified in the form of validation strings, separately and independently for every function, parameter and variable, during symbol installation. Value types (long int, short int, double float, or string) are also specified at installation time. The system considers these types (called target types), would-be values (source values), and validation strings, and tries to reconcile them.

If the target type is a string, the value is considered valid if, and only if, the source is also a string; any specified validation is ignored. Validation for a numerical target can be bypassed by specifying an empty validation string.

For numerical targets, three different types of validation expressions are supported: comparative, enumerated, and defining. A validation string contains one or more validation expressions of similar or different types, separated by semicolons. The expressions are scanned from left to right, until the source satisfies one of them.

A comparative validation expression is a combination like ">10&<20||==1" (meaning "greater than 10 and less than 20 or equal to 1). >, <, >=, <=, ==, and != are the comparisons supported; & (and) and || (or) are performed from left to right, with equal priority.

"1-YELLOW" is an example of a defining validation. A string source "YELLOW" would, by this expression, considered valid, with the numerical target being set equal to 1. When validating a return value, "1" is converted into "YELLOW". "1 @ YELLOW GREEN RED" (enumerated validation) is equal to "1-YELLOW;2-GREEN;3-RED" (the values assigned to words begin with the one specified and continue with increments of 1). Negative and non-integer values ar also supported.

V. ERROR RECOVERY.

Any running application can at any time declare an environmental error condition. The Interpreter may also declare the error condition, in the event of an undefined symbol, for example. When the condition is declared, the Interpreter sends every application known to it a special abort message. The abort message does not say which application had declared the error condition, or what error message had been put forward. Such information is sent, however, to the application currently selected as the global error message recipient (if any is so selected), which is also made foreground at this moment. The global error message recipient is selected by "apIsetmsg" Interpreter command (see); any application named "KEYBOARD" automatically assumes this status when loaded.

Inside applications, the error recovery is implemented using the C "long jump" mechanism (see an appropriate C manual for more information on long jumps). When an application receives an abort message, GITE routines inside it execute a standard C longjump function "longjmp(dv_jbuf,-1)". dv_jbuf must be declared as an external long jump buffer ("extern jmp_buf dv_jbuf;"; jmp_buf is a type defined in a standard C library file "setjmp.h"). It is essential for the correct error recovery that applications set return points for this longjump according to the GITE convention outlined in the section on GITE-aware applications. When the return points are so set, the longjump executed by an application on receiving an abort message makes the application give up any pending requests it had issued, abandon any request it was executing, reinitialize itself, and enter wait state, ready for new incoming requests. Since at an environmental error condition all applications known to the Interpreter receive an abort message, this condition leads to all of them aborting pending requests and reinitializing themselves.

VI. PROGRAMMING FEATURES.

A set of commands may be defined for the Interpreter as a program for future execution. When a program is being defined, it is stored inside the Interpreter and partially compiled. Afterwards a requester application can invoke it with a single command. This eliminates unneeded requester-Interpreter communications and table searches at run-time. Interpreter-executor communications are not affected in this case; but they may be dispensed with as well if a program contains references to symbols of one application only: the Interpreter may then download the program into this application.

The following Interpreter commands are used to manage programs. The APPLICATIONNAME parameter is put in brackets to show it as optional (the brackets are not part of the syntax). When this parameter is present, commands refer to programs downloaded into the application; otherwise they deal with programs stored in the Interpreter.

prog PROGRAMNAME [APPLICATIONNAME]

Begins a program definition.

label LABELNAME

Set a label at this point.

jz LABELNAME

jnz LABELNAME

jump LABELNAME

Jump to the label if the return value of the last request was zero / not zero / unconditionally.

ret

Return (the same as a jump to the end of the program).

endp

Ends program definition.

call PROGRAMNAME [APPLICATIONNAME]

Invoke a program already downloaded into an application / stored in the Interpreter.

dirp

Output the directory of programs stored in the Interpreter (programs downloaded into applications are not included in the list).

delp PROGRAMNAME APPLICATIONNAME

Delete a program downloaded into an application / stored in the Interpreter.

All Interpreter commands between "prog" and "endp" are stored rather than executed (with the exception of functions installed with specific directions to do otherwise; see GITE library symbol installation routine).

A program stored in the Interpreter can access any symbol installed in the environment, and call any other program. No parameters can be passed and no return values can be received. Two additional limitations are imposed on downloaded programs: they can only refer to symbols of the application into which they are downloaded, and they may not use labels, jumps and returns.

A program may be invoked by any application with a single "call" command. For Interpreter-resident programs, the command processing is considered completed when the program execution is finished; for downloaded programs -- when the program execution is initiated. In this latter case, the requester application is not directly informed when, or indeed if, the program execution is completed.

