THOROUGHBRED®
PROGRAMMING
COOKBOOK

BCS TECHNOLOGY LIMITED
805 Archer Lane
Elwood Illinois 60421

January 2009

Thoroughbred Programming Cookbook

Tenth Printing, January 2009

TABLE OF CONTENTS

INTRODUCTION

This *THOROUGHBRED PROGRAMMING COOKBOOK* manual contains technical information on a variety of general purpose functions and programs that have been developed by **BCS TECHNOLOGY LIMITED** for use with *Thoroughbred* BASIC 3GL programs. Cookbook functions act like "black boxes," encapsulating many programming tasks often found in large-scale software development projects. Using cookbook functions instead of scratch-developed code, you should be able to substantially reduce development and debugging time, as well as produce more structured and understandable software with improved features.

The *THOROUGHBRED PROGRAMMING COOKBOOK* includes both public programs and a comprehensive set of user-defined functions. The defined functions were developed to simplify conversion operations between IDOL-IV data formats and conventional 3GL variables, as well as to implement generally useful operations, such as date and time manipulation. Once the defined function set has been merged into the IDOL-IV help library, any function can be added to a program from within the `EDITF` program text editor. Refer to page 31 for more information.

LICENSING AND DISTRIBUTION

BCS TECHNOLOGY LIMITED retains sole rights to the ownership and distribution of the cookbook, all related software and accompanying documentation (e.g., this manual). We convey a restricted license to *Thoroughbred* developers to include the cookbook object library `bcs.lib` and accompanying utilities—except `cksumlib`—with software distributions intended for installation on end user systems. As part of this restricted license, we require that all cookbook functions be retained in the object library and that no program be modified in any way. Also, a copyright acknowledgment worded as follows must be present in any documentation provided with the software of which the cookbook is a component:

Distribution in any manner of the cookbook object library, accompanying utilities, the `DMDEF_FN` defined functions document file or this manual contrary to this restricted license is a violation of United States and international copyright law. *Distribution of the* **DMDEF_FN** *defined functions source file and/or the* **cksumlib** *utility to end users is prohibited.*

TYPOGRAPHICAL CONVENTIONS

This manual has been printed with the CG TIMES font. To distinguish various aspects of the text and thus avoid ambiguity, font changes and typographical conventions will apply as follows:

`Courier`	Program text to be entered as part of a function call, variable names, or values and/or character strings involved in processing.

Courier Bold User responses to input requests or when a program example is presented, highlights text of portion of the example where the function is mentioned.

{Courier} Curly braces are used to set off portions of program text that will vary depending upon the desired function. For example, input{ned} means that the function to be called may either be input or inputned.

[Courier] Rectangular brackets are used to delimit optional function parameters. For example, CALL "clrfmts"[,CMMAS$] indicates that CMMAS$ is an optional parameter. Do not confuse these brackets with those required by *Thoroughbred* array syntax, such as L$[ALL].

CG Times Italic Italic is used to emphasize a point or highlight a word or phrase that is key to a discussion. For example, *The object of being in business is to make money*.

[KEY] The general format for describing any key on the keyboard, where **KEY** is the keytop description. For example, **[F1]**.

⏮ Symbol for the **[RETURN]** or **[ENTER]** key.

[KEY1][KEY2] The general format for describing any keypress combination, in which **[KEY1]** is held down and while holding down **[KEY1]**, **[KEY2]** is typed. For example, **[CTRL][C]**.

In some narratives, references are made to the user pressing the "escape key," symbolized by **[ESC]**. The actual key code expected may be defined by a special read-only 'EK' mnemonic in the terminal driver table (TCONFIGW) being used by the task. In terminal tables provided by ***BCS TECHNOLOGY LIMITED***, 'EK' has been defined as 03 (<ETX> in the ASCII character set). Many terminals, such as the WYSE 60 and derivatives, support remapping of the keyboard. If such remapping is possible, **[ESC]** (or some key that has been arbitrarily designated as **[ESC]**) should be arranged to emit <ETX> when pressed (this usually can be accomplished in software—refer to the terminal's technical manual for details). On terminals that differentiate between the shifted and nonshifted **[ESC]** key, **[SHIFT][ESC]** should not remapped and thus should generate an <ESC> ($1B$) character as expected. If a terminal keyboard cannot be remapped the user will be required to press **[CTRL][C]** to generate the equivalent of **[ESC]**. Note that this definition of the escape key is not related in any way to the ASCII value established in the terminal driver table by question 09H2 (the BASIC escape code—usually defined as **[CTRL][X]**).

COOKBOOK OBJECT LIBRARY

The *THOROUGHBRED PROGRAMMING COOKBOOK* public programs described in this manual are encapsulated in an object library named `bcs.lib`. The object library method of distribution was selected so as to simplify packaging and maintenance of the software, as well to enhance system performance. *As part of the restricted license, you are required to retain all cookbook functions in the object library*. Breaking down the object library and running the individual programs from disk will result in system performance degradation and could result in significant problems if a cookbook program is deleted, modified or otherwise misplaced. *Do not modify any software distributed with the cookbook*. Should a new version of the cookbook be distributed your modifications will be lost. Also, there are many software dependencies in the cookbook functions. *Modifications could cause widespread havoc*.

OPENING THE COOKBOOK OBJECT LIBRARY

Prior to using any cookbook functions you must open the `bcs.lib` object library, which may be accomplished in one of two ways:

```
OPEN (CH,OPT="OLIB") "bcs.lib"
```

or

```
PRM OPENLIB=bcs.lib,<CH>
```

The first method opens the library on channel `CH` from within a running program. If you elect to use this method be sure to utilize a channel number equal to or greater than `32000` to protect the library from being accidentally closed by the `closeall` function (page 78). You can get a high unused channel number from within a running program with the following call:

```
CALL "hichan",CH
```

`hichan` (provided on the cookbook release disk along with several other utilities) will return a suitable channel number in `CH` or zero if no open channel equal to or greater than `32000` can be obtained—an unlikely event.

The second method utilizes a `PRM` statement embedded into the IPLINPUT file used to start *Thoroughbred*, causing `bcs.lib` to be automatically opened on channel `<CH>` when the task is started. For example:

```
PRM OPENLIB=bcs.lib,32100
```

will open `bcs.lib` on channel `32100`.

The IPLINPUT method has an advantage in that *Thoroughbred* will complain and abort if it cannot find the object library or if it determines the library's internal structure has been corrupted. This behavior may be a desirable feature in installations where the software cannot be run without the services of `bcs.lib`. However, in older versions of *Thoroughbred* BASIC there is a limit of eight `PRM OPENLIB` statements in an IPLINPUT file. If your version of BASIC has this limit only those object libraries that will be used system-wide should be opened in the IPLINPUT file. All others should be opened and closed as required from within your programs. As with opening the library in a running program, the `CH` value should be at least `32000` to protect the library from accidental closure.

Once opened, an object library consumes one of the file descriptors that are available to the task, the number of which is defined on the `CNF` statement line in the IPLINPUT file used to start the task. If you are using other object libraries along with `bcs.lib` be sure that enough file descriptors have been allocated to the task to allow the opening of those libraries, as well as other files. Also, be sure that the maximum number of *per user* open files permitted by the operating system is sufficient to allow all tasks to run concurrently.

COOKBOOK OBJECT LIBRARY INTEGRITY

Like any other file, the cookbook object library can be inadvertently modified or corrupted, events which may result in serious problems with any software dependent on the integrity of the library. At the time of creation, a 32 bit cyclic redundancy checksum (CRC) is computed for `bcs.lib` and the result is embedded into the library header. You can verify the CRC of your copy of `bcs.lib` by checking it with the `cklibcrc` utility included with the object library distribution:

```
CALL "cklibcrc","bcs.lib";
ON ERR(1,2,3) GOTO OK,BADCRC,NOCRC,BADFILE
```

More discussion on library integrity checking may found in the section on developing your own object libraries (page 25). See also the `cklibcrc` (page 76) narrative for a detailed description of this utility.

4GL TO 3GL INTERFACE

Many of the functions included in the *THOROUGHBRED PROGRAMMING COOKBOOK* have been created to facilitate the interaction of conventional third generation language (3GL) programs with the advanced fourth generation language (4GL) features of *Thoroughbred* Dictionary-IV. Therefore, if you wish to make the most of the cookbook you will need to possess a reasonably good understanding of the 3GL language primitives that are available to access Dictionary-IV features.

The transition from *Thoroughbred* BASIC 7.x to 8.x was marked by some significant language enhancements, one of them being the introduction of true 4GL capabilities—the Dictionary-IV environment. In *Thoroughbred* BASIC, directives and functions, such as `FORMAT INCLUDE` and `FMD`, provide the 3GL language primitives and the IDOL-IV package provides a high level interface to the primitives. We will only give IDOL-IV superficial coverage here, as everything you might need to know to use IDOL-IV features is thoroughly documented in both the manuals available from *Thoroughbred* and the IDOL-IV on-line help facilities. Our real interest is in using language primitives and cookbook functions to take advantage of 4GL concepts without sacrificing the inherent control and performance of the 3GL environment.

FOURTH GENERATION LANGUAGE BASICS

A *Thoroughbred* 4GL program is built up from a combination of high level language statements and abstractions called *objects*, the most important of which from a 3GL perspective are *formats* and *links*. These two object types are used in concert as an interface to data files and are joined with other objects, such as *screens* and *views*, to produce the IDOL-IV environment. Although a complete 4GL package, such as *Thoroughbred*'s Solution-IV accounting software, will incorporate all object types, the following discussion will focus on formats and links.

A format is a named data structure that is conceptually similar to an `IOLIST` in 3GL BASIC or a `struct` (structure) in ANSI C. A format's definition includes one or more named elements or fields—analogous to the variables in an `IOLIST` or the members of a `struct`—with each element being assigned, as a minimum, attributes that define the amount and type of data that it can store. Other attributes may specify how to pad partially filled fields, provide a transparent interface to specialized data types or optionally assign a spoken language description that may be used in your programs. An element may be defined to have multiple occurrences, a feature that is conceptually identical to a single dimension 3GL array. Proper usage of contiguous multiple occurrence elements can produce the equivalent of a multi-dimension 3GL array.

Most formats are used as a template for the internal layout of the records of a file, a format of this type being referred to as a physical format. A format that has no relationship to any file is a called a logical format. Later on, we will present examples of how to use both logical and physical formats together to achieve specific results, such as printing a line of data on a report.

A link is also a named data structure—itself defined by a format—whose purpose is to relate a data file on disk with its associated 4GL objects—format, screens, views and so forth. The retrieval of a link from the *Thoroughbred* data dictionary automatically provides the information needed to open a file, load a format into memory and so forth. *A key benefit derived from using a link is portability*: since a link defines all pertinent information about a file and its associated 4GL objects, a change to a link, such as the renaming of a format, will automatically be recognized by any program that uses link information instead of hard coded object references.

Speaking of names, it is appropriate to briefly comment on object naming conventions. In many of the narratives presented in this manual, references will be made to format names or link names, and examples such as `LLNNNNNN` will be included in the discussion. The `LL` component is the IDOL-IV library mnemonic, such as `AP` for accounts payable, `IC` for inventory control, `OP` for sales order processing, and so forth.[1] The `NNNNNN` component is a name you assign, such as `VMMAS`, `CMSHP` or `CMMAS1`. For example, the link to the accounts payable vendor master database might be named `APVMMAS` or the customer consignee (ship-to) link could be named `OPCMSHP`.

At **BCS Technology Limited**, we've developed a standardized method of assigning names to our 4GL objects: `LLMMNNNX`, in which the `LL` component is the IDOL-IV library mnemonic, `MM` is a "sub-mnemonic" indicating which part of the library is involved, `NNN` is the object and `X` is a version or variation of that object. For example, the object name `OPCMMAS` would be interpreted as: order processing library (`OP`); customers sub-library (`CM`); master record (`MAS`). Continuing with this model of name commonality, `#OPCMMAS` would be the customer master record format, `OPCMMAS` would be the link, and so forth (name commonality is discussed in the next paragraph). Variations on the `#OPCMMAS` format might include `#OPCMMASA` (a second copy of this format, used when copies of two customer masters must be simultaneously available), `#OPCMMASD` (a display format, which is a logical format structured to mirror a screen display), and so forth. Other objects that might be found in the `OP` library would be `OPSOHDR` (sales order headers), `OPSODET` (sales orders line item details), and so forth. "Generic" objects, such as the `#GCPAGHDR` page header logical format you will read about later on, usually do not adhere to this convention, other than the IDOL-IV library mnemonic.

We have extended this convention to filenames as well: `opcmmas.idx` would be used as the filename for an `MSORT` or `ISAM` file containing customer master records referred to by the `OPCMMAS` link and `#OPCMMAS` format.[2] While you are certainly free to adopt any conventions you wish, we urge you to devise a similar method to make development easier in large-scale projects.

---

[1] If you aren't sure what an IDOL-IV library is we suggest spending an evening curled up on the couch with the *Thoroughbred* IDOL-IV manuals—somewhat dry reading but very informative.

[2] For the sake of easier manipulation in the operating system shell, we recommend you utilize lower case filenames for both data files and programs, and avoid the embedding of blanks or other characters that are significant to the shell.

HARD OR SOFT CODED: WHICH IS BETTER?

If you have read this far you are most likely interested in incorporating 4GL features into your 3GL programs. If so, consider the following code fragments:

```
FORMAT INCLUDE #MYFORMAT;
#MYFORMAT.NUMBER(2)=123.45
```

The above is a hard coded format reference: the format and element names are explicitly stated in the program.

```
MYFMT$="#MYFORMAT",
NUMBER=5,
OCC=2;
FORMAT INCLUDE #MYFMT$;
LET FMT(MYFMT$,NUMBER,OCC)="123.45"
```

The above is the soft coded equivalent of the first example. Here, the format name is indirectly referenced through the string variable MYFMT$, with access to the desired element and occurrence being made through the LET FMT assignment using numeric variables instead of literal references. Note the prefixing of the MYFMT$ variable name with an octothorpe (#) in the FORMAT INCLUDE directive, which tells *Thoroughbred* that it is the format named in MYFMT$ that is to be processed (omitting the # will cause a syntax error in this case).

The hard coded style usually results in faster executing and more compact programs, often requires less work to develop, and allows formats and elements to be treated more or less like ordinary 3GL variables. When a hard coded program is SAVEd *Thoroughbred* will validate each format reference against the data dictionary and will issue error messages if inconsistencies are found (for example, omitting a FORMAT INCLUDE statement or misspelling a format or element name). Last but not least, hard coded programs tend to be self-documenting: the first of the above examples is probably easier to understand than the second.

On the down side, hard coded programs cannot be RUN unless they have been SAVEd, as *Thoroughbred* embeds information about hard coded formats and elements into the program's symbol table—which is why such programs tend to run faster than soft coded equivalents. This information does not exist or is unreliable until after a SAVE has been executed. Thus, you cannot throw together a "quicky" program with hard coded format references and just run it. Also, you cannot dynamically alter a hard coded program with EXECUTEs, DELETEs and MERGEs, as doing so will cause the program text to change and invalidate all hard coded format references. Certain types of operations are awkward with hard coded formats—we will illustrate just such an operation in a few paragraphs. Portability may be another issue: a program with hard coded formats cannot be run on a different system where formats are not identical in all respects. Lastly, hard coded format references and user-defined functions don't always coexist well in programs.

It's not uncommon for such a mixture to cause random memory faults (core dumps or segmentation faults) and other unexplainable errors, especially in older versions of *Thoroughbred*.

Soft coding overcomes many of the above limitations, at the expense of more complexity, more typing, poorer readability and slightly slower execution. However, a major advantage of the soft coded style is that such programs can be dynamically altered with `EXECUTE`s, `DELETE`s and `MERGE`s without causing errors, and `SAVE`ing a soft coded program is not necessary prior to execution. Plus soft coding allows you to pass a format name to a public program (as is done with many cookbook functions) and have the public module operate on the format's data without having advanced knowledge of the format's name or internal structure—this is a real boon for portability.

Soft coding facilitates a number of programming techniques that are clumsy to implement with hard coded references. Consider the following hard coded program fragment, which clears 36 months of sales data for six categories:

```
01000    FOR OCC=1 TO 36;
            #OPCMSBC.SALES1(OCC)=0,
            #OPCMSBC.SALES2(OCC)=0,
            #OPCMSBC.SALES3(OCC)=0,
            #OPCMSBC.SALES4(OCC)=0,
            #OPCMSBC.SALES5(OCC)=0,
            #OPCMSBC.SALES6(OCC)=0;
         NEXT OCC
```

Assuming that all six elements are contiguous in the format, the most efficient way to code this procedure, nested loops, isn't possible, as you cannot indirectly refer to an element in a hard coded format reference. Also, the element occurrence limit (36 in this example) would have to be known when the program is written.

Compare the above example to its soft coded counterpart:

```
01000    FOR ELM=FNELM(CMSBC$,"SALES1") to FNELM(CMSBC$,"SALES6");
            FOR OCC=1 TO FNOCC(CMSBC$,ELM);
               LET FMT(CMSBC$,ELM,OCC)="";
            NEXT OCC;
         NEXT ELM
```

In this example, it is assumed that `CMSBC$` refers to the `#OPCMSBC` format referenced in the first example. Note how this code, using functions that are described elsewhere in this cookbook, automatically determines the correct element numbers for the outer loop and the correct number of occurrences for the inner loop. This technique is not possible with hard coded references. Incidentally, storing a null string into a numeric element—the `LET FMT(CMSBC$,ELM,OCC)=""` assignment in the above example—is the equivalent of setting that element's value to zero.

Needless to say, we recommend you rely exclusively on soft coding to avoid portability problems and to gain access to many cookbook functions that would otherwise be inaccessible. Plus, it will be much easier to make code revisions when the need arises, as such revisions can be accomplished with the search and replace functions provided with the standard BASIC utilities.

FORMAT ACCESS WITH SOFT CODING

Thoroughbred BASIC includes a number of language primitives that facilitate the extraction from and storage into soft coded formats of data, as well as the retrieval of element attributes, which information can be used to tailor a program's behavior during runtime. This section will briefly describe each of these primitives. Our tutorial is meant to supplement the official descriptions in the *Thoroughbred* manuals, which should consulted as needed.

`FORMAT INCLUDE` This directive loads the attributes of a named format into memory and allocates storage for associated data. `FORMAT INCLUDE` is a prerequisite to any other format operation (however, see the subsection on portable database access on page 16 for an exception to this rule). The formal syntax is:

```
FORMAT INCLUDE #FORMAT{$}[,OPT="<option>"]
```

where `<option>` may be `INIT` (initialize all elements, see `FORMAT INIT` below), `DEFAULT` (initialize elements with default values, see `FORMAT DEFAULT` below) and `NONE` (do not initialize or default an already `INCLUDE`d format). If no `OPT=` clause is specified, `INIT` is assumed. Be sure to `INCLUDE` the `#IDSV` IDOL-IV system format before any others (see the discussion on the `FNDES$()` defined function on page 35 for more information).

`FORMAT DELETE` This directive removes a format from memory and releases storage that was allocated to it. The formal syntax is either:

```
FORMAT DELETE #FORMAT{$}
```
or
```
FORMAT DELETE ALL
```

which deletes all formats from memory—*use with caution*!

`FORMAT INIT` This directive initializes all elements, setting numeric and date elements to the equivalent of zero and strings to blanks. The formal syntax is either:

```
FORMAT INIT #FORMAT{$}
```
or
```
FORMAT INIT ALL
```

which initializes all INCLUDEd formats—again, use with caution.

FORMAT DEFAULT

This directive initializes an already INCLUDEd format and then loads the element with default values, if defined. The formal syntax is either:

```
FORMAT DEFAULT #FORMAT{$}[,OPT="DFONLY"]
```
or
```
FORMAT DEFAULT ALL[,OPT="DFONLY"]
```

the latter which defaults all loaded formats. The DFONLY option causes the initialization step to be skipped and defaults only those elements that are already in an initialized state, leaving elements containing valid data undisturbed. Elements for which no default values have been defined will be initialized only.

ATR

ATR is a string function that returns an attribute from a format and element. The formal syntax is ATR(FMT$,ELM,AN), where FMT$ refers to the format being acted upon, ELM is the element number from which you wish to extract an attribute and AN is a number indicating which attribute to extract. For example, N=NUM(ATR(MYFMT$,4,7)) returns the numeric value of the date indicator for the fourth element in the format referred to by MYFMT$ (it will be zero if the element is not defined as a date). The special case N=NUM(ATR(MYFMT$,0,0)) sets N equal to the number of elements in the format referred to by MYFMT$. Many of the defined functions described later on use ATR to provide useful information about formats and elements.

FMT

FMT is a string function that returns the data in an element, with an appropriate conversion based upon the element attributes. For example, D$=FMT(MYFMT$,4,2) will copy the contents of the second occurrence of the fourth element in the format referred to by MYFMT$ into the string variable D$, performing whatever transformations are necessary to produce a human-readable string.

For example, if the referenced element has been defined as a four byte SQL date, FMT will automatically convert the date value so as to produce a MM/DD/YY string.

For numeric and date elements, you can modify the form of the retrieved data with a masking operation: `D$=FMT(MYFMT$,4,2):"DNM"` uses the default mask for string formatting. Substituting `DCM` in place of `DNM` formats the string with commas (e.g., `12,345.67`). You may also use standard numeric masks where appropriate. For example, `D$=FMT(MYFMT$,4,2):"######.00#"` is an acceptable method of extracting string data from a numeric element.

`FMD`

`FMD` is a string function that returns the "raw" or literal data in an element or an entire format. For example, `D$=FMD(MYFMT$,4,2)` will copy the raw content of the second occurrence of the fourth element in the format referred to by `MYFMT$` into the string variable `D$`. A variable assignment such as `D$=FMD(MYFMT$)` will result in all the raw data in the format being copied to `D$`. This is a handy way to preserve the current contents of a format when it must be temporarily overwritten with new data.

It's important to understand that the data returned by `FMD` is in the exact form in which it is stored in the element. For example, using `FMD` to extract data from an element defined as an IEEE floating point number will result in the binary representation of an IEEE numeric being passed into a string variable, not the numeric equivalent. This characteristic of `FMD` leads to an important observation: if an element is a record key or part of one, it is mandatory you employ `FMD` to use that element's data to perform keyed random access. If you mistakenly use `FMT` it is quite possible your program will not work as expected:

```
READ (CH,KEY=FMD(MYFMT$,1)) #MYFMT$;  REM Okay

READ (CH,KEY=FMT(MYFMT$,1)) #MYFMT$;  REM Incorrect
```

See the `buildkey` function (page 70) for a means of generating a properly structured record key from a format.

`LET FMT`

`LET FMT` is the complement of `FMT`: it is used to store data into an element with appropriate conversion. For example, if the fourth element of the format referenced in `MYFMT$` is a four byte SQL date (date type `5`), the assignment expression `LET FMT(MYFMT$,4,0)="051600"` will store the date `May 16, 2000`, automatically converting it to the four byte SQL structure required by this element.

`LET FMD`

`LET FMD` is the complement of `FMD`: it is used to store raw data into an element.

However, unlike FMT, no conversion will occur with this directive. For example, the assignment expression LET FMD(MYFMT$,4,0)=$031725$ will store that exact binary value into the element. The expression LET FMD(MYFMT$)=D$ will load the contents of D$ into the data area of the format referenced by MYFMT$. *There is no validation by the* LET FMD *statement of the data to be stored*. This means that you can store anything into an element, regardless of whether the data is appropriate or not.

USING FORMATS WITH COOKBOOK FUNCTIONS

Now that some basics have been presented, let's move on to the topic of how to use some of this knowledge in 3GL programs. Here's a simple example with which to start: instead of coding something like this to read a record from a file:

```
00200   DIM D5[24]
00500   IOLIST A5$,B5$,C5$,D5$,A5,B5,C5,D5[ALL]
01000   READ (5,END=09000) IOL=00500;
```

you could do this:

```
00200   SETUP:  EMMAS$="#HREMMAS";
                FORMAT INCLUDE #EMMAS$
01000   MAIN:   READ (EMMAS,END=DONE) #EMMAS$
```

In the first example, string and numeric variables, as well as a numeric array, are structured into an IOLIST, which is used as the target for the data. It is up to the programmer to define each variable in the IOLIST, being careful to match string variables with string data, numeric variables with numeric data, and so forth. It is also necessary to code the IOLIST into each program that reads from that file. Neither the IOLIST or any of the variable names in it are particularly mnemonic, which increases the difficulty of recalling which variable is assigned to a given field in the record. Also, it is impossible to pass the IOLIST as a parameter to a public program, necessitating the use of the individual variables in the CALL statement.[3]

The second example illustrates the same operation using a format. Instead of an IOLIST, a format receives the record. As discussed in the previous section, the FORMAT INCLUDE directive handles the chore of preparing a place in memory in which to store the record.

---

[3]Many code examples will include line labels and will show branches to labels rather than lines. Tests have conclusively proved that programs in which labels are used to refer to branch, subroutine and IOLIST targets will execute more quickly than programs that refer to line numbers. Also, the mnemonic qualities of labels make it easier to keep track of what does what in a large, complex program.

A single string variable, initialized in this example with the name of a format called #HREMMAS, replaces the IOL= argument to the READ statement.[4] The internal structure of the format does not have to be known to the program in order to access the file and the format is independent of regular variables, so interference will not be a problem. Also, the code is more understandable: we know exactly which data structure is receiving the record. Best of all, the format can be accessed by a public program by passing the format name in a string variable as part of the CALL statement.

Let's expand on this last statement:

```
00200    SETUP:  EMMAS$="#HREMMAS",
                 EMMASD$= "#HREMMASD";
                 FORMAT INCLUDE #EMMAS$;
                 FORMAT INCLUDE #EMMASD$
01000    MAIN:   READ (EMMAS,END=DONE) #EMMAS$;
                 CALL "copyfmt",EMMAS$,EMMASD$,1
```

In this revised example, we read a record into the #HREMMAS format and then using the cookbook function copyfmt (copy format to format, page 84), we copy the entire data area of #HREMMAS into #HREMMASD, whose structure could be substantially different from that of #HREMMAS (the ,1 option to copyfmt tells it to perform a FORMAT INIT operation on #HREMMASD before copying the data). As you have undoubtably surmised, #HREMMAS is a physical format (it receives the record from the file) and #HREMMASD is a logical format, with an as-yet unspecified purpose (hint: the D at the end of the format name indicates in a ***BCS Technology Limited*** program that it is a "display format" used to paint the screen). The call to copyfmt illustrates the technique of portably passing format names to a public program for processing.

The above example with copyfmt also illustrates an instance where a cookbook function makes up for a limitation of the core language. A feature of *Thoroughbred* BASIC is the capability of copying the contents of an entire format to another without having to reference individual elements—conceptually like saying RECORD B = RECORD A. The exact syntax is:

```
#DESTFMT = #SRCFMT1 [& #SRCFMT2 [& #SRCFMT3...]]
```

where #DESTFMT is loaded with data from the "source" formats listed on the right hand side of the assignment expression. This capability allows the programmer to build up the data in #DESTFMT from the elements of other formats, even though the elements in the formats are in a different order.

---

[4]Note the use of the octothorpe (#) with the variable name as part of the READ statement, indicating that a format reference is required. Without this, *Thoroughbred* would attempt to load the record into EMMAS$, rather than into the format referred to by EMMAS$. Omission of # before the variable name in soft coded programs is a class of bug that can be difficult to track down because no syntax error is generated when the statement is compiled.

Unfortunately, the language syntax only provides for hard coded formats—coding something such as `#DFMT$ = #SRCFMT1$ & #SRCFMT2$` is syntactically incorrect and will cause a compile-time error (however, `LET FMD(DFMT$)=FMD(SRCFMT$)` would work if the two formats were structurally identical in all respects).

`copyfmt` circumvents this limitation with soft coded formats. By the way, you will find many instances where such copying is useful, for example, creating a sales order header from data in a customer master, consignee master, salesman master, etc. Copying also is the best means to transfer a record stored in a physical format to a logical format whose structure is designed to facilitate display of the record. This technique was illustrated above.

PORTABLE DATABASE ACCESS USING LINKS

Intrinsic to the Dictionary-IV environment is portability, obtainable, as you know, through the use of links. The cookbook provides several methods of accessing IDOL-IV objects in a portable fashion; see `openlink` (page 181) for an example. Also, starting with BASIC level 8.3, improved database access portability may be gained by `OPEN`ing a link instead of a file. The key to this operation is the use of an `OPT="LINK"` clause in an `OPEN` statement, which permits a link name to replace a filename. The following database maintenance examples will illustrate this feature. First, the "traditional" code:

```
00200   SETUP:  LINKNAM$="APVMMAS",   REM link to be accessed
                SETUP=TCB(4)+1;
                CALL "statlink",LINKNAM$,FILENAM$,FT$;
                FORMAT INCLUDE #FT$;
                CALL "elements",FT$,"FT$","E",SETUP
00210   SETUP1: EXECUTE "DELETE "+STR(SETUP);
                CH=UNT;
                OPEN (CH)FILENAM$;
                READ (CH,SRT=SORTNAM$,KEY="",DOM=MAIN)
01000   MAIN:   EXTRACT (CH,END=DONE) #FT$;
                …process as required…
                WRITE (CH) #FT$;
                GOTO MAIN
09000   DONE:   CLOSE (CH);
                FORMAT DELETE #FT$;
                END
```

The above code starts by making a call to `statlink` (page 239) to obtain the format and file names associated with the `APVMMAS` link. Next, the program `INCLUDE`s the format, calls `elements` (page 99) to generate numeric element references and `OPEN`s the file. Each `EXTRACT` and `WRITE` operation requires the specification of the format name. The sequence finishes by closing the file and deleting the format.

Now, here is the same program modified to take advantage of the OPT=LINK feature:

```
00200    SETUP:   LINKNAM$="APVMMAS",
                  SETUP=TCB(4)+1;
                  CALL "linkfmt",LINKNAM$,FT$;
                  CH=UNT;
                  OPEN (CH,OPT="LINK")LINKNAM$;
                  CALL "elements",FT$,"FT$","E",SETUP
00210    SETUP1:  EXECUTE "DELETE "+STR(SETUP);
                  READ (CH,SRT=SORTNAM$,KEY="",DOM=MAIN)
01000    MAIN:    EXTRACT (CH,END=DONE);
                  …process as required…
                  WRITE (CH);
                  GOTO MAIN
09000    DONE:    CLOSE (CH);
                  END
```

The second example replaces the call to statlink with a call to linkfmt (page 155) to get the format name and then OPENs a channel to the *link name*—not a filename. The presence of the OPT="LINK" clause will cause the OPEN statement to automatically determine the name of the file to be OPENed and automatically INCLUDE the format associated with the link.[5] There is no filename reference anywhere to be found and no FORMAT INCLUDE statement—the OPEN statement conveniently handled that step for us. In fact, we wouldn't even know the format name if it weren't for the call to linkfmt! Despite the fact that the program did not perform a FORMAT INCLUDE operation, the call to elements will work as expected.

What's particularly neat about this technique is this: if the point in the program at which the actual processing occurs is a CALL to a public program, no CALL to elements is needed and in fact, the public program can be made figure out the format name on its own with linkfmt—suffering only a small performance penalty in the process. The amount of code in the body of the main program would be minimal and by merely substituting the correct name for the public program, 100 percent portable.

Expanding on this, suppose each element in the format had a program name in the Pre-Process attribute associated with each element. Your maintenance program would call a "supervisor" public program to which you would pass the link name. The supervisor would determine the format name with linkfmt, and then, using a FOR-NEXT or WHILE-WEND loop, "walk" through the entire format, extract the Pre-Process program name from each element and CALL that program to operate on that element

---

[5]The INCLUDE operation performed by OPT="LINK" is called a "soft" INCLUDE. A soft INCLUDEd format does not appear in the resident format list maintained in the FMTNL system variable and cannot be deleted from memory with a FORMAT DELETE #FMT$ directive. Therefore, it will survive a call to the clrfmts function (page 79), even without a format list. Be aware that no format will survive a FORMAT DELETE ALL directive.

Input/output (I/O) operations on a file OPENed through an OPT=LINK procedure behave somewhat differently than with a conventional OPEN. To perform operations on the file, you use the standard EXTRACT, READ or WRITE statements as you normally would, but without specifying a format name. For example:

```
READ (CH)
```

This operation, although it appears to do nothing, will load the format associated with the link OPENed on channel CH with a record, even though you did not actually specify a format. You could also perform:

```
READ (CH) #FMT$
```

which will act exactly like the first example.

Operations such as:

```
READ (CH) IOL=IOLISTNAME
```

or

```
READ (CH) A1$,B1$,C1
```

are not allowed and will cause ERR=172 to occur.

Finally, it is essential that your program explicitly CLOSE any channel OPENed to a link when all operations on the associated database have been completed. The CLOSE (CH) operation, in addition to closing the associated file, causes *Thoroughbred* to perform a FORMAT DELETE on the associated format. Do not attempt to perform a FORMAT DELETE on the associated format unless your program explicitly INCLUDEd it *prior* to OPENing the link. Above all, *never use* CLOSE (0) *if your program has* OPENed *channels to links*. Such channels will not be properly closed and the associated formats will not be deleted from memory, nor can they be deleted by any means except FORMAT DELETE ALL. The cookbook's closeall function (page 78) will properly close OPENed links.

PERFORMANCE CONSIDERATIONS

The introduction of high level features into any programming language will negatively affect performance in some way. For example, a *Thoroughbred* BASIC 3GL program, being interpreted "byte code," will run more slowly on a given system than an equivalent written in ANSI C, whose executable binaries are machine code. Any C program will run more slowly than an equivalent written in assembly language, as the latter can benefit from manual optimizations that a C compiler cannot produce.

Going the other direction, the use of 4GL features within your Thoroughbred 3GL programs will incur some performance penalties, which individually may be negligible on modern hardware. However, the effects of scale must be considered in programs where a lot of 4GL features are employed, especially when many cookbook functions are `CALL`ed. For example, a `CALL` to a public program may have little apparent performance impact, unless it is executed thousands of times in an end-to-end database search—the `CALL` itself requires internal stack manipulations that may consume hundreds or thousands of clock cycles to complete.

Thoroughbred, like any other execution environment, performs no better than the underlying operating system and hardware. Therefore, if you wish to improve the performance of your programs you need to possess some understanding of system performance issues. In evaluating system performance it is helpful to consider processing as either compute-bound or IO-bound, and as being "cheap" or "expensive," where cheap and expensive describe to what extent system resources are required to accomplish a task.

A compute-bound process executes strictly within memory and performance is determined by the efficiency of the code, as well as the combined capabilities of the microprocessor and random access memory subsystem (which includes the chipset hardware and processor bus). Most compute-bound processing is cheap, as it involves only the processor/memory subsystem. The actual "cost" of a process will be increased if operating system services are requested in order to complete the task at hand, as an operating system subroutine call and return must be executed—the required stack manipulations add substantial overhead.

When IO-bound, a program will have initiated communication with a device that is not directly part of the processor/memory subsystem, but is instead attached to the system through an interface bus and controller. The combination of device, bus and controller will generally run at a fraction of the speed of the processor/memory subsystem—often asynchronously, causing IO-bound performance to greatly lag compute-bound performance. Therefore, IO-bound processing of any kind is inherently expensive and the expense significantly increases when mechanical devices like disk and tape drives are involved. For example, random access memory performance is measured in nanoseconds (10^{-9} seconds), whereas access times for the fastest hard drives are measured in milliseconds (10^{-3} seconds)—a potential difference of one million to one in raw performance.

In most systems, compute-bound performance may be improved by using a higher performing processor, faster memory, replacing the motherboard with one using a more efficient bus and chipset design, or some combination of these remedies. Compute-bound processing cannot be affected in any way by devices external to the processor/memory subsystem. For example, installing a faster disk drive may appear to improve the system's performance, but only if viewed in the context of an IO-bound process that is reading from or writing to a file located on that particular drive.

In UNIX-like operating systems, compute-bound processes run in either user mode or kernel mode. A process is in user mode unless it requests a service from the operating system, such as access to more memory or the current date and time of day. Certain types of kernel mode processing are inherently expensive, even though entirely compute-bound. A request for more memory is one such process, as it involves the manipulation of complex memory allocation data structures. Also complicating the processing picture is preemptive multitasking, which causes competition between any number of processes for run time and machine resources.

An IO-bound process will always run in kernel mode and may be forced to sleep if the device being accessed is in use by some other process or is temporarily busy with offline functions. Ironically, sleep itself costs nothing (the process uses no resources while sleeping), an effect which tends to favor IO-bound processes that are not sleeping. Of course, this is little solace to the user impatiently waiting for results.

Strictly speaking, an executing *Thoroughbred* BASIC program runs as a compute-bound process and in the UNIX environment, always runs in user mode. *Thoroughbred* itself will switch to kernel mode as required and, of course, will become IO-bound any time a directive such as `OPEN`, `PRINT` or `READ` is executed. Knowing this, it is possible to examine a slow-running program and determine where improvements to the code might be made. Obviously, anything that is done to reduce IO-bound operation will have the greatest effect. For example:

- Minimize output operations to the display. Writing to a display can be expensive, especially if the operating system does not allocate a lot of buffer space for terminal I/O and data rates are relatively low (under 19,200 bits per second or BPS on a serial interface). At low data rates, it is possible for the output buffer maintained by the kernel to fill, forcing your program to sleep until the buffer empties. At higher rates, this is less an issue, as the buffer can be emptied rapidly. However, many repetitive writes to a terminal can fill the buffer no matter how fast the data rate, as the terminal itself can only process incoming data so fast. Once its internal buffer has filled the terminal will signal the operating system to stop sending more data until it can catch up. During this time your program will sleep.

  BASIC programs that were written for older, slower hardware and that performed sequential file updates often `PRINT`ed to the screen as each record was processed, which reassured the user that the update was running. On modern hardware, the cost of updating the screen in this fashion will be a major part of the cost of running the update itself. Therefore, programs of this type should be modified so that the screen is updated at periodic intervals, say, every 500 records (even that might be too often).

  You also can avoid display-induced bottlenecks by arranging your code to paint the screen from left to right and top to bottom. Once the cursor has been positioned on any given screen row, it is possible to print at any column with just a column coordinate (`PRINT @C`) rather than `PRINT @(C,R)`).

In most cases, doing so will reduce the amount of raw data that must be sent to the display. Also, try to structure statements so as to reduce the number of PRINT directives: a statement such as PRINT @(C,R),A$,@(C+5),B$,@(C,R+1),C$ is more economical to process than PRINT @(C,R),A$,;PRINT @(C+5,R),B$,;PRINT @(C,R+1),C$. Lastly, take advantage of Thoroughbred's windowing capabilities to manage the display.

- Open all files at the beginning of your program. This suggestion may seem obvious to experienced programmers. Nevertheless, a lot of programs written to run in the old MAI *Basic Four* environment constantly OPENed and CLOSEd files. This was necessary with old versions of Business Basic (BB), where a program had a very limited number of channels available for I/O and often had to CLOSE one so another could be OPENed. However, all modern operating systems allow a task to simultaneously open many files, so there is no good reason to retain the old BB methods. OPENing a file is very costly, as multiple disk accesses are required to search directories, locate the file descriptor (inode) and load it into a buffer.

- Aggregate programs into object libraries. RUNing a program from an object library incurs a lot less IO-bound expense than running it from a discrete disk file. This subject is discussed in greater detail starting on page 25.

- **ADDR** frequently used public programs. Any time your program CALLs a public module *Thoroughbred* must get a copy of that program's text from disk and load it into memory, an operation that is as costly as OPENing a data file. If you ADDR a frequently used public program loading it from disk is a one-time expense. Observance of this suggestion will produce a major performance improvement in any program where the same CALL statement is repetitively executed.

- Use multikeyed file types where possible. Another programming technique inherited from the old *Basic Four* days is the use of one or more secondary sort files for cross-indexing. Prior to the development of BB86, this was the only way to maintain secondary sorts in the *Basic Four* environment. Even after BB86 multikeyed files became available, programmers continued to use secondary sort files due to the work involved in modifying existing code. Yet, a substantial performance increase would have been realized had multikeyed files been adopted.

  The obvious problem with using secondary sort files is that multiple WRITE operations are required to maintain the sorts, resulting in disk-intensive I/O operations. The multiple READ operations required are almost as expensive. The net effect is that numerous IO-bound conditions are created, which on a busy system results in significantly degraded performance for both the program maintaining the secondary sort and other programs that must compete for run time. Also, sort consistency is vulnerable in the event a program error occurs, as a WRITE operation may not be completed due to the error.

The solution is to use MSORT or TISAM files when multiple sorts are required (TISAM files are recommended due to their ability to expand as records are added). Since *Thoroughbred* itself maintains the secondary sorts (rather than the BASIC program) the likelihood of an error causing damage to the consistency of the file will be small. Plus with only one WRITE operation required to update the file, IO-bound processing will be substantially reduced, benefitting the entire system.

USING OBJECT LIBRARIES

One of the most useful and underused environment features available to the *Thoroughbred* BASIC programmer is the object library facility. Several compelling reasons exist for employing object libraries in your large scale development projects. Therefore, we will present some technical information on how object libraries work and why they are a useful resource.

OBJECT LIBRARY STRUCTURE and OPERATION

A *Thoroughbred* object library is a collection of programs stored in one file, complete with a built-in indexing system for rapid access to the library's contents. The object library concept is simple to understand: instead of loading *Thoroughbred* BASIC programs from individual disk files, load them from a library data structure, much as is done with dynamic link libraries (DLL's) in the Microsoft Windows environment. Should the need arise to release an updated distribution of programs, distribute object libraries instead of individual program files.

Internally, an object library file consists of three logical segments: header, program text and table of contents. The header—which occupies the first 64 bytes in the file—contains the administrative data needed by *Thoroughbred* to access the rest of the library. Following the header is the text segment, which contains the compiled BASIC statements of the original programs from which the library was generated. The text segment is followed by the alphanumerically sorted table of contents, which is arranged as a contiguous series of equally sized "slots."

The important header structure values are as follows (all values are in hexadecimal and offsets are zero-based relative to the start of the file):

`00-03`	Physical size of the object library file in bytes, a 32 bit integer whose value should agree with the file size reported in a directory listing.
`10-13`	Optional 32 bit cyclic redundancy checksum (CRC). Generation and verification of a CRC is discussed in greater detail on page 29.
`23`	Table of contents slot size, 8 bit unsigned quantity. By default this value is `0F` (decimal `15`).
`24-29`	Date and time of last update, stored as a 6 byte binary SQL value (this value may be decoded with the `FNLBDT` defined function—see page 41).
`2A-31`	Library's filename (padded with trailing nulls or blanks).[6]
`32-33`	Constant value of `5441`, a *Thoroughbred* "magic number" that identifies the file as an object library.
`34-35`	Maximum length of the filenames stored in the table of contents, 16 bit unsigned quantity. By default this value is `0008`.
`36-39`	Offset to the start of the table of contents, a 32 bit quantity.

[6] If the need arises to rename an object file it should be accomplished with the `RENAME` directive in BASIC, not with operating system commands.

3A-3B	Length in bytes of the table of contents, a 16 bit quantity.
40	Start of the text segment.

Assuming an eight character maximum filename length, a table of contents slot has the following structure:

00-07	Program filename padded with trailing nulls or blanks.
08-0B	Program starting offset in text segment, a 32 bit quantity.
0C-0E	Number of bytes of program text, a 24 bit quantity.

This pattern repeats for each program in the library.

As can be seen from the above information, given a maximum filename length of eight characters, each slot will occupy 15 bytes in the table of contents. From this, we can determine the number of programs in the library by dividing the length of the table of contents by the length of a slot. Since the table of contents length is expressed as an unsigned 16 bit quantity (maximum value of 65,535), a maximum of 4369 programs may be stored in a single library if eight character filenames are used. In practice, this won't be a limiting factor, as few distributions have such a large number of modules, nor would it be wise to aggregate an entire distribution into one library.

When an object library is properly opened, *Thoroughbred* will load the header and table of contents into memory. This process will be repeated for each object library thus opened. Assuming at least one object library has been properly opened, a request to load or run a program will initially be serviced by performing a binary search for the requested filename on the memory-resident table(s) of contents, said search being executed on each loaded table of contents until the requested program has been located. If the search fails *Thoroughbred* will revert to the normal load from disk method, in which a sequential search of each logical disk will take place. On the other hand, if the search is successful, the offset and size data associated with the filename entry will be used to page the program text directly from the disk-resident library into program execution memory, thus avoiding the expense of a disk directory search.

Much of the efficiency achieved with the use of an object library stems from the table of contents search. A binary search of an ordered list with fixed length elements will require no more than $INT(\log_2 N)+1$ comparisons, where N is the number of elements in the list. For example, if the list contains 1000 elements, the worst case search, no match, will require ten comparisons. Even at the theoretical maximum library size (4369 files), a worst case search will require no more than 13 comparisons. Plus, the search will be entirely compute-bound, with no time-consuming operating system calls.

In contrast, a typical disk-based file search will require the repeated execution of a series of kernel calls and disk I/O operations to sequentially examine the target directory(s) until either the requested filename entry has been found or the list of names has been exhausted.

Also, a directory filename search in most operating systems is linear, potentially resulting in N name comparison iterations. Adding to the workload is the fact that the operating system may be forced to read the directory one disk block at a time, since most operating systems store files in block random style, with no assurance that logically sequential data blocks in any given file will be physically contiguous on the disk. Given a large directory tree, numerous disk accesses may occur unless the requested filename is near the start of the directory.

Experience has shown that on most systems loading programs from an object library is as much as 100 times faster than loading from discrete files. Often, use of the object library facility can give users the perception of the system being faster than it really is, possibly forestalling performance complaints on older hardware.

