Users Guide for

GYST

(Growth and Yield Software Technology)

Ralph L. Amateis
Philip J. Radtke
Harold E. Burkhart[1]

May, 2001

[1]The development of GYST was supported by the Loblolly Pine Growth and Yield Research Cooperative at Virginia Tech. Support from Bowater Inc., Boise Cascade Corp., Champion International Corp., Chesapeake Forest Products Co., Georgia-Pacific Corp., International Paper Co., James M. Vardaman and Co., Inc., Temple-Inland Inc., Union Camp Corp., Westvaco Corp., Weyerhaeuser Co., Willamette Industries and the Virginia Department of Forestry is gratefully acknowledged.

TABLE OF CONTENTS

ABOUT GYST

GYST is a generalized user interface for implementing growth and yield models developed for the Windows (© Microsoft Corporation) operating system. The GYST interface environment stores and displays input data in spreadsheet window views. Growth and yield models that have been coded into Dynamic Linked Libraries (DLLs) access the data and return results which are stored and displayed in the GYST window views. The GYST environment allows the spreadsheet window views to be customized according to DLL developer preferences. GYST offers several advantages over other types of user interfaces.

1. Model developers need only write software (DLLs) that can implement growth and yield equations without the need to develop a customized user interface for each application.

2. GYST supports DLLs written in C or Fortran through simple string parsing routines.

3. The GYST environment can open and execute multiple DLLs (growth and yield models) simultaneously.

3. The spreadsheet display windows within GYST operate similarly to existing spreadsheet packages familiar to many computer users.

4. Importing and exporting data through the Windows clipboard makes GYST compatible with other Windows applications.

5. GYST supports stand, distribution and tree-based models and conversions from one model leve to another

GYST is compatible with Windows 95, 98, NT and 2000 operating systems.

GYST SESSIONS

Each GYST session consists of one or more spreadsheet window views: Stand, Distribution and Tree depending on the nature of the DLL. Each window view can be minimized or maximized and two or more views can be tiled or cascaded. GYST supports multiple active sessions that can access the same or different DLLs.

GYST has been structured as a spreadsheet with rows as observations and columns containing variables. For the Stand window view, each row would generally represent a forest stand and columns would hold variables pertaining to each stand. In a similar way, the Distribution window view would have rows pertaining to diameter or perhaps height class information and the columns would hold variables related to those diameter classes. The Tree window view would contain rows representing trees and the columns would have variables

associated with those trees.

Opening, Closing and Saving

Invoking GYST automatically creates a new GYST session. Alternatively, a new GYST session can be created from the *File* menu option. The *File* menu option also is used to open existing GYST sessions and to close and save GYST sessions. The default extension for a saved GYST session is .gst.

Printing

Data and results from any of the spreadsheet views can be printed through the *File* menu option as with other Windows applications.

Editing

GYST supports the customary cut, copy, paste and clear editing functions common to Windows applications. The insert command inserts a line prior to the line containing the outlined cell. Edit commands can be accessed from the *Edit* menu option or from the toolbar.

Importing and Exporting Data

Data for implementation by growth and yield models can be brought into a GYST session through one of two ways. First, data can be entered directly via the keyboard into the proper cells of the appropriate window view just as can be done with any spreadsheet. Second, data can be cut or copied from a spreadsheet onto the Windows clipboard and then pasted into the appropriate GYST spreadsheet view.

Data or output results residing in a GYST spreadsheet view can be exported to other Windows applications such as spreadsheets, graphics packages or word processors via the Windows clipboard.

ACCESSING AND USING DYNAMIC LINKED LIBRARIES

Growth and yield models coded as Dynamically Linked Libraries (DLLs) for use within GYST must be named in the *models.toc* file within the same directory (folder) as the GYST executable (gyst.exe). Each DLL named in the *models.toc* file must appear on a separate line without the .dll extension. *Models.toc* is a standard text file that can be viewed or edited using a standard text editor. Libraries can be accessed for use during a GYST session through the *Model*

menu option. The *Select Model Library...* choice spawns the *Select Model Library* window that displays the contents of the *models.toc* file. These DLLs would be available for use with GYST. Selecting a model loads the DLL for use within the active GYST session.

The user or the DLL developer can create a text file (**.txt** extension) that provides a description of each particular DLL in the DLL library and will be displayed in the Description window. For example, if there is a *demo.dll* file in the model library then there can be a *demo.txt* file that provides a description of the *demo.dll* model to users.

Several models can be run simultaneously by opening multiple GYST sessions. This is done by clicking on (activating) one document and selecting its model (DLL) from the Model menu. Then clicking on (activating) another document and selecting its model (DLL). Model information is saved when the user saves the document.

Specifying a Default DLL

It is possible to specify a default DLL which will automatically be loaded each time a GYST session is activated. This saves time when one model is used frequently. To define a default DLL, open the *moddflt.txt* file with any text editor and place the name of the desired DLL (excluding the .dll extension) on the first line. For example, if *demo.dll* is the desired default DLL, then *demo* should be typed on the first line of the *moddflt.txt* file.

Active Windows and DLL Functionality

A DLL developed for GYST has at maximum 12 entry points. These entry points correspond to three windows (stand, distribution and tree) and four actions (initializing, growing, and two updating actions). If C or Fortran code has been written for a particular entry point, then the action button corresponding to that window-entry point combination will be accessible to the user. If an entry point has not been coded, then the entry point will appear grayed out and will not be accessible to the user. For a full discussion about developing code for DLL entry points see the section DEVELOPING GROWTH AND YIELD MODELS AS DYNAMIC-LINK LIBRARIES FOR GYST.

INITIALIZING, GROWING AND UPDATING

There are three actions that users can accomplish during a GYST session: initialize, grow and update (conversion). Which action is accomplished will depend on which spreadsheet view is active and which of the action buttons on the tool bar is clicked.