VII. BUILT-IN SYMBOLS.

The following symbols are always defined in the Interpreter. All of them are declared as functions (rather than variables); the corresponding actions are carried out by the Interpreter itself. In the descriptions below parameter keywords are enclosed in brackets to show that they are optional as long as parameters are listed in the order specified.

aplstart [apl:]APPLICATIONNAME [pif:]PIFFILENAME

Start an application. PIFFILENAME is a special data file describing the application for DESQview. (Such a file is necessary to start any application under DESQview, GITE or no GITE; DESQview provides means of creating and editing these files.) APPLICATIONNAME is the name under which the application is going to be known to the Interpreter. This overrides the name the application tries to give itself when it executes dv_start (see the section on GITE library). Unlike the name in dv_start, this APPLICATIONNAME must be unique in the environment.

apldumb [apl:]APPLICATIONNAME [pif:]PIFFILENAME

Start a dumb application. Dumb applications are not expected to acknowledge themselves to the Interpreter (which non-dumb applications do by executing dv_start), and the Interpreter supports no communication to or from them, except via "aplkey" command (below). Parameters are the same as in aplstart.

aplkey [apl:]APPLICATIONNAME [key:]STRING [status:]INT

Send the STRING to the specified application as its keyboard input, character by character. INT (8 bits) is sent together with every key as status. (By DESQview convention, a key is represented by a 2-byte extended key code, the second byte being called the status byte. For standard keys, the status byte is zero; for Alt and special keys, the character code is zero, while the status byte describes the key.)

apldel [apl:]APPLICATIONNAME

Delete the specified application, remove all its symbols from the tables.

aplcheck

An application may be removed directly from DESQview, bypassing the Interpreter. This would create an inconsistency in the Interpreter tables. This command makes the Interpreter check the tables against the information available from DESQview.

aplpresent [apl:]APPLICATIONNAME

`Returns a long unsigned integer`: application ID if the application is active, 0 if not. The answer is based on Interpreter tables. Application IDs are used to identify applications for low-level API routines. If the Interpreter application list is inconsistent (see "aplcheck" above), the return value is meaningless.

aplsetmsg [apl:]APPLICATIONNAME

Set the specified application the recipient for global environment error messages. Whenever an application with the name "KEYBOARD" is activated by aplstart, it automatically becomes such a recipient.

aplsetcall [apl:]APPLICATIONNAME [call:]SYMBOLNAME [param:]PARAMETERSTRING

This invokes a function installed as SYMBOLNAME with parameters contained in PARAMETERSTRING (the string can contain any number of parameters as long as it is quoted). The symbol SYMBOLNAME must be installed as a function from the application APPLICATIONNAME. Unlike an ordinary "SYMBOLNAME PARAMETERSTRING" request, this command orders the application to invoke the function not once but repeatedly with the same actual parameters. (The application checks its mail between invocations, though: this allows it to break on error conditions, abort and "aplclrcall".) The Interpreter does not monitor the progress of the looping application; the request is considered completed as soon as the loop is initiated. No return

values are gotten by the requester while the looping is in progress, nor when it is terminated by "aplclrcall".

aplclrcall [apl:]APPLICATIONNAME

Terminate the looping initiated by "aplsetcall" inside the specified application, if any had been in progress there.

include [file:]FILENAME

Open the specified file and take next commands from there, till an "endf" command or the end of the file. The file may contain other "include" commands, up to 3 levels.

endf

In an batch file opened by "include", directs the Interpreter to finish the processing of the batch file it is found in.

tstart

Start the timer. The Interpreter has one timer inside it.

tstop

Stop the timer and read elapsed time. Returns a string like "Elapsed time 8.000000 Sec.".

cset COUNTERNUMBER INITIALVALUE

Set the specified counter to the specified long integer value. The Interpreter has 10 different counters, numbered 0 through 9.

cdec COUNTERNUMBER VALUE

cinc COUNTERNUMBER VALUE

Increment / decrement the specified counter by the specified long integer. No return value for the requester, but in an Interpreter program (see Programming Features section), 'jz' / 'jnz' conditionals, if used immediately after 'cdec' / 'cinc', will be based on the value of the counter.

symchk FLAG

Defines the Interpreter's reaction to a request for access to an undefined symbol. FLAG=0 -- declare an error condition, FLAG=1 -- ignore the request. 0 is the default.

parchk FLAG

Defines the Interpreter's reaction to a missing parameter on a request line. FLAG=0 -- use the default value (defined at installation time), FLAG=1 -- declare an error condition. 0 is the default.

prog_autodel FLAG

Defines the Interpreter's reaction to an attempt to redefine an existing program stored in the Interpreter. FLAG=0 -- declare an error condition, FLAG=1 -- supersede the old program. 0 is the default.