BUILDING AND USING OBJECT LIBRARIES

So far, we've presented the technical aspects of object libraries but we really haven't given you any compelling reasons (other than performance) to use them. Here are a few:

- You can assure that all programs associated with a distribution are in fact distributed as a unit, thus minimizing the potential for errors due to missing modules or revision mismatches.
- You will be able reduce the number and size of the disk directories associated with your package, as well as the overhead associated with searching the directories.
- You will be able to reduce the need for the ADDR/DROP mechanism often used to improve performance in distributions with many frequently used public programs.
- You will be able to exercise some degree of control over end-user modifications of your software by verifying the contents of each library when the task is started.

Object libraries may be conveniently built by running the *RPSD program in the *Thoroughbred* BASIC utilities menu. With *RPSD functions you may: display an object library's table of contents; add and delete programs (you can delete programs while viewing the table of contents); compress a library[7]; and perform an integrity check of both the library structure and the programs within. *RPSD is interactive and easy to use, as it is integrated with the IDOL-IV package. Needless to say, IDOL-IV must be fully installed on your development system and the terminal must be under the control of the *Thoroughbred* windowing system.

[7] An object library should be compressed any time a large number of programs are added or removed, as unused file space is often created by such manipulation. Compressing a library packs the programs together and reduces the overall size of the file, usually resulting in faster access.

There are a few precautions that should be observed when you build and/or update object libraries:

- Never modify any object library that has been opened by any task, including your own. *Thoroughbred* reads the header and table of contents only at the time the library is `OPEN`ed and then assumes the library structure will remain static as long as it remains `OPEN`ed. If programs are added or removed from the library while a task has it `OPEN`ed, that task's understanding of the library's structure will no longer be correct, a condition that will surely result in all sorts of strange errors when a subsequent attempt is made to `ADDR`, `LOAD` or `RUN` a program from that library.

- Do not aggregate your entire distribution into a single object library. Aside from the obvious risk of losing the entire distribution if the object library is deleted or corrupted, the resulting large file size will have an adverse effect on raw disk performance. As a file grows, more disk activity is required to position to a given offset within the file's logical boundaries, activity which is in addition to that required to actually read the contents. Therefore, best performance is attained by limiting the number of programs per object library and compressing libraries to eliminate slack space.

- Arrange your libraries so related programs are stored together. For example, store all your accounts payable programs in one library (you could name it `ap.lib`), your sales order processing programs in another (named, perhaps, `op.lib`), your inventory control modules in a third (`ic.lib`) and so forth. Using this type of organization will facilitate the distribution of software upgrades that affect only a portion of the entire package. It will also simplify the addition or subtraction of features to your system.

- Set your object libraries' access permissions to read-only. Doing so will discourage tampering or accidental deletion.

VERIFYING OBJECT LIBRARY INTEGRITY

In any large software distribution the issue of package integrity will sooner or later arise. In the *Thoroughbred* environment there are three integrity issues to consider: file corruption, missing modules and undocumented modifications. Whether your distribution is via discrete modules or object libraries, a missing or corrupted program will be detected by a `LOAD`, `RUN` or `CALL` directive (`ERR=12` or `ERR=19`). Unfortunately, there is no easy way with discrete modules to detect undocumented modifications. However, if your distribution is in object libraries it is possible to verify that each library is intact by utilizing the services of several utilities included on the cookbook distribution disk.

Included with the cookbook distribution are the utilities `cksumlib` and `cklibcrc`. On any UNIX or Linux system on which a POSIX-compliant version of `cksum` has been installed, these utilities may be used to generated a cyclic redundancy checksum (CRC) for each of your object libraries and verify each library's CRC prior to use. To do this requires two steps:

1) Generate a CRC for each object library by `RUN`ing `cksumlib`. This should be performed only after all library maintenance operations within the `*RPSD` utility have been completed. `cksumlib` will request the filename of the library, which cannot be in use by any other task on the system. Once `cksumlib` has verified that the file is indeed an object library, it will generate a 32 bit CRC that may be subsequently validated with the `cklibcrc` utility. Be sure to run `cksumlib` each time an object library is changed in any way.

   CAUTION: *Do not include* `cksumlib` *with your distributions*.

2) At runtime, verify the integrity of each object library with `cklibcrc` (see the description of `cklibcrc` on page 76). `cklibcrc` is a public program that will compute a CRC for the library and compare it to the CRC that was generated by `cksumlib`. If an anomaly is detected, `cklibcrc` will condition the `ERR` system variable to indicate the nature of the problem. Changes to the library, such as removal, renaming or modification of a program, will be detected. The object library being checked should not be `OPEN`ed by the task that is performing the verify operation.

In addition to verifying an object library's CRC, `cklibcrc` is able to report the date and time a library was generated, with ± 5 second resolution. You can use this feature to prevent operation of your software if an out-of-date library is detected at startup time. See page 76 for more information.

COOKBOOK DEFINED FUNCTIONS

As an aid to developing more powerful and robust programs, we present a set of user-defined functions to complement the called functions of the *THOROUGHBRED PROGRAMMING COOKBOOK*. These functions were developed to ease the chore of extracting and manipulating data in 4GL data formats from within 3GL programs, as well as to simplify common string and numeric derivations, such as date and time, document age, area code/phone number formatting and even compressed binary ZIP codes (a few bytes here and a few bytes there…).

The entire set of defined functions is stored in an IDOL-IV text document named `DMDEF_FN`, the source for which is provided on the cookbook release disk (see the `IDDBD.dfn` file). Once the source file has been imported to IDOL-IV it is possible to merge the function statements from `DMDEF_FN` into any program from within the `EDITF` full screen editor. Prior to merging `DMDEF_FN` you can edit out any functions your program will not require and alter line numbers as required. To avoid interference with other 3GL variables the dummy variable names used in these functions begin with the sequence `DV_`. Use caution when nesting functions to avoid errors due to dummy variable name collisions.

FNAC$(AC) **GENERATE AREA CODE STRING**

This function creates a string version of area code `AC` if it is non-zero or a string of three blanks if zero. `0<=AC<1000`.

Example: `PRINT FNAC$(17)`
…output is `017`

FNACPH$(AC,PH) **GENERATE AREA CODE & PHONE NUMBER**

This function creates a string version of area code `AC` and phone number `PH` in the format `(AAA) NNN-NNNN`.

Examples: `PRINT FNACPH$(815,5551234)`
…output is `(815) 555-1234`
`PRINT FNACPH$(0,0)`
…output is "`(  )  `"

FNACPHL$(AC,PH) **GENERATE AREA CODE & PHONE NUMBER**

This function creates a string version of area code `AC` and phone number `PH` in the format `AAA-NNN-NNNN` or a string of 12 blanks if both the area code and phone number are zero.

Examples: `PRINT FNACPHL$(815,5551234)`
…output is `815-555-1234.`
`PRINT FNACPH$(0,0)`
…output is “ ” (12 blanks)

FND$(AMOUNT) **DISPLAY VARIABLE-SIZED DOLLAR AMOUNT**

This function creates a string version of `AMOUNT` in which the fractional component is always two places and the integer component contains only significant digits.

Examples: `PRINT FND$(2)`
…output is `2.00.`
`PRINT FND$(0)`
…output is `.00.`
`PRINT FND$(-4.299)`
…output is `-4.30.`

FNDAGE(D1,D2,MA) **COMPUTE AGING INDEX**

This function determines the number of days that have elapsed from the SQL (`DTN`) numeric date `D1` (e.g., a document generation date) to the SQL date `D2`, assumed to be in the future relative to `D1` (i.e., `D2>=D1`), with the computed number of days capped by `MA` (i.e., the result will never exceed the value passed in `MA`). Typical uses for this function are aging receivables and open purchase orders, and computing accounts payable cash requirements with respect to time. *Arguments must be integers or else* `ERR=41` *will occur*. A negative date relationship (i.e., `D1>D2`) will produce a zero result.

Examples:
```
D1=DTN(“06301998”,“MMDDYYYY”),
D2=DTN(“07291998”,“MMDDYYYY”),
MA=30,
AGE=FNDAGE(D1,D2,MA)
```

The above example results in `AGE=29`.

```
D1=DTN(“06301998”,“MMDDYYYY”),
D2=DTN(“07301998”,“MMDDYYYY”),
MA=30,
AGE=FNDAGE(D1,D2,MA)
```

The above example results in `AGE=30`.

```
D1=DTN("06301998","MMDDYYYY"),
D2=DTN("08011998","MMDDYYYY"),
MA=30,
AGE=FNDAGE(D1,D2,MA)
```

The above example also results in AGE=30, with the result having been capped at 30 even though 31 days have elapsed from D1 to D2.

```
D1=DTN("06301998","MMDDYYYY"),
D2=DTN("06291998","MMDDYYYY"),
MA=30,
AGE=FNDAGE(D1,D2,MA)
```

The above example results in AGE=0.

FNDAT$(FMT$,ELM,OCC) GENERATE MM/DD/YY DATE OR BLANKS FROM FORMAT

This function takes the four or six byte binary SQL date value in occurrence OCC of element ELM in format FMT$ and produces a MM/DD/YY style string if DATE is non-zero. The leading zero of the MM component is replaced with a blank for the months January through September. If the value in the element evaluates to zero a blank field eight characters in length is produced. Only elements whose date type is 5 (binary SQL) should be passed to this function.

Examples: `PRINT FNDAT$(FMT$,ELM,OCC)`

If the value in occurrence OCC of element ELM in format FMT$ is $00008D3B$ (729752) and ELM is a four byte element, this example will print 12/27/98.

See also the FNDT$ and FNLDAT$ functions.

FNDATE(FMT$,ELM,OCC) CONVERT BINARY SQL DATE TO FLOATING NUMERIC FORM

This function converts the four or six byte binary SQL date stored in occurrence OCC of element ELM in format FMT$ into a standard date/time numeric (DTN) floating point variable. The occurrence value should be zero for elements defined with only one occurrence (a literal zero may be used). Four byte fields store dates only, whereas six byte fields store both dates and times to ±5 second resolution. PRECISION 4 is required to convert a six byte binary date to its full fractional content. The expression evaluates to zero if no date has been stored in the field.

Examples: `DATE=FNDATE("#MYFORMAT",8,0)`

See also the FNIDATE function for the integer version of this function.

FNDATE$(DATE,FMT$,ELM) CONVERT NUMERIC DATE TO BINARY SQL FORM

This function converts the date/time numeric value DATE (such as derived from the DTN function) into a four or six byte binary SQL equivalent suitable for direct storage into a data format element defined as a binary SQL date field. The binary format is determined by the attributes of element ELM of data format FMT$. Four byte elements store dates only, whereas six byte elements can store both dates and times to ±5 second resolution. PRECISION 4 is required to convert to a six byte binary date.

Examples:

```
FMT$="#SSORDHDR",
ELM=8;
PRECISION 4;
LET FMD(FMT$,ELM)=FNDATE$(CDN+10,FMT$,ELM)
```

In the above example, the binary equivalent of today's date plus 10 days and the current time is generated and stored into the 8th element of the #SSORDHDR format. The correct style of binary date is determined from the element's attributes. *Note the use of the* LET FMD *directive to store the literal string generated by this function*.

```
FMT$="#ARDOCHDR",
AR_DOCDATE=10,
AR_DUEDATE=11,
DUE_DATE=CDN+30;
PRECISION 4;
LET FMD(FMT$,AR_DOCDATE)=FNDATE$(CDN,FMT$,AR_DOCDATE);
LET FMD(FMT$,AR_DUEDATE)=FNDATE$(DUE_DATE,FMT$,AR_DUEDATE)
```

In the above example, an accounts receivable document (such as an invoice) has the field AR_DOCDATE (the generation date and time) set to the system CDN date/time value and the field AR_DUEDATE (the payment due date) set to the generation date plus 30 days. As with the first example, the correct style of binary date is determined from each element's attributes. Assuming that the due date is being stored as an integer—typical in most applications—an automatic conversion from the floating point result of the CDN+30 expression to the long integer format of the AR_DUEDATE field will occur.

FNDAY(DATE) GENERATE DAY OF THE MONTH INDEX

This function takes the SQL (DTN) date number in DATE and produces a day of the month index in the range 1 through 31 inclusive.

If the value passed in DATE is zero the day index for today's date as determined by the CDN system variable will be returned.

Example:
```
DATE=729781;
PRINT FNDAY(DATE)
```

This example prints 25.

FNDES$(FMT$,ELM) EXTRACT DATA ELEMENT DESCRIPTION

During the definition of 4GL data formats the programmer can assign spoken language descriptions to each element, such as Customer Name or Transaction Amount (these descriptions are not the same as the element names—see FNELM$ below). This function returns the description assigned to element ELM of data format FMT$ with trailing blanks stripped. A special case occurs when ELM=0, in which case the unstripped descriptions for all elements are returned, 20 characters per description.

Prior to using this function the IDOL-IV system format #IDSV must be INCLUDEd and that INCLUDE operation must precede any other INCLUDE operation or FNDES$ reference (the inclusion of #IDSV is required to set the default language, usually English).

Example:
```
FMT$="#SSORDHDR",
ELM=14;
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
D$=FNDES$(FMT$,ELM);
PRINT D$
```

The above example will print Order Entry Date. The string assigned to D$ is stripped of trailing blanks.

```
FMT$="#SSORDHDR";
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
FOR ELM=1 TO FNFLD(FMT$);
  PRINT FNDES$(FMT$,ELM);
NEXT ELM
```

The above example prints the description for every element in the #SSORDHDR format. The FNFLD function returns the number of defined elements in a format (see below).

```
FMT$="#SSORDHDR";
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
D$=FNDES$(FMT$,0)
```

This special case causes D$ to be loaded with the descriptions for every element in #SSORDHDR. The length of D$ will be the number of elements in #SSORDHDR multiplied by 20 (the unstripped length of a description).

```
FMT$="#SSORDHDR",
ELM=14;
FORMAT INCLUDE #FMT$,OPT="NONE";
FORMAT INCLUDE #IDSV,OPT="NONE";
PRINT FNDES$(FMT$,ELM)
```

The above example will not produce any output, as the #SSORDHDR format was INCLUDEd before #IDSV.

FNDN$(DIR) **GENERATE LOGICAL DISK NAME**

This function takes the logical directory number passed in DIR and generates the equivalent logical disk name. The result is a two character string in the range D0 to DZ inclusive. The directory number passed in DIR should be an integer in the range 0 through 35 inclusive.

Example: `PRINT FNDN$(10)`

The above example will print DA.

FNDOW(DATE) **GENERATE DAY OF THE WEEK INDEX**

This function takes the SQL (DTN) date number in DATE and produces a day of the week index in the range 0 through 6 inclusive, with 0 being equivalent to Sunday. If the value passed in DATE is zero the day index for today's date as determined by the CDN system variable will be returned.

Example:
```
DATE=729781;
PRINT FNDOW(DATE)
```

This example prints 1 (Monday).

FNDT$(DATE) **GENERATE MM/DD/YY DATE OR BLANKS**

This function takes the SQL (DTN) date number in DATE and produces a MM/DD/YY style string if DATE is non-zero. The leading zero of the MM component is replaced with a blank for the months January through September. If DATE is zero a blank field eight characters in length is produced.

Examples:

```
PRINT FNDT$(729762)
```

This example prints " 1/06/98".

```
PRINT FNDT$(729752)
```

The above example prints "12/27/98".

```
PRINT FNDT$(0)
```

This example prints " " (8 blanks).

See also the FNLDT$ function.

FNELM(FMT$,ELM$) **CONVERT ELEMENT NAME TO NUMBER**

This function returns the element number of element name ELM$ in format FMT$ or zero if the element name has not been defined or is null. The element name is not case-sensitive.

Examples:

```
FMT$="#SSORDHDR",
ELM$="OE_DATE";
FORMAT INCLUDE #FMT$;
ELMNUM=FNELM(FMT$,ELM$)
```

The above example assigns the element number of the element OE_DATE in the format #SSORDHDR.

```
FMT$="#SSORDHDR",
ELM$="oe_date";
FORMAT INCLUDE #FMT$;
ELMNUM=FNELM(FMT$,ELM$)
```

This example produces the same result as the previous one, as element names are not case-sensitive to this function (element names should be entered in upper case during format definition).

FNELM$(FMT$,ELM) CONVERT ELEMENT NUMBER TO NAME

This function returns the element name of element number ELM in format FMT$, with trailing blanks stripped. A special case occurs when ELM=0, in which case the unstripped element names for all elements in the format are returned, 20 characters per element name.

Examples:

```
FMT$="#SSORDHDR",
ELM=14;
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
E$=FNELM$(FMT$,ELM)
```

The previous example will return the name of the 14th element in the #SSORDHDR format. E$ is stripped of trailing blanks.

```
FMT$="#SSORDHDR";
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
FOR ELM=1 TO FNFLD(FMT$);
  PRINT FNELM$(FMT$,ELM);
NEXT ELM
```

The above example will print the element name for every element in the #SSORDHDR format. The FNFLD function returns the number of defined elements in a format (see below).

```
FMT$="#SSORDHDR";
FORMAT INCLUDE #IDSV,OPT="DEFAULT";
FORMAT INCLUDE #FMT$,OPT="NONE";
E$=FNELM$(FMT$,0)
```

The above code causes E$ to be loaded with the names for every element in #SSORDHDR. The length of E$ will be the number of elements in #SSORDHDR multiplied by 20 (the unstripped length of an element name).

FNEPOS(FMT$,ELM) FIND ELEMENT OFFSET IN FORMAT

This function returns the one-based byte offset of element ELM in format FMT$. The result may be used in combination with the FNSIZ function (below) to extract element data from a string variable into which a format's data area has been copied with the FMD string function. FNEPOS is also useful when using READ RECORD to extract data into a string variable where a particular portion of the record must be modified. ERR=17 will occur if ELM is zero. ERR=163 will occur if the element specified in ELM is greater than the total number of elements in the format.

Example:
```
FMT$="#APVMMAS",
FMTDATA$=FMD(FMT$),
ELMDATA$=FMTDATA$(FNEPOS(FMT$,4),FNSIZ(FMT$,4))
```

The above code copies the data area of the format #APVMMAS to FMTDATA$ and extracts the data from the fourth element of #APVMMAS into ELMDATA$.

See also FNSIZ.

FNEPR(RATE,NPER) COMPUTE EXTENDED PERCENTAGE RATE

This function computes an extended percentage rate (EPR), where RATE is the periodic rate to be charged, expressed as a decimal fraction (e.g., .015 would represent 1.5 percent), and NPER is the elapsed time in periods over which the EPR is to be computed. If NPER=0 this function will assume NPER=12.

Examples:
```
EPR=FNEPR(.015,36)
```

The result from the above example would be 70.91, assuming PRECISION 2.

```
APR=FNEPR(.015,0)
```

The result from the above example would be 19.56, again assuming PRECISION 2. Since the NPER value was zero, the function substituted 12 for the number of periods, and thus computed the annual percentage rate (APR) for a monthly rate of 1.5 percent.

FNFEIN$(FEIN) CONVERT FEIN TO STRING

This function generates a string from a numeric Federal Employer Identification Number (FEIN) in FEIN, with the format NN-NNNNNNN or 10 blanks if FEIN=0.

FNFLD(FMT$) DETERMINE TOTAL ELEMENTS IN FORMAT

This function returns the number of elements defined in format FMT$. The result is always a positive integer.

Example:
```
FMT$="#SSORDHDR";
FORMAT INCLUDE #FMT$;
NELM=FNFLD(FMT$)
```

This example sets NELM equal to the number of elements defined in the format #SSORDHDR.

FNFYR(FSM,DATE) COMPUTE FISCAL YEAR FROM DATE

This function computes a fiscal year for the SQL (DTN) date value in DATE based upon the fiscal year starting month FSM, where FSM is in the range 1-12 inclusive. It may be used in place of a table-based accounting period methodology.

Examples: FYR=FNFYR(FSM,DATE)

Assuming that FSM=6 (fiscal year starts in June) and DATE=729341 (Nov 11, 1997), this example assigns the value 1998 to FYR.

```
FYR=FNFYR(FSM,DATE)
```

Assuming that FSM=1 (fiscal year starts in January) and DATE=729341, the above example assigns the value 1997 to FYR. See also the FNPER and FNQTR functions below.

Note that this function is based upon generally accepted American accounting practices, in which the calendar year in which the fiscal year ends determines the fiscal year.

FNHELP$(FMT$,ELM) RETURN HELP SCREEN NAME

This function returns the help screen name assigned to element ELM of format FMT$. The resulting string will be in LLNNNNNN style with trailing blanks stripped. If no help screen has been defined the result will be a two byte string containing only the LL component.

Example:
```
FMT$="#SSORDHDR",
ELM=FNELM("oe_date");
FORMAT INCLUDE #FMT$;
H$=FNHELP$(FMT$,ELM)
```

FNIDATE(FMT$,ELM,OCC) CONVERT BINARY SQL DATE TO INTEGER NUMERIC FORM

This function converts the four or six byte binary SQL date stored in occurrence OCC of element ELM in format FMT$ into a standard date/time numeric (DTN) variable with no fractional content. The occurrence value should be zero for elements defined with only one occurrence (a literal zero may be used). The expression evaluates to zero if no date has been stored in the field.

Example: DATE=FNIDATE("#MYFORMAT",8,0)

See also the FNDATE function for the floating point version of this function.

FNJDAY(DATE) **GENERATE JULIAN DAY INDEX**

This function takes the SQL (DTN) date number in DATE and produces a Julian day index in the range 1 through 365 inclusive (366 for leap years). If the value passed in DATE is zero the Julian day index for today's date as determined by the CDN system variable will be returned.

Example:
```
DATE=729781;
PRINT FNJDAY(DATE)
```

This example prints 25.

FNLBDT(DATE$) **CONVERT 6 BYTE SQL BINARY DATE TO DTN NUMBER**

This function converts the six byte binary SQL date/time value in DATE$ into a date/time numeric (DTN) equivalent. PRECISION 4 is required to fully extract the time component (±5 second resolution).

Example:
```
X$=XFD(CH,0);
PRECISION 4;
DATE=FNLBDT(X$(66,6))
```

On a UNIX or Linux system, the above example extracts the date and time of last write to the file opened on channel CH and converts that date and time into its DTN equivalent. The result has ±5 second resolution.

FNLBDT$(DATE) **CONVERT DTN NUMBER TO 6 BYTE SQL BINARY FORMAT**

This function converts the SQL date (DTN) value in DATE into a six byte binary SQL equivalent. The resulting SQL binary date string has a ±5 second resolution.

Example:
```
PRECISION 4;
SQL_DATE$=FNLBDT$(CDN)
```

This example assigns the six byte binary SQL date equivalent of the current date and time of day to the variable SQL_DATE$ with ±5 second resolution.

FNLDAT$(FMT$,ELM,OCC) GENERATE MM/DD/YYYY DATE OR BLANKS FROM FORMAT

This function takes the four or six byte binary SQL date value in occurrence OCC of element ELM in format FMT$ and produces a MM/DD/YYYY style string if DATE is non-zero. The leading zero of the MM component is replaced with a blank for the months January through September. If the value in the element evaluates to zero a blank field ten characters in length is produced. Only elements whose date type is 5 (SQL) should be passed to this function.

Examples: PRINT FNLDAT$(FMT$,ELM,OCC)

If the value in occurrence OCC of element ELM in format FMT$ is 729752 this example will print 12/27/1998.

See also the FNDAT$ and FNLDT$ functions.

FNLDT$(DATE) GENERATE MM/DD/YYYY DATE OR BLANKS

This function takes the SQL (DTN) date number in DATE and produces a MM/DD/YYYY style string if DATE is non-zero. The leading zero of the MM component is replaced with a blank for the months January through September. If DATE is zero a blank field ten characters in length is produced.

Examples: PRINT FNLDT$(729762)

This example prints “ 1/06/1998”.

PRINT FNLDT$(729752)

The above example prints “12/27/1998”.

PRINT FNLDT$(0)

The above example prints “ ” (10 blanks).

See also the FNDT$ function.

FNLEN(FMT$,ELM) RETURN DISPLAY LENGTH OF ELEMENT IN FORMAT

This function returns the size in bytes of the display length of element ELM in format FMT$. For elements defined as string data the result is the same as the actual data size as determined by the FNSIZ function (see below).

In other cases, the value is equal to the length of the default display mask for the element in question (see the FNMASK$ function below) and will not necessarily be the same as the element's actual data length. If the element is a 4 or 6 byte binary SQL date (date type 5) the value returned by this function may be erroneous.

FNMASK$(FMT$,ELM) GENERATE NUMERIC ELEMENT DISPLAY MASK

This function returns the display mask for numeric element ELM of format FMT$. Each numeric element in a format has an associated default display mask that is automatically created during the definition of the element. It is possible to extract and use the mask independently of the FMT string function within conventional 3GL programs. If the element is a 4 or 6 byte binary SQL date (date type 5) the value returned by this function may be erroneous.

Example:
```
FMT$="#XACXAAR",
ELM=6;
FORMAT INCLUDE #FMT$;
MASK$=FNMASK$(FMT$,ELM)
```

FNMESG$(FMT$,ELM) EXTRACT ELEMENT MESSAGE ATTRIBUTE

This function returns the value stored in the message attribute field associated with element ELM of format FMT$. IDOL-IV permits the definition of several types of message fields that can determine how input and display processing will occur within the IDOL-IV context. Outside of IDOL-IV, you are free to use and interpret this attribute field as you see fit.

Example:
```
FMT$="#XACXAAR",
ELM=6;
FORMAT INCLUDE #FMT$;
MESG$=FNMESG$(FMT$,ELM)
```

FNMON(DATE) EXTRACT MONTH COMPONENT FROM DTN VALUE

This function returns a numeric month value from the date/time numeric (DTN) value in DATE. The result is in the range 1-12 inclusive.

Example:
```
MONTH=FNMON(CDN)
```

This example returns a numeric month value for today's date.

FNNTP(FMT$,ELM) EXTRACT NUMERIC TYPE FROM DATA ELEMENT

This function returns the numeric type of element ELM in format FMT$. The result will always be zero for elements not defined as numbers. Since elements defined as type 0 numbers (floating point with sign) cannot be distinguished from elements defined as strings, it may be necessary to use this function in conjunction with the FNPRC function (see below) to clarify the result.

Example:
```
FMT$="#SSORDLIN",
ELM=4;
FORMAT INCLUDE #FMT$;
NTYPE=FNNTP(FMT$,ELM)
```

FNNUM(FMT$,ELM,OCC) EXTRACT NUMBER FROM FORMAT ELEMENT DATA AREA

This function returns the numeric value stored in occurrence OCC of element ELM in format FMT$, with automatic conversion according to numeric type. If the target element has been defined as a single occurrence element OCC must be zero (or a literal zero may be used). In order to assure that the value in the selected element is received at full precision you should set precision equal to the precision value defined for the element (see the FNPRC function below). Zero is returned if this function is applied to a non-numeric element. In such a case, no execution error occurs, but the ERC system variable will be set to 1 and the ERR system variable will be set to 26. Results may be unpredictable when this function is applied to an element defined as a date.

Example:
```
FMT$="#SSORDLIN",
ELM=4,
OCC=0,
P=PRC;
PRECISION FNPRC(FMT$,ELM)-1;
QUANTITY=FNNUM(FMT$,ELM,OCC);
PRECISION P
```

This example temporarily sets precision to the precision level of the 4th element in #SSORDLIN, assigns the value stored in that element to QUANTITY and then restores precision to its previous value. An error will occur with the PRECISION FNPRC(FMT$,ELM)-1 statement if the selected element is not numeric.

FNNUM$(NUMBER,FMT$,ELM) CONVERT NUMBER TO ELEMENT STRING EQUIVALENT

This function converts the numeric value NUMBER into the internal representation used to store data into element ELM of format FMT$, with automatic conversion according to the element's numeric type.

The result can be directly written into an element's data area with the LET FMD directive or used to produce a string variable for use as a record key in instances where a numeric element is defined as a key (see also the buildkey function on page 70). While seldom used, this method of storing a numeric value into an element is presented for completeness. In most cases, it is less complicated to assign numbers to soft coded format elements with LET FMT(FMT$,ELM,OCC)=STR(NUMBER).

Example:
```
FMT$="#SSORDLIN",
ELM=4,
OCC=0,
NUMBER=167.4021;
LET FMD(FMT$,ELM,OCC)=FNNUM$(NUMBER,FMT$,ELM)
```

In the above example, the internal equivalent of NUMBER is generated for and stored into the 14th element of the #SSORDLIN format. The correct numeric type and precision is determined from the element's attributes.

FNOCC(FMT$,ELM) EXTRACT NUMBER OF OCCURRENCES IN ELEMENT

This function returns the number of occurrences in element ELM of format FMT$. The result will be zero when an element structured as a single occurrence type is specified.

Example:
```
FMT$="#MYFORMAT";
FORMAT INCLUDE #FMT$,OPT="DEFAULT";
OCC=FNOCC(FMT$,FNELM(FMT$,"balance"))
```

The above example will return the number of occurrences in the BALANCE element of the #MYFORMAT format. If the result is zero then BALANCE has only one occurrence. Otherwise, BALANCE has OCC occurrences defined. Note that the result from this function can never be 1.

FNPCT(N1,N2) COMPUTE PERCENT CHANGE

This function computes the percent change between the existing value N1 and the new value N2. If N2 is less than N1 the result will be negative. Error 40 (numeric value overflow) will occur if N1=0.

Examples:
```
COST=1.54,
PRICE=1.76;
PRINT "Sales markup is ",FNPCT(COST,PRICE)*100:"##.00%."
```

The above example will print Sales markup is 14.29%.

```
N1=1.76,
N2=1.54;
PRINT  "The cost reduced",ABS(FNPCT(N1,N2)*100)," percent."
```

The above example will print `The cost reduced 12.5 percent.`

FNPER(FSM,DATE) COMPUTE ACCOUNTING PERIOD FROM MONTH

This function computes the one-based accounting period for the SQL date passed in `DATE` relative to the fiscal year starting month `FSM`, where `FSM` is in the range `1-12` inclusive. It may be used in place of a table-based accounting period methodology.

Example:
```
DATE=729781;
PER=FNPER(FSM,DATE)
```

Assuming that `FSM=6` (fiscal year starts in June), this example assigns the value `8` (eighth accounting period) to `PER`. See also the `FNQTR` function below.

FNPER$(FSM,DATE) GENERATE PERIOD STRING FROM DATE/TIME NUMERIC

This function generates a fiscal period string in the form `YYYYPP` from the SQL (`DTN`) value in `DATE` relative to the fiscal year starting month `FSM`, where `FSM` is in the range `1-12` inclusive. The `YYYY` component is the fiscal year associated with `DATE` and the `PP` component is the accounting period. `FNPER$` may be used in place of a table-based accounting period methodology.

Example:
```
PERIOD$=FNPER$(FSM,DATE)
```

Assuming that `FSM=6` and `DATE=729323` (October 24, 1997), this example assigns the string `199805` to `PERIOD$`. See also the `FNFYR`, `FNPER` and `FNQTR` functions.

FNPH$(PH) GENERATE PHONE NUMBER STRING

This function creates a string version of the telephone number `PH` in the format `NNN-NNNN` if `PH` is non-zero. If `PH` is zero a string of eight blanks is produced.

Examples:
```
PH=9423597;
PRINT FNPH$(PH)
```

The above example produces `942-3597`.

```
PRINT FNPH$(0)
```

The previous example produces eight blanks.

```
PRINT FNPH$(21)
```

The above example produces `000-0021`.

FNPORT(P$) **CONVERT TASK ID TO PORT NUMBER**

This function converts a task identification string, such as derived from `FID(0)`, into a numeric port number. For terminal tasks, such as `T0` or `W3`, the value returned will be between `0` and `929` inclusive. Ghost tasks will return values between `930` (`G0`) and `965` (`GZ`) inclusive. Tasks that are not terminals or ghosts will return nonsense values. `FNPORT` may be used in lieu of the `idport` called function (page 132)—both return identical results.

Example:
```
P=FNPORT("U4")
```

The above example returns `66` in `P`.

```
P=FNPORT("GF")
```

The above example returns `945` in `P`.

FNPOSP$(FMT$,ELM) **EXTRACT ELEMENT POST-PROCESS ATTRIBUTE**

This function returns the value stored in the post-process attribute field associated with element `ELM` of format `FMT$`. IDOL-IV permits the definition of two types of post-process fields that can determine how post-input processing will occur within the IDOL-IV context. Outside of IDOL-IV, you are free to use and interpret this attribute field as you see fit.

Example:
```
FMT$="#XACXAAR",
ELM=6;
FORMAT INCLUDE #FMT$;
POSP$=FNPOSP$(FMT$,ELM)
```

FNPRC(FMT$,ELM) **RETURN PRECISION OF NUMERIC ELEMENT IN FORMAT**

This function returns the decimal precision plus one of element `ELM` in format `FMT$` or `0` if the element is not numeric. Internally, numeric elements have a non-zero precision value, a fact which facilitates the differentiation between true numeric elements and elements that happen to have string data in number form. To get the true precision of an element subtract `1` from the result of this function.

Examples:
```
P=PRC;
FMT$="#SSORDLIN";
FORMAT INCLUDE #FMT$,OPT="DEFAULT";
ELM=FNELM(FMT$,"ordqty");
PRECISION FNPRC(FMT$,ELM)-1
```

This example sets the current precision to that defined for the ORDQTY element in the #SSORDLIN format. Note that the above code fragment would cause ERR=41 if the selected element was not numeric.

```
P=PRC;
FMT$="#SSORDLIN";
FORMAT INCLUDE #FMT$,OPT="DEFAULT";
ELM=1;
IF FNPRC(FMT$,ELM)=0
  PRINT "Element `",FNELM$(FMT$,ELM),"' is not numeric!"
```

FNPREP$(FMT$,ELM) EXTRACT ELEMENT PRE-PROCESS ATTRIBUTE

This function returns the value stored in the pre-process attribute field associated with element ELM of format FMT$. IDOL-IV permits the definition of several types of pre-process fields that can determine how pre-input processing will occur within the IDOL-IV context. Outside of IDOL-IV, you are free to use and interpret this attribute field as you see fit.

Example:
```
FMT$="#XACXAAR",
ELM=6;
FORMAT INCLUDE #FMT$;
PREP$=FNPREP$(FMT$,ELM)
```

FNQTR(FSM,DATE) COMPUTE ACCOUNTING QUARTER FROM MONTH

This function computes the one-based accounting quarter for the SQL date passed in DATE relative to the fiscal year starting month FSM, where FS is in the range 1-12 inclusive. It may be used in place of a table-based accounting period methodology.

Example:
```
DATE=729781;
QTR=FNQTR(FSM,DATE)
```

Assuming that FSM=6 (fiscal year starts in June), this example assigns the value 3 (third quarter) to QTR. See also the FNPER function above.

FNRND5(DA,RT) **ROUND DOLLAR VALUE TO NEAREST FIVE CENTS**

This function rounds a dollar amount in DA to the nearest five cents so that, for example, 1.97 becomes 1.95 and 1.98 becomes 2.00. Similarly, 1.92 would be rounded down to 1.90 and 1.93 would be rounded up to 1.95. The RT value represents a threshold value below which no rounding will occur.

Example:
```
PRINT FNRND5(1.03,1.00)
```

The above example prints 1.05.

```
PRINT FNRND5(.99,1.00)
```

The above example prints .99, as no rounding will occur when the amount is less than the one dollar threshold.

FNSBDT(DATE$) **CONVERT FOUR BYTE BINARY DATE TO DTN NUMBER**

This function converts the four byte binary SQL date in DATE$ into a date/time numeric (DTN) integer equivalent. The epoch used by this function is Sunday December 31, 1899 (DTN value 693597). Values that decode to dates on or prior to the epoch will return zero.

FNSBDT$(DATE) **CONVERT DTN NUMBER TO FOUR BYTE BINARY FORMAT**

This function converts the date/time numeric (DTN) value in DATE into a four byte binary SQL equivalent. The epoch used by this function is Sunday December 31, 1899 (DTN value 693597). DATE values less than 693597 will produce spurious results.

FNSIZ(FMT$,ELM) **RETURN SIZE OF ELEMENT DATA AREA IN FORMAT**

This function returns the size in bytes of element ELM's data area in format FMT$. For elements defined as string data or numeric types 0, 1 or 2 the result is the same as the value returned by the FNLEN function (page 42). In other cases, the value returned won't necessarily be identical to the display form of the data. See the FNLEN function to determine an element's display length.

FNSL$(L$,C$) **EXTRACT DELIMITED SUBSTRING**

This function returns the contents of L$ up to the delimiter character C$. The delimiter is not included in the return string. If the delimiter is not present in L$ the entire content of L$ will be returned.

Example:
```
L$="aaa:bbbb:ccccc",
S$=FNSL$(L$,":")
```

The above example will return aaa in S$.

```
L$="aaa:bbbb:ccccc",
S$=FNSL$(L$,",")
```

The above example will return aaa:bbbb:ccccc in S$.

See the FNSUBS$ function (below) for returning a remainder string.

FNSSN$(SSNUM) CONVERT SOCIAL SECURITY NUMBER TO STRING

This function generates a string from a numeric Social Security number in SSNUM, resulting in the format NNN-NN-NNNN or 11 blanks if SSNUM=0.

FNSTR$(FMT$,ELM,OCC) EXTRACT STRING DATA FROM ELEMENT

This function extracts the display string equivalent of data stored in occurrence OCC of element ELM in format FMT$. If the target element has been defined as a single occurrence element OCC must be zero or a literal zero must be supplied. For string elements, the output is the unstripped string data in the element. When applied to numeric elements, the output is formatted according to the default mask for the element, which may be determined from the FNMASK$ function (above). For elements defined as SQL date fields (type 5 dates), the output will be either MM/DD/YY (four byte elements) or MM/DD/YY HH:MI:SS (six byte elements). HH:MI:SS is in 24 hour format with ±5 second resolution.

Example:
```
FORMAT$="#SSORDHDR",
ELM=8,
OCC=0;
S$=FNSTR$(FORMAT$,ELM,OCC)
```

FNSTRIP$(FMT$,ELM,OCC) EXTRACT & STRIP STRING DATA FROM ELEMENT

This function extracts and strips string data from occurrence OCC of element ELM in format FMT$. If the target element has been defined as a single occurrence element OCC must be zero or a literal zero must be supplied. *Thoroughbred* internally pads string data to the full extent of the element size. This function removes that padding. It has no effect on numeric or SQL date elements.

Example:
```
FMT$="#SSORDHDR",
ELM=1,
OCC=0;
S$=FNSTRIP$(FMT$,ELM,OCC)
```

FNSUBS$(L$,C$) **EXTRACT TRAILING SUBSTRING**

This function returns the contents of L$ following the first instance of the delimiter character C$. The first instance of the delimiter will not be included in the return string. The behavior of this function is undefined if the delimiter character is not present in L$.

Example:
```
L$="aaa:bbbb:ccccc",
S$=FNSUBS$(L$,":")
```

This example will return bbbb:ccccc in S$.

Sequences of FNSL$ (above) and FUSUBS$ can be used to break out fields from strings read in from text files.

FNSUM(FMT$,ELM,OCC,Q) **NUMERIC ELEMENT ADDITION**
or
FNSUM(FMT$,ELM,OCC,-Q) **NUMERIC ELEMENT SUBTRACTION**

This function produces a numeric value equal to the sum of Q and the numeric value stored in occurrence OCC of element ELM in format FMT$. If the target element has been defined as a single occurrence element OCC must be zero or a literal zero must be supplied. Inverting the sign on Q inverts the operation. The current precision is used, which means you should adjust precision to that of the target element for greatest accuracy.

Example:
```
FORMAT$="#SSORDLIN",
ELM=7,
OCC=0,
P=PRC;
PRECISION FNPRC(FORMAT$,ELM)-1;
NEWQTY=FNSUM(FORMAT$,ELM,OCC,QTY);
PRECISION P
```

The above example adds the value in QTY to that stored in the #SSORDLIN.SHPQTY element and assigns the result to NEWQTY, using the precision level assigned to the SHIPQTY element.

FNSUM$(FMT$,ELM,OCC,Q) NUMERIC ELEMENT ADDITION (STRING OUTPUT)
or
FNSUM$(FMT$,ELM,OCC,-Q) NUMERIC ELEMENT SUBTRACTION (STRING OUTPUT)

This function produces a string equivalent of the result of the FNSUM function described above. Refer to FNSUM for a description of the required parameters.

Example:
```
FORMAT$="#SSORDLIN",
ELM=7,
OCC=0,
P=PRC;
PRECISION FNPRC(FORMAT$,ELM)-1;
LET FMT(FORMAT$,ELM,OCC)=FNSUM$(FORMAT$,ELM,OCC,QTY);
PRECISION P
```

This example adds the value in QTY to that stored in the #SSORDLIN.SHPQTY element and assigns the result back to #SSORDLIN.SHPQTY, using the element's precision level. This code example is conceptually identical to the hard coded expression #SSORDLIN.SHPQTY=#SSORDLIN.SHPQTY+QTY.

FNTD$(DATE) GENERATE HH:MI {AM|PM} TIME STRING

This function takes the SQL (DTN) date number in DATE and produces an HH:MI {AM|PM} style time string or eight blanks if DATE=0.

Examples:
```
DATE=729503.3609;
PRINT FNTD$(DATE)
```

This example prints " 8:39 AM". Note that a leading zero in the hour is replaced with a blank.

```
DATE=729503.0110;
PRINT FNTD$(DATE)
```

This example prints 12:15 AM (15 minutes after midnight). Note that *Thoroughbred* will incorrectly return 00:15 AM if the time is derived with the expression NTD(DATE,"HH:MI AM").

FNTIM$(T) CONVERT TIM TO TIME OF DAY STRING

This function takes the floating point time-of-day value in T, as would be derived from the TIM system variable and produces an HH:MI {AM|PM} style time string. The result is a string with a constant length of eight characters.

Examples: **T$=FNTIM$(9.25)**

The above example will return " 9:15 AM". Note that a leading zero in the hour is replaced with a blank.

T$=FNTIM$(22.93)

The above example will return "10:55 PM".

T$=FNTIM$(.02)

The above example will return "12:01 AM".

FNVVAL$(FMT$,ELM) **EXTRACT ELEMENT VALID VALUES ATTRIBUTE**

This function returns the value stored in the valid values attribute field associated with element ELM of format FMT$. IDOL-IV permits the definition of several types of valid values fields that can determine the amount and type of input that may be accepted within the IDOL-IV context. Outside of IDOL-IV, you are free to use and interpret this attribute field as you see fit.

Example:
```
FMT$="#XACXAAR",
ELM=6;
FORMAT INCLUDE #FMT$;
V$=FNVVAL$(FMT$,ELM)
```

FNWEEK(DATE) **GENERATE CALENDAR WEEK NUMBER**

This function derives a calendar week number in the range 1-53 from the SQL (DTN) date value in DATE. The week number is based upon a Julian date conversion and thus may return 53 if DATE occurs in a week that spans into the following year.

Examples:
```
DATE=DTN("120897","MMDDYY");
PRINT FNWEEK(DATE)
```

This example prints 49.

```
DATE=DTN("123197","MMDDYY");
PRINT FNWEEK(DATE)
```

The above example prints 53.

FNYEAR(DATE) **EXTRACT YEAR COMPONENT FROM DTN VALUE**

This function returns a numeric year value from the SQL (DTN) date value in DATE. The result is in the range 0001-9999 inclusive.

Example: **YEAR=FNYEAR(CDN)**

This example returns a numeric year value for today's date.

FNZIPC$(ZIP$) **CONVERT ZIP CODE STRING TO COMPRESSED BINARY**

This function converts the ZIP code string in ZIP$ into a compressed binary value with a constant four byte length. ZIP$ may either be a five digit ZIP code, such as 12345, or a nine digit ZIP+4 code, such as 123456789 or 12345-6789 (either format is acceptable). The resulting binary string can be sorted like any other string data and is guaranteed to sort in ascending numeric order. It is the responsibility of the program using this function to assure that the unencoded ZIP code string is in one of the three acceptable formats (ZZZZZ, ZZZZZNNNN or ZZZZZ-NNNN).

Examples:
```
ZIP$="12345";
CZP$=FNZIPC$(ZIP$)
```

This example returns the value $075BB290$ in CZP$.

```
LET FMD(FMT$,ELM,OCC)=FNZIPC$("12345-6789")
```

This example stores the value $075BCD15$ in occurrence OCC of element ELM in format FMT$, assuming this element has been properly structured to receive a four byte unsigned binary value.

See also zipbin (page 263).

FNZIP$(CZP$) **CONVERT COMPRESSED BINARY ZIP CODE TO ASCII**

This function takes a compressed binary ZIP code string as generated by the FNZIPC$ function above and returns a display string in the form ZZZZZ-NNNN, with a constant length of 10 characters. The -NNNN portion is replaced with blanks if the compressed ZIP code does not evaluate to more than five digits.

Examples: **PRINT FNZIP$(CZP$)**

If CZP$ contains $075BB290$ this example will print "12345 ".

PRINT FNZIP$(FMD(FMT$,ELM,OCC))

If occurrence OCC of element ELM in format FMT$ contains the binary value $075BCD15$, this example will print "12345-6789".

The following examples illustrate how the combination of FNZIPC$ and FNZIP$ handles certain forms of non-standard ZIP code strings.

PRINT FNZIP$(FNZIPC$(""))

The above example will print " " (10 blanks).

PRINT FNZIP$(FNZIPC$("00000-0000"))

The above example will also print 10 blanks.

COOKBOOK CALLED FUNCTIONS

The following pages describe each cookbook called function in detail. Each narrative begins with a call syntax description, followed by a description of the parameters to be passed to the function and the parameters returned, and finishes up with a discussion of how the function operates, including a synopsis of user interactivity where applicable. One or more programming examples may be presented to illustrate how a function might be used in the context of a larger program. When the discussion includes 4GL references, it is presumed the reader has studied preceding material presented herein.