Initialization

An initialize action occurs when the user clicks the  button. Exactly which initialization occurs depends upon which window is currently active in a GYST session. For instance, when the stand-level window is active and the user clicks the  button, GYST calls

the DLL's stand_ini function, which performs model calculations needed to initialize a stand. *Initialization makes only one call to the DLL per selected row of the active window* and is primarily used for obtaining predicted values from specified input data. Output data is displayed in the same window where the input data is displayed.

Growing

A grow function (stand_grow, dist_grow, or tree_grow in the DLL) is called depending on which window is active when the  button is clicked. *Growing makes one or multiple calls to the DLL per selected row of the active window* and is primarily used for making growth projections through time from specified input data. Output data is displayed in the same window where the input data is displayed.

Selecting the "Grow" option from the Model menu option (or clicking the  button on the tool bar) opens the "Grow Options" window and presents the user with a "Grow to..." or "Grow for..." option and an Output Interval dialog box. The Output Interval dialog box allows the user to determine how often output from the DLL calls will be displayed in the spreadsheet views. For example, if a ten-year growth projection will be made and output is desired every 2 years, then a "2" should be placed in the dialog box. The default output interval is 1. In order for the Output Interval dialog box to be active, the "Show intermediate growth output" option under the Options menu item must be checked.

Selecting the "Grow to..." option opens the "Grow to" window. The user then selects a variable from the "Variable" dialog box and one of the three "Growth options". If the "Maximize " option is selected, GYST will call the DLL until the maximum value of the selected variable has been achieved (up to a specified number of iterations). This growth option is useful for growing to maximum levels of critical stand parameters such as basal area or mean annual increment. The "Minimize" option works just as the "Maximize" option except that GYST calls the DLL until the minimum value of the selected variable has been achieved. This can be used for finding the minimum value of variables that decrease with time such as relative spacing.

Selecting the "Grow to Value" option allows the user to tell GYST when to terminate the calls to the DLL. GYST will call the DLL and compare the value of the selected variable with the specified stopping value. GYST will terminate calls to the DLL when the value of the selected variable meets or exceeds the stopping value. That is, all grow-to checking is done by the >= inequality convention. This growth option can be useful for projecting to certain stand attributes such as basal area, dominant height or volumes.

Information pertinent to developing DLLs for GYST that utilize the Grow option can be found in the section DEVELOPING GROWTH AND YIELD MODELS AS DYNAMIC-LINK LIBRARIES FOR GYST.

Updating (Conversion)

An updating (conversion) function is called when the user clicks either the S, D, or T button. Updating functions produce output that GYST displays in a different window than the one where the input data appeared. *Updating makes one call to the DLL* and is primarily used for changing the resolution of data in a particular spreadsheet view. Exactly which conversion function is called depends on which window is currently active and which button the user clicks.

The complete set of active-window and button-click combinations for the twelve entry points is listed in Table 2 under the section DEVELOPING GROWTH AND YIELD MODELS AS DYNAMIC-LINK LIBRARIES FOR GYST.

OPTIONS FOR CUSTOMIZING GYST

There are several options available under the Options main menu item that can be used to customize GYST according to particular user preferences.

Overwriting or Preserving Input Data

The user has the option of preserving input data or overwriting it with the results of initialization, grow or converion functions. The choice of preserving or overwriting input data is controlled through the Options menu item. By placing a check mark by the "Overwrite existing data" option, GYST will overwrite output results onto existing input data. The default option is to place output results following existing data (not overwriting).

Show Intermediate Growth Output

By placing a check mark next to the "Show intermediate growth output" option, the user will enable the Output Interval dialog box which can be used to determine how often output from the DLL calls will be displayed in the spreadsheet views. By removing the check mark, only final output from the last call to the DLL will be displayed in the spreadsheet. The default setting for this option is to enable the intermediate output.

Customizing Cells, Rows and Columns

The Options menu item contains choices for customizing the size and alignment of cells, rows and columns. Additionally, a particular style and size of font can be selected. Any of these selections can be established as the default. Activating the "Autofit Column Labels" option adjusts the size of the selected column to fit the label for that column. See the section DEVELOPING GROWTH AND YIELD MODELS AS DYNAMIC-LINK LIBRARIES FOR GYST for information about labeling columns.

HELP FILES

GYST supports a full set of Help files which can be accessed at any point during a GYST session by pressing the F1 key. The Help menu item also offers direct access to all help information.

DEVELOPING GROWTH AND YIELD MODELS AS DLLs FOR GYST

Introduction

GYST is a Windows application designed to implement growth and yield models developed and compiled as DLLs. The DLLs are then accessed and executed at run time by GYST. This capability means that GYST is not attached to any specific model but instead becomes a "displayer" of growth and yield model results from any DLL that has been properly developed for use by GYST. The following sections are designed to help guide computer programmers developing software for use with GYST.

GYST files

There are up to four files associated with each application developed for the GYST shell. In addition, there are two files that must reside in the same directory as the GYST shell. Table 1 presents a description of these files and their characteristics.

Table 1. Specifications for four files associated with each DLL application and two GYST system files.

Filename	Extension	Required	Description
User specified name	.dll	Yes	Name for the executable file developed in C or Fortran
User specified name	.dsc	Yes	Provides description and parameters associated with the variables passed into the DLL (see: CREATING A .DSC FILE FOR A DLL)
User specified name	.err	No	Provides descriptions to users when execution errors have been encountered
User specified name	.txt	No	Provides a description or overview of the DLL to users and is displayed in the Model Description window
Moddflt	.txt	No	Establishes a default DLL that will be loaded into GYST upon initialization
Models	.toc	Yes	Contains the list of DLLs that can be selected for use by a GYST session

Selecting a language

DLLs for GYST can be written in C or Fortran. The choice of which language to use depends on the programmer's preference and experience. It is possible to use mixed-language programming as well, where parts of a DLL would be written in C and parts in Fortran with the final product being compiled as a mixed-language DLL.