The following commands return several lines of text to the requester. Only a special application like KEYBOARD can handle such a return. These commands are not designed to be used from conventional applications.

apllist

Displays the complete list of applications, their IDs, PIF-files and hidden/unhidden status. The information is based on Interpreter tables.

help

Displays all installed symbols, together with names of applications which installed them.

help SYMBOLNAME

Displays additional information on the specified symbol, including object type, validation, and for functions -- types, validations and keywords for all parameters. A limited use of wildcards is supported SYMBOLNAME: '*' matches all names, 'AB*' - all names beginning with AB. Symbols installed by a particular application may be selected using 'APPLICATIONNAME.SYMBOLNAME' or 'APPLICATIONNAME.*'.

The following commands alter the application's status from DESQview's point of view, but not from the Interpreter's.

aplgofore [apl:]APPLICATIONNAME

Set an application foreground. In DESQview, the foreground application receives direct keyboard input; also, if the application had been suspended, it resumes execution if moved to foreground. If the application's window had been hidden, the window does not appear on the screen until it is unhidden (see aplhide/aplunhide below).

aplsuspend [apl:]APPLICATIONNAME

Move the application into background, hide its windows, and stop the application's execution. The execution resumes when the application is moved to foreground.

aplhide [apl:]APPLICATIONNAME

Hide the application's window. The window disappears from the screen. The application's foreground/background status is not affected.

aplunhide [apl:]APPLICATIONNAME

Unhide the application window and redraw it on the screen. The application's foreground/background status is not affected.

aplredraw [apl:]APPLICATIONNAME

Redraw the window of the specified application.

VIII. GITE LIBRARY.

All routines are declared, and all symbols mentioned are defined, in a header file "GITE.H".

int dv_start (name, rev, cnt, argc, argv)

Applications use this routine to acknowledge themselves to the Interpreter. Returns 0 if the new application can not be accepted.

char* name — The name under which the application wants to be known. The names of active applications are available upon request to other active applications. Up to 12 characters. If the application is started through the Interpreter, this name is overridden by "aplstart" command parameters.

char* rev — Revision ID. Both (*name) and (*rev) are displayed in the application window during the execution of dv_start. The revision ID has no significance other than this.

int cnt — The desired application name in (*name) need not be unique. If an application with this name is already active, dv_start tries to use names like NAME_1, NAME_2, etc. If cnt number of such applications already exist, the dv_start request is rejected. Most applications use cnt=1.

int argc
char* argv — These describe a parameter string in the C language convention. Two command flags are supported:

-b Draw no border for the application window,

-s After starting, immediately hide the application window. Same as the 'aplhide' Interpreter command.

int dv_aplpresent (aplname)

Check the presence of an application in the Interpreter tables. Sends an "aplpresent" request to the Interpreter, fetches and returns its reply, which is 1 if the application is present, 0 otherwise.

char* aplname — Application name.

char* parse_add_sym (ptr, name, addr, type, val_type, vld)

Install a symbol in the Interpreter. If a function with parameters is being installed, the call to parse_add_sym must be followed by a call to parse_add_par (see below) for every parameter; the return value of parse_add_sym is then important for parse_add_par.

char* ptr — Not used. Should be NULL.

char* name

Symbol name, up to 12 characters. Used by other applications to refer to the symbol in service requests. Case-sensitive. Must be unique within the application; uniqueness throughout the environment is recommended but not necessary. If several applications install symbols with the same name, all references will be resolved to the one installed first; but a reference in the form APPLICATIONNAME.SYMBOLNAME will be resolved to the symbol belonging to the application mentioned.

void* addr

Address of the object (function or variable) which is being made accessible to other applications through this symbol. Note that in C language, a variable must be preceded by an '&' sign to use its address rather than value.

int type

One of the following (defined in 6SYSDEFS.H) :

S_GVAR if the object is a variable to be made read/write accessible,

S_GVAR|CONST_DATA if the object is a variable to be made read-only accessible,

S_GFNC if the object is a function, S_IFNC if the object is a function which cannot be included into a program; if the command is entered during a program definition, it (the command) is executed immediately rather than stored with the program.

int val_type

If the object is a variable, this defines its type; if the object is a function, this defines the return value type. Must be one of the following (defined in 6SYSDEFS.H):