In addition to the narratives contained herein, many functions have built-in help capabilities. For such functions, a summary description of the function's purpose, call parameters and returns may be displayed from console mode in one of three ways:

```
   CALL "<function>"
```
or
```
   CALL "<function>",-1
```
or
```
   CALL "<function>","?"
```

The first method is applicable to functions that require one or more mandatory parameters: `input` (page 133) is a typical example of this type. The second method is applicable to functions that accept optional numeric parameters: `fkydcd` (page 107) is one such example. The third method applies to functions that take optional string parameters, e.g., `closeall` (page 78). Functions to which no parameters are to be passed or returned do not have any built-in help. Be aware that the built-in help will operate only when the help request is made from console mode. Within a running program, the first method will cause `ERR=36` (`CALL/ENTER Mismatch`), and the second and third methods will either produce erroneous results or any one of several error types, depending on what the function expects as parameters.

4glpline — Generate Print Line From Display Format

Syntax:

```
CALL "4glpline",DFMT$,ELMLO,ELMHI,PITCH$,DC[ALL],LINE$
```

Call Parameters:

DFMT$	Display format name in #LLNNNNNN style. The format should be loaded with appropriate data prior to calling this function.
ELMLO	Starting element number. If zero, the first element in the format will be assumed.
ELMHI	Ending element number. If zero, the last element in the format will be assumed.
PITCH$	Printer pitch as would derived from the lpsetup (page 164), pagsetup (page 192) or rptsetup (page 223) functions.
DC[ALL]	Columns at which each field is to be printed. This is a zero-based array as would be derived from the pagsetup or rptsetup functions.
LINE$	Optional print line attribute string. To turn on line attributes, LINE$ must be formatted as follows:

1,2	The characters 0102, a sequence that indicates to 4glpline that one or more attribute flags follow.
3,2	The binary value of the channel opened to the target printer, derived with BIN(LP,2), where LP is the printer channel number.
5,n	Up to four attribute flags, defined as follows:

B	Bold faced (double strike).
E	Emphasized (shifted double strike).
I	Italic.
U	Underlined.

These flags may be in any order and either upper or lower case. See text for more information. Repeating a flag will not cause an error.

Returns:

LINE$	Formatted print line which may be directly written to a channel OPENed to a printer. See text.
ERR	Any execution error, such as undefined format.

4glpline interprets the data found in the display format DFMT$ and generates a string variable LINE$ whose content is an image of a print line. If both ELMLO and ELMHI are zero the entire format will be converted. Otherwise, the conversion will be bound by the elements specified in ELMLO and ELMHI. The structure of the print line includes column positioning values for each resulting field, as well as the pitch setting and any optional attributes. Hence, the line may be written to a printer channel with no additional processing. See the discussion at pagsetup for details on how a display format should be structured and how to create a report skeleton.

Conversion of element data into a form suitable for output may be handled by a user-written external processor defined in an element's pre-process attribute or by internal heuristics applied by 4glpline to each element in sequence. The latter method will suffice in many cases, whereas an external processor may be a better choice for certain types of binary data.

In order for 4glpline to properly operate, previous calls to the lpsetup, pagsetup or rptsetup functions (pages 164, 192 and 223, respectively) must be made to establish the value of PITCH$ and to build the column array DC[]. See the documentation for these functions for additional information. Of the three, pagsetup is the most convenient to use, as it performs all needed calculations to generate these values.

The manner in which 4glpline produces data conversion depends upon the attributes of each element in the display format:

- If an element has an "external processor" pre-process attribute defined, the program named in the pre-process attribute will be CALLed and all subsequent processing of the element's data will handled by that program. The expected form of the pre-process attribute string is as follows:

  ```
  0,<prog>,?FMT?,?ELM?[,<parm1>[,<parm2>[,<parm3>]]]...
  ```

  where the leading 0 is an IDOL-IV requirement, <prog> is the public program to be CALLed and <parm1>, <parm2>, <parm3>, etc., are optional comma-delimited parameters that will be passed into <prog> via a string array. The ?FMT? and ?ELM? parameters are required. When 4glpline calls <prog> the ?FMT? and ?ELM? parameters will have been translated into the format name and element number, respectively. Everything after ?ELM? is optional. If any parameter must include one or more commas, that parameter must be surrounded by double quotes. For example:

  ```
  0,<prog>,?FMT?,?ELM?,"either,or"
  ```

  The above would pass either,or intact to the external processor program.

The call to <prog> from 4glpline will be as follows:

```
CALL <prog>,NP,PL$[ALL],O$
```

Variables definitions are as follows:

NP	Number of parameters passed in PL$[]. Assuming that the pre-process attribute string has been correctly structured, NP > 1.
PL$[0]	Reflects the value of NP, that is, NUM(PL$[0])=NP.
PL$[1]	Name of format being processed in #LLNNNNNN style.
PL$[2]	Element number being processed.
PL$[3]	Start of optional parameters after ?ELM? in the pre-process attribute string.
PL$[NP]	Final element in the PL$[] array.
O$	The output from <prog> after processing. O$ will be appended to LINE$ and become part of 4glpline's output.

Upon exiting, <prog> must set the ERR system variable to zero to indicate that O$ is valid (see seterr at page 234). Any non-zero value of ERR will cause 4glpline to ignore O$. In the event <prog> cannot be executed 4glpline will revert to internal processing as next described.

- Elements defined as numeric will be internally converted as though the STR(N:MASK$) conversion has been applied, with the element's default (DNM) mask being used to format the string. 4glpline can also recognize an optional mask in the valid values element attribute field in the form DM="<MASK>", where <MASK> is any reasonable numeric mask, such as (###,###.00). The mask defined in the valid values field overrides the element's default mask.

- Elements defined as SQL dates (type 5) are converted to MM/YY/DD style, with a leading zero in the MM component replaced with a blank. If the element size is six bytes, indicating that a time of day value is present, the conversion will include the time in HH:MI {A|P}M style. A leading zero in the HH component will be replaced with a blank. For other date and time styles an external processor is required (see above).

- All other element types will generate string data, as derived with FMT, using the default mask (DNM) for the element.

The value in LINE$ does not enable any special printer attributes. If you wish to turn on bold faced, emphasized, italic and/or underline, configure LINE$ as described above prior to calling 4glpline. For example, to turn on bold face and underline, the value for LINE$ would be 0102+BIN(LP,2)+"BU", where LP is the printer channel number.

In order for this to have any effect, the SF/SB (bold on/off), EMON/EMOFF (emphasized on/off), ITON/ITOFF (italic) and BU/EU (underline on/off) mnemonic pairs must be defined for the target printer. If any of these mnemonics is missing 4glpline will silently ignore the attribute string in LINE$. Otherwise, 4glpline will add the necessary mnemonics to LINE$.

Example:

```
01000 MAIN: READ (CH,END=DONE) #RFMT$;
      CALL "copyfmt",RFMT$,DFMT$,1;
      CALL "4glpline",DFMT$,ELMLO,ELMHI,PITCH$,DC[ALL],LINE$
      PRINT (LP)LINE$;
      GOTO MAIN;
      …program continues…
```

The above code fragment reads a record from a file into the physical format named in RFMT$, copies that format into the display format named in DFMT$, calls 4glpline to generate the print line variable LINE$ and then writes it to the printer. It is assumed a prior call to a function such as pagsetup established all variables needed to produce the printed report.

See also lpsetup (page 164), pagsetup (page192) and rptsetup (page 223).

4glto3gl Generate 3GL Variables From 4GL Format Data

Syntax:

```
CALL "4glto3gl",FORMAT$,PRFX$,LINE,MODE
```

Call Parameters:

FORMAT$	Source data format name in #LLNNNNNN style.
PRFX$	Prefix to be attached to each generated variable name. A null prefix will cause a runtime error in the main program. See text.
LINE	Main program line number to which code is to be merged. See text.
MODE	String data conversion mode:
	0 Data will be converted with FMT.
	1 Data will be converted with FMD.

Returns:

ERR	Any execution error, such as undefined format.

4glto3gl provides a mechanism for automatically generating 3GL variables from the data stored in a memory-resident format. When called, 4glto3gl will either create a new program line in the main program, with the line number LINE, or will append its code to an existing line with the same line number. The resulting statement will consist of variable assignments, one per element in the format and if an element has multiple occurrences, one assignment per occurrence.

Variable names are identical to those of the elements from which they are derived, prefixed with the character string passed with the PRFX$ parameter. Variable names that have been derived from multiple occurrence elements will be suffixed with an NNN value, where NNN is the occurrence number. For example, if the value in PRFX$ is EM, the element type is numeric, the element name is EARNINGS and the occurrence value is 3, the resulting variable name will be EM_EARNINGS_003 (up to 999 such occurrences can be processed).

4glto3gl produces several data conversions, depending on the attributes of each element in the format:

- Elements defined as numeric generate ordinary numeric variables, with the element precision used to define the variable precision.

- Elements defined as SQL dates (type 5) also generate numeric variables. For four byte SQL date elements, the resulting variable will be an integer. Variables derived from six byte SQL dates will be floating point with four place precision.

- All other element types will generate string data. If the MODE value is zero, the data will be derived with the FMT assignment directive, using the default mask (DNM) for the element. Otherwise, data will be derived using the FMD assignment directive, resulting in the literal transformation of the data in the element.

Because 4glto3gl produces merged code, it can only affect the main program.

Example:

```
00200 PAGHDR$="#GCPAGHDR",
      PRFX="PH",
      LINE=TCB(4)+1,
      MODE=0;
      CALL "4glto3gl",PAGHDR$,PRFX$,LINE,MODE
00210 EXECUTE "DELETE"+STR(LINE);
      …program continues…
```

The above call will cause 4glto3gl to generate line 201 in the main program with the variable assignments, which when executed, will generate a string or numeric variable for each element in the #GCPAGHDR format. While the EXECUTE statement in line 210 is not essential, it is recommended to restore the program to its previous state.

APPLICATION NOTES

As described above, 4glto3gl works by adding new code to the calling program. Because the added code consists of a series of assignment statements, resulting in the expansion of the program line to which they are added, it is possible for an error to occur if the number of assignment statements is too great. *Thoroughbred* has an upper limit on the number of assignments that can be compiled in a single line of code. Therefore, if you have many calls to 4glto3gl in your program and the formats involved have many elements in their definitions, it would be wise to use several line numbers to add the code generated by 4glto3gl.

4gltotal — Generate Totals From 4GL Format

Syntax:

```
CALL "4gltotal",RFMT$,TFMT$,OPCODE
```

Call Parameters:

RFMT$	Format name in #LLNNNNNN style from which numeric data will be retrieved for computing totals. The format should be loaded with appropriate data prior to calling this function.
TFMT$	Format name in #LLNNNNNN style into which totals will be accumulated.
OPCODE	Operation to be performed:

0	Addition.
1	Subtraction.

Returns:

ERR	Exit status:

0	OK.
1	Numeric overflow detected.

OPCODE	Unchanged if ERR=0. Otherwise, will return the element number of the element in TFMT$ where overflow was detected.

4gltotal provides a convenient way to utilize a logical format as a place to collect totals generated during a report run. Rather than using reams of 3GL variables to keep track of totals accumulated as the report progresses, 4gltotal does all the work by collecting the totals into the "totals" format referred to by TFMT$.

In order for 4gltotal to accomplish anything, the format designated by TFMT$ must have an element for each numeric value to be accumulated and these elements must have identical names and compatible attributes with the elements in the format referenced by RFMT$. TFMT$ can also have non-numeric elements, such as an element defined as a string (which could be a line caption—this will be illustrated below). Non-numeric elements will be ignored by 4gltotal and will not cause a processing error.

4gltotal cannot process elements that have been defined with multiple occurrences.

Examples:

In the following example, a physical format named #OPCMMAS is used to retrieve customer records during a report and a totals format, #OPCMMAST, is used to accumulate some sales totals for display at the end of the report. Reference is made to other cookbook functions that could be used to build up a report using formats and 4GL techniques. We will assume that both formats have six elements named SALES1 through SALES6, all of which are numeric (while it would be a good idea for them to be contiguous in the formats, it's not a requirement). Also, both formats have an element NAME, defined to be a string. The NAME element in the OPCMMAST format has the default value GRAND TOTALS ------> assigned to it.

```
0200 SETUP: CSMAST$="#OPCMMAST";
            CALL "statlink","OPCMMAS",CSMASF$,CSMAS$;
            FORMAT INCLUDE #CSMAS$;
            FORMAT INCLUDE #CSMAST$,OPT="DEFAULT";
            CSMAS=UNT;
            OPEN (CSMAS)CSMASF$;
```

The above code performs the preliminary setups: statlink (page 239) determines the names of the file and format associated with the OPCMMAS link and the formats are INCLUDEd. By using the DEFAULT option when the total format is INCLUDEd the caption text GRAND TOTALS ------> is automatically loaded into the NAME field. This will be useful when it comes time to print the totals. Also, the file associated with the OPCMMAS link is OPENed. We continue:

```
            READ (CSMAS,SRT="0",KEY="",DOM=MAIN)
1000 MAIN:  READ (CSMAS,END=DONE)#CSMAS$;
            FOR ELM=1 TO FNFLD(CSMAS$);
                PRINT (LP) FMT(CSMAS$,ELM):"DNM";
            NEXT ELM;
            OPCODE=0;
            CALL "4gltotal",CSMAS$,CSMAST$,OPCODE;
            ON ERR(0) GOTO OVERFLOW,MAIN
```

The above code reads a record, prints it (we're assuming the printer has already been opened to channel LP) and then calls 4gltotal to accumulate the totals. The FNFLD defined function returns the number of elements in the format passed as an argument. This process continues until end-of-file is reached:

```
9000 DONE:  CALL "copyfmt",CSMAST$,CSMAS$,1;
            FOR ELM=1 TO FNFLD(CSMAS$);
                PRINT (LP) FMT(CSMAS$,ELM):"DNM";
            NEXT ELM;
            CLOSE (LP)
```

This last code segment performs a little magic. Recall that when the OPCMMAST format was defaulted, the text GRAND TOTALS ------> was loaded into the NAME element. To print the totals, we used copyfmt (page 84) to copy the totals in OPCMMAST into OPCMMAS, initializing OPCMMAS in the process. Because OPCMMAS has a NAME field defined, just as OPCMMAST does, the GRAND TOTALS ------> text is copied along with the SALES1 through SALES6 values. When the print loop executes, the last line on the report will have GRAND TOTALS ------> in place of the customer's name, and the SALES1 through SALES6 totals under the columns of numbers.

asctobin — Convert ASCII Number String To Binary

Syntax:

```
CALL "asctobin",STRING$
```

Call Parameters:

STRING$	ASCII number string to convert. The string's format must be aaabbbcccddd…, where aaa is the first number, bbb is the second, etc. Each number must be three numerals. Blanks may be substituted for leading zeros.

Returns:

STRING$	Converted string, one byte per three ASCII numerals.

asctobin generates a binary string from the supplied ASCII data. The string can be used as flags, setup data for output devices, etc. For each three character number described in STRING$ a one byte binary value is returned.

Example:

```
STRING$="010020030040";
CALL "asctobin",STRING$
```

Upon return, STRING$ will contain $0A141E28$.

binzip — Decompress ZIP/Postal Code

Syntax:

```
CALL "binzip",ZIPC$,ZIP$
```

Call Parameters:

ZIPC$	Four byte compressed binary form of U.S. ZIP code or Canadian postal code, such as generated by zipbin (page 263). See text.

Returns:

ZIP$	ASCII U.S. ZIP code in NNNNN-NNNN format or Canadian postal code in ANA NAN format. See text.

binzip converts the compressed binary form of a U.S. ZIP code or Canadian postal code in ZIPC$ as generated by the zipbin function into an ASCII string suitable for display. binzip can also convert ZIP codes encoded with the FNZIPC$ (page 54) function. The output returned in ZIP$ is a constant 10 characters in length, padded as required with trailing blanks. U.S. ZIP codes are returned in NNNNN-NNNN format, with the -NNNN component replaced with blanks if the decompressed ZIP code is five digits. Canadian postal codes are returned in ANA NAN format, with alphabetic characters always in upper case. If ZIPC$ is null it will decompress into ten blanks. Otherwise, if ZIPC$ is not four bytes in length ERR=46 (string size) will occur.

Examples:

```
ZIPC$=$24039AEC$;
CALL "binzip",ZIPC$,ZIP$
```

The above sequence will return "60421-6044" in ZIP$.

```
ZIPC$=$8312A661$;
CALL "binzip",ZIPC$,ZIP$
```

The above sequence will return the Canadian postal code "L4U 3F1 " in ZIP$.

See also FNZIP$ (page 54), FNZIPC$ (page 54) and zipbin (page 263).

buildkey — Build Record Key From Data Format

Syntax:

```
CALL "buildkey",NAME$,K$
```

Call Parameters:

NAME$	Format name in #LLNNNNNN style or link name in LLNNNNNN style. Use a link name to build an alternate sort key from a format associated with an MSORT or TISAM file. See text.
K$	Sort name if NAME$ refers to a link. Ignored if a format name is supplied in NAME$ or the link in NAME$ is associated with a DIRECT or SORT file. See text.

Returns:

K$	Record key or null if processing error occurs.
ERR	0 OK, K$ is valid. 1 Format or link in NAME$ not defined. 2 Requested sort not defined (MSORT or TISAM). This status is also returned if the file associated with the named link is not accessible or is not a keyed file. *The setting of* ERR *will not cause an execution error in the calling program.*

buildkey provides a portable method of generating a record key from the data in a memory-resident format without a program having to have detailed knowledge of the format's structure. buildkey operates in one of two ways, depending on whether NAME$ names a format or a link. If NAME$ names a format, buildkey will create a primary record key from any elements that have been defined as keys (using their cardinal positions in the format) and return the result in K$. The entry value of K$ will be ignored.

If NAME$ names a link, buildkey will use the link information to construct a key based upon the sort named in K$. If K$ is null or the file associated with the link is a DIRECT or SORT file, buildkey will behave in the same manner as it would if a format name had been passed in NAME$ (the format name will be gotten from the link). Since the link information is what buildkey employs to gain access to the format, it is essential that the proper link name be passed so buildkey can work with the correct format. Exit status 1 will be returned if the link and/or format have not been defined. Unexpected results may occur if the wrong link name is passed in NAME$.

Formats do not contain any information on alternate sort keys, as sort definitions are embedded in the files themselves. Hence, buildkey works out the alternate key structure from information obtained from the file associated with the link. This process will abort with exit status 2 if the file is not accessible in the execution environment, is not a keyed file or the requested sort passed in K$ has not been defined. Note that sort names in TISAM files are actually numbers in string format. To access the primary sort, you would define K$="0". The first alternate sort would be K$="1", the next would be K$="2" and so forth. MSORT sort names can be more descriptive, with up to 20 characters.

Examples:

```
NAME$="#MYFORMAT";
CALL "buildkey",NAME$,K$
ON ERR(0) GOTO ERROR,OK
```

The above example builds a key from the data in the format #MYFORMAT. The resulting key is a primary key, which will work with all keyed file types. It is assumed that the format was INCLUDEd earlier in the program and has been loaded with data.

```
NAME$="OPCMMAS",
K$="ZIPSORT";
CALL "buildkey",NAME$,K$
ON ERR(1,2) GOTO OK,NOLINK,NOSORT
```

The above example builds a key from the format associated with the link OPCMMAS, using the sort definition ZIPSORT. The sort definition name implies that an MSORT file is associated with the OPCMMAS link.

```
NAME$="OPCMMAS",
K$="1";
CALL "buildkey",NAME$,K$
ON ERR(1,2) GOTO OK,NOLINK,NOSORT
```

The above example builds a key from the format associated with the link OPCMMAS, using the sort definition 1. This would be the correct way to build a key from the first secondary sort of a TISAM file.

chkesc — Poll For ESCape Keypress

Syntax:

```
CALL "chkesc",FLAG
```

Call Parameters:

FLAG	-1	Reset keyboard poll counter (see text). Any other value has no effect.

Returns:

FLAG	0	**[ESC]** keypress not detected.
	1	**[ESC]** keypress detected.
ERR		Returns same value as FLAG.

chkesc polls the **[ESC]** key and if **[ESC]** has been pressed since the last poll, sets FLAG. Once FLAG has been set subsequent calls will have no effect until the global variable chkesc has been cleared by a call with FLAG equal to -1. FLAG should always be set to -1 before starting the loop that checks for **[ESC]**. Otherwise, the possibility will exist that a call to chkesc will not detect an **[ESC]** keypress. The value of FLAG following a call to chkesc is also passed in the ERR system variable. The conditioning of ERR will not cause an error in the calling program.

It is recommended that the mnemonic 'EK' be defined for the terminal with the character emitted when the key designated as **[ESC]** is pressed (**[ESC]** doesn't necessarily have to be the ESCape key). The hexadecimal value for 'EK' cannot be the same as the BASIC escape code (usually 18 or **[CTRL][X]** on most UNIX/Linux systems), as BASIC will intercept the 'EK' keypress and chkesc will not be able to detect anything. If 'EK' has not been defined, the default value 03 will be assumed, thus defining **[CTRL][C]** to be **[ESC]**.

Example:

```
FLAG = -1;
…main program loop begins…
CALL "chkesc",FLAG;
ON FLAG GOTO CONTINUE,ABORT
```

choice — Get User's Choices From List

Syntax:

```
CALL "choice",LIST$,PROMPT$,SEP$,COL,ROW,IDX,TIMOUT[,HELP$]
```

Call Parameters:

LIST$	List of choices, with each character in LIST$ representing a possible choice. Case does not matter. At least two unique choices (characters) must be specified in LIST$. See text.
PROMPT$	Optional input prompt to be displayed. If null, no prompt will be generated or displayed. See text.
SEP$	Character to act as a choice delimiter in the input prompt (if displayed). If null the character / will be assumed.
COL/ROW	Screen coordinates at which to display the input prompt (if specified) or to accept input. See text.
IDX	Index indicating which choice in LIST$ will be the default choice. If IDX=0 there will be no default. See text.
TIMOUT	No response timeout in seconds, zero for no timeout.
HELP$	Optional parameter specifying the name of an IDOL-IV help screen that will be displayed if the user requests help with **[CTRL][O]**. May be omitted if no help screen has been defined.

Returns:

SEP$	If null on entry will return "/".
COL/ROW	Screen coordinates at which input was accepted.
IDX	Index indicating which choice was made or zero if input was aborted or timed out. See text.
ERR	0 OK, all returns are valid. 1 Input aborted with **[ESC]**. 2 Input timed out. 110 LIST$ format invalid. See text. 111 IDX is out of range for choices in LIST$. See text. *The setting of* ERR *will not cause an execution error in the calling program.*

choice is a general purpose input function that may be configured to accept a user's single character selection from a list of acceptable characters and convert his/her choice into a numeric progression. If text is passed in PROMPT$ the user will be prompted to make a selection, with the list of acceptable choices passed in LIST$ displayed as part of the prompt.

Conversely, if PROMPT$ is null, choice will simply position the cursor as required and accept the user's input. The general purpose nature of choice makes it easy to use in a variety of selective input situations.

The acceptable choices are passed in LIST$ as single alphanumeric characters in any desired order. If the user types one of those characters and presses ⏮, IDX will return a positive integer indicating which choice was made. If IDX is non-zero on entry to choice, the corresponding character in LIST$ will appear as the default choice under the cursor and the user may merely press ⏮ to make his/her selection. For example, if LIST$="ams" and IDX=2 the default choice will be M (choice maps all alpha characters to upper case).

If PROMPT$ is non-null on entry, choice will construct an input prompt from the text passed in PROMPT$ and a delimited list of choices derived from LIST$, using the character in SEP$ as the choice delimiter in the prompt (see the below example for more detail on this feature). PROMPT$ may include mnemonics as well as ordinary text. However, be careful with mnemonics that may affect cursor positioning or other positional aspects of the display (e.g., 'CE' or 'CL'). If PROMPT$ is null, no input prompt will be displayed and the only indication to the user that choice is awaiting input will be the display of either the default choice character or a question mark under the cursor.

Screen coordinates are processed in one of several ways. Specifying any valid positive value in COL and ROW (including 0,0) will cause choice to prompt the user at COL,ROW or if no text was passed in PROMPT$, accept input at COL,ROW. If ROW=-1 the row corresponding to the current cursor position will be assumed. If COL=-1 and PROMPT$ is non-null, a value will be computed for COL that will cause the input prompt to be horizontally centered on the screen. Otherwise, the column corresponding to the current cursor position will be used as the point for input. On a normal exit, choice will return the input coordinates (not the prompt coordinates) in COL and ROW.

LIST$ must contain at least two unique alphanumeric characters—in any order—or else choice will abort and return status 110 in the ERR system variable. There is no limit to the number of choices that may be specified in LIST$ other than the string length limit imposed by *Thoroughbred* itself. However, if LIST$ is too long, a positioning error will occur when choice attempts to display the prompt.

If IDX is non-zero on entry it must be valid for the range of LIST$. For example, if LIST$="ams" and IDX=4, choice will abort and return status 111, because LIST$ contains only three characters. Passing a negative or non-integer value in IDX will also return status 111. IDX will be zero if the user aborts or choice times out.

If you wish to provide the user with context-sensitive help, pass the name of the IDOL-IV help screen in the optional HELP$ parameter. In such a case, the prompt (if supplied) will indicate to the user that help is available (displayed by pressing **[CTRL][O]**). If HELP$ is omitted or is not in a form suitable for use by the IDOL-IV 8HELP called function, choice will beep the terminal if the user requests help.

Example:

```
MAIN: LIST$= "ams",
      PROMPT$="Auto, Manual or Single Mode",
      IDX=POS("a"=LIST$),
      COL=-1,
      ROW=12,
      TIMOUT=300;
      HELP$="AMCHOICE";
      CALL "choice",LIST$,PROMPT$,"",COL,ROW,TIMOUT,IDX,HELP$;
      ON ERR(0,1,2) GOTO ERROR,OK,ABORTED,TIMED_OUT

OK:   ON IDX GOTO ERROR,AUTO,MANUAL,SINGLE
```

When the above code fragment is executed the user will see the following prompt centered on row 12:

```
Auto, Manual or Single Mode (A/M/S  [Ctrl-O]        H [ESC]:  A        ):
```

The default choice (A) will appear under the cursor and choice will format the prompt so the user will know the acceptable choices. The user will be permitted to choose any one of A, M or S or abort with **[ESC]**, with a maximum of five minutes before choice automatically times out. Assuming the user makes a choice, IDX will return 1 if the choice is A, 2 if the choice is M, or 3 if the choice is S. The point at which input was accepted will be returned in COL and ROW. A help request initiated by pressing **[CTRL][O]** will result in the IDOL-IV help screen AMCHOICE being displayed.

See also the yesno function (page 262).

cklibcrc — Verify Object Library Cyclic Redundancy Checksum

Syntax:

```
CALL "cklibcrc",LIB$[,HCRR$[,CCRC$[,GENTIM]]]
```

Call Parameters:

LIB$	Name of object library to be checked.

Returns:

ERR	0	Computed cyclic redundancy checksum (CRC) matches library CRC, library is okay.
	1	Computed and library CRC values do not match, library is suspect.
	2	No valid library CRC found.
	3	LIB$ does not name a valid object library.

The setting of ERR *will not cause an execution error in the calling program.*

HCRC$	Library's generated CRC, a 32 bit unsigned quantity. Optional parameter.
CCRC$	CRC computed by cklibcrc, also a 32 bit unsigned quantity. Optional parameter.
GENTIM	Date and time library was generated, returned in DTN format with four place precision. Optional parameter.

cklibcrc is an SVR4 UNIX or Linux utility that computes a CRC for object library LIB$ and then compares the result against the CRC embedded into the library's header by the cksumlib CRC generator program included on the cookbook distribution disk. The result of this comparison is returned to the calling program via the ERR system variable. Passing the optional HCRC$, CCRC$ and GENTIM parameters permits the calling program to extract more information from the process. In particular, the GENTIM value may be used to determine how up-to-date the library is, as GENTIM will return the date/time stamp that was embedded in the library by the *RPSD BASIC utility at the time of generation. precision 4 is required to fully extract the time component of GENTIM. cklibsrc will silently exit with ERR=0 if the operating system under which *Thoroughbred* is running is not UNIX or Linux. All returns will be invalid in this case.

ERR=2 will occur if no CRC has been generated for the file named in LIB$, which would be the case for a new object library or one that has been compressed (compression overwrites the header with new data) but not checksummed.

This, in itself, does not signify that the library is tainted, only that no CRC has been generated. ERR=3 will be returned if LIB$ is not accessible in the execution environment, is not an object library, does not have a valid date/time stamp or otherwise appears to have a corrupted header.

Example:

```
LIB$="bcs.lib";
CALL "cklibcrc",LIB$,"", "",GENTIM;
ON ERR(1,2,3) GOTO OK,BADCRC,NOCRC,BADLIB
```

The above example checks the CRC of bcs.lib. If the library and computed CRC's match control will branch to the OK line label and the date and time of generation will be returned in GENTIM.

closeall — Close All Open Channels

Syntax:

```
CALL "closeall"[,OC$]
```

Call Parameters:

OC$	Optional list of channels that are to remain open, in OCH system variable format. Omit if all channels are to be closed.

Returns:

None.

closeall examines the OCH system variable for a list of open channels and then closes each one. closeall does not close any channel whose corresponding two byte binary channel number is found in the optional OC$ parameter or whose channel number is greater than 31999. The high channel number limitation will normally protect open object libraries, as well as the IDDBD data dictionary file. Unlike CLOSE(0) (close all channels), closeall also closes any open 4GL links, thus permitting the associated data formats to be deleted from memory. See *Thoroughbred*'s description of the OCH system variable for additional information.

Examples:

```
CALL "closeall"
```

The above example closes all channels from 1 to 31999 inclusive.

```
OC$=OCH,
F1=UNT;
OPEN (F1) "file1"
F2=UNT;
OPEN (F2) "file2"
F3=UNT;
OPEN (F3) "file3"
…do some processing…
CALL "closeall",OC$
```

The above example sets OC$ equal to the currently opened channel list and then opens and processes three files. The call to closeall closes only channels F1 through F3 inclusive, leaving all other channels opened.

clrfmts — Clear Data Formats From Memory

Syntax:

```
CALL "clrfmts"[,FT$]
```

Call Parameters:

FT$	Optional list of format names that are not to be deleted, in FMTNL system variable format, with each name padded to eight characters. See text.

Returns:

None.

Error Returns:

ERR=170	Format cannot be DELETEd (open link)

clrfmts examines the FMTNL system variable for a list of INCLUDEd formats and deletes them from memory. The optional FT$ parameter is used to skip formats that are to remain in memory. The IDOL-IV system format #IDSV is not cleared, nor are any formats that were soft-included with an OPEN(CH,OPT="LINK") directive.

Examples:

```
CALL "clrfmts"
```

The above example deletes all data formats from memory except #IDSV.

```
CALL "clrfmts","SCUSRRECSSITEM  "
```

The above example deletes all data formats except #IDSV, #SCUSRREC and #SSITEM.

```
FT$=FMTNL;
FORMAT INCLUDE #MYFORMAT;
...
CALL "clrfmts",FT$
```

The above example creates a list in FT$ of the formats already INCLUDEd and then INCLUDEs #MYFORMAT. Later, the call to clrfmts with FT$ deletes only those formats that were not INCLUDEd when the format list in FT$ was created, resulting in the deletion of #MYFORMAT.

`clrtxt` Clear A Text Region

Syntax:

```
CALL "clrtxt",ROW,LINES,MODE
```

Call Parameters:

`ROW`	Starting row of region to be cleared. `0<=ROW<25`.
`LINES`	Number of screen lines to clear. `0<LINES<25-ROW`.
`MODE`	Clearing method:

`0`	Top down.
`1`	Bottom up.
`2`	Collapse to center.

Returns:

None.

`clrtxt` may be used to progressively blank screen rows in any selected area of a display (the "region"). Upon completion of the operation the cursor is positioned at `(0,ROW)`.

clrwnstk — Clear Window Stack

Syntax:

```
CALL "clrwnstk"[,WN$]
```

Call Parameters:

WN$	Optional list of window names that are not to be deleted, in WIN(GETLIST) format, with each window name padded to eight characters. See text.

Returns:

None.

clrwnstk examines the results of the WIN(GETLIST) directive, selects the main (full-screen) window if WN$ is absent or null and then deletes each window found in WIN(GETLIST). Windows whose names are included in the WN$ window list are not disturbed. The most recently created window that is left after this function has completed will become the active window. This function will not repaint the screen if the main window is the active window.

The format of WN$ must be like that returned by WIN(GETLIST), in that bytes 1 and 2 are interpreted as the binary number of 8 character window names that follow. However, the actual values of bytes 1 and 2 can be anything—clrwnstk does not interpret them.

Examples:

```
CALL "clrwnstk"
```

The above example deletes all windows and if necessary, repaints the screen with the main window.

```
WN$=WIN(GETLIST);
WINDOW CREATE (40,5,10,2) "NAME=NEWWIN";
...
CALL "clrwnstk",WN$
```

The above example creates an active window list in WN$ and then opens a new window named NEWWIN. Later, the call to clrwnstk with WN$ deletes only those windows that did not exist when the window list in WN$ was created, resulting in the deletion of NEWWIN.

`clsfiles` Close Files (channels) In List

Syntax:

```
CALL "clsfiles",F[ALL]
```

Call Parameters:

`F[ALL]`	Zero-based list of channel numbers to be closed. `F[ALL]` is derived from the `opnfiles` function described on page 183.

Returns:

`F[ALL]`	Deleted.

`clsfiles` closes only those channels listed in `F[ALL]`. The array is subsequently deleted from memory. It is permissible to call this function with a zero-`DIM`mensioned array. See `opnfiles` (page 183) for additional information.

This function has been obsoleted and should be replaced by `closeall` (page 78) *in new development*.

`color` Determine Terminal Color Capability

Syntax:

```
CALL "color"[,FLAG]
```

Call Parameters:

None.

Returns:

`FLAG`	`0`	Monochrome terminal.
	`1`	Color terminal.

`color` examines the memory-resident terminal parameters and indicates the presence or absence of color capability. In doing so, `color` also sets or clears a global variable named `color` which can be checked in user programs (it will be `0` for monochrome or `1` for color). The ability to determine terminal color capability allows a program to alter its behavior to take advantage of color displays.

copyfmt — Copy Data Format to Data Format

Syntax:

```
CALL "copyfmt",SOURCE1$[+SOURCE2$[+SOURCE3$[...]]],DEST$,FLAG
```

Call Parameters:

SOURCEn$	Format name(s) from which data are to be copied in #LLNNNNNN style. Multiple source data formats may be specified by concatenation of the data names. See text.
DEST$	Destination format name that is to receive the copies in #LLNNNNNN style.
FLAG	Preparatory operation flag:

0	No preparatory operation.
1	Initialize the destination format.
2	Default the destination format.

Returns:

FLAG	0	Copy successful.
	1	Source format name invalid.
	2	Destination format name invalid.
	3	Destination format name identical to a source format name.

The value returned in FLAG is also returned in the ERR system variable. This will not cause an execution error in the calling program.

copyfmt copies the contents of the data format(s) named in SOURCEn$ to the data format named in DEST$ with no effect on the source format(s). It produces a result like that attained with the #LLNNNNND = #LLNNNNNS hard coded assignment statement but without the need to hard code the format names. Only elements with identical names and compatible attributes will be copied. Multiple source format names may be concatenated for the purpose of mass copying data from multiple sources into a single destination. This produces a result like that of a #LLNNNNND = #LLNNNNN1 & #LLNNNNN2 & LLNNNNN3... hard coded assignment statement. copyfmt will INCLUDE any formats not already in memory.

Examples:

```
SOURCE$="#ARTDHDR",
DESTINATION$="#ARTDDET",
FLAG=2;
```

```
CALL "copyfmt",SOURCE$,DESTINATION$,FLAG;
ON FLAG GOTO OK,BAD_SOURCE,BAD_DESTINATION,SAME_FORMAT
```

The above example initializes and defaults the data format #ARTDDET and then copies compatible elements from the data format #ARTDHDR to #ARTDDET. The FLAG value indicates the success or failure of the operation.

The next example illustrates the technique of loading a destination format with compatible elements from several source formats. In this instance, a sales order header is defaulted and then loaded from a customer master record, customer ship-to record and salesman master record.

```
CSMAS$="#OPCMMAS";              REM customer master
CSSHP$="#OPCMSHP";              REM customer ship-to
SPMAS$="#OPSPMAS";              REM salesman master
SOHDR$="#OPSOHDR";              REM sales order header
FORMAT INCLUDE #CMMAS$;
FORMAT INCLUDE #CMSHP$;
FORMAT INCLUDE #SPMAS$;
FORMAT INCLUDE #SOHDR$;
```

…load the source formats with the appropriate data…

```
FLAG = 2;
CALL "copyfmt",CMMAS$+CMSHP$+SPMAS$,SOHDR$,FLAG;
ON FLAG GOTO OK,BAD_SOURCE,BAD_DESTINATION,SAME_FORMAT
```

The above example, using soft coded formats, produces the equivalent of the following hard coded statements:

```
FORMAT INCLUDE #OPCMMAS;
FORMAT INCLUDE #OPCMSHP;
FORMAT INCLUDE #OPSPMAS;
```

…load the source formats with the appropriate data…

```
FORMAT INCLUDE #OPSOHDR ,OPT="DEFAULT";
#OPSOHDR = #OPCMMAS & #OPCSSHP & #OPSPMAS
```

Note that the order in which the source formats are copied is determined by the order in which they are passed to copyfmt. That is to say, in the above example, the #OPCMMAS format is copied to #OPSOHDR before #OPCSSHP, which is copied before #OPSPMAS. If a format later in the list has an identical element to one of the earlier formats, the later format's element data will overwrite data copied from a previous format's element.

Formats with identical element names of incompatible types will not be copied. Refer to the *Thoroughbred* programmer's reference manual for a detailed discussion of this type of format manipulation.

cprint — Center and Print Text String

Syntax:

```
CALL "cprint",STRING$,ROW
```

Call Parameters:

STRING$	Text to be displayed.
ROW	Screen row on which to display STRING$. 0<=ROW<25.

Returns:

None.

cprint clears row ROW and then displays the text in STRING$ centered on ROW, relative to the current window. Mnemonics in the text string are processed as expected and do not affect centering (unless a cursor positioning sequence is part of the string). The EP, DT and DB mnemonics, which affect text size, are recognized by this function and will work as expected on most terminals. Use caution with mnemonics that alter an area of the display (such as the CE mnemonic).

Example:

```
CALL "cprint",'BB'+"CUSTOMER NOT FOUND"+'EB'+'RB',20
```

This example centers and displays the flashing message CUSTOMER NOT FOUND on row 20 and rings the terminal bell.

See also the rprint function (page 222).

cseqnum — Generate Chronological Sequence Number String

Syntax:

```
CALL "cseqnum",SEQN$
```

Call Parameters:

SEQN$	Previously generated chronological sequence number, used to guarantee that a duplicate number will not be generated. See text.

Returns:

SEQN$	12 character, human-readable, hexadecimal number generated from the SQL (CDN) date and time of calling task. See text.

cseqnum creates a 12 character, human-readable, hexadecimal number from the calling task's current date and time, as produced in the CDN system variable. The resulting string may be used as a record key to assure that records remain in strict chronological order. A typical string generated by this function would be 42798FA22E00, corresponding to February 15, 2002 at 4:16:45.500192 PM. PRECISION 6 is used to assure the finest possible granularity.

Because cseqnum generates its output from the CDN variable—which isn't guaranteed to update more than a few times per second—it is possible for the same sequence number string to be generated if several calls are made in rapid succession. To compensate for this quirk, cseqnum can "look" at the most recently generated number and increment it as necessary to guarantee that the result is numerically higher than the last number. Simply pass the last value that was generated to cseqnum and it will make sure that the returned value is higher. The following example illustrates this process.

Example:

```
DIM S$[5];                  REM Array to store sequence strings
SEQN$="";
FOR I=1 TO 5;
   CALL "cseqnum",SEQN$;    REM Get a sequence string.
   S$[I]=SEQN$;
NEXT I;
```

In the above example, five chronological sequence number strings are stored in the S$[] array for later use. Because each call to cseqnum passes the previous value of SEQN$, five unique values are returned in ascending ASCII order.

cvtpwd — Convert Password To Encrypted Form

Syntax:

```
CALL "cvtpwd",PW$[,SALT$]
```

Call Parameters:

PW$	Clear text password, three to eight characters in length.
SALT$	Optional two character sequence that may be used to prime the encryption mechanism. If omitted or null cvtpwd will generate a random salt. See text.

Returns:

PW$	Encrypted password, constant 13 characters in length.
SALT$	If null when this function is called, will return a randomly generated salt. See text.

Error Returns:

ERR=46	String size error.

cvtpwd is a UNIX/Linux utility that generates an encrypted form of the variable length password string supplied in PW$. The result is a constant length string with 13 characters having no resemblance to the original password. Of the 13 characters, the first and second will be the value passed in SALT$ or a random salt value if SALT$ is omitted or is null on entry. The remaining 11 characters will be generated from the character set associated with SALT$, resulting one of 4,096 possible encryption methods (that is to say, by varying the salt, a password such as aBc123 can be encrypted in any of 4,096 different ways).

The most secure encrypted password is generated when the original is eight characters in length, is a mixture of numerals, upper and lower case letters, and punctuation characters, and a randomly generated salt is used. Such a password will cause cvtpwd to return an output string having a uniqueness probability of greater than 1 in 6×10^{15}. ERR=46 will occur if the supplied password is less than three or more than eight characters in length, or SALT$ is non-null and not exactly two characters in length. If a salt value is passed, it must be in the ASCII range of 33 to 126 (! to ~) inclusive.

cvtpwd makes a shell call to makekey, a system utility whose location varies from one version of UNIX to another.[8] For example, makekey is in the /usr/lbin subdirectory on an HP-UX machine but is found in /usr/lib on SCO UNIX or OpenServer 5. It is essential that the PATH environment variable established when *Thoroughbred* was started includes the subdirectory in which makekey is located or else this function will fail. makekey may not exist in some Linux distributions, requiring that source code be obtained and compiled on the target system. In this case, please contact ***BCS Technology Limited*** for assistance.

As described before, the salt value determines how the password will be encrypted. Therefore, you should call cvtpwd with a null salt value to encrypt a password for the first time, a procedure that will minimize the likelihood of two identical clear text passwords generating identical encrypted equivalents. Later, when a comparison to that password must be made, pass the salt value (which you can derive from the first and second characters of the encrypted password) to cvtpwd, along with the clear text password to be compared. Then compare the output from cvtpwd against the encrypted original value to determine if the proper password was entered.

Examples:

```
PW$="obvious";
CALL "cvtpwd",PW$
```

The above example encrypts obvious with a randomly generated salt. The salt may be derived from PW$(1,2).

```
PW$="obvious"
SALT$= "*%";
CALL "cvtpwd",PW$,SALT$;
PRINT QUO,"obvious",QUO," encrypts to ",QUO,PW$,QUO,"."
```

This example prints:

```
"obvious" encrypts to "*%eyf07MhNMq.".
```

[8]There is no equivalent to makekey in Windows, which uses a less secure hashing algorithm for passwords.

daterang — Generate SQL Date Range Values

Syntax:

```
CALL "daterang",RDAT,CMLO,CMHI,LMLO,LMHI,CYLO,CYHI,LYLO,LYHI
```

Call Parameters:

RDAT	Reference date in SQL (DTN) format from which remaining parameters will be computed. If RDAT=0 today's date as derived from the CDN system variable will be used. See text.

Returns:

CMLO,CMHI	SQL date values for the month represented by the reference date RDAT. CMLO is the first day of the month and CMHI is the last day of the month.
LMLO,LMHI	SQL date values for the month previous to the month represented by the reference date RDAT. LMLO is the first day of the month and LMHI is the last day of the month. Hence LMHI=CMLO-1.
CYLO,CYHI	SQL date values for the year represented by the reference date RDAT. CYLO is the first day of the year and CYHI is the last day of the year.
LYLO,LYHI	SQL date values for the year previous to the year represented by the reference date RDAT. LYLO is the first day of the year and LYHI is the last day of the year. Hence LYHI=CYLO-1.

daterang generates a set of SQL dates that are useful for processing data based upon an expected date range. The reference date RDAT from which all values are computed may be any SQL value and will be set to INT(CDN) if it is zero when the call is made.

Example

```
RDAT=DTN("072998","MMDDYY");
CALL "daterang",RDAT,CMLO,CMHI,LMLO,LMHI,CYLO,CYHI,LYLO,LYHI
```

In this example, the following values will be returned:

```
RDAT=729601                 (July 29, 1998)
CMLO=729573, CMHI=729603    (July 1, 1998 — July 31, 1998)
LMLO=729543, LMHI=729572    (June 1, 1998 — June 30, 1998)
CYLO=729392, CYHI=729756    (January 1, 1998 — December 31, 1998)
LYLO=729027, LYHI=729392    (January 1, 1997 — December 31, 1997)
```

dollars — Generate Verbose Dollars and Cents String

Syntax:

```
CALL "dollars",D,D$
```

Call Parameters:

D	Numeric dollar amount. 0 <= D < 10^9

Returns:

D$	Verbose dollars and cents string. See text.

dollars takes a numeric amount in D and generates a string that verbosely describes the numeric value of D as dollars and cents, as would be required to print a check. Error 40 (numeric value overflow) is returned if D is outside of the allowable range.

Examples:

```
D=21457.89;
CALL "dollars",D,D$;
PRINT D$
```

The above example will print:

```
Twenty-One Thousand Four Hundred Fifty-Seven and 89/100 Dollars
```

```
D=13.04;
CALL "dollars",D,D$;
PRINT D$
```

The above example will print:

```
Thirteen and 04/100 Dollars
```

```
D=.86;
CALL "dollars",D,D$;
PRINT D$
```

The above example will print:

```
Zero and 86/100 Dollars
```

dpycal — Display Windowed Calendar

Syntax:

```
CALL "dpycal",MON,YEAR,ROW,COL,WINAME$
```

Call Parameters:

MON	Calendar month. 0<=MON<13. See text.
YEAR	Calendar year. 1751<YEAR<10000 or YEAR=0. See text.
ROW,COL	Top left corner coordinates of window frame. 0<=ROW<15, 0<=COL<71. If COL=0 window will be centered.