DLL Overview

A DLL consists of a set of functions that accept input data from the GYST program, manipulate the data, and pass the results back to GYST to be displayed to the user. The DLL connects to GYST by a set of twelve pre-defined function names. Although a customized DLL will perform its own set of calculations, it can only exchange data with GYST by using one or more of the pre-defined function names. Because these special functions give GYST access to the DLL, they are called "entry points". A list of the pre-defined entry point function names and brief descriptions of their primary uses are given in Table 2.

Table 2. Entry point function names for GYST dynamic-link libraries. The unique combination of button click and active window in GYST determines which function is called.

Name	Button	Window	Description
stand_ini	I	Stand	initialize stand level input; one line of data passed into and out of the entry point.
stand_grow	G	Stand	grow stand level input; one line of data passed into and out of the entry point.
stand_to_dist	D	Stand	convert stand level input to the diameter-distribution level; one line of data passed into the entry point, multiple lines passed out.
stand_to_tree	T	Stand	convert stand level input to the tree level; one line of data passed into the entry point, multiple lines passed out.
dist_ini	I	Diam.	initialize diameter-distribution level input; multiple lines of data from cursor position to the next blank line passed into entry point, multiple lines passed out. Alternatively, a highlighted block of data can be passed into the entry point and then passed out.
dist_grow	G	Diam.	grow diameter-distribution level input; multiple lines of data from cursor position to the next blank line passed into entry point, multiple lines passed out. Alternatively, a highlighted block of data can be passed into the entry point and then passed out.
dist_to_stand	S	Diam.	convert diameter-distribution input to the stand level; multiple lines of data from cursor position to the next blank line passed into entry point, one line passed out to the stand view. Alternatively, a highlighted block of data can be passed into the entry point and one line passed out to the stand view.
dist_to_tree	T	Diam.	convert diameter-distribution input to the tree level; multiple lines of data from cursor position to the next blank line passed into entry point, multiple lines passed out to the tree view. Alternatively, a highlighted block of data can be passed into the entry point and a block passed out to the tree view.
tree_ini	I	Tree	initialize tree level input; multiple lines of data from cursor position

			to the next blank line passed into entry point, multiple lines passed out to the tree view. Alternatively, a highlighted block of data can be passed into the entry point and a block passed to the tree view.
tree_grow	G	Tree	grow tree level input; multiple lines of data from cursor position to the next blank line passed into entry point, multiple lines passed out to the tree view. Alternatively, a highlighted block of data can be passed into the entry point and a block passed out to the tree view.
tree_to_stand	S	Tree	convert tree level input to stand level; multiple lines of data from cursor position to the next blank line passed into entry point, one line passed to the stand view. Or, a highlighted block of data can be passed into the entry point and one line passed to the stand view.
tree_to_dist	D	Tree	convert tree level input to diameter-distribution level; multiple lines of data from cursor position to the next blank line passed into entry point, multiple lines passed out to the stand view. Alternatively, a highlighted block of data can be passed into the entry point and multiple lines passed out to the distribution view.

Note that the entry point functions fall into one of three categories: initialization functions, grow functions, or conversion functions. An initialization function (stand_ini, dist_ini, or tree_ini) is called when the user clicks the I button. Exactly which of the three functions is called depends upon which window is currently active in GYST. For instance, when the stand-level window is active and the user clicks the I button, GYST calls the stand_ini function. A grow function (stand_grow, dist_grow, or tree_grow) is called depending on which window is active when the G button is clicked. One of the six conversion functions is called when the user clicks either the S, D, or T button. Exactly which conversion function is called depends on which window is currently active and which button the user clicks. It is important to note that the dist_ and tree_ functions pass multiple lines of data from the distribution and tree views of the spreadsheet. The lines that are passed include all lines from the current cursor location to the next blank line in the speadsheet. Alternatively, a block of contiguous lines can be highlighted and these will be passed to the entry point. The complete set of active-window and button-click combinations for the twelve entry points is listed in Table 2.

Initialization and grow functions produce output which GYST will display in the same window where the input data are displayed. The user has the option of preserving input data or overwriting it with the results of initialization or grow functions. Conversion functions produce output that GYST will display in a different window than the one where the input data appeared. Again, the user has the option to either preserve or overwrite data that existed in the output window at the time that the conversion function was called.

Because initialization and grow functions generate output to be displayed in the same window as the input data, their input and output data structures are identical. Conversion routines, on the other hand, have different input and output data structures. For example, the input structure of a stand_to_tree conversion function is a list of stand-level attributes but its output structure is a tree list. Exact specifications for these data structures depend on the

capabilities programmed into the model DLL. However, some conventions for data passing must be followed.

Data Passing Conventions

The first convention for data passing involves the function parameter list and its return value. All entry points have the same parameter types for DLLs written in C. The functions accept two pointers, both for character arrays (strings). The first string is a list of values sent from the cursor location in the currently active GYST spreadsheet. It is constructed by GYST using tab characters (\t) to separate values on a given line of data, and the return-newline sequence (\r\n) to indicate a break between lines of data. Depending on the active window type (stand, diameter distribution, or tree-level), the input string will consist of a single line of stand-level attributes, a series of lines of diameter distribution-level data, or a series of lines of tree-level data.

A string of stand level attributes is essentially made up of one line of data from the GYST stand-level window. Consider the user-entered data for the stand-level spreadsheet shown in Figure 1. No matter which entry-point button the user clicks (I, G, D, or T), the input string will consist of a single line made up of the values shown. These values are tab-delimited, so the string pointed to by the first function parameter will be "`Stand 1\t65\t12\t\t800.0\r\n`". Following the convention of the C programming language, all strings are terminated by the null character (\0). Note how the consecutive tab characters (\t\t) in the string indicate an empty cell in the input spreadsheet row. Instructions for extracting the input variable values from the input string are discussed in the section titled "Parsing a data string."