S_LONG long integer (32 bits),

S_SHORT short integer (16 bits),

S_FLT double precision float (32 bits),

S_STR string,

VOID if the object is a function, tells the Interpreter the function is void.

char* vld

Validation string. If the object is a variable, validates the variable; if the object is a function, validates the return value. See the section on the validation system.

void parse_add_par (head, type, keywrd, vld, dflt, val,ptr)

Used to conclude the installation procedure of a function with parameters. Each call describes one parameter of the parent function. Calls must follow the parse_add_sym(...) which began the installation, and describe parameters in the same order as in the C declaration of the parent function.

char* head	The value returned by parse_add_sym when the function installation began.
int type	Parameter type. S_LONG, S_SHORT, S_FLT, S_STR, like in parse_add_sym.
char* keywrd	A keyword associated with the parameter. Up to 12 characters or NULL if keyword access to this parameter is not desired.
char* vld	Validation string for the parameter. (See the section on the validation system.)
char* dflt	Parameter default value (in ASCII form, even for numerical parameters).
int val	Not used.
char* ptr	Not used.

void dv_proc ()

This routine moves the application moves into wait state and waits for environmental events (these include keystrokes, mail messages, and timeouts; see the section on GITE-aware applications).

void server_req (buf)

Issue an Interpreter command.

char* buf	Command text.

void server_wait_ready ()

Wait till the request is finished. It is necessary to wait for a request before issuing the next one. Waiting where no request had been issued results in a timeout condition.

void server_printf (fmt, p1, p2, p3, p4, p5, p6)

Issue an Interpreter command and wait for completion. The command text is formed using (*fmt) as the format string, and p1-p6 as parameters, like in printf. The text may not be longer than 200 characters.

char* fmt; p1-p6 according to the format in *fmt	The parameters are similar to those of an ordinary C printf. Like in printf, trailing parameters may be omitted if not used, but (unlike printf) no more than 6 parameters are allowed.

void server_printf_nowait (fmt, p1, p2, p3, p4, p5, p6)

Same as server_printf, but do not wait for the completion of the request.

int server_get_int (fmt, p1, p2, p3, p4)

long server_get_long (fmt, p1, p2, p3, p4)

double server_get_flt (fmt, p1, p2, p3, p4)

char* server_get_str (buf, maxlen, fmt, p1, p2, p3, p4)

Issue an Interpreter command, wait for completion and fetch a return value of a certain type. A return value is always passed from the Interpreter as an ASCII string. Server_get_str copies it into the specified user buffer, and an error is declared if the buffer size is not sufficient. Other routines attempt to convert the string into an integer, a long integer, and a double precision float, respectively, and pass the result as their return value; this result is undefined if the string cannot be so converted.

char* fmt; p1-p4 according to the format in *fmt	The format and parameters of the command text, like in server_printf, and the same text length limitation applies; but if fmt==NULL, no command is issued (there is no such check in server_printf).
char* buf	User buffer to put the received string in.
int maxlen	Buffer size.

unsigned long dv_set_wait_timeout (t)

Set the request completion timeout value. If the application is in wait state pending the completion of a request longer than the specified period of time, GITE routines inside it declare an environmental error condition, thereby aborting the request (as well as all other requests in the environment currently pending; see the Error Recovery section). The previous timeout value is returned by the routine.

unsigned long t	The timeout interval in 0.01sec units. The value is 1000 when the application is started.

void kbd_flash ()

Skip any currently waiting keystrokes that may be queued for the application.

void dv_dsplerr (buf)

Displays a message DESQview-style, at the left-top of the application window, in a separate box, waits for an Escape key before proceeding, then erases the message.

char* buf	Message text pointer.

void err_msg_raise (code, fmt, p1, p2, p3, p4)

Process an error. A call to this routine aborts the current request and makes a C "long jump" to a pre-defined point in the beginning of the application program ("longjmp(dv_jbuf,-1)"; see the section on GITE-aware applications for more details). If the application is executing an internal loop initiated by an "aplsetcall" command, (where the Interpreter does not monitor the progress of the request), err_msg_raise takes no other action. In all other cases, it declares an environmental error condition, and the Interpreter sends a special message to all applications known to it, making all these

applications perform similar longjumps. See Error Recovery and GITE-aware Applications sections for more information on global and local error recovery procedures, respectively.

int code	Error code. Codes below 1000 are reserved for GITE use; otherwise an application may use any value and it is not significant.
char* fmt; p1-p4 according to the format in *fmt	The format and parameters used to compile the error message text, which, if the environmental condition is being declared, then goes, through the Interpreter, to the global error message recipient application (see Error Recovery section); if the environmental error condition is not declared, err_msg_raise calls dv_dsplerr to display the message locally.

void dv_close()

Log out of the Interpreter and terminate the application.