Returns:

WINNAME$	Name assigned to the window created by this function.

dpycal generates a windowed calendar for month MON and year YEAR (YEAR must be a full year—YEAR=97 means the year is 97, not 1997). If both MON and YEAR are zero the current month and year are assumed. The window frame is anchored by the ROW and COL coordinates, which describe the location of the top left corner.

Example

```
CALL "dpycal",11,1997,5,0,CAL$
```

This example results in the following display starting on row 5, with the calendar centered on the screen:

```
   NOVEMBER 1997
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

The name used to create this window will be returned in the string variable CAL$ and may be used to collapse the window when it is no longer needed.

dpyhelp — Display On-Line 4GL Help Window

Syntax:

```
CALL "dpyhelp",H$,ROW
```

Call Parameters:

H$	Help screen name as defined in IDOL-IV. The correct format for H$ is LLNNNNNN, where LL is the IDOL-IV library prefix and NNNNNN is the help screen name, with a minimum of one character for the screen name. For example, H$="SSDESCRP".
ROW	Screen row on which to display prompts generated by this function. The bottom row of the screen is typical for most applications.

Returns:

None.

dpyhelp provides a convenient interface to the IDOL-IV 4GL help system. In order to use this function you must utilize the IDOL-IV facilities to create a suitable help screen in window format (do not create a non-window version, as it will disrupt the underlying display). When called, dpyhelp will display the help screen and some prompts on row ROW indicating to the user how to utilize help. The window is collapsed and the original screen restored when the user exits help.

The inputfmt function (page 148) includes help calls that the user can activate by pressing **[CTRL][O]**. The format into which input is being directed must have a defined documentation code and a related help screen for this feature to be useful. The fkydcd, getkey and input{dec}{ned} functions do not have this interface but will return a flag value for **[CTRL][O]** to the calling program, indicating that the user has requested help. Should this flag value be returned, your program must be prepared to act upon it.

dropall — Drop All Public Programs

Syntax:

```
CALL "dropall"[,PB$]
```

Call Parameters:

PB$	Optional list of public programs that are to remain memory resident, in standard PUB(1) system variable format. Omit if all public programs are to be dropped.

Returns:

None.

dropall examines the PUB(1) system variable for a list of public programs made memory resident with the ADDR directive and then drops each one. dropall does not drop any program whose corresponding 12 byte entry is found in the optional PB$ parameter. See *Thoroughbred*'s description of the PUB(1) system variable for additional information.

Examples:

```
CALL "dropall"
```

The above example drops all public programs made memory resident with the ADDR directive.

```
ADDR "program1";
PB$=PUB(1);
ADDR "program2";
ADDR "program3";
…do some processing…
CALL "dropall",PB$
```

In the above example, the call to dropall drops only program2 and program3, as program1 was already resident when PB$ was defined.

drvsline — Drive Bottom Terminal Status Line

Syntax:

```
CALL "drvsline",[TEXT$]
```

Call Parameters:

TEXT$	Text to be written to status line. If null or absent the status line will be turned off. Text in excess of the physical screen width will be truncated.

Returns:

None.

Prerequisites: The mnemonics WB, WD and WL must be defined, indicating that the terminal hardware has a bottom status line capability. See text.

drvsline provides a portable mechanism for displaying text on the bottom status line of a terminal. At least one printable character must be passed in TEXT$ to enable the status line (a blank is considered printable by drvsline and will result in an empty but visible status line on most terminals). Otherwise it will be turned off. The WB, WD and WL mnemonics provide the code needed to control the status line. All three must be defined in the TCONFIGW table for the terminal and all three must be non-null. If any one of these mnemonics has not been defined drvsline will exit with no error.

When the status line has been enabled it will be displayed in background reverse video on terminals that are able to support status line display attributes. The display will encompass the full screen width even if the string passed in TEXT$ is less than the full screen size.

Example:

```
CALL "drvsline","This is a test!"
```

The above example will display:

```
This is a test!
```

on most terminals that support a bottom status line.

dstrlen — Compute Display String Length

Syntax:

```
CALL "dstrlen",STRING$,LENGTH
```

Call Parameters:

STRING$	Text string whose length is to be computed.

Returns:

LENGTH	Computed display length in printable characters.

dstrlen computes the display length of a string in which mnemonics may have been embedded. The value returned in LENGTH represents the amount of space that would be occupied on a display device by the ASCII characters in the string. Each instance of whitespace will be counted as one character. Embedded mnemonics in the string are ignored, except for 'DB', 'DEON', 'DT', 'EP' and 'EXON'. If any of these mnemonics are found anywhere in the string dstrlen will double the computed length.

Examples:

```
STRING$='SF'+"This is a test."+'CL';
CALL "dstrlen",STRING$,LENGTH
```

The above example results in LENGTH=15. Note that the actual length of STRING$ is 21 bytes due to the presence of the 'SF' and 'CL' mnemonics.

```
STRING$='EP'+'SF'+"This is a test."+'CL';
CALL "dstrlen",STRING$,LENGTH
```

The above example results in LENGTH=30, due to the presence of the 'EP' mnemonic, which doubles the display width on most terminals and printers.

See also the lmargin and rmargin functions on pages 156 and 221 respectively.

duedate — Get A Due Date

Syntax:

```
CALL "duedate",WR,ER,P$,T,DT,FLAG
```

Call Parameters:

WR	Top row of three month calendar window display. See text.
ER	Row on which user's data entry will occur.
P$	Optional prompt. If null, `Due Date (MMDD or ESC):` will be substituted as a default prompt.
T	Input timeout in seconds. Zero disables timeout.
DT	Default date value in SQL (`DTN`) format. The month/day equivalent of this value will be the default input.

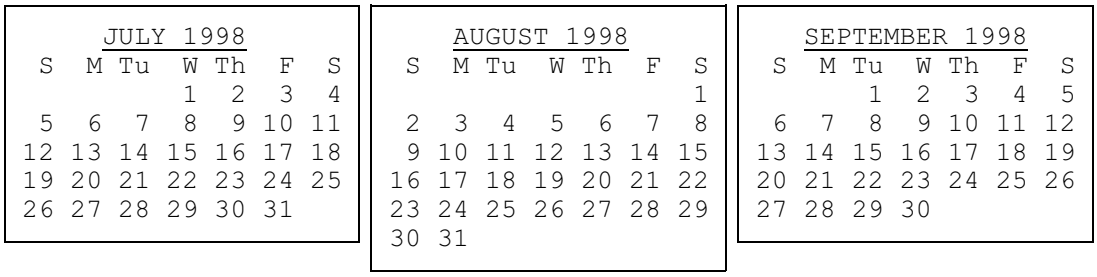
Returns:

DT	User-entered due date in SQL date format, or unchanged if FLAG>0 (see below).
FLAG	0 OK, `DT` is valid. 1 User aborted with **[ESC]**. 2 Timed out.

`duedate` provides a convenient (for the user as well as the programmer) method of accepting a "due date" in any program that requires one. When called, `duedate` displays a windowed three month reference calendar, with the first month being the current one as defined by the system `CDN` variable, and then prompts the user to enter a date in `MMDD` format (no year required). The calendar window begins on the row passed in `WR` and may extend down to row `WR+9`, depending on the months involved. Therefore, `ER` should be at least `WR+10` to avoid having the entry area overlap the calendar.

`duedate` assumes that if the entered month is earlier (numerically lower) than the current month or the month is the same as the current month and the entered day is earlier than the current day then the due date is in the following year. For example, if today's date as derived from `CDN` is `July 28, 1998` and the user enters **`0727`** `duedate` will take it to mean `July 27, 1999`. Hence `duedate` accepts dates up to one year into the future. Note that it is not possible to enter a date prior to today's date.

A typical `duedate` call appears as follows:



```
      JULY 1998
 S  M Tu  W Th  F  S
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

```
    AUGUST 1998
 S  M Tu  W Th  F  S
                  1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

```
   SEPTEMBER 1998
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

```
Due Date (MMDD or ESC): 0728
```

The above example assumes the current date is `July 28, 1998`.

elements — Generate Format Element Variable Assignment Merge Code

Syntax:

```
CALL "elements",FORMAT$,FMTVAR$,PRFX$,LINE
```

Call Parameters:

FORMAT$	Data format name in #LLNNNNNN style.
FMTVAR$	3GL variable name in main program to which the format name in FORMAT$ has been assigned or may be null. See text.
PRFX$	Element variable prefix. A null prefix will cause a runtime error in the main program. See text.
LINE	Main program line number to which code is to be merged. See text.

Returns:

ERR	Any execution error, such as undefined format.

Prerequisites: The FNELM defined function (page 37) must be present in the main program.

elements provides a mechanism for automatically generating format element numeric assignments in a running program. Instead of hard-coding element numbers into program format references, such as PRINT FMT(FORMAT$,4), which results in non-portable code, you can call elements to generate numeric variables for use in place of element numbers. Thus, if the element referenced as element 4 in the previous example is moved in the format, elements will detect the change and produce the correct numeric assignments at runtime.

When called, elements will either create a new program line in the main program, with the line number LINE, or will append its code to an existing line with the same line number. If the value passed in FMTVAR$ is null the quoted literal format name will be used in the generated program statements, resulting in slower execution and possible portability problems.

Example:

```
00200 PAGHDR$="#GCPAGHDR",
      FMTVAR$="PAGHDR$",
      PRFX="PH",
      LINE=TCB(4)+1;
      CALL "elements",PAGHDR$,FMTVAR$,PRFX$,LINE
00210 ...program continues...
```

The above call will cause `elements` to generate line 201 in the main program with the element assignments, as follows:[9]

```
00201 PH_CODE=FNELM(PAGHDR$,"CODE"),
      PH_CPL=FNELM(PAGHDR$,"CPL"),
      PH_DATE=FNELM(PAGHDR$,"DATE"),
      PH_DOCNUM=FNELM(PAGHDR$,"DOCNUM"),
      PH_LP=FNELM(PAGHDR$,"LP"),
      PH_PAGE=FNELM(PAGHDR$,"PAGE"),
      PH_PROGRAM=FNELM(PAGHDR$,"PROGRAM"),
      PH_SECTION=FNELM(PAGHDR$,"SECTION"),
      PH_SUBTITLE=FNELM(PAGHDR$,"SUBTITLE"),
      PH_TITLE=FNELM(PAGHDR$,"TITLE")
```

When line `201` is executed, the numeric variables prefixed with `PH_` will receive assignments based upon the positions of the elements in the `#GCPAGHDR` format. This will result in `PH_CODE=1`, `PH_CPL=2`, and so forth. Thus, instead of using `FMT(PAGHDR$,4)` to extract the documentation data from the `#GCPAGHDR` format you can use `FMT(PAGHDR$,PH_DOCNUM)`, a much more portable method of referencing individual elements. Should the structure of `#GCPAGHDR` be changed in the future the variable assignments will likewise change, and your program will still work.

If the variable `FMTVAR$` in the above example were null the generated statements would appear as follows:

```
00201 PH_CODE=FNELM("#GCPAGHDR","CODE"),
      PH_CPL=FNELM("#GCPAGHDR","CPL"),
      PH_DATE=FNELM("#GCPAGHDR","DATE"),
      PH_DOCNUM=FNELM("#GCPAGHDR","DOCNUM"),
      PH_LP=FNELM("#GCPAGHDR","LP"),
      PH_PAGE=FNELM("#GCPAGHDR","PAGE"),
      PH_PROGRAM=FNELM("#GCPAGHDR","PROGRAM"),
      PH_SECTION=FNELM("#GCPAGHDR","SECTION"),
      PH_SUBTITLE=FNELM("#GCPAGHDR","SUBTITLE"),
      PH_TITLE=FNELM("#GCPAGHDR","TITLE")
```

Such coding will result in somewhat slower execution and may cause portability problems in public programs (see application notes below). Note that it is permissible to call `elements` from console mode, a technique which may be employed to build code into a program that will be `CALL`ed rather than `RUN`.

---

[9]Note that it is not possible to pass the TCB(4)+1 expression in the call list, as the value of TCB(4) will reflect the line number at which the ENTER statement in elements is executed, not the line number at which the CALL was executed.

APPLICATION NOTES

As described above, `elements` works by adding new code to the calling program. The added code consists of a series of assignment statements, resulting in the expansion of the program line to which they are added. Hence it is possible for an error to occur if the number of assignment statements is too great, as *Thoroughbred* has an upper limit on the number of variable assignments that can be compiled in a single line of code. Therefore, if you have many calls to `elements` in your program and the formats involved have many elements in their definitions it would be wise to use several line numbers to add the generated code.

`elements` cannot be used to modify any encrypted program or within a public program to affect the public program. In such cases, it is necessary to manually execute `elements` during program development to add the required assignment statements. Be sure to use the full call syntax for maximum portability. For example:

```
CALL "elements",PAGHDR$,"PAGHDR$","PH",LINE
```

is preferable to:

```
CALL "elements",PAGHDR$,"","PH",LINE
```

as the second example will effectively hard code the format name into the program. Should the format later be renamed within IDOL-IV a runtime error will occur.

`erasetmp` Erase Temporary Files

Syntax:

```
CALL "erasetmp"
```

Call Parameters:

None.

Returns:

None.

`erasetmp` closes and erases all temporary files created by the calling task in the directory pointed to by the UNIX or Linux `TMPDIR` environment variable (see the `tempdir` function on page 240). `erasetmp` identifies a task's temporary files by a `.TTT` filename suffix, where `.TTT` is the numeric task identification number derived from the `idport` function (page 132).

esc — Program Halt Screen Conditioner

Syntax:

```
CALL "esc"
```

Call Parameters:

None.

Returns:

None.

`esc` brings the terminal display to a condition of sanity suitable for a program halt (such as due to an `ESCAPE` or error). `esc` clears the window stack, enables the cursor, disables display features such as blinking and reverse video and restores the terminal driver to a default state. If your program halts with an error and leaves the display in a less than optimal state you should call this function to restore the display to a workable condition. *Note that* `esc` *does not cause a program* `ESCAPE`.

`fillflds` Fill Format Elements with Forms Alignment Patterns

Syntax:

```
CALL "fillflds",FORMAT$
```

Call Parameters:

`FORMAT$` Data format name to be filled in `#LLNNNNNN` style.

Returns:

None.

`fillflds` fills the elements of the data format name specified in `FORMAT$` with alignment pattern data according to the data type for each element. Elements with no special attributes are populated with multiple `X`'s. Elements with date attributes are set to today's date and time (if the field allows the time). Numeric elements are loaded with a number appropriate for the extent of the field (as defined by the mask value for that element—see `ATR(FMT$,ELEMENT,25)` for an element's mask definition). The format is `INCLUDE`d and initialized before being filled.

Example:

```
CALL "fillflds","#XOCXOIH"
```

This call fills all elements in the data format `#XOCXOIH` with a pattern appropriate to each element type. `#XOCXOIH` is `INCLUDE`d and initialized.

fixcaps — Capitalize Leading Characters/Strip Blanks/Pad Length

Syntax:

```
CALL "fixcaps",STRING$[,LENGTH]
            or...
CALL "fixcaps",FMT$,ELM[,OCC]
```

Call Parameters:

STRING$	Character string to be processed.
LENGTH	Desired string length. If omitted LENGTH is assumed to be the length of STRING$.
or...	
FMT$	Data format name in #LLNNNNNN style.
ELM	Data format element number (see caution below).
OCC	Data format element occurrence if the data format element ELM has multiple occurrences. This parameter should be zero or omitted if element ELM is a single occurrence element.

Returns:

STRING$	Processed if passed as a character string, unchanged if a format reference.

In the first form of the call, fixcaps capitalizes the leading character in each word of STRING$ and truncates or pads the results to the length specified by LENGTH. If LENGTH is zero STRING$ is neither padded or truncated. Otherwise, STRING$ is returned at length LENGTH, with blanks used to pad as required or trailing characters dropped. In all cases, multiple blanks between words are reduced to a single blank.

In the second form of the call, fixcaps capitalizes the leading character in each word found in occurrence OCC of element ELM in format FMT$. No length adjustment is involved, as the element attributes determine the string length. Multiple blanks between words are reduced to a single blank.

Caution: fixcaps does no data type checking of format elements. Calling fixcaps with ELM set to a non-string element (such as one defined as a four or six byte binary SQL date) will most likely result in corruption of that element and any files associated with the format.

For capitalization purposes, `fixcaps` defines a word as a contiguous string of characters terminated by a blank, period, comma, dash (–), apostrophe (`'`), quote (") or ampersand (`&`), or a contiguous string of characters bound by parentheses. These rules will properly capitalize the majority of English words, including such constructs as o'clock (`O'Clock`), joe's (`Joe's`), s&s (`S&S`), and so forth. There may be a few instances where incorrect capitalization will occur, an unfortunate byproduct of the capriciousness of English language spelling rules.

Examples:

```
STRING$="now is the time";
CALL "fixcaps",STRING$
```

`STRING$` returns `Now Is The Time.`

```
STRING$="now is the time",
LENGTH=10;
CALL "fixcaps",STRING$,LENGTH
```

`STRING$` returns `Now Is The.`

```
FT$="#OPCMMAS",
ELM=1;
FORMAT INCLUDE #FT$,OPT="INIT";
LET FMT(FT$,ELM)="1234 w o'keefe rd";
CALL "fixcaps",FT$,ELM
```

In the above example, the data format `#OPCMMAS` is `INCLUDE`d and initialized, and the first element is loaded with the character string `1234 w o'keefe rd`. Upon return from `fixcaps` that element will contain `1234 W O'Keefe Rd`. No length adjustment is made, as the size attribute for the element is utilized by `fixcaps`. Note that the `OCC` value can be omitted if the selected data element is not defined for multiple occurrences.

fkydcd — Detect and Decode Control Keypress

Syntax:

```
CALL "fkydcd"[,TIMOUT[,MODE]]
```

Call Parameters:

TIMOUT		No response time out period in seconds. Disabled if zero or omitted.
MODE	0	Turn off cursor (default).
	1	Turn on cursor (optional).

Returns:

ERR — Detected key exit code (suggested variable names are on right margin):

0	**[RETURN]** or **[ENTER]**	K_RT
1-12	**[F1]** - **[F12]**	K_F1 – K_F12
13	Cursor right	K_CR
14	Cursor left	K_CL
15	Cursor down	K_CD
16	Cursor up	K_CU
17	**[BACK SPACE]**	K_BS
18	**[DELETE CHAR]**	K_DC
19	**[INSERT CHAR]**	K_IC
20	**[INSERT LINE]**	K_IL
21	**[DELETE LINE]**	K_DL
22	**[ERASE LINE]**	K_EL
23	**[ERASE PAGE]**	K_EP
24	**[TAB]**	K_HT
25	**[SHIFT][TAB]** (backtab)	K_BT
26	**[HOME]**	K_CH
27	**[CTRL][P]** or **[PRINT]** (see text)	K_SP
28	**[PAGE DOWN]**	K_PD
29	**[PAGE UP]**	K_PU
30	**[CTRL][O]** (help request)	K_HR
31	**[ESC]** ('EK' or abort)	K_ESC
32	Timed out	K_TIM
127	BASIC escape detected (e.g., **[CTRL][X]**)	

fkydcd is a general purpose control keypress detection function which returns to the calling program when the user presses one of the above-listed control keys or when a no input timeout occurs. A numeric exit value indicating which key was pressed is returned via the system ERR variable, making it possible to utilize ON ERR...GOTO... to conveniently route program execution. Alternatively, you can test for discrete values of ERR. *The setting of the* ERR *system variable by* fkydcd *does not cause an execution error in the calling program.*

In order to detect K_ESC the mnemonic 'EK' must be defined in the terminal driver table (TCONFIG8 or TCONFIGW) as a single character and a terminal key capable of emitting the 'EK' character must be pressed. If no 'EK' definition exists for the terminal the default value 03 (**[CTRL][C]** or <ETX>) will be assumed.

ERR=127 will be generated if the user presses the BASIC escape key, as would be done to interrupt a program. The ASCII value of that key is defined in the terminal driver table and is usually defined as 18 (**[CTRL][X]** or <CAN>). Trapping for this value allows you to perform debugging on a running program awaiting user input.

fkydcd normally turns off the cursor while awaiting input. You can override this feature by passing the optional MODE value. MODE=1 will cause fkydcd to enable the cursor until an exit condition occurs, at which time the cursor will be turned off.

Examples:

```
01000 MAIN: PRINT "F1 Continue, F3 Restart, ESC Abort: ",
01010 LOOP: CALL "fkydcd",60;
            IF ERR=127
              GOTO HALT
            FI;
            ON ERR(K_F1,K_F3,K_ESC,K_TIM) GOTO LOOP,CONT,RST,ABT
02000 CONT: REM "[F1] detected"
03000 RST:  REM "[F3] detected"
09000 ABT:  REM "[ESC] or timeout detected"
63900 HALT: ESCAPE; REM "program interrupted"
```

The above example illustrates both the ON ERR...GOTO... and discrete value methods of processing the result of a call to fkydcd.

```
01000 MAIN: PRINT "ESC To Abort: ",;
            CALL "seterr",0;
            WHILE ERR<K_ESC OR ERR>K_TIM;
             CALL "fkydcd",60,1;
            WEND
```

The above example idles in a `WHILE/WEND` loop until the user presses **[ESC]** or a timeout occurs. All other keypresses are ignored. The call to `seterr` (page 234) prior to entering the loop assures that `ERR` is not set to one of the two acceptable exit values due to a previous error condition in the program. Also, note that the cursor has been enabled in this example.

If you wish to detect a `QWERTY` keypress along with control or editing keys use the `getkey` function (page 124), whose operation is similar to `fkydcd`.

frmtdel — Delete Dictionary-IV Format Definition

Syntax:

```
CALL "frmtdel",FRMT$
```

Call Parameters:

FRMT$		Format name to be deleted in LLNNNNNN style. Do not prepend the name with an octothorpe (#).

Returns:

ERR	0	OK, format deleted.
	1	Format not defined.
	2	Format locked by another task.
	3	Unable to open data dictionary.

The above ERR values will not cause an execution error in the calling program.

frmtdel deletes the named format in FRMT$ from the IDDBD data dictionary, causing subsequent attempts to INCLUDE the format to fail. frmtdel should be used to remove temporary formats created by the frmtgen (page 111) function. frmtdel has no effect on currently INCLUDEd formats.

See also frmtgen (page 111).

frmtgen — Generate Dictionary-IV Format Definition

Syntax:

```
CALL "frmtgen",NFMT$,ELM$[ALL]
```

Call Parameters:

NFMT$	Unique format name in LLNNNNNN style. *Do not prepend this name with an octothorpe* (#). LL should be an existing IDOL-IV library definition.
ELM$[ALL]	Data describing the format structure. See text.

Returns:

ERR	0	OK, format definition was generated and may be INCLUDEd in subsequent code.
	1	Invalid parameter(s) in element definition(s).
	2	Format exists.
	3	Invalid format name, must conform to the LLNNNNNN style and the library specified by LL must exist.

The above ERR values will not cause an execution error in the calling program. IDOL-IV API errors are internally trapped and converted to one of the above ERR values.

frmtgen creates new Dictionary-IV format definitions "on the fly" using some simple parameterized data that may generated during program runtime or by reading from a file. This gives the programmer the ability to create and use logical formats as temporary data structures for a wide variety of tasks, such as defining custom screens and reports, creating temporary file layouts or passing complex data into public programs. Upon completion of the task for which the format was created it may be deleted from the data dictionary with the frmtdel (page 110) function.

In order for frmtgen to create a format, your program must pass the new format name in NFMT$ and the element definition data in ELM$[]. A maximum of 255 elements may be defined, and various special attributes, such as valid values or pre-processing, may be associated with each element. The values passed in ELM$[1-n] are either element definitions, called primary definitions, or special attributes definitions that are to be associated with an element. Elements are generated in the order in which their primary definitions have been ordered in ELM$[].

The structure of ELM$[] is as follows:

ELM$[0] The format's description, such as Customer Master or Sales Order Header. If null, no description will be assigned. The maximum description length is 40 characters and should be limited to alphanumeric characters, blanks and punctuation. It is recommended that each format be given a reasonably terse description.

ELM$[1] The first element's primary definition, the structure of which is described below. Other than ELM$[0], this is the only required entry in the ELM$[] array and would result in a format with one element. ELM$[1] cannot be anything except a primary definition.

ELM$[2] This may be the next element's primary definition or may be a special attribute to be applied to the element defined in ELM$[1].

ELM$[3] The basic sequence continues. If ELM$[2] is a primary definition, this could be another primary definition (thus defining the next element) or a special attribute to be applied to the element defined in ELM$[2]. Or, if ELM$[2] is a special attribute (which would be applied to the element defined in ELM$[1]), this could be another special attribute to be applied to the definition in ELM$[1]. This is the means by which multiple special attributes can be applied to a single element.

ELM$[n] Definitions continue up to a maximum of 255 primary definitions per format. Since up to eight special attributes can be applied to an element, ELM$[] could have as many as 2296 elements, including the description in ELM$[0].

A primary definition entry in ELM$[] is a comma-delimited list of fields in the form:

```
0,<name>,<size>,<fs>,<hs>,<ki>,<et>,<nt>,<pi>,<di>,<yn>
```

All fields must be present and are defined as follows:

0 Zero in the first field indicates to frmtgen that this is a primary definition. This value would be something other than zero if this was a special attribute definition (described below).

<name> The element's name, up to a maximum of 20 characters. The name will be internally converted to upper case. IDOL-IV element name rules apply.

`<size>` The element's size in bytes, which must be logical for the type of data to be stored. If the element is to be numeric, `<size>` must include a precision specification, such as `4.0` or `5.6`. If the element is to have multiple occurrences, the number of occurrences must be appended to the size. For example, `3.0*20` would be appropriate for a numeric element that is to be three bytes in size with zero precision and 20 occurrences. Similarly, `20*10` would be suitable for defining a 20 character string element with `10` occurrences. There should be no space between any characters in the size field.

`<fs>` Field separator indicator. If the element's data is to be followed by a field separator when written to a file, this field must be `1`. Otherwise, it must be `0`, indicating that no field separator should follow this element. In most cases, this field should be `0`, as it has no value except when the format is to be associated with a file whose structure includes field separators.

`<hs>` IDOL-IV help screen definition, six characters maximum or null. `frmtgen` will internally convert the help screen name to upper case. Do not include the two character library name in this parameter. That is to say, the help screen `OPPMACV` would be defined as `PMACV`., as the library will have been defined in the format name (e.g., `OPPMMAS`).

`<ki>` Key indicator. If this element is to be a record key or part of one, this field must be `1`. Otherwise, it must be `0`, indicating that this element is not a key or part of one. This field should always be `0` for logical formats, as it has no value except when the format is to be associated with a keyed file.

`<et>` Entry type. This field should be `0`, `1`, `2` or `3` according to the IDOL-IV data entry rules you wish to associate with this element. Refer to the IDOL-IV release notes for the meanings of these values.

`<nt>` Numeric type. This field must be `0-9` or `A-F` inclusive. Refer to the IDOL-IV release notes for the meanings of these values. If the numeric type is anything other than `0` the element size field must include a precision specification.

`<pi>` Pad indicator. This field should be `0-6` inclusive. Refer to the IDOL-IV release notes for the meanings of these values. If defined as `6`, a text field valid values special attribute definition must follow the primary definition.

<di> Date type indicator. This field should be 0-7 inclusive, with 0 indicating that this element is not a date. Refer to the IDOL-IV release notes for the meanings of these values. The size parameter must be appropriate for the date type, e.g., 6 for a binary SQL date/time field.

<yn> Yes/No indicator. This field should be 1 if this element is a Yes/No type or 0 if it is not. The element size must be 1 for a Yes/No element. Refer to the IDOL-IV release notes for the meanings of these values.

If any special attribute definitions are to be associated with a primary definition they must immediately follow the primary entry in ELM$[]. The general form of a special attribute definition is:

```
n,<SA>
```

where n defines the type of special attribute and <SA> contains the attribute details. Acceptable values for n are as follows:

1 Defines the valid values attribute and is required if the <pi> field in the primary definition is 6, indicating the element is a text field.

2 Defines the pre-set (default) element data value. The default value must be correct for the element type. For example, specifying a word as a default value for a numeric element would cause an error.

3 Defines the delete record process.

4 Defines the element access security rules attribute.

5 Defines the special prompt (message) attribute.

6 Defines the pre-process attribute.

7 Defines the post-process attribute.

8 Defines the audit rules attribute.

<SA> contains whatever information is needed to create the special attribute definition, the information being transferred verbatim into the special attribute. If <SA> is to include comma-delimited parameters it must be surrounded by quotes to prevent frmtgen from misinterpreting the commas as field delimiters.

For example, here is the correct way to define a pre-process attribute with comma-delimited fields in the <SA> portion, which in this example would be applied to the primary definition in ELM$[1]:

```
ELM$[2]="6,"+QUO+"mmssdpy,21,16,1"+QUO
```

frmtgen would see the above as an entry with two fields: 6, indicating that this is a pre-process special attribute, and mmssdpy,21,16,1, which would be embedded into the pre-process attribute space for the element. The following version of the above would be incorrect and will cause frmtgen to terminate with ERR=1:

```
ELM$[2]="6,mmssdpy,21,16,1"
```

frmtgen would see this as an entry having five fields, not two.

Special attribute definitions can be in any order following the associated primary definition. However, it is recommended that they be arranged in numeric order for ease of management.

Example:

The following code creates the format AR001SF with two elements named SUBTOTAL and TOTAL, both of which are numeric, the description Scratch Format, and INCLUDEs the new format for subsequent use. SUBTOTAL includes a pre-process definition:

```
01000 MAIN:     DIM ELM$[3];
                NFMT$="AR001SF",
                ELM$[0]="Scratch Format",
                ELM$[1]="0,SUBTOTAL,6.2,0,,0,0,A,1,0,0",
                ELM$[2]=""6,"+QUO+"mmssdpy,21,16,1"+QUO,
                ELM$[3]="0,TOTAL,7.2,0,,0,0,A,1,0,0";
                CALL "frmtgen",NFMT$,ELM$[ALL];
                ON ERR(0) GOTO ERROR,MAIN01

01010 MAIN01:   FORMAT INCLUDE #NFMT$;
…program continues…
```

Once AR001SF is no longer needed the following code could be executed:

```
08000 EOF:      FORMAT DELETE #NFMT$;
                CALL "frmtdel", NFMT$
```

removing the format from memory and deleting it from the IDOL-IV data dictionary.

See also frmtdel (page 110).

frmttext — Format and Justify Raw Text String

Syntax:

```
CALL "frmttext",RAW$,MAXLEN,MODE,LINE$[ALL]
```

Call Parameters:

RAW$	Character string to be processed.	
MAXLEN	Maximum formatted line length in characters. 9<MAXLEN<32767	
MODE	0	Text left justified with no line padding.
	1	Text left justified and padded to equal length lines.
	2	Text left justified and padded to MAXLEN length lines.
	3	Text fully justified and padded to MAXLEN length lines.

Returns:

MODE	Number of text lines generated.
LINE$[ALL]	Individual text lines, beginning with LINE$[1] and ending with LINE$[MODE].

Errors:

41	Invalid MODE or MAXLEN value.

The ERR *system variable is unchanged following a normal exit from this function.*

frmttext processes the unformatted or "raw" text string in RAW$ into one or more formatted paragraphs and generates lines of output in LINE$[ALL] suitable for driving screen or printer displays. The input string is scanned for whole words and appropriate adjustment is made to produce left- or full-justified output with no broken words. A zero length input string will not cause an error and returns no output.

The point at which a new line is to be started is normally determined by the value in MAXLEN—no line will ever exceed MAXLEN characters. It is also possible to force a new line by embedding an ASCII linefeed character ($0A$) into RAW$. Multiple consecutive linefeeds will produce multiple consecutive blank lines. A single final linefeed will be ignored.

frmttext defines a word as a contiguous string of non-blank characters based upon the ASCII character set.

A sentence is defined as a contiguous string of words terminated by a period, question mark or exclamation point, or one of these three punctuation marks followed by a single or double quote (e.g., !" or ?'). Formatting will cause adjacent words to be separated by a single blank, regardless of the amount of inter-word spacing present in RAW$, unless full justification mode is selected. Sentences on the same line will be separated by a minimum of two blanks—possibly more if full justification mode is selected.

Full justification is accomplished by increasing the spacing between words and sentences, working from the right hand end of the text line to the left hand end. frmttext will recursively adjust each line up to 10 times in an effort to achieve full justification. This algorithm will abort and result in improper formatting if an insufficient number of words exists in the line being processed. The final line of a paragraph will not be affected by the justification process but will be padded to MAXLEN length.

In the first and second examples that follow, it is assumed that an unformatted text string exists in the variable RAW$.

Examples:

```
MAXLEN=40,
MODE=0;
CALL "frmttext",RAW$,MAXLEN,MODE,LINE$[ALL];
FOR L=1 TO MODE;
   PRINT LINE$[L];
NEXT L
```

The above example generates formatted text lines and outputs them to the screen. Each line is stripped of all padding and limited to 40 characters maximum, producing a text array suitable for screen display.

```
MAXLEN=60,
MODE=0;
CALL "frmttext",RAW$,MAXLEN,MODE,LINE$[ALL];
LP=UNT;
OPEN (LP,OPT="INITTAB") "P1";
PRINT (LP,ERC=1) 'OPEN','PI10','NLQ';
FOR L=1 TO MODE;
   PRINT (LP)@(10),LINE$[L];
NEXT L;
PRINT (LP,ERC=1)'CLOSE',;
CLOSE (LP)
```

The above example generates formatted text lines and outputs them to printer P1 using near letter quality (NLQ) mode at 10 characters per inch, if available. Each line is stripped of all padding and limited to 60 characters maximum.

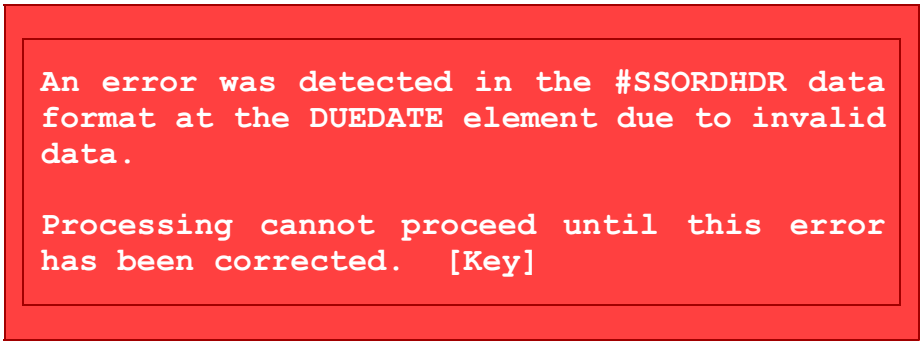
By printing at the tenth character position, a block of left-justified text nominally centered on 8½ inch wide paper will be produced. The `'CLOSE'` mnemonic, if defined in the IDOL-IV table for the selected printer, will eject the page and reset the printer to a default state.

The final example shows an application for `MODE=3` formatting, as well as the use of two linefeeds to break up the text.. In this case, an unformatted text string describing an error condition is displayed in a pop-up text box and held until the user presses ⏮ to continue.

```
RAW$="An error was detected in the #SSORDHDR data format at the
DUEDATE element due to invalid data."+$0A$+$0A$+
"Processing cannot proceed until this error has been corrected.
[Key]",
MAXLEN=40,
MODE=3;
CALL "frmttext",RAW$,MAXLEN,MODE,LINE$[ALL];
ROW=0,
COL$='BACKGR'+'WHITE'+'RED'+'RB';
CALL "msgbox",LINE$[ALL],COL$,ROW,2;
CALL "pause",0;
WINDOW DELETE (COL$);
```

…collapse window and continue processing

The user will see a text box similar to the following, vertically and horizontally centered on the screen, and the terminal bell will ring:



The text box will display with a red background and white foreground on terminals capable of color. Note how the linefeeds embedded into the raw text resulted in a blank line between the first and second sentences. The use of `MODE=3` causes all text lines to be padded to `MAXLEN` length and with right margins aligned, an effect that will force `msgbox` to size the text box to fit the text (see `msgbox` on page 176).

gentabs — Generate Printer Horizontal Tab Setup String

Syntax:

```
CALL "gentabs",LP,PITCH,TAB$,FLAG
```

Call Parameters:

LP	Open printer channel number.
PITCH	Current printer pitch in characters per inch.
TAB$	Numeric tab values in the form AAABBBCCC..., where AAA, BBB, CCC, etc., are the ASCII equivalents of the numeric TAB stops. See text.

Returns:

TAB$		Formatted tab setup string if successful. Unchanged if FLAG>0 (see text).
FLAG	0	Operation successful. TAB$ valid.
	1	Tab value out of range for current PITCH.
	2	Invalid TAB$ format on call.
	3	Invalid PITCH value.
	4	Invalid target output device. Either LP is not opened to a printer or the opened printer does not have a CPL mnemonic defined for it (see text).

gentabs processes the numeric data contained in TAB$ and produces a formatted setup string consisting of the printer's tab setup preamble, followed by the actual tab data with appropriate field separation and concluded with the tab setup postamble. It does this for the device opened on channel LP, assuming that the following are true:

- The device opened on channel LP is a printer.
- The printer's IDOL-IV driver script contains a CPL mnemonic. The CPL mnemonic defines the maximum possible characters per line at each of the five standard pitches (10, 12, 15, 17.125 and 20 characters per inch). It is in the form AAABBBCCCDDDEEE, where AAA is the value for 10 characters per inch, BBB is the value for 12 characters per inch, etc.
- The printer's driver script contains the following tab setup mnemonics: CHT, EHT, SHT, SHTSEP, SHTSIZ and SHTTYP. These mnemonics define the clear horizontal tabs, setup postamble and setup preamble escape sequences, followed by the tab field separator, tab field size in bytes, and the tab field data type (binary or ASCII), respectively.

- The PITCH value is either 10, 12, 15, 17 or 20. The PITCH value is used to determine the highest legal tab stop. For example, if PITCH=10, attempting to set a tab stop at column 200 would not be allowed, as no currently available printer can print 200 columns at 10 characters per inch. The actual limit is determined from the values associated with the CPL mnemonic.

In defining tab stops, the first column is always at least column one. Succeeding column values must be progressively higher. A TAB$ setup value such as 000020040060 will fail, as the first value will be interpreted as column zero, which is not allowed. Generally speaking, the highest allowed tab stop on most printers is 255 if the data type is binary, as that is all that can be passed in a single byte.

Using tabs to position data can result in faster printing in some situations. The usual method of columnar printing (e.g., PRINT (LP)@(C),"TEXT",@(C+20),"MORE TEXT"...) generates a lot of output, as positioning is accomplished by padding the line with numerous blanks. In contrast, a horizontal tab character occupies one byte but can move the print position many columns. Also, by using tabs it is possible to intermix font changes and other printer control sequences on a single line without causing column position errors. A sequence such as the following:

```
PRINT (LP)@(41),'SF',"DATA",@(70),'SB',"MOREDATA"
```

will not work as expected, as the escape sequences associated with the SF and SB mnemonics (which can be used to enable and disable bold face print) are mistakenly added to the character count, causing the "MOREDATA" column be misaligned.

Examples:

The following examples assume that the target printer is a wide model (132 to 136 characters per line in pica mode).

```
TAB$="005025050105",
PITCH=10,
LP=UNT;
OPEN (LP)"P1";
CALL "gentabs",LP,TAB$,PITCH,FLAG;
IF FLAG
   GOTO TAB_SETUP_ERROR
FI;
PRINT(LP)'PI10',TAB$,; REM Send setup string to printer.
PRINT(LP)'HT',COL_5$,'HT',COL_25$,'HT',COL_50$,'HT',COL_105$
```

The above example sets tab stops at columns 5, 25, 50 and 105, and then prints data at each column.

```
TAB$="015070125149",
PITCH=12,
LP=UNT;
OPEN (LP)"P1";
CALL "gentabs",LP,TAB$,PITCH,FLAG;
IF FLAG
   GOTO TAB_SETUP_ERROR
FI;
PRINT (LP)'PI12',TAB$,;
PRINT(LP)'HT',COL_15$,'HT',COL_70$,'HT',COL_125$,'HT',COL_149$
```

The above example sets tab stops at columns 15, 70, 125 and 149, with the pitch set to 12 characters per inch. The next example creates three tab setup strings for a form, one string for the form's header, another for the line items area and a third for the footer. The printer pitch is 10 characters per inch for all areas of the form:

```
HDRTAB$="005040055072",
LINTAB$="001006018043051069",
FTRTAB$="040068",
PITCH=10,
LP=UNT;
OPEN (LP)"P1";
CALL "gentabs",LP,HDRTAB$,PITCH,FLAG;
IF FLAG
   GOTO BAD_HDRTAB
FI;
CALL "gentabs",LP,LINTAB$,PITCH,FLAG;
IF FLAG
   GOTO BAD_LINTAB
FI;
CALL "gentabs",LP,FTRTAB$,PITCH,FLAG;
IF FLAG
   GOTO BAD_FTRTAB
FI;
PRINT (LP)'PI10',HDRTAB$,;
...print header data...
PRINT (LP)LINTAB$,;
...print line item data...
PRINT (LP)FTRTAB$,;
...print footer data...
PRINT (LP)'CLOSE',;
CLOSE (LP)
```

`getarecs` Get Number of Active Records In File

Syntax:

```
CALL "getarecs",FNAME$,NREC
```

Call Parameters:

`FNAME$`	File to be examined, must be a `DIRECT`, `INDEXED`, `ISAM`, `MSORT` or `SORT` file type.

Returns:

`NREC`	Number of active records, zero if none or `-1` if file type is not `DIRECT`, `INDEXED`, `ISAM`, `MSORT` or `SORT`.
`ERR`	`0` OK, `NREC` is valid. `1` File not found. `2` File locked by another task. `3` Corrupted file header.

The above `ERR` *values will not cause an execution error in the calling program.*

`getarecs` returns the number of active (i.e., in use) records in a `DIRECT`, `INDEXED`, `ISAM` or `MSORT` file in `NREC`. For other file types, `NREC` will return `-1`. The target file does not have to be open but must be accessible in the execution environment. Since `ISAM` and `MSORT` file types actually consist of two distinct files (index and data) be sure to specify the index filename (usually `<filename>.idx`), not the data filename (usually `<filename>.dat`).

getcpl — Get Characters per Line At Standard Printer Pitches

Syntax:

```
CALL "getcpl",LP$,FLAG,CPL[ALL]
```

Call Parameters:

LP$	Printer device name, such as P1. A channel does not have to be opened to the printer.

Returns:

FLAG	0	Operation successful. CPL[ALL] valid.
	1	Undefined printer.
	2	CPL mnemonic not defined for target printer (see text).
CPL[ALL]	Zero-based table of characters-per-line values at the five standard pitches: 10, 12, 15, 17 and 20 characters per inch, respectively.	

getcpl retrieves the characters-per-line data for printer LP$ and returns it into a numeric array. The CPL[ALL] data may be used to dynamically calculate a form layout according to the particular printer that is to print the form. In this way, each printer's width capabilities can be best used.

getcpl will fail if a CPL mnemonic has not been defined for the target printer or has been improperly arranged. The CPL mnemonic defines the maximum possible characters per line at each of the five standard pitches. It is in the form AAABBBCCCDDDEEE, where AAA is the value for 10 characters per inch, BBB is the value for 12 characters per inch…and EEE is the value for 20 characters per inch. This data can be gleaned from the technical manual for the printer. A typical CPL mnemonic for a wide printer would be 136163204233272. CPL for a "narrow" (standard width) printer would be 080096120137160. Note that numeric values under 100 are padded with leading zeros.

`getkey` Get Single Keypress

Syntax:

```
CALL "getkey",K,TIMOUT
```

Call Parameters:

`TIMOUT`	No input timeout period in seconds or zero for no timeout.

Returns:

`K`	ASCII value of detected QWERTY keypress or zero if keypress is a recognized control, function or editing key. See text.
`ERR`	Exit code for detected keypress or exit condition, valid only if `K=0`:

0	**[RETURN]** or **[ENTER]**	K_RT
1-12	**[F1]** - **[F12]**	K_F1 – K_F12
13	Cursor right	K_CR
14	Cursor left	K_CL
15	Cursor down	K_CD
16	Cursor up	K_CU
17	**[BACK SPACE]**	K_BS
18	**[DELETE CHAR]**	K_DC
19	**[INSERT CHAR]**	K_IC
20	**[INSERT LINE]**	K_IL
21	**[DELETE LINE]**	K_DL
22	**[ERASE LINE]**	K_EL
23	**[ERASE PAGE]**	K_EP
24	**[TAB]**	K_HT
25	**[SHIFT][TAB]** (backtab)	K_BT
26	**[HOME]**	K_CH
27	**[CTRL][P]** or **[PRINT]**	K_SP
28	**[PAGE DOWN]**	K_PD
29	**[PAGE UP]**	K_PU
30	**[CTRL][O]** (help request)	K_HR
31	**[ESC]** (`'EK'` or abort)	K_ESC
32	Timed out	K_TIM

getkey is a general purpose keypress detection primitive that returns to the calling program when the user presses a QWERTY key, one of the above-listed control keys, or when a no input timeout occurs. The pressing of a QWERTY key will always return a non-zero ASCII value in K (e.g., 65 or 41 for the letter A). If K=0 then a control, function or editing key has been pressed or a timeout has occurred and the ERR system variable will be appropriately conditioned, making it possible to utilize ON ERR...GOTO... to conveniently route program execution. Alternatively, you can test for discrete values of ERR. *The setting of the* ERR *system variable by* getkey *will not cause an execution error in the calling program.*

In order to detect ERR=31 the mnemonic 'EK' must be defined in the terminal driver table (TCONFIG8 or TCONFIGW) as a single character and a terminal key capable of emitting the 'EK' character must be pressed. If no 'EK' definition exists for the terminal the default value 03 ([CTRL][C] or <ETX>) will be assumed.

getkey does not perform any kind of terminal or display conditioning operations. Therefore, your code should position to the area on the screen where input is to be accepted, enable the cursor if required, unlock the keyboard if it has been locked and do what ever else is needed to manage the display.