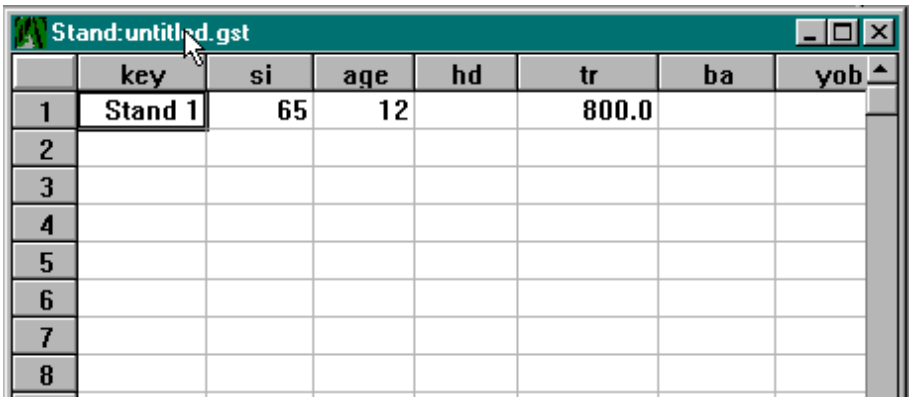


Figure 1. An example of stand-level input to a DLL entry point function as it appears in the GYST stand-level view.

The second string is available to the DLL but is not used or modified by GYST or the user. Any changes made to the second string will be preserved by GYST and passed to the next entry-point function called. This "hidden" string is included in the DLL architecture to allow programming flexibility. It is left to the DLL programmer to determine whether using the hidden string will be necessary to a particular DLL application. GYST will store up to 1024 characters in this second "hidden" string. If the DLL is being programmed in C then only the first character

string argument is necessary; the second argument is optional. If Fortran is the programming language, both string arguments must be declared, even if the second argument is not used.

Entry point functions return an integer that allows GYST to report DLL error conditions to the user. A return code of zero tells GYST that the DLL call executed without error. A non-zero return code indicates that some error occurred within the DLL and spawns a DLL-dependent error message. Additional information about error codes is given in a subsequent section.

An example of a C function definition for the stand_grow function compiled using Microsoft Visual C++ (® Microsoft Corporation) begins with:

```
int far pascal export stand_grow(char far * string, char far * workbuffer)
```

This function calling convention is followed to allow compatibility with DLLs written in Fortran. The corresponding stand_grow function compiled using DIGITAL ® Visual Fortran 6.0 (Digital Equipment Corporation) begins with: `integer function STAND_GROW (string, workbuffer)`. The only portions of these lines that should be changed by the DLL programmer are the argument names, *string*, and/or *workbuffer*. The data appearing at the cursor location in the active spreadsheet will be encoded into the array pointed to by the variable *string*, and the "hidden" data string is pointed to by the variable *workbuffer*. The *workbuffer* string will be an empty string for the first entry point call of any GYST session. Until a DLL function places data in the *workbuffer* array it will remain empty (null). It is important to note that that function names (such as STAND_GROW in the above example) must be upper case letters for the Fortran programming language.

Programmers should avoid defining functions with names differing from the entry point names only by letter case. By the pascal calling convention, all function names in C are converted to upper case by the compiler.

Entry Point Functionality

Table 2 describes the basic purposes of the twelve DLL entry point functions. It also indicates the combination of active-window and button-clicks that would result in a particular DLL function being called. For instance, when the tree-level window is active and the user clicks the 

button (alternatively, the "Model" menu "Update Stand" command) GYST will call the function tree_to_stand. Assuming the user has placed the cursor at the beginning of a series of lines of data and that each line is made up of variable values that describe a particular tree, the objective of the user is clearly to aggregate the tree list into a stand-level summary. Output of the tree_to_stand conversion is completely dependent on the algorithms coded in the particular DLL that the user has linked to GYST during this session.

Exactly what a particular entry-point function should do is evident in its function name, but the specifics of initialization, growth, and conversion are dependent on the data requirements and abilities of the models programmed into the DLL. Programmers should consider the

following note about “Grow” entry points, those accessed by clicking the  button (alternatively, the “Model” menu “Grow” command). GYST will repeatedly call DLL grow functions until a user-specified number of iterations has been completed. The DLL should perform one growth iteration per function call. The length of the iteration is determined solely by the DLL; furthermore, the growth interval may be specified as a function of time (e.g. age) or some other variable in the input list.

Extending the Capabilities of DLLs

The entry points for DLL functions were designed to accomplish tasks associated with growth and yield modeling at three typical levels of resolution: stand, diameter distribution, and single tree. However, the program framework is not limited to these capabilities. As long as the DLL programmer conforms to the data passing conventions described above, it is not necessary to limit the use of GYST for stand, diameter distribution, and tree-level modeling. The framework of GYST is sufficiently flexible to allow a programmer to transform virtually any list of input variables into a block of output data.

For example, a tree list made up of multiple species could be aggregated to the stand level preserving species information using the tree_to_dist entry point function. The DLL function would aggregate the input tree list into stand-level attributes for each species in the tree list. The output string would consist of multiple lines of stand-level data, one line per species in the input tree list. In this case, the conversion is not truly from the tree-level to the diameter distribution-level, but instead to a useful stand-level-by-species resolution. Of course, the user would have to be made aware that the  button (alternatively, the “Model” menu “Update Diameter” command) will generate a stand/species level conversion for this particular DLL rather than a diameter distribution-level conversion.

By taking advantage of the flexibility with which GYST interacts with its DLLs, developers can use GYST to interface with many types of models at a variety of resolutions. It is quite reasonable to expect that models across a broad range of resolutions - from physiological models with resolutions of a single leaf to landscape models with coarse spatial resolutions - can be implemented using the GYST shell.