The GITE library also includes higher level routines for interfacing with utility applications provided for the environment. See the section on utility applications.

IX. DUMB APPLICATIONS.

A dumb application is one that chooses not to declare itself to the Interpreter. Such applications will not receive any requests from the Interpreter, and are not expected to send any; they do not need any GITE library routines to handle such requests. Any task can be run as a dumb application, even if it had been developed without any knowledge of GITE.

If a dumb application is not invoked through the Interpreter, its presence will not be detected. If a dumb application is invoked through the Interpreter ("apldumb" command), its whereabouts will be known, and the Interpreter will be able to emulate keystrokes for it ("aplkey" command). This one-way "send keystroke" operation is the only type of communication with a non-cooperative application supported by GITE (and DESQview).

X. GITE-AWARE APPLICATIONS.

A GITE-aware application is one which supports the GITE communication protocol, implemented in the GITE library. Such applications are required to declare themselves to the Interpreter, and it does not matter whether or not they are invoked through the Interpreter.

In GITE (and in DESQview), an application is not open to any incoming communications until it switches itself into wait state, either because it has finished processing an incoming request, or because it is waiting completion of a request of its own. As long as the application is actively executing, even an abort command from the Interpreter or an unsolicited keystroke will only be queued for the application but not get through to it. (A keystroke is called unsolicited if it arrives when the application has not issued a keyboard input request.)

When an application is in wait state, DESQview does not give it any CPU time (and thus other application are not slowed down). DESQview moves the application out of the wait state when an environment event for this application occurs. Such events include the following:

- A mail package is received for the application,
- An unsolicited keystroke is received for the application, either directly from the keyboard (if the application is at the moment foreground), or emulated by another application,
- The time interval previously set in the timer has expired.

The DESQview mail and timer mechanisms are crucial for the functioning of the GITE, and to avoid confusion in communications the user is not allowed to use either. Any mail or timer event over and above those intended for, and processed by, the GITE library routines inside the application, either is ignored, or creates an error condition. However, an application must include an "int apl_kbd_event (int k)" routine to process unsolicited keystrokes.

By DESQview convention, a keystroke is represented by a 2-byte extended code (the second byte is called key status), where for standard keys the first byte is the ASCII character code and the status is zero, while for Alt and special keys the first byte is zero and the status byte describes the key. When an unsolicited keystroke event occurs for the application (either because a key was pressed while the application was foreground, or because a keystroke was emulated by another application), GITE routines dequeue the extended code and invoke "apl_kbd_event". The integer parameter passed to "apl_kbd_event" describes the key, but not in the DESQview convention: for standard keys, it equals ASCII character code, and for special keys, (256+<key status byte>). "Apl_kbd_event" may process the event in any way; but its return value must be a keystroke code like its parameter, and directs future processing of the event to be done by GITE library routines.

For different return values (the key code symbols are defined in GITE.H), this processing is:

- CTRL_Q -- Log off the Interpreter and terminate the application;
- CTRL_E -- Erase the application window;
- CTRL_N -- Display a memory allocation map for the application;
- CTRL_Z -- Call 'err_msg_raise (5001, "Keyboard abort")' (which will create an environmental error condition, see the description of "err_msg_raise");
- CTRL_T -- Call an dummy function called "test()" (this is useful when using a debugger and having a breakpoint at this function);
- F1_KEY -- Invoke a routine displaying a help text for the above mentioned keys, and then the user "cmnd_help()" routine (described below);
- 0x00 -- No action;
- All others -- in this software version, no action.

Usually, if the keystroke is application-specific, "apl_kbd_event" returns zero; if not, it just returns the value of its parameter, thus allowing standard keystroke processing.

There are two other routines which must be included in the user code for a GITE-aware application. GITE library routines call them in situations described below.

void apl_close ()

An application-specific exit function. Called when the application is being shut down by GITE library routines (which happens when they process a "dv_close()" call, a ctrl/Q keystroke or an abort message from the Interpreter). An application may be terminated or terminate itself directly, however; in this case apl_close is not called (and Interpreter tables may contain inconsistencies after that).

void cmnd_help ()

Application-specific help. Called when GITE library routines process an F1 keystroke, after general help text is displayed by these routines themselves.