Example:

```
01000 MAIN:    CALL "getkey",K,600;
               On K GOTO MAIN01,MAIN02
01010 MAIN01:  ON ERR(1,31,32)GOTO
                    MAIN,SPLCHK,ABORT,TIMOUT,HALT;
01020 MAIN02:  REM "Process QWERTY keypress."
02000 SPLCHK:  REM "[F1] detected."
09000 ABORT:   REM "[ESC] detected"
09100 TIMOUT:  REM "Timed out."
```

The above example expects the user to type a QWERTY key, **[F1]** or **[ESC]**, and will time out after 10 minutes of inactivity. It illustrates the ON ERR...GOTO... method of processing the result of a call to getkey. See also fkydcd (page 107).

`getpos` Get Current Cursor Coordinates

Syntax:

```
CALL "getpos",ROW,COL
```

Call Parameters:

None.

Returns:

`ROW,COL`	Zero based row and column cursor coordinates relative to the current window.

Error Returns:

`ERR=70`	Window driver not enabled.

`getpos` retrieves the current cursor coordinates based upon the extent of the current window. The calling task must have been started with the windows driver enabled. It is permissible for a ghost task to call this function without error.

getscrn — Get Physical Screen Contents

Syntax:

```
CALL "getscrn",TEXT$,ATTR$,WIDTH
```

Call Parameters:

None.

Returns:

TEXT$	Text portion of entire visible display, including blank areas.
ATTR$	Attribute map of entire screen. Attributes are bitwise values and are interpreted as follows:

```
Bit Clear/Set
_______________________________
 0  Background/foreground.
 1  Normal/reverse video.
 2  Normal/underlined.
 3  Static/flashing.
 4  Not used (always clear).
 5  Text mode/graphic mode.
 6  Not used (always clear).
 7  Not used (always clear).
_______________________________
```

WIDTH	Physical screen width in columns (e.g., 80 for a standard terminal).
ERR	Returns error 70 (windows driver not enabled) if a non-windowing terminal is in use.

getscrn is a convenient way to retrieve the visible screen contents into string variables for processing within a program. The attributes and physical width are returned so as to facilitate the conversion of the text component in TEXT$ into a reasonable facsimile of what the user sees. Unlike the 'TR' terminal read mnemonic, getscrn does not impose any I/O overhead on the host system, as it retrieves the screen data directly from the internal display map maintained by the *Thoroughbred* windows driver. The result is rapid screen retrieval with no effect on the terminal itself. *Thoroughbred* windows must be enabled for the task or else error 70 will be returned.

The data returned in both TEXT$ and ATTR$ consists of a contiguous series of bytes, one per screen position, in column major order. For a typical 80 column, 24 row display, TEXT$ and ATTR$ will each return 80*24 or 1920 bytes.

Any row of text or attribute data may be extracted from these strings with the expression `TEXT$(R*WIDTH+1,WIDTH)` or `ATTR$(R*WIDTH+1,WIDTH)`, where `R` is the row (`0<=R<STL(TEXT$)/WIDTH`). A character or attribute byte may be extracted with the expression `TEXT$(R*WIDTH+C+1,1)` or `ATTR$(R*WIDTH+C+1,1)`, where `C` is the column (`0<=C<WIDTH`) in which the character or attribute byte is located. Refer to the *Thoroughbred* windows directives for information on how to manipulate a screen image.

Example:

The following example retrieves a screen image, creates a set of string arrays for the text and attribute components and then writes the text to printer `P1`, producing a rudimentary screen dump.

```
CALL "getscrn",TEXT$,ATTR$,WIDTH;
HEIGHT=STL(TEXT$)/WIDTH;
DIM TXT$[HEIGHT-1],ATR$[HEIGHT-1];
FOR R=0 TO HEIGHT-1;
   TXT$[R]=TEXT$(R*WIDTH+1,WIDTH),
   ATR$[R]=ATTR$(R*WIDTH+1,WIDTH);
NEXT R;
OPEN (1) "P1";
FOR R=0 TO HEIGHT-1;
   PRINT (1)TXT$[R];
NEXT R;
CLOSE (1)
```

getxfd — Get Extended File Statistics

Syntax:

```
CALL "getxfd",CH,FSIZ,ATIM,MTIM,STIM,XF$
```

Call Parameters:

CH$	Channel OPENed to file.

Returns:

FSIZ	File size in bytes.
ATIM	Last access date and time in SQL (DTN) format.
MTIM	Last modification (write) date and time in SQL (DTN) format.
STIM	Last status change date and time in SQL (DTN) format.
XF$	Fully qualified path and filename.
ERR	0 OK, all returned parameters are valid.
	1 Invalid channel or channel not opened to a file.

getxfd is a convenient way to retrieve some extended data about an open file. The date and time values are returned with PRECISION 4 precision. The setting of the ERR system variable will not cause an execution in the calling program. If ERR=1 then all returns are invalid.

iddevice — Identify Open Device Type

Syntax:

```
CALL "iddevice",F,D
```

Call Parameters:

F	Channel number open to unknown device. 0<=F<32765	
Returns:		
F	Zero if an invalid channel number on call. Otherwise unchanged.	
D	0	Device not recognized or invalid channel number.
	1	Terminal.
	2	Disk or file.
	3	Printer.
	4	Ghost.

iddevice provides a convenient method of determining an open channel's device type. The standard *Thoroughbred* device type codes are not a linear progression. iddevice masks that aspect of the language.

Example:

```
CALL "iddevice",0,D
```

Upon return, D=1 if the calling task is a terminal or D=4 if the calling task is a ghost.

idfile Identify Open Filename

Syntax:

```
CALL "idfile",F,F$
```

Call Parameters:

F	Channel number open to unknown file. 0<=F<32765

Returns:

F	Zero if an invalid channel number on call. Otherwise unchanged.
F$	Filename associated with the open channel.

idfile provides a convenient method of determining the open filename associated with an open channel. All file types are recognized.

Examples:

```
CALL "idfile",0,F$
```

The above example accomplishes the same thing as F$=FID(0).

```
CALL "tempfid",F$;              REM Get temporary filename
CALL "tempdir",TMPDIR;          REM Get tmp directory number
SORT F$,KSIZ,NKEYS,TMPDIR,0; REM Generate temporary sort file
TF=UNT;
OPEN (TF)F$;
…process as required…
CALL "idfile",TF,F$;            REM Recall temporary filename
CLOSE (TF);
ERASE F$;                       REM Delete the temporary file
```

The above example shows a typical use for idfile, as well as several other file utilities. See also tempdir (page 240) and tempfid (page 242).

idport — Convert FID(0) To Numeric ID

Syntax:

```
CALL "idport",P
```

Call Parameters:

None.

Returns:

P	Numeric port ID: 0<=P<=965.

idport generates a zero-based numeric port identification number based upon the task's FID(0) value. For terminal tasks the value returned in P will be between 0 and 929 inclusive. Ghost tasks will return values between 930 (G0) and 965 (GZ) inclusive.[10]

Example:

```
CALL "idport",P
```

If the calling task is TC then P=12.

See also the FNPORT defined function.

[10]Due to a bug in level 8.4.0, *Thoroughbred* does not assign task IDs in the exact manner described in the programmer's reference manual, causing idport to skip some ID numbers. Following the assignment of task TZ, *Thoroughbred* incorrectly moves to U0 instead of Ta as documented. This problem appears to have been resolved in levels 8.4.1 and later.

input/inputned Get Keyboard Input String

Syntax:

```
CALL "input{ned}",COL,ROW,MX,MN,TIMOUT,CONFIG$,O$
```

Call Parameters:

COL,ROW — Zero-based column and row screen coordinates where the input field is to start. 0<=COL<80, 0<=ROW<24. *Special case*: if ROW+COL=0 then the current cursor position will be used.

MX,MN — Maximum and minimum number of characters the user may type. 0<MX<(ROW*80+COL), 0<=MN<=MX. MN=0 enables input default. Any non-zero value for MN will force the user to type at least MN characters before a ⏮ keypress will be accepted.

TIMOUT — No input timeout period in seconds, with TIMOUT=0 interpreted as timeout not enabled.

CONFIG$ — Input options configuration string. Each character in CONFIG$ defines a particular feature or characteristic of input{ned}. Characters 1-11 must be defined, while characters 12 and 13 are optional.

1,1 ASCLO: ASCII low character limit.
2,1 ASCHI: ASCII high character limit.
These values determine what part of the ASCII character range constitutes acceptable input. Permissible ranges are 31<ASCLO<=ASCHI<127. For example, a statement such as CONFIG$(1,2)="AZ" would set the permissible input to the upper case alphabet only.

3,1 input{ned} uses this value to indicate to the user the location and extent of the input field. For example, if you define MX=20 and CONFIG$(1,3)="." the user will see ..... where his input will appear. This field "fill pattern" is displayed in background video (the user's input will appear in foreground video). Any printable character in the ASCII character range is acceptable.

`4,1` This value determines if case conversion will be applied to alphabetic characters in the user's input. The permissible values are:

`0`	No conversion performed.
`1`	Convert to upper case.
`2`	Convert to lower case.
`3`	Initial capitalize. Text may be entered as a mix of lower and upper case letters, as well as non-alpha characters. When the user presses ⏮ `input{ned}` will capitalize the first letter of each word in the input string. See the narrative for the `fixcaps` function (page 105) for capitalization rules.

Case conversion for options `1` and `2` occurs before ASCII limit tests are applied. Therefore, a configuration such as `CONFIG$(1,2)="AZ"` and `CONFIG$(4,1)="2"` will result in no input being accepted, as all alpha input will be converted to lower case and thus will not fit into the `A` to `Z` range.

`5,1` This flag value determines how `input{ned}` will behave when the user presses **[ESC]**. Valid flag values are:

`0`	**[ESC]** keypress ignored.
`1`	**[ESC]** keypress returns a special termination code to calling program.

`6,1` This flag value determines if `input{ned}` will "echo" the user's typed input. Valid flag values are:

`0`	Echo enabled.
`1`	Echo disabled but cursor moves as user types.
`2`	Echo disabled and cursor does not move as user types.

Echo option `2` also inhibits generation of the input field location and extent pattern controlled by `CONFIG$(3,1)`—i.e., the user will be typing "blind."

`7,2` This flag value determines if input can be terminated by one of the terminal function keys. Valid flag values are:

00 Function keys not recognized as input terminators.
NN Highest function key number , where 00<NN<13. For example, if CONFIG$(7,2)="10" **[F1]** through **[F10]** will be recognized and **[F11]** and **[F12]** will be ignored.

9,1 This flag value determines if input{ned} can be terminated by pressing the **[TAB]** key (or **[CTRL][I]**). Valid flag values are:

0 **[TAB]** behaves as a conventional **[TAB]** key and when pressed, advances the cursor eight characters.
1 **[TAB]** keypress returns a special termination code to calling program.

10,1 This flag value determines if input{ned} can be terminated by pressing **[PRINT][SCRN]** or its equivalent. Valid values are:

0 **[PRINT][SCRN]** keypress ignored.
1 **[PRINT][SCRN]** keypress returns a special termination code to calling program.

11,1 This flag value determines input{ned}'s behavior when the user types to the full extent of the input field (i.e., user's input equals the MX value). Valid flag values are:

0 User must press ⏮ to process his/her input.
1 input{ned} will automatically terminate.

12,1 If present, input{ned} uses this value to pad the output string to the length specified by the MX value. If no padding is to be performed omit this character. Any character may be specified.

13,1 If present, input{ned} uses this value to determine how to pad the output string. Acceptable values are L (left justify), C (center) and R (right justify). If this value is omitted the default padding will be left justify.

O$ If input default has been enabled (i.e., MN=0) the content of O$ will appear in the input field as the default value. Otherwise, O$ will be ignored.

Returns:

`COL,ROW` If `COL+ROW` equaled zero when `input{ned}` was called the input field cursor coordinates will be returned in these variables.

`O$` User's typed input, padded with `CONFIG$(12,1)` (if present) and justified as directed with `CONFIG$(13,1)` (if present). Unchanged if user defaults or special termination occurs.

`MX` Number of characters typed by user or zero if user defaults or special termination occurs.

`MN` General exit code. If `MX>0` `MN` will hold the ASCII value of the first character in `O$`. If `MX=0` `input{ned}` has exited with a special termination condition, and `MN` will indicate the cause. Positive `MN` values are interpreted as follows:

0	Defaulted.
1-12	Terminated by function key <MN>.
13	Terminated by **[TAB]**.
14	Terminated by **[ESC]**.
15	Timed out.
16	**[PRINT][SCRN]** or equivalent pressed.
17	**[CTRL][O]** pressed (see text).

The above exit codes are also returned via the `ERR` system variable, permitting the use of `ON ERR(...) GOTO` to route program execution following an `input{ned}` call. *The setting of* `ERR` *will not cause an execution error in the calling program.*

Negative `MN` values represent trapped error conditions and are interpreted as follows:

1-11	`CONFIG$(ABS(MN),1+(MN=7))` invalid.
12	`COL` out of range.
13	`ROW` out of range.
14	`MX` out of range.
15	`MN` out of range.

If more than one defective value is detected in CONFIG$ the first one encountered will be the one flagged as an error. Untrapped errors return to the calling program in the system ERR variable and should be processed in the usual fashion.

APPLICATION NOTES

input{ned} provides a programmable keyboard interface suitable for a wide variety of data input tasks. input should be employed when an interactive editing input session is required. Use inputned for input tasks where little or no editing is required (such as numeric entry). In both cases, the user will be presented with an input field demarcated by the character in CONFIG$(3,1) (subject to echo control) and the cursor will be positioned at the left end (home position) of the field. If the expression (COL+MX)>79 is true the field will extend over multiple screen rows. If MN=0 and O$ is a non-null character string it will appear in the input field as the default input. These visible actions are the user's cue to begin typing.

Although the initial user interface appears to be the same in both input and inputned, keyboard response differs. input recognizes many of the terminal's text editing keys whereas inputned does not. Operational differences are described below in the inputned notes section.

In addition to the usual QWERTY keys, input recognizes the following editing and control keys. The key cap labels correspond to the Wyse WY-60 ASCII keyboard layout and may have slightly different names on other keyboards. Where possible, the equivalent PC-compatible keys have been mentioned (they will be in curly braces **{LIKE THIS}**).

⏮ ⏮ (**{ENTER}**) terminates input and returns the user's typed character string back to the calling program. The unused portion of the input field will be deleted from the screen and the cursor will be left at the COL/ROW coordinates.

[BACKSPACE] **Back space**. Back spacing deletes the character to the left of the cursor and shifts the input field in the expected manner.

[DEL] **Delete**. Pressing **[DEL]** (**{DELETE}**) deletes the character under the cursor and shifts the input field in the expected manner. The **[DEL CHAR]** key has the same effect.

[INS CHAR] **Insert mode toggle**. By default, `input` starts in overstrike mode. Pressing **[INS Char]** (**{INSERT}**) will enable insert mode and pressing it again will enable overstrike mode. While in insert mode a flashing reverse video arrow (>) will appear at the right end of the input field. Insert mode is automatically canceled if the input field becomes full or if all characters are deleted. Insert mode cannot be enabled if the cursor is positioned at the right end of the input field or if the input field is full.

[CLR LINE] **Clear to line end**. Pressing **[CLR LINE]** deletes all user input from the cursor position onward. If the cursor is at the home position the default input will be restored (if any).

[CLR SCRN] **Clear input field**. **[CLR SCRN]** is usually activated with the keypress combination **[SHIFT][CLR LINE]** on most terminals. Pressing **[CLR SCRN]** clears the entire input field and homes the cursor.

[◂] [▸] **Cursor left/right**. The **[◂]** and **[▸]** keys move the cursor horizontally in the input field.

[▴] [▾] **Cursor up/down**. If the input field extends over multiple rows the **[▴]** and **[▾]** keys will move the cursor vertically in the input field.

[HOME] **Cursor position toggle**. If the cursor is not at the home position pressing **[HOME]** will home it. If the cursor is homed **[HOME]** will move it to the right end of the input field if there is at least one character in the field.

[TAB] **Tab**. The behavior of **[TAB]** is controlled by `CONFIG$(9,1)`. See the call parameters comments above.

[SHIFT][TAB] **Backtab**. If the cursor is not in the home position **[SHIFT][TAB]** will move it eight characters toward the home position, producing the opposite effect of **[TAB]**.

[ESC] **Escape**. The behavior of **[ESC]** is controlled by `CONFIG$(5,1)`. See the call parameters comments above. Also, recall that the key designated as **[ESC]** is not necessarily the “escape” key.

[CTRL][O] The **[CTRL][O]** keypress combination is used to indicate that the user has requested on-line help. See the `dpyhelp` function (page 93) for details on implementing on-line help.

`input{ned}` will alert the user if s/he attempts to input too many characters, terminate input with ⏮ when less than `MN` characters have been typed or terminate input with a function key when that function key has not been enabled. Also, `input` will alert the user if s/he attempts to move the cursor beyond the boundaries of the active input field or fills the input buffer while insert mode is on.

`inputned` Notes

`inputned` is a restricted form of `input`, with the following user interface differences:

- Only **[BACK SPACE]** and **[DEL]** (or **[DEL CHAR]**) are recognized as text editing keys. **[Back Space]** operates the same as in `input`. However, **[DEL]** accomplishes the same thing as **[CLR SCRN]** in `input`. That is to say, pressing **[DEL]** will clear the entire input field and home the cursor.

- As soon as the user types a character in the home position the default input (if any) will be deleted from the input buffer and replaced with the new input. However, should the user back space to the home position, the default input will reappear.

`inputned` is best suited for numeric input sequences where the user would be expected to replace one number with another (e.g., a menu selection or a line item quantity), or for short alphanumeric input (e.g., `yes/no` queries or part number entries). In both of these cases, the tight control of `input` is retained but with a simplified user interface.

inputdat — Get Calendar Date Keyboard Input

Syntax:

```
CALL "inputdat",COL,ROW,MASK,DATE,TIMOUT,CONFIG$,O$
```

Call Parameters:

`COL,ROW` Zero-based column and row screen coordinates where the date input field is to start. `0<=COL<80`, `0<=ROW<24`. If `ROW+COL=0` then the current cursor position will be used.

`MASK` Numeric flag indicating which of three possible date masks will be used to qualify input and display the result.

MASK	QUALIFY	DISPLAY
1	MMDD	MM/DD
2	MMDDYY	MM/DD/YY
3	MMDDYYYY	MM/DD/YYYY

The actual input field size is equal to the display mask size, although the input is accepted and validated using the qualifying mask size. For example, if `MASK=2` the user will be expected to enter the date in `MMDDYY` format but the total screen space used will be eight characters, equal in length to the corresponding `MM/DD/YY` display mask.

`DATE` Default input date value in SQL (`DTN`) format or zero for no default date. If non-zero, the corresponding date, formatted according to the qualifying mask, will appear in the input field and the user will be allowed to default. Fractional content in the `DATE` value is permitted but will not be displayed or returned.

`TIMOUT` No input timeout period in seconds, with `TIMOUT=0` interpreted as timeout not enabled.

`CONFIG$` Input options configuration string. Each character in `CONFIG$` defines a particular feature or characteristic of `inputdat`. Bytes 1-5 must be defined, while byte 6 is reserved and may be omitted in the current version of `inputdat`. A literal string may be substituted for `CONFIG$`.

`1,1` This flag value determines how `inputdat` will react when the user presses **[ESC]**. Valid flag values are:

- `0` **[ESC]** keypress ignored.
- `1` **[ESC]** keypress returns a special termination code to calling program.

`2,2` This flag value determines if `inputdat` can be terminated by pressing one of the terminal function keys. Valid flag values are:

- `00` Function keys not recognized as input terminators.
- `NN` Highest function key number , where `00<NN<13`. For example, if `CONFIG$(2,2)="10"` **[F1]** through **[F10]** will be recognized and **[F11]** and **[F12]** will be ignored.

`4,1` This flag value determines if `inputdat` can be terminated by pressing **[PRINT][SCRN]** or its equivalent. Valid values are:

- `0` **[PRINT][SCRN]** keypress ignored.
- `1` **[PRINT][SCRN]** keypress returns a special termination code to calling program.

`5,1` This flag value determines if `inputdat` can be terminated by pressing the **[TAB]** key (or **[CTRL][I]**). Valid flag values are:

- `0` **[TAB]** keypress ignored.
- `1` **[TAB]** keypress returns a special termination code to calling program.

`6,1` Reserved. May be any value or omitted.

Returns:

`COL,ROW` If `ROW+COL` equaled zero when `inputdat` was called the input field cursor coordinates will be returned in these variables.

`DATE` SQL (`DTN`) equivalent to the user's entered date. See text for details on how `inputdat` generates this number for `MMDD` and `MMDDYY` input masks. Unchanged if user defaults or special termination occurs.

`O$` Display form of the entered date, formatted according to the `MASK` value supplied to `inputdat`. Unchanged if user defaults or special termination occurs.

`MASK` General exit code. Positive `MASK` values are interpreted as follows:

`0`	OK or defaulted.
`1-12`	Terminated by function key `<MASK>`.
`13`	Terminated by **[TAB]**.
`14`	Terminated by **[ESC]**.
`15`	Timed out.
`16`	**[PRINT][SCRN]** or equivalent pressed.
`17`	**[CTRL][O]** pressed (see text).

The above exit codes are also returned in the `ERR` system variable, permitting the use of `ON ERR(...) GOTO` to route program execution following an `inputdat` call. *The setting of* `ERR` *will not cause an execution error in the calling program.*

Negative `MASK` values represent trapped error conditions and are interpreted as follows:

`1-6`	`CONFIG$(ABS(MASK),1+(MASK=2))` invalid.
`7`	`COL` out of range.
`8`	`ROW` out of range.
`9`	`MASK` out of range.

If more than one defective value is detected in `CONFIG$` the first one encountered will be the one flagged as an error. Untrapped errors return to the calling program in the system `ERR` variable, causing an execution error, and should be processed in the usual fashion.

APPLICATION NOTES

`inputdat` provides a programmable keyboard interface suitable for accepting and qualifying calendar dates. By choosing the appropriate `MASK` value `inputdat` can accept one of three possible entry forms and automatically qualify the user's typed data, making appropriate adjustments to the high order component of the year. When the user presses ⏮ `inputdat` will display the entered date in the input field, using a display format based upon the `MASK` value.

Upon calling this function, the user will be presented with an input field demarcated either by the input form of the SQL date value passed in `DATE` or by the text form of the qualifying mask if `DATE=0`. For example, if `MASK=2` and `DATE=0` the user will see the following input field:

```
MMDDYY
```

The qualifying mask is displayed because there is no default date.

If `MASK=2` and `DATE=729662` (September 28, 1998) the user will see the following input field:

```
092898
```

In either case, the cursor will be positioned over the first character in the input field, indicating to the user that s/he should enter a date. `inputdat` accepts only the digits `0` through `9` and one or more of the following keypresses:

⏮	⏮ (**{ENTER}**) terminates input and returns the user's entered date back to the calling program. If possible, the text form of the date is displayed in the input field. Note that there is no separate default control: if you wish to receive a valid date when the user defaults you must supply one in the `DATE` variable when `inputdat` is called.
[BACKSPACE]	**Back space**. Back spacing deletes the character to the left of the cursor in the expected manner.
[DEL]	**Delete**. Pressing **[DEL]** (**{DELETE}**) deletes the entire input field and restores the default value if present The **[DEL CHAR]** and **[HOME]** keys have the same effect.
[◂]	**Cursor left**. **[◂]** has the same effect as **[BACKSPACE]**.
[TAB]	**Tab**. The behavior of **[TAB]** is controlled by `CONFIG$(5,1)`. See the call parameters comments above.
[ESC]	**Escape**. The behavior of **[ESC]** is controlled by `CONFIG$(1,1)`. See the call parameters comments above. Also, recall from the introduction that the key designated as **[ESC]** is not necessarily the "escape" key.
[SHIFT][TAB]	**Backtab**. Has the same effect as **[ESC]**.

[CTRL][O] The **[CTRL][O]** keypress combination is used to indicate that the user has requested on-line help. See the dpyhelp function (page 93) for details on implementing on-line help.

inputdat will alert the user if s/he attempts to input too many digits, terminate input with ⏮ when insufficient digits have been entered, enter an invalid date value, or terminate input with a function key when that function key has not been enabled.

inputdat includes processing to handle situations where no year or a partial year is entered as part of the date (MASK values 1 or 2). The basis for calculating the year is the current task date as returned by the CDN system variable. The general rules for the "incomplete" mask types are as follows:

- MMDD (MASK=1)

  inputdat assumes the year is the current year if the month value is greater than the current month, or the month value is the same as the current month and the day value is the same or greater. For example, if today's date is September 22, 1998 and the user enters 0922 inputdat will use 1998 as the year for generating the resulting SQL date number. Entering 1031 will also use 1998. If, on the other hand, the user enters 0921 inputdat will use 1999 as the year. Hence inputdat can generate a valid date up to one year into the future using the MMDD mask form.

- MMDDYY (MASK=2)

  inputdat uses the standard *Thoroughbred* "century date splitting" feature, in which YY values are calculated up to 50 years back from the current year or 49 years forward. For example, if today's date is September 22, 1998 and the user enters 010148, inputdat will use 1948 as the year for generating the resulting SQL date number. On the other hand, should the user enter 123147 inputdat will use 2047 as the year. Hence inputdat can generate a valid date up to 50 years in the past or 49 years into the future using the MMDDYY mask form. Note that the input value 010100 would convert to 730122, equivalent to January 1, 2000 and that the input value 022900 would be legitimate, as the year 2000 was a leap year.

Example:

```
01000 ENTDATE: PRINT 'CS',"Enter Date (ESC Abort): ",;
      DATE=CDN,
      MASK=2,
      TIMOUT=600;
      CALL "inputdat",0,0,MASK,DATE,TIMOUT,"100000",O$;
```

```
      ON ERR(0,14,15,16,17) GOTO ENTDATE,OK_HERE,ABORT,
                            TIMED_OUT,PRINT_SCRN,DISPLAY_HELP
01100 OK_HERE: …date entered, program continues…
```

In the above example, the user is prompted to enter a date in `MMDDYY` format and has ten minutes in which to respond. Today's date is the default date and the user may terminate with **[ESC]**. The `ON ERR(...) GOTO` method of program execution control is utilized following the call. Note that the `MASK` value could also have been used, but `ON ERR(...) GOTO` makes it easier to filter out unwanted exit returns.

inputdec — Get User Input In Decimal (Calculator) Format

Syntax:

```
CALL "inputdec",COL,ROW,ISIZ,FSIZ,TIMOUT,NFLAG,TFLAG,NUM$
```

Call Parameters:

COL,ROW		Input field coordinates. 0<=COL<80, 0<=ROW<24. If COL+ROW=0 the current cursor position will be used.
ISIZ		Number of places in integer part of input. 0<=ISIZ<15.
FSIZ		Number of places in fractional part of input. 0<FSIZ<15.
TIMOUT		No input timeout period in seconds; zero = no timeout.
NFLAG	0	Positive input only.
	1	Sign may be toggled. See remarks.
TFLAG	0	**[TAB]** ignored.
	1	**[TAB]** intercepted.

Returns:

COL,ROW		Input field start coordinates if COL+ROW=0 on call to this function.
ISIZ		Exit status:
	0	Normal exit; input terminated with ⏮.
	1	Input terminated with **[TAB]**, if enabled.
	2	Input terminated with **[ESC]**.
	3	Timed out.
	4	Input terminated with **[CTRL][O]** (see text).
NUM$		String representation of number typed by user, valid if ISIZ=0. Excess zeros are not stripped.
FSIZ		Number typed by user, valid if ISIZ=0. FSIZ returns the number with sufficient precision to account for the full fractional content in the user's input.

inputdec provides a convenient means for receiving floating point numeric input from the user. Its operation is much like that of a desk calculator. Upon calling this function, the user will see a static input field, with the decimal point appropriately positioned, an arrow (⬅) at the right end of the field and the cursor flashing at the position where the next typed digit will appear. As the user types digits, the entry field will shift left and the new digit will appear at the cursor position. For example, here is the input sequence as seen by the user when entering the value **123.45**:

```
   .00◄
   .01◄
   .12◄
  1.23◄
 12.34◄
123.45◄
```

If sign change has been enabled (`NFLAG=1`), the user may toggle the sign by pressing **[-]** (minus). The input field will change to reverse video when the sign has been toggled to minus. A single digit is deleted with **[BACK SPACE]**, which shifts the remainder of the input to the right. Pressing **[DEL]** clears the entire field. **[CTRL][O]** is used to indicate that the user has requested on-line help (see `dpyhelp` on page 93).

Example:

```
PRINT 'CS',"Enter Dollar Amount: ",;
R=0,
C=0,
ISIZ=4,
FSIZ=2,
TIMOUT=300;
CALL "inputdec",C,R,ISIZ,FSIZ,TIMOUT,1,1,N$;
IF ISIZ
   ON ISIZ-1 GOTO TABBED,ESCAPED,TIMEDOUT,HELP
FI;
```

…program continues…

This example prompts the user for a dollar amount and then configures `inputdec` to accept a maximum of `±9999.99` as input. The user is allowed to terminate input with **[TAB]** and has five minutes in which to respond.

inputfmt — Get Keyboard Input String, Direct to Data Format Element

Syntax:

```
CALL "inputfmt",COL,ROW,ELM,OCC,TIMOUT,CONFIG$,IFMT$
```

Call Parameters:

`COL,ROW` Zero-based column and row screen coordinates where the input field is to start. `0<=COL<80, 0<=ROW<24`. If `ROW+COL=0` then the current cursor position will be used.

`ELM` Element number in data format. `0<ELM<=NUM(ATR(IFMT$,0,0))`.

`OCC` Occurrence in element number `ELM`. Must be zero if the element specified in `ELM` is a single occurrence element.

`TIMOUT` No input timeout period in seconds, with `TIMOUT=0` interpreted as timeout not enabled.

`CONFIG$` Input options configuration string, 12 bytes in length. Each byte in `CONFIG$` controls a particular feature or characteristic of inputfmt.

`1,1` `ASCLO`: ASCII low character limit.

`2,1` `ASCHI`: ASCII high character limit.

These values determine what part of the ASCII character range constitutes acceptable input. Permissible ranges are `31<ASCLO<=ASCHI<127`. The best way to configure these values (assuming hard coded values) is with a statement such as `CONFIG$(1,2)="AZ"`. This would set the permissible input to the upper case alphabet only.

`3,1` `inputfmt` uses this value to indicate to the user the location and extent of the input field. For example, if the selected element's size is such that a maximum of 20 characters can be stored and `CONFIG$(3,1)="."` the user will see `....................` where his input will appear. This field "fill pattern" is displayed in background video (the user's input will appear in foreground video). Any printable character in the ASCII character range is acceptable.

`4,1` This value determines if case conversion will be applied to alphabetic characters in the user's input. The permissible values are:

0 No conversion performed.
1 Convert to upper case.
2 Convert to lower case.
3 Initial capitalize. Text may be entered as a mix of lower and upper case letters, as well as non-alpha characters. When the user presses ⏮ `inputfmt` will capitalize the first letter of each word in the input string. See the narrative for the `fixcaps` function (page 105) for capitalization rules.

Case conversion for options `1` and `2` occurs before ASCII limit tests are applied. Therefore, a configuration such as `CONFIG$(1,2)="AZ"` and `CONFIG$(4,1)="2"` will result in no input being accepted, as all alpha input will be converted to lower case and thus will not fit into the `A` to `Z` range.

`5,1` This flag value determines how `inputfmt` will react when the user presses **[ESC]**. Valid flag values are:

0 **[ESC]** keypress ignored.
1 **[ESC]** keypress returns a special termination code to calling program.

`6,1` This flag value determines if `inputfmt` will "echo" the user's typed input. Valid flag values are:

0 Echo enabled.
1 Echo disabled but cursor moves as user types.
2 Echo disabled and cursor does not move as user types.

Echo option `2` also inhibits generation of the input field location and extent pattern controlled by `CONFIG$(3,1)`—the user will be typing completely "blind."

`7,2` This flag value determines if `inputfmt` can be terminated by pressing one of the terminal function keys. Valid flag values are:

00 Function keys not recognized as input terminators.
NN Highest function key number , where "00<NN<13". For example, if CONFIG$(7,2)="10" **[F1]** through **[F10]** will be recognized and **[F11]** and **[F12]** will be ignored.

9,1 This flag value determines if inputfmt can be terminated by pressing the **[TAB]** key (or **[CTRL][I]**). Valid flag values are:

0 **[TAB]** behaves as a conventional **[TAB]** key and when pressed, advances the cursor eight characters.
1 **[TAB]** keypress returns a special termination code to calling program.

10,1 This flag value determines if inputfmt can be terminated by pressing **[PRINT][SCRN]** or its equivalent. Valid values are:

0 **[PRINT][SCRN]** keypress ignored.
1 **[PRINT][SCRN]** keypress returns a special termination code to calling program.

11,1 This flag value determines inputfmt's behavior when the user types to the full extent of the input field (i.e., user's input is equal to the maximum size defined for the element). Valid flag values are:

0 User must press ⏮ to process his/her input.
1 inputfmt will automatically terminate.

12,1 This flag value may be used to enable default when the element has no data and is configured to require a minimum amount of input (element input attribute types 2 and 3). Valid flag values are:

0 Normal. inputfmt enables or disables default according to the input type attribute of the element and whether default data is available.
1 Override. Default unconditionally enabled.

Refer to the applications notes for additional information on default control.

IFMT$ Data format name to be processed in #LLNNNNNN style.

Returns:

COL,ROW If ROW+COL equaled zero when inputfmt was called the input field cursor coordinates will be returned in these variables.

IFMT$ User's typed input stored in FMT(IFMT$,ELM,OCC). Unchanged if user defaults or special termination occurs.

ELM Number of characters typed by user or zero if user defaults or special termination occurs.

OCC General exit code. If ELM>0, OCC will hold the ASCII value of the first character in the input field. If ELM=0 inputfmt has exited with a special termination condition and OCC will indicate the cause. Positive OCC values are interpreted as follows:

0	Defaulted.
1-12	Terminated by function key <OCC>.
13	Terminated by **[TAB]**.
14	Terminated by **[ESC]**.
15	Timed out.
16	**[PRINT][SCRN]** or equivalent pressed.
17	**[CTRL][O]** pressed (see text).

The above exit codes are also returned via the ERR system variable, permitting the use of ON ERR(...) GOTO to route program execution following an inputfmt call. *The setting of* ERR *does not cause an execution error in the calling program.*

Negative OCC values represent trapped error conditions and are interpreted as follows:

1-11	CONFIG$(ABS(MINLEN),1+(MINLEN=7)) invalid.
12	COL out of range.
13	ROW out of range.
14	ELM out of range for data format IFMT$ or the specified element does not accept string data.
15	OCC out of range for element ELM.

If more than one defective value is detected in CONFIG$ the first one encountered will be the one flagged as an error.

Untrapped errors return to the calling program in the system ERR variable and should be processed in the usual fashion.

APPLICATION NOTES

inputfmt provides a programmable keyboard interface suitable for an interactive editing input session with direct output to a data format element. The user interface is identical to that described for the input function (page 133).

inputfmt derives the maximum and minimum input field sizes directly from the target element's attributes. The element's size determines the maximum value and the input type in combination with the presence or absence of data in the element determines the minimum value and whether default is possible. The following table summarizes the possible input conditions that inputfmt can automatically configure, based upon element input type:

Input Type	Type Meaning	Data Present	MINLEN	Default
0	Optional, variable	*	0	On
1	Optional, fixed	*	=MAXLEN	On
2	Mandatory, variable	No	1	Off
2	Mandatory, variable	Yes	1	On
3	Mandatory, fixed	No	=MAXLEN	Off
3	Mandatory, fixed	Yes	=MAXLEN	On

* Don't care.

inputfmt includes a built-in interface to the IDOL-IV pop-up help facility. If the user presses **[CTRL][O]** and a help module has been defined for the target element, inputfmt will display that help screen. If no help screen has been defined inputfmt will return the help requested status code in the OCC variable (as well as in ERR), as described above.

Where possible, inputfmt will not permit the entry of string data into elements whose attributes prohibit such input. For example, an element defined to be a six byte SQL date field cannot accept raw string data. In most cases, inputfmt will trap such an attempt and abort with an error, rather than risk corruption.

labtolin — Resolve Program Line Label to Line Number

Syntax:

```
CALL "labtolin",PROG$,LABEL$,LINENO
```

Call Parameters:

PROG$	Program name to be searched. See text.
LABEL$	Line label to be found, not case sensitive.

Returns:

LINENO	Line number in program PROG$ at which LABEL$ is defined or zero if unable to resolve label name or unable to access program PROG$.

labtolin searches the symbol table of a program on disk named in PROG$ for a statement where a label named in LABEL$ has been defined and returns the associated line number in LINENO. Possible uses for this capability included the configuration of self-modifying code or the verification of the integrity of a program. LINENO will be zero if the program is not readable in the current execution environment or does not have a statement that has been identified with the label in LABEL$. labtolin cannot be used on encrypted programs.

Examples:

```
CALL "labtolin","apvme02","dispatch",LINENO;
IF LINENO=0
   PRINT "Program apvme02 is corrupted!"
FI
```

The above example searches the program apvme02 for a statement where the label DISPATCH has been defined and if found, returns the associated statement line number in LINENO.

```
PROG$=CVT(PGN,128),
LABEL$="route"
CALL "labtolin",PROG$,LABEL$,LINENO;
IF LINENO=0
   PRINT "Program ",PROG$," is corrupted!"
FI
```

The above example searches the disk copy of the currently executing program for the statement labeled ROUTE.

linkfile — Link To Data File

Syntax:

```
CALL "linkfile",LINK$,FILE$
```

Call Parameters:

LINK$	IDOL-IV data link name in LLNNNNNN format.

Returns:

LINK$	Link title. See text.
FILE$	Filename associated with the link name passed in LINK$. Null if the link does not exist or has no filename associated with it.

linkfile provides a means of determining which files must be utilized by a calling program. The benefit realized by using linkfile instead of hard coding filenames is portability: if the filename associated with a particular link has been changed the program calling linkfile will automatically receive the new filename and a runtime error will not occur when an attempt is made to open the file (assuming the file itself has been correctly renamed).

Upon a successful return from this call LINK$ will contain the title that was assigned to the link when it was created. This title can be used for file maintenance screen titles, etc. LINK$ is not changed if the call is unsuccessful. If your program needs to call other functions that require a link name it should do so before calling linkfile.

Example:

```
LINK$="GCCPMAS";
CALL "linkfile",LINK$,FILE$;
F=UNT;
OPEN (F) FILE$
```

The above example gets the filename associated with the GCCPMAS link into FILE$ and then opens the file. The link title Corporate Profile Master is returned in LINK$.

To return both the format and file name associated with a link use the statlink function (page 239).

linkfmt — Link To Data Format

Syntax:

```
CALL "linkfmt",LINK$,FORMAT$
```

Call Parameters:

LINK$	IDOL-IV data link name in LLNNNNNN format.

Returns:

FORMAT$	Data format name associated with the link name passed in LINK$, in #LLNNNNNN format. Null if the link does not exist or the associated format name is invalid.

linkfmt returns the name of the format associated with a link. The benefit realized by using linkfmt instead of hard coding format names is portability: if a programmer renames a format, all associated links will be updated by IDOL-IV, linkfmt will automatically return the new format name and a runtime error will not occur when an attempt is made to reference the format.

Example:

```
LINK$="GCCPMAS";
CALL "linkfmt",LINK$,FORMAT$
```

In the above example, FORMAT$ will return #GCCPMAS, as that is the format name associated with the GCCPMAS link.

To return both the format and file name associated with a link use the statlink function (page 239).

lmargin — Compute Margin to Center Text String

Syntax:

```
CALL "lmargin",STRING$,WIDTH,COL
```

Call Parameters:

STRING$	Text string to be centered. 0<LEN(STRING$)<=WIDTH.
WIDTH	Maximum display columns on intended display device.

Returns:

COL	Computed zero-based starting column for text string. If a negative value is returned STRING$ is too long for the display device width.

lmargin computes the zero-based starting column position needed to center a text string on a display device. Embedded mnemonics in the string are ignored, except for 'DB', 'DT' and 'EP', which will cause lmargin to compute COL based upon WIDTH÷2. Avoid using any mnemonics that can affect cursor positioning (e.g., @(C,R) or 'LI').

Examples:

```
STRING$='SF'+"This is a test."+'CL';
CALL "lmargin",STRING$,80,COL
```

The above example results in COL=32.

```
STRING$='EP'+'SF'+"This is a test."+'CL';
CALL "lmargin",STRING$,80,COL
```

The above example results in COL=12, the required position to center an expanded string.

See also the scrnsize function (page 226) for a method of determining the current display width of a terminal.

loadfkey — Load or Clear A Terminal Function Key

Syntax:

```
CALL "loadfkey",FKEY,TEXT$,FLAG
```

Call Parameters:

FKEY		Function key number to be loaded or cleared, 1 to N, where N is the highest SHIFTed function key available on the terminal.
TEXT$		Text to download to selected function key, see text below. TEXT$ is ignored if the selected operation is a function key clear (see FLAG parameter below).
FLAG	0	Clear.
	1	Load.

Returns:

FLAG	0	OK, function key loaded or cleared.
	1	Unsuccessful—returned for an illegal function key number or if the terminal does not support function key control in software.

Prerequisites:

The mnemonics CR, F0, F1 and F2 must be defined in the TCONFIGW table for the affected terminal. Contact **BCS Technology Limited** for information on how these mnemonics should be defined.

loadfkey implements a terminal-independent method of loading character strings into the shifted function keys on any terminal that supports such an operation. To load a function key, set the key number in FKEY, place the desired character string into TEXT$ and set FLAG to 1. The text may be any ASCII characters, including (if necessary) escape sequences. loadfkey will replace any occurrence of the UNIX pipe symbol (|) with a carriage return ($0D$). To clear a key, set the function key number in FKEY and set FLAG to 0. The mnemonics CR, F0, F1 and F2 must be defined in the TCONFIGW table for the affected terminal or else loadfkey will silently exit and set FLAG=1.

loadfkey does not trap for an invalid function key number (other than zero) nor does it monitor the length of the character string passed in TEXT$. Most terminals have a hardware-defined limit on how much text can be associated with a function key, and will also have a limit on the total amount of function key text that can be stored in terminal memory.

Overloading the available function key memory may cause unpredictable effects with some terminals

Examples:

```
FKEY=5,
TEXT$= "Ship via truck|",
FLAG=1;
CALL "loadfkey",FKEY,TEXT$,FLAG;
IF FLAG
   PRINT "Unable to load function key",FKEY,"!"
FI
```

The above example load function key **[F5]** with the text string `Ship via truck` and appends a carriage return. If the user presses **[SHIFT][F5]** at an input prompt, the effect will be as though s/he had typed `Ship via truck` and pressed ⏮.

```
FKEY=5,
FLAG=0;
CALL "loadfkey",FKEY,"",FLAG;
IF FLAG
   PRINT "Unable to clear function key",FKEY,"!"
FI
```

The above example clears function key **[F5]**. Because no text is involved double quotes may be used as a placeholder for the `TEXT$` parameter.

`loadprof` Load, Generate or Update A Corporate Profile

Syntax:

```
CALL "loadprof"[,FLAG]
```

Call Parameters:

`FLAG`	Optional. If present, a corporate profile will be (re)generated from the data in the `#GCCPMAS` profile data format. If omitted, `loadprof` will attempt to load the profile for the company code in the `CODE` field of `#GCCPMAS`, in which case the value of `FLAG` is unimportant. See text.

Preparatory Operation:

The format `#GCCPMAS` must exist in the data dictionary and the `CODE` field of the `#GCCPMAS` format must be loaded with a company code in order to load a profile. All elements of `#GCCPMAS` must be loaded with appropriate data to generate or update a profile. See text.

Returns:

`FLAG`	`0`	OK, profile generated or updated.
	`1`	Unable to create or open profile database. See text.

`loadprof` provides the means to maintain and access a database of corporation profiles. Rather than hard coding company information into a program, you can utilize `loadprof` to get that information as needed. You can also employ `loadprof` to make software portable to different companies. A number of other cookbook functions (e.g., `pagehdr`) utilize this function to generate various text strings. Hence, a corporate profile should be generated to get the full results of calls to `pagehdr`, etc.

In order to load a corporate profile you must store the company code into the `CODE` field of the `#GCCPMAS` format and omit the `FLAG` parameter. In order to create or update a profile you must load all elements of the `#GCCPMAS` format with appropriate data and include the `FLAG` parameter (the value of `FLAG` doesn't matter; it is the presence of the variable that triggers the operation). The defined elements in this format are listed in numeric order below:

ADDRESS — This field is the legal business address of the company. It should be a street address rather than a post office box (although a P.O. box can be a part of the address). Any text up to 30 characters may be supplied.

AREACODE — This field is the company's voice area code. It is internally stored as a 16 bit binary number. The best way to write an area code into this field is with the `LET FMT` directive.

CITY — This field is the city in which the company headquarters are located. Any text up to 18 characters may be supplied.

CITY_TAX_RATE — This field is the city sales tax rate, expressed as a fractional percentage. For example, if the city sales tax rate is `0.75` percent, the proper entry into this field will be `.0075`. The best way to write a sales tax rate into this field is with the `LET FMT` directive.

CODE — This field is the profile's record key and should be a three character mnemonic code for the company. It must be lower case letters.