PARSING AND REBUILDING STRINGS

Since the variable values (data) in the spreadsheet views of GYST are combined into a character string and passed by pointer to the DLL for processing, developing DLLs for use with the GYST shell requires the use of code to manipulate character strings. Character arrays (strings) are a common variable type in both C and Fortran. Once appropriate general routines have been written for accomplishing string parsing and rebuilding tasks, they can be used over again with only minor adjustments.

Parsing a Data String
Using the C Language

Once the incoming character string from GYST has been received by the DLL, it must be parsed (separated) into the variable values to be used by the growth and yield algorithms within the DLL. We have developed a useful routine (available as a .txt file with the GYST system) for parsing a data string in C (Table 3).

Table 3. Example C function for breaking apart the incoming *string* variable from GYST into the values from the spreadsheet view. (Line numbers are added for ease of reading and are not part of the programming language).

```
/*************************************************************************/
/* parse skips ahead to the next field in the input string.               */
/* All fields are delimited by tab characters except for the last        */
/* field which is followed by a carriage return and newline              */
/* character.                                                            */
/* NOTE: ptr points to the GYST string variable                          */
/*       buffptr points to the part of string which contains the value of  */
/*              interest                                                 */
/*       beginstr points to the place in string where we start looking for */
10/*               a value of interest
*/
/*************************************************************************/
void parse(char *ptr, char *buffptr, int *beginstr) {
  int a=0;
  char temp;

  /* increment a until '\t', '\r', or '\n' character encountered */
  /* this gets you to the end of the current variable in the string */
  while ((*(ptr+*beginstr+a) !='\t') && (*(ptr+*beginstr+a)!='\r') &&
                 (*(ptr+*beginstr+a) !='\n'))a++;
20
  /* save the character at this position in a temp variable */
  temp = *(ptr+*beginstr+a);

  /* put a NULL character at this position */
  *(ptr+*beginstr+a)='\0';

  /* now copy this piece of the string to the buffptr */
  /* if a blank value is encountered, a == 0 and ptr+*beginstr == NULL */
  strcpy(buffptr,(ptr+ *beginstr));
30
  /* replace the NULL character with what was originally there */
  *(ptr+*beginstr+a)=temp;

  /* set *beginstr to the (presumed) beginning of the next variable */
  *beginstr+=a+1;

  /* take care of situation where string is missing right-hand values */
  if(temp == '\n')                          /* indicates nothing read */
  {
40  *beginstr+=(-1);               /* reset beginstr to where it was  */
    strcpy(buffptr,"0"); /* send a zero back because nothing was read */
  }
}
```

Using the parse( ) function to break apart a string into its component variables is

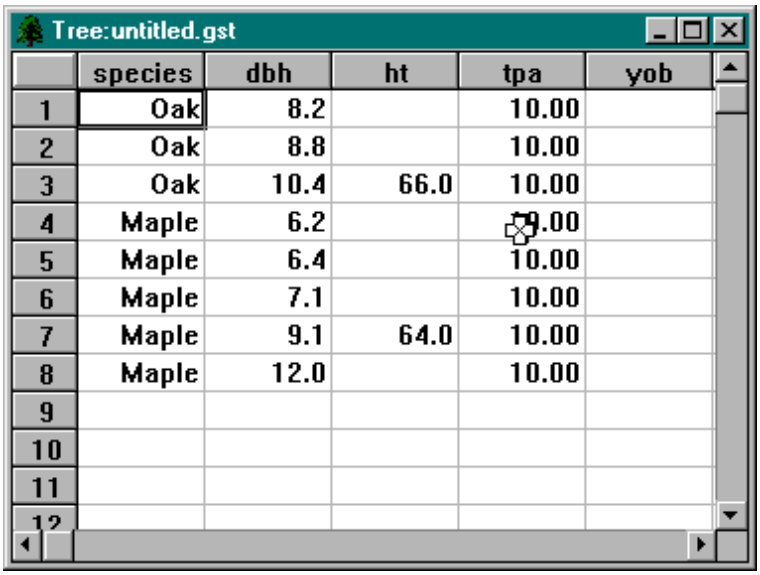
illustrated in Table 4 (this parse function available as a .txt file with the GYST system). At line 20 the array index counter is set to zero, corresponding to the first element in the string. The call to parse( ) in line 21 places the contents of *string* to the left of its first '\t' character into the variable *buff*. The contents of *buff* are subsequently copied to the variable *key* (line 22). The call to parse( ) in line 24 places the next variable value in *string* into the variable *buff*. In line 25 the contents of *buff* is converted to an integer value by atoi( ) and stored in the variable *si*. The process is repeated to obtain values for the variables *age*, *hd*, and *tr* (note: atof( ) is used to convert string to floating-point variable types).

Table 4. C code illustrating the use of function parse( ) to extract variable values from a GYST input string. Line numbers are added for ease of reading and are not part of the programming language.

```
int far pascal export stand_grow(char far * string, char far * workBuffer) {

/*********************************************************************/
     /* variable declarations
*/
     /* beginstr - array index for the part of string that parse() looks at
*/
     /* si - site index
*/
     /* age - stand age
*/
     /* buff - part of string containing a single variable value
*/
     /* key - user input stand ID or description
*/
     /* hd - dominant/codominant tree height
*/
     /* tr - trees per acre
*/

/*********************************************************************/
     int beginstr, si,age;
     char buff[256], key[256];
     double hd, tr;
     /*****************************************************************/
     /***** Break the stand-level string down into its components ******/
     /* Order of variables in string:                                 */
     /* key, si, age, hd, tr                                          */
     /*****************************************************************/
     beginstr=0;
     parse(string,buff,&beginstr);
     strcpy(key,buff);

     parse(string,buff,&beginstr);
     si=atoi(buff);

     parse(string,buff,&beginstr);
     age=atoi(buff);

     parse(string,buff,&beginstr);
     hd=atof(buff);

     parse(string,buff,&beginstr);
     tr=atof(buff);

/* check for nonsensical input and continue by calling growth functions */
```

Consider the example illustrated in Figure 2. The string that would be generated by GYST and passed (by pointer) as argument 1 to a tree-level entry-point function for this data is printed below the spreadsheet image in the figure. Note that GYST does not insert a '\t' character after the value 10.00, which is the final input value on each line. Furthermore, the blank value for the fifth variable (*yob*) on each line is ignored by GYST. In general, the final non-blank variable value in a data line is not followed by a tab character. GYST inserts blank values (denoted by consecutive tab characters "\t\t") into the data string only if there is at least one occupied cell to the right of the blank value(s) in the data line.