The sequence of actions in a GITE-aware application is usually following (see an example of such an application in Appendix B). When it is run, it acknowledges itself to the Interpreter ("dv_start" routine), installs its symbols ("parse_add_sym" and "parse_add_par" routines), performs self-initializations (if any), and moves itself into wait state ("dv_proc" routine). On respective Interpreter mail messages, GITE library routines inside the application report and change installed variables or invoke installed functions. When an installed function takes control, it may send Interpreter commands to be executed by other applications, and get return values from them, as necessary ("server_printf", "server_get_int" and other routines). If the operation cannot continue, the function declares an environmental error condition ("err_msg_raise" routine). In the end the function produces a return value and returns to the GITE library routine from which it had been called; this routine passes the return value to the Interpreter and returns the application into wait state.

In addition, return points must be set for long jumps executed as a part of the error recovery procedures. Return points must be set using a standard C library routine "setjmp", and the context must be stored in the buffer called "dv_jbuf" (this is a global name, the space for the buffer is allocated in GITE library routines). (For more information on the long jump mechanism, see an appropriate C manual.) In case an error is detected by the environment, a return point must be defined before the first call to "dv_start"; but since an application must not reinstall its symbols after a run-time error, the return point must be redefined in the self-initalization section of the application (between symbol installations and a call to "dv_proc"). "setjmp" must be called before initializations which have to be done after each environmental error, and after those which, like symbol installations, are only done at start-up time. Usual C restrictions on setjmp function calls also apply: the function from which setjmp is issued may not be exited before "dv_proc" is called. The placement of "setjmp"s is very important for correct error recovery; see an example of a correct placement in Appendix B.

Any external events are processed as described whenever the application moves itself into wait state, whether having finished an incoming request or pending completion of a request of its own. If the event is an environmental error condition, the application, from GITE routines inside it, executes a long jump "longjmp(dv_jbuf)" back into its self-initialization section, works its way to the call to "dv_proc", and returns to the wait state.

XI. UTILITY APPLICATION ENVIRONMENT.

Several utility applications are provided in the environment for basic user interface functionality. Their presence in the environment is not essential if a (sufficient level of) user interface is provided by other applications. The utility

environment, however, provides a tool for programming user interfaces, and helps present a set of GITE applications to the user as a single unified system. Standard user interface utility applications include KEYBOARD (a direct user interface with the Interpreter), MENU (a 1-2-3-style menu system for invoking Interpreter commands), and VOUT (an output handler).

1. The Keyboard Application.

The KEYBOARD application (when foreground) accepts direct ASCII user input from the keyboard and relays it to the Interpreter; any replies from the Interpreter are displayed by the KEYBOARD application in its window. The application supports most conventional mechanisms of editing command lines, including Backspace to erase the last typed in character, Escape to erase current line, and arrows up and down to move between previously entered lines. When loaded, the KEYBOARD application assumes the status of global error message handler, so that in case of an environmental error condition it is informed which application had declared the condition and what error message had been put forward; this information is also promptly displayed in the KEYBOARD application window. The KEYBOARD application may be used for interactive execution and debugging of applications.

2. The Menu Application.

The MENU application supports one or several menu trees. It provides commands to configure the trees, and a command to "pass control" to a menu tree which has already been configured. At this point the MENU application goes to foreground and takes control of the user keyboard input. The user then can move between nodes of the tree, and, at the leaf level of the tree, execute Interpreter commands. At each node, MENU displays prompts of all choices open at the moment. As the cursor is moved, 1-2-3-style, from prompt to prompt with arrow keys, another screen line shows a help text associated with the item it is currently at. The selection of an item is made with the Enter key, at which point one of the following actions is taken by the MENU application, depending on the type of the selected item:

- If the item type is EXIT, the menu is erased from the screen, the MENU application goes into background, the command which "passed control" to the tree is considered completed, and MENU is ready to accept further commands.
- If the item type is UP, the user moves to the node immediately above the current one; if the current node is the root of the tree, no action is taken. (The same happens if the Escape key is pressed at any item.)
- If the item type is MENU, the user moves to a lower-level node. (MENU type items of nodes are associated with lower-level nodes at tree-configuration time.)
- If the item type is EXEC, MENU issues an Interpreter command and waits for its completion (after which the 1-2-3-style selection resumes). To prevent recursion in the environment, no requests to the MENU application should be made in the meanwhile. The Interpreter command being issued is defined at tree-configuration time, and cannot be changed or modified by the user at run-time (which is probably the main limitation of this interface system).