COUNTY_TAX_RATE — This field is the county sales tax rate, expressed as a fractional percentage. For example, if the county sales tax rate is `1.25` percent, the proper entry into this field will be `.0125`. The best way to write a sales tax rate into this field is with the `LET FMT` directive.

DATE — This field can be used as a system-wide (global) date for applications that require date concurrency between all tasks. It is stored as a four byte binary SQL date and may be decoded with the `FNDATE` function (see defined functions on page 31). A user-written program is required to extract the date from this field or to update it as required.

EMAIL_ADDR — This field is the Internet E-Mail address of the company. When combined with individual usernames, an E-Mail source/destination address is formed. For example, to send E-Mail to a person named `Bob Jones` at `Acme Screw Machine`, it might be routed to `bobjones@acmescrew.com`. Therefore, the corporate E-Mail address for this example would be `acmescrew.com`. Any text up to 30 characters may be stored.

FAXAREACODE — This field is the company's FAX area code. It is internally stored as a 16 bit binary number.

The best way to write an area code into this field is with the LET FMT directive.

FAXPHONE — This field is the company's FAX telephone number. It is internally stored as a 24 bit binary number. The best way to write a phone number into this field is with the LET FMT directive.

FEIN — This field is the company's federal employer identification number (FEIN). It is required information for many reports that are routinely required by federal, state and local authorities. It is internally stored as a 32 bit binary number. The best way to write an FEIN into this field is with the LET FMT directive.

NAME — This field is the full, legal name of the company. Any text up to 33 characters may be stored. For example, Acme Screw Machine Inc.

NICKNAME — This field is a shortened version of the legal company name stored into the NAME field. For example, if the company name is Acme Screw Machine Inc then this field would probably be either Acme Screw Machine or Acme Screw. Any text up to 25 characters may be stored.

PHONE — This field is the company's voice telephone number. It is internally stored as a 24 bit binary number. The best way to write a phone number into this field is with the LET FMT directive.

RESALENUM — This field is the company's sales tax exemption or resale number. Most wholesalers require this number in order to process non-taxable purchases. Up to 16 characters may be stored.

SHORTNAME — This field is a truncated version of the legal company name stored into the NAME field. For example, if the company name is Acme Screw Machine Inc then this field would probably be Acme. Any text up to 12 characters may be stored.

STATE — This field is the two character, Postal Service abbreviation for the state in which the company is located. For example, if the company is located in Illinois the state code will be IL. Only two character, upper case state abbreviations are acceptable.

SYSABREV	This field is a mnemonic representation of the information processing system's name as described in the SYSNAME field (below). For example, Acme Screw Information System might be abbreviated to ASIS (which you could pronounce "a sys"). Any text up to six characters may be stored.
SYSNAME	This field is the formal name of this company's information processing system. For example, Acme Screw Information System.
TAXCITY	This field is the name of the city that has taxing authority over this company. Any text up to 18 characters may be stored.
TAXCOUNTY	This field is the name of the county that has taxing authority over this company. Any text up to 18 characters may be stored.
TAXSTATE	This field is the two character, Postal Service abbreviation for the state that has taxing authority over this company. Only two character, upper case state abbreviations are acceptable.
WEBSITE	This field is the corporate website uniform resource locator (URL), aka web address. For example, Acme Screw Machine's website might be www.acmescrew.com. Any text up to 30 characters may be stored.
ZIPCODE	This field is the company's ZIP code, stored in ASCII format. Nine digit ZIP codes must be stored as NNNNNZZZZ rather than the usual NNNNN-ZZZZ format.

When loadprof is called to generate a profile an attempt will be made to create the profile database file (gccpmas.dbf) if it cannot be found anywhere in the *Thoroughbred* execution environment. By default, loadprof attempts to create the file in the directory returned by the workdir function (page 261). The FLAG parameter will indicate if the file was created and opened. Calls to load a profile will silently exit if the profile database and/or profile format GCCPMAS cannot be found.

`lockprog` Lock A Program For Single User Access

Syntax:

```
CALL "lockprog",PROG$,PROG
```

Call Parameters:

`PROG$`	Name of program to be locked.

Returns:

`PROG`	Channel number on which `PROG$` has been locked or zero if unable to lock program `PROG$`.

`lockprog` provides a mechanism for temporarily limiting program access to a single user. Such an action might be necessary during certain types of update operations. Control of the locked program should be released with a `CLOSE (PROG)` directive.

lpsetup — Generate Printer Report Setup Parameters

Syntax:

```
CALL "lpsetup",LP$,NFLD,CFL,IFS,AVL,CLL,LMC,PITCH$,FLAG
```

Call Parameters:

LP$	Device name of the target printer, such as P1. The printer does not have to be opened to a channel.
NFLD	Number of fields (columns) to be printed. 1<NFLD<100.
CFL	Combined length of all fields. See text.
IFS	Minimum acceptable inter-field spacing. If zero the value 1 is assumed. See text.
AVL	Maximum acceptable inter-field spacing. See text. IFS<=AVL.

Returns:

FLAG	0	OK, all returned values are valid.
	1	Invalid target printer.
	2	CPL mnemonic not defined for target printer.
	3	Computed line length exceeds printer capabilities.
IFS		Computed inter-field spacing.
AVL		Maximum number of printable columns at the pitch returned in PITCH$. (see text).
CLL		Computed length of printed line including inter-field spacing.
LMC		Left margin column at which printing should start to horizontally center the printed line on the page.
PITCH$		Computed printer pitch to correctly print the line. See text.

lpsetup is a general purpose printer setup utility that computes the parameters needed to properly space and position a line of text based upon some simple setup data provided by the calling program. At the same time, lpsetup masks the peculiarities of the target printer from the calling program, permitting some degree of coding independence.

To use lpsetup follow this general procedure:

A. **Determine which printer is to be used for output**. This value (such as P1) will be passed in the LP$ variable.

B. **Determine how many columns will be printed in the body of the report**. This value will be passed in the NFLD variable. At least two columns must be defined to use lpsetup.

C. **Add up the display lengths for each of the columns to be printed**. For each column, there are two values to consider:

1. **The length of the column heading or caption**. If the caption consists of stacked text, the caption length is equal to that of the longest text string.

2. **The display length of the data**. For numeric or date fields the display length is equal to the length of the display mask(s) including signs (masks should always be used for numerics and dates to guarantee a consistent field length). For string data, the length is the actual length of the character string. If any string data column is to be truncated use the truncated length.

   If you are using data formats, you can derive the display width of each element by evaluating FNLEN(FORMAT$,ELEMENT) (see page 31). Note that FNLEN() is not appropriate for date and time fields (evaluate LEN(FNDT$(0)) for date fields and LEN(FNTD$(0)) for time fields). A more accurate method is to call pagsetup (page 192) or rptsetup (page 223) to compute field lengths.

For each column, the greater of the two lengths established in steps 1 and 2 above will be used. Structure a zero-based numeric array to hold these lengths. For example:

```
DIM FW[NFLD-1];
FW[0]=<field #1 width>,FW[1]=<field #2 width>,...
```

Add the display lengths for all the columns. This sum is passed to lpsetup in the CFL variable:

```
CFL=0;
FOR I=0 TO NFLD-1;
   CFL=CFL+FW[I];
NEXT I
```

D. **Determine the minimum and maximum acceptable inter-field spacing**. Inter-field spacing is the "white space" that separates adjacent columns. Pass the minimum in the IFS variable and the maximum in the AVL variable.

When `lpsetup` computes the inter-field spacing it will maintain the value within the boundaries established in `IFS` and `AVL`. It is permissible for `IFS` and `AVL` to be the same.

E. **Call `lpsetup` with the values determined in steps A through D above**. Be sure to test the flag value so as to trap any error that may occur.

```
CALL "lpsetup",LP$,NFLD,CFL,IFS,AVL,CLL,LMC,PITCH$,FLAG;
IF FLAG
   GOTO ERROR
FI;
DIM SL$(CLL,"-"),DL$(CLL,"=");
DL$=PITCH$+@(LMC)+DL$
```

Upon return, `PITCH$` will contain the escape sequence needed to set the printer pitch to the proper value. `lpsetup` determines the correct pitch by evaluating the computed line length (`CLL`) in relation to the target printer's capabilities at various pitches.[11] An error will occur if the computed line length exceeds the printer's capabilities at its finest pitch. Also, note how the `CLL` variable can be used to create dashed lines (`SL$` and `DL$`) equal in length to the printed line length. These dashed lines can be used to set off the captions from the data, and sections of data from each other.

F. **Compute the display column values for each field using code similar to the following**:

```
DIM C[NFLD-1];
C[0]=LMC;
FOR I=1 TO NFLD-1;
   C[I]=C[I-1]+FW[I-1]+IFS;
NEXT I
```

The column value for any given field (other than the first one, whose value will either be zero or equal to `LMC`) can be determined by adding the display width of the previous field and the computed inter-field spacing to the column value of the previous field. Here, `C[]` is a column array and `FW[]` is a field width array, determined in step C above. The values `LMC` and `IFS` are returned from `lpsetup` following a successful call.

---

[11] ***BCS Technology Limited*** printer definitions include a reference mnemonic named `CPL` (**C**haracters **P**er **L**ine) embedded in the printer's IDOL-IV driver table. `CPL` indicates how many character columns can be printed at the five standard pitches: 10, 12, 15, 17.125 and 20 characters per inch (selected with the `PInn` mnemonic, where `nn` is `10`, `12`, `15`, `17` or `20`). `lpsetup` will fail with an error if it cannot find the `CPL` mnemonic.

G. **At print-time, output your line in a manner similar to the following**:

```
PRINT (LP)PITCH$,@(C[0]),<field #1>,@(C[1]),<field #2>,...
```

This will set the printer pitch and output the fields in the correct locations on the page. The `PITCH$` variable was returned by `lpsetup` and the `LP` variable is the channel number that was opened to the printer.

To generate a page header for your report refer to the `pagehdr` function (page 188).

Note that in the above code examples it is assumed that both the column captions and the data are left justified relative to each other (i.e., caption and data are both printed starting at the same character position on the page). If you wish to have centered and/or right justified captions you will need to set up two column arrays, one for the captions and one for the data. It may be easier to use the `pagsetup` function (page 192) to generate a page layout, as it calculates some of the details that were described above.

maketemp — Generate Temporary Direct or Sort File

Syntax:

```
CALL "maketemp",KSIZ,RSIZ,TF$,TF[,ADF]
```

Call Parameters:

KSIZ	Key size in bytes. 0<KSIZ<145.
RSIZ	Record size in bytes. 0 or 5<RSIZ<65001. See text.
ADF	Optional autodelete flag:
	0 Do not delete file on close (default).
	1 Delete file on close. See text.

Returns:

TF$	Generated filename derived from the tempfid function (page 242).
TF	Channel to which the temporary file has been opened. Zero if unable to create and open the file.

maketemp provides a convenient mechanism for generating and opening temporary, autoexpanding direct and sort files during program runtime. To generate a SORT file set the RSIZ value to zero. Otherwise, set RSIZ to an appropriate record size to create a DIRECT file (minimum record size of 6). In either case, the file will be created with rw------- permissions in the directory pointed to by the TMPDIR environment variable in UNIX/Linux and will be opened on channel TF.

If the optional ADF parameter is present and set to 1, maketemp will create the file in a way that will cause it to be automatically deleted when the channel in TF is closed, a BEGIN or END directive is executed, or the task is released to the operating system. This feature is available only with UNIX and UNIX-like operating systems. On any operating system you can automatically close and delete temporary files created by maketemp with the erasetmp function (page 102).

modmctrl — Execute Off-Line Modem Control Sequences

Syntax:

```
CALL "modmctrl",RCH,WCH,CMD$,TIMOUT,RSP,RSP$,RAW$
```

Call Parameters:

RCH	Input (read) channel opened to modem.
WCH	Output (write) channel opened to modem.
CMD$	Modem command to be executed, such as S0=0. Do not prepend an AT string to the command. See text.
TIMOUT	Period in seconds to wait for a response from the modem.

Returns:

RSP	Numeric modem response code, such as 3. See text.
RSP$	Interpretation of numeric response code in RSP, such as no carrier. See text.
RAW$	Raw (uninterpreted) response string generated by modem following receipt of command.
ERR	0 OK, all returns are valid. 1 Timed out with no response from modem. 110 Modem not generating numeric responses. See text.

The setting of ERR *to any of the above values will not cause an execution error in the calling program.*

modmctrl is a function that may be used to issue off-line commands to a modem that has been opened on channels RCH and WCH, as with the modmopen function (page 172). modmctrl will prepend the command in CMD$ with the AT (attention) command and write to the modem output channel. Following the issuance of the command, modmctrl will listen for a response on the input channel for a maximum of TIMOUT seconds. If a response is received, modmctrl will attempt to interpret it, returning the numeric response code in RSP and a terse interpretation of that code in RSP$. The unexpurgated output of the modem will appear in RAW$.

Initially, the modem will be in off-line or command mode and able to receive and act upon commands sent via modmctrl. Upon connecting to a remote system the modem will switch from off-line to on-line mode, at which time it will no longer recognize commands, instead passing any data through to the remote system.

To return to off-line mode without disconnecting the call you must write the modem escape string (usually +++) on the modem's output channel. The modem will require a quiet period of at least one second immediately before and after the escape string so as to not confuse it with actual data. Here is a suggested method for switching from on-line to off-line mode:

```
WAIT 2
WRITE RECORD(WCH)"+++"
WAIT 2
```
…modem is now in command or off-line mode…

Once the modem has gone off-line it will react to commands again. You can place it back on line with the ATO command, either by directly writing it to the modem output channel or by calling modmctrl.

As mentioned above, modmctrl attempts to interpret modem responses to commands. In order for this to occur the modem must be configured to generate numeric responses (such as with the ATV0 command). modmctrl will abort with ERR=110 if it detects that the modem is returning text responses. The modmopen function (page 172) configures the modem to produce numeric responses.

Basic (X0) responses are interpreted as follows:

RSP	RSP$	MEANING
0	ok	Modem successfully executed command.
1	connected	Connected to remote modem.
2	ring in	Inbound call detected.
3	no carrier	No carrier detected from remote modem.
4	error	Modem could not execute command.

Extended (X4) responses generate additional RSP values, which are interpreted as follows:

RSP	RSP$	MEANING
5	connected	Connected to remote modem at 1200 BPS.
6	no dial tone	No dial tone detected for outbound call.
7	busy	Dialed line busy.
8	no answer	Dialed line did not answer call.
10	connected	Connected to remote modem at 2400 BPS.

RSP values equal to or higher than 13 indicate that a connection was made at a speed higher than 2400 and will return connected in RSP$. Consult your modem manual for details on the extended responses that are supported.

Example:

The following example uses modmctrl to call a remote system to exchange data. It is assumed that the modem has been opened on channels RCH (read) and WCH (write):

```
01000       CMD$= "DT18005551234",  !phone number to dial
            TIMOUT=30;
            CALL "modmctrl",RCH,WCH,CMD$,RSP,RSP$,RAW$;
            ON ERR(1,110) GOTO OK,TIMED_OUT,NOT_NUMERIC
01010 OK:   CALL "seterr",RSP;
            ON ERR(3,4,6,7,8) GOTO OK2,NO_CARRIER,ERROR,
               NO_DIAL_TONE,BUSY,NO_ANSWER
01020 OK2: …connected to remote modem, begin processing…
```

In the above sequence a remote system at 1(800) 555-1234 is called. Specific error branches are configured for the case where either the modem fails to respond (that is, modmctrl times out) or returns a non-numeric response to the dial-out command. Assuming modmctrl exits with ERR=0, indicating that the modem responded to the dial command, the next step is to determine if a connection was actually made, and if not, why the call wasn't completed. The easiest way to set up the program logic to determine this is to use the seterr function (page 234) to condition ERR according to the response received from the modem. The code at line 1010 causes a branch to occur to OK2 if a connection was made or to other code based upon why the call was not completed. You can use this technique to advise the user why his/her call didn't go through.

See also modmopen (page 172) and portread (page 212).

modmopen — Open, Lock and Initialize Serial Modem

Syntax:

```
CALL "modmopen",PORT$,SPEED,CBITS,SBITS,PARITY,RFMT,OPTCTL,TIMOUT,RCH,WCH
```

Call Parameters:

PORT$ — Serial port device name to which modem is connected, such as `/dev/tty1a16`. See text.

SPEED — Bits per second (baud) rate at which port is to run, passed as an index value:

SPEED	BPS Rate
0	1200
1	2400
2	4800
3	9600
4	19200
5	38400
6	57600
7	76800
8	115200
9	230400

Speeds above `38400` are not supported by all systems or modems. See text.

CBITS — Number of bits per character, either `7` or `8`.

SBITS — Number of stop bits, either `1` or `2`.

PARITY
- `0` No parity.
- `1` Odd parity.
- `2` Even parity.

RFMT
- `0` Basic modem responses (`X0`) enabled.
- `1` Extended modem responses (`X4`) enabled. See text.

OPTCTL
- `0` Data compression and error control enabled.
- `1` Data compression disabled, error control enabled.
- `2` Data compression and error control disabled.

Returns:

RCH — Input (read) channel opened to modem. Not valid if any error occurs.

WCH — Output (write) channel opened to modem. Not valid if any error occurs.

ERR	0	OK, channels RCH and WCH are valid.
	1	Invalid or inaccessible device name—cannot open. See text.
	2	Port locked by another *Thoroughbred* task.
	3	Modem failed to respond to initialization command. See text.

The setting of ERR *will not cause an execution error in the calling program.*

modmopen is a UNIX/Linux function that creates a full duplex data path between the calling program and a modem attached to a serial port on the host system. Once this path has been created it is possible to command the modem to call a remote system and exchange data. The modem must be compatible with the industry standard Hayes AT command set and must be able to maintain a fixed serial port baud rate regardless of the connection speed negotiated with a remote modem.[12]

modmopen will attempt to acquire exclusive control over the modem attached to the port specified in PORT$ and if successful, will attempt to configure the port to the speed and data format specified by the SPEED, CBITS, SBITS and PARITY parameters. Following port configuration, modmopen will initialize the modem to make it ready for communication. If modem initialization is successful, input and output channels will be returned in RCH and WCH respectively. modmopen will fail and return ERR=3 if anything in the setup parameters (such as port speed) prevents communication with the modem during the initialization phase. This same error will also be returned if the modem rejects the initialization commands because of an invalid parameter. Refer to the manual for your modem to determine which features it supports.

In order to successfully acquire control of the target modem, the calling task must have read and write permission on the device file name associated with the modem's serial port. On most UNIX/Linux systems the device file should be set to rw-rw-rw- permissions with the chmod 666 <device_name> command. For example, if the modem is attached to /dev/tty1a16 you would execute chmod 666 /dev/tty1a16. Less desirably, you could give ownership of the device to the user who started the *Thoroughbred* task.

Initialization will configure the modem to generate basic (X0) numeric responses to off-line commands, run at a fixed baud rate as determined by the SPEED index and use data compression and error control if possible. If the modem can support extended responses, set RFMT to 1 to enable the X4 response set, which will provide more detailed information as to how certain modem commands were processed. In the event the remote system cannot support data compression and/or error control, turn off these features as required by conditioning the OPTCTL parameter. Avoid running the modem faster than 2400 if error control is turned off, as telephone line quality may prevent reliable communication with the remote system.

[12] Virtually all modern modems conform to these requirements.

If possible, the modem should be configured via DIP switches or software commands so it will respond to the DTR (data terminal ready) control line by resetting when DTR is deasserted (which will usually occur when both channels to the modem are closed). If this modem feature can be enabled, closing the channels to the modem will be all that is required to disconnect a call and return the modem to the offline state. Otherwise, it will be necessary to transmit the modem escape sequence (usually +++), followed by ATH0 to disconnect the call and ATZ to reset the modem.

Once channels have been opened to the modem communication can begin. To send commands to the modem while it is off-line, use the modmctrl function (page 169), which will handle both the command aspect and the retrieval and interpretation of modem responses. Once the modem has gone on-line (i.e., connected to another modem), read from the input channel with the portread function (page 212), which will take care of matters such as port timeout or no data present. Data should be written to the output channel with WRITE RECORD (not WRITE, which will attach a linefeed character to the end of the data). When all communications have completed both channels should be closed to release the modem. It is recommended your program close WCH before it closes RCH. This sequence will assure that the write buffers to the modem port are flushed before control is relinquished.

Some versions of UNIX (such as HP-UX and SCO) assign two device names to each serial port, a "modem control" device name and a "non-modem control" name. Use only the non-modem control name with this function. With SCO, for example, you would use tty1a16 (the non-modem control device name) instead of tty1A16 (which is the modem control name for the same port). Also note that the locking of the serial port occurs only within *Thoroughbred* and does not prevent other UNIX programs or operating system calls from concurrently accessing the port. Use caution with UNIX commands like cu, stty and uucp.

Example:

```
PORT$="/dev/tty1a16",
SPEED=3,
CBITS=7,
SBITS=1,
PARITY=2,
RFMT=1,
OPTCTL=1
TIMOUT=30;
CALL "modmopen",PORT$,SPEED,CBITS,SBITS,PARITY,RFMT,OPTCTL,TIMOUT,RCH,WCH;
ON ERR(1,2,3) GOTO OK,INVALID_PORT,PORT_LOCKED,BAD_MODEM
```

The above example attempts to open a connection to the modem on /dev/tty1a16, running it at 9600 bits per second, with 7 bits per character, 1 stop bit and even parity.

Extended responses and error control are enabled and data compression is disabled. `modmopen` will time out in 30 seconds if it cannot communicate with the modem for any reason. If control of the modem can be acquired, an input channel will be returned in `RCH` and an output channel in `WCH`.

See also `modmctrl` (page 169) and `portread` (page 212).

msgbox — Display Message In Dialog Box

Syntax:

```
CALL "msgbox",L$[ALL],COL$,R,S[,CO]
```

Call Parameters:

L$[ALL]	Array of text lines to be displayed, one-based.
COL$	Display attributes, such as color, flash, etc. see text.
R	Top row of text box border. If zero, box will be vertically centered.
S	0 Box has a "shadow."
	1 Box has a "halo."
	2 No shadow or halo.
CO	Optional column offset from center. See text.

Returns:

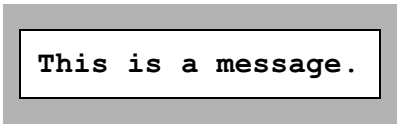
L$[]	Deleted from memory.
COL$	Window name used to create text box.

msgbox provides a method of creating attractive dialog boxes to report status conditions and other operational messages. The display attributes passed in COL$ can be a mixture of colors, flash and other text attributes.

msgbox automatically adjusts its size to accommodate the longest text line passed in L$[]. If you wish to have the text left justified in the box pad all lines to the length of the longest line. Otherwise, each text line will be centered in the box. See the frmttext function (page 116) on formatting a raw text string to use with this function.

msgbox allows you to display the box with a shadow:

a halo:



or, no shadow or halo. The box contents are displayed in reverse video.

The optional CO offset parameter may be used to offset the box from the center of the screen. If omitted, the message box will be horizontally centered. A negative CO value will move the box CO columns to the left of center, a positive value to the right.

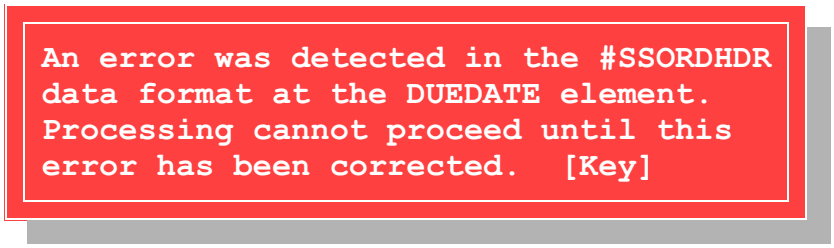
Example:

The following example, taken from the frmttext narrative, illustrates a typical use for msgbox. In this example, the user is alerted to a trapped program error by passing an unformatted text string (MSG$) to frmttext and the output of frmttext to msgbox.

```
MSG$="An error was detected in the #SSORDHDR data format at the
DUEDATE element. Processing cannot proceed until this error has
been corrected.  [Key]",
MAXLEN=40,
MODE=1;
CALL "frmttext",MSG$,MAXLEN,MODE,LINE$[ALL];  REM format text.
COL$='BACKGR'+'WHITE'+'RED'+'RB';
CALL "msgbox",LINE$[ALL],COL$,0,0;
CALL "pause",0; REM Wait for user response.
WINDOW DELETE (COL$);
```

…collapse window and continue processing

The user will see the following text box vertically and horizontally centered on the screen and the terminal will beep:



The call to frmttext converts the unformatted ("raw") text in MSG$ into lines of text that will left justify when msgbox is called. The text box itself will display with a red background and white foreground on terminals capable of color. Because msgbox displays in reverse video what would normally be considered the background color (white in this case) becomes the foreground color and *vice versa*.

nextfid — Generate Next Temporary Filename In Sequence

Syntax:

```
CALL "nextfid",F$
```

Call Parameters:

F$	Current temporary filename in XXXXXXXX.TTT format.

Returns:

F$	Next XXXXXXXX.TTT filename in sequence.

nextfid takes a temporary filename generated with the tempfid function (page 242) and converts it into the next logical filename in the sequence. It is used when multiple temporary files need to be generated by a single task. Spurious results will occur if the filename passed in F$ does not confirm to the XXXXXXXX.TTT format. See the tempfid narrative for a full discussion of the temporary filename structure.

Example:

```
CALL "tempfid",F1$;          REM Get a temporary filename.
F2$=F1$;                     REM Copy that filename.
CALL "nextfid",F2$;          REM Generate next filename in sequence
CALL "tempdir",TMPDIR;       REM Get temp directory number.
SORT F1$,KSIZ1,0,TMPDIR,0; REM Generate 1st temp sort file.
SORT F2$,KSIZ2,0,TMPDIR,0; REM Generate 2nd temp sort file.
TF1=UNT;
OPEN (TF1)F1$;               REM Open 1st file.
TF2=UNT;
OPEN (TF2)F2$;               REM Open 2nd file.
```

…process as required…

In the above example, if the call to tempfid returns 00DEE3BC.032 in F1$ the following call to nextfid will return 00DEE3BD.032 in F2$.

numsorts — Get Number of Defined Sorts In MSORT or ISAM File

Syntax:

```
CALL "numsorts",F$,NS
```

Call Parameters:

F$	MSORT or ISAM filename.

Returns:

NS	Number of defined sorts or zero if file cannot be opened or is not an MSORT or ISAM type.

numsorts returns the number of defined sorts in the file named in F$, which should be an MSORT or ISAM file. By definition, an MSORT or ISAM file will have at least one sort, which is the primary sort. The file does not have to be opened when this function is called.

opendict — Open IDOL-IV Data Dictionary File

Syntax:

```
CALL "opendict"[,DF]
```

Call Parameters:

None.

Returns:

DF	Channel number on which the data dictionary file was opened.

opendict determines the channel number on which the IDOL-IV data dictionary file (IDDBD) has been opened and if it has not been opened, opens it on the highest possible channel number. If the variable DF is part of the call the channel number will be returned. If your programs frequently access the IDOL-IV data dictionary and/or make use of a lot of cookbook functions, opening the data dictionary in this fashion can aid execution speed.

openlink — Open IDOL-IV Link

Syntax:

```
CALL "openlink",LINK$,TITLE$,FORMAT$,DFILE$,TFILE$,DFCH,TFCH
```

Call Parameters:

LINK$	IDOL-IV link name in LLNNNNNN style. If preceded with an exclamation point (!) openlink will attempt to lock the data file. See text.

Returns:

LINK$	Link name without optional lock symbol (!) if open operation was successful. Unchanged if operation fails for any reason. See text.
TITLE$	Link title. See text.
FORMAT$	Data format associated with link in #LLNNNNNN style.
DFILE$	Data filename associated with link.
TFILE$	Text filename associated with link, if defined.
DFCH	Channel on which DFILE$ has been OPENed.
TFCH	Channel on which TFILE$ has been OPENed or zero if text file has not been defined or is not accessible in the execution environment.
ERR	0 OK, all returns are valid. 1 Link name not found. 2 Data file not found. 3 Data or text file (if defined) locked by another task. 4 Unable to lock data file (if requested). 110 Unable to INCLUDE format associated with link.

The setting of ERR *to any of the above values will not cause an execution error in the calling program.* All returns will be null or zero if the operation fails.

openlink provides a portable way to access a database using only an IDOL-IV link name. Upon a successful return, openlink will have INCLUDEd the format named in FORMAT$, OPENed (and locked, if requested) the data file named in DFILE$ and if defined, OPENed the text file named in TFILE$, with the channel numbers being returned in DFCH and TFCH, respectively. It is not an error if a text file has not been defined or created: in such a case, TFILE$ will be null and TFCH will be zero. Null/zero values are returned in the event any trapped error occurs.

Upon a successful return from this call TITLE$ will contain the title that was assigned to the link when it was created. This title can be used for file maintenance screen titles, printed reports, etc.

The presence of an exclamation point (!) in the first character of the link name will cause openlink to attempt to lock the data file associated with the link. For example, if LINK$="!GLARLOG" then openlink will attempt to lock the data file associated with the GLARLOG link. If the open-with-lock operation is successful, the ! will be striped from the link name in LINK$. Otherwise, LINK$ will not be changed. The text file (if any) will not be locked.

Example:

```
LINK$="!GLARLOG";
CALL "openlink",LINK$,TITLE$,FORMAT$,DFILE$,TFILE$,DFCH,TFCH
```

This example will attempt to open and lock the GLARLOG link data file, as well as INCLUDE the associated format. If the operation succeeds, all returns will be valid and the calling task will have exclusive control over the data file.

See also opnfiles (page 183) and statlink (page 239).

opnfiles — Open Files by IDOL-IV Link Name with Optional Lock

Syntax:

```
CALL "opnfiles",LINKLIST$,ROW,FLAG,F[ALL]
```

Call Parameters:

LINKLIST$	List of link names, each in LLNNNNNNf format, where LLNNNNNN is the link name and f is the lock flag. Link names must be padded to eight characters. See text for details.
ROW	Screen row on which to print advisory messages. Zero suppresses advisory messages.

Returns:

LINKLIST$	Link names, each in LLNNNNNN format with the lock flag stripped. See text for details.
FLAG	0 Operation successful. 1 Unresolved link reference. Either a link is not defined or has no associated filename. 2 Unable to open a file because it was not found or has been locked by another process. 3 Unable to lock a file because another process already has it open.
F[ALL]	Channel numbers associated with each opened file in link order. Invalid if FLAG>0.
ROW	Sequence number indicating which link caused an error. For example, if FLAG=3 and ROW=2 the file associated with the second link in LINKLIST$ could not be locked.

opnfiles opens and optionally locks files associated with a list of IDOL-IV links passed in LINKLIST$. The operation is of an "all or nothing" nature: either all files are successfully opened and locked (if required) or none are. The lock flag associated with each link name should be 0 (zero) if locking is not required or 1 (one) if locking is required. See the example below.

opnfiles has built-in advisory capabilities to let a user know if the attempt to open and lock a file fails. To activate advisory messages set ROW to a nonzero value; the resulting text boxes will be positioned so ROW bisects them. The messages are self-explanatory to the user.

If you prefer to do your own advisory processing set ROW=0 and use the information returned in FLAG and ROW to advise the user if a failure occurs.

opnfiles makes a reasonable effort to complete the operation in its entirety. Up to 15 attempts are made to access a busy file, with attempts occurring at one second intervals. If advisory messages have been enabled and a file is busy the status message One moment please... will be displayed as access attempts continue. When 15 failed attempts have been made opnfiles will report The <description> file is busy. Access has been temporarily denied. opnfiles uses the link description to fill in the <description> portion of the message.

Example:

In the following example, four files are opened and one of them is locked. Advisory messages are enabled so they display at row 20. If the operation is successful numeric variables with the same names as the links are generated and set equal to the corresponding file channel numbers.

```
LINKLIST$="SSITEM  1SSORDHDR0SSORDLIN0SSPRSPCT0",
ROW=20;
CALL "opnfiles",LINKLIST$,ROW,FLAG,F[ALL];
IF FLAG
   ON FLAG-1 GOTO BAD_LINK,CANT_OPEN,CANT_LOCK
FI;
L=8;
FOR I=0 TO STL(LINKLIST$)/L-1;
   EXECUTE LINKLIST$(I*L+1,L)+"=F[I]";
NEXT I;
```

…process as required…

If the above operation is successfully completed all files will be opened, the file associated with the SSITEM link will be locked and a zero-based list of channel numbers will be returned in F[] in link order (e.g., F[1] will correspond to the SSORDHDR link's file). The FOR/NEXT loop will produce numeric variables named SSITEM, SSORDHDR, SSORDLIN and SSPRSPCT, each set to the open channel of the corresponding file. Note that the SSITEM link name was padded with spaces to eight characters. Also, note that the entire operation was accomplished without hard-coding a single filename or channel number.

This function is obsolete and should not be used for new development. See openlink (page 181).

opnprntr — Select and Open A Printer Device

Syntax:

```
CALL "opnprntr",LP,LP$,FM$,D$,PROW,SROW,TIMOUT,TFLAG
```

Call Parameters:

LP		Channel number on which to open printer. See text.
LP$		Default printer or null. See text.
FM$		Reserved for future use. May be replaced with "".
PROW		Row on which to display prompts and messages generated by this function. See text.
SROW		Screen row for top border of printer selection box. See text.
TIMOUT		Input timeout period in seconds; zero disables timeout.
TFLAG	0	Any device may be selected.
	1	Only printers and file output may be selected.
	2	Only printers may be selected.

Returns:

LP		Channel on which printer has been opened.
LP$		Device name of selected printer or output filename if print-to-file was selected; null if aborted or timed out.
D$		Physical location description of selected printer; null if aborted or timed out. See text.
TFLAG	0	Selected device is spooled.
	1	Selected device is direct connect (non-spooled).
	2	Selected device is slaved to a terminal.
	3	Selected device is the user's terminal.
ERR	0	OK, device selected.
	1	User aborted with **[ESC]**.
	2	Selection timed out.

opnprntr is a "front end" to the selprntr function (page 227) that, in addition to allowing the user to interactively select a printer device, handles the process of opening a channel to that device and performing a suitable initialization sequence to prepare the device for receiving output. Refer to the selprntr narrative for details on the selection portion of the user interface.

If a non-zero value is passed in the LP variable, that value will become the channel on which the printer device will be opened. On the other hand, if LP is zero, a channel number will be assigned. If you elect to assign a channel by setting LP to something other than zero, be careful to not accidentally use the number of an already opened channel, as opnprntr will perform a CLOSE (LP) operation before opening the channel to the printer device.

In many cases, it is desirable to establish a default printer for a user based upon where s/he is located in the building. Such a default may be passed through LP$ as either a *Thoroughbred* printer device name, such as P3, or a number that opnprntr can interpret as an index into the list of available devices (e.g., "1" defaults to the first listed device, "2" to the second, and so forth). opnprntr will ignore any default value that doesn't make sense in the execution environment. For example, "PQ" will be ignored if no printer device designated as PQ exists in the environment defined by the IPL file used to start *Thoroughbred*.

The PROW parameter determines the row on which messages and prompts generated by opnprntr will appear. If PROW is zero, the second row from the bottom of the screen (usually row 22) will be used as the default. The SROW parameter determines the top row for the device selection box. If SROW is zero, the selection box will be vertically centered on the screen. In most cases, it is best to leave these values at zero and allow opnprntr to work out the display details.

The TFLAG parameter may be used to prevent the selection of print-to-file or terminal output when such a selection would be inadvisable for a particular program (e.g., printing invoices and checks).[13] Upon return, TFLAG will pass a numeric value indicating the nature of the connection to the selected device. This information can be used to alter the way in which a program interacts with the selected output device.

Upon return, opnprntr will load D$ with location data derived from the configurable printer table maintained in IDOL-IV (see selection 13 from the IDOL-IV Utilities Menu). This information should have been entered at the time the printers were defined. D$ will be null if selection was aborted or timed out.

opnprntr performs device-specific initialization sequences once a channel has been opened. If the selected device is a printer (as opposed to the user's terminal or file output), it will be initialized with the 'OPEN' mnemonic. 'OPEN' usually sets certain printer defaults, such as pitch, line spacing, font and so forth.

---

[13] Even though you may not want to allow print-to-file in the finished program you should make this option available during program development and testing. It's a handy way to test the printing portion of a program without consuming a lot of paper or forms.

The exact effect of 'OPEN' is dependent on how the IDOL-IV terminal driver table for the selected printer was structured—incomplete tables may not even have an 'OPEN' mnemonic defined, in which case no initialization will be performed. Should the user select output to terminal, an 'EM' mnemonic will be printed, which will prevent undefined mnemonic errors (ERR=29) from occurring during a report run. Note that the effect of 'EM' will be lost if the program escapes to console mode for any reason.

The selection of print-to-file (PF) will cause opnprntr to prompt the user for a filename to which output will be directed. Any filename up to eight alphanumeric characters is acceptable. opnprntr will not permit the entry of a filename having characters that are not letters or numerals—this guarantees that no characters that are significant to the operating system shell will creep into the filename. The resulting output filename takes the form <filename>.Tx, where Tx is the terminal ID of the task as derived from FID(0). The file itself will be created in the tmp subdirectory—the exact directory is determined by a call to the tempdir (page 240) function.

Example:

```
LP=0,
LP$= "P2",
TIMOUT=300,
TFLAG=0;
CALL "opnprntr",LP,LP$,"",D$,0,0,TIMOUT,TFLAG
ON ERR(0) GOTO ABORT,CONTINUE
```

In this example, the setting of LP to zero will cause opnprntr to assign a channel number. P2 is set as the default printer and a selection timeout period of five minutes is established. Both PROW and SROW are replaced with zeros, which will cause opnprntr to automatically position the selection window (and the filename prompt window if print-to-file has been selected). Upon return, execution will branch to the CONTINUE statement if the user selected a printer, or to the ABORT statement if s/he escaped or selection timed out. Because the TFLAG parameter was zero there were no restrictions placed on the selection of file or terminal output. Also, note the use of quotes in place of the reserved FM$ parameter.

pagehdr — Generate A Top-of-Page Report Header

Syntax:

```
CALL "pagehdr"[,FLAG]
```

Preparatory Operations:

The logical format #GCPAGHDR must be loaded with appropriate data.

Call Parameters:

FLAG	0	Title is printed at standard height and width. Default if FLAG is omitted.
	1	Title is printed in double width if supported by target printer.
	2	Title is printed in double height if supported by target printer.

Returns:

None. See text.

pagehdr provides a code-independent method of generating a header at the top of each page of a printed report. pagehdr functions by reading information found in the #GCPAGHDR logical data format and generating the required fields needed to structure and print the header. Once initial data has been loaded into #GCPAGHDR each call will cause pagehdr to handle such things as incrementing the page number and printing the header.

Prior to calling this function your program must load #GCPAGHDR as required, #GCPAGHDR consists of the following elements (in numeric order):

CODE	This field is an index into the corporate profile database that is maintained with the loadprof function (page 159). Profile codes are three character alphanumeric sequences. A profile is necessary if a company name is to appear in the page header. Refer to loadprof for further information.
CPL	This numeric field is the maximum number of characters per line the target printer can produce at 10 characters per inch. If it is zero, the first call to pagehdr will get that information from the printer driver. In most cases, this field should be initialized to zero and pagehdr allowed to insert the proper value. An improper value may cause various header formatting errors.

DATE

This field is a six byte SQL date which generates the date and time fields seen in the header. If this field is zero, the first call to pagehdr will set it to today's date and time as derived from the CDN system variable. If you wish to set it to some other date you must generate a date and time in a compatible format. This can be done in the following way, assuming the format is soft-included:

```
PRECISION 4;
LET FMD("#GCPAGHDR.DATE")=FNDATE$(CDN,"#GCPAGHDR.DATE",0)
```

This will convert a *Thoroughbred* date/time numeric (DTN) in DATE into the six byte SQL format required by the element, with ±5 second precision.

DOCNUM

This field ("documentation number") may be any text up to 24 characters in length.

LP

This field is the opened channel number assigned to the target printer. 0<#GCPAGHDR.LP<32765.

PAGE

This field is the current page number in two byte binary format. Each call to pagehdr will increment it prior to printing. Thus, if this field is initialized to zero page numbering will start at page 1. To set the page number to some other page use the following sequence:

```
#GCPAGHDR.PAGE=PAGENUM-1
```
or
```
LET FMT("#GCPAGHDR.PAGE")=STR(PAGENUM-1)
```

where PAGENUM is the page number. Note that the page number must be set to one less than required as the page number is incremented before the header is printed. 0<=#GCPAGHDR.PAGE<65536.

PROGRAM

This field is the name of the program producing the report. Any text up to eight characters is allowed. In most cases, you can load this field with the expression:

```
#GCPAGHDR.PROGRAM=PGN
```

where PGN is the system variable holding the program name currently being executed.

SECTION — This field is an optional report section number that may be printed with the page number. In some cases, it may be desirable to break up a report into multiple sections. For example, you may wish to produce a sales report that starts a new section for each salesperson, in which case you would increment the section number each time another salesperson's data appeared on the report and perform a formfeed to start the new section. You may assign a section number with one of the following sequences:

```
#GCPAGHDR.SECTION=SECTIONNUM
```
or
```
LET FMT("#GCPAGHDR.SECTION")=STR(SECTIONNUM)
```

where SECTIONNUM is the desired section number. 0<=#GCPAGHDR.SECTION<32768. If the section number is zero no section reference will be printed on the report header. See the example below.

SUBTITLE — This field is an optional report subtitle (see the TITLE field below). Any text up to 50 characters is allowed. Aside from the stripping of leading and trailing spaces, pagehdr will print this field unexpurgated in the center of the page.

TITLE — This field is the title text of the header. Any text up to 50 characters is allowed. pagehdr will strip leading or trailing spaces, capitalize the text and surround it with stars (*). Hence sales comparison will be transformed into ***** SALES COMPARISON ***** and printed in the center of the page and if possible, bold faced.

The optional FLAG parameter may be utilized to alter the default appearance of the header title. Support for double width and/or double height must be present in the IDOL-IV printer driver script to take advantage of this feature.

Example:

```
LP=UNT;
OPEN (LP)"P1";
FORMAT INCLUDE #GCPAGHDR;
#GCPAGHDR.CODE="abc",
#GCPAGHDR.DOCNUM="4.1.35.4.2",
#GCPAGHDR.LP=LP,
#GCPAGHDR.PROGRAM="CXQ04B",
#GCPAGHDR.SECTION="3",
#GCPAGHDR.SUBTITLE="Sorted by Description",
#GCPAGHDR.TITLE="inventory report";
```

…program retrieves data & is ready to print a page…
```
CALL "pagehdr";
```

Assuming the date is November 30, 1997 at 10:16 AM, the above example will produce the following output to printer P1 on the first call to `pagehdr`:

```
Jan 03, 2000                    COMPANY NAME                Section 3 -- Page 1
4.1.35.4.2                 *** INVENTORY REPORT ***          10:16 AM (ssicr02)
                             Sorted by Description
```

Following the generation of the page header two linefeeds will occur. Your program should then start printing the report body. Subsequent calls to `pagehdr` at the start of each new page will increment the page number but will not affect any other data.

pagsetup — Set Up Report Page Parameters

Syntax:

```
CALL "pagsetup",DFMT$,LP$,CAPROWS,IFSL,IFSH,CC$,DL$,LMC,NFLD,PITCH$,SL$,DC[ALL]
```

Call Parameters:

DFMT$	Display format from which page will be printed. See text.
LP$	Printer device name on which report will generated. The printer does not have to be opened on a channel when pagsetup is called.
CAPROWS	Maximum number of column description rows to be generated. See text.
IFSL	Minimum amount of whitespace between adjacent columns.
IFSH	Maximum amount of whitespace between adjacent columns.

Returns:

CAPROWS	Actual number of column description rows that was generated.
CC$	Column description string with embedded positioning data. See text.
DL$	Double dashed line (======) equal in length to that of the computed length of the print line, prefixed with the left margin column.
LMC	Left margin column, zero based.
NFLD	Number of elements in the display format—also the number of columns that will be printed on a line.
PITCH$	The printer pitch required to generate the report, returned as a printable mnemonic.
SL$	Single dashed line (------) equal in length to that of the computed length of the print line, prefixed with the left margin column.
DC[]	Column positions, a zero-based, one dimensional array of offsets from the printer's left margin.
ERR	Exit status:

0	OK.
1	Invalid printer device name.
2	No CPL mnemonic defined for target printer.
3	Computed print line length exceeds printer's capabilities.
4	Undefined or structurally defective display format.

The setting of ERR *will not cause an execution error in the calling program.*

pagsetup is a front end to several other cookbook functions (most notably, lpsetup and rptsetup, pages 164 and 223 respectively) whose coordinated output produces the setup data needed to generate a report page layout. These functions examine the structure and element attributes of the display format passed in DFMT$ and return the data needed to format a print line on a report. In addition to this basic function, pagsetup will generate a character string that, when printed, will produce column headings derived from the spoken language descriptions in the display format. By modifying the structure and attributes of the display format, as well as the content of the parameters passed when pagsetup is called, it is possible to change the layout of a report without having to make significant code changes.

pagsetup is one of several 4GL related functions that may be used in concert to quickly produce reports from files that have related formats and links. The following will describe pagsetup's capabilities in greater detail and illustrate the use of this function along with other 4GL functions to generate a simple report.

DISPLAY FORMAT

In order to use pagsetup, it is necessary to define a logical format called a display format. The display format acts as a template describing what must be printed on a line and how it should appear. As almost everything pagsetup needs to know can be derived from the display format's attributes, you have considerable flexibility in arranging columns on the page and allocating space to print them. Also, you have the option of either associating a public program with an element for special processing of the element's data, or allowing internal processing to format each column. Thus, you can readily alter the appearance of columns by changing the size and/or attributes of the associated elements and not have to modify your main report generator program. It is even possible to utilize a custom page layout on-the-fly by calling frmtgen (page 111) to create a temporary format to describe the page.