These properties are illustrated by the input string in Figure 2. For lines where the *ht* variable is blank, consecutive tab characters are written to the string. Note that if there had been two blank values followed by an occupied cell, GYST would have written three consecutive tab characters before the occupied cell values. Every *tpa* value (10.00) is immediately followed by a carriage-return/newline combination, indicating the end of the data line. As noted previously, there are no tab characters after the last non-blank data value in a line.



	species	dbh	ht	tpa	yob
1	Oak	8.2		10.00	
2	Oak	8.8		10.00	
3	Oak	10.4	66.0	10.00	
4	Maple	6.2		10.00	
5	Maple	6.4		10.00	
6	Maple	7.1		10.00	
7	Maple	9.1	64.0	10.00	
8	Maple	12.0		10.00	
9					
10					
11					
12					

```
string =
"Oak\t8.2\t\t10.00\r\nOak\t8.8\t\t10.00\r\nOak\t10.4\t66.0\t10.00\r\nMap
le\t6.2\t\t10.00\r\nMaple\t6.4\t\t10.00\r\nMaple\t7.1\t\t10.00\r\nMaple\
t9.1\t64.0\t10.00\r\nMaple\t12.0\t\t10.00\r\n"
```

Figure 2. Example treelist as viewed in the GYST tree-level spreadsheet.

The parse( ) function returns (by pointer) a string with a single zero character for any right-side blank values it encounters in a line of data (Table 3, lines 37-42). However, parse( ) does not return a zero string ("0") when it encounters blanks denoted by consecutive tab characters. Instead, it returns a null string, which is converted to zero by the atoi( ) or atof( ) function (Table 4).

Reconstructing a C Data String for Output to GYST

Reconstructing a data string for output to GYST is the opposite of parsing the input data string, but is greatly simplified by the use of the sprintf( ) function defined in `stdio.h`. The data stored in variables key, si, age, hd, and tr in Table 4 can be passed to the GYST stand-level spreadsheet by the statement:

```
sprintf(string,"%s\t%d\t%d\t%.1f\t%.1f\r\n",key,si,age,hd,tr);
```

Note that values within the data line are delimited by tab characters and a carriage-return/newline sequence signals the end of the data line.

Parsing a String Using Fortran

Table 5 (available as table5.txt with the GYST system files) shows the Fortran code for an example STAND_GROW function that receives the incoming tree list character string from GYST and divides it into the individual variable values. The data are then passed to the functions for computing growth and yield values, after which the string is reconstructed for passing back to GYST.

Table 5. Fortran code of an example STAND_GROW function for the GYST shell.

```
C*************************************************************************
C     The STAND_GROW function grows the model values when the Grow
C     toolbar button is clicked in GYST
C*************************************************************************
C
C***Section 1 for defining variables and breaking down the character variable "string"*****
C
      integer function STAND_GROW (string,buffer)
c
      character*1000 string
      character*100 buffer
      integer strbeg, strend, drainage, si, age,
     1 nit, phos, numtrees,years,id
      real*4    ba,hd,hdr,bar,survr
      character*80 tmpstr
      character*1 tab
      character*1 nl
      character*1 cr
      character*1 null
c
      tab = char(9)    !the tab separates variables in the string
      nl  = char(10)   !signals line feed
      cr  = char(13)   !cr signals end of a record (line)
      null = char(0)   !ends the string
```

```
c
      stand_grow=0                          ! this is the return code: 0 = O.K.
       strbeg = 1                          ! now call the parse subroutine as many times as needed
       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I8)') id                    !get id
         if(id.gt.9999.or.id.lt.0) then             !error trap it
             stand_grow=8
             return
         endif
       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I8)') drainage              !get drainage
         if(drainage.gt.2.or.drainage.lt.0) then    !error trap it
             stand_grow=1
             return
         endif

       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I8)') si                    !get site index
         if(si.gt.90.or.si.lt.40) then              !error trap it
             stand_grow=2
             return
         endif

       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I8)') age                   !get age
         if(age.gt.30.or.age.lt.8) then             !error trap it
             stand_grow=3
             return
         endif

       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I8)') numtrees              !get number trees
         if(numtrees.gt.1000.or.numtrees.lt.100) then  !error trap it
             stand_grow=4
             return
         endif

       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,F5.1)') ba                  !get ba
          read(tmpstr,'(BN,F5.1)') ba               !get ba but check first for null character
          if((ba.gt.0.0.and.ba.lt.30.0).or.(ba.gt.150.0)) then !error trap it
             stand_grow=5
             return
          endif

       call parse (string,strbeg,strend,tmpstr)
       read(tmpstr,'(BN,I3)') nit                   !get nitrogen
         if(nit.lt.50.or.nit.gt.300) then           !error trap it
             stand_grow=6
             return
         endif
```

```
      call parse (string,strbeg,strend,tmpstr)
      read(tmpstr,'(BN,I2)') phos                !get phosphorus
        if(phos.lt.0.or.phos.gt.50) then         !error trap it
            stand_grow=7
            return
        endif

      call parse (string,strbeg,strend,tmpstr)
      read(tmpstr,'(BN,I2)') years               !get years from fert
        if(years.lt.0.or.years.gt.25) then       !error trap it
            stand_grow=9
            return
        endif
C
C***end Section 1 and start Section 2 which contains calls to growth and yield models***
C
C    Place growth and yield calculations here with calls to
C    functions and subroutines as necessary
C
C*************************************************************************
    hd=hdf(si,age)                                !dominant height at fertilization
    if(ba.eq.0.0) then
     ba=baf(numtrees,hd,age,si)                   !if ba at time of fert
                                                  ! not input, compute it
    endif
    years=years + 1                               !project it
    hdr=hdres(drainage,si,hd,age,numtrees,years,nit,phos)  !hd response
    bar=bares(drainage,hd,numtrees,ba,years,nit,phos)      !ba response
    survr=survres(si,age,numtrees,nit,years)      !survival response
C*************************************************************************
C
C***end Section 2 and start Section 3 which reconstructs the new character variable "string" for
C***passing back to GYST
C
C    Done with growth and yield computations so place
C    the variables back into a string for returning to GYST
C
    strbeg = 1
     write(tmpstr,'(i4)') id
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
     write(tmpstr,'(i1)') drainage
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
     write(tmpstr,'(i2)') si
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
```

```
      strbeg = strbeg + 1
    write(tmpstr,'(i2)') age
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(i4)') numtrees
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(f5.1)') ba
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(i3)') nit
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(i2)') phos
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(i2)') years
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(f4.2)') hdr
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(f4.1)') bar
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
      string(strbeg:strbeg) = tab
      strbeg = strbeg + 1
    write(tmpstr,'(f6.1)') survr
      call build (string,strbeg,strend,tmpstr)
      strbeg = strend + 1
        string(strbeg:strbeg) = cr    !end of  record
      strbeg = strbeg + 1
        string(strbeg:strbeg) = nl    !new line
      strbeg = strbeg + 1
        string(strbeg:strbeg) = null  !end of string
c
    end
```