The following two commands are implemented in the MENU application to configure menu trees. New trees, nodes, and leaves may be added to the configuration at any time, but once created, they cannot be removed or modified. The notation below is the same as in the "Built-in symbols" section.

menunew [name:]NODENAME

Add a new node with the specified name.

menuitem [name:]NODENAME [code:]{EXIT|UP|MENU|EXEC} [str:]PROMPT [dscr:]HELP [prog:]SMB

Add an item under the (existing) parent node named NODENAME. PROMPT and HELP are the item's prompt and help strings. During selection, the cursor is moved between prompts of items underlying the current node, all of which are displayed until the selection is made; while the help string appears only for the item the cursor is at at any given time. Code selects the type of the item; if an item is selected, the action taken depends on this type, as described above. The interpretation of the string SMB depends on the type of the item. If this type is EXEC, SMB is the Interpreter command to issue; if the type is MENU, SMB is the name of the lower-level node to go to; if the type is UP or EXIT, SMB is ignored.

The following command is used to draw, and "pass control" to, a menu tree. The tree must already exist by the time this command is issued.

menu [name:]NODENAME

Pass control to a tree underlying the specified node. The node must not necessarily be the root node of the tree, but the user will not be able to move above it even if it isn't. The request is completed when, and only when, the user selects an EXIT-type item.

The following command changes the position of the menu inside the MENU application window. It is not usually used, but when it is, it is essential that the screen area allocated is large enough and not outside the application window.

menuconfig [row:]N1 [col:]M1 [rows:]N2 [cols:]M2

Specifies the position of the upper left corner (N1, M1), and the size (N2, M2) of the menu area relative to the MENU application window.

The usual sequence of operations with the MENU application is this. MENU is invoked ("aplstart" command) in the Interpreter startup batch file before applications which are going to use it. A single tree structure is used in most cases, and it is also configured from the startup file, with commands described above. Then, also from the startup file, other applications are invoked ("aplstart" again), and initialize themselves. At this time applications expand the tree structure ("menunew", "menuitem"), making their own commands available to users of the menu system. Applications using MENU are aware of pre-configured nodes in the tree, and usually only add EXEC-type items. When all the necessary applications are invoked and have initialized, the interface system is kicked off by a "menu" command, which usually concludes the Interpreter startup batch file.

The following is an example of configuring a simple menu tree.

menunew "root"

Root node.

menuitem "root" EXEC "Test_1" "Execute test number one" T1

First item: execute test number one. T1 is the command the test is invoked with.

menuitem "root" EXEC "Test_2" "Execute test number two" T2

Second item: execute test number two. T2 is the command the test is invoked with.

menuitem "root" MENU "Exit" "Exit the system" "quit"

Another item: exit the system.

menunew "quit"

The underlying node to quit.

menuitem "quit" EXIT "Yes" "Quit the system?"

Quit confirm.

menuitem "quit" UP "No" "Quit the system?"

Quit cancel, go back up.

The GITE library has a feature to help make MENU-using applications more environment-independent. The dv_start routine (always called in the beginning of an application) checks the presence of the MENU application in the environment; this information is then used by the following routine to prevent an error condition if MENU is not present (the application thus does not crash and can be used without modification through KEYBOARD, for example).

void menu_add_exec (p1, p2, p3, p4)

If the MENU application had been available at the time dv_start() was called, this routine issues an Interpreter command "menuitem <p1> EXEC <p2> <p3> <p4>" (with parameter strings substituted for p1, p2, p3, p4). If the MENU application had not been available, no action is taken.

char* p1, p2, p3, p4 Parameter strings for "menuitem".

3. The Video Output Application.

The VOUT output handler application is another element of the utility environment. It must be invoked before applications which are going to use it (usually from the Interpreter startup batch file), because VOUT's presence is checked as a part of the "dv_start" procedure. VOUT does not have any conventional Interpreter commands, and is only accessible via the following GITE library routine.

void vid_printf (fmt, p1, p2, p3, p4, p5, p6, p7, p8)

If VOUT had been available at the time of dv_start() execution, an output line is sent to VOUT; otherwise the output line is displayed locally in the caller application window (as with a simple printf()).

char* fmt; p1-p8 according to the format in *fmt	The parameters are similar to those of an ordinary C printf function. The output line is formed using (*fmt) as the format string and p1-p8 as parameters. As in printf(), trailing parameters may be omitted if not used, but (unlike printf) no more than 8 parameters are allowed.

An application like VOUT may provide ways of redirecting and/or post-processing output data. In the present release, however, all output lines received by VOUT are simply displayed in its window on the First-Received-First-Displayed basis.

APPENDIX A. KNOWN BUGS AND PROBLEMS.

1. Symbol table memory is not correctly released when applications are removed by "apldel" command, the Interpreter may run out of memory and crash after a while.