The following should be observed in creating the display format:

- **There must be an element in the format for each column that is to be printed**. All elements must be single occurrence types, as pagsetup cannot work with multiple occurrence elements. At least two elements must be defined or else pagsetup will abort with a format structure error (ERR=4).

- **Element positions in the format correspond to column positions on the printed page**. For example, if the data in an element called NAME is to be printed in the third column from the left, then the NAME element must be the third element in the format. Once the report has been defined you can rearrange columns on the page by merely rearranging the element order in the format.

This assumes that your report uses the other 4GL functions that work with logical formats (e.g., `4glpline`).

- **Each element must have a spoken language description if the column in which it is to be printed is to have a caption** (heading). For example, an element called `NAME` could have a spoken language description of `Customer Name`, which would become the column heading. The description will be printed exactly as it has been entered, requiring that you capitalize as needed. A spoken language description consisting solely of a tilde character (~) will be treated as a blank description. This feature has been provided because the format definition tools in IDOL-IV require that every element in a format have a spoken language description.

  The spoken language description field in IDOL-IV is limited to 20 characters, which may not always be sufficient. In such a case, you may enter a more verbose description into the special prompt (message) attribute field of the element definition. Any text in that field will take precedence over the spoken language description, even if the spoken language description is a tilde.

- **Each element in the display format must have attributes that are compatible with the data that is to be printed**. `pagsetup` examines the attributes of each element to determine how large the field will be in printed characters, how to align the field relative to the column and how to align the column heading with the printed field. In particular, numeric fields must be defined with sufficient precision for the number being displayed and should be a size sufficient to hold the largest expected total if totals are to be printed. Otherwise, column alignment and formatting problems, as well as possible overflow errors, will arise.

  Normally, numeric elements are formatted according to the default mask that is generated from the element size and precision during definition. If you wish to format numeric values with a different mask you may define that mask in the element's valid values attribute field. The alternate mask definition takes the form `DM="<MASK>"`, where `<MASK>` is any reasonable numeric mask. For example, `DM="(##,###,###.00)"` is a valid mask. Any mask defined in the valid values field will override the element's default mask and thus will dictate how much column space will be required to print the largest expected value.

  Numeric types can differ between the physical and display formats. For example, a numeric element `SALES1` could be defined as signed BCD with `4.2` precision in the physical format. It is permissible to defined `SALES1` in the display format as signed ASCII (type 0) with `12.2` precision.

The `copyfmt` function (page 84) will automatically convert the internal format as required to load `SALES1` in the display format with the value of `SALES1` in the physical format.

Elements for which pre-process attributes have been defined are evaluated for size and alignment by making a test call to the program named in the pre-process attribute. Undefined results will occur if the test call cannot be completed for any reason or the test call returns a null output. External processing of a display element's data supersedes any internal heuristics that would normally be applied by `pagsetup` (more on external processing may be found at `4glpline` on page 59).

In addition to the above, you should consider the following:

- **Since report data is ultimately derived from files whose contents will be read into physical (record) formats, it makes sense to define display format elements that have names and attributes that correspond to the physical format(s)**. Doing so allows you to utilize `copyfmt` to transfer the data from the physical format(s) to the display format with little effort. Otherwise, you will be faced with writing many statements to copy data from the physical format to the display format, greatly compromising the generality of your program and opening the door to bugs caused by improperly named formats and/or elements.

- **You can reduce space consumption on the page by truncating string data in instances where such truncation will cause no harm**. For example, the physical format for an inventory item master record may contain a `DESCRIPTION` element defined to be 50 characters in length. For reporting purposes, it may have been determined that 30 characters of the description will suffice. Thus, you would define the `DESCRIPTION` element of your display format to be 30 characters. `copyfmt` will automatically truncate the field as required.

- **If running (sub)totals are required as part of the report, create one or more (sub)total formats that are structurally identical to the portion of the physical format (not the display format) in which the number fields are located**. Doing so will permit you to accumulate (sub)totals by using the convenient `4gltotal` function (page 65; the example with this narrative also illustrates a neat trick you can use with your total format). When it is time to print the totals you can use `copyfmt` to copy them to the display format for printing. We'll illustrate this technique in an example later on.

- **When using binary SQL dates, the date size in the display format does not have to be the same as in the physical format**. For example, the physical format may define an element `DATE_OE` to be a 6 byte binary SQL date, which is able to store both the date and the time of day with five second resolution. If only the date is necessary on the report, you would define `DATE_OE` in the display format to be a 4 byte binary SQL date.

When `copyfmt` is called an automatic conversion will take place, and the printed output will be a date only.

COLUMN SIZING

Part of what `pagsetup` does is compute how much space must be allocated on the page to each column. This entails two distinct operations: determining how much space the printed data will require and how much space the column heading will need. The data space is determined from either the size of the element if string data, the mask assigned to the element if numeric, special masking that deals with binary SQL date fields, or a test call to an external program defined in the element's pre-process attribute. Absent a pre-process attribute and external program to perform the processing, internal heuristics are used to work out column sizing and alignment.

For string data, the space consumed is exactly equal in size to that of the element. That is, if the element size is 30, the column width will be 30 characters. Display space consumption by numeric elements is determined by the formatting mask, which in turn, is determined by a combination of the element's size, precision, numeric type and in some cases, padding, or by the presence of an alternative mask in the valid values attribute.

For example, if a numeric element has been defined as unsigned BCD (numeric type C) with `4.2` sizing and precision, the default mask will be `########.00` and the required space to print the field will be 11 characters. The same element defined as signed BCD (numeric type A) will have a default mask of `#######.00-` and the required space to print the field will also be 11 characters. If this element had been defined as a type 0 (signed ASCII representation) the default mask will be `#.00-`. Note that the default mask may be overridden by entering an alternate mask in the valid values attribute field.

Binary SQL dates are subjected to special treatment because they may either represent dates or dates and times. For 4 byte binary SQL dates, the mask used is `MM/DD/YY` and hence the field size will always be eight characters. If the element is 6 bytes the mask will be `MM/DD/YY HH:MI AM`, resulting in a constant field size of 17 characters. If a different date/time format is required you must define an pre-process attribute for the element and write a suitable public program to handle the formatting.

In some cases, the data space will be less than the space occupied by the column heading, in which case the longest segment of the column heading will determine the column width. Segment size is based upon both the number of rows allocated to column headings (a minimum of two) and the longest word in the element's spoken language description. `pagsetup` automatically determines how to split up a description and how to align it relative to the data.

Example:

In the following example, we present the makings of a simple report generator, which also uses other 4GL functions previously described.

```
0200 SETUP: DFMT$="#MYFMTD", REM display format
            RFMT$="#MYFMTR"; REM physical format
            TFMT$="#MYFMTT"; REM totals format
            FORMAT INCLUDE #DFMT$;
            FORMAT INCLUDE #RFMT$;
            FORMAT INCLUDE #TFMT$,OPT="DEFAULT";
            CAPROWS=0; REM use the default of 2
            IFSL=3;    REM we want at least 3 characters of whitespace
            IFSH=6;    REM and, no more than 6 between columns
            LP$= "P1"; REM printer name
            CALL "pagsetup",DFMT$,LP$,CAPROWS,IFSL,IFSH,CC$,DL$,
                                 LMC,NFLD,PITCH$,SL$,DC[ALL];
            ON ERR GOTO SETUP01,BAD_PRINTER,NO_CPL,LINE_TOO_LONG,
            BAD_FMT
0210 SETUP01: LP=UNT; REM printer channel
            OPEN (LP) LP$;
            CH=UNT;
            OPEN (CH) FILENAME$; REM this is the data file
            READ (CH,KEY="",DOM=MAIN)
1000 MAIN: READ (CH,END=DONE) #RFMT$; REM get a record
            CALL "copyfmt",RFMT$,DFMT$,1; REM copy data
            CALL "4glpline",DFMT$,0,0,PITCH$,DC[ALL],LINE$;
                REM generate a print line &...
            PRINT (LP)LINE$; REM print it
            CALL "4gltotal",RFMT$,TFMT$,0; REM add totals
            GOTO MAIN
9000 DONE: CALL "copyfmt",TFMT$,DFMT$,1; REM copy totals
            CALL "4glpline",DFMT$,0,0,PITCH$,DC[ALL],LINE$;
            PRINT (LP)LINE$; REM print totals
            PRINT (LP) 'FF',; REM eject final page
            END
```

parsdata — Parse Delimited Data Into String Array

Syntax:

```
CALL "parsdata",SRC$,SEP$,NP,PL$[ALL]
```

Call Parameters:

SRC$	String containing data. See text.
SEP$	Character(s) delimiting data segments in SRC$. See text.

Returns:

NP	Number of parameters returned in PL$[].
PL$[0]	Reflects value of NP.
PL$[1-NP]	Positional parameters extracted from SRC$. PL$[1] specifies the first parameter, PL$[2], the second, and so forth. PL$[NP] specifies the final parameter. If NP=0 only PL$[0] will be valid.
ERR	0 Okay, all returns valid. 1 Invalid source string structure, such as unbalanced quotes. See text. 2 No delimiter character specified in SEP$. *The setting of* ERR *to any of the above values will not cause an execution error in the calling program.*

parsdata parses delimited fields in a character string (SRC$) into an array of elements (PL$[]), ordered as they were in the source string. The delimiting character is specified in SEP$ and may include more than one character. If one of the fields in SRC$ includes the delimiter in SEP$ that field must be quoted (balanced quotes).

Examples:

```
SRC$="Pete,Paul,Mary,Nancy",
SEP$=",";
CALL "parsdata",SRC$,SEP$,NP,PL$[ALL];
ON ERR(0) GOTO ERROR,OKAY
```

Assuming the above call returned with ERR=0, the results would be as follows:

```
NP=4              (number of parameters returned)
PL$[0]="4"        (reflects value of NP)
PL$[1]="Pete"
PL$[2]="Paul"
PL$[3]="Mary"
PL$[4]="Nancy"
```

The next example illustrates how to handle the case where SEP$ also appears in one of the positional parameters passed in SRC$:

```
SRC$=QUO+"Peter, Paul & Mary"+QUO+",Nancy",
SEP$=",";
CALL "parsdata",SRC$,SEP$,NP,PL$[ALL];
ON ERR(0) GOTO ERROR,OKAY
```

The use of balanced quotes around the first field prevents parsdata from interpreting the comma after Peter as a delimiter. Assuming the call returned with ERR=0, the results would be as follows:

```
NP=2
PL$[0]="2"
PL$[1]="Peter, Paul & Mary"
PL$[2]="Nancy"
```

The above could also be coded as:

```
SRC$="Peter, Paul & Mary|Nancy",
SEP$="|";
CALL "parsdata",SRC$,SEP$,NP,PL$[ALL];
ON ERR(0) GOTO ERROR,OKAY
```

thus eliminating the need for quoting part of the source string.

More than one delimiter may be specified for cases where several possible delimiters might be in the source string:

```
SRC$="Pete,Paul|Mary,Nancy";
CALL "parsdata",SRC$,",|",NP,PL$[ALL];
ON ERR(0) GOTO ERROR,OKAY
```

The above example will return the same results as the first example. Note the literal use of the delimiters in place of the SEP$ variable.

`pause` Pause Program For Keypress or Timeout

Syntax:

```
CALL "pause"[,TIMOUT]
```

Call Parameters:

`TIMOUT`	Optional response timeout period in seconds. See text.

Returns:

None.

`pause` waits for a user to press any recognized control key before returning control to the calling program.[14] The optional `TIMOUT` parameter should be set to a non-zero value to enable input timeout or to zero to disable input timeout. If the `TIMOUT` variable is omitted `pause` will time out in ten seconds.

[14] See the `fkydcd` narrative (page 107) for recognized keys.

pclgpe — Hewlett-Packard PCL Compatible Graphics Printing Engine

Syntax:

```
CALL "pclgpe",LP,CPI,DPI,LPI,PLD$
```

Call Parameters:

LP	Channel opened to target printer, cannot be zero.
CPI	Printer width setting in characters per inch, must be a positive integer. The value should correspond to the selected character pitch (e.g., 10, 12, 15, etc.).
DPI	Printer graphics resolution in dots (pixels) per inch, must be a positive integer or zero. If zero, 300 DPI will be assumed.
LPI	Printer line spacing setting in lines per inch, must be a positive integer. Most printers default to 6 LPI.
PLD$	Page layout descriptor, generally arranged as follows: `OBJ,[@(C,R),]P1,P2,P3 [,OBJ,[@(C,R),]P1,P2,P3...]` See text for more information about PLD$.

Returns:

ERR	0	Okay, parameters processed.
	1	LP not a valid printer channel.
	2	Syntax error in PLD$.
	3	Invalid parameter in PLD$.
	4	Invalid resolution in DPI.

The setting of ERR *to any of the above values will not cause an execution error in the calling program.* Invalid values in CPI, DPI and/or LPI (other than invalid resolution in DPI) will cause ERR=41. Note that "Okay" status doesn't imply that the printer was able to process the print job. See text.

pclgpe is an interface to any printer that understands the Hewlett-Packard (HP) printer control language, version 4 or later (PCL4). This function provides some simple processing that can produce graphic boxes and lines on a page, as well as address the printer's "cursor" to any desired column and line. pclgpe parses the content of the PLD$ variable to determine what to do, using the basic page settings passed in CPI, DPI and LPI.

The CPI and LPI values set the page density for printed characters and thus affect all aspects of the page layout. As the values for CPI and LPI are increased, characters are packed closer together and thus should be printed at a smaller point size. PCL-compatible printers generally allow a wide latitude in these values, which facilitates the layout of many types of forms.

The DPI value determines the resolution at which the printer will generate graphic shapes in dots per inch (DPI). Most PCL-compatible printers offer a range of resolutions, and in the case of higher end models, DPI can cover a very wide range. The default for most printers is 300 DPI. DPI settings lower than 300 or higher than 1200 are only supported on a few models. The default of 300 is generally sufficient for most applications.

The actual page layout is described in the PLD$ variable. As described above, PLD$ has the general form:

```
OBJ,[@(C,R),]P1,P2,P3 [,OBJ,[@(C,R),]P1,P2,P3...]
```

where OBJ is a mnemonic that specifies the operation to be performed, @(C,R) addresses the cursor to a specific column and row (page line), and parameters P1, P2 and P3 set the attributes associated with the specified object. Not all parameters are required for all objects.

Mnemonics and parameters are delimited by commas, omission of which will cause pclgpe to exit with a syntax error (ERR=2). In most cases, cursor addressing is optional and may be omitted. Syntax is not case-sensitive and spaces between mnemonics and parameters may be inserted for clarity. Successive object definitions may be concatenated so as to generate an entire page layout with a single call.

The current version of pclgpe understands the following OBJ mnemonics:

ADC	Address (set) the printer's "cursor."
BOX	Draw a graphic rectangle.
HDL	Draw a doubled horizontal line like ═══.
HSL	Draw a horizontal line like ———.
VDF	Shade area with variable density fill pattern like ▒▒▒.
VDL	Draw a doubled vertical line like ‖.
VSL	Draw a vertical line like |.

As would be expected, higher dot per inch (DPI) resolution will generally produce higher quality graphics (subject to printer capabilities), at the expense of slower printing and increased ink or toner consumption. Detailed descriptions of each mnemonic follow.

ADC,@(C,R) Address Cursor

The ADC mnemonic will position the printer's cursor to column C and row (line) R. Coordinates are zero-based and are affected by the current CPI and LPI settings. For example, if CPI=10 and LPI=6, then ADC,@(20,18) will position the cursor 2 inches from the left margin and 3 inches below the top margin. If LPI=8 the same cursor address will position the cursor 2.25 inches below the top margin. ADC,@(0,0) will "home" the cursor to the top left corner of the page (where top and left depend on the current margin settings). ADC is the only mnemonic where the @(C,R) syntax is required; no other parameters are permitted.

BOX,[@(C,R),] W,H

The BOX mnemonic will print a graphic rectangle on the page, with the top left corner starting at the current cursor position, unless cursor addressing is inserted after the mnemonic. The W and H parameters are non-zero integers that specify the rectangle width in columns and height in rows (lines), respectively. The physical size of the rectangle will affected by CPI and LPI, since those parameters determine column and line spacing.

HDL,[@(C,R),] W,LEC,REC
HSL,[@(C,R),] W,LEC,REC

The HDL and HSL mnemonics will print a graphic doubled (═) or plain (─) horizontal line on the page, with the left end starting at the current cursor position, unless cursor addressing is inserted after the mnemonic. The W parameter is a non-zero integer that specifies the width of the line in columns, with the physical line width being affected by CPI. The LEC and REC parameters are integers that specify, respectively, the left and right end characters used to terminate the line. These characters are counted as part of the line width. Valid values for LEC and REC are as follows:

0 No character.
1 Left connect (╞).
2 Right connect (╡).
3 Four way connect (╪).
4 Top left corner (╒).
5 Top right corner (╕).
6 Bottom left corner (╘).
7 Bottom right corner (╛).
8 Top connect (╤).
9 Bottom connect (╧).

VDF,[@(C,R),] W,H,PD

The VDF mnemonic will print a filled (shaded) rectangle on the page, with the top left corner starting at the current cursor position, unless cursor addressing is inserted after the mnemonic. The W and H parameters are non-zero integers that specify the rectangle width in columns and height in rows (lines), respectively. The PD parameter is an integer in the range 1 to 100 inclusive that specifies the rectangle density as a percentage, where 1 produces a barely visible rectangle and 100 produces a solid rectangle. If the intention is to superimpose text on the rectangle, fill density should be between 10 and 20 percent for best results.

A typical filled rectangle with superimposed printing would appear as follow:

```
Acme Screw Machine Inc
2142 W 115th St
Chicago IL 60606
```

The above effect could be achieved with the following code sequence:

```
C=5;                            REM starting column for rectangle
R=5;                            REM starting row for rectangle
LL=0;                           REM text line length
NL=3;                           REM number of text lines to print
PD$="20"                        REM pattern density
CM$=",";                        REM a comma, it's used a lot
AT$=",@(";                      REM cursor addressing preamble
RP$=")"                         REM right paren, also used a lot
DIM L$[NL-1];
L$[0]="Acme Screw Machine Inc",
L$[1]="2142 W 115th St",
L$[2]="Chicago IL 60606";
FOR I=0 to NL-1;
   LL=MAX(LL,LEN(L$[I]));       REM determine length of longest line
NEXT I;
LL=LL+2;                        REM the actual rectangle width
PLD$="VDF"+AT$+STR(C)+CM$+STR(R)+RP$+CM$+STR(LL)+CM$+STR(NL)+CM$+PD$;
CALL "pclgpe",LP,CPI,DPI,LPI,PLD$;     REM draw 20% filled rectangle
C=C+1;                          REM starting column for text
FOR I=0 to NL-1;
   CALL "pclgpe",LP,CPI,DPI,LPI,"ADC"+AT$+STR(C)+CM$+STR(R+I)+RP$;
   PRINT (LP)L$[I];             REM address cursor & print
NEXT I
```

The exact effect of the VDF mnemonic is hardware-dependent and even varies within Hewlett-Packard's product line. Some experimentation may be required to determine the best density to use.

VDL,[@(C,R),] H,UEC,LEC
VSL,[@(C,R),] H,UEC,LEC

The VDL and VSL mnemonics will print a graphic doubled (‖) or plain (|) vertical line on the page, with the upper end starting at the current cursor position, unless cursor addressing is inserted after the mnemonic. The H parameter is a non-zero integer that specifies the height of the line in rows, with the physical line height being affected by LPI. The UEC and LEC parameters are integers that specify, respectively, the upper and lower end characters used to terminate the line. These characters are counted as part of the line height. Valid values for UEC and LEC are as follows:

0 No character.
1 Top connect (┬).
2 Bottom connect (┴).
3 Four way connect (┼).
4 Top left corner (┌).
5 Top right corner (┐).
6 Bottom left corner (└).
7 Bottom right corner (┘).
8 Left connect (├).
9 Right connect (┤).

Application Notes

The recommended procedure for using this function in a working program is as follows:

1) Open the printer and print an 'OPEN' mnemonic to establish the printer defaults. If you call the opnprntr function (page 185) to open the printer the defaulting operation will be automatically handled for you.

2) Define PLD$ according to the desired graphic page layout. Keep in mind that all @(C,R) coordinates are zero-based and are affected by the printer pitch and lines per inch settings, as determined by the CPI and DPI parameters.

3) Call this function to output the graphic page. Upon return, be sure to check the value of the ERR system variable for a non-zero value, which would indicate that some kind of processing error occurred.

4) Print the rest of your page using the usual *Thoroughbred* syntax. Each page must be followed by a formfeed in order to flush the printer's buffer to paper and eject the page.

It is permissible to make multiple calls to this function for such purposes as addressing the printer's cursor. You may also intersperse calls to `pclgpe` with ordinary `PRINT` statements to print text, for example, to address the printer's cursor before each `PRINT` statement.

5) Following the printing of the final page, print a `CLOSE` mnemonic to flush the buffer, eject the page and restore the printer to its default state. Finish by closing the printer channel to despool the job.

popup — Generate Pop-Up Message Box

Syntax:

```
CALL "popup",TEXT$[,MLL[,ROW[,FGCLR$[,BGCLR$]]]]
```

Call Parameters:

TEXT$	Unformatted text of message to be displayed.
MLL	Maximum text line length to which "raw" text will be formatted. Optional, see text.
ROW	Optional row on which to anchor text box. If zero or omitted box will be vertically centered.
FGCLR$	Optional foreground (text) color, e.g., 'WHITE'. If null or omitted white will be assumed.
BGCLR$	Optional background color, e.g., 'BLUE'. If null or omitted blue will be assumed.

Returns:

TEXT$	Name of window created by this function.

popup is a simplified front end to the msgbox function (page 176). popup handles the formatting of the raw text into a style suitable for use by msgbox and then calls msgbox with preselected display parameters. Thus it is possible to generate a pop-up advisory to the user with relatively little effort. Do not insert mnemonics into the raw text, as they will cause a formatting error.

If desired, you may override the default parameters by supplying appropriate replacements. In particular, the optional MLL parameter may be used to "shape" the box as desired. By default, MLL is equal to WIDTH/2 characters, where WIDTH is the physical screen width in columns. Hence, the default box size will be (WIDTH/2)+4 columns (the additional columns are for the box borders). Using MLL, popup will format the raw text to generate the appropriate number of lines.

Example:

```
CALL "popup","File transfer completed. [Key]";
CALL "pause",60;
WINDOW POP;
```
…continue processing

The user will see a text box similar to the following vertically and horizontally centered on the screen:

The text box itself will display with a dark blue background and white foreground on terminals capable of color. You can substitute any color combination you wish with the optional `BGCLR$` and `FGCLR$` parameters.

portopen — Open and Lock Serial Port for Raw Access

Syntax:

```
CALL "portopen",PORT$,SPEED,CBITS,SBITS,PARITY,RCH,WCH
```

Call Parameters:

PORT$ Serial port device name, such as `/dev/tty1a1`. See text.

SPEED Bits per second (baud) rate at which port is to run, passed as an index value:

SPEED	BPS Rate
0	1200
1	2400
2	4800
3	9600
4	19200
5	38400
6	57600
7	76800
8	115200
9	230400

Speeds above 38400 are not supported by all systems. See text.

CBITS Number of bits per character, either `7` or `8`.

SBITS Number of stop bits, either `1` or `2`.

PARITY
- `0` No parity.
- `1` Odd parity.
- `2` Even parity.

Returns:

RCH Input (read) channel opened to serial port. Not valid if any error occurs.

WCH Output (write) channel opened to serial port. Not valid if any error occurs.

ERR
- `0` OK, channels `RCH` and `WCH` are valid.
- `1` Invalid or inaccessible device name—cannot open. See text.
- `2` Port locked by another *Thoroughbred* task.

The setting of `ERR` *will not cause an execution error in the calling program.*

portopen is a UNIX/Linux function that creates a full duplex data path between the calling program and a serial port on the host system, allowing a *Thoroughbred* task to communicate with serial interface data acquisition devices like magnetic stripe card readers and bar code scanners, as well as with modems. portopen will attempt to acquire exclusive control over the port specified in PORT$ and if successful, will attempt to configure the port to the speed and data format specified by the SPEED, CBITS, SBITS and PARITY parameters. Input and output channels will be returned in RCH and WCH respectively. portopen performs no other conditioning of the target port and does not check the validity of the speed and data format values against the system's capabilities. An improper combination of these parameters may cause the serial port to behave in an unpredictable manner.

In order to successfully acquire control of the target serial port, the calling task must have read and write permission on the device file name. On most UNIX/Linux systems the device file should be set to rw-rw-rw- permissions with the chmod 666 <device_name> command. For example, if the serial port is attached to the /dev/tty1a16 device file you would execute chmod 666 /dev/tty1a16. Less desirably, you could give ownership of the device to the user who started the *Thoroughbred* task.

Once channels have been opened to the serial port communication can begin. Data should be read from the input channel with READ RECORD and written to the output channel with WRITE RECORD. When all communications have completed both channels should be closed to release the port. It is recommended that your program close WCH before closing RCH. This sequence will assure that the write buffers to the port are flushed before control of the port is relinquished.

Some versions of UNIX (such as HP-UX and SCO) assign two device names to each serial port, a "modem control" device name and a "non-modem control" name. *Use only the non-modem control name with this function*. With SCO, for example, you would use tty1a16 (the non-modem control device name) instead of tty1A16 (which is the modem control name for the same port). Also note that the locking of the serial port occurs only within *Thoroughbred* and does not prevent other UNIX programs or operating system calls from concurrently accessing the port. Use caution with commands like cu, stty and uucp.

Example:

```
PORT$="/dev/tty1a16",
SPEED=8,
CBITS=8,
SBITS=1,
PARITY=0;
CALL "portopen",PORT$,SPEED,CBITS,SBITS,PARITY,RCH,WCH;
ON ERR(1,2) GOTO OK,INVALID_PORT,PORT_LOCKED
```

The above example attempts to open a connection to the serial port `/dev/tty1a16`, running it at `115200` bits per second, with `8` bits per character, `1` stop bit and no parity. If control of the port can be acquired, an input channel will be returned in `RCH` and an output channel in `WCH`.

See also `modmopen` (page 172), `modmctrl` (page 169) and `portread` (page 212).

portread — Read Data From Serial Port

Syntax:

```
CALL "portread",RCH,TIMOUT,MODE,DATA$
```

Call Parameters:

RCH	Input (read) channel opened to serial port.	
TIMOUT	No data timeout period in seconds. 0<TIMOUT<32768	
MODE	0	Acquire data with READ RECORD(.
	1	Acquire data with READ(.

Returns:

DATA$	Returned data, not valid if any error occurs.	
ERR	0	OK, DATA$ is valid.
	1	Timed out with no data.
	2	Aborted with BASIC escape (usually **[CTRL][X]**). See text.

The setting of ERR *will not cause an execution error in the calling program.*

portread is a UNIX/Linux function that reads data from a serial port, such as opened with the portopen or modmopen functions. If MODE=0 ("raw" mode) portread will return unexpurgated data from the device attached to the serial port. On the other hand, if MODE=1 end–of–line characters such as <CR> and/or <LF> will be stripped from the data stream. DATA$ will be null if this function times out or is interrupted with the BASIC escape key (as defined in TCONFIGW—usually CTRL-X). Note that TIMOUT cannot be zero: a minimum of one second must be allowed to determine if data is present.

Example:

```
TIMOUT=5,
MODE=0;
CALL "portread",RCH,TIMOUT,MODE,DATA$;
ON ERR(1,2) GOTO OK,NO_DATA,ABORTED
```

The above example attempts to acquire raw data from the serial port opened on RCH and waits up to five seconds for data to be made available.

See also modmopen (page 172), modmctrl (page 169) and portopen (page 209).

ppparse — Parse Pre/Post Process Element Attribute Data

Syntax:

```
CALL "ppparse",EFMT$,EELM,PP,PP$[ALL]
```

Call Parameters:

EFMT$	Format name in #LLNNNNNN style..	
EELM	Element number from which to parse pre/post process (PP) data.	
PP	0	Parse pre-process attribute.
	1	Parse post/process attribute.

Returns:

PP	Number of parameters returned in PP$[].	
PP$[0]	Reflects value of PP.	
PP$[1-PP]	Positional parameters specified in PP attribute string, subject to possible translation (see text). PP$[1] specifies the first parameter, PP$[2], the second, and so forth. PP$[PP] specifies the final parameter. If PP=0 only PP$[0] will be valid.	
ERR	0	Okay, all returns valid.
	110	Unable to INCLUDE format named in EFMT$.
	111	Invalid element number.
	112	Invalid PP attribute string structure.

The setting of ERR *to any of the above values will not cause an execution error in the calling program.*

ppparse parses the selected PP attribute of the selected element in the format named in EFMT$ and returns the parameters in the string array PP$[] in the same order as specified in the PP attribute.

The general form of a PP attribute string is as follows:

```
n,<parm1>[,<parm2>[,<parm3>]]]...
```

where n is either 0 or 1 (required within IDOL-IV), and <parm1>, <parm2>, <parm3>, etc., are comma-delimited parameters. Only <parm1> is required and conventionally names a program that is to be executed to process the element. <parm2>, <parm3>, etc., if present, would be passed into the program for processing. If any parameter includes one or more commas, that parameter must be surrounded by quotes.

It is permissible for any parameter, including <parm1> to be null. For example, a PP attribute such as the following is acceptable:

```
0,,,abc
```

The above would return a null string in PP$[1] and PP$[2], and abc in PP$[3].

ppparse will apply substitution processing when any of the following case-insensitive parameters are encountered in any position after <parm1> in the PP attribute string:

?DTI?	Element's date type attribute, 0 if not a date.
?ELM?	Element number being processed.
?ETI?	Element's entry type, 0-3 inclusive.
?FMT?	Format name being processed, with a leading octothorpe.
?HLP?	Help module name if defined.
?KEY?	Key field attribute, 0 if not a key (false), 1 if true.
?LEN?	Display length using the default (DNM) mask.
?MSG?	Text in special prompt (message) attribute if any.
?MSK?	Default display mask.
?NTI?	Numeric type indicator, 0-9 inclusive or A-F inclusive. Unambiguous use of this substitution mandates evaluating the numeric precision attribute.
?OCC?	Number of element occurrences, 0 if a single occurrence element.
?PRC?	Numeric precision, will be -1 if the element is non-numeric.
?SIZ?	Storage size in bytes of the element, not necessarily equal to the display length.
?VVS?	Valid values attribute string if defined.

Examples:

```
EFMT$="#OPPMMAS",
EELM=12,
PP=0;
CALL "ppparse",EFMT$,EELM,PP,PP$[ALL];
ON ERR(0) GOTO ERROR,OKAY
```

PP=0 tells ppparse to parse the pre-process attribute associated with the twelfth element in the OPPMMAS format. PP+1 would cause ppparse to process the post-process attribute. Since the value of PP might have been changed by a previous ppparse call, it is essential to correctly condition PP prior to making a new call.

Assuming the example call to `ppparse` returned with `ERR=0`, and assuming the pre-process attribute contained `0,ppinput,?fmt?,?elm?,37,21`, the results would be as follows:

```
PP=5                REM number of parameters returned in PP$[]
PP$[0]="5"          REM reflects value of PP
PP$[1]="ppinput"    REM public program name to be called
PP$[2]="#OPPMMAS"   REM format being processed
PP$[3]="12"         REM element number being processed
PP$[4]="37"         REM positional parameter
PP$[5]="21"         REM positional parameter
```

`ppparse` checks the leading numeral in the `PP` attribute string to determine that it is either `0` or `1`, but does not return it in `PL$[]`.

prntscrn Print Visible Screen

Syntax:

```
CALL "prntscrn",LP
```

Call Parameters:

LP	Channel number of channel that has been opened to the target printer. 0<LP<32765.

Returns:

None.

prntscrn provides a convenient method of generating a screen dump hard copy. The output of prntscrn is a literal physical screen copy as returned by the getscrn function (page 127). Following the dump a formfeed will occur. It is permissible to dump to a file by opening a "printer" that has been defined as "output to file" in the IPL file that started the task.

prntscrn attempts as much as possible to reproduce display attributes. For example, where possible, foreground video will be printed in bold face, underline in underline and so forth. prntscrn can translate graphics characters such as ├ and ┌ if the graphics mnemonics G0 through GF inclusive have been defined for the target printer. Otherwise, graphics characters will be replaced by ASCII equivalents.

prntscrn will exit with error 41 (integer range) if LP is not an integer, is zero, is negative, or if the device referred to by LP is not a printer device.

pscscmp — Compare Pattern String to Character String

Syntax:

```
CALL "pscscmp",CS$,CP$
```

Call Parameters:

`CS$` The character string being checked, may be null.

`CP$` The comparison pattern, which may contain ordinary ASCII characters, wildcard characters `?`, `*`, `[]` (balanced brackets), and Boolean symbols. The presence of the latter will apply special logic to the comparison:

- `&` Logical AND. `a&b` means result is true if `a` AND `b` are present in the character string in `CS$`. Logical AND is of no particular value unless wildcard characters are part of the comparison pattern. See text.
- `|` Logical OR. `a|b` means result is true if `a` OR `b` is present in `CS$`. As with logical AND, logical OR is of no value unless wildcard characters are part of the comparison pattern. See text.
- `!` Logical NOT. If `!` is the first character in `CP$` the result of the comparison will be inverted. The presence of `!` in any other character position will be ignored and will be treated as a regular character.
- `^` Ignore case. If `^` is the first character in `CP$` or the second character if logical NOT has been specified, comparisons will carried out with case-insensitivity. `^` in any other character position will be treated as a regular character.

Evaluation of the above is from left to right, with logical AND taking precedence over logical OR.

Returns:

`ERR` `0` Result of comparison is TRUE. The meaning of TRUE is inverted if logical NOT was specified.

1 Result of comparison is FALSE. The meaning of FALSE is inverted if logical NOT was specified.

If both CP$ and CS$ are null TRUE will be returned, as the two strings are considered to be "equal" in such a case. *The setting of* ERR *to either of the above values will not cause an execution error in the calling program.*

pscscmp extends the functionality of the *Thoroughbred* LIKE relational operator to add full Boolean comparison testing of character strings, with a simple TRUE or FALSE result being returned. In addition to performing a basic A$ LIKE B$ form of comparison, pscscmp can evaluate more complex comparisons with minimal coding effort on the part of the programmer.

The use of wildcards in the comparison pattern in CP$ can greatly enhance the effectiveness of pscscmp, and, in fact, are necessary in order to gain any value from the logical AND and OR functions. Wildcards are defined as follows:

? Match with a single character. That is, ? replaces one character, allowing a pattern such as a?cd to match abcd or aZcd.

* Match all characters. That is, * replaces an arbitrary number of characters, allowing a pattern such as *b* to match this is a big string. The pattern *s*r* would also match this is a big string. Evaluation is always from left to right. Hence the pattern *r*s* will not match this is a big string.

[] Match character range. Balanced brackets permit the specification of a range of characters in the pattern, and when judiciously used with other wildcards and/or Boolean logic, can result in the evaluation of strings for a number of possibilities. Balanced brackets are a special form of OR logic, in which a substitution occurs for any character that falls within the range specified in the brackets. For example, the pattern [ab]cde will match with any four character string that begins with a OR b and finishes with cde. *[ab]cde* will match with any string where the combination acde or bcde is present, such as 123acde456 or 789bcde. Note that the brackets must be used in pairs (i.e., balanced) in order for this type of comparison to succeed.

Examples:

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="*aaa*&*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison will return TRUE (ERR=0). The presence of the * wildcard will produce a "match with anything in that position" type of logic. Therefore, the expression in CP$ may be interpreted as, "The result will be true if the substrings aaa AND 111 are present in CS$."

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="*aaa*|*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison will also return TRUE. In this case, the expression in CP$ may be interpreted as, "The result will be true if aaa OR 111 is present in CS$."

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="*aAa*&*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison will return FALSE (ERR=1), as the pattern *aAa* does not match anything in CS$.

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="^*aAa*&*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison will return TRUE, as case-sensitivity was turned off for the entire comparison by the presence of ^ as the first character in CP$.

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="!^*aAa*&*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison, although almost identical to the previous one, will return FALSE due to the presence of the logical NOT symbol at the start of the comparison string.

```
CS$="aaa111bbb222ccc333ddd444eee",
CP$="*a[aA]a*&*111*";
CALL "pscscmp",CS$,CP$
```

The above comparison will return TRUE, as the use of balanced brackets allows the second a to be upper or lower case. This example illustrates how to suppress case sensitivity for a limited part of the comparison pattern.

realtime — Synchronize Task Date and Time to Operating System

Syntax:

```
CALL "realtime"
```

Call Parameters:

None.

Returns:

None.

`realtime` sets the *Thoroughbred* `DAY` and `TIM` system variables to the date and time maintained by Linux or UNIX. The resulting time value is corrected to the local time zone of the calling task. `realtime` should be used to restore `DAY` and `TIM` after they have been changed to some other value.

rmargin — Compute Margin to Right Justify Text String

Syntax:

```
CALL "rmargin",STRING$,WIDTH,COL
```

Call Parameters:

STRING$	Text string to be right justified. 0<LEN(STRING$)<=WIDTH.
WIDTH	Maximum display columns on intended display device.

Returns:

COL	Computed zero-based starting column for text string. If a negative value is returned STRING$ is too long for the display device width.

rmargin computes the zero-based starting column position needed to right justify a text string on a display device. Embedded mnemonics in the string are ignored, except for 'DB', 'DT' and 'EP', which will cause rmargin will compute COL based upon WIDTH ÷ 2. Avoid using any mnemonics that can affect cursor positioning (e.g., @(C,R) or 'LI').

Examples:

```
STRING$='SF'+"This is a test."+'CL';
CALL "rmargin",STRING$,80,COL
```

The above example results in COL=65.

```
STRING$='EP'+'SF'+"This is a test."+'CL';
CALL "rmargin",STRING$,80,COL
```

The above example results in COL=25, the required position to right justify an expanded string. Refer to the scrnsize function (page 226) for a method of determining the current display width of a terminal.

rprint — Right Justify and Print Text String

Syntax:

```
CALL "rprint",STRING$,ROW
```

Call Parameters:

STRING$	Text to be displayed.
ROW	Screen row on which to display STRING$. 0<=ROW<24.

Returns:

None.

rprint displays the text in STRING$ right justified on ROW, relative to the current window. Mnemonics in the text string are processed as expected and do not affect positioning (unless a cursor positioning sequence is part of the string). The EP, DT and DB mnemonics, which affect text size, are recognized by this function and will work as expected on most terminals. Use caution with mnemonics that alter an area of the display (such as the CE mnemonic).

Example:

```
CALL "rprint",'SB'+"Date "+'SF'+NTD(CDN,"MM/DD/YYYY"),0
```

This example displays the current date in the top right hand corner of the screen.

See also the cprint function (page 86).

rptsetup — Generate Report Setup Parameters From Format

Syntax:

```
CALL "rptsetup",RFMT$,NCHR,CFL,FLDP$,NFLD,CH$[ALL],CL[ALL],DL[ALL]
```

Call Parameters:

RFMT$	Format name from which parameters will be generated in #LLNNNNNN style. See text.
NCHR	Maximum number of rows on which to generate column descriptions (headings). If less than 2, 2 will be assumed.

Returns:

NCHR	Actual number of column description rows that was generated, never more than the initial value passed in NCHR.
NFLD	Number of elements in the display format.
CFL	Combined print length of all elements in format. See text.
FLDP$	Flags describing data field position relative to column, one per field from left to right, interpreted as follows: 0 Left justify data, usually strings and unformatted numbers. 1 Right justify data, usually formatted numbers, dates or time strings.
CH$[]	Descriptions (headings) associated with each element, with trailing blanks stripped. This is a two dimensional, zero based, row major array.
CL[]	Effective character length for each description in CH$[], based upon the longest component of each description.
DL[]	Display length for each element, based upon each element's default (DNM) display mask.
ERR	Exit status: 0 OK. 1 Undefined format. 2 Insufficient number of elements in format (must be at least 2).

The setting of ERR *will not cause an execution error in the calling program.*

rptsetup is a function that examines the structure and element attributes of the display format passed in RFMT$ and return the data needed to create and format a print line on a report.

The sizing of data fields is determined by evaluating for a pre-process attribute associated with each element and making a test call is made to the associated public program, using the size of the return data as the field size. If no pre-process attribute has defined, heuristics are applied to the element attributes to determine what the expected output will be.

In addition to the sizing function, `rptsetup` will generate a character string array that will produce column headings derived from either the special prompt (message) attribute of each element (if defined) or from the spoken language descriptions in the display format. If no message attribute has been defined and a spoken language description consists only of a tilde or is empt, no column heading will be generated for the element's column.

While it is possible to use `rptsetup` as a component of your programs, it is more convenient to call `pagsetup` to generate a report layout, as the latter handles the computations needed to work out the layout details.

See also `pagsetup` on page 192.

rvsname — Reverse First and Last Names In String

Syntax:

```
CALL "rvsname",NAME$,RNAME$
```

Call Parameters:

NAME$	String variable containing name to be reversed.

Returns:

NAME$	Reversed name. See text for details.

`rvsname` swaps the last whole word in a character string with the remainder of the string. Although any string may be processed by this function, it is meant to reverse the word order of a person's name so that `John Q. Adams` is transformed into `Adams John Q.` `rvsname` strips all punctuation, as well as trailing blanks.

`rvsname` recognizes some common name suffixes such as `Jr.`, `Sr.`, `II.` and does not consider them to be the last whole word in the string. Thus, `Martin Luther King, Jr.` is transformed into `King Jr Martin Luther` and the `Jr` suffix is kept with the last name as it should.

Example:

```
NAME$="Richard M. Nixon";
CALL "rvsname",NAME$,RNAME$;
PRINT RNAME$
```

The above code fragment will print `Nixon Richard M` to the terminal.

scrnsize — Get Logical Screen Display Size

Syntax:

```
CALL "scrnsize",WNAME$,WIDTH,HEIGHT
```

Call Parameters:

WNAME$	Window name. If null, the currently active window or I/O area is assumed. To return the physical screen size use WNAME$="0" as the window name.

Returns:

WIDTH	Number of columns in logical screen or zero if the window name in WNAME$ is invalid.
HEIGHT	Number of rows in logical screen or zero if the window name in WNAME$ is invalid.

scrnsize returns the number of columns and rows in any defined window or I/O region within a window, including the full screen. By default, when *Thoroughbred* starts the windows driver, it creates a full screen window named "0" (the "main" window) whose size is the same as the physical screen size of the terminal. If no other windows or I/O regions are active, calling scrnsize with a null window name will return either the physical terminal boundaries (e.g., 80 columns by 24 rows for a WYSE-60) or the size of an active I/O region defined in the main window. The same effect is achieved by specifying "0" as the window name. If a different window is active and WNAME$ is null, then the size of that active window's I/O region will be returned. To get the size of a defined but inactive window you must specify that window's name in WNAME$.

Examples:

```
CALL "scrnsize","",WIDTH,HEIGHT
```

The above example returns the size of the currently active window.

```
CALL "scrnsize","abc",WIDTH,HEIGHT
```

The above example returns the size of the window name abc if it has been defined. If it has not, both WIDTH and HEIGHT will be zero.

selprntr — Select A Printer Device

Syntax:

```
CALL "selprntr",LP$,D$,ROW,TIMOUT[,TFLAG]
```

Call Parameters:

LP$		Default printer or null. See text.
ROW		Screen row for top border of selection box. If zero, selection box will be vertically centered on the screen.
TIMOUT		Input timeout period in seconds; zero disables timeout.
TFLAG	0	Any device may be selected; default.
	1	Only printers and file output may be selected.
	2	Only printers may be selected.

Returns:

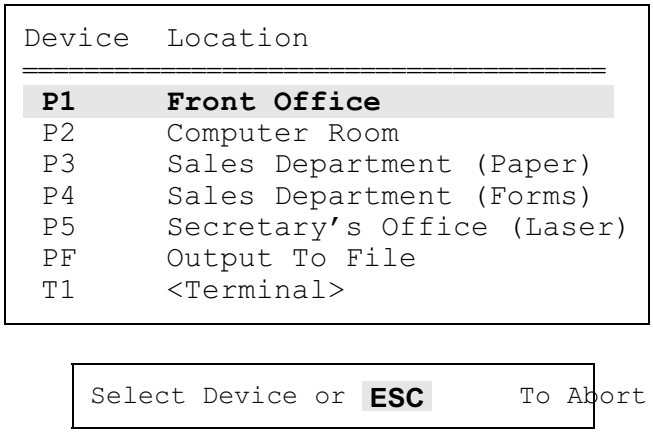
LP$		Device name of selected printer; null if aborted or timed out.
D$		Physical location description of selected printer; null if aborted or timed out. See text.
TFLAG	0	Selected device is spooled.
	1	Selected device is direct connect (non-spooled).
	2	Selected device is slaved to a terminal.
	3	Selected device is the user's terminal.
ERR	0	OK, device selected.
	1	User aborted with **[ESC]**.
	2	Selection timed out.

selprntr provides a convenient printer selection user interface that is portable across most *Thoroughbred* installations. When called, selprntr will examine the available printer devices configured into *Thoroughbred* and open a dialog box listing the printers that have been found. Depending on the value of the optional TFLAG parameter, "output to file" and terminal selection options may appear. If a default printer has been passed in the LP$ parameter it will be highlighted by a reverse video bar. Otherwise, the highlight bar will be on the first device found. These actions are the user's cue to make a selection.

In many cases, it is desirable to establish a default printer for a user based upon where s/he is located in the building.

Such a default may be passed through LP$ as either a *Thoroughbred* printer device name, such as P3, or a number that selprntr can interpret as an index into the list of available devices (e.g., "1" defaults to the first listed device, "2" to the second, and so forth). selprntr will silently ignore any default value that doesn't make sense in the execution environment. For example, "PQ" will be ignored if no printer device designated as PQ existed in the environment defined by the IPL file used to start *Thoroughbred*.