The entry point STAND_GROW has three "sections". The first section is for variable declaration and parsing *string* into the separate variables that comprise the spreadsheet view. Variables *strbeg* and *strend* are pointers that indicate the beginning and ending positions of a portion of *string*. The variables *string*, *strbeg* and *strend* are passed to subroutine parse which returns updated values of *strbeg* and *strend* along with a character variable called *tmpstr* that contains the variable of interest. Then an internal read statement is used to convert *tmpstr* to an integer or real number. This first section is also where error trapping can be accomplished (more on error trapping later).

The second section contains calls to growth and yield functions and subroutines using the variables that were parsed from *string*. The third section takes the new values computed from the functions and subroutines and reconstructs the character variable *string* for passing back to GYST.

A 0 integer return value means the DLL executed properly and GYST can update the spreadsheet view with the new values. A non-zero integer return value means that an error occurred during DLL execution and GYST will display a programmer-defined error message corresponding to the integer return value.

The actual parsing of *string* is accomplished in subroutine parse shown in Table 6 (available as a .txt file with GYST).

Table 6. An example Fortran subroutine for breaking apart the incoming *string* variable from GYST into the values from the spreadsheet view.

```
C*****************************************************************************
C   This is the parse subroutine for splitting out variables from
C   the string that is passed into the DLL from GYST
C*****************************************************************************
      subroutine parse (string,strbeg,strend,tmpstr)
      character*1000 string
      character*80 tmpstr
      integer strend,strbeg
      character*1 tab
      character*1 cr
      character*1 nl
      character*1 null
c
      tab = char(9)
      cr = char(13)
      nl = char(10)
      null = char(0)
c
c  First special case where two tabs are together.  For this case
c  we want to pass back a zero in tmpstr.
c
      if (string(strbeg:strbeg).eq.tab) then
```

```
        strbeg = strbeg + 1
        strend = strbeg
        tmpstr = '0'      !if two tabs together, send back a zero value.
        return
      endif
c
c  Check for end of stand record
c
      if (string(strbeg:strbeg).eq.nl) then
        strbeg = strbeg + 1     !check for the end of the stand list (null)
        if (string(strbeg:strbeg).eq.null) then
          tmpstr = null
          strend = strbeg
          return
        endif
        return
      endif
c
c  Here we get the good data into a temporary string by looking for a
c  tab or cr character
c
        strend = strbeg
 200  if ((string(strend:strend) .eq. tab).or.
     1    (string(strend:strend) .eq. cr)) goto 300
        strend = strend + 1
        goto 200
 300  tmpstr = string(strbeg:strend)
       strbeg = strend + 1
       strend = strbeg
       return
       end
```

The variable *string,* received from GYST, contains individual stand variable values. The integer variables *strbeg* and *strend* are pointers which indicate the beginning and ending position, respectively, of a segment of *string* which contains a variable value. The variables *string*, *strbeg* and *strend* are passed to subroutine parse which looks for a tab or newline character to identify which portion of *string* contains the variable of interest. It then uses an internal write statement to put that portion of *string* into the variable *tmpstr* which is then passed back to function STAND_GROW. In function STAND_GROW, another internal write statement is used to convert *tmpstr* into the proper character, real or integer value. This process is continued until the entire contents of *string* have been divided out and assigned to proper variables for use in growth and yield models.

Rebuilding a Fortran String

After processing through growth and yield models, *string* must be reconstructed with the updated stand variables for passing back to GYST. This is just the reverse of the parsing process. Table 7 (available as a .txt file with GYST) shows Fortran code for subroutine build that reconstructs *string* for passing the updated information to GYST.