APPENDIX B. AN EXAMPLE OF A GITE-AWARE APPLICATION.

```
/*****************************************
        General purpose test aplication
*****************************************/

#include <stdio.h>
#include <setjmp.h>
#include <gite.h>

        /*
        Standard defines for symbol installation.
        Function declaration
        Key codes
        */

char rev[] = "0.00";
char apiname[] = "tr";


/*-------------------------------------------*/
/*-------------- REQUIRED FUNCTIONS ---------------*/
/*-------------------------------------------*/


main (int argc, char **argv)
        {
        extern jmp_buf dv_jbuf;                         /* REQUIRED */

        if (setjmp (dv_jbuf))                           /* REQUIRED */
                {
    /* The control is transferred here if an error is detected
     * during initialization.
                * Example: Display error.
                */
                dv_dsplerr ("Error during initialisation ");
                exit (1);
                }

        /* Install application into the GITE environmnet */
        dv_start (apiname, rev, 1, argc, argv);                 /* REQUIRED */

        /* Perform application specific Intialisation. Typicaly:
         *   Install it's symbols
         *   Check presence of required applications
         *  Example:
         */
        apl_symb ();
        apl_init ();

        if (setjmp (dv_jbuf))                   /* REQUIRED */
                {
    /* Control is transferred here if an error detected
```

```
 * by this application, or an environmental error condition
 * declared by any application.
                */
               }

/*  Perform functions to be done after start and after error recovery
        */

       /*  Wait for requests and Process them */
       dv_proc ();              /* REQUIRED */
}


/*-------------------------------------------------*/
/* Process keyboard event.
   RETURNS:
                                KEY CODE - can be different from the code passed to the function
                  0 -  further processing not required.
*/

int  apl_kbd_event (k)
 int k;          /* Key code */
        {
        switch (k)
                {
                        /* Example: Execution key */
                case    CTRL_I:
                        sample_fnc (1, 2L, 3.4, "S5");
                        break;

                        /* Example: Conversion of key */
                case    CTRL_L:
                        return (CTRL_N);                                /* RECOMENDED */

                default:
    /* Return keys which are not processed */
                        return (k);                                     /* RECOMENDED */
                }

        /* Tell that this keys were processed */
        return (0);             /* RECOMENDED */
 }

/*-------------------------------------------------*/
/* Application specific exit function */
void apl_close ()
        {
        }


/*-------------------------------------------------*/
/* Display application specific help */
void cmnd_help ()
{
```

```
        /* Example: Implementation */
        dv_printf ("\n");
        dv_printf (      "^I  -  Call sample function");
        dv_printf (      "^L  -  Convert it to ^N");
        }


/*-------------------------------------------*/
/*-------------- APPLICATION EXAMPLES --------------*/
/*-------------------------------------------*/

/* Example: Intialisation function */
apl_init ()
        {
        int i;
        unsigned long timeout;

/* Example of verification of application presence */
        /* Example: send command send to Interpreter and get integer value */
 i = server_get_int ("aplpresent ABC");

        /* Example: Error generation */
        if (!i)
                err_msg_raise (1001, "Aplication ABC is not present *");

        /* Example: Disable wait for resonse timeout */
        timeout = dv_set_wait_timeout ((unsigned long) 0);

        /* Example: command send to Interpreter, wait for end of execution  */
        server_printf ("ABC.start");

        /* Example:  Restore timeout */
        dv_set_wait_timeout (timeout);
        }

/*-------------------------------------------*/

/* Exmaple: Integer to be installed in Interpreter */
int     sample_int = 1;

/* Exmaple: Function to be installed in Interpreter */
sample_fnc (p1, p2, p3, p4)
int         p1;
long        p2;
double p3;
char            p4;
        {

        printf ("Sample Function call: p1 = %d, p2 = %ld, p3 = %lf, p4 = %s\n",
                p1, p2, p3, p4);

        /* Example: command send to Interpreter, wait for end of execution  */
        server_printf ("aplcheck");
```

```
        }

/*-------------------------------------------*/
/* Example: Symbol installation */
api_symb ()
        {
        char *s;

        s = parse_add_sym (NULL,     "sfnc", sample_fnc ,S_GFNC, VOID,NULL);
                parse_add_par  (s,S_SHORT,"par1","0 @ OFF ON","1",0,NULL);
                parse_add_par  (s,S_LONG, "par2",">10&<1000", "100",0,NULL);
                parse_add_par  (s,S_FLT,  "par3",NULL,    "1.2",0,NULL);
                parse_add_par  (s,S_STR  ,"par4",NULL,    "sample",0,NULL);
        s = parse_add_sym (NULL,     "sint", &sample_int ,S_GVAR, S_SHORT,NULL);
        }
```