A typical selprntr selection display might appear as follows:



The T1 device name will vary; it is the value of FID(0). The location data is derived from the configurable printer table maintained in IDOL-IV (see selection 13 from the IDOL-IV Utilities Menu). This information should have been entered at the time the printers were defined.

The user may select the desired device by using the up and down arrow keys (▲ or ▼), followed by ⏮, or may press **[ESC]** to abort, with the ERR system variable indicating the exit status. *The setting of* ERR *will not cause an execution error in the calling program.* Assuming that a selection has been made, the actual printer device name (e.g., P1) will be returned in LP$ and the physical location description as displayed in the selection box will be returned in D$. Otherwise, these variables will be null. Note that selprntr does not actually open a channel to the selected device nor does it determine if the user has permission to access the device. The calling program must handle those chores.

The "output to file" option (device PF) takes advantage of a somewhat obscure feature of *Thoroughbred*: the ability to direct printer output to something other than an actual hardware port or the operating system's print spooler.

An ordinary printer definition in the IPL file designates a device filename or a spooler destination as the output path, resulting in output being directed to a particular printer. The PF device's definition uses the UNIX cat command to send the output to /dev/null—which in itself isn't very useful. However, it is possible to redirect output to an actual file by adding the DEV= option to the OPEN statement. For example, you can direct printer output to a file named dump by using the following syntax to open a channel to PF:

```
OPEN (CH,OPT="INITTAB",DEV="132,,1,5,cat > tmp/dump") "PF";
```

The effect of this statement will be to open a channel to PF but actually send the output to the tmp/dump file. The 132 value defines the maximum line length: attempting to print more than 132 characters per line will cause an end-of-record error (the maximum permissible line length is 32767). The OPT="INITTAB" clause loads the mnemonic definitions for the PF device from the IDOL-IV printer definition table. Most of these mnemonics are null, resulting in no escape sequences being written into the output file.

If terminal output is selected the OPEN statement that opens the channel to the terminal should be immediately followed by an 'EM' mnemonic to avoid undefined mnemonic errors when output sent to the terminal contains printer control mnemonics (e.g., 'DHON'). Note that the effect of the 'EM' mnemonic will be lost if the program drops into console mode, as from an error.

The optional TFLAG parameter may be used to prevent the selection of print to file or terminal output when such a selection would be inadvisable for a particular program (e.g., printing paychecks). If TFLAG is passed to selprntr it will return with a numeric value indicating the nature of the connection to the selected device. This information can be used to alter the way in which a program interacts with the selected device.

Example:

```
LP$= "P2",
TIMOUT=300,
TFLAG=0;
CALL "selprntr",LP$,D$,0,TIMOUT,TFLAG
ON ERR(0) GOTO ABORT,CONTINUE
```

In this example, P2 is set as the default printer and a selection timeout period of five minutes is established. Upon return, execution will branch to the CONTINUE statement if the user selected a printer, or to the ABORT statement if she escaped or selection timed out. Because the optional TFLAG parameter was zero there were no restrictions placed on the selection of file or terminal output.

selsort — Select Sort From MSORT or ISAM File

Syntax:

```
CALL "selsort",LINK$,TIMOUT,PROW,SORTNAME$,KEYLEN,DESCRP$,ELM[ALL]
```

Call Parameters:

LINK$	Link name in LLNNNNNN format from which a file and format name will be derived. See text.
TIMOUT	No input timeout period in seconds, zero for no timeout.
PROW	Screen row on which to display prompts and messages.

Returns:

SORTNAME$	Selected sort name, which can be used with a SRT= clause in EXTRACT, FIND and READ directives.
KEYLEN	Key length associated with selected sort.
DESCRP$	Description of selected sort as seen by user during selection process.
ELM[]	List of format elements making up selected sort. ELM[0] returns the number of elements and ELM[1] through ELM[ELM[0]] list the element numbers in the order in which they are concatenated to form the sort key structure. No list will be returned if selsort cannot resolve the selected sort to a format structure.
ERR	0 OK, all returned values are valid. 1 Aborted by user. 2 Timed out. 110 Link or format associated with link not defined. 111 File associated with link not accessible. 112 File associated with link not MSORT or ISAM. 113 Inconsistent sort structure. *The setting of* ERR *will not cause an execution error in the calling program.* See text for more information on what status codes 110 through 113 mean.

selsort implements a convenient method of picking a sort sequence from an MSORT or ISAM file in which secondary sorts have been defined.[15] When called, selsort presents the user with a pop-up window, in which a list of available sorts is displayed.

[15] It is possible to use selsort with a file with no secondary sorts—only the primary sort will appear in the selection window.

The user may pick a sort by highlighting it with a reverse video selection bar and pressing ⏮, or press **[ESC]** to abort.

A number of requirements must be met if `selsort` is to be useful in your programs. `selsort` gets the information it needs by extracting file and format name information from the link named in `LINK$`. Using that information, `selsort` associates the structure of each sort in the file with elements in the format. The spoken language descriptions associated with these elements are used to build the display seen in the selection window. Therefore, spoken language definitions, such as `Customer Name` or `ZIP Code`, must be defined in the format associated with the link named in `LINK$`.

Status codes `110`, `111` or `112` will be returned if the link itself has not been defined or is associated with a format that cannot be `INCLUDE`d, the file associated with the link is not accessible in the execution environment or is not an `MSORT` or `ISAM` type. Status code `113` will be returned if the sort structure of the file associated with `LINK$` cannot be resolved to the element structure of the format associated with `LINK$`. This may happen when the format definition has been changed and new sorts have not generated to correspond with the format changes. `ERR=113` will also occur when a sort has been built from one or more substrings derived from several elements, in which case `selsort` will not be able to relate sort segments to individual elements.

Example:

```
1000 LINK$="ICIMMAS",
     LINK1$=LINK$;
     CALL "statlink",LINK1$,ICMASF$,ICMAS$;
     FORMAT INCLUDE #ICMAS$
     ICMAS=UNT;
     OPEN (ICMAS) ICMASF$;
     CALL "selsort",LINK$,300,22,K$,KEYLEN,DESCRP$,ELM[ALL];
     ON ERR(0,1,2) GOTO ERROR,OK,ABORT,TIMED_OUT
1010 ERROR: ON ERR(110,111,112) GOTO STRUCT,NOLINK,NOFILE,NOSORT
1020 STRUCT: PRINT LINK$,": Inconsistent sort structure.";END
1030 NOLINK: PRINT LINK$,": Link or format not defined.";END
1040 NOFILE: PRINT LINK$,": Link file not found.";END
1050 NOSORT: PRINT LINK$,": Link file not MSORT/ISAM type.";END
1060 OK: PRINT "You have selected the ",DESCRP$," sort.";
     CALL "buildkey",ICMAS$,K$;
     READ (ICMAS,SRT=SORTNAME$,KEY=K$,DOM=NOTFOUND) #ICMAS$
…program continues…
```

See the `buildkey` function (page 70) for more information on generating secondary sort keys from formats.

sendem — Send Electronic Mail

Syntax:

```
CALL "sendem",FROM$,RLIST$,BLIST$,CLIST$,SUBJ$,BODY$,ALIST$
```

Call Parameters:

FROM$	Sender's E-mail address. If null, the user information returned by INF(3,2) and INF(3,3) will be utilized. An E-mail address passed in FROM$ must be in the form sender@domain.
RLIST$	Recipient's E-mail address in the form user@domain. Multiple recipients may be specified by separating each address with a comma, e.g., user1@domain1,user2@domain2, etc.
BLIST$	Blind carbon copy (BCC:) recipient's E-mail address in the form user@domain. Multiple BCC: recipients may be specified by separating each E-mail address with a comma, e.g., user1@domain1,user2@domain2, etc. See text.
CLIST$	Carbon copy (CC:) recipient's E-mail address in the form user@domain. Multiple CC: recipients may be specified by separating each E-mail address with a comma, e.g., user1@domain1,user2@domain2, etc. See text.
SUBJ$	Message subject, cannot be null.
BODY$	Message body text, cannot be null. See text.
ALIST$	Optional name of a file to attach to the message, null if no file is to be attached. Multiple files may be specified by separating each filename with a comma. See text.

Returns:

ERR	0	OK, message passed to local system's mail facilities.
	1	No recipient(s) specified.
	2	No subject specified.
	3	No body text.
	4	Unable to send message due to operating system error. See text.

sendem provides a convenient way to transmit electronic mail from any Thoroughbred application running on any Linux or UNIX system on which standard mail facilities have been installed and configured. The local mail user agent (MUA) must be read- and execute-accessible to all system users using the command mailx—if necessary, create a symbolic link with that name to the local MUA program, and must understand basic mailx syntax.

sendem's BCC:, CC: and file attachment capabilities are enhanced features that depend on the ability of the local MUA to support them. Any current version of Linux will include the enhanced mailx MUA (formerly called nail), which supports BCC:, CC: and attachments. The standard version of mailx on SCO OpenServer is based on an older variant that does not support these features.[16] Regardless of the MUA that has been installed, sendem will make a reasonable amount of effort to send the message, returning 4 in the ERR system variable if the local MUA rejects the message or is not accessible in the execution environment.

sendem does not check the structure or validity of any E-mail addresses, nor does it determine if any file that is to be attached to the message is actually accessible. To assure that an attachment can be opened, read access on the file must be available to the user running the program from which sendem has been called, and a fully qualified pathname should be specified. Without a fully qualified pathname, the (non-recursive) search for the attachment will be limited to the subdirectory specified in the DIR Thoroughbred system variable.

The message body itself may be ordinary text, up to the maximum amount that a string variable can hold. The message is sent in plain text format according to the localized character set of the host machine. To insert linefeeds into the text embed the ASCII linefeed character <LF> ($0A$) at the desired point. Two successive linefeeds will result in a paragraph break.

Example:

The following code will send an electronic message from laura@somewhere.com to pete@somewhere.com and mary@somehere.com, BCC: to sam@acme.com, CC: to kathy@somewhere.com and will attach the file 1st_quarter_sales:

```
FROM$="laura@somewhere.com",
RLIST$="pete@somewhere.com,mary@somewhere.com",
BLIST$="sam@acme.com",
CLIST$="kathy@somewhere.com",
SUBJ$="First Quarter Sales Results",
BODY$="First quarter sales results attached."+$0A$+$0A$+"Laura",
ALIST$="1st_quarter_sales";
CALL "sendem",FROM$,RLIST$,BLIST$,CLIST$,SUBJ$,BODY$,ALIST$;
ON ERR(0,4) GOTO PARM_ERROR,OKAY,CANT_SEND
```

[16] It is possible to download and compile the mailx source to add BCC:, CC: and attachment support to OpenServer. Be aware that the enhanced version of mailx cannot read the MMDF style mailboxes created by SCO's lmail delivery program. If full compatibility is required procmail must be used in place of lmail. A compiled and tested version of mailx is available from *BCS Technology Limited*.

seterr — Condition ERR System Variable

Syntax:

```
CALL "seterr",ER
```

Call Parameters:

ER	Value to which the ERR system variable is to be set. 0<=ER<256

Returns:

None.

seterr can be used to set the ERR system variable to any desired value, taking advantage of the fact that ERR doesn't change until a different error condition occurs during program execution. Hence you may use seterr in your public programs to condition ERR to reflect program results prior to exit. *The conditioning of* ERR *will not cause an execution error in the calling program*. However, it is possible to use the ON ERR(...) GOTO construct to route program execution. The value passed to seterr must be an integer in the range of zero to 255 or else a real ERR=41 (integer range) will occur.

Examples:

```
CALL "seterr",50
```

The above example will cause ERR to equal 50.

```
01000 MAIN: PRINT "ESC To Abort: ",;
      CALL "seterr",0;
      WHILE ERR<31 OR ERR>32;
         CALL "fkydcd",60;
      WEND
```

The above example conditions ERR to zero and then idles in a WHILE/WEND loop until the user presses **[ESC]** or an input timeout occurs. The conditioning of ERR before entering the WHILE/WEND loop assures the loop will not prematurely terminate due to ERR already being equal to one of the two test values (31 or 32).

spellchk — Check Spelling of Text String

Syntax:

```
CALL "spellchk",TEXT$,NWORD,WORD$[ALL],OCC[ALL]
```

Call Parameters:

TEXT$	Text string to be spell-checked. See text.

Returns:

NWORD	Number of words in TEXT$ found to be misspelled, zero if none.
WORD$[]	Zero-based list of misspelled words, sorted into ascending ASCII order. See text.
OCC[]	Zero-based number of occurrences for each misspelled word returned in WORD$[]. See text.

spellchk breaks up the text in TEXT$ into individual words and then spell-checks the resulting word list. Words flagged as misspelled will be returned in the WORD$[] array. The OCC[] array will indicate how many times each misspelled word occurs in TEXT$, with a one-for-one correspondence to WORD$[]. These arrays are not returned if NWORD=0. TEXT$ is always returned unchanged.

spellchk considers a word to be any non-blank sequence of characters. Normal punctuation is ignored and if associated with a misspelled word, will not be returned in the word list. Most plausible derivations of a word are recognized as valid if the base word is valid. Many proper names and technical terms are also recognized. spellchk is not case-sensitive. Several seconds of processing time may be required to complete a spell check if the text passed in TEXT$ is very long (e.g., 5,000 or more words).

Example:

```
TEXT$="Nnow is thhe time to com to thhe aid of the party.";
CALL "spellchk",TEXT$,NWORD,WORD$[ALL],OCC[ALL];
```

The above example will return the following values:

```
NWORD 3
WORD$[0] "Nnow"
WORD$[1] "com"
WORD$[2] "tthe"
OCC[0] 1
OCC[1] 1
OCC[2] 2
```

The above data indicates that three misspelled words were returned in `WORD$[]`, `Nnow` and `com` were each detected once and `tthe` was detected twice.

statfmt — Generate Data Format File Creation Statistics

Syntax:

```
CALL "statfmt",FORMAT$,KSIZ,RSIZ,NKEY,KFLD[ALL]
```

Call Parameters:

FORMAT$	Data format name for which statistics are to be generated in #LLNNNNNN style.

Returns:

KSIZ	Key size in bytes of combined key elements or zero if no key elements have been defined in the format. See text.
RSIZ	Record size in bytes or zero if all elements are keys. See text.
NKEY	Number of elements defined as keys.
KFLD[ALL]	One-based index of key element numbers if NKEY is non-zero. See text.

statfmt provides the means to gather information needed to create a file whose structure conforms to the attributes of a data format named in FORMAT$. The KSIZ variable reflects the key size required to create a direct or sort file, and will be zero if no key elements have been defined (in which case you would instead create an indexed, serial or text file). The RSIZ variable is computed from the size of the format's data area and will be zero if all elements in the format have been defined as keys (in which case, a sort file would be appropriate). RSIZ includes the space occupied by multiple occurrence fields, as well as multiple occurrence and single occurrence element field separators, if used.[17]

If variable NKEY returns a non-zero value KFLD[] will return a list of element numbers corresponding to elements defined as keys, and in the correct sequence for the format's key structure. Otherwise, KFLD[] will deleted from memory.

Example:

```
FORMAT$="#MYFORMAT";
CALL "statfmt",FORMAT$,KSIZ,RSIZ,NKEY,KFLD[ALL];
CALL "maketemp",KSIZ,RSIZ,TF$,TF
```

[17] Elements designated as keys are not permitted to have multiple occurrences—the format editor in IDOL-IV will enforce this rule. In general, you should avoid the use of field separators in your data formats unless you are maintaining compatibility with an existing file structure.

The above sequence stats the #MYFORMAT data format, and creates and opens a temporary file with key size KSIZ and record size RSIZ in the tmp subdirectory, returning the filename in TF$ and the file's open channel number in TF. To write a new record into this file you could use code similar to the following:

```
K$="";
FOR E=1 TO NKEY;
   K$=K$+FMD(FORMAT$,KFLD[E]);
NEXT I;
WRITE (TF,KEY=K$) #FORMAT$
```

In building the record key K$*, it is essential you use the* FMD *string function to extract data from elements, not* FMT. FMD returns the literal data in the element, whereas FMT formats that data according to the element attributes and default mask. The result obtained with FMT may be markedly different than what is expected and could produce a key value whose size and/or content is incompatible with the file to which the data is to be written.

See also the buildkey function (page 70).

statlink — Get Format & Filename Associated With Link

Syntax:

```
CALL "statlink",LINK$,FILE$,FORMAT$[,TEXTF$]
```

Call Parameters:

LINK$	IDOL-IV link name in LLNNNNNN style.

Returns:

LINK$	Link title. See text.
FILE$	Filename associated with link or null if not defined or link not found.
FORMAT$	Data format associated with link in #LLNNNNNN style or null if not defined or link not found.
TEXTF$	Text file associated with link or null if not defined or link not found. Optional parameter.

statlink provides a portable way to get information needed to access a database using only an IDOL-IV link name. statlink returns the data format name and filename associated with the link specified in LINK$. Null values are returned if the link name is undefined or the link contains no format or filename references. Upon a successful return from this call LINK$ will contained the title that was assigned to the link when it was created. This title can be used for file maintenance screen titles, etc. LINK$ is not changed if the call is unsuccessful.

Example:

```
LINK$="GCCPMAS";
CALL"statlink",LINK$,FILE$,FORMAT$,TEXTF$
```

This example will return gccpmas.dbf in FILE$, #GCCPMAS in FORMAT$ and Corporate Profile Master in LINK$. TEXTF$ will be null because no text file was defined for this link.

tempdir — Get Temporary Directory Number

Syntax:

```
CALL "tempdir",D[,D$]
```

Call Parameters:

None.

Returns:

D	Logical disk number of the directory defined by the UNIX environment variable TMPDIR. If TMPDIR has not been defined D will return zero.
D$	If passed in the call syntax, returns the full qualified name (e.g., /abc/def/tmp) of the directory defined by the UNIX environment variable TMPDIR (there is no trailing slash in the name). If TMPDIR has not been defined this variable will be null.

tempdir matches the directory defined in the UNIX TMPDIR environment variable to its directory number assignment in the *Thoroughbred* IPL file that started the task. This function also creates a global variable named tmpdir (note the lower case spelling—global variable names are case-sensitive) with the directory number. The temporary directory should be used for files that a program creates and destroys during runtime. Refer to the tempfid (page 242) and nextfid (page 178) functions for the generation of temporary filenames keyed to a particular task.

tempdirn — Get Disk Name of Temporary Directory

Syntax:

```
CALL "tempdirn",D$
```

Call Parameters:

None.

Returns:

D$	Disk name of the directory defined by the UNIX environment variable TMPDIR, D0 to DZ, or null if TMPDIR has not been defined.

tempdirn matches the directory defined in the UNIX TMPDIR environment variable to its directory number assignment in the *Thoroughbred* IPL file that started the task, returning the resulting logical disk name, such as D9. See also tempdir (page 240).

tempfid — Generate Temporary Filename

Syntax:

```
CALL "tempfid",F$
```

Call Parameters:

None.

Returns:

F$	Temporary filename keyed to calling task. See text.

tempfid uses information from the *Thoroughbred* CDN system variable to generate a temporary filename suitable for both UNIX/Linux and MS-DOS/Windows environments. The resulting filename takes the form XXXXXXXX.TTT in which the XXXXXXXX portion is a series of alphanumeric characters derived from CDN and TTT is the decimal equivalent of the task's numeric identification derived from the idport function (page 132). A typical filename generated by this function for port TG would be 00EA1A49.016.

tempfid generates part of the filename from the CDN variable—which doesn't update more than a few times per second. Therefore, it is possible for the same filename to be generated if several calls are made in rapid succession. To compensate for this quirk make only one call to tempfid and then use the nextfid function (page 178) to generate a sequence of names if more than one is needed. The following example illustrates this process.

Example:

```
CALL "tempfid",F1$;          REM Get a unique filename.
F2$=F1$;                     REM Copy that filename.
CALL "nextfid",F2$;          REM Next filename in sequence.
F3$=F2$;                     REM Copy that filename.
CALL "nextfid",F3$;          REM Next filename in sequence.
CALL "tempdir",TMPDIR;       REM Get tmp directory number.
SORT F1$,KSIZ1,0,TMPDIR,0; REM 1st temporary sort file.
SORT F2$,KSIZ2,0,TMPDIR,0; REM 2nd temporary sort file.
SORT F3$,KSIZ3,0,TMPDIR,0; REM 3rd temporary sort file.
…process as required…
```

TERMINAL MNEMONICS

The following special terminal control mnemonics have been defined in the W60ENH terminal driver table in TCONFIGW, which is supplied with most systems built by **BCS TECHNOLOGY LIMITED**. They take advantage of the technical capabilities of the WYSE 60 terminal (as well as derivatives such as the WY150, WY160E and WY-GPT). A similar table named W60DCM has been defined to work with Futuresoft's *Dynacomm* emulation program for personal computers. Refer to the *Thoroughbred* technical manual for details on the standard mnemonics supplied with TCONFIGW.

B0 **Select default font bank**
B1 **Select alternate font bank**

These mnemonics facilitate switching between the default or native mode font bank (character set) and multinational font bank, which contains special graphics characters not available in the default bank. See also the FB mnemonic for instructions on how to change the multinational font bank character set. Font bank switching is not supported in *Dynacomm*.

BA **Enable hardware autoscroll**
EA **Disable hardware autoscroll**

When autoscroll is enabled the screen will scroll up when the cursor is moved "below" the bottom display row (usually physical row 24). When autoscroll is disabled the cursor will "wrap" to the top row rather than scrolling the screen.

DA **Down arrow**
LA **Left arrow**
RA **Right arrow**
UA **Up arrow**

When the alternate font bank (selected with B1) has been loaded with the "graphics 1" character set, these mnemonics will print the characters ▾, ◂, ▸ and ▴ respectively.

EK **Character emitted by [ESC]**

This informational mnemonic is used by various cookbook functions to detect the pressing of the key designated as **[ESC]**. To read this character use MNE("EK").

EP **Begin expanded print**
NP **End expanded print**

EP causes the text on the current row to be printed at double width. Printing NP on the same row will return the text to normal width.

Note that `EP` is transparent to *Thoroughbred* and is also recognized by the `cprint` and `lmargin` functions.

`F0` **Function key load preamble**
`F1` **Function key load indexing base character**
`F2` **Function key load postamble**
`NK` **Number of programmable function keys**

These mnemonics are used by the `loadfkey` function to select a function key and download text. They should not be used in regular programs.

`FB` **Load alternate font bank with character set**

This mnemonic is used to copy one of the resident character sets to the alternate font bank (see `B1`). The syntax to be used is `PRINT 'FB',"<SET>"`, where `<SET>` is one of the following characters:

`@`	Native
`A`	Multinational
`B`	Standard ASCII
`C`	Graphics 1
`D`	PC equivalent
`E`	Graphics 2
`F`	Graphics 3
`G`	Standard ANSI

Refer to pages E-36 through E-38 in the *WY-60 User's Guide* for a listing of each of the character sets. This feature is not supported by *Dynacomm*.

`R0` **Select flashing block cursor**
`R1` **Select flashing underline cursor**
`R2` **Select static block cursor**
`R3` **Select static underline cursor**

The above mnemonics can be used to change the appearance and behavior of the terminal's hardware cursor. For example, `PRINT 'R1'`, will cause the terminal to display a blinking underscore cursor.

If you do not wish for the cursor to be visible immediately following the selection of a new cursor type turn it off with the `CO` mnemonic.

`SI` **Turn on [CAPSLOCKS]**
`SO` **Turn off [CAPSLOCKS]**

These mnemonics ("shift in" and "shift out") may be used to enable or disable the uppercase lock feature of the terminal. Following `PRINT 'SI'`, the word `CAPS` will appear on the terminal status line (if enabled).

`textbox` Display Text Box

Syntax:

```
CALL "textbox",L$[ALL],ROW,WIDTH,FG,SC
```

Call Parameters:

`L$[ALL]`		Text lines to be displayed, one-based.
`ROW`		Top row of box coordinates, zero-based.
`WIDTH`	`0`	Standard width.
	`1`	Double width, if supported by terminal.
`FG`	`0`	Display in foreground.
	`1`	Display in background.
`SC`	`0`	Display with lower right shadow.
	`1`	Display with halo.
	`2`	No shadow or halo.

Returns:

None.

`textbox` can be used to produce screen banners and other display enhancements. It is the basis of the previously described `msgbox` function (`msgbox` calls `textbox`). Three possible box styles can be produced in either foreground or background intensity: shadowed, haloed or plain. If background intensity is selected the shadow or halo will be subdued as well. Haloing is not possible if `ROW` is zero or the width of the box extends the full width of the screen. The box contents are always displayed in reverse video.

The text boxes produced by this function are not windows. If you need windowed text boxes you should use the `msgbox` function (page 176).

textflde Edit 4GL Text Field

Syntax:

```
CALL "textflde",LINK$,ELM
```

Call Parameters:

LINK$	IDOL-IV link name containing text field to be edited in LLNNNNNN format.
ELM	Number of element containing text field. See text.

Returns:

ERR	Any execution error. Error numbers returned from the IDOL-IV APIs will not be regular *Thoroughbred* errors. See text.

Preparatory Operations:

The elements defined as record keys in the format associated with link LINK$ must contain valid key data for the file associated with link LINK$. See text.

textflde is a convenient interface into the IDOL-IV text editing subsystem. textflde performs operations necessary to editing a text field as defined within the format associated with link LINK$. When called, textflde will verify that a text file was defined for LINK$, open a prompt window and then call the IDOL-IV 8TEXTF API to permit the user to enter and edit text, using the standard IDOL-IV full screen editor. Upon exiting from the editor the text will be written into the text file, the prompt window will be closed and control will be returned to the calling program. Any error that occurs within textflde or 8TEXTF will be returned to the calling program. Errors generated within the 8TEXTF API will not be standard *Thoroughbred* error values. Refer to the API application notes for the meanings of any such errors.

In order to properly utilize textflde it is necessary for the format associated with link LINK$ to contain at least one element defined as a text field. Since IDOL-IV permits a format to contain many text fields, it is necessary to tell textflde which text field to edit by specifying the element number in ELM. textflde will abort if the specified element has not been defined as a text field. Refer to the help screens within the IDOL-IV format editing function for more information on defining text fields.

The 8TEXTF API loads and saves text by constructing a text field key from the key elements in the record format that is associated with link LINK$ and joining that key to the text field ID character in the valid values attribute of the text field element.

The result should be a unique key within the text file associated with the link. Key elements must be loaded with valid key data and the resulting key must match a record in the data file associated with link `LINK$`. The best way to assure that this requirement is met is to first extract the appropriate record into the format and then call `textflde` to edit the desired text field.

See also `textfldl` (page 249), `textfldv` (page 251) and `textfldw` (page 253).

textfldl Load 4GL Text Field

Syntax:

```
CALL "textfldl",LINK$,ELM,TEXT$[ALL],LINES
```

Call Parameters:

LINK$	IDOL-IV link name containing text to be loaded in LLNNNNNN format.
ELM	Text field element number. See text.

Returns:

TEXT$[ALL]	Retrieved text lines, one-based. See text.	
LINES	Number of text lines returned. This value is also returned in TEXT$[0].	
ERR	0	OK, TEXT$[] is valid.
	1	Invalid link name.
	2	Invalid format name.
	3	Invalid element number or element is not defined as a text field.

The above ERR values will not cause an execution error in the calling program.

Preparatory Operations:

The elements defined as record keys in the format associated with link LINK$ must contain valid key data for the file associated with link LINK$. See text.

textfldl performs operations necessary to retrieve text from a text file associated with link LINK$. When called, textfldl will build a key from the data in the format associated with the line and retrieve text into TEXT$[], with each line equivalent to the original created in the IDOL-IV full screen text editor (as with the textfldе function). Each text line within TEXT$[] will be padded with spaces so that all lines are equal in length to the width of the editing window that was used to create the original text.

In order to properly utilize textfldl it is necessary for the format associated with link LINK$ to contain at least one element defined as a text field. Since IDOL-IV permits a format to contain many text fields, it is necessary to tell textfldl which text field to load by specifying the element number in ELM. textfldl will abort if the specified element has not been defined as a text field. Refer to the help screens within the IDOL-IV format editing function for more information on defining text fields.

IDOL-IV retrieves text by constructing a text field key from the key elements in the record format that is associated with link `LINK$` and joining that key to the text field ID character in the valid values attribute of the text field element. The result should be a unique key within the text file associated with the link. Key elements must be loaded with valid key data and the resulting key must match a record in the data file associated with link `LINK$`. The best way to assure that this requirement is met is to first read or extract the appropriate record into the format and then call `textfldl` to load the desired text.

See also `textflde` (page 247), `textfldv` (page 251) and `textfldw` (page 253).

textfldv View 4GL Text Field

Syntax:

```
CALL "textfldv",LINK$,ELM,TITLE$,TROW,PROW,TIMOUT,OP
```

Call Parameters:

LINK$	IDOL-IV link name containing text to be loaded in LLNNNNNN format.
ELM	Text field element number. See text.
TITLE$	Text window title, displayed on top window border.
TROW	Row for top border of window. If zero this value will be computed from the PROW (prompt row) value. See text.
PROW	Row on which to display prompts. If zero, the physical screen height -1 will be assumed. See text.
TIMOUT	User response time out period in seconds, zero for no time out.
OP	Operation mode: 0 Display text if available. 1 Silently check for text availability.

Returns:

OP	0 Text was displayed or is available. 1 Text not found.
ERR	Any execution error, including IDOL-IV API errors.

Preparatory Operations:

The elements defined as record keys in the format associated with link LINK$ must contain valid key data for the file associated with link LINK$. See text.

textfldv performs operations necessary to retrieve and display text from a text file associated with link LINK$. When called, textfldv will build a record key built from the data in the format associated with the line, retrieve any associated text and open a window in which to display it, with the window width determined by the width of the editing window that was used to create the original text. Unless the PROW and TROW parameters are set to something other than zero, textfldv will position both the text and prompt windows according to the physical screen size and the default window height of 12 text lines. You can adjust the display if necessary by specifying alternative PROW and TROW values.

In order to properly utilize `textfldv` it is necessary for the format associated with link `LINK$` to contain at least one element defined as a text field and for text to be available to display. Refer to the `textfldl` function (page 249) for more information on what is required.

Once the text window has been opened the user will see a "scroll bar" in the right hand window frame if the amount of text to be displayed is greater than what will fit in the window. Pressing cursor down will scroll the text about one half the window, pressing cursor up will likewise scroll up one half the window. Pressing page down or its equivalent will scroll the text one full window, page up will do so in reverse. Press the home key will go to the end of the text if at the beginning, the top of the window if at the end of the text, followed by the beginning of the text. As the view position in the text changes so will the appearance of the scroll bar. Pressing **[ESC]** will close the window and exit, as will timing out.

See also `textfldе` (page 247).

textfldw — Write 4GL Text Field

Syntax:

```
CALL "textfldw",LINK$,ELM,TEXT$[ALL]
```

Call Parameters:

LINK$	IDOL-IV link name containing text to be loaded in LLNNNNNN format.
ELM	Text field element number. See text.
TEXT$[ALL]	Text lines to be written, one-based. See text.

Returns:

ERR	0	OK, TEXT$[] was written.
	1	Invalid link name.
	2	Invalid format name.
	3	Invalid element number or element is not defined as a text field.
	4	Memory capacity error, either because there are too many elements in the TEXT$[] array or because the total amount of text to be written exceeds available task resources.

The above ERR values will not cause an execution error in the calling program.

Preparatory Operations:

The elements defined as record keys in the format associated with link LINK$ must contain valid key data for the file associated with link LINK$ and the TEXT$[] array must be loaded as required. Note that TEXT$[0] is not used. See text.

textfldw performs operations necessary to store text into a text file associated with link LINK$. When called, textfldw will build a key from the data in the format associated with the line and write the text in TEXT$[] into the associated text file. Lines within TEXT$[] that exceed the characters per line specification defined in the valid values attribute of the data format associated with LINK$ will be truncated.

In order to properly utilize textfldw it is necessary for the format associated with link LINK$ to contain at least one element defined as a text field. Since IDOL-IV permits a format to contain many text fields, it is necessary to tell textfldw which text field to write by specifying the element number in ELM.

`textfldw` will abort if the specified element has not been defined as a text field. Refer to the help screens within the IDOL-IV format editing function for more information on defining text fields.

IDOL-IV determines how to index the text data by constructing a text field key from the key elements in the record format that is associated with link `LINK$` and joining that key to the text field ID character in the valid values attribute of the text field element. The result should be a unique key within the text file associated with the link. Key elements must be loaded with valid key data and the resulting key must match a record in the data file associated with link `LINK$`. The best way to assure that this requirement is met is to first extract the appropriate record into the format and then call `textfldw` to write the desired text.

See also `textfldе` (page 247), `textfldl` (page 249) and `textfldv` (page 251).

textsubs Text Substitution and Expansion Macro Processor

Syntax:

```
CALL "textsubs",TEXT$
```

Call Parameters:

TEXT$	Unprocessed text string.

Returns:

TEXT$	Processed text string.

textsubs is a general purpose text substitution and expansion macro processor that can ease the chore of displaying various kinds of character data on screen. It is particularly useful in processing screen text read from a file. The basis for the function's operation is the macro escape sequence, a string of two characters preceded with a backslash (\). When textsubs encounters the backslash in the text stream it will attempt to replace the next two characters with some kind of graphic character or alternate text.

textsubs facilitates two distinct operations:

- **Word substitution**. textsubs can replace certain macros with words associated with the system. For example, a company's name can be inserted into a text stream. It is also possible to use macros to insert the date or time of day.

- **Character translation**. textsubs can replace macros with graphic characters normally accessed through terminal mnemonics such as G0 or GF, or generate vertical and horizontal graphic lines from a single macro escape sequence. The character translation feature of textsubs makes it possible to read text out of a file and insert graphic characters where needed without the hassle of trying to convert the textual representation of a mnemonic into the internal representation understood by *Thoroughbred*.

Macros fall into two groups: those that stand alone and those that can include a numeric repeat value. Standalone macros may be part of a continuous text stream, whereas macros that use a repeat value must be the only thing in a text stream. The general form of the input string in TEXT$ is either:

```
Word word \ab word word \cd word word...
```

where word is ordinary text and **\ab** and **\cd** are standalone macros or:

```
\ab[n]
```

where is `\ab` is a repeating macro and `[n]` is the optional repeat value. Any text with embedded macros will change length during processing, something which must be considered while developing a screen layout.

The following is a list of the standalone macros understood by `textsubs`.

Macro	Substitution
`\bl`	└ (bottom left corner)
`\br`	┘ (bottom right corner)
`\bt`	┴ (bottom connect tee)
`\ce`	`'CE'` (clear to screen end)
`\cl`	`'CL'` (clear to line end)
`\cn`	Full corporate name derived from previously loaded corporate profile (see `loadprof` on page 159)
`\cr`	┼ (cross)
`\dt`	Today's date derived from the `CDN` system variable in `MM/DD/YY` format. A leading zero in the month is replaced with a blank.
`\fn`	Corporate name derived from previously loaded corporate profile (see `loadprof` on page 159)
`\hb`	▉ (high intensity block)
`\lb`	▒ (low intensity block)
`\lt`	┤ (left connect tee)
`\mb`	▒ (medium intensity block)
`\nn`	Corporate nickname derived from previously loaded corporate profile (see `loadprof` on page 159)
`\rt`	├ (right connect tee)
`\sa`	System name derived from previously loaded corporate profile (see `loadprof` on page 159)
`\sn`	Corporate short name derived from previously loaded corporate profile (see `loadprof` on page 159)
`\td`	Time of day derived from the `CDN` system variable in `HH:MI {A|P}M` format. A leading zero in the hour is replaced with a blank.
`\tl`	┌ (top left corner)
`\tr`	┐ (top right corner)
`\tt`	┬ (top connect tee)

Example:

```
TEXT$="This is the \fn.";
CALL "textsubs",TEXT$;
PRINT TEXT$
```

The above example prints `This is the <formal system name>.`, where `<formal system name>` is the system name defined in the corporate profile.

The following macros must be the only text in the text supplied to `textsubs` and may be paired with an optional numeric parameter representing a repeat value. If no repeat value is provided horizontal characters will be repeated by an amount equal to the current window width and vertical characters by an amount equal to the current window height. Any repeat value between 1 and 999 is acceptable.

Macro	Substitution
\hd	= (horizontal double line)
\hs	– (horizontal single line)
\vd	‖ (vertical double line)
\vs	| (vertical single line)

Examples:

```
TEXT$="\hd20";
CALL "textsubs",TEXT$;
PRINT TEXT$
```

The above example prints ========== (20 horizontal double line characters).

```
TEXT$="\hs";
CALL "textsubs",TEXT$;
PRINT TEXT$
```

The above example prints

(80 horizontal single line characters) on a standard, full width window. Note that omitting the repeat value causes the current window width to be used as the repeat value.

`textview` Display Text Array

Syntax:

```
CALL "textview",TITLE$,TROW,PROW,TIMOUT,TEXT$[ALL]
```

Call Parameters:

`TITLE$`	Text window title, displayed on top window border. See text.
`TROW`	Row for top border of window. If zero this value will be computed from the `PROW` (prompt row) value. See text.
`PROW`	Row on which to display prompts. If zero, the physical screen height -1 will be assumed. See text.
`TIMOUT`	User response time out period in seconds, zero for no time out.
`TEXT$[]`	One-based array of text lines, with a maximum line length of `<width>-2` characters, where `<width>` is the physical screen width in columns. See text.

Returns:

`ERR`	`0`	Text window closed with **[ESC]**.
	`1`	Timed out.
	`2`	Text array improperly defined or empty.

The setting of `ERR` *to the above values will not cause an execution error in the calling program.*

`textview` displays the text passed in the `TEXT$[]` array in a pop-up window, with the window size determined by the length of the longest printable text line (the window will go off the right hand screen border if the longest line is too long). Unless the `PROW` and `TROW` parameters are set to something other than zero, `textview` will position both the text and prompt windows according to the physical screen size and a default window height of 12 text lines. You can adjust the display if necessary by specifying alternative `PROW` and `TROW` values.

If a title has been defined in `TITLE$` it will be centered on the top border of the window, framed in angle brackets (<>) and displayed in reverse video. For example, if `TITLE$="THE TITLE"` it will be displayed as `<THE TITLE>` . *Do not embed mnemonics in the title.*

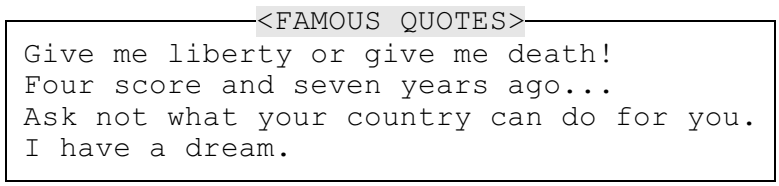
Once the text window has been opened the user will see a "scroll bar" in the right hand window frame if the amount of text to be displayed is greater than what will fit in the window.

Pressing cursor down will scroll the text about one half the window, pressing cursor up will likewise scroll up one half the window. Pressing page down or its equivalent will scroll the text one full window, page up will do so in reverse. Pressing the home key will go to the end of the text if at the beginning, the top of the window if at the end of the text, followed by the beginning of the text. As the view position in the text changes so will the appearance of the scroll bar. Pressing [ESC] will close the window and exit, as will timing out.

Example:

```
DIM TEXT$[4];
TEXT$[1]="Give me liberty or give me death!",
TEXT$[2]="Four score and seven years ago...",
TEXT$[3]="Ask not what your country can do for you.",
TEXT$[4]="I have a dream.";
CALL "textview","FAMOUS QUOTES",0,0,120,TEXT$[ALL];
ON ERR GOTO CLOSED,TIMED_OUT,NO_TEXT
```

The above example will produce the following display:



winname — Generate Random Window Name

Syntax:

```
CALL "winname",WNAME$
```

Call Parameters:

None.

Returns:

WNAME$	Randomly generated window name, 8 characters in length. WNAME$ is guaranteed to be unique in the task environment when this function is correctly implemented. See text.

winname generates a random, eight character string suitable for naming a window that is to be created. To assure uniqueness, winname verifies the name it has generated against the task's windows stack and, if necessary, alters the name until it is unique. Consecutive calls should not be made to winname. That is to say, a call to winname should not be followed by another winname call until a WINDOW CREATE directive to use the previously generated window name has been executed. Otherwise, the second (and possibly third) call may return a duplicate window name.

Examples:

```
CALL "winname",WNAME1$;
WINDOW CREATE (40,10,5,3)"NAME="+WNAME1$ ! create 1st window
CALL "winname",WNAME2$
WINDOW CREATE (20,5,15,6)"NAME="+WNAME2$ ! create 2nd window
```

The above example illustrates the proper way to utilize winname when multiple window names are required. Always assign the first window name to a window before requesting another name.

```
CALL "winname",WNAME1$;
CALL "winname",WNAME2$
WINDOW CREATE (40,10,5,3)"NAME="+WNAME1$
WINDOW CREATE (20,5,15,6)"NAME="+WNAME2$
```

The above example illustrates the wrong way to utilize winname when multiple window names are required. In all likelihood, WNAME1$ and WNAME2$ will be identical, causing an error when the second WINDOW CREATE statement is executed.

workdir — Get Permanent Data File Directory Number

Syntax:

```
CALL "workdir",D[,D$]
```

Call Parameters:

None.

Returns:

D	Logical disk number of the directory defined by the UNIX environment variable WRKDIR. If WRKDIR has not been defined this function will return zero.
D$	If passed in the call syntax, logical disk name (e.g., D4) of the directory defined by the UNIX environment variable WRKDIR. If WRKDIR has not been defined this variable will return D0.

workdir matches the directory defined by the UNIX WRKDIR environment variable to its directory number assignment in the *Thoroughbred* IPLINPUT file that started the task. This function also creates a global variable named wrkdir (note the lower case spelling—global variable names are case-sensitive) with the directory number. The work directory should be used for permanent data files only.

yesno — Get User's Yes/No Response

Syntax:

```
CALL "yesno",ROW,COL,TIMOUT,FLAG
```

Call Parameters:

ROW/COL	Row/column coordinates for input. If both are zero the current cursor position is assumed.	
TIMOUT	No response timeout in seconds, zero for no timeout.	
FLAG	0	No default response.
	1	Default response = **[N]**o.
	2	Default response = **[Y]**es.

Returns:

ROW/COL	Row/column coordinates for input if these values were zero on call.	
FLAG	0	[N]o or [F2] detected.
	1	[Y]es or [F1] detected.
	2	[F4] or [ESC] detected.
	3	Timed out.

The value of FLAG is also returned in the ERR system variable. *This will not cause an execution error in the calling program.*

yesno accepts user Yes/No responses and converts them into a numeric progression. The option of pressing [F1], [F2] or [F4] is made available for users and programmers accustomed to the old *MAI Basic Four* convention of relating those keys to Yes, No or abort. If the user elects to type a Y or N as a response s/he must press ⏮ to initiate action. Pressing [F1] or [F2] will cause an immediate reaction—no ⏮ keypress is needed—and the echoing of Y or N in the input field.

Example:

```
PRINT @(4,6),"Run This Program (Y/N/ESC)? ",;
FLAG=1;
CALL "yesno",C,R,600,FLAG;
ON ERR(1,2,3) GOTO ABORT,RUNPROG,ABORT,TIMED_OUT
```

zipbin — Encode ZIP/Postal Code

Syntax:

```
CALL "zipbin",ZIP$,ZIPC$
```

Call Parameters:

ZIP$	U.S. ZIP code in NNNNN[[-]NNNN] format or Canadian postal code in ANANAN format. See text.

Returns:

ZIPC$		Compressed four byte (32 bit) binary representation of ZIP$. See text.
ERR	0	OK, ZIPC$ is valid.
	1	Invalid ZIP or Canadian postal code format in ZIP$.

zipbin converts the ASCII form of a U.S. ZIP code or Canadian postal code as passed in ZIP$ into a 32 bit binary value that is guaranteed to sort in ascending order. The encoded format is determined by the content of ZIP$. If ZIP$ contains a valid Canadian postal code, bit 31 of the value returned in ZIPC$ will be set. You can test for the conversion type with the expression SGN(DEC(AND(ZIPC$,$80000000$))). The result will be -1 if ZIPC$ contains a Canadian postal code or 0 if ZIPC$ contains a U.S. ZIP code. In a sorted list containing both ZIP codes and Canadian postal codes, the ZIP codes will have lower precedence in the collating sequence. That is to say, the ZIP code 99999-9999 will come before the Canadian postal code A0A0A0.

A ZIP code may be either a five or nine digit value, with an optional hyphen (-) between the fifth and sixth digits. Any value in the range 00000 to 99999-9999 inclusive may be encoded. For example, valid formats include 60421, 60421-6044 or 604216044. 60421-0000 or 604210000 is functionally identical to 60421. A Canadian postal code must be in the format ANANAN or ANA NAN, where A represents an alphabetic character and N represents a numeral. Any value in the range A0A0A0 to Z9Z9Z9 inclusive may be encoded. An optional blank is allowed between the third and fourth characters. For example, the postal code L4U3F1 could also be in the form L4U 3F1, l4u3f1 or l4u 3f1, as the conversion is not case-sensitive.

Examples:

```
ZIP$="60421-6044";
CALL "zipbin",ZIP$,ZIPC$;
ON ERR GOTO OK,INVALID
```

The above sequence will return $24039AEC$ in ZIPC$.

```
ZIP$="L4U 3F1";
CALL "zipbin",ZIP$,ZIPC$;
ON ERR GOTO OK,INVALID
```

The above sequence will return $8312A661$ in ZIPC$.

```
ZIP$="";
CALL "zipbin",ZIP$,ZIPC$;
ON ERR GOTO OK,INVALID
```

The above sequence will return 00000000 in ZIPC$. No error will occur if ZIP$ is null.

See also FNZIP$ (page 54), FNZIPC$ (page 54) and binzip (page 69).

INDEX