Table 7. An example Fortran subroutine for reconstructing the *string* variable with updated variable values for passing back to GYST.

```
C*************************************************************************
C   This is the build subroutine for reconstructing
C   the string that is passed into GYST from the DLL
C   It essentially reverses the PARSE routine
c*************************************************************************
      subroutine build (string,strbeg,strend,tmpstr)
      character*1000 string
      character*10 tmpstr1,tmpstr
      character*1 char(10)
      equivalence (tmpstr1, char(1))
      integer strend,strbeg
c
c  Find the number of non-blank characters of tmpstr
c
      tmpstr1 = tmpstr
      do 10 i=1,10
        if(char(11-i) .ne. ' ') go to 20
   10 continue
   20 lnblnk = 11-i
c
c  Put the non-blank characters into tmpstr1 which shortens it
c
      do 30 i=1,lnblnk
        tmpstr1(i:i) = char(i)
   30 continue
c
c  Now put tmpstr1 into string without any trailing blanks
c
      strend = strbeg+lnblnk
      string(strbeg:strend) = tmpstr1
      return
      end
```

Error Checking

Initial error checking can be done after parsing the input string and prior to calling customized initialization, growth, or conversion algorithms. Error checking at other stages of function execution may be appropriate to a particular DLL. Non-zero return values are used by

GYST to produce error messages for the user. Each nonzero integer is linked by GYST to a pre-defined set of error messages that can be displayed to the user. For example, if the error code 1 is returned to GYST by the DLL, the error message corresponding to error code 1 will be displayed to the user.

Error messages for display to the user are stored in a text file in the working directory of the GYST executable program. The error message file shares the same filename as the DLL but has the filename extension **`.err`**. For example, the error message file associated with a DLL named `demo.dll` would be saved under the filename `demo.err`. Table 8 gives a listing for a simple error message file capable of reporting six error messages to the GYST user. Note that each error message occupies one line of the file and is preceded on that line by a nonzero integer and a space delimiter. The integer should be matched to the return value that the DLL sends to GYST in the event of an error. From the message contents in Table 8 it is apparent that error codes 1-3 would be generated in the stand_grow function, and error codes 4-6 would be generated in the stand_ini function.

Table 8. Sample error message file for a GYST DLL.

```
1 Cannot grow a stand of this age.  Set age between 0 and 36
2 Cannot grow a stand with this number of trees.  Set the value between 80 and 1500
3 Cannot grow a stand with this site index.  Set site index between 40 and 85
4 Cannot initialize a stand of this age.  Set age between 8 and 36
5 Cannot initialize the stand. Set the number of trees between 80 and 1500
6 Cannot initialize a stand with this site index.  Set site index between 40 and 85
```

To aid in programming, the DLL developer may find it useful to define error codes with a series of preprocessing directives. These may be organized in a header file such as the listing given in Table 9. On encountering an error, the program returns a preprocessor definition (e.g. `return GROW_AGE_LIMIT`) instead of the integer value it represents (e.g. `return 1`). The advantage to the programmer is that the preprocessor definition should be more meaningful in the code than an integer return value.

Table 9. Header file of preprocessor commands to define error codes.

```
/* errors.h */

#define OK               0
#define GROW_AGE_LIMIT   1
#define GROW_TR_LIMIT    2
#define GROW_SI_LIMIT    3
#define INIT_AGE_LIMIT   4
#define INIT_TR_LIMIT    5
#define INIT_SI_LIMIT    6
```

CREATING A .DSC FILE FOR A DLL

GYST must have some way of recognizing the characteristics of the variables that will appear in the spreadsheet views for a particular session and that are passed to the DLL for

processing by growth and yield models. The way this is done is through the **.dsc** file, which provides GYST with a description of the variables. The following rules apply to **.dsc** files.

1. For each DLL available to GYST there must be a corresponding **.dsc** file located in the same directory as the DLL. For example, if there is a *demo.dll* file then there must also be a *demo.dsc* file that GYST can access in the same directory.

2. Each **.dsc** file must have a section that matches each spreadsheet view in the DLL. The *[stand]* section contains information about the variables used in the stand-level spreadsheet view. The *[distribution]* section of the **.dsc** file contains information about the variables used in the distribution-level spreadsheet view. Likewise, the *[tree]* section contains information about the tree-level variables.

3. Each variable has up to four arguments that describe it with each argument separated by a comma. The first argument is a name. The name will appear as a column header in the spreadsheet view and identify the variable to the user. A name may occupy up to seven lines in the column header. The ^ symbol separates lines in the column header (Table 10).

The second argument defines the variable type. There are three possible variable types: character (char), integer (int), and floating point real (float).

The third argument is for real variables and tells GYST how many decimal places to carry for that particular real variable. **It is very important to remember that GYST will only hold as many decimal places for a particular real variable as is specified in the .dsc file**. Developers of DLLs must consider the impact that rounding of real numbers may have on the precision of growth projections and make sure that enough decimal places are carried by GYST to ensure precise results. This may be particularly important when developing individual tree growth and yield models.

The fourth argument defines which variables in the spreadsheet view can be used in the "Grow Options" window for making projections to user specified variable values (the "Grow to..." selection). A "p" in the fourth argument position indicates to GYST that that particular variable can be projected to a specified value. Thus only variables with a "p" for the fourth argument will appear in the "Grow to" window. For integer variables that would be used in the "Grow to..." selection, a space delimiter for the third parameter would be used (e.g. age,int, ,p).

Table 10 shows a **.dsc** file for a stand-level growth and yield model DLL that has stand-level view output for GYST but not distribution- or tree-level view output. It has eight integer variables and four real variables and projections using the "Grow to..." options can be accomplished using the *yrs_frm_fert* or the *ba_resp* variables.

Table 10. Example **.dsc** file for a stand-level DLL that does not have active windows for the

distribution or tree views.

```
[stand]
Tree^Number, int
Drainage, int
Site^Index, int
Fertilization^Age, int
Trees^at^Fertilization, int
Basal^Area^at^Fertilization, float,1
Nitrogen^(lbs/ac), int
Phosphorus^(lbs/ac), int
Years^from^Fertilization, int, , p
Dominant^Height^Response, float,2
Basal^Area^Response, float,1, p
Survival^Response, float,1
```

It must be remembered that the order of the variables that appear in each section of the .dsc file is the order that will be shown in each spreadsheet view and also the order of the variables in the character string passed into and out of GYST. That is, GYST creates the character string for passing to the DLL from the variable list in the **.dsc** file. Therefore, the developer of the DLL must also create the corresponding **.dsc** file and users must not alter this file after it has been created.